Learning Moore Machines from Input-Output Traces

Georgios Giantamidis\textsuperscript{1} and Stavros Tripakis\textsuperscript{1,2}

\textsuperscript{1} Aalto University
\textsuperscript{2} University of California, Berkeley

Abstract. The problem of learning automata from example traces (but no equivalence or membership queries) is fundamental in automata learning theory and practice. In this paper we study this problem for finite state machines with inputs and outputs, and in particular for Moore machines. We introduce three algorithms for solving this problem: (1) the PTAP algorithm, which transforms a set of input-output traces into an incomplete Moore machine and then completes that machine with self-loops; (2) the PRPNI algorithm, which uses the well-known RPNI algorithm for automata learning to learn a product of automata encoding a Moore machine; and (3) the MooreMI algorithm, which directly learns a Moore machine using PTAP extended with state merging. We prove that MooreMI always learns the right machine when the training set is a characteristic sample, which is generally not true for the other two algorithms. We also compare the algorithms experimentally in terms of the size of the learned machine and several notions of accuracy, introduced in this paper. Finally, we compare with OSTIA, an algorithm that learns a more general class of transducers, and find that OSTIA generally does not learn a Moore machine, even when fed with a characteristic sample.

1 Introduction

An abundance of data from the internet and from other sources (e.g., sensors) is revolutionizing many sectors of science, technology, and ultimately our society. At the heart of this revolution lies machine learning, a broad spectrum of techniques to derive information from data. Traditionally, objects studied by machine learning include classifiers, decision trees, and neural networks, with applications to fields as diverse as artificial intelligence, marketing, finance, or medicine [19].

In the context of system design, an important problem, with numerous applications, is automatically generating models from data. There are many variants of this problem, depending on what types of models and data are considered, as well as other assumptions or restrictions. Examples include, but are by no means limited to, the classic field of system identification [18], as well as more recent works on synthesizing programs, controllers, or other artifacts from examples [25,11,23,22,3].

In this paper we consider a basic problem, that of learning a Moore machine from a set of input-output traces. A Moore machine is a type of finite-state machine (FSM) with inputs and outputs, where the output always depends on the current state, but not on the current input [16]. Moore machines are typically deterministic and complete, meaning that for given state and input, the next state is always defined and is unique; and for given state, the output is also always uniquely defined. Such machines are useful in many applications, for instance, for representing digital circuits or controllers. In this paper we are interested in learning deterministic and complete Moore machines.

We want to learn a Moore machine from a given set of input-output traces. One such trace is a sequence of inputs, $\rho_{in}$, and the corresponding sequence of outputs, $\rho_{out}$, that the machine must produce when fed with $\rho_{in}$. As in standard machine learning methods, we call the set of traces given to the learning algorithm the \textit{training set}. Obviously, we would like the learned machine $M$ to be \textit{consistent} w.r.t. the training set $R$, meaning that for every pair $(\rho_{in}, \rho_{out}) \in R$, $M$ must output $\rho_{out}$ when fed with $\rho_{in}$. But in addition to consistency, we would like $M$ to behave well w.r.t. several \textit{performance} criteria, including complexity of the learning algorithm, size of the learned machine $M$ (its number of states), and \textit{accuracy} of $M$, which captures how well $M$ performs on a \textit{testing} set of traces, different from the training set.

\textsuperscript{*} This work was partially supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).
Even though this is a basic problem, it appears not to have received much attention in the literature. In fact, to the best of our knowledge, this is the first paper which tackles precisely this problem. This is despite a large body of research on grammatical inference [7] which has studied similar, but not exactly the same problems, such as learning deterministic finite automata (DFA), which are special cases of Moore machines with a binary output, or subsequential transducers, which are more general than Moore machines.

Our contributions are the following:

1. We define formally the LMoMIO problem (learning Moore machines from input-output traces) and introduce several notions of accuracy. We also adapt the notion of characteristic sample, which is known for DFA [7], to the case of Moore machines. Intuitively, a characteristic sample of a machine $M$ is a set of traces which contains enough information to “reconstruct” $M$. The characteristic sample requirement (CSR) states that, when given as input a characteristic sample, the learning algorithm must produce a machine equivalent to the one that produced the sample. CSR is important, as it ensures identification in the limit: this is a key concept in automata learning theory which ensures that the learning algorithm will eventually learn the right machine when provided with a sufficiently large set of examples [9].

2. We present three algorithms to solve the LMoMIO problem, and analyze them in terms of computational complexity and other properties. We show that although all three algorithms guarantee consistency of the learned machine with the training set, only the most advanced among them, called MooreMI, satisfies the characteristic sample requirement.

3. We report on a prototype implementation of all three algorithms and experimental results. The experiments show that MooreMI outperforms the other two algorithms not only in theory, but also in practice.

4. We show that the well-known transducer-learning algorithm OSTIA [21] cannot generally learn a Moore machine, even in the case where the training set is a characteristic sample of a Moore machine. This further justifies the study of specialized learning algorithms for Moore machines.

2 Related Work

There is a large body of research on learning automata and state machines, which can be divided into two broad categories: learning with queries (or active learning), and learning from examples. A seminal work in the first category is Angluin’s work on learning DFAs with membership and equivalence queries [4]. This work has been subsequently extended to other types of machines, such as (extended) Mealy machines or register automata [24,14,13,6]. These works are not directly applicable to the problem studied in this paper, since we explicitly forbid both membership and equivalence queries.

A seminal work in the second category is Gold’s algorithm for learning DFAs from sets of positive and negative examples [10]. Subsequent work includes the RPNI [20] algorithm (that learns DFAs), of which an incremental version also exists [8], and derivatives, like the EDSM [17] (that also learns DFAs, but unlike RPNI does not guarantee identification in the limit) and OSTIA [21] (that learns subsequential transducers) algorithms. All the RPNI-based algorithms operate by first building some sort of prefix tree FSM and then attempting to minimize it via state merging. This is the case also for the MooreMI algorithm that we present in this paper.

Apart from state merging algorithms like the above, learning from examples has also been attempted with techniques from artificial intelligence, such as genetic algorithms [2] and ant colony optimization [5], as well as with reduction to satisfiability [12,27].

[26] is close to our work, but the algorithm described there does not always yield a deterministic Moore machine, while ours does. Related to ours is also the work in [15], which deals with learning finite state machine abstractions of non-linear analog circuits. The algorithm described in [15] is very different from ours, however, and uses the circuit’s number of inputs to determine a subset of the states in the learned abstraction. Also, identification in the limit is not considered in [15].
Fig. 1: Examples of finite state machines.

3 Preliminaries

3.1 Finite state machines and automata

A finite state machine (FSM) is a tuple $M$ of the form $M = (I, O, Q, q_0, \delta, \lambda)$, where:

- $I$ is a finite set of input symbols.
- $O$ is a finite set of output symbols.
- $Q$ is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times I \rightarrow Q$ is the transition function.
- $\lambda : Q \rightarrow O$, in which case the FSM is a Moore machine.
- $\lambda : Q \times I \rightarrow O$, in which case the FSM is a Mealy machine.

If both $\delta$ and $\lambda$ are total functions, we say that the FSM is complete. If any of $\delta$ and $\lambda$ is a partial function, we say that the FSM is incomplete. Examples of a Moore and a Mealy machine are given in Figure 1. Both FSMS are complete.

We also define $\delta^* : Q \times I^* \rightarrow Q$ and $\lambda^* : Q \times I^* \rightarrow O^*$ as follows ($X^*$ denotes the set of all finite sequences over some set $X$; $\epsilon \in X^*$ denotes the empty sequence over $X$; $w \cdot w'$ denotes the concatenation of two sequences $w, w' \in X^*$): For $q \in Q$, $w \in I^*$, and $a \in I$:

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, w \cdot a) = \delta(\delta^*(q, w), a)$

The rest of this paper focuses on Moore machines, thus we define $\lambda^*$ only in the case where $M$ is a Moore machine (the adaptation to a Mealy machine is straightforward):

- $\lambda^*(q, \epsilon) = \lambda(q)$
- $\lambda^*(q, w \cdot a) = \lambda^*(q, w) \cdot \lambda(\delta^*(q, w \cdot a))$

Two Moore machines $M_1 = (I_1, O_1, Q_1, q_{0,1}, \delta_1, \lambda_1)$ and $M_2 = (I_2, O_2, Q_2, q_{0,2}, \delta_2, \lambda_2)$, are said to be equivalent if $I_1 = I_2$, $O_1 = O_2$, and $\forall w \in I_1^* : \lambda_1^*(q_{0,1}, w) = \lambda_2^*(q_{0,2}, w)$.

A Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$ is minimal if for any other Moore machine $M' = (I', O', Q', q'_0, \delta', \lambda')$ such that $M$ and $M'$ are equivalent, we have $|Q| \leq |Q'|$, where $|X|$ denotes the size of a set $X$.

Notice that in the case two Moore machines are minimal, testing equivalence is reduced to a graph isomorphism test.

A deterministic finite automaton (DFA) is a tuple $A = (\Sigma, Q, q_0, \delta, F)$, where:

- $\Sigma$ (the alphabet) is a finite set of letters.
- $Q$ is a finite set of states.
- $q_0 \in S$ is the initial state.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
complete states, respectively. The concepts of set of output symbols is binary, say \( O \).

Given sets of input and output symbols \( I, O \), we define a **Moore** machine \( M = (I, O, Q, q_0, \delta, \lambda) \), where \( Q \) is the set of accepting states, \( q_0 \) is the initial state, \( \delta \) is the transition relation, and \( \lambda \) is the output function. Viewing a DFA as a Moore machine with binary output, a positive example \( w, 1 \) corresponds to the Moore example \( (w, 1) \), while a negative example corresponds to the Moore example \( (w, 0) \).

### 3.2 Input-output traces and examples

Given sets of input and output symbols \( I, O \), respectively, a **Moore** \( (I, O) \)-trace is a pair of finite sequences \( (x_1, x_2, \ldots, x_n, y_0, y_1, \ldots, y_n) \), for some natural number \( n \geq 0 \), such that \( x_i \in I \) and \( y_i \in O \) for all \( i \leq n \). That is, a **Moore** \( (I, O) \)-trace is a pair of input sequence and an output sequence, such that the output sequence has length one more than the input sequence. Note that \( n \) may be 0, in which case the input sequence is empty (i.e., has length 0), and the output sequence contains just one output symbol.

Given a **Moore** \( (I, O) \)-trace \( \rho = (x_1, x_2, \ldots, x_n, y_0, y_1, \ldots, y_n) \), and a **Moore** machine \( M = (I, O, Q, q_0, \delta, \lambda) \), we say that \( \rho \) is consistent with \( M \) if \( y_0 = \lambda(q_0) \) and for all \( i = 1, \ldots, n \), \( y_i = \lambda(q_i) \), where \( q_i = \delta(q_{i-1}, x_i) \).

Similarly to the concept of a **Moore** \( (I, O) \)-trace we define a **Moore** \( (I, O) \)-example as a pair of a finite input symbol sequence and an output symbol: \( (x_1, x_2, \ldots, x_n, y) \), where \( x_i \in I \), for \( i = 1, \ldots, n \), and \( y \in O \). We say that a **Moore** machine \( M = (I, O, Q, q_0, \delta, \lambda) \) is consistent with a **Moore** \( (I, O) \)-example \( \rho = (x_1, x_2, \ldots, x_n, y) \) if \( \lambda(\delta^*(x_1, x_2, \ldots, x_n)) = y \).

Since a DFA can be seen as the special case of a **Moore** machine with a binary output alphabet, the concept of a **Moore** \( (I, O) \)-example is naturally carried over to DFAs, in the form of **positive and negative examples**. Specifically, a finite word \( w \) is a positive example for a DFA if it is accepted by the DFA, and a negative example if it is rejected. Viewing a DFA as a **Moore** machine with binary output, a positive example \( w \) corresponds to the **Moore** example \((w, 1)\), while a negative example corresponds to the **Moore** example \((w, 0)\).

### 3.3 Prefix tree acceptors and prefix tree acceptor products

Given a finite and non-empty set of positive examples over a given alphabet \( \Sigma \), \( S_+ \subseteq \Sigma^* \), we can construct, in a non-unique way, a tree-shaped, incomplete DFA, that accepts all words in \( S_+ \), and rejects all others. Such a DFA is called a **prefix tree acceptor** [7] (PTA) for \( S_+ \). For example, a PTA for \( S_+ = \{ b, aa, ab \} \) is shown in Figure 3.

We extend the concept of PTA to **Moore** machines. Suppose that we have a set \( S_{IO} \) of **Moore** \( (I, O) \)-examples. Let \( N = \lceil \log_2 |O| \rceil \) be the number of bits necessary to represent an element of \( O \). Then, given a
Fig. 3: A PTA for $S_+ = \{b, aa, ab\}$.

(a) The PTA for $S_{1+} = \{ab\}$.

(b) The PTA for $S_{2+} = \{aa\}$.

Fig. 4: A PTAP for $S_{IO} = \{(b, 0), (aa, 1), (ab, 2)\}$, with $I = \{a, b\}$, $O = \{0, 1, 2\}$, and $f = \{0 \mapsto (0, 0), 1 \mapsto (0, 1), 2 \mapsto (1, 0)\}$. The positive and negative example sets are: $S_{1+} = \{ab\}$, $S_{1-} = \{b, aa\}$, $S_{2+} = \{aa\}$, $S_{2-} = \{b, ab\}$.

4 Characteristic samples

An important concept in automata learning theory is that of a characteristic sample [7]. A characteristic sample for a DFA is a set of words that captures all information about that automaton’s set of states and behavior. In this paper we extend the concept of characteristic sample to Moore machines.

4.1 Characteristic samples for Moore machines

Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine. Let $<$ denote a total order on input words, i.e., on $I^*$, such that $w < w'$ if either $|w| < |w'|$, or $|w| = |w'|$ but $w$ comes before $w'$ in lexicographic order ($|w|$ denotes the length of a word $w$). For example, $b < aa$ and $aaa < aba$.

Given a state $q \in Q$, we define the shortest prefix of $q$ as the shortest input word which can be used to reach $q$:

$$S_P(q) = \min \{w \in I^* \mid \delta^*(q_0, w) = q\}.$$  

Notice that $M$ is minimal, which implies that all its states are reachable (otherwise we could remove unreachable states). Therefore, $S_P(q)$ is well-defined for every state $q$ of $M$.

Next, we define the set of shortest prefixes of $M$, denoted $S_P(M)$, as:

$$S_P(M) = \{S_P(q) \mid q \in Q\}$$
Let \( M \) words in \( S \) satisfies both Conditions 1 and 2, and therefore is a characteristic sample. Given \( S \) the set when fed to the two machines. The reason why the learning algorithm produces the wrong machine is that two machines of Figure 5 are not equivalent. For instance, the input word Figure 5b (this machine was produced by our MooreMI algorithm, described in Section 5.2). Clearly, the sample of the machine of Figure 5a. If we use \( I,O \) necessary. This becomes clear if we look at machines having the same output on several states. For example, \( \overline{0} \), property, i.e., if \( \overline{0} = 2 = \overline{2} = 2 \), \( \lambda = 2 = \lambda(q_2) \), \( \delta(q_2, a) = q_0 \), \( \delta(q_3, a) = q_2 \) and \( \lambda(q_0) = 0 \neq 2 = \lambda(q_2) \). Notice that \( S_{IO}^2 \) can be constructed from \( S_{IO}^1 \) by extending with the letter \( a \) the input words of the latter that land on \( q_2 \) or \( q_3 \).

The intuition, then, behind Condition 2 is that states in \( M \) that have the same outputs cannot be distinguished by just those (outputs); additional suffixes that differentiate them are required.

### 4.2 Computation, minimality, size, and other properties of characteristic samples

It is easy to see that adding more traces to a characteristic sample preserves the characteristic sample property, i.e., if \( S_{IO} \) is a characteristic sample for a Moore machine \( M \) and \( S_{IO}^1 \supseteq S_{IO} \), then \( S_{IO}^1 \) is also a characteristic sample for \( M \). Also, arbitrarily extending the input word of an existing \((I,O)\)-trace in \( S_{IO} \) and accordingly extending the corresponding output word, again yields a new characteristic sample for \( M \). The questions are raised, then, whether there exist characteristic samples that are minimal in some sense, how many elements they consist of, what are the lengths of their elements, and how can we construct them.

In the following, we outline a simple procedure that, given a minimal Moore machine \( M \), returns a characteristic sample \( S_{IO} \) that is minimal in the sense that removing any \((I,O)\)-trace from it or dropping
and $M$ lengths are bounded by $M$ identical to $M$.

Therefore, (2) affects the final result. It is easy to see now that, after this step, (1) no element of $S$ is a partial order, and therefore transitive, the order in which we remove the redundant elements does not affect the existence of at least one characteristic sample for any minimal Moore machine $M$.

Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine, $S_I$ an initially empty set of input words and $S_{IO}$ the set of $(I, O)$-traces formed by the elements of $S_I$ and the corresponding output words. We compute $S_P(M)$ and $N_L(M)$, and add the elements of the latter to $S_I$. Then, for each pair of words $(u, v) \in S_P(M) \times N_L(M)$ leading to different states $q_u = \delta^*(q_0, u)$, $q_v = \delta^*(q_0, v)$, we compute $M_D(q_u, q_v)$ and add it to $S_I$. Now, $S_{IO}$ already is a characteristic sample. However, it may contain redundant elements that can safely be removed.

We can do this by simply considering each element of $S_I$ and removing it if it is a prefix of another element (this step can be sped up by choosing an appropriate data structure to represent $S_I$, e.g. using a trie, we would simply just keep the words represented by the leaf nodes). Note that since the prefix relation on words is a partial order, and therefore transitive, the order in which we remove the redundant elements does not affect the final result. It is easy to see now that, after this step, (1) no element of $S_I$ is the prefix of another, (2) $S_{IO}$ is still a characteristic sample, and (3) removing any element from $S_I$ or dropping any number of letters at the end of an input word in it (and accordingly adjusting the corresponding output word) will result in a set that is not a characteristic sample. By doing so, we also constructively establish the existence of at least one characteristic sample for any minimal Moore machine $M$.

By definition, there is a $1-1$ correspondence between the elements of $S_P(M)$ and the states of $M$. Therefore, $|S_P(M)| = |Q|$. It follows that $|N_L(M)| \leq |S_P(M)| \cdot |I| + 1 = |Q| \cdot |I| + 1$ and, consequently, $|S_{IO}| = |S_I| \leq |N_L(M)| + |S_P(M)| \cdot |N_L(M)| = (|Q| \cdot |I| + 1) \cdot (|Q| + 1)$. In other words, the size of $S_{IO}$ is $O(|Q|^2|I|)$.

We now provide bounds on the lengths of the elements of $S_{IO}$. The lengths of shortest prefixes are bounded by the longest non-looping path in $M$, which in turn is bounded by $|Q|$. It follows that the nucleus element lengths are bounded by $|Q|+1$. Let now $q_u$ and $q_v$ be different states of $M$ and consider $M_1 = (I, O, Q, q_u, \delta, \lambda)$ and $M_2 = (I, O, Q, q_v, \delta, \lambda)$, i.e. $M_1$ and $M_2$ have $q_u$ and $q_v$ as initial states, respectively, but are otherwise identical to $M$. Finding a (minimum) distinguishing suffix of $q_u$ and $q_v$ is now reduced to finding a (minimum) input word that leads to different output words when transduced by $M_1$ and $M_2$. To find such a word, we first construct a DFA $A = (I, Q \times Q, (q_u, q_v), \delta_A, F)$, where $\forall (q_1, q_2) \in Q \times Q : \forall a \in I : \delta_A((q_1, q_2), a) = (\delta(q_1, a), \delta(q_2, a))$ and $F = \{ (q_1, q_2) \in Q \times Q \mid \lambda(q_1) \neq \lambda(q_2) \}$. A word accepted by this DFA is a distinguishing suffix of $q_u$ and $q_v$, and it is easy to see that we only need to test words of length up to $|Q \times Q|$ in order to find one. We can conclude from the above that the sum of lengths of elements in $S_{IO}$ is $O(|Q|^4|I|)$.
5 Learning Moore machines from Input-Output Traces

5.1 Problem definition

The problem of learning Moore machines from input-output traces (LMoMIO) is defined as follows. Given an input alphabet $I$, an output alphabet $O$, and a set $R_{\text{train}}$ of Moore $(I, O)$-traces, called the training set, we want to synthesize automatically a deterministic, complete Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, such that $M$ is consistent with $R_{\text{train}}$, i.e., $\forall (\rho_1, \rho_O) \in R_{\text{train}} : \lambda^*(\rho_1) = \rho_O$. ($R_{\text{train}}$ is assumed to be itself consistent, in the sense it does not contain two different pairs with the same input word.)

In addition to consistency, we would like to evaluate our learning technique w.r.t. various performance criteria, including:

- Size of $M$, in terms of number of states. Ideally, $M$ should be minimal, that is, if $M'$ is another machine which is also consistent with $R_{\text{train}}$, then $M$ has no more states than $M'$.
- Accuracy of $M$, which, informally speaking, is a measure of how well $M$ performs on a set of traces, $R_{\text{test}}$, different from the training set. $R_{\text{test}}$ is called the test set. Accuracy is a standard criterion in machine learning.
- Complexity (e.g., running time) of the learning algorithm itself.

In the rest of this paper, we present three learning algorithms which solve the LMoMIO problem, and evaluate them w.r.t. the above criteria. Complexity of the algorithm and size of the learned machine are standard notions. Accuracy is standard in machine learning topics such as classification, but not in automata learning. Thus, we elaborate on this concept next.

There are more than one ways to measure the accuracy of a learned Moore machine $M$ against a test set $R_{\text{test}}$. We call an accuracy evaluation policy (AEP) any function that, given a Moore $(I, O)$-trace $(\rho_1, \rho_O)$ and a Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, will return a real number in $[0, 1]$. We will call that number the accuracy of $M$ on $(\rho_1, \rho_O)$. In this paper, we use three AEPs which we call strong, medium, and weak, defined below. Let $(\rho_1, \rho_O) = (x_1 x_2 \cdots x_n, y_0 y_1 \cdots y_n)$ and $z_0 z_1 \cdots z_n = \lambda^*(q_0, \rho_1)$.

- Strong: if $\lambda^*(q_0, \rho_1) = \rho_O$ then 1 else 0.
- Medium: $\frac{1}{n+1} \cdot |\{ i \mid y_0 y_1 \cdots y_i = z_0 z_1 \cdots z_i \}|$.
- Weak: $\frac{1}{n+1} \cdot |\{ i \mid y_i = z_i \}|$.

The strong AEP says that the output of the learned machine $M$ must be identical to the output in the test set. The medium AEP returns the proportion of the largest output prefix that matches. The weak AEP returns the number of output symbols that match. For example, if the correct output is 0012 and $M$ returns 0022 then the strong accuracy is 0, the medium accuracy is $\frac{2}{3}$, and the weak accuracy is $\frac{2}{4}$.

Given an accuracy evaluation policy $f$ and a test set $R_{\text{test}}$, we define the accuracy of $M$ on $R_{\text{test}}$ as the averaged accuracy of $M$ over all traces in $R_{\text{test}}$, i.e.,

$$\frac{\sum_{(\rho_1, \rho_O) \in R_{\text{test}}} f((\rho_1, \rho_O), M)}{|R_{\text{test}}|}.$$ 

It is often the case that the test set $R_{\text{test}}$ contains traces generated by a “black box”, for which we are trying to learn a model. Suppose this black box corresponds to an unknown machine $M_?$. Then, ideally, we would like the learned machine $M$ to be equivalent to $M_?$. In that case, no matter what test set is generated by $M_?$, the learned machine $M$ will always achieve 100% accuracy. Of course, achieving this ideal depends on the training set; if the latter is “poor” then it does not contain enough information to identify the original machine $M_?$. A standard requirement in automata learning theory states that when the training set is a characteristic sample of $M_?$, then the learning algorithm should be able to produce a machine which is equivalent to $M_?$. We call this the characteristic sample requirement (CSR). CSR is important, as it ensures identification in the limit, a key concept in automata learning theory [9]. In what follows, we show that among the algorithms that will be presented in §5.2, only MooreMI satisfies CSR.
Before proceeding, we remark that a given Moore \((I,O)\)-trace \((\rho_I, \rho_O) = (x_1 x_2 \cdots x_n, y_0 y_1 \cdots y_n)\) can be represented as a set of \(n + 1\) Moore \((I,O)\)-examples, specifically \(\{(\epsilon, y_0), (x_1, y_1), (x_1 x_2, y_2), \ldots, (x_1 x_2 \cdots x_n, y_n)\}\). Because of this observation, in all approaches discussed below, there is a preprocessing step to convert the training set, first into an equivalent set of Moore \((I,O)\)-examples, and second, into an equivalent set of \(N\) pairs of positive and negative example sets (the latter conversion was described in §3.3).

### 5.2 Algorithms to solve the LMoMIO problem

**The PTAP algorithm** This algorithm is a rather straightforward one. The set of Moore \((I,O)\)-examples obtained after the preprocessing step described above is used to construct a PTAP, as described in §3.3. Recall that a PTAP is a collection of \(N\) PTAs having the same state-transition structure. The synchronous product of these \(N\) PTAs is then formed, completed, and returned as the result of the algorithm. Completion in this case consists in adding self-loops to states that are missing outgoing transitions for some input symbols. The added self-loops are labeled with the missing input symbols.

Although the PTAP algorithm is relatively easy to implement and runs efficiently, it has several drawbacks. First, since no state minimization is attempted, the resulting Moore machine can be unnecessarily large. Second, and most importantly, the produced machines generally have poor accuracy, since completion is done in a trivial manner.

**The PRPNI algorithm** Again, consider the \(N\) pairs of positive and negative example sets obtained after the preprocessing step. The PRPNI algorithm starts by executing the RPNI DFA learning algorithm [20] on each pair, thus obtaining \(N\) learned DFAs. Then, the synchronous product of these DFAs is formed, completed, and returned as the algorithm result.

The completion step of the PRPNI algorithm is more intricate than the completion step of the PTAP algorithm. The reason is that the synchronous product of the learned DFAs may contain reachable states whose bit encoding does not correspond to any valid output in \(O\). For example, suppose \(O = \{0, 1, 2\}\), so that we need 2 bits to encode it, and thus \(N = 2\) and we use RPNI to learn 2 DFAs. Suppose the encoding is \(0 \mapsto 00, 1 \mapsto 01, 2 \mapsto 10\). This means that the code 11 does not correspond to any valid output in \(O\). However, it can still be the case that in the product of the two DFAs there is a reachable state with the output 11, i.e., where both DFAs are in an accepting state. Note that this problem does not arise in the PTAP algorithm, because all PTAs there have the same state-transition structure, which is also the structure of their synchronous product.

To solve this invalid-code problem, we assign all invalid codes to an arbitrary valid output. In our implementation, we use the lexicographic minimum. In the above example, the code 11 will be assigned to the output 0.

Compared to the PTAP algorithm, the PRPNI algorithm has the advantage of being able to learn a minimal Moore machine when provided with enough \((I,O)\)-traces. However, it can also perform worse in terms of both running time and size (number of states) of the learned machine, due to potential state explosion while forming the DFA product. The PTAP algorithm does not have this problem because, as explained above, the structure, and therefore also the number of states, of the product is identical to those of the component PTAs.

**The MooreMI algorithm** As we saw above, both the PTAP and PRPNI algorithms have several drawbacks. In this section we propose a novel algorithm, called, MooreMI, which remedies these. Moreover, we shall prove that MooreMI satisfies CSR.

The MooreMI algorithm begins by building a PTAP represented as \(N\) PTAs, as in the PTAP algorithm. Then, a merging phase follows, where a merge operation is accepted only if all resulting DFAs are consistent with their respective negative example sets. In addition, a merge operation is either performed on all DFAs at once or not at all. Finally, the synchronous product of the \(N\) learned DFAs is formed, completed by adding self-loops for any missing input symbols, as in the PTAP algorithm, and returned. The pseudocode of the algorithm is given below. The main MooreMI procedure calls the `merge` function as a subroutine. `merge`
computes the result of merging the given red and blue states of the given DFA component. It also performs additional potentially necessary state merges to preserve determinism.

```python
def MooreMI(trace_set, \( \Sigma_I, \Sigma_O \)):
    (list_of_pos_example_sets,
     list_of_neg_example_sets,
     bits_to_output_func) := preprocess_moore_traces(trace_set)
    N := ceil( log_2( |\Sigma_O| ) )
    DFA_list := build_prefix_tree_acceptor_product(
        list_of_pos_example_sets, \( \Sigma_I, \Sigma_O \))
    red = \{ q_e \}
    blue = \{ q_a \text{ for } a \in \Sigma_I \} \cap DFA_list[0].Q
    while blue \neq \emptyset:
        q_blue = pick_next(blue)
        blue := blue \{-q_blue\}
        merge_accepted := false
        for q_red \in red:
            for i \in \{0, ..., N - 1\}:
                new_DFA_list[i] := merge(DFA_list[i], q_red, q_blue)
                if \forall i \in \{0, ..., N - 1\}:
                    is_consistent
                        new_DFA_list[i],
                        list_of_neg_example_sets[i]):
                        merge_accepted := true
                break
        if merge_accepted:
            DFA_list := new_DFA_list
            blue := blue \cup ( \{ one-letter successors of red states \} \
              \cap DFA_list[0].Q )
        else:
            red := red \cup \{q_blue\}
            blue := blue \cup ( \{ one-letter successors of q_blue \} \
              \cap DFA_list[0].Q )
    return product(
        DFA_list,
        bits_to_output_func).make_complete()
```
def merge(DFA, q_red, q_blue):
  q_u := unique_parent_of(q_blue)
a_u := unique_input_from_to(q_u, q_blue)
  DFA.δ(q_u, a_u) := q_red
  merge_stack := [(q_red, q_blue)]
while merge_stack ≠ []:
  (q_1, q_2) := pop(merge_stack)
  if q_1 = q_2 : continue
  if (q_1, q_2) ≠ (q_red, q_blue) and q_2 < q_1:
    q_1, q_2 := q_2, q_1
  DFA.Q := DFA.Q − {q_2}
  if q_2 ∈ DFA.F:
    DFA.F := DFA.F ∪ {q_1}
  for a ∈ DFA.Σ:
    if is defined(DFA.δ(q_2, a)):
      if is defined(DFA.δ(q_1, a)):
        push(merge_stack,
             DFA.δ(q_1, a),
             DFA.δ(q_2, a)))
      else:
        DFA.δ(q_1, a) := DFA.δ(q_2, a)
  return DFA

After the initial preprocessing step (line 6), the algorithm builds a prefix tree acceptor product (line 10) and then repeatedly attempts to merge states in it, in a specific order (line 16). While not appearing in the original RPNI algorithm, the convention of marking states as red or blue was introduced later in [17]. States marked as red represent states that have been processed and will be part of the resulting machine. States marked as blue are immediate successors of red states and represent states that are currently being processed. Initially, the set of red states only contains the initial state q_ϵ, and the set of blue states contains the one-letter successors of q_ϵ (lines 13, 14). Unmarked states will eventually become blue (lines 38, 43), and then either merged with red ones (lines 27, 36) or become red states themselves (line 42).

Most of the auxiliary functions whose implementations are not shown in the pseudocode above have self explanatory names. For instance, the push and pop functions push and pop, respectively, elements to / from a stack, and the functions in lines 53, 54 compute the unique parent of and corresponding input symbol leading to the given blue state (uniqueness of both is guaranteed by the tree-shaped nature of the initial PTA). The function pick_next, however, deserves some additional explanation. Notice first that after the prefix tree acceptor product is constructed and before the merging phase of the algorithm begins, each state can reached by a unique input word which is used to identify that state. For example, the state reached by
the word \textit{abba} is referred to as state \( q_{abba} \). The word used to identify a state may change during merging operations. The total order on words \(<\) defined in §4.1 can now naturally be extended on states of the learned machine as follows: \( q_u < q_v \iff u < v \), in which case we say that \( q_u \) is smaller than \( q_v \). The \textit{pick next} function above simply returns the smallest state of the blue set, according to the order we just defined.

MooreMI is able to learn minimal Moore machines, while avoiding the state explosion and invalid code issues of PRPNI. To see this, notice first that, at every point of the algorithm, the \( N \) learned DFAs are identical in terms of states and transitions, modulo the marking of their states as final. Indeed, this invariant holds by construction for the \( N \) initial prefix tree acceptors, and the additional merge constraints make sure it is maintained throughout the algorithm. Therefore, the product formed at the end of the algorithm can be obtained by simply “overlaying” the \( N \) DFAs on top of one another, as in the PTAP approach, which implies no state explosion. The absence of invalid output codes is also easy to see. Invalid codes generally are results of problematic state tuples in the DFA product, that cannot appear in MooreMI due to the additional merge constraints. Indeed, if a state tuple occurs in the final product, it must also occur in the initial prefix tree acceptor product, and if it occurs there, its code cannot be invalid.

5.3 Properties of the algorithms

All three algorithms described above satisfy consistency w.r.t. the input training set. For PTAP and PRPNI, this a direct consequence of the properties of PTAs, of the basic RPNI algorithm, and of the synchronous product. The proof for MooreMI is somewhat more involved:

**Theorem 1.** The output of the MooreMI algorithm is a complete Moore machine, consistent with the training set. Formally, let \( S_{IO} \) be the set of Moore \((I,O)\)-traces used as input for the algorithm, and let \( M = (I,O,Q,q_0,\delta,\lambda) \) be the resulting Moore machine. Then, \( \delta \) and \( \lambda \) are total functions and \( \forall (\rho_1,\rho_0) \in S_{IO} : \lambda^*(q_0,\rho_1) = \rho_0 \).

**Proof:** The fact that \( \delta \) and \( \lambda \) are total is guaranteed by the final step of the algorithm (line 49). Consistency with the training set can be proved inductively. Let \( N \) denote the number of DFAs learned by the algorithm, which is equal to the number of bits required to represent an element of \( O \). By definition, the Moore machine implicitly defined (by means of a synchronous product) by the \( N \) prefix tree acceptors initially built by the algorithm is consistent with the training set. Assume that, before a merge operation is performed, the Moore machine implicitly defined by the (possibly incomplete) DFAs learned so far is consistent with the training set. It suffices to show that the result of the next merge operation also has this property. Suppose it does not. This means that there exists a \((\rho_1,\rho_0) \in S_{IO} \) such that \( \lambda^*(q_0,\rho_1) \neq \rho_0 \), which implies that in at least one of the learned DFAs, at least one state was added to the corresponding set of final states, while it should not have been (note that performing a merge operation on a DFA always yields a result accepting a superset of the language accepted prior to the merge). In other words, there is at least one learned DFA that is not consistent with its corresponding projection of the training set. However, due to the additional merge constraints that were introduced, this cannot happen, since all DFAs must be compatible with a merge in order for it to take place (line 29).

We now show that MooreMI satisfies the characteristic sample requirement, i.e., if it is fed with a characteristic sample for a machine \( M \), then it learns a machine equivalent to \( M \). If \( M \) is minimal then the learned machine will in fact be isomorphic to \( M \). We first introduce some auxiliary definitions and notation, and make some observations which are important for the proof of the result.

Let \( M = (I,O,Q_m,q_{0m},\delta_m,\lambda_m) \) be the minimal Moore machine from which we derive a characteristic sample, then given as input to the MooreMI algorithm. Let \( M_A = (I,O,Q_A,q_e,\delta_A,\lambda_A) \) be the machine produced by the algorithm. We will use plain \( Q \) and \( \delta \) to denote the state set and possibly partial transition function of the learned machine in an intermediate step of the algorithm.

It can be seen in the pseudocode of the \texttt{merge} function (line 66) that when two states \( q_u, q_v \) are merged in order to preserve determinism, the input word used to identify the resulting state is \( \text{min}_{<} \{u,v\} \), where \(<\) is the total order defined in §4.1. When we say that \( q_u \) is \textit{smaller} than \( q_v \) or \( q_v \) \textit{bigger} than \( q_u \), we will
mean $u = \min \{u, v\}$. We remark that when a blue state is merged with a red one, the latter is always smaller. This is a direct consequence of the tree-shaped nature of the initial prefix tree acceptor product, the fact that blue states are one-letter successors of red ones, and the specific order in which blue states are considered during the merging phase.

By saying that a state $q_a \in Q$ corresponds to a shortest prefix of $M$, we mean that $u \in S_P(M)$. By saying that a state $q_e \in Q$ corresponds to an element in $N_L(M)$, we mean that the state $q_e$ can be reached from $q_e$ using an element in $N_L(M)$.

red and blue refer to the sets of red and blue states, as in the pseudocode of MooreMI. Given a red state $q_u$, we will use $Merged(q_u)$ to denote the set of states that have been merged with / into $q_u$.

In the following, we assume that the training set used as input to the MooreMI algorithm is a characteristic sample for a minimal Moore machine $M$.

**Lemma 1.**

(a) Each red state corresponds to an element of $S_P(M)$ and as a consequence, to a state in $M$.

(b) Each blue state corresponds to an element of $N_L(M)$.

(c) $\forall q_u \in \text{red}: \forall q_e \in Merged(q_u) : \delta^*_m(q_{0,m}, v) = \delta^*_m(q_{0,m}, u)$.

**Proof.** By induction. Initially, red $= \{q_e\}$, blue $\subseteq \{q_u \mid \alpha \in I\}$, and (a), (b), (c) all hold trivially. We assume they hold for the current sets of red, blue and unmarked states and will show they still hold after all possible operations performed by the algorithm:

- If a state $q_e \in \text{blue}$ is merged into a state $q_u \in \text{red}$, then (a) trivially holds: The red state set remains the same, and the successors of $q_e$ are marked blue. Since they now are successors of a state corresponding to a shortest prefix (the red state $q_u$), they correspond to elements in the nucleus of $M$, so (b) holds too. Suppose now that (c) does not hold, i.e. it is $\delta^*_m(q_{0,m}, v) \neq \delta^*_m(q_{0,m}, u)$. Since, by the induction hypothesis, $u \in S_P(M)$ and $q_e$ corresponds to an element in $N_L(M)$, by the characteristic sample definition, there exist $(I, O)$-traces that distinguish $q_e$ and $q_u$ and prohibit their merge. But $q_e$ and $q_u$ were successfully merged, therefore $\delta^*_m(q_{0,m}, v) = \delta^*_m(q_{0,m}, u)$ and (c) holds.

- If a state $q_u \in \text{blue}$ is promoted to a red state, then it is distinct from all other red states. Moreover, since (i) the algorithm considers blue states in a specific order and (ii) whenever we perform a merge between two states $q_u$ and $q_y$ to preserve determinism the result is identified as $q_{\min(x,y)}$, $q_e$ is the smallest state distinct from the existing red states, therefore it corresponds to a shortest prefix. Its successors are now marked blue and since $q_u$ corresponds to a shortest prefix, they correspond to states in $N_L(M)$. Also, since the newly promoted red state is a shortest prefix distinct from the previous ones, it corresponds to a unique, different state in $M$. The above imply that (a) and (b) hold. Moreover, (c) trivially holds too.

- Regarding the additional state merges possibly required to maintain determinism after (1), they can occur between a red and a blue state, in which case the same as in (1) hold, between a blue state and a state that is either blue or unmarked, in which case we have what we want by the induction hypothesis, and between two unmarked states, in which case we do not need to show anything. However, we should mention here that for every pair of states being merged to preserve determinism, the two states involved necessarily represent the same state in $M$. Suppose, without loss of generality that after merging states $q_u$ and $q_v$ as in (1), states $q_{ua} = \delta^*(q_u, a)$ and $q_{va} = \delta^*(q_v, a)$ need to also be merged to preserve determinism. If $q_{ua}$ and $q_{va}$ do not represent the same state in $M$, their minimum distinguishing suffix $w = M_D(q_{ua}, q_{va})$ exists. But then, $\alpha \cdot w$ is a distinguishing suffix for $q_u$ and $q_v$, which means that $q_u$ and $q_v$ represent different states in $M$. However, this cannot be, because, since by the induction hypothesis $u \in S_P(M)$ and $q_e$ corresponds to an element in $N_L(M)$, by the characteristic sample definition, if $q_u$ and $q_v$ were different states, $(I, O)$-traces prohibiting their merge would be present in the algorithm input. Therefore, $q_{ua}$ and $q_{va}$ represent the same state in $M$. The same argument can now be made if e.g. states $q_{uab}$ and $q_{eab}$ need to be merged to preserve determinism after $q_{ua}$ and $q_{va}$ are merged, and so on.

**Lemma 2.** $|Q_m| \leq |QA|$
Proof. Suppose that $|Q_m| > |Q_A|$, i.e. there exists $q \in Q_m$ such that there is no equivalent of $q$ in $Q_A$. However, by the definition of the characteristic sample, the shortest prefix of $q$ appears in the algorithm input, and, according to Lemma 1, it must eventually form a red state on its own. Therefore, there is no such state as $q$, and $|Q_m| \leq |Q_A|$ holds.

**Corollary 1.** The previous lemmas imply the existence of a bijection $f_{iso} : Q_A \rightarrow Q_m$ such that $f_{iso}(q_u) = \delta_m^*(q_{0_m}, u)$.

**Lemma 3.** $\forall q_u \in Q_A : \lambda_A(q_u) = \lambda_m(f_{iso}(q_u))$

Proof. We have shown that $q_u \in Q_A$ corresponds to a unique state in $M$, specifically $\delta_m^*(q_{0_m}, u)$. We have also shown that the algorithm is consistent with the training examples. This implies $\lambda_A(q_u) = \lambda_m(\delta_m^*(q_{0_m}, u))$. Now, since, by definition, $f_{iso}(q_u) = \delta_m^*(q_{0_m}, u)$, we have what we wanted.

**Lemma 4.** $\forall q_u \in Q_A : \forall a \in I : \delta_m(f_{iso}(q_u), a) = f_{iso}(\delta_A(q_u, a))$

Proof. Let $\delta_A(q_u, a) = \delta_A^*(q_e, u \cdot a) = q_v \in Q_A$. By definition, we have $f_{iso}(q_u) = \delta_m^*(q_{0_m}, u)$ and $f_{iso}(q_v) = \delta_m^*(q_{0_m}, v)$. In addition, $\delta_m^*(f_{iso}(q_u), a) = \delta_m^*(\delta_m^*(q_{0_m}, u), a) = \delta_m^*(q_{0_m}, u \cdot a)$. But $\delta_A(q_e, u \cdot a) = q_v = \delta_A^*(q_v, u)$, therefore, from Lemma 1 (c) we have $\delta_m^*(q_{0_m}, u \cdot a) = \delta_m^*(q_{0_m}, v)$. Finally, $\delta_m(f_{iso}(q_u), a) = \delta_m(q_{0_m}, u \cdot a) = f_{iso}(q_v) = f_{iso}(\delta_A(q_u, a))$, as we wanted.

**Theorem 2.** If the input to MooreMI is a characteristic sample of a minimal Moore machine $M$, then the algorithm returns a machine $M_A$ that is isomorphic to $M$.

Proof. Follows from Corollary 1, Lemmas 3, 4 and the observation that $f_{iso}(q_e) = q_{0_m}$. The bijection $f_{iso}$ constitutes a witness isomorphism between $M$ and $M_A$.

### 5.4 Performance optimizations

Compared to the pseudocode our implementation includes several optimizations. First, to limit the amount of copying involved in performing a merge operation, we perform the required state merges in-place, and at the same time record the actions needed to undo them in case the merge is not accepted.

Second, the merge function needs to know the unique (due to the tree-shaped nature of PTAs) parent of the blue state passed to it as an argument. The naive way of doing this, simply iterating over the states until we reach the parent, can seriously harm performance. Instead, in our implementation, we build during PTA construction, and maintain throughout the algorithm, a mapping of states to their parents, and consult this when needed.

Third, in the negative examples consistency test, many of the acceptance checks involved are redundant. For example, suppose that starting from the initial state it is only possible to reach red states (i.e. not blue or unmarked ones) within $n$ steps (transitions). Then, there is no need to include negative examples of length less than $n$ in the consistency test. Our implementation optimizes such cases by integrating the consistency test with the merge operation.

### 5.5 Complexity analysis

Let $I$ and $O$ be the input and output alphabets, and let $S_{IO}$ be the set of Moore $(I, O)$-traces provided as input to the learning algorithms. Let $N = \lceil \log_2(|O|) \rceil$ be the number of bits required to encode the symbols in $O$. Let $S_{1+}, S_{-1}, ..., S_{N+}, S_{N-}$ be the positive and negative example sets obtained by the preprocessing step at the beginning of each algorithm. Let $m_+ = \sum_{i=1}^{N} \sum_{w \in S_{1+}} |w|$, $m_- = \sum_{i=1}^{N} \sum_{w \in S_{-1}} |w|$, and $k = \sum_{(p_i, p_o) \in S_{IO}} |p_i|^2$. The time required for the preprocessing step is $O(N \cdot k)$, and is the same for all three algorithms. The time required for the rest of the phases of each algorithm is

- $O((N + |I|) \cdot m_+)$ for PTAP,
- $O((N + |I|) \cdot m_N^+ + N \cdot m_2^+ \cdot (m_+ + m_-))$ for PRPNI, and
- $O((N + |I|) \cdot m_+ + N \cdot m_N^+ \cdot (m_+ + m_-))$ for MooreMI.

It can be seen that the complexity of MooreMI is no more than logarithmic in the number of output symbols, linear in the number of inputs, and cubic in the total length of training traces.
Table 1: 50 states - average size of training set : 1305 - average input word length in training set : 3.5

| Algo | Time  | States | Accuracy            |
|------|-------|--------|---------------------|
|      |       |        | Strong  | Medium | Weak  |
| 1    | 0.973 | 2113   | 0.0%    | 32.44% | 35.39%|
| 2    | 12.753| 8925   | 0.0%    | 33.82% | 36.57%|
| 3    | 0.348 | 50     | 100.00% | 100.00%| 100.00%|

Table 2: 150 states - average size of training set : 4540 - average input word length in training set : 4

| Algo | Time  | States | Accuracy            |
|------|-------|--------|---------------------|
|      |       |        | Strong  | Medium | Weak  |
| 1    | 8.329 | 7135   | 0.0%    | 28.28% | 31.13%|
| 2    | 60.0  | Timeout| -       | -      | -     |
| 3    | 2.545 | 150    | 100.00% | 100.00%| 100.00%|

6 Implementation & Experiments

All three algorithms presented in §5.2 have been implemented in Python. The source code, including random Moore machine and characteristic sample generation, learning algorithms and testing, spans roughly 2000 lines of code. The code and experiments are available upon request and will be made available online in a future version of this work.

6.1 Experimental comparison

We randomly generated several minimal Moore machines of sizes 50 and 150 states, and input and alphabet sizes $|I| = |O| = 25$. From each such machine, we generated a characteristic sample, and ran each of the three algorithms on this characteristic sample. Then we took the learned machines generated by the algorithms, and evaluated these machines in terms of size (# states) and accuracy. For accuracy, we used a test set of size double the size of the training set. The length of words in the test set was double the maximum training word length.

The results are shown in Tables 1 and 2. “Algo 1,2,3” refers to PTAP, PRPNI, and MooreMI, respectively. “Time” refers to the execution time of the learning algorithm (in seconds). “States” refers to the number of states of the learned machines. For accuracy, we used the three AEPs, Strong, Medium, and Weak, defined in §5.1. In the tables, each row represents the average performance of a specific algorithm over training sets generated by 5 different Moore machines. The only exception is row 2 of Table 1, where one of the 5 experiments timed out and the reported averages are over 4 experiments. “Timeout” means that the algorithm was unable to terminate within the given time limit (60 seconds) in any of the 5 experiments with 150 states.

As expected, MooreMI always achieves 100% accuracy, since the input is a characteristic sample (we verified that indeed the machines learned by MooreMI are in each case equivalent to the original machine that produced the training set). But as it can be seen from the tables, neither PTAP nor PRPNI learn the correct machines, even though the training set is a characteristic sample.

The tables also show that PTAP and PRPNI generate much larger machines than the correct ones. This in turn explains why MooreMI performs better in terms of running time than the other two algorithms, which spend a lot of time completing the large number of generated states.

6.2 Comparison with OSTIA

OSTIA [21] is a well-known algorithm that learns *onward subsequential transducers*, a class of transducers more general than Moore and Mealy machines. Then, a question arising naturally is whether it is possible to use OSTIA for learning Moore machines. In particular, we would like to know what happens when the input
to OSTIA is a set of Moore (I,O)-traces: will OSTIA learn a Moore machine? The answer here is negative, as indicated by an experiment we performed. We constructed a characteristic sample for the Moore machine in Figure 5a and ran the OSTIA algorithm on it (we used the open source implementation described in [1]). The resulting machine is depicted in Figure 6. Notice that there are transitions whose corresponding outputs are words of length more than 1 (e.g., transition label \( b/0122 \)), or even the empty word (output of initial state \( q_0 \)). We conclude that in general OSTIA cannot learn Moore machines, even when the training set is a set of Moore traces, and is also a characteristic sample.

7 Conclusion & Future Work

We presented three algorithms to learn Moore machines from input-output traces. The most advanced of the three algorithms, MooreMI, has desirable performance properties, both theoretical (it satisfies the characteristic sample requirement) as well as in practice.

Future work includes: (1) studying learning for Mealy and other types of state machines; (2) developing incremental versions of the learning algorithms presented here; (3) further implementation and experimentation; and (4) application of the methods presented here for learning models of various types of black-box systems.

References

1. H. I. Akram, C. de la Higuera, H. Xiao, and C. Eckert. Grammatical inference algorithms in matlab. In ICGI’10, Proceedings, pages 262–266. Springer, 2010.
2. A. V. Aleksandrov, S. V. Kazakov, A. A. Sergushichev, F. N. Tsarev, and A. A. Shalyto. The use of evolutionary programming based on training examples for the generation of finite state machines for controlling objects with complex behavior. J. Comput. Sys. Sc. Int., 52(3):410–425, 2013.
3. R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing Finite-state Protocols from Scenarios and Requirements. In HVC, volume 8855 of LNCS. Springer, 2014.
4. D. Angluin. Learning regular sets from queries and counterexamples. Inf. Comput., 75(2):87–106, 1987.
5. I. P. Buzhinsky, V. I. Ulyantsev, D. S. Chivilikhin, and A. A. Shalyto. Inducing finite state machines from training samples using ant colony optimization. J. Comput. Sys. Sc. Int., 55(2):256–266, 2014.
6. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning extended finite state machines. In SEFM 2014, Proceedings, pages 250–264, 2014.
7. C. de la Higuera. Grammatical Inference: Learning Automata and Grammars. CUP, 2010.
8. P. Dupont. Incremental regular inference. In ICGI-96, Proceedings, pages 222–237, 1996.
9. E. M. Gold. Language identification in the limit. Information and Control, 10(5):447–474, 1967.
10. E. M. Gold. Complexity of automaton identification from given data. Information and Control, 37(3):302–320, 1978.
11. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *38th POPL*, pages 317–330, 2011.
12. M. J. H. Heule and S. Verwer. Exact DFA identification using SAT solvers. In J. M. Sempere and P. García, editors, *ICGI*, pages 66–79. Springer, 2010.
13. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI 2012, Proceedings*, pages 251–266, 2012.
14. B. Jonsson. Learning of automata models extended with data. In *SFM 2011, Advanced Lectures*, pages 327–349, 2011.
15. A. V. Karthik, S. Ray, P. Nuzzo, A. Mishchenko, R. Brayton, and J. Roychowdhury. ABCD-NL: Approximating continuous non-linear dynamical systems using purely boolean models for analog/mixed-signal verification. In *ASP-DAC*, pages 327–349, 2011.
16. Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.
17. K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *ICGI-98, Proceedings*, pages 1–12, 1998.
18. L. Ljung, editor. *System Identification (2nd Ed.): Theory for the User*. Prentice Hall, 1999.
19. T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
20. J. Oncina and P. García. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992.
21. J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993.
22. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *ACM SIGSOFT, FSE ’14*, 2014.
23. S. A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *DAC*, pages 356–365, June 2012.
24. M. Shahbaz and R. Groz. Inferring mealy machines. In *FM 2009*, pages 207–222, 2009.
25. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
26. K. Takahashi, A. Fujiyoshi, and T. Kasai. A polynomial time algorithm to infer sequential machines. *Systems and Computers in Japan*, 34(1):59–67, 2003.
27. V. Ulyantsev and F. Tsarev. Extended finite-state machine induction using SAT-solver. In *ICMLA 2011, Volume 2: Special Sessions and Workshop*, pages 346–349, 2011.