International Journal on Software Tools for Technology Transfer (2021) 23:1–29
https://doi.org/10.1007/s10009-019-00544-0

GENERAL

Regular Paper

Learning Moore machines from input–output traces

Georgios Giantamidis[1,2] · Stavros Tripakis[3] · Stylianos Basagiannis[1]

Published online: 6 November 2019

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 develop 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 the 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 has the fundamental *identification in the limit* property. We compare the algorithms experimentally in terms of the size of the learned machine and several notions of accuracy, introduced in this paper. We also carry out a performance comparison against two existing tools (LearnLib and flexfringe). 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*.

Keywords Finite state machine · Moore machine · Mealy machine · Automata learning · Passive learning · Characteristic sample

1 Introduction

An abundance of data from the Internet, sensors, and other sources is revolutionizing many sectors of science, technology, and ultimately our society. At the heart of this revolution lie *machine learning* and *data mining*, 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 [1].

In the context of system design, an important problem with numerous applications is the automatic generation of models from data [2,3]. 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 [4] as well as more recent works on synthesizing programs, controllers, or other artifacts from examples [3,5–10]. Model learning is also closely related to program debugging [11].

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 [12]. Moore machines are typically *deterministic* and *complete*, meaning that for given state and input, the next state is always defined and is unique; 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.

This work was partially supported by the Academy of Finland and the U.S. National Science Foundation (Awards #1329759 and #1139138). This work was partially supported by the Irish Development Agency (IDA) for UTRC Ireland related to Network of Excellence in Aerospace Cyber Physical Systems.

✉ Georgios Giantamidis
GiantaGE@utrc.utc.com

[1] United Technologies Research Centre Ireland, Cork, Ireland

[2] Aalto University, Otaniemi, Finland

[3] Northeastern University, Boston, MA, USA

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

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 formalizes and studies this problem. This is despite a large body of research on *grammatical inference* [13] 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). Apart from the correctness criterion of *consistency* (that the learned machine be consistent with the given traces), we also introduce several *performance* criteria including size and accuracy of the learned machine, and computational complexity of the learning algorithm.
2. We adapt the notion of *characteristic sample*, which is known for DFA [13], 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 [14].
3. We develop 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, only the most advanced among them, called *MooreMI*, satisfies the characteristic sample requirement. We also show that MooreMI achieves identification in the limit.
4. 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. We also adapt our best algorithm (MooreMI) to learn Mealy machines, compare with two existing tools that learn Mealy machines, LearnLib [15] and flexfringe [16] and find that our implementation outperforms both in terms of running time and memory consumption.
5. We show that the well-known transducer-learning algorithm OSTIA [17] cannot generally learn a Moore machine, even in the case where the training set is a characteristic sample of a Moore machine. This implies that an algorithm to learn a more general machine (e.g., a transducer) is not necessarily good at learning a more special machine and therefore further justifies the study of specialized learning algorithms for Moore machines.

An earlier version of this work has appeared as conference paper [18]. The additional contributions of this journal paper with respect to the earlier work are: (1) an extensive description of our learning algorithms, including pseudocode; (2) proofs of the main results, and additional results (lemmas) used in those proofs; (3) a new subsection on performance optimizations (Sect. 5.5); (4) a more detailed complexity analysis (Sect. 5.6); (5) a significantly revised and extended section on implementation and experimental results (Sect. 6); (6) a new section on performance comparison with existing tools (Sect. 7); and (7) an extended bibliography and related work discussion. In particular, Sect. 6 constitutes one of the principal novel contributions of this paper, as it describes in detail our implementation and experimental setups, including information on random machine generation and trace generation, and provides results on randomly generated machines as well as on benchmarks from the literature. We also present a new and improved random trace generation method (Sect. 6.4).

2 Related work

There is a large body of research on learning automata and state machines. In a nutshell, this research can be classified into *active* and *passive* learning, and within passive learning, into *exact* and *heuristic* approaches. Our work falls into the passive, heuristic category. More details are provided below. Automata learning is an active area of research which is currently seing a resurgence (e.g., see [3,19–25]). An excellent recent survey of classic works such as [26,27] as well as recent results, applications and case studies can be found in [2].

Automata and state machine learning can be divided into two broad categories: learning with (examples and) queries

(*active* learning) and learning only from examples (*passive* learning). A seminal work in the first category is Angluin's work on learning DFAs with membership and equivalence queries [28]. This work has been subsequently extended to other types of machines, such as Mealy machines [29], symbolic / extended Mealy machines [30,31], I/O automata [32], register automata [33,34], or hybrid automata [35]. These works are not directly applicable to the problem studied in this paper, as we explicitly forbid both membership and equivalence queries. Therefore, our work is about passive learning. In practice, performing queries (especially complete equivalence queries) is often infeasible.

In the domain of passive learning, a seminal work is Gold's study of learning DFAs from sets of positive and negative examples [14,36]. In this line of work, we must distinguish algorithms that solve the *exact identification* problem, which is to find a *smallest* (in terms of number of states) automaton consistent with the given examples, from those that learn not necessarily a smallest automaton[1] (Let us call them *heuristic* approaches.) Gold showed that exact identification is NP-hard for DFAs [36]. Several works solve the exact identification problem by reducing it into Boolean satisfiability [37,38].

Heuristic approaches are dominated by state merging algorithms like Gold's algorithm for DFAs [36], RPNI [39] (also for DFAs), for which an incremental version also exists [40], and derivatives, like EDSM [41] (which also learns DFAs, but unlike RPNI does not guarantee identification in the limit), OSTIA [17] (which learns subsequential transducers) and others [42–44]. This line of work also includes gravitational search algorithms [45], genetic algorithms [46], ant colony optimization [47], rewriting [48], as well as state splitting algorithms [49]. [45] learns Moore machines, but unlike our work does not guarantee identification in the limit. [46–49] all learn Mealy machines.

All algorithms developed in this paper belong to the heuristic category in the sense that we do not attempt to find a *smallest* machine. However, we would still like to learn a *small* machine. Thus, size is an important *performance* criterion, as explained in Sect. 5.1. Like RPNI and other algorithms, MooreMI is also a state-merging algorithm.

[50] is close to our work, but the algorithm described there does not always yield a deterministic Moore machine, while our algorithms do. This is important because we want to learn systems like digital circuits, embedded controllers (e.g., modeled in Simulink), etc., and such systems are typically deterministic. The k-tails algorithm for finite-state machine inference [51] may also result in non-deterministic machines. Moreover, this algorithm does not generally yield smallest machines, since the initial partition of the input words into equivalence classes (which then become the states of the learned machine) can be overly conservative.[2]

The work in [52] deals with learning finite-state machine abstractions of nonlinear analog circuits. The algorithm described in [52] is very different from ours 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 [52].

Learning from "inexperienced teachers", i.e., by using either (1) only equivalence queries or (2) equivalence plus membership queries that may be answered inconclusively, has been studied in [53,54].

Related but different from our work are approaches which synthesize state machines from *scenarios and requirements*. Scenarios can be provided in various forms, e.g., message sequence charts [9,10], event sequence charts [55], or simply, input–output examples [56]. Requirements can be temporal logic formulas as in [9,10,56], or other types of constraints such as the *scenario constraints* used in [55]. In this paper, we have examples, but no requirements.

Also related but different from ours is work in the areas of *invariant generation* and *specification mining*, which extract properties of a program or system model, such as invariants [57–60], temporal logic formulas [61,62] or non-deterministic finite automata [63].

FSM learning is related to FSM testing [64]. In particular, notions similar to the *nucleus* of an FSM and to *distinguishing suffixes* of states, which are used to define characteristic samples, are also used in [65,66]. The connection between conformance testing and regular inference is made explicit in [67] and [64] describes how an active learning algorithm can be used for fault detection.

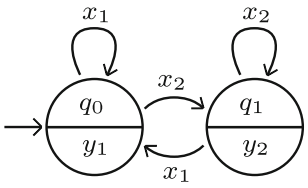
3 Preliminaries

3.1 Finite-state machines and automata

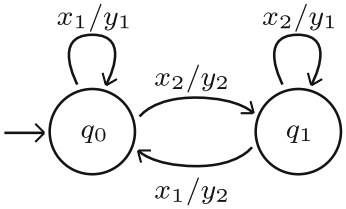
A *finite-state machine* (FSM) is a tuple M of the form $M = (I, O, Q, q_0, \delta, \lambda)$, where:

[1] The term *smallest* automaton is used in the exact identification problem, instead of the more well-known term *minimal* automaton. Among equivalent machines, one with the fewest states is called *minimal*. Among machines which are all consistent with a set of traces but not necessarily equivalent, one with the fewest states is called *smallest*.

[2] We have implemented the k-tails algorithm and applied it on the characteristic sample for the Moore machine in Fig. 5a, described in Sect. 4.1. Using $k = 0$, we get a non-deterministic machine of three states. Using any $k > 0$, we get a deterministic machine of eight states. This excessive number of states is due to the way the k-tails equivalence relation is defined. In particular, in order for two input words to be considered equivalent, they must have successors in the training set with the same letters. This implies that a word with no successors in the training set can never be equivalent with a word with some successors, even if both words represent the same state in the target machine.



(a) Moore machine M_1 on input-output sets $I = \{x_1, x_2\}$ and $O = \{y_1, y_2\}$.



(b) Mealy machine M_2 on input-output sets $I = \{x_1, x_2\}$ and $O = \{y_1, y_2\}$.

Fig. 1 Examples of finite-state machines

- 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*.
- $\delta : Q \times I \to Q$ is the *transition function*.
- λ is the *output function*, which can be of two types:
  - $\lambda : Q \to O$, in which case the FSM is a *Moore machine*.
  - $\lambda : Q \times I \to O$, in which case the FSM is a *Mealy machine*.

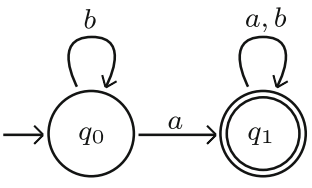
If both δ and λ are total functions, we say that the FSM is *complete*. If any of δ and λ is a partial function, we say that the FSM is *incomplete*. Examples of a Moore and a Mealy machine are given in Fig. 1. Both FSMs are complete.

We also define $\delta^* : Q \times I^* \to Q$ 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 $a \in I$:

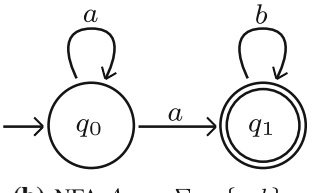
- $\delta^*(q, \epsilon) = q$.
- $\delta^*(q, w \cdot a) = \delta(\delta^*(q, w), a)$.

We also define $\lambda^* : Q \times I^* \to O^*$. The rest of this paper focuses on Moore machines; thus, we define λ^* only in the case where M is a Moore machine (the adaptation to a Mealy machine is straightforward):

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



(a) DFA A_1 on $\Sigma = \{a, b\}$.



(b) NFA A_2 on $\Sigma = \{a, b\}$.

Fig. 2 Examples of finite-state automata

Two Moore machines M_1, M_2, with $M_i = (I_i, O_i, Q_i, q_{0\_i}, \delta_i, \lambda_i)$ are said to be *equivalent* iff $I_1 = I_2$, $O_1 = O_2$, 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:

- Σ (the *alphabet*) is a finite set of *letters*.
- Q is a finite set of *states*.
- $q_0 \in S$ is the *initial state*.
- $\delta : Q \times \Sigma \to Q$ is the *transition function*.
- $F \subseteq Q$ is the set of *accepting states*.

A DFA can be seen as a special case of a Moore machine, where the set of input symbols I is Σ, and the set of output symbols is binary, say $O = \{0, 1\}$, with 1 and 0 corresponding to accepting and non-accepting states, respectively. The concepts of *complete* and *incomplete* DFAs, as well as the definition of δ^*, are similar to the corresponding ones for FSMs. Elements of Σ^* are usually called *words*. A DFA $A = (\Sigma, Q, q_0, \delta, F)$ is said to accept a word w if $\delta^*(q_0, w) \in F$.

A *non-deterministic finite automaton* (NFA) is a tuple $A = (\Sigma, Q, Q_0, \Delta, F)$, where Σ, Q, and F are as in a DFA, and:

- $Q_0 \subseteq Q$ is the *set of initial states*.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

Examples of a DFA and an NFA are given in Fig. 2. Accepting states are drawn with double circles.

Given two NFAs, $A_1 = (\Sigma, Q_1, Q_0^1, \Delta_1, F_1)$ and $A_2 = (\Sigma, Q_2, Q_0^2, \Delta_2, F_2)$, their *synchronous product* is the NFA $A = (\Sigma, Q_1 \times Q_2, Q_0^1 \times Q_0^2, \Delta, F_1 \times F_2)$, where $((q_1, q_2), a, (q_1', q_2')) \in \Delta$ iff $(q_1, a, q_1') \in \Delta_1$ and $(q_2, a, q_2') \in \Delta_2$. The synchronous product of automata is used in several algorithms presented in the sequel.

3.2 Input–output traces and examples

Given sets of input and output symbols I and O, respectively, a *Moore* (I, O)*-trace* is a pair of finite sequences $(x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$, 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 a 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 $\rho = (x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$ and a Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, we say that ρ *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: $(x_1 x_2 \cdots x_n,\ y)$, 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 $\rho = (x_1 x_2 \cdots x_n,\ y)$ if $\lambda(\delta^*(q_0, x_1 x_2 \cdots x_n)) = y$.

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 Σ, $S_+ \subseteq \Sigma^*$, 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* [13] (PTA) for S_+. For example, a PTA for $S_+ = \{b, aa, ab\}$ is shown in Fig. 3. The reason why the construction is non-unique is because we can always extend the tree with non-accepting branches. For example, the state q_{aa} in Fig. 4a could be removed without changing the set of words accepted by that PTA.

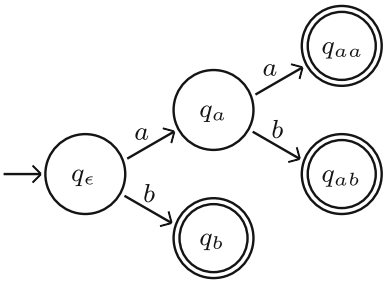


Fig. 3 A PTA for $S_+ = \{b, aa, ab\}$

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 function f that maps elements of O to bit tuples of length N, we can map S_{IO} to N pairs of positive and negative example sets, $\{(S_{1+}, S_{1-}), (S_{2+}, S_{2-}), \ldots, (S_{N+}, S_{N-})\}$. In particular, for each pair $(w, y) \in S_{IO}$, if the i-th element of $f(y)$ is 1, then S_{i+} should contain w and S_{i-} should not. Similarly, if the i-th element of $f(y)$ is 0, then S_{i-} should contain w and S_{i+} should not.

We can subsequently construct a *prefix tree acceptor product* (PTAP), which is a collection of N PTAs (one for each positive example set, S_{i+}, for $i = 1, \ldots, N$) that have the same state-transition structure. An example of a PTAP consisting of two PTAs is given in Fig. 4.

4 Characteristic samples

An important concept in automata learning theory is that of a *characteristic sample* [13].[3] 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

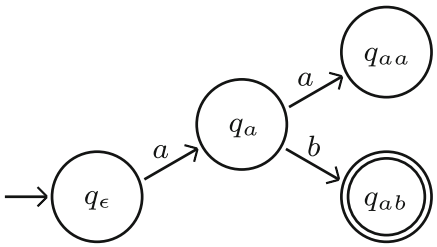
Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine. Let $<$ denote a total order on input words, i.e., on I^*, such that $w < w'$ iff either $|w| < |w'|$, or $|w| = |w'|$ but w comes before w' in lexicographic order. ($|w|$ denotes the length of a word w.) For example, $b < aa$ and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

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:

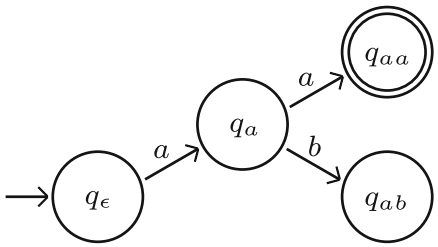
$$S_P(q) = min_<\{w \in I^* \mid \delta^*(q_0, w) = q\}.$$

[3] Note that there are generally different kinds of characteristic samples for different learners [13]. In this paper, our definition of the characteristic sample is designed with our MooreMI algorithm in mind, which is the natural extension for Moore machines of the RPNI algorithm.

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: $S_{1+} = \{ab\}$, $S_{1-} = \{b, aa\}$, $S_{2+} = \{aa\}$, $S_{2-} = \{b, ab\}$



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



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

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:

$$S_P(M) = \{S_P(q) \mid q \in Q\}$$

We can now define the *nucleus* of M which contains the empty word and all one-letter extensions of words in $S_P(M)$:

$$N_L(M) = \{\epsilon\} \cup \{w \cdot a \mid w \in S_P(M),\ a \in I\}.$$

We also define the *minimum distinguishing suffix* for two different states q_u and q_v of M, as follows:

$$M_D(q_u, q_v) = min_<\{w \in I^* \mid \lambda^*(q_u, w) \neq \lambda^*(q_v, w)\}.$$

$M_D(q_u, q_v)$ is guaranteed to exist for any two states q_u, q_v because M is minimal.

Let W be a set of input words, $W \subseteq I^*$. $Pref(W)$ denotes the set of all prefixes of all words in W:

$$Pref(W) = \{x \in I^* \mid \exists w \in W, y \in I^* : x \cdot y = w\}.$$

Definition 1 Let S_{IO} be a set of Moore (I,O)-traces, and let S_I be the corresponding set of input words: $S_I = \{\rho_I \in I^* \mid (\rho_I, \rho_O) \in S_{IO}\}$. S_{IO} is a *characteristic sample* for a Moore machine M iff:

1. $N_L(M) \subseteq Pref(S_I)$.
2. $\forall u \in S_P(M) : \forall v \in N_L(M) : \forall w \in I^* :$

$$\delta^*(q_0, u) \neq \delta^*(q_0, v) \wedge w = M_D(\delta^*(q_0, u), \delta^*(q_0, v)) \Rightarrow \{u \cdot w, v \cdot w\} \subseteq Pref(S_I).$$

For example, consider the Moore machine M_1 from Fig. 1. We have: $S_P(q_0) = \epsilon$, $S_P(q_1) = x_2$, $S_P(M_1) = \{\epsilon, x_2\}$, and $N_L(M_1) = \{\epsilon, x_1, x_2, x_2x_1, x_2x_2\}$. The following set is a characteristic sample for M_1:

$$S_{IO} = \{\ (x_1,\ y_1y_1),\ (x_2x_1,\ y_1y_2y_1),\ (x_2x_2,\ y_1y_2y_2)\ \}.$$

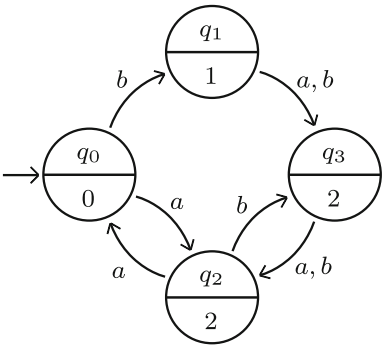
While it is intuitive that a characteristic sample should contain input words that in a sense *cover* all states and transitions of M (Condition 1 of Definition 1), it may not be obvious why Condition 2 of Definition 1 is necessary. This becomes clear if we look at machines having the same output on several states. For example, consider the Moore machine M in Fig. 5a. The set of (I, O)-traces $S^1_{IO} = \{(aa, 020), (ba, 012), (bb, 012), (aba, 0222), (abb, 0222)\}$ satisfies Condition 1 but not Condition 2 (because $S_P(q_2) = a$, $ba \in N_L(M)$, $\delta^*(q_0, ba) = q_3$, $M_D(q_2, q_3) = a$, but no input word in S^1_{IO} has baa as a prefix) and therefore is not a characteristic sample of the machine of Fig. 5a. If we use S^1_{IO} to learn a Moore machine, we obtain the machine in Fig. 5b. (This machine was produced by our MooreMI algorithm, described in Sect. 5.3.3.) Clearly, the two machines of Fig. 5 are not equivalent. For instance, the input word baa results in different outputs when fed to the two machines. The reason why the learning algorithm produces the wrong machine is that the set S^1_{IO} does not contain enough information to clearly distinguish between states q_2 and q_3.

Instead, consider the set $S^2_{IO} = \{(aa, 020), (baa, 0122), (bba, 0122), (abaa, 02220), (abba, 02220)\}$. S^2_{IO} satisfies both Conditions 1 and 2 and therefore is a characteristic sample. Given S^2_{IO} as input, our MooreMI algorithm is able to learn the correct machine, i.e., the machine of Fig. 5a. In this case, the minimum distinguishing suffix of states q_2 and q_3 is simply the letter a, since $\delta(q_2, a) = q_0$, $\delta(q_3, a) = q_2$ and $\lambda(q_0) = 0 \neq 2 = \lambda(q_2)$. Notice that S^2_{IO} can be constructed from S^1_{IO} by extending with the letter a the input words of the latter that land on q_2 or q_3.

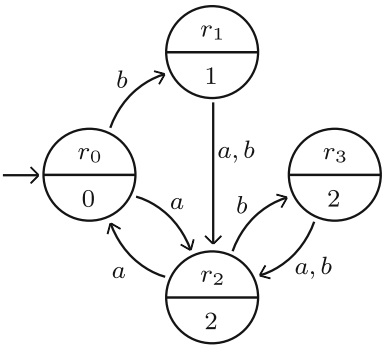
The intuition, then, behind Condition 2 is that states in M that have the same outputs cannot be distinguished by just those (outputs); additional suffixes that differentiate them are required.

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 $S'_{IO} \supseteq S_{IO}$, then S'_{IO} is also a characteristic sam-



(a) Target minimal Moore machine.



(b) Moore machine learned by our MooreMI algorithm if we use a set of traces that does not satisfy Condition 2 of Definition 1.

Fig. 5 Example illustrating the need for Condition 2 of Definition 1

ple 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 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.

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 and of it will result in S_{IO} not being a characteristic sample.

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_{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_{train}, i.e., $\forall\ (\rho_I, \rho_O) \in R_{\text{train}} : \lambda^*(\rho_I) = \rho_O$. ($R_{\text{train}}$ 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. Note that, contrary to the *exact identification* problem [36], we do *not require* M to be the smallest (in terms of number of states) machine consistent with R_{train}.
- *Accuracy* of M, which, informally speaking, is a measure of how well M performs on a set of traces, R_{test}, *different* from the training set. R_{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_{test}. We call an *accuracy evaluation policy* (AEP) any function that, given a Moore (I, O)-trace (ρ_I, ρ_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 (ρ_I, ρ_O). In this paper, we use three AEPs which we call *strong*, *medium*, and *weak*, defined below. Let $(\rho_I, \rho_O) = (x_1x_2\cdots x_n,\ y_0y_1\cdots y_n)$ and $z_0z_1\cdots z_n = \lambda^*(q_0, \rho_I)$.

- *Strong*: if $\lambda^*(q_0, \rho_I) = \rho_O$ then 1 else 0.
- *Medium*: $\frac{1}{n+1} \cdot |\{i \mid y_0y_1\cdots y_i = z_0z_1\cdots z_i\}|$.
- *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 $\frac{2}{4}$, and the weak accuracy is $\frac{3}{4}$. Ideally, we want the learned machine to achieve a high accuracy with respect to the strong AEP. However, the medium and weak AEPs are also useful, because they allow to distinguish, say, a machine which is "almost right" (i.e., outputs the right sequence except for a few symbols) from a machine which is always or almost always wrong. In fact, the medium and weak AEPs were inspired, respectively, by the hierarchical loss and hamming loss criteria, which appear in multi-label classification problems in the machine learning community [68].

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

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

It is often the case that the test set R_{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 [14]. In what follows, we show that among the algorithms that will be presented in Sect. 5.3, only MooreMI satisfies CSR.

5.2 Trace preprocessing

Before proceeding, we remark that a given Moore (I, O)-trace $(\rho_I, \rho_O) = (x_1x_2\cdots x_n,\ y_0y_1\cdots y_n)$ can be represented as a set of $n+1$ Moore (I, O)-examples, specifically $\{(\epsilon, y_0), (x_1, y_1), (x_1x_2, y_2), \ldots, (x_1x_2\cdots x_n, y_n)\}$. 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 Sect. 3.3.) During the latter conversion, we also construct a partial mapping g from bit tuples to output letters, which we make use of later in the algorithms. As an example of how this is computed, consider Fig. 4. In this case, g would simply be the inverse of the function f, i.e., we would have $g = \{(0, 0) \mapsto 0,\ (0, 1) \mapsto 1,\ (1, 0) \mapsto 2\}$. This example also illustrates that the mapping is partial, since the bit tuple $(1, 1)$ is unmapped. The partial mapping g is referred to as `bits_to_output_func` in all pseudocode snippets.

5.3 Algorithms to solve the LMoMIO problem

5.3.1 The PTAP algorithm

This algorithm is a rather straightforward one (Fig. 6). The set of Moore (I, O)-examples obtained after the preprocessing

```
def PTAP(trace_set, Σ_I, Σ_O):

  (list_of_pos_example_sets,
   list_of_neg_example_sets,
   bits_to_output_func)
     := preprocess_moore_traces(trace_set)

  DFA_list := build_prefix_tree_acceptor_product(
              list_of_pos_example_sets, Σ_I, Σ_O)

  A := product(DFA_list, bits_to_output_func)

  return A.make_complete()
```

Fig. 6 The PTAP algorithm

```
def PRPNI(trace_set, Σ_I, Σ_O):

  (list_of_pos_example_sets,
   list_of_neg_example_sets,
   bits_to_output_func)
     := preprocess_moore_traces(trace_set)

  DFA_set = ∅

  for (S_+, S_−) ∈ zip(
        list_of_pos_example_sets,
        list_of_neg_example_sets):
    DFA_set := DFA_set ∪ { RPNI(S_+, S_−, Σ_I) }

  A := product(DFA_set, bits_to_output_func)

  return A.make_complete().fix_invalid_codes()
```

Fig. 7 The PRPNI algorithm

```
def RPNI(S_+, S_−, Σ):

  DFA := build_prefix_tree_acceptor(S_+, Σ)

  red := { q_ε }
  blue := { q_a | a ∈ Σ } ∩ DFA.Q

  while blue ≠ ∅:

    q_blue := pick_next(blue)
    blue := blue − {q_blue}

    merge_accepted := false

    for q_red ∈ red:

      new_DFA := merge(DFA, q_red, q_blue)

      if is_consistent(new_DFA, S_−):
        merge_accepted := true
        break

    if merge_accepted:
      DFA := new_DFA
      blue := blue ∪ ( { one−letter
        successors of red states }
        ∩ DFA.Q )
    else:
      red := red ∪ {q_blue}
      blue := blue ∪ ( { one−letter
        successors of q_blue }
        ∩ DFA.Q )

  return DFA
```

Fig. 8 The RPNI algorithm. The *merge* procedure is shown in Fig. 9 and discussed in Sect. 5.3.3. The *pick_next* function returns the smallest state of the blue set, according to the order defined in Sect. 5.3.3. *is_consistent* simply checks whether the given DFA rejects all words in the given negative example set. If so, it returns true. Otherwise, it returns false

step described above (Sect. 5.2) is used to construct a PTAP, as described in Sect. 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. Note that a PTA is a special case of an NFA: The PTA is deterministic, but it is generally incomplete. The synchronous product of PTAs is therefore the same as the synchronous product of NFAs. The product of PTAs is deterministic, but also generally incomplete and therefore needs to be completed in order to yield a complete DFA. *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.

5.3.2 The PRPNI algorithm

Again, consider the N pairs of positive and negative example sets obtained after the preprocessing step of Sect. 5.2. The PRPNI algorithm (Fig. 7) starts by executing the RPNI DFA learning algorithm [39] on each pair, thus obtaining N learned DFAs. For completion, RPNI is recalled in Fig. 8. The zip function turns two lists into a list of pairs (e.g., $zip([1,2,3],[a,b,c]) = [(1,a),(2,b),(3,c)]$). Once the N DFAs have been obtained, their synchronous product is formed, completed, and returned as the algorithm result. As in the case of the PTAP algorithm, the synchronous product of the DFAs in the PRPNI algorithm is deterministic, but generally not complete.

In the case of PRPNI, an additional post-processing step (called *fix_invalid_codes* in the pseudocode) follows the completion step (*make_complete*). 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 $O = \{0,1,2\}$,

```
1  def merge(DFA, q_red, q_blue):
2
3    q_u := unique_parent_of(q_blue)
4    a_u := unique_input_from_to(q_u, q_blue)
5
6    DFA.δ(q_u, a_u) := q_red
7
8    merge_stack := [(q_red, q_blue)]
9
10   while merge_stack ≠ []:
11
12     (q_1, q_2) := pop(merge_stack)
13
14     if q_1 = q_2 : continue
15
16     if (q_1, q_2) ≠ (q_red, q_blue)
17         and q_2 < q_1:
18       q_1, q_2 := q_2, q_1
19
20     DFA.Q := DFA.Q − {q_2}
21
22     if q_2 ∈ DFA.F:
23       DFA.F := DFA.F ∪ {q_1}
24
25     for a ∈ DFA.Σ:
26       if is_defined(DFA.δ(q_2, a)):
27         if is_defined(DFA.δ(q_1, a)):
28
29           push(merge_stack,
30                 DFA.δ(q_1, a),
31                 DFA.δ(q_2, a)))
32         else:
33           DFA.δ(q_1, a) := DFA.δ(q_2, a)
34
35   return DFA
```

Fig. 9 The *merge* procedure

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 are guaranteed to 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.

5.3.3 The MooreMI algorithm

As explained above, both PTAP and PRPNI have several drawbacks. In this section, we propose a novel algorithm called MooreMI, which remedies these. Moreover, we prove that MooreMI satisfies CSR. In fact, as we shall see in Sect. 5.4, MooreMI is the only one among these three algorithms that satisfies CSR.

The MooreMI algorithm (Fig. 10) 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.

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 [41]. 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 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 3, 4 (Fig. 9) 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

```
1  def MooreMI(trace_set, Σ_I, Σ_O):
2
3    (list_of_pos_example_sets,
4     list_of_neg_example_sets,
5     bits_to_output_func)
6       := preprocess_moore_traces(trace_set)
7
8    N := ceil( log_2( |Σ_O| ) )
9
10   DFA_list := build_prefix_tree_acceptor_product(
11             list_of_pos_example_sets, Σ_I, Σ_O)
12
13   red := { q_ε }
14   blue := { q_a | a ∈ Σ_I } ∩ DFA_list[0].Q
15
16   while blue ≠ ∅:
17
18     q_blue := pick_next(blue)
19     blue := blue − {q_blue}
20
21     merge_accepted := false
22
23     for q_red ∈ red:
24
25       for i ∈ {0, ..., N − 1}:
26         new_DFA_list[i] :=
27           merge(DFA_list[i], q_red, q_blue)
28
29       if ∀ i ∈ {0, ..., N − 1}:
30              is_consistent(
31                   new_DFA_list[i],
32                   list_of_neg_example_sets[i]):
33           merge_accepted := true
34           break
35
36     if merge_accepted:
37       DFA_list := new_DFA_list
38       blue := blue ∪ ( { one−letter
39           successors of red states }
40           ∩ DFA_list[0].Q )
41     else:
42       red := red ∪ {q_blue}
43       blue := blue ∪ ( { one−letter
44           successors of q_blue }
45           ∩ DFA_list[0].Q )
46
47   return product(
48     DFA_list,
49     bits_to_output_func).make_complete()
```

Fig. 10 The MooreMI algorithm

the prefix tree acceptor product is constructed and before the merging phase of the algorithm begins, each state can be reached by a unique input word which is used to identify that state. For example, the state reached by the word *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 Sect. 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 *pick_next* function 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.

5.4 Properties of the algorithms

All three algorithms described above satisfy consistency w.r.t. the input training set. For PTAP and PRPNI, this is 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, therefore the result for MooreMI is stated as a theorem:

Theorem 1 (Consistency) *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, δ and λ are total functions and $\forall\ (\rho_I, \rho_O) \in S_{IO} : \lambda^*(q_0, \rho_I) = \rho_O$.*

Proof The fact that δ and λ 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 $(\rho_I, \rho_O) \in S_{IO}$, such that $\lambda^*(q_0, \rho_I) \neq \rho_O$, 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 $M = (I, O, Q_m, q_{0\_m}, \delta_m, \lambda_m)$ be the minimal Moore machine from which we derive a characteristic sample, then given as input to the MooreMI algorithm. Let $M_A = (I, O, Q_A, q_\epsilon, \delta_A, \lambda_A)$ be the machine produced by the algorithm. We will use plain Q and δ 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 *merge* function (line 16) 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 $min_<\{u, v\}$, where $<$ is the total order defined in Sect. 4.1. When we say that q_u is *smaller* than q_v or q_v *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 $q_u \in Q$ *corresponds* to a shortest prefix of M, we mean that $u \in S_P(M)$. By saying that a state $q_v \in Q$ *corresponds* to an element in $N_L(M)$, we mean that the state q_v can be reached from q_ϵ 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) $\forall q_u \in$ `red` $: \forall q_v \in Merged(q_u) : \delta^*_m(q_{0\_m}, v) = \delta^*_m(q_{0\_m}, u)$.

Proof By induction. Initially, `red` $= \{q_\epsilon\}$, `blue` $\subseteq \{q_a \mid a \in I\}$, 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:

(1) If a state $q_v \in$ `blue` is merged into a state $q_u \in$ `red`, then (a) trivially holds: The red state set remains the same, and the successors of q_v 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_v corresponds to an element in $N_L(M)$, by the characteristic sample definition, there exist (I, O)-traces that distinguish q_v and q_u and prohibit their merge. But q_v and q_u were successfully merged; therefore, $\delta^*_m(q_{0\_m}, v) = \delta^*_m(q_{0\_m}, u)$ and (c) holds.
(2) If a state $q_v \in$ `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_x and q_y to preserve determinism the result is identified as $q_{min_<(x,y)}$, q_v 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_v 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.
(3) 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, $a \cdot w$ 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_v corresponds to an element in $N_L(M)$, by the char-

acteristic 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_{vab} need to be merged to preserve determinism after q_{ua} and q_{va} are merged, and so on. □

Lemma 2 $|Q_m| \leq |Q_A|$.

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 $\delta_A(q_u, a) = \delta_A^*(q_\epsilon, u \cdot a) = q_v \in Q_A$. 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 $\delta_A^*(q_\epsilon, u \cdot a) = q_v = \delta_A^*(q_\epsilon, v)$, therefore, from Lemma 1 (c) we have $\delta_m^*(q_{0\_m}, u \cdot a) = \delta_m^*(q_{0\_m}, v)$. Finally, $\delta_m(f_{iso}(q_u), a) = \delta_m(q_{0\_m}, u \cdot a) = \delta_m(q_{0\_m}, v) = f_{iso}(q_v) = f_{iso}(\delta_A(q_u, a))$, as we wanted. □

Theorem 2 (Characteristic sample requirement) *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 $f_{iso}(q_\epsilon) = q_{0\_m}$. The bijection f_{iso} constitutes a witness isomorphism between M and M_A. □

Finally, we show that the MooreMI algorithm achieves identification in the limit.

Theorem 3 (Identification in the limit) *Let* $M = (I, O, Q, q_0, \delta, \lambda)$ *be a minimal Moore machine. Let* $(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \ldots$ *be an infinite sequence of* (I, O)*-traces generated from M, such that* $\forall \rho \in I^* : \exists i : \rho = \rho_I^i$ *(i.e., every input word appears at least once in the sequence). Then there exists index k such that for all $n \geq k$, the MooreMI algorithm learns a machine equivalent to M when given as input the training set* $\{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \ldots, (\rho_I^n, \rho_O^n)\}$.

Proof Let $S_{IO}^n = \{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \ldots, (\rho_I^n, \rho_O^n)\}$, for any index n. Since M is a minimal Moore machine, there exists at least one characteristic sample $S_{IO} = \{(r_I^1, r_O^1), (r_I^2, r_O^2), \ldots, (r_I^N, r_O^N)\}$ for it. By the hypothesis, $\forall j \in \{1, \ldots, N\} : \exists i_j$ such that $\rho_I^{i_j} = r_I^j$. Let then $k = max_{j \in \{1,\ldots,N\}} i_j$. It is easy to see now that $S_{IO}^k = \{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \ldots, (\rho_I^k, \rho_O^k)\}$ is a characteristic sample (as a superset of S_{IO}). From the properties of characteristic samples, it also follows that for any $n \geq k$, S_{IO}^n is also a characteristic sample. (Since in that case $S_{IO}^n \supseteq S_{IO}^k$.) Finally, from Theorem 2, when MooreMI is given S_{IO}^n, for any $n \geq k$, as input, it will output a Moore machine isomorphic, and therefore equivalent, to M. □

What about the PTAP and PRPNI algorithms? Do they achieve identification in the limit? It is easy to see that PTAP does not achieve identification in the limit in general. The reason for this error is due to the trivial completion method that PTAP uses (i.e., self-loops in the leaves of the PTA). For example, consider the input alphabet $\{a\}$ and a binary output alphabet, so that Moore machines reduce to DFAs. Suppose we want to learn the regular language $(aa)^*$. Every word with an even number of a's belongs to this language, whereas every word with an odd number of a's does not. Note that since the output alphabet is binary, the PTA product computed by PTAP contains a single PTA. Also note that, no matter what training set we use, this PTA has the form of a "chain" of states linked with a-labeled transitions. After completion, the last state in this chain has an a-labeled self-loop. Regardless of whether the last state in the chain is accepting or rejecting, the self-loop implies that the learned automaton will incorrectly accept / reject some words. Since this happens no matter how large a training set we use, PTAP does not identify in the limit.

As for PRPNI, even if it identifies in the limit, it does not satisfy the CSR property. This is demonstrated for example in Table 1 in the experimental section that follows. As we can see in that table, both PTAP and PRPNI achieve $\leq 1\%$ strong accuracy, even though a characteristic sample is used as the training set. Note that this does not mean that PRPNI does not admit a different kind of characteristic sample—it only means that a characteristic sample as defined in Sect. 4.1 is not enough for PRPNI to identify the correct machine. Indeed, since PRPNI is unable to take into account merge consistency information across the individually learned DFAs, this information needs to come in the form of additional training traces. Therefore, while it is possible that PRPNI also admits a different kind of characteristic

sample, it is expected that this will generally be larger than (perhaps a superset of) the one defined in this paper.

5.5 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. In particular, we construct the initial PTAs based not only on positive but also on negative examples, and mark states not only as accepting but also as rejecting when appropriate, as described in [69]. Then, during the merge operation, if an attempt to merge an accepting state with a rejecting one occurs, the merge is rejected.

Finally, note that the decomposition of Moore (I, O)-traces into positive and negative examples, described in Sect. 5.2, while necessary for the PRPNI algorithm, is not needed for PTAP or MooreMI. The last two can work directly on a set of Moore (I, O)-traces, translating it into PTAs augmented with state output on every node. In turn, this output information can be used in the consistency check during merging. In order to keep the presentation of the algorithms uniform and prevent the reader from context switching, we decided to make use of said decomposition in the presentation of all three algorithms. However, in our implementation both PTAP and MooreMI operate directly on the given set of Moore (I, O)-traces, which also provides a nice performance boost.

5.6 Complexity analysis

In order to build a prefix tree acceptor, we need to consider all prefixes of words in the set of positive examples S_+. This yields a complexity of $O(\sum_{w\in S_+} |w|)$, where $|w|$ indicates the length of the word w. A prefix tree acceptor product is represented by N prefix tree acceptors that have the same state transition structure, where N is the number of bits required to represent an output letter. Therefore, constructing a prefix tree acceptor product having $2^{N-1} < |O| \leq 2^N$ distinct output symbols, requires $O(N \cdot \sum_{w\in S_+^{all}} |w|)$ work, where S_+^{all} denotes the union of the N positive example sets, S_{i+}. (We need to consider all for each PTA, because we want the PTAs to have the same state-transition structure.)

During the main loop of the basic RPNI algorithm, at most $|Q_{PTA}|^2$ merge operations are attempted, where Q_{PTA} denotes the set of states in the PTA. Each merge operation (including all additional state merges required to maintain determinism) requires $O(|Q_{PTA}|)$ work. After every merge operation, a compatibility check is performed to determine whether it should be accepted or not, requiring $O(\sum_{w\in S_-} |w|)$ work. Bearing in mind that $|Q_{PTA}|$ is bounded by $\sum_{w\in S_+} |w|$, all this amounts for a total work in the order of $O((\sum_{w\in S_+} |w|)^2 \cdot (\sum_{w\in S_+} |w| + \sum_{w\in S_-} |w|))$.

In the PRPNI algorithm, the basic RPNI loop is repeated N times in sequence, which amounts for a total complexity of $O(\sum_{i=1}^{N} (\sum_{w\in S_{i+}} |w|)^2 \cdot (\sum_{w\in S_{i+}} |w| + \sum_{w\in S_{i-}} |w|))$. In the MooreMI approach, N DFAs are learned in parallel, and the total work done is $O(N \cdot (\sum_{w\in S_+^{all}} |w|)^2 \cdot (\sum_{w\in S_+^{all}} |w| + \sum_{w\in S_-^{all}} |w|))$, where S_-^{all}, similarly to S_+^{all}, denotes the union of the N negative example sets, S_{i-}. Note here that since the sets S_{i+} (resp. S_{i-}) are not disjoint in general, $\sum_{w\in S_+^{all}} |w|$ (resp. $\sum_{w\in S_-^{all}} |w|$) is bounded by $\sum_{i=1}^{N} \sum_{w\in S_{i+}} |w|$ (resp. $\sum_{i=1}^{N} \sum_{w\in S_{i-}} |w|$).

Forming the DFA product to obtain a Moore machine requires $O(N \cdot |Q_{PTA}|)$ work for the PTAP and MooreMI algorithms, but $O(N \cdot \prod_{i=1}^{N} |Q^i_{PTA}|)$ work for the PRPNI approach. Similarly, completing the resulting Moore machine requires $O(|I| \cdot |Q_{PTA}|)$ work for the PTAP and MooreMI algorithms, and $O(|I| \cdot \prod_{i=1}^{N} |Q^i_{PTA}|)$ work for PRPNI, where I is the input alphabet (which can be inferred from the training set).

Note that the above hold in the case we do not apply the final performance optimization. If we do, the terms corresponding to consistency checks ($\sum_{w\in S_{i-}} |w|$, $\sum_{w\in S_-^{all}} |w|$) are removed, and, since the prefix tree acceptors are now built using both positive and negative examples, S_+ and S_+^{all} are replaced by $S_+ \cup S_-$ and $S_+^{all} \cup S_-^{all}$, respectively.

Summarizing the above, 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 $S_{1+}, S_{1-}, \ldots, S_{N+}, S_{N-}$ be the positive and negative example sets obtained by the preprocessing step at the beginning of each algorithm. Let $m_+ = \sum_{i=1}^{N} \sum_{w\in S_{i+}} |w|$, $m_- = \sum_{i=1}^{N} \sum_{w\in S_{i-}} |w|$, and

$k = \sum_{(\rho_I,\rho_O)\in S_{IO}} |\rho_I|^2$. 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 $O((N+|I|)\cdot m_+ + N \cdot m_+^2 \cdot (m_+ + m_-))$ 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. This polynomial complexity does not contradict Gold's NP-hardness result [36], since the problem we solve is not the exact identification problem (see also Sect. 2).

6 Implementation and experiments

All three algorithms presented in Sect. 5.3 have been implemented in the programming language D [70]; the source code is available online at https://github.com/ggiorikas/FSM-learning

6.1 Experimental evaluation overview

The three algorithms were evaluated on a series of experiments. Each experiment consisted of the following steps:

1. a Moore machine (called the *generator* machine) was used to generate training and test traces;
2. the learning algorithms were run on the training traces;
3. the resulting learned machines were tested on the test traces.

Some of the generator machines used in the experiments were randomly generated. Other generator machines were adapted from case studies used in previous works: In the sequel we refer to these as the *benchmark* machines. As training traces we used characteristic samples as well as randomly generated traces. For the randomly generated traces, several settings with increasing total trace length were used, to illustrate how the results of the algorithms improve when more information is provided. All experiments were run on a machine with a 2.6 GHz Intel Core i5 processor and 16 GB of RAM. We did not set a time limit, however, in order to avoid excessive memory consumption (in particular during product computation of PRPNI), in the case of PRPNI we limited the number of states in the learned machine to one million and triggered an out-of-memory error whenever this number was exceeded.

6.2 Random machine generation

We randomly generated several minimal Moore machines of sizes 50 and 150 states, and input and alphabet sizes $|I| = |O| = 25$. The random generation procedure (inspired by the one used in the Abbadingo One DFA learning competition [22]) takes as inputs a random seed, the number of states, and the sizes of the input and output alphabets of the machine. Two intermediate steps are worth mentioning: (1) After assigning a random output to each state, we fix a random permutation of states and assign the i-th output to the i-th state. This ensures that all output symbols appear in the machine. (2) After assigning random destination states to each (state, input symbol) pair, we fix a random permutation of states that begins with the initial state and add transitions with random letters from the i-th to the $(i+1)$-th state. This ensures that all states in the machine are reachable. Finally, a minimization algorithm is employed to minimize the generated machine if necessary.

6.3 Benchmark machines

The benchmark finite-state machines were adapted from case studies in previous works: the Text Editor, JHotDraw and CVS models from [71], and the Elevator Door Controller model from [72]. The original machines were highly incomplete Mealy machines, so we converted them to complete Moore machines before using them for our experiments. By *highly incomplete* we mean that, from each state of a machine, only a few inputs are typically legal, and transitions for the remaining illegal inputs are missing. We converted these Mealy machines to Moore machines by (i) introducing a new initial state to account for the additional initial output exhibited in the behavior of a Moore machine, and (ii) pushing outputs from transitions to states. Note that we did not have to introduce additional states in this second step as, in all machines, all incoming transitions to a given state occur with the same output symbol. We then completed the resulting Moore machines by adding a sink error state as the destination of all missing transitions. We also introduced new output symbols INIT for the initial state and ERROR for the error state.

The Text Editor machine (Fig. 11) describes valid action sequences for a simple text editor, where a new document can be loaded only after the currently loaded document is closed, and where a document can only be saved after it has been edited. The inputs of the machine represent user actions, and the outputs indicate whether the corresponding input sequence is valid or not. All missing transitions lead to the error state and are ommitted to avoid visual clutter (in this and other machine figures).

JHotDraw[4] is a GUI (Graphical User Interface) framework for Java. The machine of Fig. 12 represents (from a user action perspective) the process of adding figures and text boxes in JavaDraw, a sample application included in the JHotDraw distribution.

[4] https://sourceforge.net/projects/jhotdraw/.

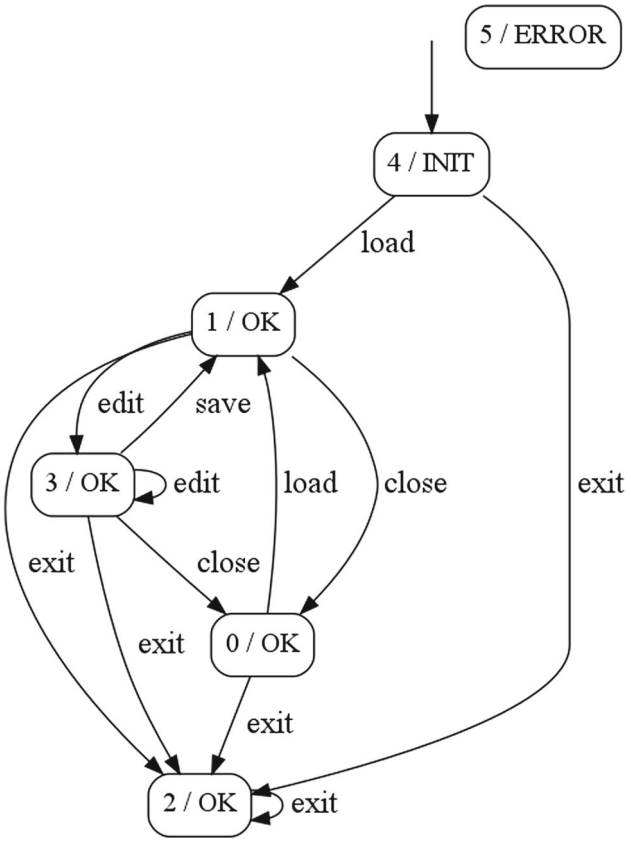


Fig. 11 The text editor machine. The machine is complete. All missing transitions lead to the ERROR state and are not shown for the sake of readability

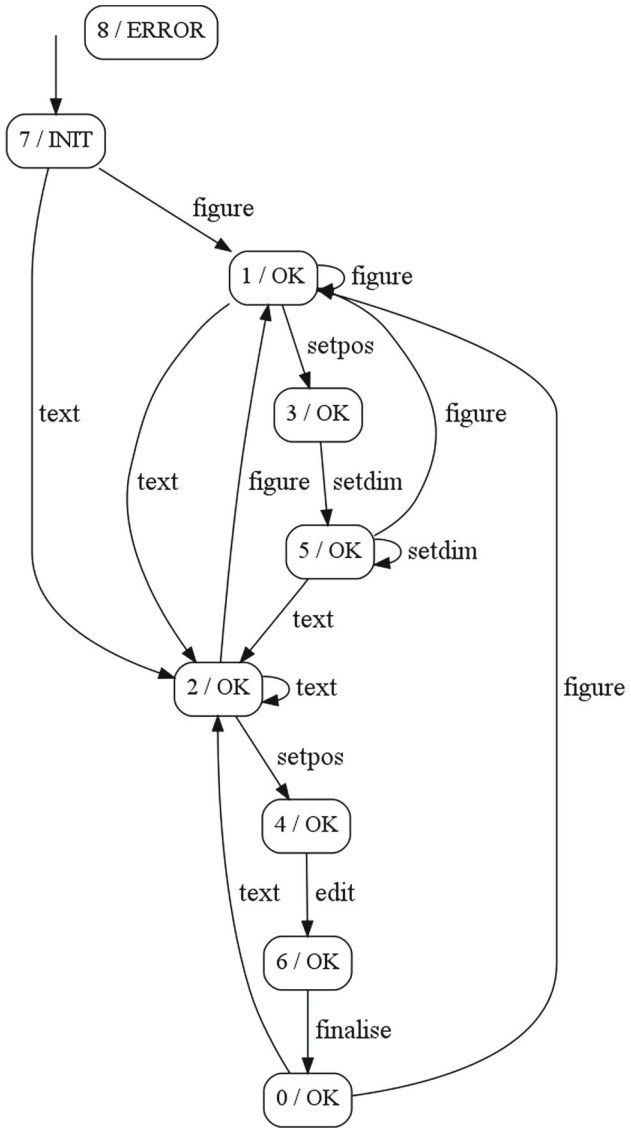


Fig. 12 The JHotDraw/JavaDraw machine

The Jakarta Commons Net project[5] provides a variety of low level network protocol implementations. On top of them, Lo et al. [73] implemented a CVS (Concurrent Versions System) client, an adaptation of which is shown in Fig. 13.

Figure 14 shows a machine modeling an elevator door controller [72]. The input and output symbols correspond to sensor input and controller outputs. Specifically, $e1$1 and $e1$2 indicate "open doors" and "close doors" inputs, $e2$ signals to the controller that the doors were successfully opened, $e3$ that an obstacle prevents the doors from closing, and $e4$ that the doors are jammed. Regarding outputs, $z1$ and $z2$ correspond to opening and closing the doors, and $z3$ is a call to the emergency service.

6.4 Trace generation

The characteristic sample generation procedure has been outlined in Sect. 4.2. In the following, we describe the random trace generation procedures that we used in our experiments.

In the earlier conference version of this paper [18], we chose a naive randomized algorithm for generating both training and test sets. Below, we illustrate a problem with this naive method, and propose a new and improved method.

6.4.1 A naive method: fixed-word-length trace generation

The input to this method is a Moore machine M, a random seed (for reproducibility), the number of input–output traces to be generated, N, as well as the desired length of each of these traces, L. First, N words of length L each are generated, using letters from the input alphabet of M. Then, each of these input words are fed into M to generate the corresponding output words.

This method is simple, but may fail to generate "high-quality traces", as we illustrate with an example. Suppose we provide as input to the naive trace generation algorithm

[5] https://jakarta.apache.org/.

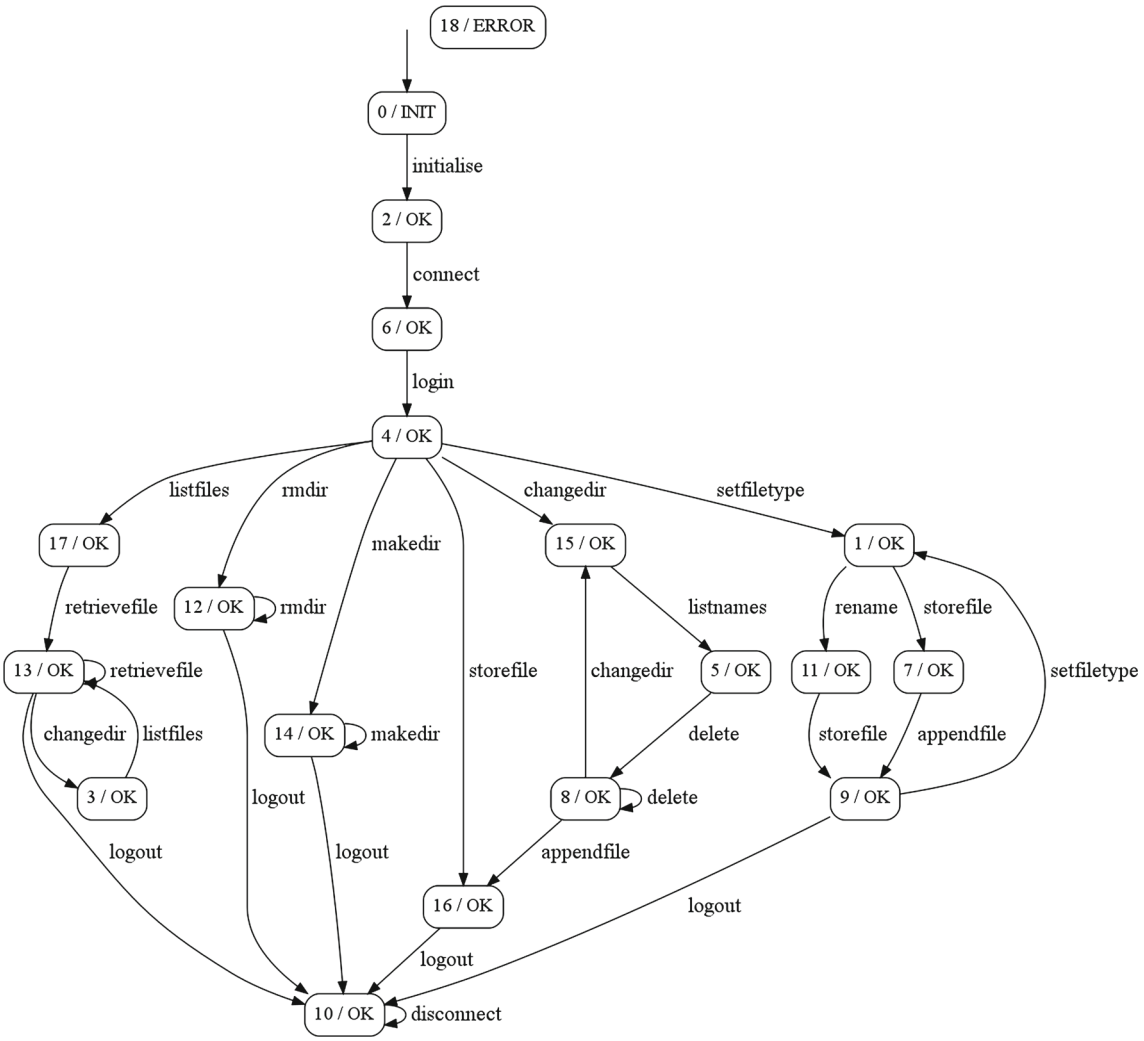


Fig. 13 The CVS machine

$N = 4$, $L = 10$ and a Moore machine with input alphabet $I = \{a,\ b,\ c,\ d\}$. One set of input traces consistent with the requested size is:

$$\{aaaaaaaaaa, aaaaaaaaab, aaaaaaaaac, aaaaaaaaad\}.$$

The total length of the set is 40 characters and represents 14 different *input histories*: $\epsilon = a^0, a = a^1, a^2, \cdots, a^{10}, a^9b$, a^9c, a^9d. Now, consider another set of input traces, also consistent with the requested size:

$$\{aaaaaaaaaa, baaaaaaaab, caaaaaaaac, daaaaaaaad\}.$$

Although this set also has total length 40, it represents 41 different input histories. We can expect that the more input histories a trace set represents the better, as the more likely it is that different input histories cover different states of the machine.

As this example shows, the total length of a set of traces is not a good measure of the "quality" of the set in terms of coverage of input histories. A better measure is the size of the corresponding *input-history tree* that the set represents, in terms of number of nodes in the tree. This tree is very much like the prefix tree acceptor that we have seen already. In the example above, the size of the tree of the first set is 13, whereas of the second set it is 40. The new random trace generation algorithm that we present below generates a tree of a given *size* in the sense above.

An additional problem in our case that renders high-quality trace generation more difficult for the benchmark machines is that, as mentioned above, they are highly incomplete. This means that they only have very few legal inputs at each state, which results in a high percentage of fixed-word-length randomly generated input traces leading to the ERROR state, thus providing virtually no information about the rest of the machine. This issue of naive random trace

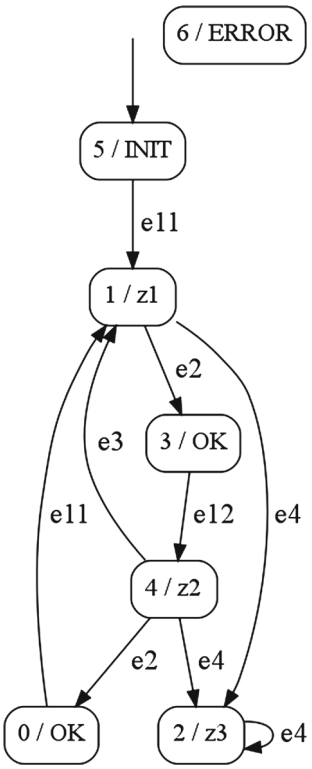


Fig. 14 The elevator door controller machine

generation algorithms being inadequate for realistic models representative of software behavior had also come up in the StaMInA [23] automata learning competition and led to the adoption of a random walk based approach. Drawing inspiration from this, our new trace generation algorithm, described next, also incorporates random walk elements to accommodate for high-quality trace generation for our benchmark machines.

6.4.2 Tree trace generation

The input to the tree trace generation algorithm is a Moore machine M, a random seed, and the desired tree node count, L. The algorithm builds a tree of size L. Every path in the tree from the root to a leaf corresponds to an input trace. As in the fixed-word-length algorithm, the input traces are transduced by M to yield the corresponding output traces.

The algorithm builds the tree by repeatedly performing random walks on M starting from the initial state. The length of each random walk is determined using a normal distribution with a mean of twice the diameter[6] of M and a standard deviation of half the diameter of M. This length is extended (one step at a time) if the generated input trace has appeared before. (This can be checked very efficiently by querying the tree we have built so far.) During each random walk, the probability of selecting a state to be visited next is inversely proportional to the number of times it has been visited so far relative to the rest of the alternatives at that point. After each random walk, the generated input trace is used to expand the tree and once the tree size reaches L the procedure stops.

[6] The diameter of M is the smallest number of transitions needed to reach any state of M starting from the initial state.

6.4.3 Generating training and test sets for experimental evaluation

In all experiments (both for randomly generated and benchmark machines), we used two types of trace sets for *training*: (1) characteristic samples, and (2) trace sets generated by the tree algorithm described above. In the case of traces generated by the tree algorithm, various trace set sizes were considered with the following property: for a given machine and sizes $L1 < L2$, the tree produced for $L1$ is a subtree of the tree produced for $L2$. (This was achieved by using the same seed for the random number generator in the implementation.)

For the trace sets used for *testing*, we used the tree algorithm, both for the randomly generated and for the benchmark machines. In order to be able to (1) measure the ability of the learning algorithms to actually generalize and not simply memorize the training set and (2) meaningfully compare the learning results for different training set sizes, we decided to use one big fixed-size test set for each generator machine. The tree size parameter was fixed to $2 \cdot 10^6$ for the randomly generated machines and $2 \cdot 10^5$ for the benchmark machines.

6.5 Results on randomly generated machines

The results of the experimental evaluation for the randomly generated machines are shown in Tables 1, 2, 3, 4 and 5. Each row represents the average (avg), median (mdn) or standard deviation (sdv) of results for 10 randomly generated Moore machines, for the corresponding algorithm. Dashed entries mean that all 10 corresponding experiments ran out of memory. Note that in non-dashed entries none of the 10 experiments ran out of memory. The caption of each table lists the settings of the corresponding experiment: which training set generation method was used (characteristic sample or tree), average (over 10 Moore machines) training set size ((I, O)-trace count) and average input word length for each case. As described in Sect. 6.4.3, all test sets were generated using the tree algorithm.

As expected, MooreMI generally outperforms the other two algorithms in all metrics, and in some cases significantly so. PTAP's inability to identify in the limit is also reflected in the results (notice that the number of learned states is the same as the tree size parameter used)—the rise in accuracy is merely due to the training set size approaching the test set size.

Table 1 Method: characteristic sample, avg training set size: 1275.8 (50 states), 4451.3 (150 states), avg input word len: 3.4286 (50 states), 3.8929 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.0165	2315.7	0.008	33.832	36.541	0.0544	8199.8	0.009	29.881	32.618
	mdn	0.0138	2324.5	0.01	33.97	36.595	0.0544	8205	0.01	29.955	32.7
	sdv	0.0057	41.3958	0.004	0.5271	0.5045	0.0024	155.204	0.003	0.2996	0.2526
PRPNI	avg	4.2798	82,277.1	0.017	34.311	36.809	–	–	–	–	–
	mdn	4.3025	84,159.5	0.02	34.58	37.045	–	–	–	–	–
	sdv	1.3125	20,882.5	0.0046	0.5819	0.6231	–	–	–	–	–
MooreMI	avg	0.0519	50	100	100	100	0.4536	150	100	100	100
	mdn	0.0511	50	100	100	100	0.4414	150	100	100	100
	sdv	0.0048	0	0	0	0	0.0338	0	0	0	0

Table 2 Method: tree 1000, avg training set size: 140.9 (50 states), 109.0 (150 states), avg input word len: 8.0513 (50 states), 10.0227 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.0059	1000	0.031	25.614	28.785	0.0067	1000	0.04	20.18	23.339
	mdn	0.0058	1000	0.03	25.545	28.765	0.0062	1000	0.04	20.265	23.43
	sdv	0.0008	0	0.003	0.2731	0.3421	0.001	0	0	0.2297	0.276
PRPNI	avg	–	–	–	–	–	–	–	–	–	–
	mdn	–	–	–	–	–	–	–	–	–	–
	sdv	–	–	–	–	–	–	–	–	–	–
MooreMI	avg	0.0218	65.9	0.534	31.938	35.374	0.0277	93.3	0.04	21.158	24.408
	mdn	0.0199	65.5	0.515	31.885	35.42	0.0273	92	0.04	21.24	24.475
	sdv	0.0035	2.8089	0.0684	0.4904	0.408	0.0024	5.1391	0	0.2906	0.3032

Table 3 Method: tree 10,000, avg training set size: 1594.4 (50 states), 1184.7 (150 states), avg input word len: 8.0028 (50 states), 10.0325 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.0752	10,000	0.371	34.737	37.492	0.0688	10,000	0.399	27.547	30.413
	mdn	0.0701	10,000	0.37	34.705	37.49	0.0678	10,000	0.4	27.585	30.41
	sdv	0.0146	0	0.003	0.0986	0.1179	0.0031	0	0.003	0.1116	0.1341
PRPNI	avg	–	–	–	–	–	–	–	–	–	–
	mdn	–	–	–	–	–	–	–	–	–	–
	sdv	–	–	–	–	–	–	–	–	–	–
MooreMI	avg	0.1911	125.5	51.989	79.065	80.207	1.1478	354.2	0.489	31.123	34.16
	mdn	0.1825	126	52.95	79.635	80.71	1.1425	352	0.49	31.145	34.16
	sdv	0.0443	13.025	9.1848	4.5481	4.2777	0.051	5.2498	0.0094	0.304	0.311

Table 4 Method: tree 100,000, avg training set size: 18,104.9 (50 states), 13,019.5 (150 states), avg input word len: 8.0061 (50 states), 10.0076 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.8065	100,000	4.131	45.378	47.605	0.7858	100,000	4.366	36.522	39.03
	mdn	0.755	100,000	4.13	45.385	47.64	0.7801	100,000	4.36	36.555	39.01
	sdv	0.1354	0	0.0104	0.0935	0.1763	0.0342	0	0.0162	0.1211	0.1621
PRPNI	avg	3.5585	24,651.7	98.637	99.562	99.683	–	–	–	–	–
	mdn	2.2394	3073	98.88	99.66	99.745	–	–	–	–	–
	sdv	3.9425	68,215.5	1.4605	0.4823	0.3457	–	–	–	–	–
MooreMI	avg	0.3631	50	100	100	100	1.1815	220.4	95.923	98.439	98.508
	mdn	0.3622	50	100	100	100	1.0768	223.5	95.84	98.4	98.47
	sdv	0.0144	0	0	0	0	0.3627	34.1532	2.0841	0.7941	0.76

Table 5 Method: tree 1,000,000, avg training set size: 210,700.0 (50 states), 144,881.0 (150 states), avg input word len: 8.0059 (50 states), 9.9993 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	10.2782	1,000,000	47.558	74.448	75.448	10.9528	1,000,000	48.463	69.195	70.392
	mdn	9.9208	1,000,000	47.55	74.445	75.44	10.7495	1,000,000	48.46	69.195	70.4
	sdv	1.8331	0	0.0352	0.0655	0.0953	2.4395	0	0.0215	0.0385	0.0673
PRPNI	avg	27.8298	50	100	100	100	30.8077	11,420	99.941	99.98	99.987
	mdn	27.5391	50	100	100	100	29.7683	150	100	100	100
	sdv	3.3386	0	0	0	0	3.819	13,846	0.0779	0.0261	0.0168
MooreMI	avg	3.5939	50	100	100	100	4.2064	150	100	100	100
	mdn	3.5039	50	100	100	100	4.1011	150	100	100	100
	sdv	0.2197	0	0	0	0	0.2373	0	0	0	0

An interesting observation is that, while it runs out of memory quite often, PRPNI manages to yield good results when enough information is provided. There is a simple explanation for this. First, in all cases where PRPNI ran out of memory the reason was state explosion during product computation of the learned DFAs. When the training set size is small, the individually learned DFAs are very likely to have different state-transition structure, which leads to state explosion during product computation, apparent in the number of states learned by PRPNI in Table 1. However, since the training set is small enough there, the product is also small enough for the computation not to run out of memory. When the training set size is big enough for all individually learned DFA to have the same state-transition structure, product computation is as fast as in the other two algorithms (no state explosion). This behavior is apparent in Table 5. For intermediate training set sizes, product computation is too resource demanding and the algorithm runs out of memory (e.g., Tables 2, 3, 4 for the 150 states machines).

6.6 Results on benchmark machines

The experimental results for these machines are summarized in Tables 7, 8, 9, 10 and 11 and Figs. 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 and 27. Table 11 summarizes the results of the experiments in which characteristic samples were used as the training set. The results of the experiments where the tree algorithm was used for training set generation are summarized in Tables 7, 8, 9 and 10 and Figs. 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 and 27. There is one table and three figures for each benchmark machine. Each row in a table represents results (running times and learned states for all three algorithms) on a specific training set. The *Size* column shows the size parameter given as input to the tree algorithm for the training set generation. The *Time* and *States* columns show, respectively, running times and numbers of learned states for all three algorithms (*A1* corresponds to PTAP, *A2* to PRPNI and *A3* to MooreMI).

Table 6 Performance comparison results with existing tools that learn Mealy machines

Tool	Time (s)			Peak memory usage (GB)
	Parsing	Learning	Total	
LearnLib	3.851	7.143	11.994	1.8
flexfringe	13.806	181.032	194.838	2.8
MealyMI	3.062	2.891	5.953	1.1

Table 7 Text editor (Figs. 16, 17, 18)

Size	Time (s)			States		
	A1	A2	A3	A1	A2	A3
100	0.000394	0.000615	0.000366	100	13	5
200	0.000662	0.001182	0.000594	200	22	6
300	0.000997	0.001412	0.001057	300	22	6
400	0.001236	0.001895	0.000999	400	20	6
500	0.002728	0.002804	0.001486	500	20	6

Table 8 JHotDraw (Figs. 19, 20, 21)

Size	Time (s)			States		
	A1	A2	A3	A1	A2	A3
250	0.001125	0.001856	0.000554	250	27	7
500	0.001700	0.004272	0.002054	500	37	8
750	0.003153	0.006055	0.002313	750	47	8
1000	0.003409	0.006001	0.002893	1000	52	9

Table 9 Elevator door controller (Figs. 22, 23, 24)

Size	Time (s)			States		
	A1	A2	A3	A1	A2	A3
200	0.000701	0.002247	0.000526	200	53	7
400	0.001390	0.003930	0.000720	400	55	7
600	0.001970	0.003279	0.001080	600	37	7
800	0.002760	0.004421	0.001331	800	38	7

Table 10 CVS (Figs. 25, 26, 27)

Size	Time (s)			States		
	A1	A2	A3	A1	A2	A3
20,000	0.125104	0.137925	0.054600	20,000	114	20
40,000	0.321886	0.536724	0.115188	40,000	92	23
60,000	0.394782	1.009475	0.172411	60,000	106	19
80,000	0.535994	0.563334	0.237576	80,000	90	19

Table 11 Results for benchmark machines where characteristic samples are used as training sets

FSM	Time (s)	States	Accuracy (%)		
			Strong	Medium	Weak
PTAP					
Editor	0.000214	47	53.97	81.58	87.59
JHotDraw	0.000417	98	57.59	84.62	86.63
Elevator	0.000197	36	43.32	66.15	67.81
CVS	0.002326	400	49.07	85.79	87.04
PRPNI					
Editor	0.000487	11	29.39	74.33	86.13
JHotDraw	0.001061	17	48.42	78.97	89.35
Elevator	0.000982	27	5.26	49.02	73.10
CVS	0.035927	37	72.93	89.73	97.81
MooreMI					
Editor	0.000260	6	100	100	100
JHotDraw	0.000643	9	100	100	100
Elevator	0.000225	7	100	100	100
CVS	0.005695	19	100	100	100

The three figures for each benchmark machine show results (accuracy scores) obtained by the three different algorithms. Each figure contains three plots representing values for accuracy metrics measured on machines learned with training sets of various sizes. Solid lines correspond to the strong accuracy metric, dashed lines to the medium, and dotted lines to the weak one.

For each benchmark machine, we kept increasing the training set size until MooreMI learned the correct machine. PTAP never learns the correct machine, which is expected, since it does not identify in the limit. Neither does PRPNI ever learn the correct machine, which serves as an additional indication that even if it identifies in the limit the characteristic sample it requires is generally larger than the one defined in this paper.

The general conclusion from all experiment results (on both random and benchmark machines) is that MooreMI is clearly the best option of the three algorithms, as, in contrast to PTAP, it identifies in the limit and never exhibits the state explosion problems encountered with PRPNI.

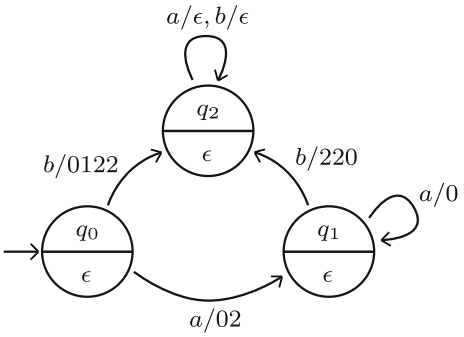


Fig. 15 The transducer learned by OSTIA given a characteristic sample for the Moore machine in Fig. 5a as input

7 Performance comparison with existing tools

In this section, we compare our best algorithm against similar existing implementations. Specifically, we compare against LearnLib [15] and flexfringe [16], both of which provide (among other things) passive learning algorithms for learning Mealy machines.

7.1 MealyMI

The similarity of Moore and Mealy machines naturally raises the question to what extent methods to learn Moore machines can be used to learn Mealy machines (and vice versa). A thorough study of this question is beyond the scope of the current paper. Nevertheless, in order to be able to compare our results with LearnLib [15] and flexfringe [16], we adapted our best algorithm, MooreMI, so that it can also learn Mealy machines; we call the resulting algorithm MealyMI. The adaptation consists of several adjustments: (i) We modified the core data structures representing Moore machines and (Moore-style) prefix tree acceptors to associate outputs with transitions instead of states and also got rid of the initial output, (ii) we modified the random machine generation, trace generation and input parsing algorithms accordingly to correctly handle these new representations, and last but not

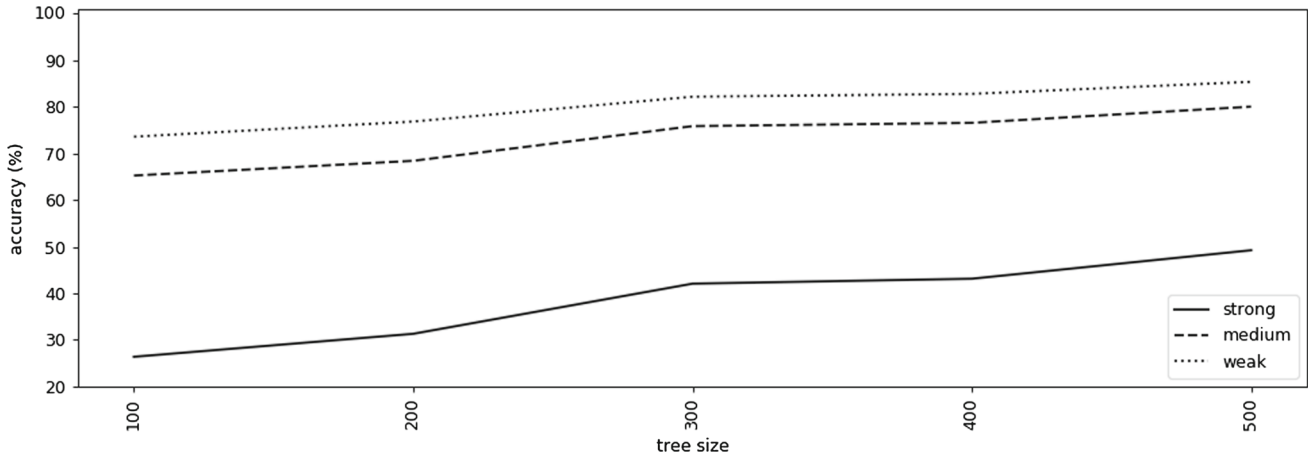


Fig. 16 Text editor (Table 7)—PTAP

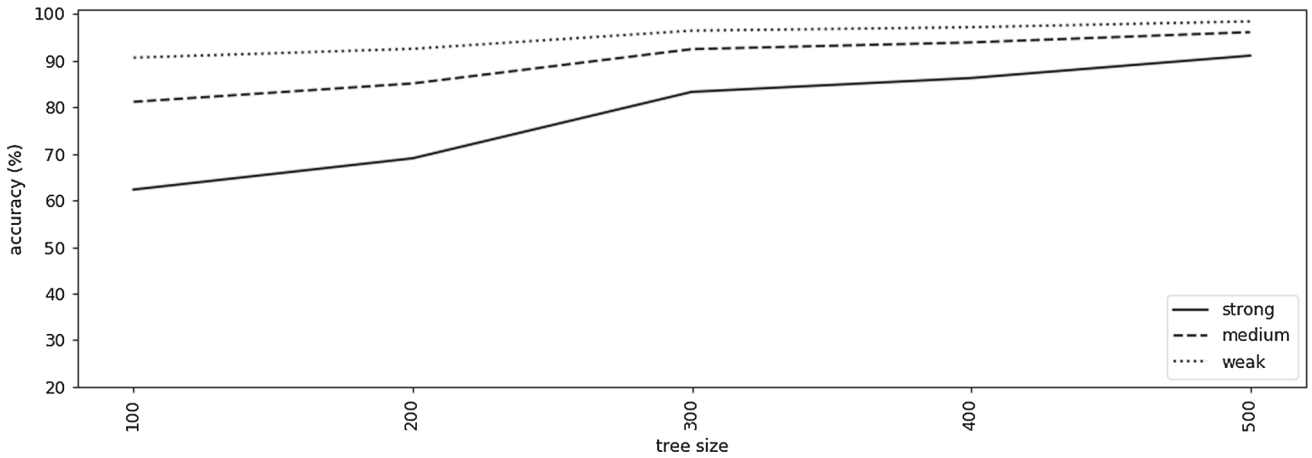


Fig. 17 Text editor (Table 7)—PRPNI

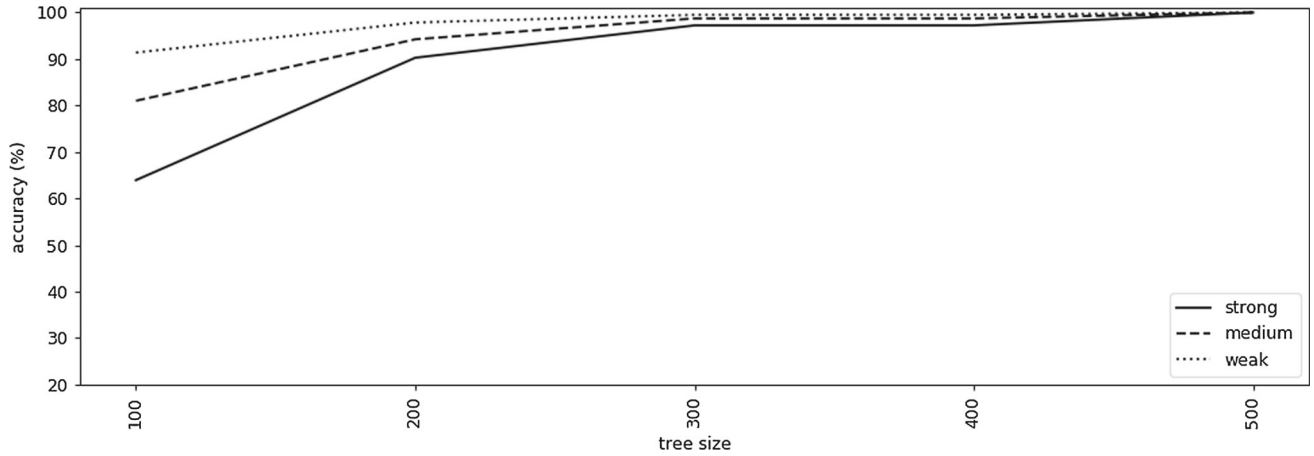


Fig. 18 Text editor (Table 7)—MooreMI

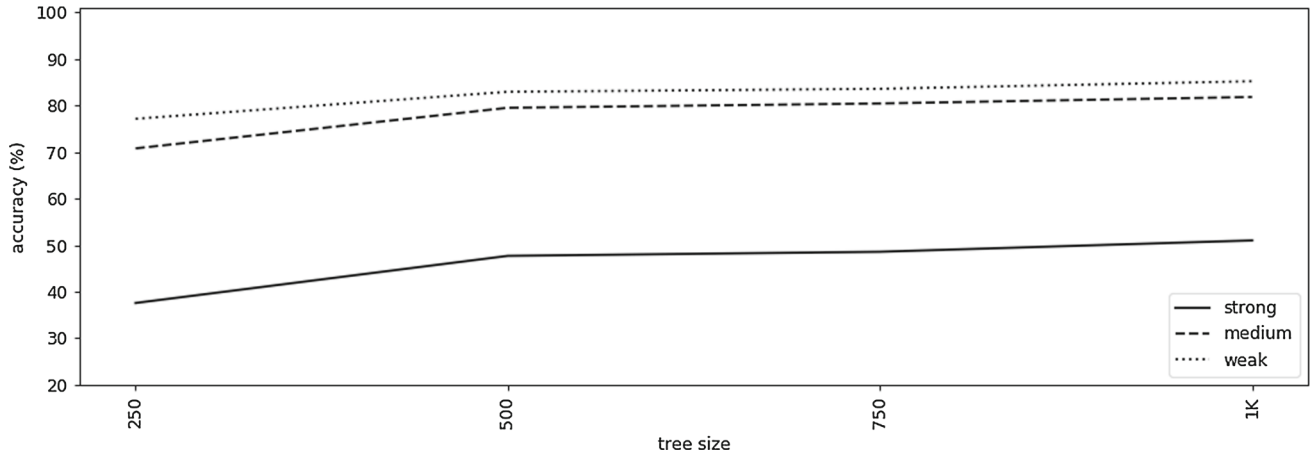


Fig. 19 JHotDraw (Table 8)—PTAP

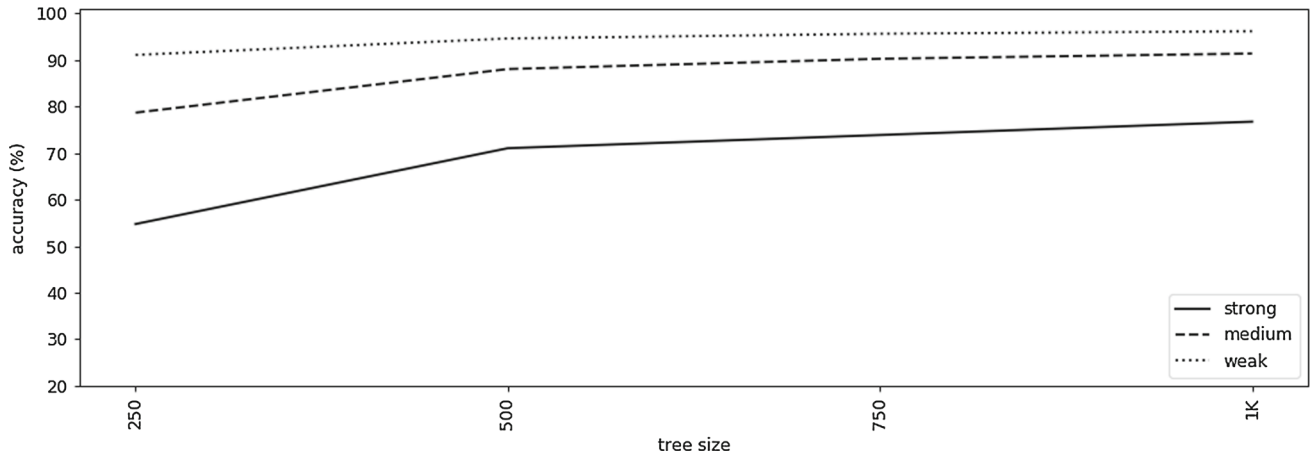


Fig. 20 JHotDraw (Table 8)—PRPNI

least, (iii) we correspondingly modified the learning algorithm itself (all phases—prefix tree building, state merging and completion); for example, the consistency check between two states in MealyMI examines whether the corresponding outputs for each input symbol are equal, while in MooreMI it simply examines whether the two states have the same output.

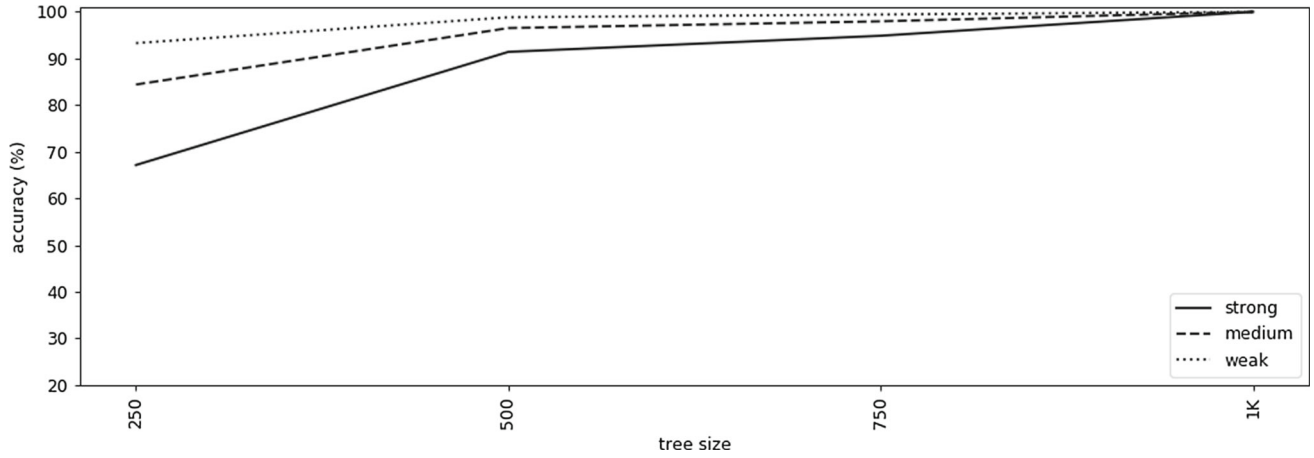


Fig. 21 JHotDraw (Table 8)—MooreMI

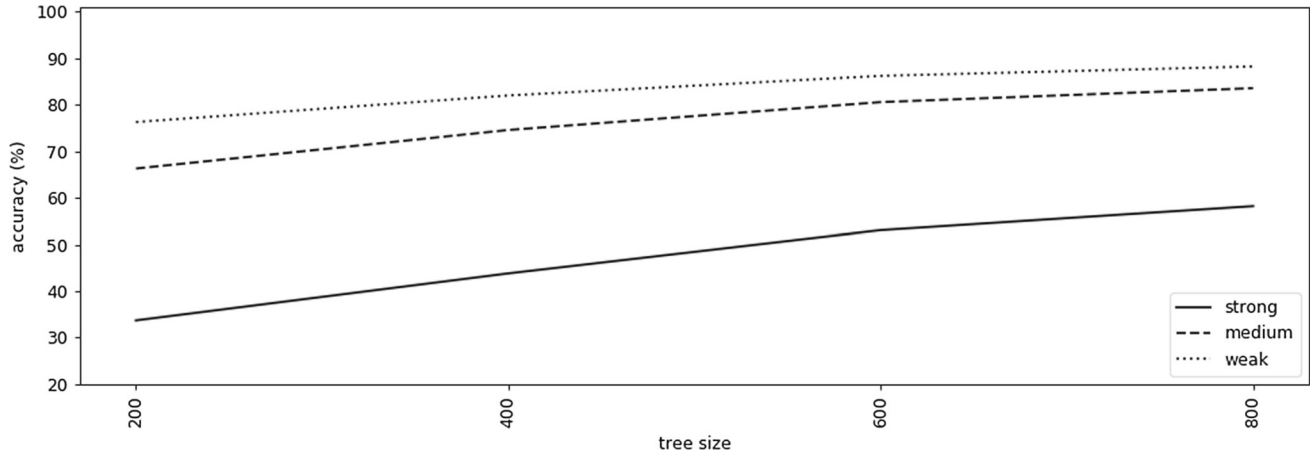


Fig. 22 Elevator door controller (Table 9)—PTAP

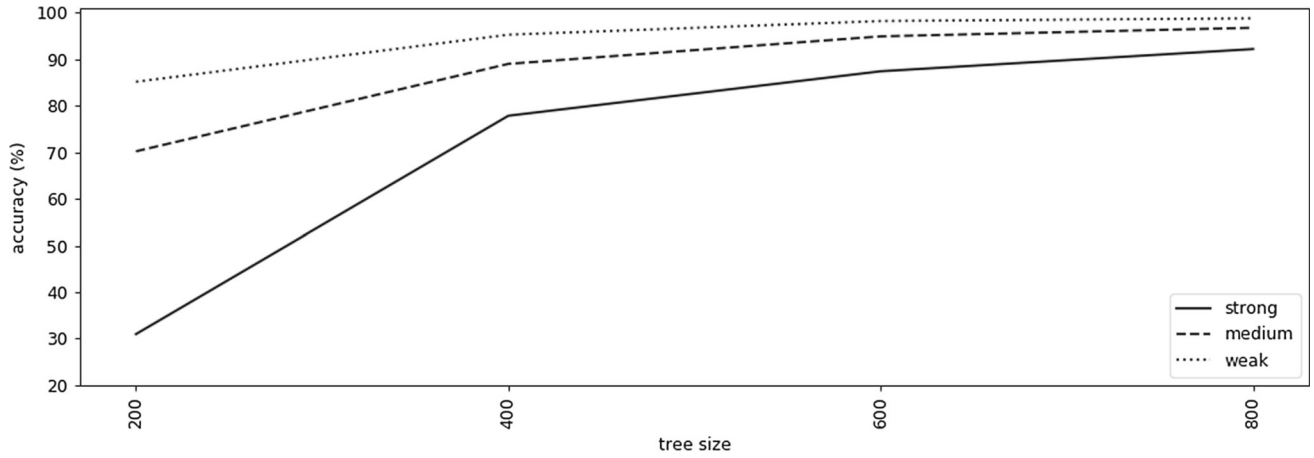


Fig. 23 Elevator door controller (Table 9)—PRPNI

Results of MealyMI are presented in Sect. 7.2 that follows. A formal analysis of MealyMI is left for future work. Future work also includes investigation of alternative ways to learn Mealy machines, e.g., by using MooreMI as a black box. This could be done, for example, by encoding a Mealy machine as a DFA over the cartesian product of the Mealy machine's input and output alphabets. However, such transformations are likely to (i) lead to suboptimal approaches w.r.t perfor-

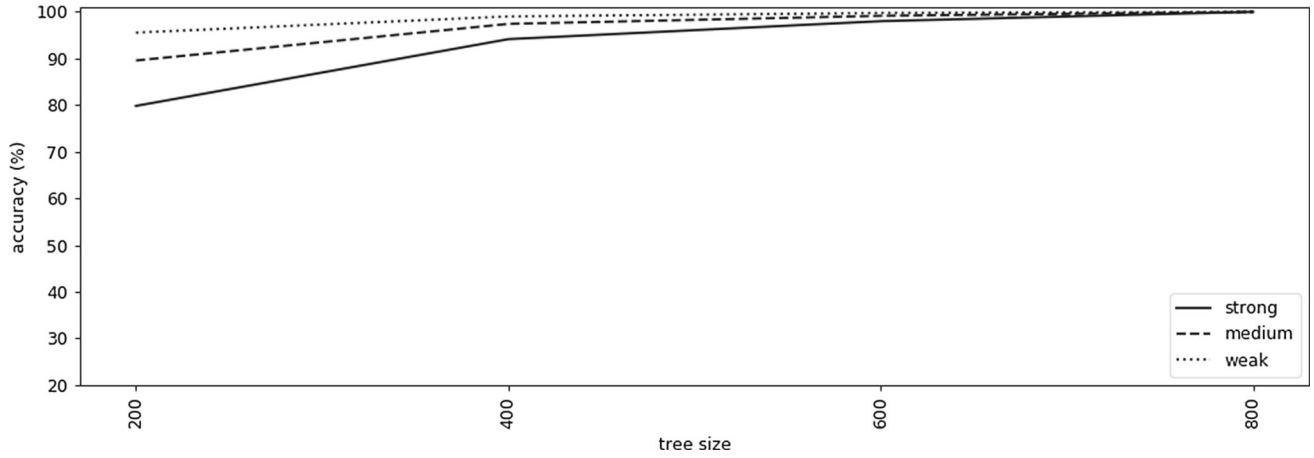


Fig. 24 Elevator door controller (Table 9)—MooreMI

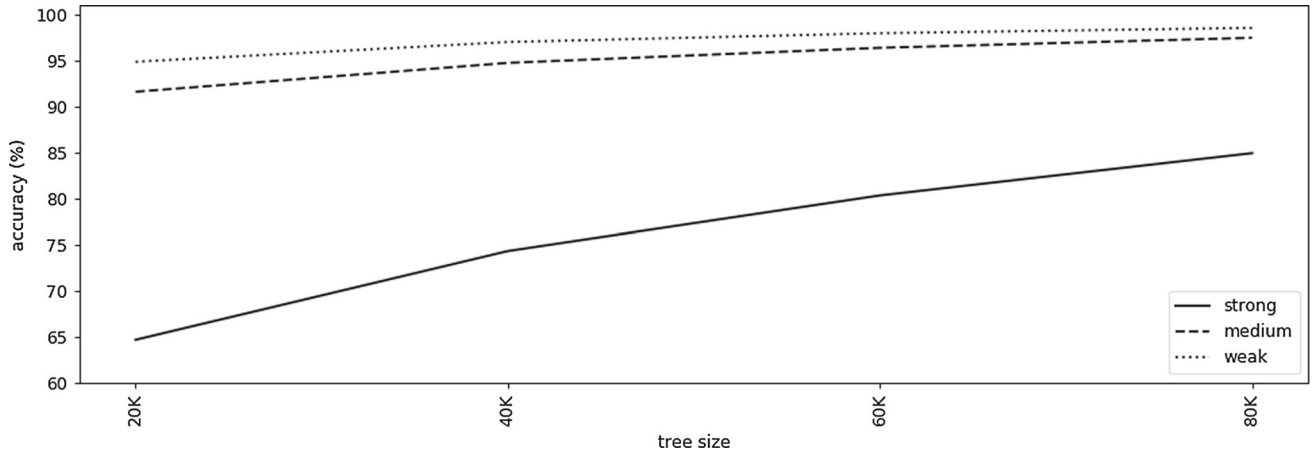


Fig. 25 CVS (Table 10)—PTAP

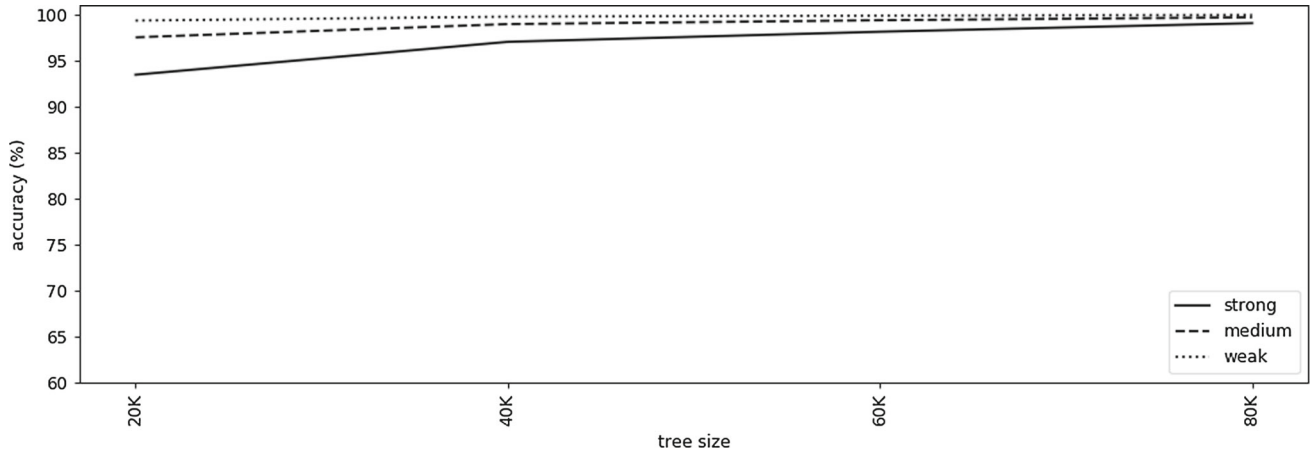


Fig. 26 CVS (Table 10)—PRPNI

mance of the core learning loop and (ii) potentially require additional pre-/post-processing steps of appropriately encoding the given (Mealy) input traces and translating the learned Moore machine back into a Mealy machine. Future work includes investigation of the opposite direction as well: to what extent can Mealy machine learning algorithms and tools be used as black boxes for learning learn Moore machines. Again, we believe that this black box approach is not appro-

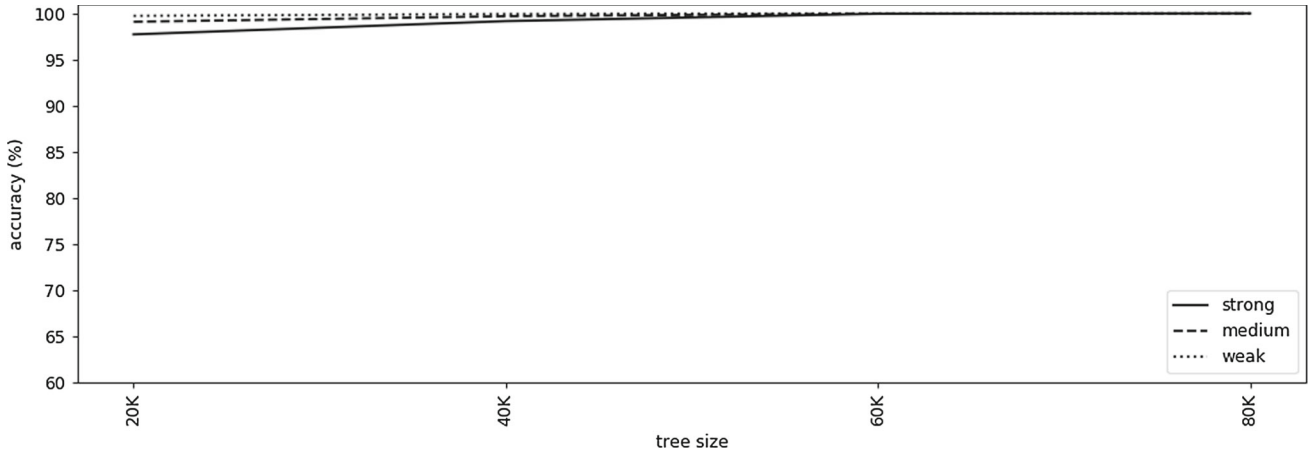


Fig. 27 CVS (Table 10)—MooreMI

priate, for a number of reasons: first, the input–output traces for a Moore machine are not directly compatible with Mealy machines, and therefore need to be transformed somehow; second, the learned Mealy machine must also be transformed into a Moore machine. The exact form of such transformations and their correctness remain to be demonstrated. Such transformations may also incur performance penalties which make a learning method designed specifically for Moore machines more attractive in practice.

7.2 Comparison approach and results

Since all tools we compare here implement essentially the same approach from a theoretical point of view (application of the red-blue state merging framework on Mealy machine learning—note that while flexfringe also offers exact identification functionality through a SAT solver we do not make use of this here), we only focused on measuring implementation efficiency (i.e., running time and memory consumption).

To do this, we evaluated the three approaches on learning a randomly generated Mealy machine with 30 states, 10 input symbols and 20 output symbols, using a set of 400K (I, O)-traces (more than enough for the correct machine to be identified). Both the machine and the traces used were generated using adaptations of our respective Moore machine and trace generation algorithms (outlined in Sect. 7.1).

We have separated each algorithm into input parsing and learning phases and report results for these individually, as the input parsing phase for LearnLib was implemented by us (the library does not provide such functionality), and it would not be fair to take it into account in our comparison (even though we strived for an efficient implementation there as well).

As the input format for the traces we decided to use the one proposed by flexfringe, so that we do not have to modify flexfringe in this respect. However, note that we did perform some trivial performance optimizations on flexfringe for our comparison (e.g., commented out code that prints to standard output and error streams during learning, commented out printing of the initial PTA to a file, compiled all files with -O3, the maximum level of speed related compiler optimizations etc.)

A summary of the results is reported in Table 6. Note that in all cases the same (correct) machine was learned. The results show clearly that MealyMI outperforms both existing implementations by a large margin. Specifically, our learning core is more than two times faster than the one in LearnLib and more than an order of magnitude faster than the one in flexfringe. Peak memory usage is also better in our case.

These results are not really surprising. Regarding LearnLib, since it is implemented in Java, a 2x-3x difference from a natively compiled language like D is expected. Regarding flexfringe, one should keep in mind that the implementation was written with genericity and extensibility in mind; e.g., it is able to learn not only Mealy machines, but also, depending on the options provided, DFAs with different merge heuristics (not just RPNI), probabilistic automata, etc. In order to accommodate for these goals a sacrifice in efficiency is not something unexpected. For example, since some of the merge heuristics require computation of scores for all available red-blue state pairs (for a given blue state) before selecting which one to merge, this is hardcoded behavior in flexfringe and happens even when options that induce RPNI-like behavior (e.g., `shallowfirst=1`) are used (in which case it would suffice to only check the pairs until the first compatible merge is found, as LearnLib and MealyMI do).

8 Comparison with OSTIA

OSTIA [17] is a well-known algorithm that learns *onward subsequential transducers*, a class of transducers more gen-

eral 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 Fig. 5a and ran the OSTIA algorithm on it. (We used the open source implementation described in [74].) The resulting machine is depicted in Fig. 15. 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, as is the case with PRPNI, in general OSTIA needs more information than MooreMI to correctly identify a Moore machine from example traces. In particular, OSTIA cannot learn a Moore machine, even when the training set is a characteristic sample as defined in this paper (Sect. 4.1). This is not unexpected; since OSTIA is tailored to learn a more general transducer variant, enforcing the constraint of one output symbol per input symbol on transitions can only be achieved by incorporating additional appropriate example traces in the training set.

9 Conclusion and future work

We formalized the problem of learning Moore machines for input–output traces and developed three algorithms to solve this problem. We showed that the most advanced of these algorithms, MooreMI, has desirable theoretical properties: In particular it satisfies the characteristic sample requirement and achieves identification in the limit. We also compared the algorithms experimentally and showed that MooreMI is also superior in practice.

Future work includes: (1) extending the study of inference for Mealy and other types of state machines; although there exist several tools that learn machines of type Mealy and others, e.g., [15,16,44], to our knowledge, the theoretical background (e.g., characteristic sample definition, identification in the limit, etc.) is still missing from this work; (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. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
2. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
3. Tripakis, S.: Data-driven and model-based design. In: 1st IEEE International Conference on Industrial Cyber-Physical Systems (ICPS) (2018)
4. Ljung, L.: System Identification: Theory for the User, 2nd edn. Prentice Hall, Upper Saddle River (1999)
5. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)
6. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: 38th POPL, pp. 317–330 (2011)
7. Seshia, S.A.: Sciduction: combining induction, deduction, and structure for verification and synthesis. In: DAC, pp. 356–365 (2012)
8. Ray, B., Posnett, D., Filkov, V., Devanbu, P.: A large scale study of programming languages and code quality in github. In: ACM SIGSOFT, FSE'14 (2014)
9. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: HVC, Volume 8855 of LNCS. Springer (2014)
10. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (2017)
11. Zeller, A.: Why Programs Fail—A Guide to Systematic Debugging, 2nd edn. Academic Press, Cambridge (2009)
12. Kohavi, Z.: Switching and Finite Automata Theory, 2nd edn. McGraw-Hill, New York (1978)
13. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. CUP, Cambridge (2010)
14. Gold, E.M.: Language identification in the limit. Inf. Control **10**(5), 447–474 (1967)
15. Raffelt, H., Steffen, B.: Learnlib: a library for automata learning and experimentation, vol. 3922, pp. 377–380 (2006)
16. Verwer, S., Hammerschmidt, C.: flexfringe: a passive automaton learning package, pp. 638–642 (2017)
17. Oncina, J., García, P., Vidal, E.: Learning subsequential transducers for pattern recognition interpretation tasks. IEEE Trans. Pattern Anal. Mach. Intell. **15**(5), 448–458 (1993)
18. Giantamidis, G., Tripakis, S.: Learning Moore machines from input–output traces. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) 21st International Symposium on Formal Methods (FM 2016), Volume 9995 of LNCS, pp. 291–309 (2016)
19. Mens, I.-E., Maler, O.: Learning regular languages over large ordered alphabets. Log. Methods Comput. Sci. **11**(3) (2015). https://doi.org/10.2168/LMCS-11(3:13)2015
20. Argyros, G., Stais, I., Kiayias, A., Keromytis, A.D.: Back in black: towards formal, black box analysis of sanitizers and filters. In: IEEE Symposium on Security and Privacy, SP 2016, pp. 91–109 (2016)
21. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Tools and Algorithms for the Construction and Analysis of Systems—23rd International Conference, TACAS 2017, volume 10205 of LNCS, pp. 173–189 (2017)
22. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In: Honavar, V., Slutzki, G. (eds.) Grammatical Inference. Springer, Berlin (1998)
23. Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., Dupont, P.: Stamina: a competition to encourage the development and assessment of software model inference techniques. Empir. Softw. Eng. **18**(4), 791–824 (2013)
24. Verwer, S., Eyraud, R., Higuera, C.: Pautomac: a probabilistic automata and hidden markov models learning competition. Mach. Learn. **96**(1), 129–154 (2014)
25. Jasper, M., Mues, M., Murtovi, A., Schlüter, M., Howar, F., Steffen, B., Schordan, M., Hendriks, D., Schiffelers, R., Kuppens, H., Vaandrager, F.W.: Rers 2019: combining synthesis with real-world models. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 101–115. Springer International Publishing, Cham (2019)

26. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies, number 34. Princeton University Press (1956)
27. Gill, A.: State-identification experiments in finite automata. Inf. Control **4**, 132–154 (1961)
28. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
29. Shahbaz, M., Groz, R.: Inferring mealy machines. In: FM 2009, pp. 207–222 (2009)
30. Jonsson, B.: Learning of automata models extended with data. In: SFM 2011, Advanced Lectures, pp. 327–349 (2011)
31. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Learning extended finite state machines. In: SEFM 2014, Proceedings, pp. 250–264 (2014)
32. Aarts, F., Vaandrager, F.: Learning I/O automata. In: CONCUR. Springer, pp. 71–85 (2010)
33. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: VMCAI 2012, Proceedings, pp. 251–266 (2012)
34. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.W.: Learning register automata with fresh value generation. In: Theoretical Aspects of Computing—ICTAC, volume 9399 of LNCS, pp. 165–183 (2015)
35. Medhat, R., Ramesh, S., Bonakdarpour, B., Fischmeister, S.: A framework for mining hybrid automata from input/output traces. In: Embedded Software (EMSOFT), pp. 177–186 (2015)
36. Gold, E.M.: Complexity of automaton identification from given data. Inf. Control **37**(3), 302–320 (1978)
37. Heule, M.J., Verwer, S.: Software model synthesis using satisfiability solvers. Empir. Softw. Eng. **18**(4), 825–856 (2013)
38. Ulyantsev, V., Zakirzyanov, I., Shalyto, A.: BFS-based symmetry breaking predicates for DFA identification. In: Language and Automata Theory and Applications (LATA), volume 8977 of LNCS. Springer, pp. 611–622 (2015)
39. Oncina, J., Garcia, P.: Identifying regular languages in polynomial time. In: Advances in Structural and Syntactic Pattern Recognition, pp. 99–108 (1992)
40. Dupont, P.: Incremental regular inference. In: ICGI-96, pp. 222–237 (1996)
41. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: ICGI-98, pp. 1–12 (1998)
42. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A.: Inferring models of concurrent systems from logs of their behavior with csight. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York, NY, USA, pp. 468–479 (2014)
43. Verwer, S., de Weerdt, M., Witteveen, C.: A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: Sempere, J.M., García, P. (eds.) Grammatical Inference: Theoretical Results and Applications, pp. 203–216. Springer, Berlin (2010)
44. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empir. Softw. Eng. **21**(3), 811–853 (2016). https://doi.org/10.1007/s10664-015-9367-7
45. Spichakova, M.: An approach to inference of finite state machines based on gravitationally-inspired search algorithm. Proc. Estonian Acad. Sci. **62**(1), 39–46 (2013)
46. Aleksandrov, A.V., Kazakov, S.V., Sergushichev, A.A., Tsarev, F.N., Shalyto, A.A.: 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)
47. Buzhinsky, I.P., Ulyantsev, V.I., Chivilikhin, D.S., Shalyto, A.A.: Inducing finite state machines from training samples using ant colony optimization. J. Comput. Sys. Sc. Int. **53**(2), 256–266 (2014)
48. Meinke, K.: CGE: a sequential learning algorithm for mealy automata. In: Sempere, J.M., García, P. (eds.) Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13–16, 2010. Proceedings, volume 6339 of LNCS. Springer, pp. 148–162 (2010)
49. Veelenturf, L.P.J.: Inference of sequential machines from sample computations. IEEE Trans. Comput. **27**(2), 167–170 (1978)
50. Takahashi, K., Fujiyoshi, A., Kasai, T.: A polynomial time algorithm to infer sequential machines. Syst. Comput. Jpn. **34**(1), 59–67 (2003)
51. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **21**(6), 592–597 (1972)
52. Karthik, A.V., Ray, S., Nuzzo, P., Mishchenko, A., Brayton, R., Roychowdhury, J.: ABCD-NL: approximating continuous non-linear dynamical systems using purely Boolean models for analog/mixed-signal verification. In: ASP-DAC, pp. 250–255 (2014)
53. Grinchtein, O., Leucker, M.: Learning finite-state machines from inexperienced teachers. In: ICGI, pp. 344–345 (2006)
54. Leucker, M., Neider, D.: Learning minimal deterministic automata from inexperienced teachers. In: ISoLA, pp. 524–538 (2012)
55. Heitmeyer, C.L., Pickett, M., Leonard, E.I., Archer, M.M., Ray, I., Aha, D.W., Trafton, J.G.: Building high assurance human-centric decision systems. Autom. Softw. Eng. **22**(2), 159–197 (2015)
56. Ulyantsev, V., Buzhinsky, I., Shalyto, A.: Exact finite-state machine identification from scenarios and temporal properties. STTT **20**(1), 35–55 (2018)
57. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI'08. ACM, pp. 281–292 (2008)
58. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: CAV. Springer, pp. 420–432 (2003)
59. Gupta, A., Rybalchenko, A.: Invgen: an efficient invariant generator. In: Computer Aided Verification, CAV. Springer, pp. 634–640 (2009)
60. Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C., Latronico, E.: Automatic requirement extraction from test cases. In: Runtime Verification, RV'10 (2010)
61. Jin, X., Donz, A., Deshmukh, J.V., Seshia, S.A.: Mining requirements from closed-loop control models. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(11), 1704–1717 (2015)
62. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: Automated Software Engineering (ASE), pp. 81–92 (2015)
63. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL'02. ACM, pp. 4–16 (2002)
64. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proc. IEEE **84**(8), 1090–1123 (1996)
65. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
66. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N.: Fsm-based conformance testing methods: a survey annotated with experimental evaluation. Inf. Softw. Technol. **52**(12), 1286–1297 (2010)
67. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: FASE, volume 3442 of LNCS. Springer, pp. 175–189 (2005)
68. Sorower, M.S.: A literature survey on algorithms for multi-label learning. Technical report (2010)
69. Coste, F., Nicolas, J.: ICGI-98, chapter How considering incompatible state mergings may reduce the DFA induction search tree. Springer, pp. 199–210 (1998)
70. The D Programming Language. https://dlang.org/

71. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: ASE, pp. 248–257 (2008)
72. Tsarev, F., Egorov, K.: Finite state machine induction using genetic algorithm based on testing and model checking. In: 13th Annual Genetic and Evolutionary Computation Conference, GECCO, pp. 759–762 (2011)
73. Lo, D., Khoo, S.-C.: Smartic: towards building an accurate, robust and scalable specification miner. In: FSE. ACM, New York, NY, USA, pp. 265–275 (2006)
74. Akram, H.I., de la Higuera, C., Xiao, H., Eckert, C.: Grammatical inference algorithms in matlab. In: ICGI'10. Springer, pp. 262–266 (2010)