Towards Efficient Active Learning of PDFA[*]

F. Mayr S. Yovine F. Pan
Facultad de Ingeniería
Universidad ORT Uruguay
Montevideo, Uruguay
mayr@ort.edu.uy
yovine@ort.edu.uy

N. Basset T. Dang
Verimag
Université Grenoble-Alpes
Grenoble, France
nicolas.basset1@univ-grenoble-alpes.fr
thao.dang@univ-grenoble-alpes.fr

June 17, 2022

Abstract

We propose a new active learning algorithm for PDFA based on three main aspects: a congruence over states which takes into account next-symbol probability distributions, a quantization that copes with differences in distributions, and an efficient tree-based data structure. Experiments showed significant performance gains with respect to reference implementations.

Keywords Active learning · PDFA · Quantization

1 Introduction

We are interested in the problem of efficiently learning probabilistic deterministic finite automata (PDFA) in the context of the general active MAT-learning framework proposed in [1] where a *learner* and a so-called *minimum adequate teacher* interact by asking and responding questions, respectively. The learner's purpose is to unveil the hidden target automaton only known to the teacher. The latter allows the former to ask two kinds of questions, namely *membership* (**MQ**) and *equivalence* (**EQ**) queries.

MQ seeks to discover the automaton's outcome for a particular string. Originally, the purpose of **MQ** is to know whether a string belongs to the language to be learnt or, equivalently, whether it is accepted by the associated automaton. This name has been kept in later applications to other classes of automata. In [2,3], the framework is used to prove learnability of multiplicity automata which compute functions mapping strings to elements of a *field*. In this case, the name **MQ** is retained even when the answer is the value of the computed function, which may not be Boolean. The same approach is followed in [15] where **MQ** is used to refer to the function that returns the probability of the last symbol in a string.

EQ provides means for determining to what extent the hypothesis automaton produced by the learner approximates the target one. In the case of *exact* learning, this query checks whether they both produce the same outcome to **MQ** for all strings. When **MQ** are not binary, such as for PDFA, it could be useful to relax equality in order to learn *similar* hypotheses that tolerate small discrepancies. For instance, in [5, 12, 15] equality is replaced by a *similarity* relation. However, that similarity relation is not an equivalence as it lacks transitivity.

In the context of DFA, it is known that using appropriate data structures, such as trees, and efficiently processing counterexamples lead to significant gains in terms of computation time [8, 10]. To the best of our knowledge, these approaches have not yet been explored for active learning of PDFA. The algorithms proposed in [2, 9] rely on **MQ** that compute the probability of a string and on an observation table to store the results. In [14], **MQ** return state distributions, that is, the probability that the target probabilistic automaton enters a state after reading an input string. This requires knowing the number of states of the target in advance. These works focused on theoretical results without implementations of them being publicly available. In [7], learning PDFA is a building block of an assume-guarantee

---

[*]4th LearnAut 2022. Paris, France, July 4, 2022.

framework for verification of probabilistic systems. **MQ** asks for the probability of accepting a string and results are stored in an observation table. However, the overall goal is not to learn a PDFA equivalent to a hidden target but only an appropriate assumption for doing a compositional proof of correctness. To achieve this, the algorithm uses an **EQ** that relies on language inclusion of PDFA and probabilistic model-checking. All these works rely on exact equality of **MQ** outcomes. To deal with noise in distributions, WL^* [15] proposes a non-equivalence similarity relation between probability distributions and develops an algorithm to learn similar hypotheses according to it. Note that WL^* uses clustering to group responses to **MQ** stored in the table.

Based on these observations, we formulate a learning framework for PDFA where the teacher's answers to **MQ** are probability distributions over the symbols a string could be continued with. These are called *next-symbol* probability distributions. Besides, in the case the learner is allowed to produce hypotheses whose next-symbol probability distributions are approximations of the target ones, we propose to resort to quantization as an abstraction which allows us to define a coarser equivalence relation than the latter. This ensures the learnt automaton to be equivalent to the target one modulo this relation. Finally, we propose and implement QuaNT, a learning algorithm for PDFA that uses an adaptation of the tree structure of [10]. In order to assess its performance, we compared QuaNT to a clustering-based algorithm that uses an observation table and a tolerance-based non-equivalence similarity relation, similar to WL^*. The experiments carried out showed that QuaNT was orders of magnitude more efficient.

The paper is organized as follows. Section 2 gives core definitions and results of interest. Section 3 presents our PDFA learning algorithm QuaNT. Section 4 discusses experimental results. Section 5 summarizes the contributions.

2   PDFA

Let Σ be a finite *alphabet* and $\Sigma_\$$ to be the set $\Sigma \cup \{\$\}$, where $\$$ is a special *terminal* symbol not in Σ. $\Delta(\Sigma_\$)$ is probability simplex over $\Sigma_\$$. A PDFA A over Σ is a tuple (Q, q_{in}, π, τ), where Q is a finite set of *states*, $q_{in} \in Q$ is an *initial* state, $\pi : Q \to \Delta(\Sigma_\$)$ associates a probability distribution over $\Sigma_\$$ to each state, and $\tau : Q \times \Sigma \to Q$ is the *transition* (total) function. Fig. 1(*a-b*) depicts an example. Let P_q be the probability distribution over Σ^*, such that $P_q = P(s|q)$ is the probability of $s \in \Sigma^*$ from state $q \in Q$, defined as:

$$P(\lambda|q) = \pi(q)(\$), \lambda \text{ is the empty string}, \quad P(\sigma s|q) = \pi(q)(\sigma) \cdot P(s|\tau(q,\sigma)), \sigma \in \Sigma, s \in \Sigma^*$$

A PDFA A computes a function f_A from Σ^* to $[0, 1]$. For any string $s \in \Sigma^*$, $f_A(s) = P_{q_{in}}(s)$. For instance, the PDFA in Fig. 1 maps the empty string λ to 0 and every string a^n, $n \geq 1$, to 0.5^n. We define $\tau^*(q, s)$ to be the natural extension of τ to strings, that is, the state reached by A when going through s starting at state q:

$$\tau^*(q, \lambda) = q \qquad \tau^*(q, \sigma s) = \tau^*(\tau(q, \sigma), s)$$

Similarly, we define $\pi^*(s|q)$ to be the probability distribution of the state reached by A when going through s from state q:

$$\pi^*(s|q) = \pi(\tau^*(q, s))$$

We denote by $\tau^*(s)$ and $\pi^*(s)$ the state reached when going through s from the initial state q_{in} and the associated distribution, respectively.

2.1   State equivalence

We define the relation $\equiv$ as follows: for every $q, q' \in Q$, $q \equiv q'$ if for every $s \in \equiv$, $\pi^*(s|q) = \pi^*(s|q')$. Clearly, $\equiv$ is an equivalence relation between states.

Proposition 1. $\forall q_1 \equiv q_2 \in Q$, *1)* $\pi(q_1) = \pi(q_2)$, *and 2)* $\forall \sigma \in \Sigma, \tau(q_1, \sigma) \equiv \tau(q_2, \sigma)$. *That is, $\equiv$ is a congruence.*

It follows that $\equiv$ induces a *quotient* PDFA $A/_\equiv = (Q/_\equiv, [q_{in}], \pi/_\equiv, \tau/_\equiv)$, where $Q/_\equiv$ is the set of equivalence classes, $[q] \in Q/_\equiv$ denotes the class of $q \in Q$, and for any $q \in Q$, the probability distribution is $\pi/_\equiv([q]) = \pi(q)$, and the transition function is $\tau/_\equiv([q], \sigma) = [\tau(q, \sigma)]$ for any $\sigma \in \Sigma$. [2]

Proposition 2. *For every PDFA A, $A/_\equiv$ computes the same function as A.*

Therefore, $\equiv$ can be extended to PDFA. For every PDFA A and B, $A \equiv B$ if their respective initial states q_{in}^A and q_{in}^B are equivalent, that is, $q_{in}^A \equiv q_{in}^B$.

Corollary 1. $A \equiv B$ *implies $f_A = f_B$. The converse does not hold in general.*

---

[2]This quotient is different from [13] defined by a partition resulting from merging states according to a compatibility criterion which is not an equivalence relation.

2.2 Minimality

A PDFA is *minimal* if any other PDFA that computes the same function has no less states [2]. The simple PDFA in Fig. 1 is minimal since clearly the function cannot be computed by a PDFA with a single state. Based on $\equiv$ we define a weaker notion of minimality as follows. A PDFA A is said to be *weakly minimal* if for every $q, q' \in Q$, $q \not\equiv q'$. By definition, for every PDFA A, $A/_{\equiv}$ is weakly minimal.

Proposition 3. *Every minimal PDFA A is also weakly minimal. The converse does not hold in general.*

An example of Proposition 3 is the PDFA in Fig. 1. However, the converse is not true, that is, there are weakly minimal PDFA which are not minimal. Consider for instance PDFA A and B in Fig. 2. We have that $f_B((ab)^{n+1}) = (0.1 \cdot 0.2)(0.2 \cdot 0.1)^n \cdot 0.3 = (0.1 \cdot 0.2)^{n+1} \cdot 0.3 = f_A((ab)^{n+1})$, for $n \geq 0$. For any other string $s \neq (ab)^{n+1}$, $f_A(s) = f_B(s) = 0$. Hence, A and B compute the same function. Moreover, they are both weakly minimal because $\pi^*(\lambda|q) \neq \pi^*(\lambda|q')$ for every pair of states $q, q' \in Q$, for each one of the automata, respectively. But B is certainly not minimal because it has more states than A. This example also shows that $f_A = f_B$ does not entail $A \equiv B$.

2.3 Similarity of distributions

The equivalence defined above is not robust in the sense that two states whose probability distributions differ very slightly are not equivalent. Consider the PDFA A_ε (c) in Fig. 1, where $\varepsilon \in (0, 0.5]$. It maps every string a^n, $n \geq 1$, to $(0.5 - \varepsilon)^{n-1}(0.5 + \varepsilon)$. This PDFA is not equivalent to PDFA A in Fig. 1. However, f_{A_ε} tends to f_A as ε tends to 0. Moreover, $\pi(q_1')$ tends to $\pi(q_1)$. Previous works have addressed this issue by introducing a *tolerance* parameter t. In [15], states are compared using the probability of the last symbol of a non-empty string defined as:

$$\pi_q^\ell(s\sigma) = \pi^*(s|q)(\sigma) \text{ where } \sigma \in \Sigma_\$, s \in \Sigma^*, q \in Q \tag{1}$$

It is said that q, q' are *t-equal*, for $t \in [0, 1]$, denoted $q \approx_t q'$, if $L_\infty(\pi_q^\ell, \pi_{q'}^\ell) \leq t$ where $L_\infty(v, v') = \max_x |v(x) - v'(x)|$. PDFA A and B are *t-equal*, denoted $A \approx_t B$, if $q_{\text{in}}^A \approx_t q_{\text{in}}^B$. However, this approach does not lead to an equivalence relation between states. That is why, to cope with small perturbations in probabilities while preserving equivalence between states, we propose to resort to a quantization defined over π^* as follows.

Let $\kappa \in \mathbb{N}$, $\kappa \geq 1$, be a *quantization* parameter. For $n \in \mathbb{N}$, $0 \leq n < \kappa - 1$, we define I_κ^n to be the left-closed right-open interval $[n\kappa^{-1}, (n+1)\kappa^{-1})$, and for $n = \kappa - 1$, the closed interval $[n\kappa^{-1}, 1]$. Now, for every real number $x \in [0, 1]$, we define $\langle x \rangle_\kappa = I_\kappa^n$ such that $x \in I_\kappa^n$. For instance, for $\kappa = 2$, we have the quantization $[0, 0.5), [0.5, 1]$. Two numbers $x, y \in \mathbb{R}$ are *κ-equivalent*, denoted $x =_\kappa y$ if $\langle x \rangle_\kappa = \langle y \rangle_\kappa$. This definition extends naturally to $\Delta(\Sigma_\$)$. We denote $\langle \Delta(\Sigma_\$) \rangle_\kappa$ the partition of $\Delta(\Sigma_\$)$ induced by κ. For $\delta \in \Delta(\Sigma_\$)$, $\langle \delta \rangle_\kappa$ is called the *quantization vector* of δ. For example, $\langle (0.1, 0.3, 0.6) \rangle_2 = (I_2^0, I_2^0, I_2^1)$.

Being $=_\kappa$ an equivalence relation, we can define a quantized version of $\equiv$ as follows: $q \equiv_\kappa q'$ if for every $s \in \Sigma^*$, $\pi^*(s|q) =_\kappa \pi^*(s|q')$. This induces a unique quotient over states $Q/_{\equiv_\kappa}$. Moreover, $\equiv_\kappa$ can be extended to PDFA: for every PDFA A and B, $A \equiv_\kappa B$ if $q_{\text{in}}^A \equiv_\kappa q_{\text{in}}^B$. This allows extending the notion of weakly minimality defined over $\equiv$ to $\equiv_\kappa$: A is said to be *κ-weakly minimal* if for every $q, q' \in Q$, $q \not\equiv_\kappa q'$. However, unlike $\equiv$, its quantized version $\equiv_\kappa$ does not induce a unique quotient PDFA because $\pi/_{\equiv_\kappa}([q]_\kappa)$ can be *any* distribution in $\langle \pi(q) \rangle_\kappa$. Therefore, we define $A/_{\equiv_\kappa} = \{B \mid A \equiv_\kappa B \wedge |Q^A/_{\equiv_\kappa}| = |Q^B|\}$, that is, the set of κ-weakly minimal PDFA which are κ-equivalent to A.

To illustrate quantization, consider the PDFA A (a) and A_ε (c) in Fig. 1. Suppose $\varepsilon \in (0, 0.1)$. For $\kappa = 5$, we have the quantization $[0, 0.2), \ldots, [0.8, 1]$. Then, $\langle \pi(q_1)(a) \rangle_5 = I_5^2 = [0.4, 0.6) = \langle \pi(q_1')(a) \rangle_5$, since $0.4 < 0.5 - \varepsilon < 0.5$, and $\langle \pi(q_1)(\$) \rangle_5 = I_5^2 = [0.4, 0.6) = \langle \pi(q_1')(\$) \rangle_5$, since $0.5 < 0.5 + \varepsilon < 0.6$. Thus, we have that $q_i \equiv_5 q_i'$. Hence, $A \equiv_5 A_\varepsilon$.

Proposition 4. *For every q, q', if $q \equiv_\kappa q'$ then $q \approx_{\kappa^{-1}} q'$.*

3 PDFA Learning: QuaNT

QuaNT is a learning algorithm that constructs a PDFA by interacting with a teacher which makes use of oracles $\mathbf{MQ}_{\text{QuaNT}}$ and $\mathbf{EQ}_{\text{QuaNT}}$. It has three major differences with WL*. First, $\mathbf{MQ}_{\text{QuaNT}}$ returns the next-symbol probability simplex of a string, that is $\mathbf{MQ}_{\text{QuaNT}}(s) = \pi^*(s)$ for $s \in \Sigma^*$; while $\mathbf{MQ}_{\text{WL}^*}(s)$ returns $\pi_{q_{\text{in}}}^\ell(s)$, where π^ℓ is defined in (1). Second, it relies on quantization rather than on tolerance for comparing probability distributions. Since our quantization method induces a congruence, there is no need for clustering states. Third, along the lines of [10] and [8] algorithms, QuaNT builds a *classification* tree instead of a table. Similarly to these algorithms, tree leafs are PDFA states identified by so-called *access* strings and inner nodes are *distinguishing* suffixes. Nevertheless,

in QuaNT the tree is not binary but n-ary, where n is the number of different classes which $\Delta(\Sigma_\$)$ is partitioned into, and tree arcs and leafs are labelled with elements of $\langle\Delta(\Sigma_\$)\rangle_\kappa$ and $\Delta(\Sigma_\$)$, respectively.

N-ary classification tree   Given $s \in \Sigma^*$, we denote $[s]_\kappa$ the class of the state reached by traversing s in the target PDFA A, i.e., $[\tau^*(s)]_\kappa$. Hereinafter, we omit the quantization parameter κ when clear from context. The tree $T_{Acc,Dis}$ maintains a set Acc of access strings and a set Dis of distinguishing strings where for every pair of distinct strings $s, s' \in Acc$: (1) $[s] \neq [s']$, (that is, $\tau^*(s) \not\equiv_\kappa \tau^*(s')$), and (2) $\exists d \in Dis$ s.t. $\pi^*(sd) \neq_\kappa \pi^*(s'd)$. The distinguishing string that labels the root of the tree is always λ which is also an access string, so that the initial state of the PDFA can be accessed.

Proposition 5. *The number of leaves of T is at most $|Q/_{\equiv_\kappa}|$.*

Fig. 3 (left) shows an example of n-ary classification tree. The quantization parameter is $\kappa = 10$. Tree leafs correspond to the states of the PDFA shown on the right, identified with their associated access strings: $Acc = \{\lambda, 0, 1, 10\}$. Every leaf is labeled with a vector corresponding to the probability distribution of the state, where the first element is the probability of \$, the second element is the probability of 0, and the last element is the probability of 1. For example: $\pi(q_\lambda) = (0, 0.5, 0.5)$. Tree arcs are labeled with quantization vectors. To simplify the visualization, only partition indexes are shown. For instance, $(0, 5, 5)$ corresponds to the quantization vector $(I_{10}^0, I_{10}^5, I_{10}^5)$. The root λ of the tree has an arc for each one of the classes in which quantization partitions the set of probability distributions of the states of the PDFA. In the example, there are three, namely quantization vectors $(0, 5, 5)$, $(1, 6, 3)$, and $(1, 3, 6)$, corresponding to distributions $(0, 0.5, 0.5)$, $(0.1, 0.6, 0.3)$, and $(0.1, 0.3, 0.6)$, respectively. The tree explains that states q_λ and q_1 are not equivalent, i.e., $[\lambda] = q_\lambda \neq q_1 = [1]$, because $\pi^*(\lambda|q_\lambda) = \pi(q_\lambda) \neq \pi(q_1) = \pi^*(\lambda|q_1)$, and they are distinguished by $\lambda \in Dis$. They are also not equivalent to q_0 and q_{10} for the same reason. These two states, which have the same probability distribution, are indeed distinguished by the string $1 \in Dis$ because $\pi^*(1|q_0) = \pi(q_{10}) \neq \pi(q_1) = \pi^*(1|q_{10})$.

Finding the class of a state (sifting)   Given a string $s' \notin Acc$, the tree allows to efficiently determine its class $[s']$. That means either finding an access string $s \in Acc$ such that $[s'] = [s]$ or creating a new class by adding s' to Acc. To this end, we define the *sifting* operation as follows. Sift starts at the root of T. Let $d \in Dis$ be the distinguishing string at the current node of the tree. In this case, we perform a membership query to get $\langle\mathbf{MQ}_{\mathrm{QuaNT}}(s'd)\rangle$ and then we descend to the subtree labeled with such quantization vector. Sift continues in this manner until a leaf $s \in Acc$ is reached, in which case, $[s] = [s']$. If there is no arc labeled with the same quantization vector, we have discovered a new class and must update the tree (*sift-update*) by adding s' to Acc with probability $\mathbf{MQ}_{\mathrm{QuaNT}}(s')$ and a new arc $(d, \langle\mathbf{MQ}_{\mathrm{QuaNT}}(s'd)\rangle, s')$. Sifting can be efficiently implemented because the number of membership queries is bounded by the depth of T, and finding quantization vectors and asking membership queries can be cached to minimize the number of vector comparisons and queries to the target system, respectively.

Building a tentative hypothesis $\widehat{A}$ Given a tree T, it is easy to construct $\widehat{A}$ using sift. Each state of $\widehat{A}$ is uniquely identified with an access string in Acc. For each state q_s, $s \in Acc$, and symbol $\sigma \in \Sigma$, $\tau(q_s, \sigma) = q_{s'}$, where $s' = \mathrm{sift}(s\sigma)$, and $\pi(q_s)$ is the probability distribution associated with the leaf node s. In the case that sift generates an update, the building process is restarted. Because of Prop. 5, this eventually terminates. Moreover, by construction, $\widehat{A}$ is a PDFA.

Equivalence queries   For $\mathbf{EQ}_{\mathrm{QuaNT}}$ we use an adaptation of Hopcroft-Karp where states are compared as follows: $\langle\pi(\widehat{q})\rangle_\kappa = \langle\pi(q)\rangle_\kappa$, where q is a state of the target PDFA A and $\widehat{q}$ is a state of the hypothesis PDFA $\widehat{A}$.

Processing a counterexample   Let γ be a counterexample returned by $\mathbf{EQ}_{\mathrm{QuaNT}}$, that is $\pi_A^*(\gamma) \neq_\kappa \pi_{\widehat{A}}^*(\gamma)$. Let γ_i be the i-th symbol of γ, $\gamma[i]$ be the prefix of γ of length i, i.e., $\gamma[i] = \gamma_1...\gamma_i$, $s_i = \mathrm{sift}(\gamma[i])$, and $\widehat{s}_i$ the string associated to the state $\tau_{\widehat{A}}^*(\gamma[i])$. Let $1 \leq j \leq |\gamma|$ be the first index such that $\widehat{s}_j \neq s_j$. This means that $[\widehat{s}_j] \neq [s_j]$ and $\widehat{s}_{j-1} = s_{j-1}$, but the states reached by $\widehat{A}$ and A after traversing $\gamma[j-1]$ are not equivalent since when continued with γ_j they reach non-equivalent states. That is, $[s_{j-1}] \neq [\gamma[j-1]]$. Therefore, T has to be updated by adding a new leaf node $\gamma[j-1]$ representing the newly discovered class. Then, $\gamma[j-1]$ is added to Acc. Now, let $d \in Dis$ be the least common ancestor in T of $\widehat{s}_j$ and s_j. Then $\gamma_j d$ is a distinguishing string for s_{j-1} and $\gamma[j-1]$. In terms of tree operations, the leaf s_{j-1} is replaced by an inner node $\gamma_j d$ and two children, namely s_{j-1} and $\gamma[j-1]$.

Complete algorithm   Algorithm 1 shows QuaNT pseudocode. The algorithm begins by executing BuildSingleStatePDFA which creates an initial hypothesis $\widehat{A}$, with a single state q_λ with a loop for each symbol and executes

---

Algorithm 1: QuaNT

Parameter: Quantization Parameter κ

Output : PDFA $\widehat{A}$

1 $\widehat{A} \leftarrow \text{BuildSingleStatePDFA}(\kappa)$;

2 $\gamma \leftarrow \mathbf{EQ}_{\text{QuaNT}}(\widehat{A}, \kappa)$;

3 **if** $\gamma = \bot$ **then**

4 $\quad$ **return** $\widehat{A}$;

5 $T \leftarrow \text{InitializeTree}(\gamma, \kappa)$;

6 **while** $\gamma \neq \bot$ **do**

7 $\quad$ $\widehat{A} \leftarrow \text{BuildAutomaton}(T)$;

8 $\quad$ $\gamma \leftarrow \mathbf{EQ}_{\text{QuaNT}}(\widehat{A}, \kappa)$;

9 $\quad$ **if** $\gamma \neq \bot$ **then**

10 $\quad\quad$ $T \leftarrow \text{UpdateTree}(T, \gamma, \kappa)$;

11 **return** $\widehat{A}$;

$\mathbf{MQ}_{\text{QuaNT}}(\lambda)$ to get the probability distribution for the state. Then, it calls $\mathbf{EQ}_{\text{QuaNT}}(\widehat{A}, \kappa)$, which either returns $\bot$ and terminates or a counterexample γ enabling the initialization of the tree. The first tree T has a root labeled with the distinguishing string λ and two children, one with the access string λ and the other with the counterexample γ. Once T is initialized, the main loop of the algorithm begins. It consists in using T to build a PDFA $\widehat{A}$, then using $\mathbf{EQ}_{\text{QuaNT}}$ to compare it with the target PDFA A. If a counterexample is returned, T is updated, resulting in new states being discovered. Otherwise it means all states in $Q/_{\equiv_\kappa}$ have been found which implies $\widehat{A} \equiv_\kappa A$.

Proposition 6. *For any PDFA A,* QuaNT *terminates and computes a PDFA $\widehat{A} \in A/_{\equiv_\kappa}$.*

4 Experiments

In this section we present the results of the experiments carried out to compare QuaNT with a clustering-based algorithm that uses an observation table and a tolerance-based non-equivalence similarity relation, similar to WL*, that we call L$_\text{p}^*$ (see Appendix A). We compared the learning algorithms on randomly generated PDFA. The generation technique works in two steps. First, it constructs random DFA over Σ. Second, DFA are transformed into PDFA by assigning a probability distribution over $\Sigma_\$$ to every state. The first step uses the method described in [11] based on results from [4]. Let n be the desired number of reachable states of a DFA, called its *nominal* size. The method consists in randomly generating DFA of a total of $n \cdot m \cdot \rho_m^{-1}$ possibly unreachable states, for $m = |\Sigma|$, where $\rho_m = m - W_0 \cdot m \cdot e^{-m}$ and W_0 is the Lambert-W function, and then computing its accessible part by a depth-first traversal. It is important to remark that this method does not guarantee the *actual* size of the accessible part to be exactly n, but to be normally distributed around n. To obtain exactly n accessible states, the method could be repeated using a rejection algorithm. However, in practice, this proved to be very inefficient, being almost impossible to generate DFA of accessible size bigger than 100 in reasonable time. All experiments threw perfect scores for all computed metrics (word error rate, normalized discounted cumulative gain, log probability error [15]) for all algorithms on the same test set of strings. Therefore, the analyses of the experimental results are mainly focused on execution time and structure size. For ease of comparison, figures show trend lines.

4.1 Experiment 1

In this experiment we compared QuaNT and L$_\text{p}^*$. For this, 10 random PDFA over a binary alphabet ($m = 2$) of nominal sizes $n = 100, 200, 300$ were generated, and each algorithm was run 10 times for each PDFA. For QuaNT $\kappa = 1000$, and for L$_\text{p}^*$, $t = \kappa^{-1}$ (Prop 4). Fig. 4a shows learning time medians for every actual size. Notably, L$_\text{p}^*$ execution time grows much faster than QuaNT's. Indeed, QuaNT achieves a speedup of approximately $0.2n$, reaching around 60x for the biggest PDFA (see Fig. 6a). This experiment also showed that the size of L$_\text{p}^*$'s observation table grows bigger than QuaNT's tree which partly explains the gains in execution time (Fig. 4b).

4.2 Experiment 2

In this experiment 10 random PDFA of nominal size $n = 100$ were generated for alphabet size $m = 2, 4, 8, 16, 32$. We compared QuaNT and L_p^* with $\kappa = 1000$ and $t = \kappa^{-1}$. Each algorithm was run 10 times for each PDFA. Fig. 5a shows the learning time medians for every alphabet size. As it can be seen L_p^* seems to be more sensitive to the growth in the alphabet size.

4.3 Experiment 3

In this experiment, we compare the algorithms for different values of tolerance and quantization parameter: $\kappa = 10, 100, 500, 1000, 2000, 3000$, with $t = \kappa^{-1}$. For every parameter configuration 10 random PDFA of nominal size $n = 300$ and alphabet size $m = 2$ were generated, and each algorithm was run 10 times for each PDFA. Fig. 5b shows the median learning times. As it can be seen both algorithms appear to stabilize its execution time after some parameters sizes ($\kappa = 500, t = 1/1000$).

4.4 Experiment 4

Here, QuaNT was evaluated on bigger nominal sizes $n = 1000, 2000, 5000$, fixing $\kappa = 1000$ and $m = 2$. For every parameter configuration, 10 random PDFA were generated and each algorithm was run 10 times for each PDFA. Fig. 6b shows median learning times. Clearly, QuaNT still manages to learn PDFA from systems that are intractable for L_p^*. Assuming a linear speedup of $0.2n$ from Experiment 4.1, the learning time of a PDFA of size 5000 would be almost a month for L_p^*.

4.5 Experiment 5

In previous experiments it is noted that nearly all states in the randomly generated PDFA have distinct next symbol distributions, that is, most states are distinguished by λ, thus producing shallow tree structures (depth 1 or 2). In order to analyze cases where states share next symbol distributions we parameterized the PDFA random generation by a number d of distributions to use. The algorithm first randomly generates a set of d distributions and then labels each state by uniformly picking one in this set.

For this experiment, 10 random PDFA over a binary alphabet ($m = 2$) of nominal size $n = 300$ were generated for different values of d (ranging from 2 to 16). Each algorithm was run 10 times for each PDFA. For QuaNT, $\kappa = 1000$, and for L_p^*, $t = \kappa^{-1}$ (Prop 4). Fig. 7a shows learning time medians for every d value. Notably, L_p^* execution time still grows faster than QuaNT's. The only case where QuaNT achieves worse learning time than L_p^* is for $d = 2$. Otherwise, QuaNT significantly benefits from the increase in d. Regarding structure sizes, L_p^*'s observation table grows bigger than previous experiments, being negatively affected by smaller values of d. However QuaNT maintains similar sizes to those observed in experiment 1 (Fig. 7b).

5 Conclusions

We defined a robust notion of similarity of states in PDFA based on a congruence over states and a quantization of their probability distributions. This induces a precisely defined inductive bias for PDFA learning as a set of quantized weakly minimal PDFA. Based on this, we developed a new PDFA active MAT-learning algorithm called QuaNT which uses an n-ary tree to efficiently learn a PDFA in the hypothesis space. In order to empirically assess the efficiency of QuaNT, we presented an adaptation of WL*, namely L_p^*, which works with a non-equivalence similarity relation over distributions and an observation table. Algorithms were compared on a number of randomly generated PDFA. The experiments confirmed notable execution time gains achieved by QuaNT.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[2] A. Beimel, F. Bergadano, N. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. *Journal of the ACM*, 47(3):506–530, 2000.

[3] F. Bergadano and S. Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM Journal on Computing*, 25(6):1268–1280, 1996.

[4] A. Carayol and C. Nicaud. Distribution of the number of accessible states in a random deterministic automaton. *Leibniz Int. Proc. in Informatics*, 14:194–205, 2012.

[5] A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *J. Machine Learning Research*, 5:473–497, 2004.

[6] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[7] L. Feng, T. Han, M. Kwiatkowska, and D. Parker. Learning-based compositional verification for synchronous probabilistic systems. In *ATVA'11*, pages 511–521. LNCS 6996, 2011.

[8] M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV'14*, pages 307–322. LNCS 8734, 2014.

[9] A. Kaznatcheev and P. Panangaden. Weighted automata are compact and actively learnable. *Inf. Process. Lett.*, 171:106–133, 2021.

[10] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.

[11] C. Nicaud. Random deterministic automata. In *MFCS'14*, pages 5–23. LNCS 8634, 2014.

[12] M. Tappler, B. K. Aichernig, G. Bacci, M. Eichlseder, and K. G. Larsen. L*-based learning of markov decision processes (extended version). *Form. Asp. Comput.*, 33(4–5):575–615, 2021.

[13] F. Thollard, P. Dupont, and C. de la Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *7th Conf. Machine Learning*, page 975–982, 2000.

[14] W.G. Tzeng. Learning probabilistic automata and markov chains via queries. *Machine Learning*, 8:151–166, 1992.

[15] G. Weiss, Y. Goldberg, and E. Yahav. Learning deterministic weighted automata with queries and counterexamples. In *Adv. in Neural Information Proc. Sys.*, volume 32, 2019.

A $\;\;$ L$_\mathrm{p}^*$

Here we present L$_\mathrm{p}^*$, a variant of WL* [15]. Similarly, it uses an observation table $O_{Pre,Suf}$ for storing outcomes of $\mathbf{MQ}_{\mathrm{L}^*}$, where $Pre \subset \Sigma^*$ is the set of *prefixes* (stored in row indices) and $Suf \subset \{\$\} \cup \Sigma^+$ is the set of *suffixes* (stored in column indices). For every $p \in Pre$ and $s \in Suf$, $O_{Pre,Suf}[p,s] = \mathbf{MQ}_{\mathrm{L}^*}(ps) = \pi_{q_\mathrm{in}}^\ell(ps)$, where π^ℓ is defined in (1). Unlike WL*, Pre is divided in two parts [6], namely *RED* which are the rows used to construct *states*, and *BLUE* which are the rows representing continuations. The algorithm ensures that for every row in *RED* all its continuations are in Pre. L$_\mathrm{p}^*$ consists of three main steps. The first one expands $O_{Pre,Suf}$ through the use of $\mathbf{MQ}_{\mathrm{L}^*}$ until it becomes *closed* and *consistent*. The second one constructs a hypothesis automaton using a greedy clustering technique rather than DBScan. The third one calls $\mathbf{EQ}_{\mathrm{L}_\mathrm{p}^*}$ with the proposed hypothesis. For this we adapted Hopcroft-Karp algorithm for checking t-similarity. When A and $\widehat{A}$ are found to be not t-similar, $\mathbf{EQ}_{\mathrm{L}_\mathrm{p}^*}$ returns a counterexample which is added to *RED* together with all its prefixes, and all their continuations to *BLUE* (provided they are not already in *RED*). These steps are repeated as long as $\mathbf{EQ}_{\mathrm{L}_\mathrm{p}^*}$ yields a counterexample, otherwise it stops and returns the last hypothesis.

B $\;\;$ Proofs

Proof of Proposition 1

1. By hypothesis, $\pi^*(\lambda|q_1) = \pi^*(\lambda|q_2)$. By definition, $\pi^*(\lambda|q_i) = \pi(q_i)$, $i = 1, 2$. Therefore, $\pi(q_1) = \pi(q_2)$.

2. By hypothesis, $\forall s \in \Sigma^*$, $\pi^*(\sigma s|q_1) = \pi^*(\sigma s|q_2)$. By definition, $\pi^*(\sigma s|q_i) = \pi^*(s|\tau(q_i, \sigma))$, $i = 1, 2$. Thus, $\forall s \in \Sigma^*$, $\pi^*(s|\tau(q_1, \sigma)) = \pi^*(s|\tau(q_2, \sigma))$. Hence, $\tau(q_1, \sigma) \equiv \tau(q_2, \sigma)$. $\square$

Proof of Proposition 2   Clearly, for all $q, q' \in Q$, if $q \equiv q'$ then $P(s|q) = P(s|q')$ for every $s \in \Sigma^*$. Hence, $P(s| [q_{\text{in}}]) = P(s|q_{\text{in}})$. $\square$

Proof of Proposition 3   Suppose that A is not weakly minimal. Then, there are states in A which are equivalent. Thus, $A/_{\equiv}$, which computes the same function as A, has strictly less states than A, which contradicts the hypothesis. $\square$

Proof of Proposition 4 $q \equiv_\kappa q'$ implies $\pi^*(s|q) =_\kappa \pi^*(s|q')$ for all $s \in \Sigma^*$. Therefore, $\pi_q^\ell(s\sigma) =_\kappa \pi_{q'}^\ell(s\sigma)$ for all $s \in \Sigma^*, \sigma \in \Sigma_\$$. Then, $|\pi_q^\ell(s\sigma) - \pi_{q'}^\ell(s\sigma)| \leq \kappa^{-1}$ for all $s \in \Sigma^*, \sigma \in \Sigma_\$$. Hence, $q \approx_{\kappa^{-1}} q'$. $\square$

Proof of Proposition 5   If $s, s' \in Acc$ then $[s] \neq [s']$, hence $|Acc| \leq |Q/_{\equiv_\kappa}|$. $\square$

Proof of Proposition 6
Termination First notice that all inner operations terminate. Second, when a counterexample is returned, the number of leaves of T increases by at least 1. Therefore by Proposition 5 and the fact that $|Q/_{\equiv_\kappa}| \leq |Q|$, QuaNT terminates.
Correctness When QuaNT terminates, the hypothesis $\widehat{A}$ constructed is such that $\widehat{Q} = Q/_{\equiv_\kappa}, \widehat{\pi} = \pi/_{\equiv_\kappa}, \widehat{\tau} = \tau/_{\equiv_\kappa}$. Therefore $\widehat{A} \equiv_\kappa A$. $\square$
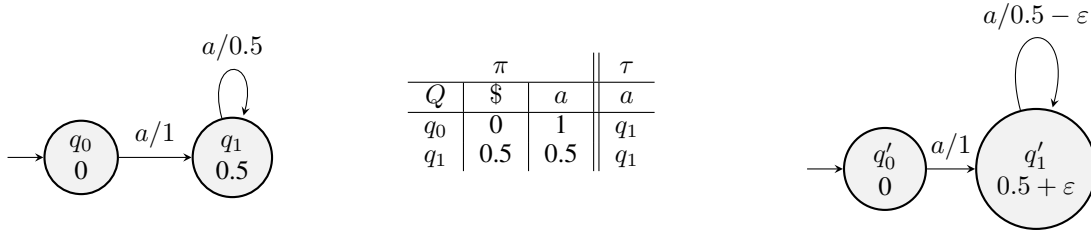
C   Figures

Figure 1: (*a-b*) PDFA over $\Sigma = \{a\}$ with $q_{\text{in}} = q_0$. (*c*) PDFA A_ε.
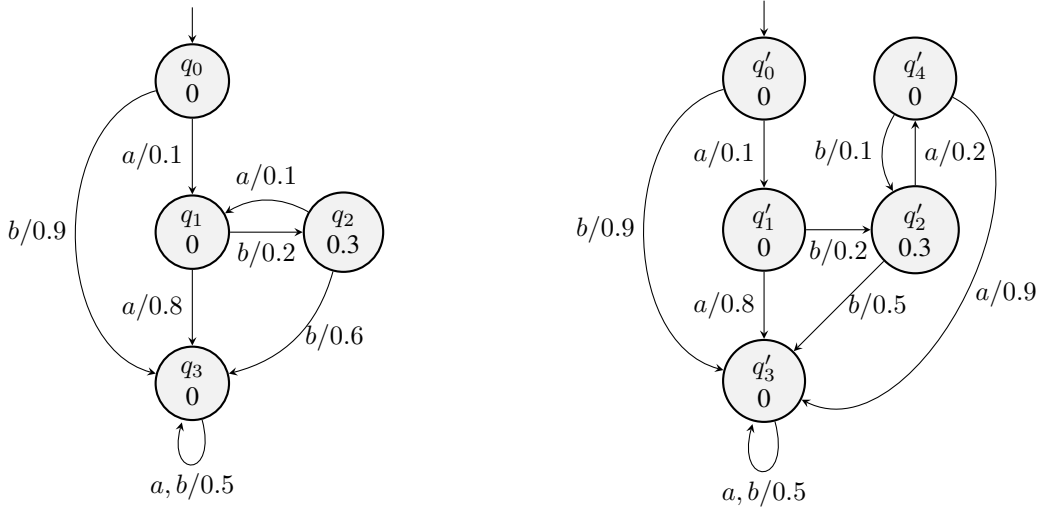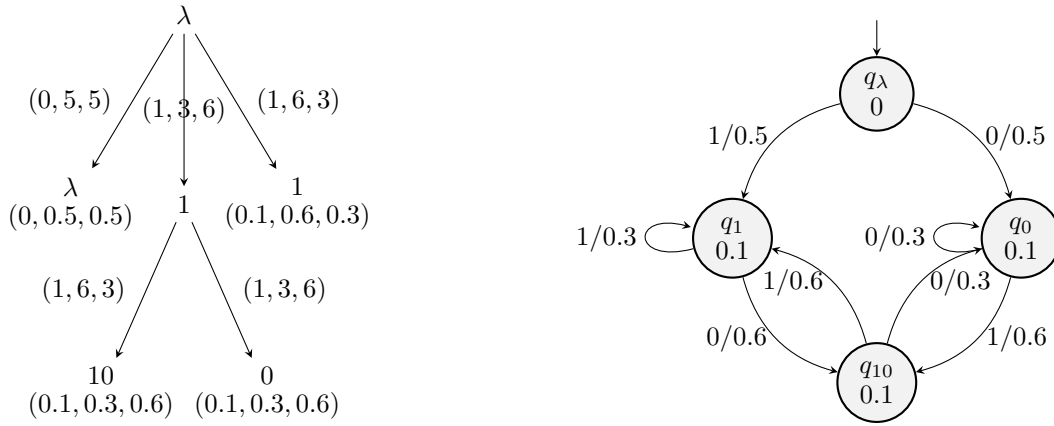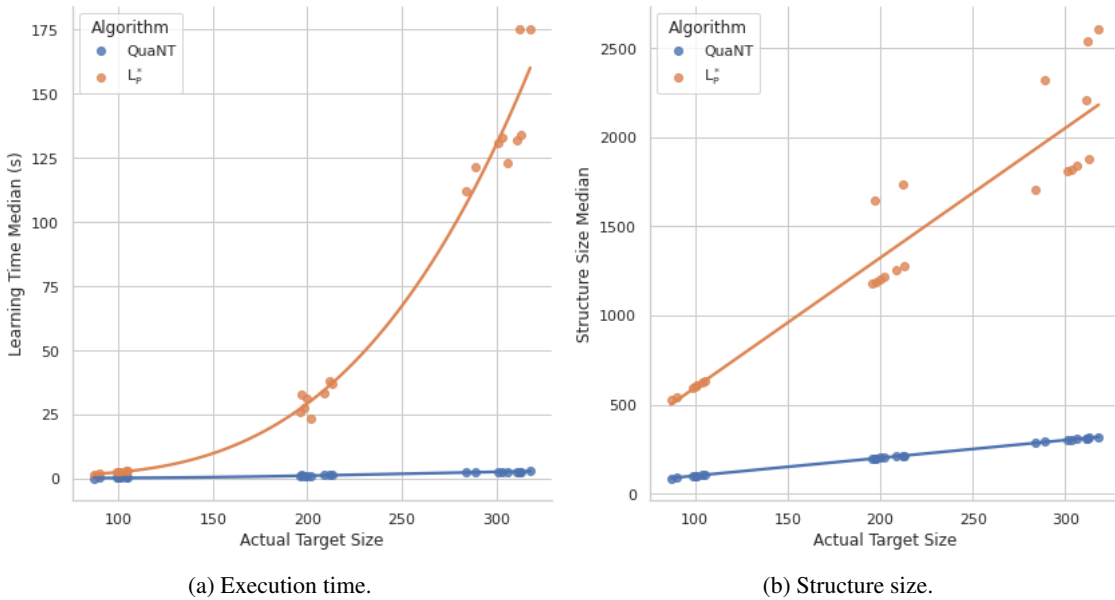

Figure 2: PDFA A (left) and B (right).

Figure 3: An n-ary classification tree (left), and the corresponding PDFA (right)

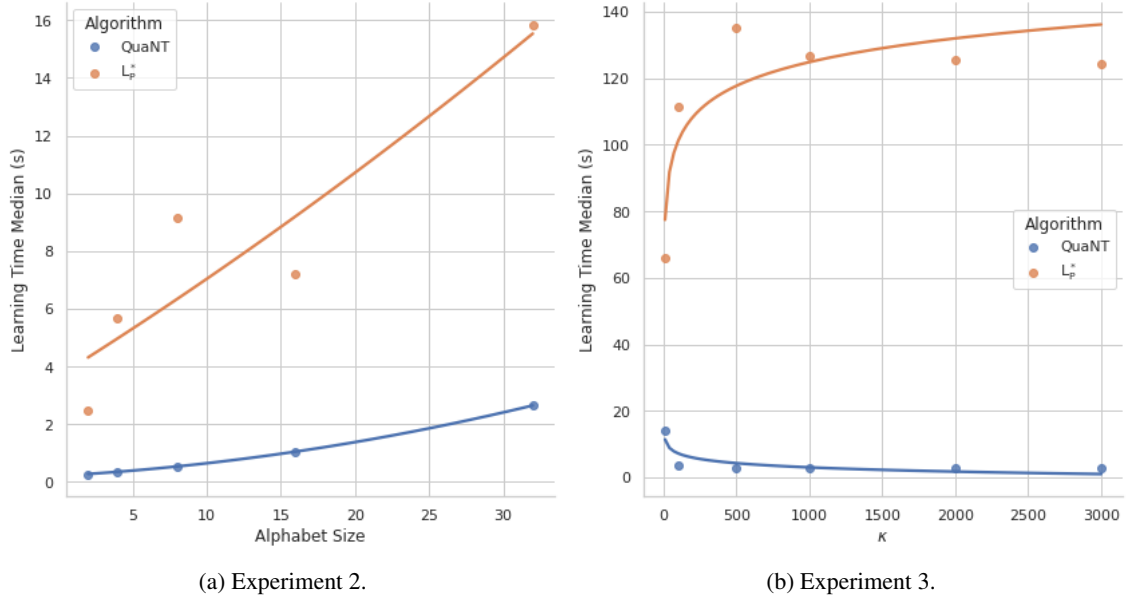(a) Execution time.

(b) Structure size.

Figure 4: Experiment 1
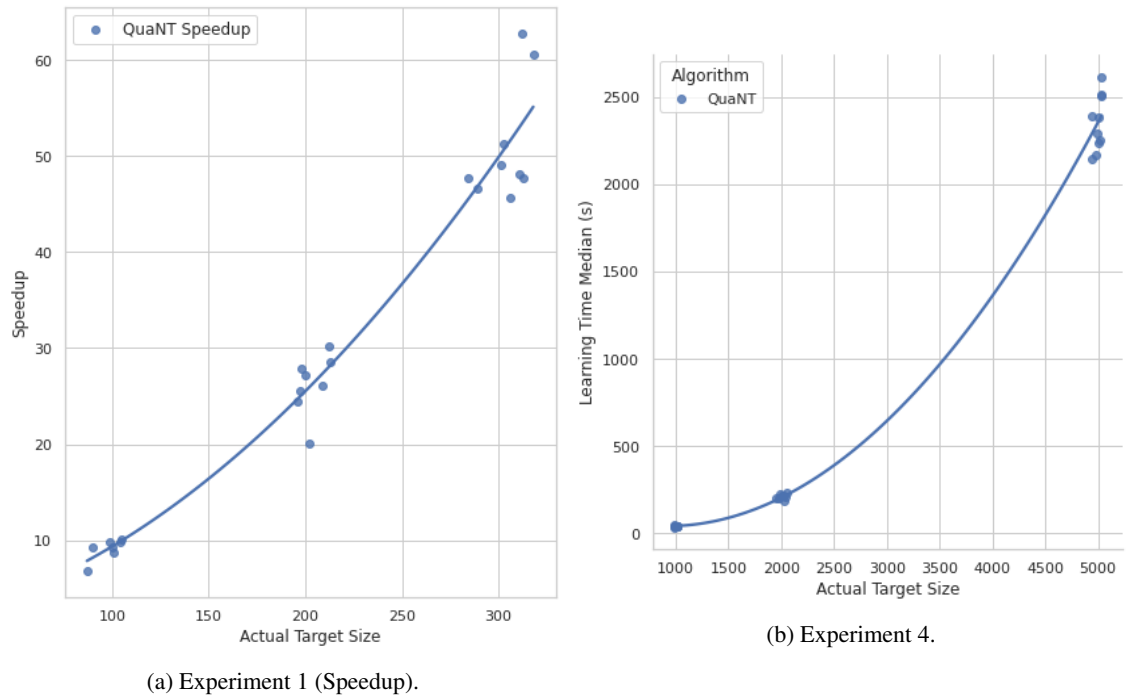
9

(a) Experiment 2.   (b) Experiment 3.

Figure 5: Experiments 2 and 3.

(a) Experiment 1 (Speedup).

(b) Experiment 4.

Figure 6: Experiments 1 (Speedup) and 4.

(a) Execution time.

(b) Structure size.

Figure 7: Experiment 5.