Pattern Recognition 150 (2024) 110326

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/pr

A categorical interpretation of state merging algorithms for DFA inference

Juan Miguel Vilar

Institute of New Imaging Technologies, Universitat Jaume I, Av. de Vicent Sos Baynat s/n, 12071, Castelló de la Plana, Spain

ARTICLE INFO

Keywords:
Grammatical inference
DFA
Category theory
RPNI
EDSM

ABSTRACT

We use Category Theory to interpret the family of algorithms for inference of DFAs that work by merging states. This interpretation allows us to characterize the structure of the search space and to define criteria for the convergence of these algorithms to the correct DFA. We also prove that the well-known EDSM algorithm does not identify DFAs in the limit.

1. Introduction

Deterministic finite state automata (DFA) are pervasive models used in Computer Science with many applications ranging from classic ones such as models for lexical analysers [1] or parsing [2] to pattern matching [3], process mining [4], in medical sciences for analysis of ECG data [5] and analysis of clinical documentation [6], and neural network interpretation in bioinformatics [7] and computational linguistics [8]. In many cases, these models are learned from data like in the previously mentioned [4] and [7] or in [9].

Category Theory is gaining a lot of momentum in Computer Science due to the powerful abstractions and generalizations that it provides. It also provides a uniform way of understanding and using those abstractions. This has been especially noticeable in functional programming but also in the most theoretical aspects of Computer Science, including Automata Theory and the growing interest in understanding automata learning from a categorical perspective. Examples of this are the work on using the concept of T-algebras as the representation of automata for learning in [10] or, also following an algebraic approach, the work in [11]. A general categorical framework is present in the works of Gerco van Heerdt et al. [12,13]. Automata can also be modelled as functors [14], an approach that is closer to our approximation. However, most if not all the previous works are based on Angluin's L* algorithm [15]. The paradigm followed is that of *query learning*: the *learner*, an algorithm, can make queries to a teacher, an oracle. The main result of [15] is that a *learning algorithm* can identify a regular language if it can make two types of query: whether a string belongs to the target language and whether a hypothesis is correct.

However, an adequate teacher is not always available. Often the resource at hand is a corpus of samples of the desired language. In these situations, the *learning in the limit* paradigm can be more appropriate. Under this paradigm, the learning algorithm receives an increasing sequence of samples from the target language. The task of the learner is to produce a hypothesis for each of those samples. In this case, there is no defined point at which the task is accomplished, the learner is considered successful if there is a point in the sequence of samples such that from there on, all the hypotheses coincide with the target.

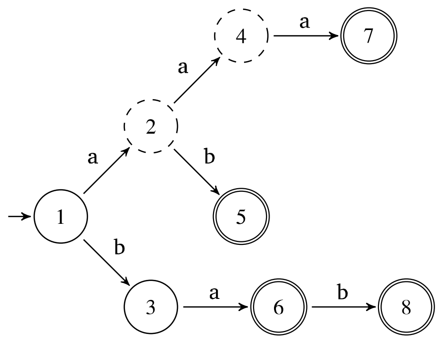
A family of algorithms that follow the paradigm of identification in the limit can be grouped under the label of "state merging". The roots of these methods can be traced to the works of Lang [16] and Oncina and García [17], and surveys can be found in the works of Bugalho et al. [18] and Tîrnăucă [19]. These algorithms were originally developed in the context of Grammatical Inference but they have found applications like the above-mentioned work in process mining [4]. These methods start by representing the input sample as an automaton in the form of a tree. This automaton exactly represents the input sample but does not generalize it. Generalization is achieved by progressively merging the states of the tree. When the sample is sufficiently representative of the target language, the merging process is guaranteed to produce the minimal automaton that recognizes it.

1.1. An example of learning by state merging

In this section, we use an example to show how the RPNI algorithm [17] works. The target is the language of the strings of a's and b's with an odd number of a's. The available information is that the language contains the strings $\{ab, ba, aaa, bab\}$ and it does not contain those in the set $\{\lambda, b\}$, where λ is the empty string. This information is represented by the following tree:

E-mail address: jvilar@uji.es.

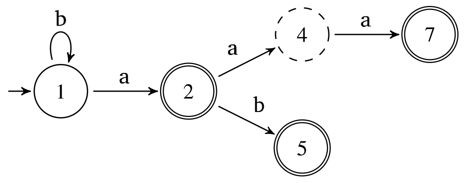
https://doi.org/10.1016/j.patcog.2024.110326
Received 19 June 2023; Received in revised form 12 December 2023; Accepted 5 February 2024
Available online 8 February 2024



(1)

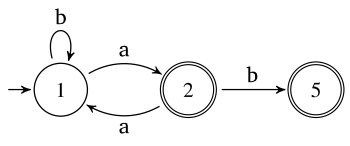
The initial state is marked by an arrow, accepting states are marked by a double line, and those states for which is not known whether they are final have a dashed contour.

The nodes are traversed in breadth-first order trying to merge each state with the previous ones. The first merge tried is 2 with 1, but it is rejected since to keep the automaton deterministic it would imply merging also states 4 and 7 to state 1. But state 7 accepts and 1 rejects so they cannot be merged. The second merge is 3 with 1, which is successful, leading to the following automaton:



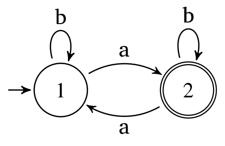
(2)

Note that state 6 merged with state 2 and 8 with 5. State 2 is now accepting since it has merged with state 6. State 4 can be merged with state 1:



(3)

Finally, state 5 cannot be merged with state 1 but it can be merged with state 2:



(4)

This is the minimal DFA that recognizes the language.

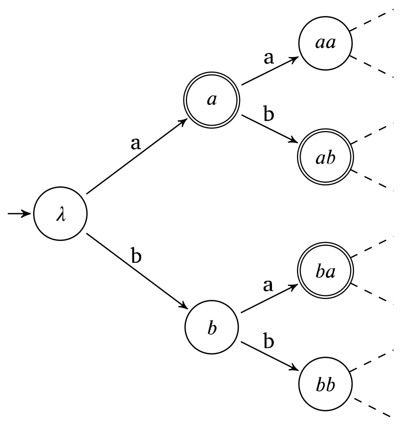
1.2. Why category theory?

A category is a collection of objects and morphisms between them. We are interested in two categories. The first one is the category of the automata that recognize a language, whose objects are automata and the morphisms are correspondences between their states that preserve the language. This is explained in Section 2.1 using the work of Colcombet and Petrişan [20]. The second category has as objects the automata that a merging algorithm can produce and its morphisms are the merges. This category is defined in Section 5.3 and we will see that when interpreted as the search space for merging algorithms, it has a richer structure than the space presented in [21].

We also use extensively the concept of functor. A functor relates maps the objects and morphisms of one category to the objects and morphisms of another category in a way that preserves the structure of the original category. We will define a functor between the two categories mentioned above in Section 5.4. This functor will allow us to give conditions for the convergence of learning algorithms in Section 5.

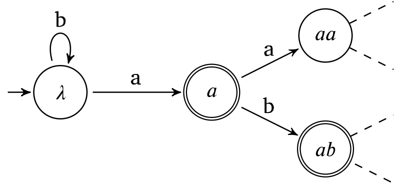
1.3. The relationship of learning with minimization

As explained above, there is a category that contains every automata that recognizes a language. An important automaton in this category is the *initial automaton*, so called because it is related by a morphism to every other automaton in the category. The initial automaton is an infinite tree and we can interpret the morphism from it to another automaton as the merging of the states of this tree. The initial automaton for the language of the strings with an odd number of a's is



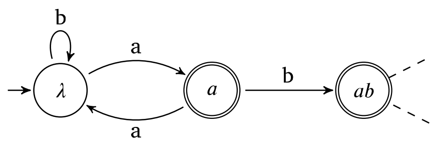
(5)

The tree (1) representing the sample can be seen as a partial view of (5) and the merges of Section 1.1 can be translated to it. The merge of the states 3 and 1 in the initial tree can be translated here as the merging of the states b and λ, leading to this automaton:



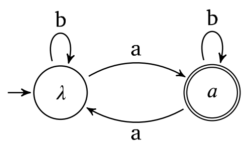
(6)

The merge of 4 and 1 in (2) can be translated as the merge of aa and λ, therefore we can interpret (3) as the partial view of this automaton:



(7)

Finally, the merge of 5 and 2 leads to the merge of ab and a that produces an automaton isomorphic to Eq. (4):



(8)

When looking at them from the perspective of Category Theory, each of these merges is a natural transformation.

1.4. Objective of the paper

The objective of this paper is to formalize the intuitions mentioned above and to use them to describe the search space of merging algorithms and to derive conditions for identification in the limit of DFAs by state merging algorithms. Our definition of the search space is richer than that of [21] and the conditions we present allow us to easily prove the convergence of several classic algorithms. We also prove that EDSM [22] does not identify in the limit the class of DFAs.

2. Basic concepts

In this section, we present the concepts and notation used in the rest of the paper. We consider two formalizations of automata, the classic one, which the reader can find in any introductory book like [23], and the functorial one which is based on [20] and is explained in the next section. The general concepts we will use from Category Theory can be found in introductory texts like [24] and some of them are concisely explained when used, those directly related to automata are explained next.

2.1. Automata as functors

We follow the approach of Colcombet and Petrişan [20] to define automata using functors. Let $\mathcal{I}$ be an arbitrary small category and $\mathcal{O}$ a full subcategory. We can consider $\mathcal{I}$ as the specification of the inner computations of the automata and $\mathcal{O}$ as the specification of the observable behaviour. Then:

- A $\mathcal{C}$-*language* is a functor: $\mathcal{L} : \mathcal{O} \to \mathcal{C}$.
- A $\mathcal{C}$-*automaton* is a functor: $\mathcal{A} : \mathcal{I} \to \mathcal{C}$.
- A $\mathcal{C}$ automaton $\mathcal{A}$ *accepts* a $\mathcal{C}$-language $\mathcal{L}$ when $\mathcal{A} \cdot \iota = \mathcal{L}$:

$$\begin{tikzcd} \mathcal{O} \arrow[r, "\mathcal{L}"] \arrow[rr, bend left, "\iota"] & \mathcal{C} & \mathcal{I} \arrow[l, "\mathcal{A}"] \end{tikzcd}$$ where ι is the inclusion functor.

We particularize this definition for the case of deterministic automata. Usually, a *deterministic automaton* A is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states [23]. The use of categories changes the definitions. The initial state is identified with a function[1] $\vec{q_0} : \mathbf{1} \to Q$. The transition function is considered as a family of functions, one for each element of the alphabet $\delta_a : Q \to Q$ so that $\delta_a(p) = \delta(a, p)$.

We also define for every string $w \in \Sigma^*$ the function $\delta_w : Q \to Q$ with $\delta_\lambda(q) = q$ and $\delta_{wa} = \delta_a \cdot \delta_w$ for $a \in \Sigma$. Finally, for every $q \in Q$, the function $\delta_q : \Sigma^* \to Q$ is defined as $\delta_q : w \mapsto \delta_w(q)$.

Let $\mathbf{2} = \{0, 1\}$. We can identify F with a function $F : Q \to \mathbf{2}$ in such a way that $F(q)$ is zero if q is not final or one otherwise.

With these observations A can be seen as a functor from $\mathcal{I}_{\Sigma^*}$ to the category **Set**, where $\mathcal{I}_{\Sigma^*}$ is the free category defined from the diagram

$$\begin{tikzcd} \mathsf{in} \arrow[r, "\triangleright"] & \mathsf{states} \arrow[loop above, "a"] \arrow[r, "\triangleleft"] & \mathsf{out} \end{tikzcd} \tag{9}$$

where there is an arrow for every $a \in \Sigma$. The effect of A on objects is: $\mathrm{A}(\mathsf{in}) = \mathbf{1}, \mathrm{A}(\mathsf{states}) = Q, \mathrm{A}(\mathsf{out}) = \mathbf{2}$. and on arrows: $\mathrm{A}(\triangleright) = q_0, \mathrm{A}(a) = \delta_a, \mathrm{A}(\triangleleft) = F$.

For instance, automaton (4) corresponds to the functor A defined by: $\mathrm{A}(\mathsf{in}) = \mathbf{1}, \mathrm{A}(\mathsf{states}) = \{1, 2\}, \mathrm{A}(\mathsf{out}) = \mathbf{2}$, and $\mathrm{A}(\triangleright) = \vec{1}, \mathrm{A}(a) = \delta_a$, $\mathrm{A}(b) = \delta_b, \mathrm{A}(\triangleleft) = F$ where $\delta_a(1) = 2, \delta_a(2) = 1, \delta_b(1) = 1, \delta_b(2) = 2, F(1) = 0, F(2) = 1$.

This allows us to take a string $a_1 \dots a_n$ and associate to it the morphism $\triangleright a_1 \dots a_n \triangleleft$ in $\mathcal{I}_{\Sigma^*}$. This morphism is translated by A into the function $F \cdot \delta_{a_n} \cdot \ldots \cdot \delta_{a_1} \cdot \vec{q_0} : \mathbf{1} \to \mathbf{2}$, which can be interpreted as the result of applying the automaton to the string. Returning to our example, the string ab is associated with the morphism $\triangleright ab \triangleleft$. This morphism is translated by A into the function:

$$F \cdot \delta_b \cdot \delta_a \cdot \vec{1} = F \cdot \delta_b \cdot \vec{2} = F \cdot \vec{2} = \vec{1}.$$

So the string is accepted.

The *language accepted* by an automaton A, $L(\mathrm{A})$, is the set of strings accepted by it. This set corresponds to the function $L(\mathrm{A}) : \Sigma^* \to \mathbf{2}$.

[1] The set $\mathbf{1}$ is a singleton and we use $\vec{a}$ to denote the function $\vec{a} : \mathbf{1} \to A$ that selects the element a of a set A.

Let $\mathbf{Auto}(L)$ be the category of automata that accept the language L. The morphisms of this category are the natural transformations between the functors representing the automata[2]. Its initial automaton (the one that has a morphism from it to every other automaton) is denoted by $\mathcal{I}(L)$ and corresponds to the diagram:

$$\begin{tikzcd} \mathbf{1} \arrow[r, "\vec{\lambda}"] & \Sigma^* \arrow[loop above, "\delta_a"] \arrow[r, "L?"] & \mathbf{2} \end{tikzcd}$$

Its states are the strings of Σ^*, the initial state is the empty string λ, the transition function is the function $\delta_a(w) = wa$, and the final states are the strings that belong to L. An example is the automaton (5) of Section 1.3

The final automaton (the one that has a morphism from every other automaton to it) of $\mathbf{Auto}(L)$ is denoted by $\mathcal{F}(L)$ and corresponds to the diagram:

$$\begin{tikzcd} \mathbf{1} \arrow[r, "L"] & \mathbf{2}^{\Sigma^*} \arrow[loop above, "\delta_a"] \arrow[r, "\lambda?"] & \mathbf{2} \end{tikzcd}$$

Its states are the languages of Σ^*, the initial state is the language L, the transition function is $\delta_a(L) = \{w \in \Sigma^* \mid aw \in L\}$ (the left quotient of L by the letter a), and the final states are those languages that contain the empty string.

The minimal automaton recognizing a language L, $\mathtt{Min}(L)$, can be characterized as the result of the EM-factorization of the morphism from $\mathcal{I}(L)$ to $\mathcal{F}(L)$ [20]:

$$\begin{tikzcd} \mathcal{I}(L) \arrow[r, two heads, "e"] & \mathtt{Min}(L) \arrow[r, tail, "m"] & \mathcal{F}(L) \end{tikzcd} \tag{10}$$

Colcombet and Petrişan also use factorization to characterize the reachable and observable subatomata of an automaton [20]:

- $\mathtt{Reach}(\mathrm{A})$ is the factorization of the only morphism between $\mathcal{I}(L)$ and A.
- $\mathtt{Obs}(\mathrm{A})$ is the factorization of the only morphism between A and $\mathcal{F}(L)$.

They use a diagram analogous to the following while proving their Lemma 2.3 [20]:

$$\begin{tikzcd} & & \mathrm{A} \arrow[drr, bend left] & & \\ \mathcal{I}(L) \arrow[urr, bend left] \arrow[r, two heads] \arrow[drr, bend right, two heads] & \mathtt{Reach}(\mathrm{A}) \arrow[ur, tail] \arrow[rr, two heads] & & \mathtt{Obs}(\mathtt{Reach}(\mathrm{A})) \arrow[r, tail] & \mathcal{F}(L) \\ & & \mathtt{Min}(L) \arrow[ur, dashed, tail, two heads] \arrow[urr, bend right, tail] & & \end{tikzcd} \tag{11}$$

This diagram will be useful to define the search space of grammatical inference.

3. Partial automata

Learning algorithms use automata that are only partially defined. In this section, we introduce the necessary concepts to work with these automata.

[2] The natural transformations in this case are the functions between the states of two automata that keep the language.

3.1. Partial functions and generalizations

We model partial functions as morphisms in the category of pointed sets. Given a set A, the notation $A_\perp$ denotes the pointed set $(A \cup \{\perp\}, \perp)$ where $\perp$ is not an element of A. We write $f : A \nrightarrow B$ for the partial function $f : A_\perp \to B_\perp$. The set of elements $a \in A$ such that $f(a) \in B$ is denoted by D_f.

As pointed set morphisms preserve the points, we have that $f(\perp) = \perp$.

Given a function $f : A \to B$, we use f^* to refer to its extension as a partial function $A \nrightarrow B$ such that for all $a \in A$, $f^*(a) = f(a)$ and $f^*(\perp) = \perp$.

Each partial function $f : A \nrightarrow B$ can be associated with a span (two morphisms with the same origin) in **Set** as follows:

$$\begin{tikzcd}
 & \mathrm{D}_f \arrow[dl, hook', "\iota"'] \arrow[dr, "\bar{f}"] & \\
A & & B
\end{tikzcd} \tag{12}$$

where ι is the inclusion. When $f : A \nrightarrow B$ and $g : B \nrightarrow C$ are composed, we obtain the following diagram:

$$\begin{tikzcd}
 & & \mathrm{D}_{g\cdot f} \arrow[dl, hook', "\iota"'] \arrow[dr, "\hat{f}"] & & \\
 & \mathrm{D}_f \arrow[dl, hook', "\iota"'] \arrow[dr, "\bar{f}"] & & \mathrm{D}_g \arrow[dl, hook', "\iota"'] \arrow[dr, "g"] & \\
A & & B & & C
\end{tikzcd} \tag{13}$$

Note that $\mathrm{D}_f \xleftarrow{\iota} \mathrm{D}_{g\cdot f} \xrightarrow{\hat{f}} \mathrm{D}_G$ is a pullback of $\mathrm{D}_f \xrightarrow{\bar{f}} B \xleftarrow{\iota} \mathrm{D}_G$.

To simplify the notation, the use of f in an arrow between two sets must be understood as the restriction of f to those sets. This way, the $\hat{f}$ and $\bar{f}$ in (12) and (13) will be written simply as f.

A partial function $g : A \nrightarrow B$ *generalizes* the partial function $f : A \nrightarrow B$ if for every $x \in A$ where $f(x)$ is defined, $g(x)$ is also defined and $f(x) = g(x)$. We denote this as $f \preceq g$. The corresponding diagram is:

$$\begin{tikzcd}
 & \mathrm{D}_f \arrow[dl, hook', "\iota"] \arrow[dr, "f"] \arrow[dd, hook, "\iota"] & \\
A & & B \\
 & \mathrm{D}_g \arrow[ul, hook, "\iota"] \arrow[ur, "g"] &
\end{tikzcd} \tag{14}$$

The following two lemmas present two useful properties.

Lemma 1. *If $f : A \nrightarrow B$, $g : A \nrightarrow B$, and $h : A \nrightarrow B$ are three partial functions such that $f \preceq g$ and $g \preceq h$ then $f \preceq h$.*

Lemma 2. *If $f : A \nrightarrow B$, $g : A \nrightarrow B$, $h : B \nrightarrow C$, and $l : B \nrightarrow C$ are four partial functions such that $f \preceq g$ and $h \preceq l$ then $h \cdot f \preceq l \cdot g$.*

3.2. Partial automata

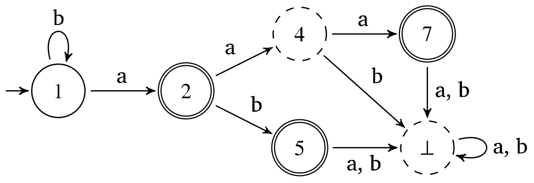
The intermediate automata generated by learning algorithms have partial transition and final functions. We formalize them in this section.

Partial automata are functors from the category of the diagram (9) to the category of sets. They can be described by the following diagram:

$$\begin{tikzcd}
\mathbf{1}_\perp \arrow[r, "q_0"] & Q_\perp \arrow[loop above, "\delta_a"] \arrow[r, "F"] & \mathbf{2}_\perp
\end{tikzcd}$$

Like before, a string $a_1 \dots a_n$ can be associated to the morphism $\triangleright a_1 \dots a_n \triangleleft$ in $\mathcal{I}_{\Sigma^*}$. This morphism is translated by A into the function $F \cdot \delta_{a_n} \cdot \dots \cdot \delta_{a_1} \cdot \vec{q}_0 : \mathbf{1} \nrightarrow \mathbf{2}$. In this case, the result is $\perp$ if any of the functions is undefined.

The fact that the set of states is $Q_\perp$ can be interpreted as the automata having a sink state. For instance, automaton Eq. (2) in Section 1.1 can be drawn as follows:



(15)

4. Merges

Given a partial automaton $\mathrm{A}(Q, \Sigma, \delta, q_0, F)$ a *merge* of A is a pair (A^M, ϕ), where A^M is another partial automaton $(Q^M, \Sigma, \delta^M, q_0^M, F^M)$ and $\phi : Q \to Q^M$ is an epimorphism[3] that when composed generalizes the initial state and the transition and final functions. That is, $\phi \cdot \vec{q}_0 \preceq \vec{q}_0^M$, $\phi \cdot \delta_a \preceq \delta_a^M \cdot \phi$ and $F \preceq F^M \cdot \phi$.

The diagram corresponding to $\phi \cdot \vec{q}_0 \preceq \vec{q}_0^M$ is

$$\begin{tikzcd}
 & \mathrm{D}_{\vec{q}_0} \arrow[dl, hook', "\iota"] \arrow[r, "\vec{q}_0"] \arrow[dd, hook, "\iota"] & Q \arrow[dd, two heads, "\phi"] \\
\mathbf{1} & & \\
 & \mathrm{D}_{\vec{q}_0^M} \arrow[ul, hook, "\iota"'] \arrow[r, "\vec{q}_0^M"] & Q^M
\end{tikzcd} \tag{16}$$

The diagram for $\phi \cdot \delta_a \preceq \delta_a^M \cdot \phi$ can be obtained from (13) and (14):

$$\begin{tikzcd}
 & & \mathrm{D}_{\phi\cdot\delta_a} \arrow[dl, hook', "\iota"] \arrow[dr, "\delta_a"] \arrow[dddd, hook, "\iota"] & & \\
 & \mathrm{D}_{\delta_a} \arrow[dl, hook', "\iota"] \arrow[d, "\delta_a"] & & \mathrm{D}_\phi \arrow[dll, hook', "\iota"] \arrow[dr, two heads, "\phi"] & \\
Q & Q & & Q^M & Q^M \\
 & \mathrm{D}_\phi \arrow[ul, hook, "\iota"] \arrow[urr, two heads, "\phi"] & & \mathrm{D}_{\delta_a^M} \arrow[u, hook, "\iota"] \arrow[ur, "\delta_a^M"] & \\
 & & \mathrm{D}_{\delta_a^M\cdot\phi} \arrow[ul, hook, "\iota"] \arrow[ur, two heads, "\phi"] & &
\end{tikzcd}$$

This can be simplified since ϕ is total, $\mathrm{D}_\phi = Q$, and $\mathrm{D}_{\phi\cdot\delta_a} = \mathrm{D}_{\delta_a}$:

$$\begin{tikzcd}
\mathrm{D}_{\delta_a} \arrow[r, "\delta_a"] \arrow[d, hook, "\iota"] \arrow[dd, hook, bend right, "\iota"'] & Q \arrow[dr, two heads, "\phi"] & \\
Q \arrow[r, two heads, "\phi"] & Q^M & Q^M \\
\mathrm{D}_{\delta_a^M\cdot\phi} \arrow[u, hook, "\iota"] \arrow[r, two heads, "\phi"] & \mathrm{D}_{\delta_a^M} \arrow[u, hook, "\iota"] \arrow[ur, "\delta_a^M"] &
\end{tikzcd}$$

We can see that ϕ preserves the transitions by the subdiagram:

$$\begin{tikzcd}
\mathrm{D}_{\delta_a} \arrow[r, "\delta_a"] \arrow[d, "\phi"] & Q \arrow[d, two heads, "\phi"] \\
\mathrm{D}_{\delta_a^M} \arrow[r, "\delta_a^M"] & Q^M
\end{tikzcd} \tag{17}$$

This diagram can be extended to strings:

$$\begin{tikzcd}
\mathrm{D}_{\delta_w} \arrow[r, "\delta_w"] \arrow[d, "\phi"] & Q \arrow[d, two heads, "\phi"] \\
\mathrm{D}_{\delta_w^M} \arrow[r, "\delta_w^M"] & Q^M
\end{tikzcd} \tag{18}$$

[3] An *epimorphism* in **Set** is a surjective function.

A similar process gives us the following diagram for the condition $F \preceq F^M \cdot \phi$:

$$\begin{tikzcd}
\mathrm{D}_F \arrow[d, "\phi"] \arrow[dr, "F"] & \\
 & \mathbf{2} \\
\mathrm{D}_{F^M} \arrow[ur, "F^M"'] &
\end{tikzcd} \tag{19}$$

When the automaton is total, diagrams (16) to (19) can be redrawn taking into account that the domains are known and we arrive at:

$$\begin{tikzcd}
 & Q \arrow[dd, "\phi"] & Q \arrow[r, "\delta_a"] \arrow[dd, "\phi"] & Q \arrow[dd, "\phi"] & Q \arrow[dd, "\phi"] \arrow[dr, "F"] & \\
\mathbf{1} \arrow[ur, "\vec{q}_0"] \arrow[dr, "\vec{q}_0^M"'] & & & & & \mathbf{2} \\
 & Q^M & Q^M \arrow[r, "\delta_a^M"] & Q^M & Q^M \arrow[ur, "F^M"'] &
\end{tikzcd} \tag{20}$$

Therefore, the merges of total automata are natural transformations.

As the merge is determined by ϕ we usually refer to it simply as ϕ.

Two states p and q of A are *merged* by ϕ if $\phi(p) = \phi(q)$. A state q is *reachable* from p if there is a string w such that $\delta_w(p) = q$. The states reachable from two merged states are also merged, as shown in this lemma.

Lemma 3. *Given two states p and q that are merged by ϕ, if p' and q' are reachable from p and q with the same string w then $\phi(p') = \phi(q')$.*

Proof. We can find the pairs of states in the domain of δ_w that are merged by ϕ using this pullback:[4]

$$\begin{tikzcd}
M \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathrm{D}_{\delta_w} \arrow[d, two heads, "\phi"] \\
\mathrm{D}_{\delta_w} \arrow[r, two heads, "\phi"] & \mathrm{D}_{\delta_w^M}
\end{tikzcd}$$

We can add two copies of Diagram (18):

$$\begin{tikzcd}
M \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathrm{D}_{\delta_w} \arrow[d, two heads, "\phi"] \arrow[r, "\delta_w"] & Q \arrow[dd, two heads, "\phi"] \\
\mathrm{D}_{\delta_w} \arrow[r, two heads, "\phi"] \arrow[d, "\delta_w"] & \mathrm{D}_{\delta_w^M} \arrow[dr, "\delta_w^M"] & \\
Q \arrow[rr, two heads, "\phi"] & & Q^M
\end{tikzcd}$$

Let m be any element of M. The two states $\pi_1(m)$ and $\pi_2(m)$ fulfil $\phi(\pi_1(m)) = \phi(\pi_2(m))$, and $\phi(\delta_w(\pi_1(m))) = \phi(\delta_w(\pi_2(m)))$. By construction, there is an m with $\pi_1(m) = p$ and $\pi_2(m) = q$, and the result follows. □

Now consider any merge different from the identity. This merge can be decomposed in a sequence of what can be called direct merges. A merge ϕ is a *direct merge* of p and q if, for every two states p' and q' that are merged by ϕ, there is a string w such that either $(p', q') = (\delta_w(p), \delta_w(q))$ or $(p', q') = (\delta_w(q), \delta_w(p))$. So a direct merge is the minimal merge that merges p and q.

The following theorem shows that every merge can be decomposed in a series of direct merges.

Theorem 4. *Given a partial automaton $A(Q, \Sigma, \delta, q_0, F)$ and a merge ϕ, if p and q are two states merged by ϕ, there is a merge ψ such that $\phi = \psi \cdot \zeta_{pq}$, where ζ_{pq} is a direct merge of p and q.*

[4] Throughout the paper, we use π for the morphisms that project product-like objects.

Proof. Consider those strings w such that $\delta_p(w)$ and $\delta_q(w)$ are both defined. They can be found with this pullback:

$$\begin{tikzcd}
V \arrow[r, "\pi_1"] \arrow[d, "\pi_2"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathrm{D}_{\delta_p} \arrow[d, hook, "\iota"] \\
\mathrm{D}_{\delta_q} \arrow[r, hook, "\iota"] & \Sigma^*
\end{tikzcd} \tag{21}$$

Let (Q^D, ζ_{pq}) be a coequalizer of $\delta_p \cdot \pi_1$ and $\delta_q \cdot \pi_2$ (a coequalizer is a function such that $\zeta_{pq} \cdot \delta_p \cdot \pi_1 = \zeta_{pq} \cdot \delta_q \cdot \pi_2$):

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p \cdot \pi_1"] \arrow[r, shift right, "\delta_q \cdot \pi_2"'] & Q \arrow[r, two heads, "\zeta_{pq}"] & Q^D
\end{tikzcd} \tag{22}$$

We prove that $(\mathrm{A}^D, \zeta_{pq})$ is a merge by writing the components $(Q^D, \Sigma, \delta^D, q_0^D, F^D)$ of A^D and proving that they fulfil diagrams from (16) to (19).

If we define $q_0^D = \zeta_{pq}(q_0)$ we have trivially that:

$$\begin{tikzcd}
 & \mathrm{D}_{\vec{q}_0} \arrow[r, "\vec{q}_0"] \arrow[dl, hook, "\iota"'] \arrow[d, hook, "\iota"] & Q \arrow[d, "\zeta_{pq}"] \\
\mathbf{1} & \mathrm{D}_{\vec{q}_0^D} \arrow[l, hook, "\iota"] \arrow[r, "\vec{q}_0^D"] & Q^D
\end{tikzcd}$$

Define δ_a^D as the unique morphism that makes this diagram commute:

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[d, "\delta_a"] & Q^D \arrow[d, dashed, "\delta_a^D"] \\
 & Q_\perp \arrow[r, two heads, "\zeta_{pq}^*"] & Q_\perp^D
\end{tikzcd}$$

To check (17), we can draw the pullbacks corresponding to δ_a and δ_a^D:

$$\begin{tikzcd}
 & V \arrow[d, shift right, "\delta_q"'] \arrow[d, shift left, "\delta_p"] & & \\
\mathrm{D}_{\delta_a} \arrow[r, hook, "\iota"] \arrow[d, "\delta_a"'] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[rrr, dashed, bend left=60, "h"] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[d, "\delta_a"] & Q^D \arrow[d, "\delta_a^D"] \arrow[dr, phantom, "\llcorner", very near start] & \mathrm{D}_{\delta_a^D} \arrow[l, hook', "\iota"'] \arrow[d, "\delta_a^D"] \\
Q \arrow[r, hook, "\iota"] \arrow[rrr, bend right, "\zeta_{pq}"'] & Q_\perp \arrow[r, two heads, "\zeta_{pq}^*"] & Q_\perp^D & Q^D \arrow[l, hook', "\iota"']
\end{tikzcd}$$

where h exists because $\zeta_{pq} \cdot \delta_a$ and $\zeta_{pq} \cdot \iota$ are part of a commuting square. The relevant part of the diagram is:

$$\begin{tikzcd}
\mathrm{D}_{\delta_a} \arrow[r, "\delta_a"] \arrow[dr, dashed, "h"] \arrow[ddr, bend right, "\zeta_{pq} \cdot \iota"'] & Q \arrow[dr, "\zeta_{pq}"] & \\
 & \mathrm{D}_{\delta_a^D} \arrow[r, "\delta_a^D"] \arrow[d, hook, "\iota"'] \arrow[dr, phantom, "\lrcorner", very near start] & Q^D \arrow[d, hook, "\iota"] \\
 & Q^D \arrow[r, "\delta_a^D"] & Q_\perp^D
\end{tikzcd}$$

Where you can see the commuting square and that (17) directly derives from the upper part.

Define F^D as the unique morphism that makes this diagram commute:

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[dr, "F"] & Q^D \arrow[d, dashed, "F^D"] \\
 & & \mathbf{2}_\perp
\end{tikzcd}$$

Again, we can check (19) by drawing the pullbacks:

$$\begin{tikzcd}
& V \arrow[d, shift right, "\delta_q"'] \arrow[d, shift left, "\delta_p"] & & \\
\mathrm{D}_F \arrow[r, hook, "\iota"] \arrow[d, "F"] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[rrr, dashed, bend left=60, "h"] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[dr, "F"] & Q^D \arrow[d, "F^D"] & \mathrm{D}_{F^D} \arrow[l, hook', "\iota"'] \arrow[d, "F^D"] \arrow[dl, phantom, "\llcorner", very near start] \\
Q \arrow[rr, "F"] \arrow[rrr, bend right, "\zeta_{pq}"'] & & \mathbf{2}_\perp & Q^D \arrow[l, "F^D"]
\end{tikzcd}$$

Finally, by Lemma 3 $\phi(\delta_p(w)) = \phi(\delta_q(w))$ for all $w \in V$, therefore, there must be a unique ψ that composed with ζ_{pq} makes the following commute:

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[dr, "\phi"'] & Q^D \arrow[d, two heads, "\psi"] \\
& & Q^M
\end{tikzcd}$$

ζ_{pq} is epic since it is a coequalizer and ψ is epic because ϕ is epic. □

Since coequalizers are isomorphic, we can talk about the direct merge of p and q without a problem.

Direct merges commute in the following sense. The result of merging the states p and q and then (the images of) the states r and s is isomorphic to the result of merging r and s and then (the images of) p and q. Graphically:

$$\begin{tikzcd}
\mathrm{A} \arrow[r, "\zeta_{pq}"] \arrow[dr, "\zeta_{rs}"'] & \mathrm{A}^D \arrow[r, "\zeta_{r's'}"] & \mathrm{A}^R \arrow[d, two heads, "l"] \\
& \mathrm{A}^M \arrow[r, "\zeta_{p'q'}"] & \mathrm{A}^N
\end{tikzcd}$$

Theorem 5. *Let:*

- *$A(Q, \Sigma, \delta, q_0, F)$ be an automaton.*
- *p, q, r, and s be states of Q.*
- *p' and q' be the images of p and q after the direct merge of r and s: $p' = \zeta_{rs}(p)$, $q' = \zeta_{rs}(q)$.*
- *r' and s' be the images of r and s after the direct merge of p and q: $r' = \zeta_{pq}(r)$, $s' = \zeta_{pq}(s)$.*
- *$A^D(Q^D, \Sigma, \delta^D, q_0^D, F^D)$ be the direct merge of p and q in A.*
- *$A^R(Q^R, \Sigma, \delta^R, q_0^R, F^R)$ be the direct merge of r' and s' in A^D.*
- *$A^M(Q^M, \Sigma, \delta^M, q_0^M, F^M)$ be the direct merge of r and s in A.*
- *$A^N(Q^N, \Sigma, \delta^N, q_0^N, F^N)$ be the direct merge of p' and q' in A^N.*

Then A^R and A^N are isomorphic.

Proof. Consider the construction of the states of the automata:

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] & Q^D \arrow[r, two heads, "\zeta_{r's'}"] & Q^R \\
V' \arrow[r, shift left, "\delta_r"] \arrow[r, shift right, "\delta_s"'] & Q \arrow[r, two heads, "\zeta_{rs}"] & Q^M \arrow[r, two heads, "\zeta_{p'q'}"] & Q^N
\end{tikzcd}$$

Where V comes from the pullback in Diagram (21) and V' is analogous to V for δ_r and δ_s.

First, we prove that the following diagram commutes:

$$\begin{tikzcd}
V \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[rrd, bend right, "\zeta_{p'q'} \cdot \zeta_{rs}"'] & Q^D \arrow[r, two heads, "\zeta_{r's'}"] \arrow[rd, dashed, "h"] & Q^R \arrow[d, two heads, "l"] \\
& & & Q^N
\end{tikzcd}$$

Where h and l are unique.

We have that $\delta_{p'} = \zeta_{rs} \cdot \delta_p$ and $\delta_{q'} = \zeta_{rs} \cdot \delta_q$. Therefore:

$$\begin{aligned}
\zeta_{p'q'} \cdot \zeta_{rs} \cdot \delta_p &= \zeta_{p'q'} \cdot \delta_{p'} && \text{Equation for } \delta_{p'} \\
&= \zeta_{p'q'} \cdot \delta_{q'} && \zeta_{p'q'} \text{ is a coequalizer} \\
&= \zeta_{p'q'} \cdot \zeta_{rs} \cdot \delta_q. && \text{Equation for } \delta_{q'}
\end{aligned}$$

So, $\zeta_{p'q'} \cdot \zeta_{rs}$ coequalizes δ_p and δ_q. Therefore, h is uniquely determined since ζ_{pq} is a coequalizer.

Similarly, $\delta_{r'} = \zeta_{pq} \cdot \delta_r$ and $\delta_{s'} = \zeta_{pq} \cdot \delta_s$. Therefore:

$$\begin{aligned}
h \cdot \delta_{r'} &= h \cdot \zeta_{pq} \cdot \delta_r && \text{Equation for } \delta_{r'} \\
&= \zeta_{p'q'} \cdot \zeta_{rs} \cdot \delta_r && h \cdot \zeta_{pq} \text{ commutes with } \zeta_{p'q'} \cdot \zeta_{rs} \\
&= \zeta_{p'q'} \cdot \zeta_{rs} \cdot \delta_s && \zeta_{rs} \text{ is a coequalizer} \\
&= h \cdot \zeta_{pq} \cdot \delta_s && h \cdot \zeta_{pq} \text{ commutes with } \zeta_{p'q'} \cdot \zeta_{rs} \\
&= h \cdot \delta_{s'}. && \text{Equation for } \delta_{s'}
\end{aligned}$$

So, h coequalizes $\delta_{r'}$ and $\delta_{s'}$. Therefore, l is uniquely determined since $\zeta_{r's'}$ is a coequalizer.

A symmetric argument allows the definition of a unique $l' : Q^N \to Q^R$ confirming that they are isomorphic. This isomorphism is translated to the isomorphisms of transitions and final states as in the proof of the previous theorem. □

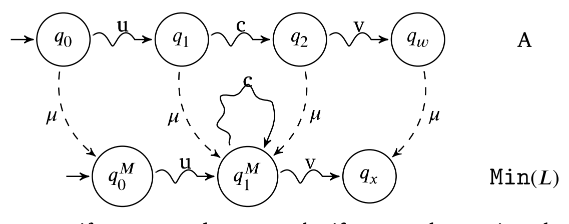
Note that $\mathtt{Min}(L)$, the minimal automaton that recognizes L, can be seen as the result of a merge of $\mathcal{I}(L)$, the initial automaton of $\mathbf{Auto}(L)$. Therefore, $\mathtt{Min}(L)$ can be produced as the result of a sequence of direct merges. This sequence can be finite.

Lemma 6. *If L is a regular language, there is a finite sequence of direct merges that when applied to $\mathcal{I}(L)$ produces $\mathtt{Min}(L)$.*

Proof. We build such a sequence in steps. In each step we have an automaton $\mathrm{A}(Q, \Sigma, \delta, q_0, F)$. In the first step, A is $\mathcal{I}(L)$. Assume that $\mathtt{Min}(L)$ is the automaton $(Q^M, \Sigma, \delta^M, q_0^M, F^M)$. Consider the merge $\mu : Q \to Q^M$ that produces $\mathtt{Min}(L)$ from A. A string w points to a state that *has to be merged* if the state $q_w = \delta_{q_0}(w)$ is different from $q_x = \delta_{q_0}(x)$ where x is the shortest string such that $\delta^M_{q_0^M}(x) = \delta^M_{q_0^M}(w)$. Note that this means that $\mu(q_x) = \mu(q_w)$, explaining why q_w "has to be merged".

Now consider the shortest such w or, if there is more than one of minimal length any of them. The next automaton is obtained as the direct merge of q_w and q_x. We claim that only a finite amount of such steps can be taken.

First, note that the path followed by w in A cannot have cycles since a cycle would allow us to find a shorter string pointing to q_w. Similarly, the path followed by w in $\mathtt{Min}(L)$ does not have a cycle. For the sake of contradiction, suppose that w can be decomposed in three parts, $w = ucv$ where c cycles in $\mathtt{Min}(L)$. The relationship between the paths traversed by w in A and $\mathtt{Min}(L)$ is:



(23)

Now, if $q_1 = q_2$ we have a cycle; if $q_1 \neq q_2$ then uc is a shorter string than w which leads to a state that needs to be merged. Either way, we have a contradiction.

Therefore, in each step, we have a new w that does not cycle, and since L is regular, there is a finite number of such strings. □

5. Samples and learning

In this section, we formalize the process of learning. First, we adapt the definitions in [25,26] of the concept of learning in the limit.

A *sample* of a language L is a pair (X, S) where $X \subseteq \Sigma^*$ is a set of strings and $S : X \to \mathbf{2}$ is the restriction of L to X: $S(x) = L(x)$.

Given two samples (X, S) and (X', S') we write $(X, S) \leq (X', S')$ if $X \subseteq X$ and for all $x \in X$, $S(x) = S'(x)$. This means that sample S' has all the information from S and possibly some more.

A *learning algorithm* has as input a sample (X, S) and produces an automaton A that is *compatible* with it. I.e. the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, "S"] \arrow[d, hook, "\iota"'] & \mathbf{2} \\ \Sigma^* \arrow[ur, "L(\mathrm{A})"'] & \end{tikzcd} \tag{24}$$

A *presentation* of a language L is a sequence of samples $\{(X_i, S_i)\}_{i=1}^{\infty}$ such that $(X_i, S_i) \leq (X_j, S_j)$ for $i \leq j$ and for all $w \in \Sigma^*$ there is an i such that $w \in X_i$. So we can interpret a presentation as a sequence of increasingly larger samples that eventually contain any string.

A learning algorithm *identifies in the limit* the class of DFAs if, for any regular language L and any presentation $\{(X_i, S_i)\}_{i=1}^{\infty}$ of L, there is a finite n such that for every (X_i, S_i) with $i \geq n$ the output of the algorithm is a DFA that recognizes L

To alleviate notation and when we do not need to consider the strings, we refer to S as the sample.

An important concept is that of *characteristic sample*. A characteristic sample for a language and a learning algorithm is a sample (CX, CS) such that the algorithm returns a representation of the language for every sample that contains it, i.e. $(X, S) \geq (\mathrm{CX}, \mathrm{CS})$. We also refer to CX as the *characteristic set*.

Identification in the limit is only possible for an algorithm if every language has a characteristic sample for that algorithm [27].

5.1. The PTA

A DFA A is a *tree* if there is a single path from the initial state to every other state and no path arriving at the initial state.

A *prefix tree acceptor* (PTA) for a sample S is a tree T that represents S and only it. This means that for every string w in the sample, $L(\mathrm{T})(w) = S(w)$, and for every string not in the sample, $L(\mathrm{T})(w) = \perp$.

One construction of the PTA is the automaton $\mathrm{T}(\hat{X}, \Sigma, \delta, \lambda, F)$ where the states $\hat{X}$ are the set of prefixes of the strings in X. The transition function δ_a is defined as $\delta_a(w) = wa$ if wa is the prefix of a string in X and $\delta_a(w) = \perp$ otherwise. The final function F is the function that is equal to S in the strings of X and equal to $\perp$ in every other string.

5.2. Merging algorithms

The algorithms we consider take an input sample and represent it as a PTA. Then, they apply a sequence of direct merges to the initial tree until no more merges are possible. The final automaton is returned as the result of the algorithm. A survey of several algorithms of this type can be found in [19].

Assume we have the following functions:

- PTA(S): returns the PTA corresponding to sample S.
- canMerge(A): returns true if two states of A can be merged.
- chooseMerge(A): returns a pair of states of A that can be merged.
- merge(A, p, q): returns the result of merging states p and q of A.

The general form of the learning algorithm is as follows:

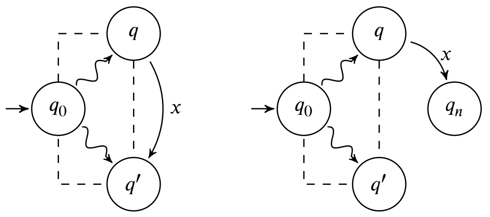
Algorithm 1: The generic merging inference algorithm.

```
Input: A sample: $S$
Output: An automaton compatible with $S$
A ← PTA($S$);
while canMerge($A$) do
    $(p, q)$ ← chooseMerge(A);
    A ← merge(A, $p$, $q$);
end
return $A$;
```

This is not intended to be efficient. For instance, there is no data structure to avoid repeatedly exploring impossible merges in canMerge. Also, there is a large overlap between canMerge and chooseMerge. Instead, the algorithm adequately represents the merges produced by concrete instantiations, that is, the merges produced by RPNI [17] are the same as those produced by Algorithm 1 with an adequate choice of chooseMerge, like in the example of Section 5.6.

Theorem 4 guarantees that if the automaton corresponding to the target language can be obtained by merging the PTA, then it can be obtained by Algorithm 1. But this crucially depends on the choice of the function chooseMerge.

It can be noted that Algorithm 1 is more general than the algorithm presented by de la Higuera et al. [28]. Their algorithm begins with a trivial automaton consisting of a single state and no transitions. The algorithm is parameterized by a function ϕ (like ours is parameterized by chooseMerge) that returns a symbol x and a pair of states (q, q') from the automaton being built. If the pair of states and the symbol define a transition that is compatible with the sample, the transition is added to the automaton. Graphically, the first case corresponds to the left diagram and the second case, to the right diagram:

 (25)

There is a correspondence between these two cases and the working of Algorithm 1. The first case corresponds to the situation in which the pair of states (q, q') can be merged. In this case, the algorithm merges the states and adds the transition from the merged state to itself. The second case corresponds to the finding that the states cannot be merged.

We can revisit the beginning of the example in Section 1.1 to compare the working of both algorithms. The initial automaton in de la Higuera's algorithm has only the initial state while Algorithm 1 would build the automaton (1). When de la Higera's algorithm discovers that it is not possible to move from the initial state with an a, it adds a new state (second case) whereas Algorithm 1 keeps the automaton intact. Next, de la Higuera's algorithm adds a new transition from the initial state to itself with a b (first case), which corresponds to the merge of states 1 and 3 that results in automaton (2). The process ends when the remaining arcs are added.

However, there is a big difference between both algorithms. De la Higuera's algorithm does not allow merging already merged states. Instead, it only considers the roots of the subtrees that have not been merged yet and tries to merge them to the consolidated states of the automaton.

5.3. The search space of grammatical inference

Dupont et al. propose a structure for the search space of Grammatical Inference [21]. They view the merging of states as the product of any equivalence relation between the states of the automaton, even when it gives rise to a nondeterministic automaton. Furthermore, they do not include the concept of partiality in the final function and they do not include the information from negative samples into the search space. Instead, they consider the negative information as a way of defining the border of the lattice that defines their search space.

In this paper, we restrict ourselves to the inference of DFAs and by including both positive and negative samples in the PTA, we arrive at a different structure. Our understanding of the search space is that it is the category that contains as objects the PTA and all the automata that can be obtained from it by considering merges of states. The arrows of the category are the merges. Let **Merge**(S) be that category. By construction this category has two properties: it is thin and the PTA is an initial object of **Merge**(S).

Each election of the function `chooseMerge` defines a subcategory of **Merge**(S) which contains the sequence of automata produced by Algorithm 1.

5.4. The sampling functor

Now let us define a functor $\mathbf{S}_{LS}$ that relates the automata that recognize a language to those that can be generated by merging the PTA obtained from a sample of that language. This functor, which we call the *sampling functor*, allows us to interpret the sample S as a probe in the category **Auto**(L) of the automata that recognize the language L. Furthermore, this will allow us to reason about the behaviour of Algorithm 1 using the automata in **Auto**(L).

Define the image of an automaton $\mathrm{A}(Q, \Sigma, \delta, q_0, F)$ by $\mathbf{S}_{LS}$ as the partial automaton $\mathrm{A}^R(Q^R, \Sigma, \delta^R, q_0^R, F^R)$ in **Merge**(S). The set of states Q^R is the set of states reached by the strings of $\hat{X}$, $Q^R = \{\delta_{q_0}(w) \mid w \in \hat{X}\}$. The transition function δ^R is the restriction of δ to Q^R: $\delta_a^R(p) = \delta_a(p)$ if $p \in Q^R$ and $\delta_a(p) \in Q^R$, $\delta_a^R(p) = \bot$, otherwise. And F^R is only defined in those states reached by strings of X: $F^R(p) = F(p)$ if there is a string $w \in X$ such that $\delta_{q_0}(w) = p$, $F^R(p) = \bot$, otherwise.

Since a morphism $f : \mathrm{A} \to \mathrm{A}^M$ in **Auto**(L) corresponds to a function $Q \to Q^M$ its image $\mathbf{S}_{LS}(f)$ can be defined simply as the restriction of f to Q^R.

To check that $\mathbf{S}_{LS}$ is well-defined, note that the image of $\mathcal{I}(L)$ (the initial element of **Auto**(L)) is the PTA and for each automaton A in **Auto**(L), there is a morphism $f : \mathcal{I}(L) \to \mathrm{A}$ that corresponds to the merge in **Merge**(S) that produces $\mathbf{S}_{LS}(\mathrm{A})$ from the PTA.

This functor allows us to connect the automata in Section 1.3 to those in Section 1.1. This is because the image of a direct merge in **Auto**(L) is a direct merge in **Merge**(S).

Lemma 7. *Let $\zeta_{pq} : A \to A^D$ be a direct merge of $A, A^D \in$ **Auto**(L). Then $\mathbf{S}_{LS}(\zeta_{pq}) : \mathbf{S}_{LS}(A) \to \mathbf{S}_{LS}(A^D)$ is a direct merge in **Merge**(S).*

Proof. First, note that as A is a total automaton, Diagram (22) can be redrawn as:

$$\begin{tikzcd}
\Sigma^* \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] & Q^D
\end{tikzcd}$$

We can combine it with (22) for A^D and add the morphisms corresponding to the functor to arrive at:

$$\begin{tikzcd}
\Sigma^* \arrow[r, shift left, "\delta_p"] \arrow[r, shift right, "\delta_q"'] & Q \arrow[r, two heads, "\zeta_{pq}"] \arrow[dd, "\mathbf{S}_{LS}"'] & Q^D \arrow[d, shift right, "\mathbf{S}_{LS}"'] \arrow[dd, dashed, bend left, "h"] \\
 & & \mathbf{S}_{LS}(Q^D) \arrow[u, hook, shift right, "l"'] \\
V \arrow[r, shift left, "\delta_p \cdot \pi_1"] \arrow[r, shift right, "\delta_q \cdot \pi_2"'] & \mathbf{S}_{LS}(Q) \arrow[r, two heads, "\zeta'_{pq}"'] \arrow[ur, "\mathbf{S}_{LS}(\zeta'_{pq})"] & Q^M \arrow[u, dashed, "l"]
\end{tikzcd}$$

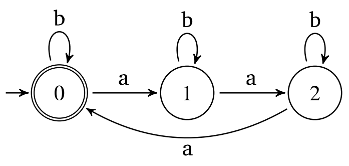
Where the inclusion of $\mathbf{S}_{LS}(Q^D)$ is a consequence of the definition of $\mathbf{S}_{LS}$ and the morphisms h and l exist because ζ_{pq} and ζ'_{pq} are coequalizers.

From there, we conclude that $\mathbf{S}_{LS}(Q^D)$ and Q^M are isomorphic. $\square$

And from Theorem 4 we arrive at the following corollary.

Corollary 8. *The functor $\mathbf{S}_{LS}$ preserves epimorphims.*

Unfortunately, we cannot use the same idea to define a functor in the other direction. To see why, consider learning the language L of the strings with a number of a's that is a multiple of three. The corresponding automaton is:



(26)

For any sample with at least one string with $3n + 1$ a's and a string with $3m + 2$ a's we can join in the PTA the nodes corresponding to the two longest such strings (one from the $3n + 1$ group, one from the $3m + 2$) and the resulting automaton is not related to any in **Merge**(L). This also means that an algorithm that would use this strategy for `chooseMerge` would fail to identify in the limit this language.

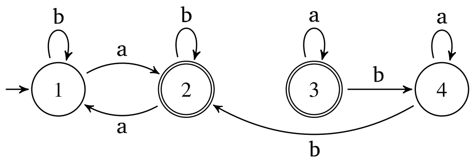
Therefore, we are interested in restricting the choices of `chooseMerge` to those merges that are the image of a merge by $\mathbf{S}_{LS}$.

To do that, let us consider the subcategory of **Merge**(S) generated by $\mathbf{S}_{LS}$. Let us call it **SM**(S).

Note that by construction the image of the automaton A is built using only reachable states, therefore we have the following lemma:

Lemma 9. *For every automaton A in **Auto**(L), $\mathbf{S}_{LS}(\mathrm{A}) = \mathbf{S}_{LS}(\mathtt{Reach}(\mathrm{A}))$.*

This lemma tells us that when moving from the category of all the automata recognizing the language L to the category of the automata derived from the PTA using merges, we need to consider only the reachable parts of the automata. For instance, let us return to our initial example from Section 1.1. An automaton that recognizes strings with an odd number of a's is the following:



(27)

And since states 3 and 4 are not reachable, the image of (27) by $\mathbf{S}_{LS}$ is (4).

Now we can conclude the following:

Theorem 10. *The automata $\mathbf{S}_{LS}(\mathcal{I}(L))$ and $\mathbf{S}_{LS}(\mathtt{Min}(L))$ are initial and final objects of **SM**(S).*

Proof. As noted above, the image of $\mathcal{I}(L)$ is the PTA, which is initial in **Merge**(S), therefore it is also initial in **SM**(S).

Let A be an automaton in **SM**(S) and A′ and automaton in **Auto**(L) such that $\mathbf{S}_{LS}(\mathrm{A}') = \mathrm{A}$. By Lemma 9, $\mathbf{S}_{LS}(\mathrm{A}') = \mathbf{S}_{LS}(\mathtt{Reach}(\mathrm{A}'))$, and by Diagram (11) there is an epimorphism from $\mathtt{Reach}(\mathrm{A}')$ to $\mathtt{Min}(L)$. Corollary 8 implies that there is an epimorphism from A to $\mathbf{S}_{LS}(\mathtt{Min}(L))$. And this morphism is unique since **Merge**(S) is thin. $\square$

Returning to the example in Section 1.1, this means that every automaton that can be built using the sampling functor on an automaton that recognizes the language of strings with an odd number of a's will be the result of merging the states of (1) and from every such automaton there will be a sequence of merges that leads to automaton (4).

From here we can derive the conditions for Algorithm 1 to produce a correct hypothesis.

Corollary 11. *If the sample S is such that $\mathbf{S}_{LS}(\mathtt{Min}(L)) \simeq \mathtt{Min}(L)$ and* `chooseMerge` *always produces a merge that results in an automaton in **SM**(S), then Algorithm 1 produces an automaton isomorphic to $\mathtt{Min}(L)$.*

We can relate this corollary to the example in Section 1.1. There, the order of the merges always produced an automaton that was the image of an automaton of $\mathbf{Auto}(L)$. Therefore, the minimal automaton for the language (automaton (8)) is isomorphic to its image by the sampling functor (automaton (4)), which is the product of the algorithm.

5.5. Another interpretation of learning in the limit

Consider two samples (X, S) and (X', S') where $X' \subseteq X$. We can define the sampling functor $\mathbf{S}_{SS'}$ between $\mathbf{SM}(S)$ and $\mathbf{SM}(S')$ in a way analogous to the definition of $\mathbf{S}_{LS}$. The definition of the image of an automaton $\mathbf{S}_{SS'}(\mathrm{A})$ is again the result of keeping the states reached by the elements of $\hat{X}$ and the image of a morphism is the restriction of that morphism to the new states.

With this definition, the composition of the sampling functors is simply $\mathbf{S}_{S'S''} \cdot \mathbf{S}_{SS'} = \mathbf{S}_{SS'}$ when the corresponding sets of strings are such $X'' \subseteq X' \subseteq X$. Furthermore, the functor $\mathbf{S}_{SS}$ is the identity for $\mathbf{SM}(S)$. Therefore, the different $\mathbf{SM}(S)$ together with the sampling functors form a category.

This category gives another interpretation of the process of identification in the limit. The different $\mathbf{SM}(S)$ are progressively better approximations to the subcategory of $\mathbf{Auto}(L)$ that is generated by the merges. And this subcategory can somehow be considered as $\mathbf{SM}(L)$.

5.6. Data-independent algorithms

There is a class of algorithms, like RPNI [17] that try to merge the states following a predefined order. These algorithms are called *data-independent* [28]. We can characterize these algorithms as having a function `chooseMerge` parameterized by a predefined list of pairs of strings. Each pair of strings is used to define the pair of states reached by the strings. The first such pair of states that can be merged is returned by `chooseMerge`.

Let $\mathrm{P} = \{(v_i, w_i)\}_{i=1}^{\infty}$ be a sequence of pairs of strings such that for every pair $(v, w) \in \Sigma^* \times \Sigma^*$ there is an i with $(v, w) = (v_i, w_i)$ or $(v, w) = (v_i, w_i)$. The data-independent version of `chooseMerge` is

Algorithm 2: The data-independent version of `chooseMerge`.

```
Input: An automaton A and a sequence P = $\{(v_i, w_i)\}_{i=1}^{\infty}$ of pairs
Output: Two states $(p, q)$ of A to be merged
$i \leftarrow 1$;
while True do
    $p \leftarrow \delta_{q_0}(v_i)$;
    $q \leftarrow \delta_{q_0}(w_i)$;
    if $p \neq q \wedge$ compatible$(A, p, q)$ then
        return $(p, q)$;
    end
    $i \leftarrow i + 1$;
end
```

where `compatible` is a function that checks that the states can be merged in the automata.

As an example, consider RPNI. For the alphabet $\Sigma = \{a, b\}$, the sequence is $\mathrm{P} = \{(\lambda, a), (\lambda, b), (a, b), (\lambda, aa), (a, aa), (b, aa), (\lambda, ab), (a, ab), (b, ab), \ldots\}$. So, the order in which merges will be tried in the case of the samples seen in Section 1.1 is $\{(\lambda, a), (\lambda, b), (a, b), (\lambda, aa), (a, aa), (b, aa), (\lambda, ab), (a, ab), (b, ab), \ldots\}$.

Note that our definition of data-independent algorithm is more general than the one in [28] since they have a different version of the base algorithm. Our version allows more orders to be used. For instance, a possible order would first check the merge of the states reached by a and b and after that, the merge of the state reached by a with the initial state. This order is not allowed in [28].

DFAs are learnable in the limit by data-independent algorithms without regard to the sequence of strings.

Theorem 12. *DFAs are learnable in the limit by Algorithm 1 for any P when Algorithm 2 is chosen for* `chooseMerge`.

Proof. Let $\mathtt{Min}(L)$ be the automaton $(Q^M, \Sigma, \delta^M, q_0^M, F^M)$. We build a characteristic set CX such that Algorithm 1 generates an automaton isomorphic to $\mathtt{Min}(L)$ when used in any sample that includes CX.

The set has two parts that correspond to the conditions of Corollary 11. To ensure that the image of $\mathtt{Min}(\mathrm{L})$ is isomorphic to it, we include a string for each state in Q^M and each edge of $\mathtt{Min}(\mathrm{L})$. We also introduce strings that prevent the merges outside $\mathbf{SM}(S)$ to ensure that no merge ends outside it.

Then, we can write CX as

$$\mathrm{CX} = \{\sigma_M(p) \mid p \in Q^M\} \cup \{\sigma_M(p)a \mid p \in Q^M \wedge a \in \Sigma\} \cup D(\mathcal{I}(L), \mathrm{P}). \quad (28)$$

Here, $\sigma_M(p)$ is the shortest string x such that $\delta^M_{q_0}(x) = p$ and D is a function that produces the strings that avoid "bad merges".

The value of D is computed by essentially simulating Algorithm 1 in the automata in $\mathbf{Auto}(L)$. When the simulation finds that the algorithm would make an incorrect merge, a pair of strings is added to prevent it.

The value of $D(\mathrm{A}, P)$ is the empty set for any A isomorphic to $\mathtt{Min}(L)$ and every P.

For an automaton $\mathrm{A}(Q, \Sigma, \delta, q_0, F)$ nonisomorphic to $\mathtt{Min}(L)$ the value of $D(\mathrm{A}, P)$ depends on (v_1, w_1). Let $p = \delta_{q_0}(v_1)$, $q = \delta_{q_0}(w_1)$ and $P' = \{(v_i, w_i)\}_{i=2}^{\infty}$:

- If $p = q$ there is no need to merge, therefore $D(\mathrm{A}, P) = D(\mathrm{A}, P')$.
- If $p \neq q$ there are two possibilities:
  - $\delta^M_{q_0^M}(v_1) = \delta^M_{q_0^M}(w_1)$ and then the two states can be merged, so we perform the direct merge of p and q, and we get $D(\mathrm{A}, P) = D(\zeta_{pq}(\mathrm{A}), P')$.
  - $\delta^M_{q_0^M}(v_1) \neq \delta^M_{q_0^M}(w_1)$ and then the two states cannot be merged. Let x and y be two strings such that $L(v_1 x) \neq L(w_1 x)$. Then $D(\mathrm{A}, P) = \{v_1 x, w_1 x\} \cup D(\mathrm{A}, P')$.

In summary, for A nonisomorphic to $\mathtt{Min}(L)$:

$$D(A, P) = \begin{cases} D(A, P') & \text{if } p = q, \\ D(\zeta_{pq}, P') & \text{if } p \neq q \text{ and } \delta^M_{q_0^M}(v_1) = \delta^M_{q_0^M}(w_1), \\ \{v_1 x, w_1 x\} \cup D(A, P') & \text{otherwise.} \end{cases}$$

D is well-defined: there is a finite sequence of merges that leads the automaton to $\mathtt{Min}(L)$ (Lemma 6) and it is indifferent where this sequence appears (Theorem 5), therefore the base case of D will be eventually reached.

To see that Algorithm 1 produces the correct result in every sample that contains CX, note that the sequence of automata traversed by the algorithm is exactly the image by the sampling functor of the sequence of automata traversed in constructing $D(\mathcal{I}(L), P)$. $\square$

Corollary 13. *RPNI identifies in the limit the class of DFAs.*

5.7. Data-dependent algorithms

In general, `chooseMerge` will examine A and decide what states to merge in a variety of ways. Some general heuristics are employed, one of which is that it is better to try to merge states near the initial state. One way to capture this idea is to consider that when the sample grows enough the number of candidates that can be merged is limited to a finite set. That means that when the sample is large enough there is a region around the initial state where merges happen.

If the set of states that `chooseMerge` can produce is finite, it is possible to construct a characteristic set that contains a counterexample for each bad merge. This idea is formalized in the following theorem where the notation $\sigma_A(p)$ corresponds to the shortest string w such that $\delta_{q_0}(w) = p$.

Theorem 14. *If the version of* `chooseMerge` *in Algorithm 1 is such that for every regular language L there is a sample M such that the set*

$$F = \bigcup_{A \in \mathbf{Auto}(L)} \bigcup_{M \subseteq S \subseteq \Sigma^*} \{(\sigma_A(p), \sigma_A(q)) \mid (p,q) = \texttt{chooseMerge}(\boldsymbol{S}_{LS}(A))\}$$

is finite, then Algorithm 1 identifies DFAs in the limit.

Proof. Like in the proof of Theorem 12, let $\texttt{Min}(L)$ be the automaton $(Q^M, \Sigma, \delta^M, q_0^M, F^M)$. Also, like we did in the proof of Theorem 12, we build a characteristic set CX. The set has three parts: the M from the hypothesis to ensure the finiteness of F; the strings needed to reach the states of Q^M; and the strings needed to avoid those merges in F that may fall outside $\mathbf{SM}(S)$.

The value of CX is then

$$\mathrm{CX} = M \cup \{\sigma_M(p) \mid p \in Q^M\} \cup \{\sigma_M(p)a \mid p \in Q^M \wedge a \in \Sigma\} \cup D. \quad (29)$$

Here, the set D is derived from F by collecting one pair of strings of the form $\sigma_A(p)x$ and $\sigma_A(q)x$ for every $(\sigma_A(p), \sigma_A(q))$ in F so that $L(\sigma_A(p)x) \neq L(\sigma_A(p)x)$. As F is finite, so is D and therefore CX.

Like in the case of Theorem 12, the set CX is a characteristic sample so DFAs are identified in the limit. □

This theorem can be used to prove that the *Blue Fringe* algorithm identifies DFAs in the limit. This algorithm was first presented in [22]. The behaviour of `chooseMerge` for this algorithm is to find a set of *red states*. These are found by first marking q_0 as red. Then iteratively each neighbour of a red state that cannot be merged to a red state is also marked red. When all red states are merged the set of states that are not red but are neighbours to a red state are marked blue. Finally, the states returned are the pair formed by a red and a blue state that maximizes the total number of states merged.

Once the sample is large enough, there can be no more red states than those in $\texttt{Min}(L)$, therefore the set of possible outcomes of `chooseMerge` is finite and we have that Blue Fringe identifies DFAs in the limit.

Corollary 15. *The Blue Fringe algorithm identifies in the limit the class of DFAs.*

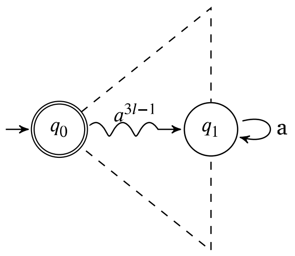
A similar argument can be used to prove that the data-dependent algorithm presented in [28] identifies DFAs in the limit.

It is interesting to note that the finiteness of F is sufficient but not necessary. We can see that with the following (contrived) example. Imagine a version of `chooseMerge` that works as follows: if all the states in the automaton are accepting, it returns the two states further from the initial state; if not all states are accepting, it works like RPNI. When using this version of `chooseMerge`, Algorithm 1 behaves like RPNI for most languages, so it identifies them. The only exception is Σ^*, but for any sample of it, there is no problem in merging states in any order, so it also identifies it. Finally, note that as the states chosen are those further from the initial state, the set F is not finite since larger samples produce longer strings to reach those states.

5.8. The case of EDSM

Another algorithm presented in [22] is EDSM (short for Evidence Driven State Merging). The way `chooseMerge` works in this algorithm is by considering all possible direct merges and scoring them. Then the merge with the highest score is returned. The score of a merge is the number of states that are merged or, equivalently, the difference in the sizes of the original and the merged automata. This algorithm performs well in practice. Unfortunately, it does not fulfil the hypotheses of Theorem 14. In this case, it is because EDSM fails to identify the regular languages in the limit.

To see why that is the case, consider again the language of strings with a number of a's that is multiple of three and whose automaton is (26). Let us assume that there were a characteristic sample (CX, CS). Let s be the number of elements of CX. Note that the score of merging any two states in the PTA is going to be less than s. Now create a new sample with all the strings from CX plus all the strings of the form a^{3k+1} and a^{3k+2} for every k from 1 to a large enough value. The PTA for this new sample has a shape similar to that in Box I, so there is a part corresponding to all the samples in CX and a long tail of states reached by the additional strings of a's. After an initial part of length l every three such states contain one with no information and two rejecting. It is always possible to merge the first and the second of those states giving rise to:



(31)

which precludes finding the correct automaton.

The score of this merge will be greater than s if the number of added strings is large enough, so it will be preferred over any merge involving the states in the PTA corresponding to CX. It could be possible that there were merges with higher scores, but all of them will involve the newly added strings and will produce the same incorrect loop.

Therefore, there is no characteristic sample for this language and EDSM fails to identify in the limit the class of DFAs.

6. Conclusions

Category Theory is a powerful tool to analyse from a theoretical perspective the functioning of inference algorithms. It has been successfully used in the analysis of query-based learning algorithms and we have applied it in this work to state merging algorithms for DFA inference. One of the reasons for this success is that it provides insight and new methods for proving aspects like the convergence of algorithms. The work presented here can help in the development of new algorithms in two main aspects. First, it explains the search space that the algorithms must explore. Second, the criteria derived for the convergence will ease the proof of convergence of those algorithms.

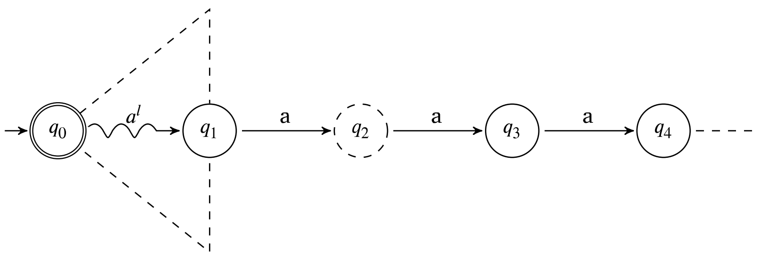
There are still different aspects that need to be addressed in future work. Perhaps the most obvious one is that the high level of abstraction about the nature of the algorithms makes it difficult to extract conclusions about the computational costs of running them. Another pending aspect is the viability of using these techniques on other types of automata, especially subsequential transducers and probabilistic automata. Also, from a more formal perspective, it would have been nicer if the conditions for convergence could be used for the characterization of convergent algorithms.

CRediT authorship contribution statement

Juan Miguel Vilar: Conceptualization, Investigation, Project administration, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.



(30)

Box I.

Data availability

No data was used for the research described in the article.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, second ed., Addison Wesley, 2006.
[2] R.D. Graham, P.C. Johnson, Finite state machine parsing for internet protocols: Faster than you think, in: Proceedings - IEEE Symposium on Security and Privacy, vol. 2014-January, Institute of Electrical and Electronics Engineers Inc., 2014, pp. 185–190, http://dx.doi.org/10.1109/SPW.2014.34.
[3] L. Vespa, N. Weng, Deterministic finite automata characterization and optimization for scalable pattern matching, Trans. Archit. Code Optim. 8 (2011) http://dx.doi.org/10.1145/1952998.1953002.
[4] S. Agostinelli, F. Chiariello, F.M. Maggi, A. Marrella, F. Patrizi, Process mining meets model learning: Discovering deterministic finite state automata from event logs for business process analysis, Inf. Syst. 114 (2023) 102180, http://dx.doi.org/10.1016/J.IS.2023.102180.
[5] Z. Li, H. Derksen, J. Gryak, C. Jiang, Z. Gao, W. Zhang, H. Ghanbari, P. Gunaratne, K. Najarian, Prediction of cardiac arrhythmia using deterministic probabilistic finite-state automata, Biomed. Signal Process. Control 63 (2021) 102200, http://dx.doi.org/10.1016/J.BSPC.2020.102200.
[6] K.S. Allen, D.R. Hood, J. Cummins, S. Kasturi, E.A. Mendonca, J.R. Vest, Natural language processing-driven state machines to extract social factors from unstructured clinical documentation, JAMIA Open 6 (2023) 24, http://dx.doi.org/10.1093/jamiaopen/ooad024.
[7] E. Muškardin, B.K. Aichernig, I. Pill, M. Tappler, Learning finite state models from recurrent neural networks, Lecture Notes in Comput. Sci. 13274 LNCS (2022) 229–248, http://dx.doi.org/10.1007/978-3-031-07727-2_13/TABLES/4.
[8] W. Merrill, Sequential neural networks as automata, in: Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges, Association for Computational Linguistics, Stroudsburg, PA, USA, 2019, pp. 1–13, http://dx.doi.org/10.18653/v1/W19-3901.
[9] J.S. Abbasi, F. Bashir, K.N. Qureshi, M.N. ul Islam, G. Jeon, Deep learning-based feature extraction and optimizing pattern matching for intrusion detection using finite state machine, Comput. Electr. Eng. (2021) http://dx.doi.org/10.1016/j.compeleceng.2021.107094.
[10] H. Urbat, L. Schröder, Automata Learning: An Algebraic Approach, in: ACM International Conference Proceeding Series, 2020, pp. 900–914, http://dx.doi.org/10.1145/3373718.3394775, arXiv:1911.00874.
[11] B. Jacobs, A. Silva, Automata learning: A categorical perspective, Lecture Notes in Comput. Sci. 8464 LNCS (2014) 384–406, http://dx.doi.org/10.1007/978-3-319-06880-0_20.
[12] G. van Heerdt, M. Sammartino, A. Silva, Learning automata with side-effects, Lecture Notes in Comput. Sci. 12094 LNCS (2020) 68–89, http://dx.doi.org/10.1007/978-3-030-57201-3_5.
[13] G. van Heerdt, T. Kappé, J. Rot, M. Sammartino, A. Silva, A categorical framework for learning generalised tree automata, Lecture Notes in Comput. Sci. 13225 LNCS (2022) 67–87, http://dx.doi.org/10.1007/978-3-031-10736-8_4.
[14] T. Colcombet, D. Petrişan, R. Stabile, Learning automata and transducers: A categorical approach, in: C. Baier, J. Goubault-Larrecq (Eds.), 29th EACSL Annual Conference on Computer Science Logic, CSL 2021, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 183, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 15:1–15:17, http://dx.doi.org/10.4230/LIPIcs.CSL.2021.15.
[15] D. Angluin, Learning regular sets from queries and counterexamples, Inform. and Comput. 75 (2) (1987) 87–106, http://dx.doi.org/10.1016/0890-5401(87)90052-6.
[16] K.J. Lang, Random DFA's can be approximately learned from sparse uniform examples, in: Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory, 1992, pp. 45–52, http://dx.doi.org/10.1145/130385.130390.
[17] J. Oncina, P. García, Identifying regular languages in polynomial time, in: Adv. Struct. Syntactic Pattern Recognit., World Scientific, 1993, pp. 99–108, http://dx.doi.org/10.1142/9789812797919_0007.
[18] M. Bugalho, A.L. Oliveira, Inference of regular languages using state merging algorithms with search, Pattern Recognit. 38 (9) (2005) 1457–1467, http://dx.doi.org/10.1016/J.PATCOG.2004.03.027.
[19] C. Tîrnăucă, A survey of state merging strategies for DFA identification in the limit, Triangle (2018) 121, http://dx.doi.org/10.17345/triangle8.121-136.
[20] T. Colcombet, D. Petrişan, Automata minimization: A functorial approach, Log. Methods Comput. Sci. 16 (1) (2020) 32:1–32:28, http://dx.doi.org/10.23638/LMCS-16(1:4)2020.
[21] P. Dupont, L. Miclet, E. Vidal, What is the search space of the regular inference? Lecture Notes in Comput. Sci. 862 LNAI (January 2006) (1994) 25–37, http://dx.doi.org/10.1007/3-540-58473-0_134.
[22] K.J. Lang, B.A. Pearlmutter, R.A. Price, Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm, Lecture Notes in Comput. Sci. 1433 (1998) 1–12, http://dx.doi.org/10.1007/bfb0054059.
[23] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, third ed., Pearson Education, 2007.
[24] M. Barr, C. Wells, Category Theory for Computing Science, 1998, p. 556, URL http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf.
[25] E.M. Gold, Language identification in the limit, Inf. Control 10 (5) (1967) 447–474, http://dx.doi.org/10.1016/S0019-9958(67)91165-5.
[26] C. De la Higuera, Grammatical Inference: Learning Automata and Grammars, Cambridge University Press, 2010.
[27] E.M. Gold, Complexity of automaton identification from given data, Inf. Control 37 (1978) 302–320, http://dx.doi.org/10.1016/S0019-9958(78)90562-4.
[28] C. de la Higuera, J. Oncina, E. Vidal, Identification of DFA: Data-dependent versus data-independent algorithms, in: International Colloquium on Grammatical Inference, Springer, 1996, pp. 313–325.

Juan Miguel Vilar holds a B.Sc. in Computer Studies from the Liverpool Polytechnic (now Liverpool John Moores University), England, and a M.Sc. and Ph.D. in Computer Science from the Universidad Politécnica de Valencia, Spain. He currently works as a Professor at the Departament de Llenguatges i Sistemes Informàtics of the Universitat Jaume I at Castelló, Spain. His research interests include pattern recognition, machine learning, and statistical models.