Using Multiplicity Automata to Identify Transducer Relations from Membership and Equivalence Queries

Jose Oncina

Dept. Lenguajes y Sistemas Informáticos
Universidad de Alicante (SPAIN)
oncina@dlsi.ua.es

Abstract. Multiplicity Automata are devices that implement functions from a string space to a *field*. Usually the real number's field is used. From a learning point of view there exist some algorithms that are able to identify any multiplicity automaton from membership and equivalence queries.
In this work we show that those algorithms can also be used if the algebraic structure of a field is relaxed to a *divisive ring* structure, that is, the commutativity of the product operation is dropped.
Moreover, we define an algebraic structure, which is an extension of the string monoid, that allows the identification of any transduction that can be realized by finite state machines without empty-transitions.

1 Introduction

In the same way a language is defined as a subset (usually infinite) of strings, a transducer can be defined as a subset of pairs or strings. The first string is interpreted as the *input* string and the second as the *output* string, *i.e.* in a translation task the pair (*"to be", "ser"*) can represent that the English verb *"to be"* can be translate to Spanish by the verb *"ser"*.

One of the biggest classes of transductions that is known to be identifiable is that of the *subsequential functions*. These functions can be describe as the set of transductions that can be implemented by deterministic finite state machines in which the arcs and the states are labeled with strings of the output string space. The translation is given by the concatenation of the strings in the labels of the arcs and the final state used in the parsing of the input string (note that since the automaton is deterministic there is at most one path).

In this work we are interested in learning transducers in the exact learning model. In this model, proposed by Angluin in 1988 [Ang88], the learner is allowed to actively search information by asking queries to a teacher. Two types of queries are allowed:

– *membership* queries. When the learner can ask for the translation of some sentence.

– *equivalence* queries. When the learner thinks it has a suitable hypothesis, it can ask the teacher if it is correct. If the model is correct the teacher answers YES and the process stops. If it is incorrect, the teacher answers with a counter example.

In 1996 Vilar [Vil96] proposed an algorithm to efficiently identify any subsequential transducer in this model.

The most important drawback of the subsequential functions is that they can not cope with ambiguities. For example, the verb *"to be"* can also be translated to Spanish by the verb *"estar"*, but subsequential functions, as they are based on deterministic machines, are unable to give several options.

In this work we are going to describe a class of transductions that includes properly the subsequential functions and permits the expression of ambiguous transductions. This class includes all the transductions that can be performed by non deterministic ϵ-free automata where the edges and the states can have several labels.

The proposed algorithm is a reinterpretation of an algorithm to identify Multiplicity Automata (MA) [BBB$^+$00] in the exact learning framework. Usually MA implement string to real functions. MA are very similar to stochastic automata where there are no restrictions to force the function to be a distribution probability. They can be described as a non deterministic automata with labels in the edges and states (usually real numbers). The value assigned to a string is the sum over all possible parses of the string of the product of the labels in the edges and the final state used in each parse.

Our idea relies in substituting the field of the real numbers (with its multiplication and addition) which is usually used in MA by an alphabet with the concatenation playing the role of the multiplication and the inclusion in a multiset playing the role of the addition. Note that the commutativity is lost when replacing the product of reals by the concatenation of strings. That means we are not longer working in a field but, with some extensions described later, in a *divisive ring*. It is easy to check that, in the demonstration of the properties of his algorithm, Beimel *et al* did not use the commutativity of the product, and therefore, their results remains valid in the case of *divisive rings*.

2 Notation

2.1 String expressions

Let $\Sigma = \{a, b, \dots\}$ be a finite set or *Alphabet*. The set $E(\Sigma)$ of *string expressions* over Σ is defined as follows:

- $\epsilon \in E(\Sigma)$ and $\emptyset \in E(\Sigma)$
- $a \in \Sigma \Rightarrow a \in E(\Sigma)$
- $x \in E(\Sigma) \Rightarrow -x \in E(\Sigma)$
- $x \in E(\Sigma) \Rightarrow x^{-1} \in E(\Sigma)$
- $x, y \in E(\Sigma) \Rightarrow x + y \in E(\Sigma)$

– $x, y \in E(\Sigma) \Rightarrow x \cdot y \in E(\Sigma)$

Let $x, y, z \in E(\Sigma)$, we define the following relation of equivalence:

– $(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)$	product associativity
– $(x + y) + z \equiv x + (y + z)$	addition associativity
– $x + y \equiv y + x$	addition commutativity
– $x \cdot (y + z) \equiv x \cdot y + x \cdot z$	product distributivity
– $x + \emptyset \equiv \emptyset + x \equiv x$	addition neutral element
– $x \cdot \epsilon \equiv \epsilon \cdot x \equiv x$	product neutral element
– $x + (-x) \equiv \emptyset$	addition inverses
– $x \cdot x^{-1} \equiv \epsilon$	product inverses

Let us call $[\Sigma]$ the collection of the equivalence classes in $E(\Sigma)$. Then $([\Sigma], +, \cdot)$, in abstract algebra, has a structure of *divisive ring*, that is, a *field* where the product operator is not commutative.

The product is interpreted as the usual concatenation of strings (this is the reason for avoiding the commutativity) where ϵ (the neutral element) represents the empty string. The set has been enriched with concatenation symmetric elements, so it is expected to have strings like $ab^{-1}a$. We can think of this strings as "intermediate" results. It should be assured that, at the end of all the computations, no strings with "special symbols" appear. In the sequel, we use the usual juxtaposition notation (xy instead of $x \cdot y$).

The addition is interpreted as a multiset inclusion. Then the expression $x+y$ represents a multiset with two elements, x and y, $x + x + x$ is a multiset with three elements (that are equal), $x+y+-y$ a multiset with just one element (the x) since $y - y \equiv \emptyset$ and $y + \emptyset \equiv y$.

2.2 Multiplicity Automata

Definition 1 (Multiplicity Automata). *Let Σ be an alphabet and let K be a ring. A* Multiplicity Automaton *(MA) is a 3-tuple (λ, μ, γ) such that:*

– $\lambda \in K^{1 \times n}$ *(a $1 \times n$ matrix with values in K)*
– $\gamma \in K^{n \times 1}$ *matrix (a $n \times 1$ matrix with values in K)*
– μ *is morphism of monoids $\mu : \Sigma^* \to K^{n \times n}$ (return a $n \times n$ matrix)*

That is, $\mu(\lambda) = I$ (the unit matrix), and $\forall x, y \in \Sigma^*, \mu(xy) = \mu(x)\mu(y)$. Then, μ can be represented as a set of $|\Sigma|$ $K^{n \times n}$ matrices.

Let $\alpha = (\lambda, \mu, \gamma)$ be a multiplicity automaton, we can define the function performed by α as $f_\alpha : \Sigma^* \to K$ such that:

$$f_\alpha(x) = \lambda\mu(x)\gamma$$

A multiplicity automaton $\alpha = (\lambda, \mu, \gamma)$ can also be interpreted as a weighted non deterministic automata with n states, where λ_i is the weight of beginning in state q_i, $[\mu(a)]_{i,j}$ is the weight of the arc that goes from state q_i to state q_j with the symbol a and γ_i is the weight of ending in state q_i. The weight of a path is

computed as the product of the weight of the initial state times the weights of the arcs used in its parsing times the weight of the ending state. The weight of a string is the sum of the weights of all the possible paths of the string in the automaton.

It is easy to see that any multiplicity automata function can be implemented by a multiplicity automaton with only one initial state and with at most one more state than the original. Then, without loss of generality, the vector λ in the definition can be fixed to a vector such that $\lambda_1 = \epsilon$ and $\lambda_i = \emptyset, 1 < i \leq n$.

In our case instead of a generic ring we are going to use the divisive ring of the string expressions.

Example 1. Let $\Sigma_i = \{a\}$ and $\Sigma_o = \{0, 1\}$ be respectively the input and output alphabets, let $f : \Sigma_i^* \to [\Sigma_o]$ be a function such that:

$$f(a^n) = \begin{cases} 0^n & \text{if } n \text{ is odd} \\ 1^n & \text{if } n \text{ is even} \end{cases} \tag{1}$$

It is easy to see that the MA depicted in Figure 1 realizes this function.

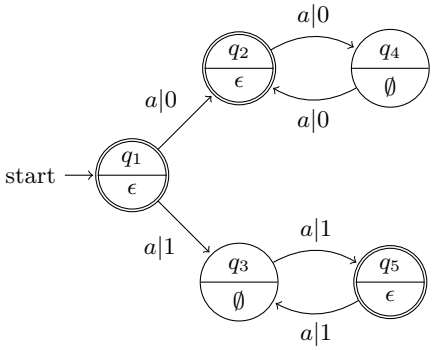


Fig. 1. MA for function in equation 3

Example 2. The matrix representation of the automaton from Figure 1 is:

$$\lambda = \begin{pmatrix} \epsilon & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix} \qquad \mu_a = \begin{pmatrix} \emptyset & 0 & 1 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & 0 & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & 1 \\ \emptyset & 0 & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & 1 & \emptyset & \emptyset \end{pmatrix} \qquad F = \begin{pmatrix} \epsilon \\ \epsilon \\ \emptyset \\ \emptyset \\ \epsilon \end{pmatrix} \tag{2}$$

Example 3. It is not very difficult to see that the function in Equation 3 can also be realized by the MA automaton in Figure 2.

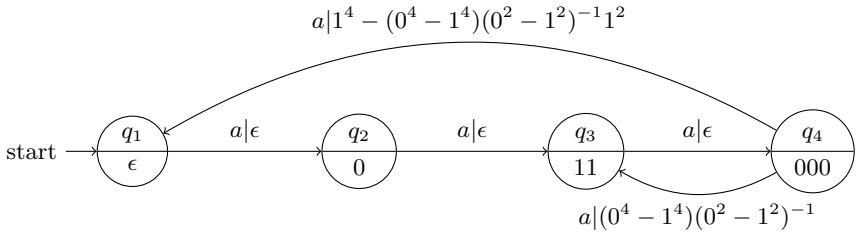


Fig. 2. A minimum size automaton that realizes equation 1 function

It is easy to see that any transducer based on a non deterministic ϵ-free automaton with a finite number of strings in the edges or states can be represented as a MA over the string expressions divisive ring.

Let we see an example to illustrate that.

Example 4. Let $\Sigma_i = \{a\}$ and $\Sigma_o = \{0\}$ be respectively the input and output alphabets, let $f : \Sigma_i^* \to [\Sigma_o]$ be a function such that:

$$f(a^n) = \sum_{i=0}^{n} 0^i \tag{3}$$

That is:

$$\begin{aligned} f(\epsilon) &= \epsilon && (\equiv \{\epsilon\}) \\ f(a) &= \epsilon + 0 && (\equiv \{\epsilon, 0\}) \\ f(aa) &= \epsilon + 0 + 00 && (\equiv \{\epsilon, 0, 00\}) \\ &\dots \end{aligned}$$

It is easy to see that the MA depicted in Figure 3 realizes this function.

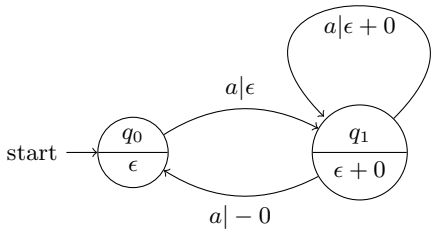


Fig. 3. MA that realizes the function in equation 3

2.3 Hankel Matrix

The MA inference algorithm that we are going to use relies on some properties of the Hankel matrix. Although the following definitions and theorems are stated for fields we can realize the commutativity of the product is never used and then, they continues being valid for divisive rings. Part of the following material can be found in some Beimel *et al* papers [BBB+96] [BBB+00], we just transliterate it for divisive rings to make the paper more self content.

Let $\mathcal{D}$ be a divisive ring (a field where the product is not commutative), Σ be an alphabet, ϵ be the empty string, and $f : \Sigma^* \to \mathcal{D}$ be a function. The *Hankel matrix* of the function f is an infinite matrix F where each of its rows and columns are indexed by strings in Σ^*. The (x, y) entry of F contains the value $f(xy) \in \mathcal{D}$

We use F_x to denote the xth row of F. The (x, y) entry of F may be therefore denoted as $F_x(y)$ or as $F_{x,y}$.

Example 5. The Hankel matrix of the function in equation 1 is:

$$F = \begin{pmatrix} \epsilon & 0 & 11 & 000 & 1111 & \dots \\ 0 & 11 & 000 & 1111 & 00000 & \dots \\ 11 & 000 & 1111 & 00000 & 111111 & \dots \\ 000 & 1111 & 00000 & 111111 & 0000000 & \dots \\ 1111 & 00000 & 111111 & 0000000 & 11111111 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} \tag{4}$$

The following theorem of Carlyle and Paz [CP71] and Fliess [Fli74] is a fundamental theorem from the theory of formal series. Although it was stated for fields it is easy to check that is is also valid for divisive rings as we state it here.

Theorem 1. *Let $f : \Sigma^* \to \mathcal{D}$ such that $f \not\equiv 0$ and let F be the corresponding Hankel matrix. Then, the size r of the smallest MA α such that $f_\alpha \equiv f$ satisfies $r = rank(F)$ (over the divisive ring $\mathcal{D}$)*[1]

The importance of the theorem is double: first, it relates the size of the minimal automaton for f to the rank of its Hankel matrix. And second, the proof is constructive. It gives a way to build a MA from any finite rank Hankel Matrix.

Given a function $f : \Sigma^* \to \mathcal{D}$ such that the corresponding matrix F has finite rank r, let $F_{x_1}, F_{x_2}, \dots, F_{x_r}$ be r linearly independent rows of F (*i.e.* a basis) corresponding to strings $x_1, x_2, \dots, x_r$. (since $f \not\equiv 0$, it holds that $F_\epsilon \neq 0$, then F_ϵ can always be an element of the basis. Then, we take $x_1 = \epsilon$).

Then the MA $\alpha = (\lambda, \mu, \gamma)$ that realizes the function f_α is:

- $\lambda = (\epsilon, \emptyset, \dots, \emptyset)$.
- $\gamma = (f(x_1), \dots, f(x_r))^t$.

[1] The demonstration of the theorem can also be found in [BBB+96] [BBB+00].

– for every $a \in \Sigma$, define the ith row of the matrix $\mu(a)$ as the (unique) coefficients of the row $F_{x_i}a$ when expressed as a linear combination of $F_{x_1}, \ldots, F_{x_r}$. That is,

$$F_{x_i a} = \sum_{j=1}^{r} [\mu(a)]_{i,j} F_{x_j} \tag{5}$$

Example 6. It can be shown that F_ϵ, F_a, F_{aa} and F_{aaa} in the Hankel matrix of Equation 4 forms a basis.

We are going to show that F_{aaaa} can be expressed as a linear combination of F_ϵ, F_a, F_{aa} and F_{aaa}.

That is, we have to solve the system of equations:

$$\begin{array}{llll} \alpha_1 & +\alpha_2 0 & +\alpha_3 1^2 & +\alpha_4 0^3 = 1^4 \\ \alpha_1 0 & +\alpha_2 1^2 & +\alpha_3 0^3 & +\alpha_4 1^4 = 0^5 \\ \alpha_1 1^2 & +\alpha_2 0^3 & +\alpha_3 1^4 & +\alpha_4 0^5 = 1^6 \\ \alpha_1 0^3 & +\alpha_2 1^4 & +\alpha_3 0^5 & +\alpha_4 1^6 = 0^7 \end{array}$$

It is straightforward to use the Gauss method to solve the system (paying attention not to use the product commutativity) and find the solution:

$$\begin{aligned} \alpha_4 &= \emptyset \\ \alpha_2 &= \emptyset \\ \alpha_3 &= (0^4 - 1^4)(0^2 - 1^2)^{-1} \\ \alpha_1 &= 1^4 - (0^4 - 1^4)(0^2 - 1^2)^{-1} 1^2 \end{aligned}$$

Observe that these are the values that were used to depict the MA from Figure 2.

3 The Beimel *et al* algorithm

The algorithm works using a finite version of the Hankel matrix $\hat{F}$. Let X and Y be two sets where the indexes of the finite version of the Hankel matrix are stored.

The algorithm works as follows:

1. $X = \{x_1 = \epsilon\}, Y = \{y_1 = \epsilon\}$ and $\ell = 1$
2. Build a MA $\alpha = (\lambda, \mu, \gamma)$ using Theorem 1
3. Ask an equivalence query.
   If the answer is YES halt with output α.
   Otherwise, let z be the counterexample.
   (a) Find (using membership queries) a string wa which is a prefix of z such that:
      i. $\hat{F}_w = \sum_{i=1}^{\ell} [\lambda \mu(w)]_i \hat{F}_{x_i a}$; but
      ii. there exists a y such that: $\hat{F}_{wa} \neq \sum_{i=1}^{\ell} [\lambda \mu(w)]_i \hat{F}_{x_i a}(y)$
   (b) $X = X \cup \{x_{\ell+1} = w\}$, $Y = \{y_{\ell+1} = ay\}$, $\ell = \ell + 1$
   GO TO step 2

Beimel *et al* showed that the algorithm works in $O((|\Sigma_i| + m)rM(r))$ time using r equivalence queries and $O((|\Sigma_i| + \log m)r^2)$ membership queries. Where Σ_i is the input alphabet, r is the rank of the Hankel matrix, $M(r)$ is the complexity of multiplying two $r \times r$ matrices ($O(r^{2.376})$) and m is the length of the longest counter-example.

Once more, it can be checked in the work of Beimel *et al* that the commutativity of the product is not used and then, the algorithm is still valid in divisive rings.

Note that in our case, the equivalence query should return a string expression describing *all* the possible output strings and the counter example of an equivalence query should return a pair (string, string expression) such that the string expression is a description of all the possible transduction of the input string.

Example 7. Let us try to find a transducer for the function in Equation 3:

In such a case we need 2 states and the system to solve is:

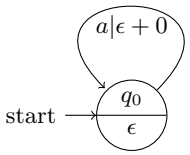
$$\begin{array}{lll} \alpha_1\epsilon & +\alpha_2(\epsilon + 0) & = \epsilon + 0 + 0^2 \\ \alpha_1(\epsilon + 0) & +\alpha_2(\epsilon + 0 + 0^2) & = \epsilon + 0 + 0^2 + 0^3 \end{array}$$

The solution is:

$$\alpha_1 = -0 \qquad\qquad \alpha_2 = \epsilon + 0$$

And the transducer in Figure 2 is obtained.

Note that the transducer from Figure 4 does not produces the transduction of Equation 3 since some of the output strings have a different number of repetitions (multiplicity).



$$\begin{aligned} f(\epsilon) &= \epsilon \\ f(a) &= \epsilon + 0 \\ f(aa) &= (\epsilon + 0)(\epsilon + 0) = \epsilon + 0 + 0 + 00 \\ &\dots \end{aligned}$$

Fig. 4. MA that does no realize function in equation 3

4 Conclusions and open questions

This work shows how to use an algorithm devised to learn multiplicity automata from membership and equivalence queries to identify transducer relations.

The proposed method is able to identify the class of transductions than can be expressed as finite state (and arc) machines with no ϵ-transitions.

As this is a first step to deal with ambiguous transductions it remains many problems to solve in order to be able to apply similar technique in more realistic situations:

- The way the membership and equivalence queries should be answered is too demanding. Information about *all* the transductions for the involved input string should be provided. Can we still be able to learn if only information about just one transduction is provided in each query?
- Since the method ensures identification, if the target function does not produce strings with inverse symbols, neither the produced function will do. But it can produce a complex string expression that, when simplified, is just a plain string (as happens in the MA in Figure 2). Does there exist a general method to simplify and compare string expressions? Does there exist a method to know if a multiplicity automaton produces only plain strings? If we compare the automata from Figures 1 and 2, obviously the first one is more understandable than the second one. Does there exist a method to remove complex expressions is arcs and states, possibly adding more states?
- The multiplicity is important in learning. Automata in Figures 3 and 4 show that, when the multiplicity doesn't matter, smaller automata can be obtained. How large can this reduction be? Does any learnable function remain learnable if the multiplicity is not taken into account?

References

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[BBB+96] A. Beimel, F. Bergadano, N.H. Bshouty, E. Kushilevitz, and S. Varricchio. On the applications of multiplicity automata in learning. In *IEEE Symposium on Foundations of Computer Science*, pages 349–358, 1996.

[BBB+00] A. Beimel, F. Bergadano, N.H. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. *J. ACM*, 47(3):506–530, 2000.

[CP71] J.W. Carlyle and A. Paz. Realizations by stochastic finite automaton. *J. Comput. Syst. Sci.*, 5:26–40, 1971.

[Fli74] M. Fliess. Matrices de Hankel. *J. Math. Pures Appl.*, 53:197–222 (Erratum in vol. 54, 1975), 1974.

[Vil96] J.M. Vilar. Query learning of subsequential transducers. In *ICGI'96: Proceedings of the 3rd International Colloquium on Grammatical Inference*, pages 72–83, London, UK, 1996. Springer-Verlag.