# IDENTIFYING REGULAR LANGUAGES IN POLYNOMIAL TIME*

José Oncina
Departamento de Tecnologia Informática y Computación
Universidad de Alicante, Spain.

and

Pedro García
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Spain.

ABSTRACT

The regular languages are commonly used as models in Syntactical Pattern Recognition tasks. A wide variety of inference algorithms have been developed to learn these models but usually these algorithms only make use of positive information, eventhough the negative is also available. In this paper we present an algorithm that always obtains a deterministic automaton compatible with the positive and negative data, that can identify in the limit any regular language and works in a polynomial time. Some experiments are performed to show its behaviour.

## 1. Introduction

One of the more deeply studied problems in the field of the inductive learning[1,2] is the regular language inference problem described by means of finite automata. The interest in this problem is because of the position of these languages in the Chomsky hierarchy. The regular language family is the simplest and best known, it is because of this that it can be used as the starting point to study larger families. At the same time the learning techniques developed for this problem can be extended to other domains. On the other hand, relating with the Syntactical Pattern Recognition framework (SPR), some tasks can be suitably represented by means of regular languages. A wide variety of inference algorithms have been applied to learn models[3,4,5,6]. However, the inference methods used in SPR learn using positive information only.

The only algorithm in existence capable of inferring correctly the regular language family (using positive and negative data) is the one proposed by Gold[7]. This algorithm will not assure the data consistence (when there is not enough data) and therefore it can not be used in SPR tasks where the lack of data is usual.

In this paper two algorithms are described. The first one was previously presented in[8] and[9]. This algorithm is capable of inferring the regular language family (using positive and negative data), always produces a (normally very small but possibly no deterministic)

* Work partially supported by the Spanish CICYT under grant TIC-0448/89

automaton compatible with the data, and works in a polynomial time. The second one is an improvement on the first, it has similar characteristics but always produces a deterministic automaton.

These algorithms are based on a state clustering. It is well known that if $S_+$ is a structurally complete sample of a regular language L (all the transitions on the canonical automaton of the language L (A(L)) are used in the acceptation of the strings in $S_+$) there exists a partition $\pi$ on the state set of the prefix tree acceptor of $S_+$, $PT(S_+)$ such that $PT(S_+)/\pi$ is isomorphic to A(L). The words in $\Sigma^*$-L allow us to reject some partitions, then the inference problem can be put as a guided search in the reticule of all the possible partitions. Unfortunately the search space grows exponentially with the size of the state set in $PT(S_+)$ and then with the size of $S_+$. In place of the exhaustive search, these algorithms try to merge pairs of states in $PT(S_+)$ according to an order and only does it if the resulting automaton rejects all the negative sample.

These algorithms, as the one proposed by Gold have the disadvantage of not being incremental. The Gold algorithm has a drawback, it can not generalize unless the sample has a characteristic set. On the other hand our algorithm is free of this inconvenience and then it is better to use in learning tasks.

## 2. Mathematical Background and Notation

Let $\Sigma$ be a finite set or *alphabet*, and $\Sigma^*$ the *free-monoid* over $\Sigma$. For any string $x \in \Sigma^*$, $|x|$ denotes the *length* of $x$ and $\lambda$ is the symbol for the string of length zero. For every $x,y \in \Sigma^*$, $xy$ is the *concatenation* of $x$ and $y$, with $|xy| = |x| + |y|$. Let $u = vw$ be a string, we say that $v$ ($w$) is a *prefix* (*suffix*) of $u$. A *language* is any subset of $\Sigma^*$.

If L is a language over $\Sigma$, we define the set of prefix over L by:

$$Pr(L) = \{ u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in L \}$$

and the set of tails of $u$ in L by:

$$T_L(u) = \{ v \in \Sigma^* \mid uv \in L \}$$

A finite automaton A (FA) is defined by a five-tuple $(Q,\Sigma,\delta,q_0,F)$ where Q is a finite set of states, $q_0$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta$: $Q \times \Sigma \rightarrow 2^Q$ is a partial function. We say that $q$ is an *a-successor* of $p$ If $p \in \delta(q,a.)$. We say that $p$ is a *brother* of $q$ if there exists an $r$ such that $p$ and $q$ are a-succesors of $r$. A is deterministic if for all $q \in Q$ and for all $a \in \Sigma$, $\delta(q,a)$ has at the most one element. The language accepted by A is denoted by L(A). A language is regular iff it is accepted by a FA.

If $A = (Q,\Sigma,\delta,q_0,F)$ is a FA and $\pi$ is a partition of Q, we denote by $B(q,\pi)$ the only block that contains $q$ and we denote the quotient set $\{ B(q,\pi) \mid q \in Q \}$ as $Q/\pi$. Given a FA A and a partition $\pi$ over Q, we define the quotient automaton $A/\pi$ as:

$$A/\pi = (Q/\pi,\Sigma,\delta',B(q_0,\pi),\{B \in Q/\pi \mid B \cap F \neq \emptyset\})$$

where $\delta'$ is defined as:

$\forall B,B' \in Q/\pi$, $\forall a \in \Sigma$, $B' \in \delta'(B,a)$ if $\exists q,q' \in Q$, $q \in B$, $q' \in B'$ : $q' \in \delta(q,a)$.

Given A and $\pi$ over Q, we have that $L(A) \subseteq L(A/\pi)$.

Given a DFA A = (Q,Σ,δ,$q_0$,F) L(A)=L, the partition $\pi_L$ is defined as $B(q,\pi_L)=B(q',\pi_L)$ iff $\forall x\in\Sigma^*$ $\delta(q,x) \in F$ iff $\delta(q',x) \in F$ defines an $A/\pi_L$ that is the DFA that has the minimum number of states and accepts L(A); this automaton is called the *canonical automaton* of the language L and we denote it by A(L).

Given L, we can also define the *canonical automaton* A(L) = (Q,Σ,δ,$q_0$,F) as:

$$Q = \{ T_L(u) \mid u\in Pr(L) \};\ q_0 = T_L(\lambda);\ F = \{ T_L(u) \mid u\in L \};$$

$$\delta(T_L(u),a) = T_L(ua) \text{ where } u,ua\in Pr(L)$$

A Sample S of a language L is a finite set of words that we can represent as $S=(S_+,S_-)$ where $S_+$ is a subset of L (positive sample) and $S_-$ is included in the complementary language of L (negative sample).

Let $S_+$ be a positive sample from a regular language L, we say that $S_+$ is *structurally complete* if all the transitions on A(L) are used in the acceptation of the strings in $S_+$.

Let $S_+$ be a positive sample from a regular language L, we can define the prefix tree acceptor of $S_+$ as $PT(S_+) = (Pr(S_+),\Sigma,\delta, \lambda,S_+)$ where δ is defined as: $\delta(u,a) = ua$ where $u, ua \in Pr(S_+)$. This automaton only accepts the strings belonging to $S_+$.

It is well known that if $S_+$ is a structurally complete sample of a regular language L, then there exists a partition π on the states of $PT(S_+)$ such that $PT(S_+)/\pi$ is the A(L).

We denote < as the lexicographic order in $\Sigma^*$. Given a positive sample $S_+$ and a partition π over the set of all string prefixes of the sample, we can define an order between the blocks of the partition.

Let $S_+$ be a positive sample, let π be a partition over $Pr(S_+)$ and let $B_i,B_j$ be two blocks of π. we are going to say that $B_i < B_j$ iff some $u\in B_i : \forall\ v\in B_j,\ u < v$ exists.

Given a partition π over $Pr(S_+)$ and given $B_i,B_j\in\pi$ we define *merge*$(\pi,B_i,B_j)$ as:

*merge*$(\pi,B_i,B_j) = \{ B\in\pi \mid B \neq B_i, B \neq B_j \} \cup \{ B_i \cup B_j \}$

In the rest of the paper we assume that the subindex of a block is the same as the subindex of the smallest string belonging to the block. If a block only has one string we will represent it indistinctly as a block or as a string.

## 3. The Automaton Learning Algorithm

Let $S_+$ be a set of strings belonging to an unknown regular language L, and let $S_-$ be a set of strings not belonging to L. If $S_+$ is big enough we can suppose that $S_+$ is structurally complete and then, there exists a partition $\pi_c$ over the set of states of the prefix tree acceptor of $S_+$ (PT(S+)) such that, if a *merge* is performed on all the states belonging to the same block, we obtain the canonical automaton of the regular language L.

The proposed algorithm consists of a procedure that starts building the prefix tree acceptor of the positive sample ($S_+$), and then proceeds by orderly trying the *merge* of states of $PT(S_+)$. At the end of the process we expect the obtained automaton to accept the positive sample and to reject the negative. It is obvious that the $PT(S_+)$ accepts the positive sample and rejects the negative but each time we perform a *merge* we are increasing the language accepted by the automaton, then it is possible that the automaton accepts a negative string. It can be shown[9] that if both positive and negative samples are big enough,

this only happens in the algorithm if and only if the two merged states belong to different blocks of the partition $\pi_c$. Then, when we have enough information, all the states are represented in the prefix tree acceptor, and in the algorithm we are merging all the compatible states, then the algorithm produces the partition $\pi_c$. It also can be sown [Oncina and García,92] that for each regular language there exists a positive and negative sample (that grows with the square of the size of the canonical automaton of the language) such that if this sample, possibly increased with more strings, is supplied to the algorithm then the resulting automaton is isomorphic to the canonical automaton of the regular language.

Given a complete sample $S = (S_+,S_-)$ of an unknown regular language L, this algorithm produces an automaton hypothesis $PT(S_+)/\pi_r$ where $\pi_r$ can be recursively calculated as:

| | |
|---|---|
| $\pi_0 = Pr(S_+) = \{ u_0, .. , u_r \}$ | (We suppose that the prefixes are indexed in lexicographical order, then $u_0=\lambda$) |
| $\pi_n = merge(\pi_{n-1},B,u_n)$ | if $\exists$ B, B'$\in \pi_{n-1}$: B and $u_n$ are a-successor of B' and B is the lower a-successor of B' such that B<$u_n$ and $S_-\cap L(PT(S_+)/merge(\pi_{n-1},B,u_n)) = \emptyset$ |
| $\pi_n = merge(\pi_{n-1},B,u_n)$ | if $\exists$ B$\in \pi_{n-1}$: B is the lower block in $\pi_{n-1}$ such that B<$u_n$ and $S_-\cap L(PT(S_+)/merge(\pi_{n-1},B,u_n)) = \emptyset$. |
| $\pi_n = \pi_{n-1}$ | otherwise |

The order in which the *merge* operations are performed is very important because it assures that the sub-automaton formed by the explored states is isomorphic to a sub-automaton of the canonical automaton. As it can be seen in the algorithm, the blocks (states) are selected in lexicographical order (for $u_1$ to $u_r$) and the merging is performed with smaller brother blocks (B<$u_n$) if the resulting automata rejects the negative sample ($S_-\cap L(PT(S_+)/merge(\pi_{n-1},B,u_n)) = \emptyset$). If it is not the case, this step is repeated but now using all the blocks that are smaller than $u_n$. If this last step was not successful the partition remains unchanged.

The algorithm must explore all the blocks of the initial partition (<$||S_+||+1$), for each block the number of merges that we must try is always lower than $||S_+||$ (there can not be more than $||S_+||$ states lower than $u_n$) and for each merge we must verify that $S_-\cap L(PT(S_+)/\pi_n) = \emptyset$. This has a computational cost of $O(n||S_-||)$ (the automaton can be non deterministic) where n is the number of states of $PT(S_+)/\pi_n$ and this is always lower than the number of states of the prefix tree acceptor that is bounded by $||S_+|| + 1$. Then the complexity of the algorithm is $O(||S_+||^3||S_-||)$.

Using $S_+=\{011, 101\}$ as the positive sample and $S_-=\{1, 01\}$ as the negative sample fig. 1 illustrates the automata corresponding to some key steps of the algorithm. First the algorithm builds a partition with all the prefixes of the positive sample then $\pi_0=\{\lambda, 0, 1, 01, 10, 011, 101\}$, the corresponding automaton (the prefix tree acceptor) is shown in fig. 1(a). Next, as the state 0 has not any brother, it tries to merge the blocks $\lambda$ and 0 (fig. 1(b)), as the induced automaton rejects the negative sample ($S_-\cap L(PT(S_+)/merge(\pi_0,\lambda,0)) = \emptyset$) then $\pi_1=\{\{\lambda,0\}, 1, 01, 10, 011,101\}$. Now it is the turn of the state 1, it has not

any smaller brother then it tries a *merge* with the state λ, but the resulting automaton accepts the negative string 01, then $\pi_2=\pi_1$. To build $\pi_3$ it looks for smaller brothers of the state 01, it finds the state 1 and as the automaton resulting from the *merge* (fig. 1(c)) of these two states rejects the negative sample then $\pi_3=\{\{\lambda,0\}, \{1, 01\}, 10, 011,101\}$. In the next step it tries to merge the states λ and 10 without success (the negative sample 1 is accepted) but merging the states 1 and 10, we obtain a suitable automaton (fig. 1(d)) then $\pi_4=\{\{\lambda,0\}, \{1, 01, 10\}, 011,101\}$. Continuing with the algorithm the states 011 and 101 are merged with the state λ and we obtain the partition $\pi_6=\{\{\lambda,0, 011, 101\}, \{1, 01, 10\}\}$ fig. 1(e).

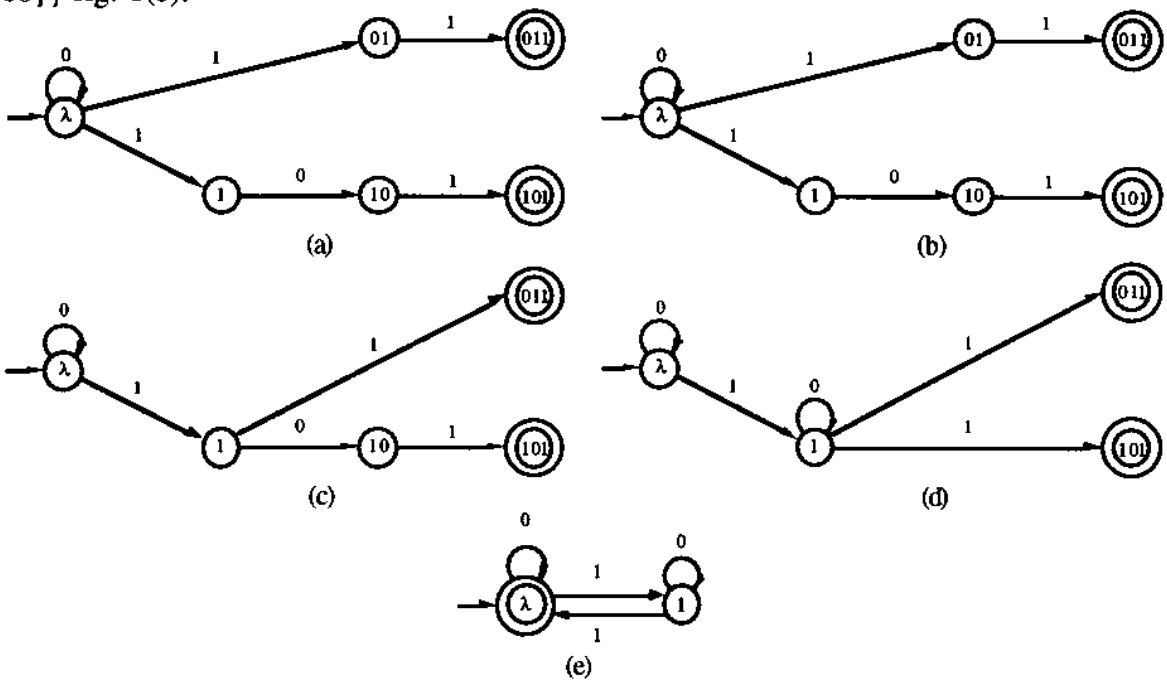


Fig. 1. Some key steps of the first algorithm

This algorithm has a slight disadvantage, it can produce a non-deterministic automaton when the sample is not big enough, although we can easily avoid it. We know that the canonical automaton is deterministic, then when in the algorithm we try to merge two states and the resulting automaton is not deterministic, we can merge the states that cause this non-determinism before testing if the negative samples are rejected. If the first automaton rejects the negative sample it is because the two states are equivalent (when we have enough information) and then, as the canonical automaton is deterministic, all the states that cause non-determinism are going to be merged in later steps. When we perform these additional merges we are increasing the language accepted by the automaton, then if the first automaton accepts a negative sample the last must do so too.

We go on defining the operation *D* that transforms, merging states, a non deterministic automaton into another deterministic automaton.

$D(PT(S_+)/\pi) = D(PT(S_+)/merge(\pi,B_i,B_j))$ if $\exists B_i, B_j, B_k$: $B_i, B_j$ are a-successors of $B_k$ and $B_i \neq B_j$

$D(PT(S_+)/\pi) = PT(S_+)/\pi$ otherwise

Now we define a new *merge* operation, the deterministic merge (*DMerge*). This operation produces a partition such that:

$PT(S_+)/DMerge(\pi,B_i,B_j) = D(\ PT(S_+)/merge(\pi,B_i,B_j)\ )$

(*DMerge*$(\pi,B_i,B_j)$ is the partition that induces over the prefix tree acceptor $PT(S_+)$ the construction of the deterministic automaton resulting from $D(\ PT(S_+)/merge(\pi,B_i,B_j))$.

In the worst case it is possible that the only way to transform the non-deterministic automaton to a deterministic one will be by merging all the states. Then this function has a computational cost of O(n), where n is the number of states of the automaton.

The modified algorithm can be defined in the following way:

Given a complete sample $S = (S_+,S_-)$ of an unknown regular language L, this algorithm produces an automaton hypothesis $PT(S_+)/\pi_r$ where $\pi_r$ can be recursively calculated as:

$\pi_0=Pr(S_+)=\{\ u_0,..,u_r\}$

$\pi_n = \pi_{n-1}$ if $\exists B\in\pi_{n-1}$: $u_n\in B$ and $B< u_n$

$\pi_n = DMerge(\ \pi_{n-1},B,u_n)$ if it is not the previous case and $\exists B\in\pi_{n-1}$: B is the smaller block such that $B<u_n$ and $S_-\cap L(PT(S_+)/DMerge(\pi_{n-1},B,u_n))=\emptyset$.

$\pi_n = \pi_{n-1}$ otherwise

We can observe that the first rule of the previous algorithm is not valid now because all the automata that we obtain are deterministic and then no state has a brother. The first rule of this algorithm is for skipping a step if the state being treated was previously merged to another block. In the second rule we have to try to merge the present state with all the previous ones then the number of *DMerge* functions that are executed is bounded by the number of states ($<||S_+||$). Each *DMerge* has a cost proportional to the number of states ($O(||S_+||)$) and for each one the cost to know if $S_-\cap L(PT(S_+)/\pi_n)=\emptyset$ is $O(||S_-||)$ (the automaton is deterministic). Then the cost of the second step is $O(\ (||S_+||+||S_-||)||S_+||\ )$. This step can be repeated for all the states of the prefix tree acceptor the complexity of the algorithm is $O(\ (||S_+||+||S_-||)||S_+||^2\ )$.

Using $S_+=\{011, 101\}$ as the positive sample and $S_-=\{1\}$ as the negative sample, fig. 2 illustrates the automata corresponding to some key steps of the algorithm (we can see that, with this negative sample, the previous algorithm obtains a non deterministic automaton). As in the previous case, at the beginning, the algorithm builds a partition with all the prefixes of the positive sample then $\pi_0=\{\lambda, 0, 1, 01, 10, 011, 101\}$, the corresponding automaton (the prefix tree acceptor) is shown in fig. 2(a). Next, it tries to perform a *DMerge* between the blocks $\lambda$ and 0. To do this we must first make a *merge* operation between these two states. The automaton induced by this partition is shown in figure 2(b). Now we must apply the *D* operation over this automaton, as there exists two brother blocks, 1 and 01, we must make a *merge* operation between them (figure 2(c)) and we apply again the *D* operation. As this automaton is deterministic, this operation ends and the partition that results from the *DMerge* is a partition $\pi_1$ such that $PT(S_+)/\pi_1$ is the automaton of the figure 2(c), then $\pi_1=\{\{\lambda,0\},\{1, 01\}, 10, 011,101\}$. And now is when we must verify that $S_-\cap L(PT(S_+)/\pi_1)=\emptyset$, that is, if the automaton rejects the negative

sample. In the next step we try a *DMerge* of the blocks λ and 1, and we obtain the automaton shown in figure 2(f) passing by the automata shown in the figure 2(d) and 2(e). This automaton accepts the negative sample 1, then, as there are not more blocks for trying to merge in the partition and applying the third rule of the algorithm, $\pi_2=\pi_1$. In the following steps we try a *DMerge* between λ and 10 without success because the resulting automaton accepts the negative sample 1. Next we try 1 and 01 and we obtain the automaton of the figure 2(h). Finally, we *DMerge* the states λ and 011 obtaining the automaton of the figure 2(i).

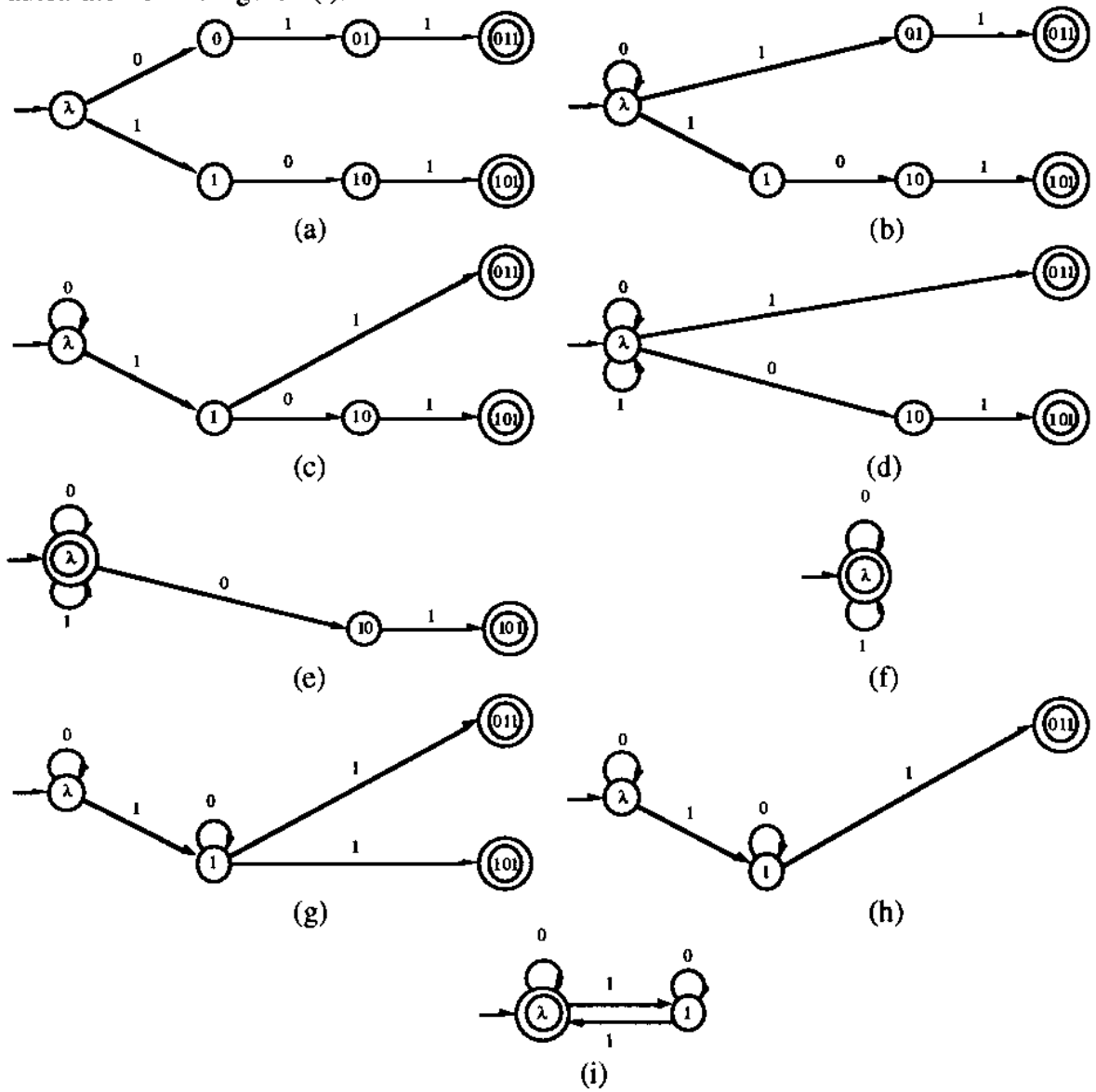


Fig. 2. Some key steps of the second algorithm

## 4. Experiments

Theoretical properties assessing the adequate behaviour in the limit (convergence) of the inference algorithm have been discussed in the previous section. However, from a Pattern Recognition viewpoint, results concerning finite data behaviour seem to be of greater interest. To obtain such results a number of experiments have been carried out. Some of these experiments and their results will be presented in this section.

The chosen tasks were to recognize if a decimal number is divisible by a constant (from two to ten). For each of the tasks a series of increasing-size random training-sets, each including the previous one, where drawn from a uniform distribution in the range of decimal numbers from 0 to $10^4$-1. The random procedure was prevented from generating repeated samples and the test-set consisted of the all the decimal numbers from 1 to $10^4$-1. Each random number was included as a positive or negative sample depending on its divisibility by the constant.

An example of one of these tasks is shown in figure 3. The items shown are: the test-set error rates and the number of states and of edges of the automaton found in each experiment.

It is worth noting that, with small training-sets, the inferred automaton tend to be rather large and error-prone, while both sizes and errors reduce dramatically as enough source structure is made available through the training data.

Each of these tasks were repeated six times for each constant. Figure 4 shows the mean sample size needed for the convergence (number of errors = 0) and for decreasing the number of errors to a rate lower than 5%.

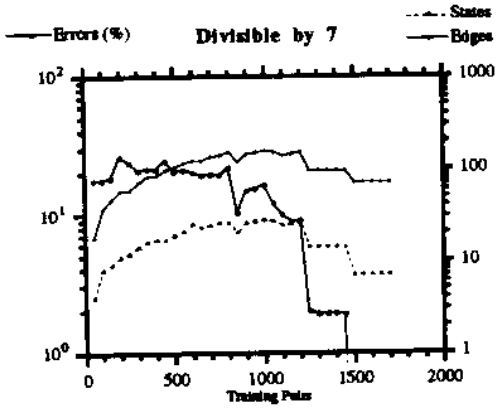


Fig 3. Behaviour of the algorithm for the "divisible by seven" tasks (uniform distribution).

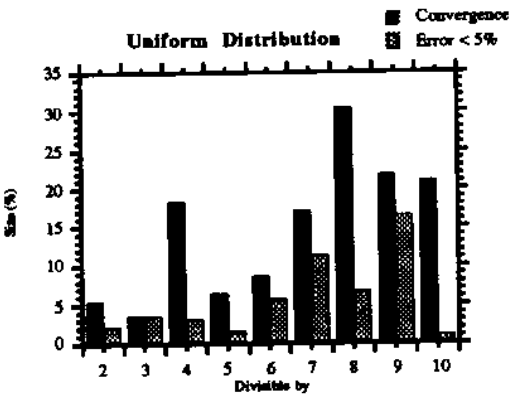


Fig 4. Size of the training set for the convergence and for obtaining an error rate lower than 5% for the task "Divisible by" for different constants (uniform distribution).

It should be noted that these results were obtained without taking into account how "relevant" the (random) training data were for the considered learning task. To investigate how small the training set could become if appropriately selected, a further experiment was carried out involving the following *greedy* procedure. First, starting with an automaton learnt from the first positive training data, "0", the test data was submitted in numerical

order to recognize up to one decimal number appeared which was incorrectly classified. Then this number was used as a training data. The first phase of this procedure stopped when all the strings were correctly classified. In the second phase, the training strings that were selected in the first phase were considered, in turn, to see whether they could be discarded without change in the inferred automaton. The results are shown in the figure 5.

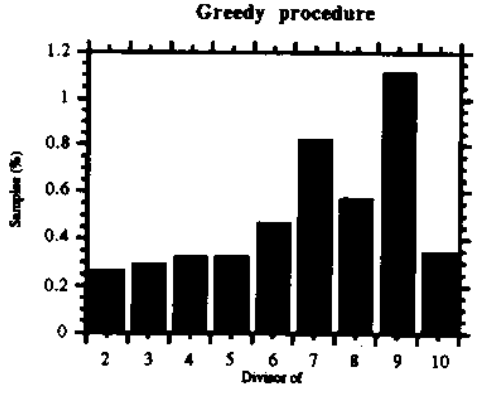


Fig 5. Size of the training set using the greedy method for the task "Divisible by" for different constants.

Theoretically, it is known that the algorithm can obtain more information about the structure of language from short string than from the longer ones. To observe this behaviour the set of tasks were repeated but now the numbers were randomly drawn from a non-uniform distribution in which the lengths of the strings were (approximately) equiprobable (we call it the exponential distribution). As previously, the behaviour of the algorithm for the "divisible by 7" task is shown in figure 6, and figure 7 shows the different sizes needed for the convergence and for obtaining an error rate lower than the 5%.

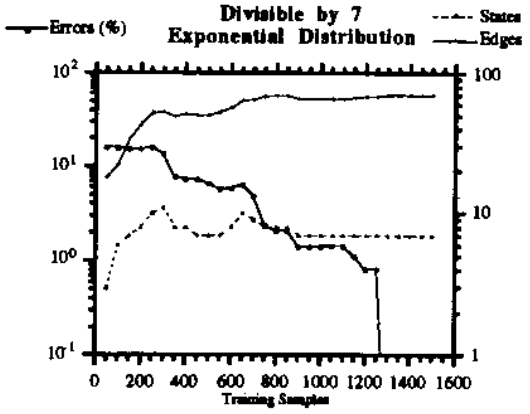


Fig 6. Behaviour of the algorithm for the "divisible by seven" task (non uniform distribution).

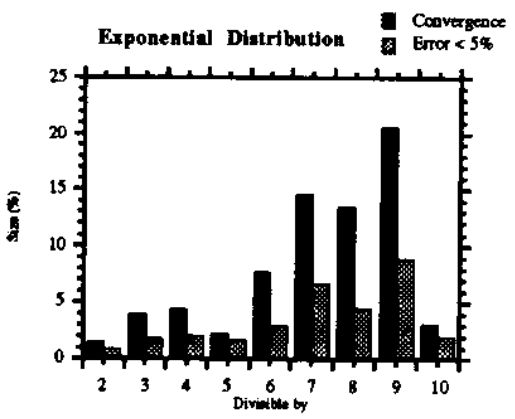


Fig 7. Size of the training set for the convergence and for obtaining an error rate lower than 5% for the task "Divisible by" for different constants.

## 5. Conclusions

The results of the experiments described in the last section indicate that the algorithm can find relatively accurate models for differents tasks with rather small training sets above all if the samples are representative.

In conclusion, we have presented an algorithm that have the following characteristics:

- It uses positive and negative information.
- It can identify any regular language in the limit.
- It always produces a deterministic automaton compatible with the data.
- It works in a polynomial time ( $O((\|S_+\|+\|S_-\|)\|S_+\|^2)$ ).
- It obtains good models with not very large training sets.

## 6. References


2. D. Angluin and C. H. Smith. "Inductive inference: theory and methods". *Computing Surveys*, 15(3), pp 237-269, 1983.
5. K. S. Fu and T. L. Booth. "Grammatical Inference: Introduction and Survey". parts 1 and 2. *IEEE Trans. Sys. Man and Cyber.*, SMC-5: 95-111, pp 409-423, 1975.
4. K. S. Fu. "Syntactic Pattern Recognition and Applications". *Prentice-Hall,* New York, 1982.
1. E. M. Gold. "Language identification in the limit". *Information and Control,* 10: pp 447-474, 1967.
7. E. M. Gold. "Complexity of automaton identification from given data". *Information and Control*, 37 pp 302-320, 1978.
3. R. C. González and M. G. Thomason. "Syntactic Pattern Recognition, an introduction". *Addison-Wesley*, Reading Mass., 1978.
6. L. Miclet. "Grammatical Inference". In *Syntactic and Structural Pattern Recognition.* H. Bunke and A. San Feliu (eds.) *World Scientific*, pp 237-290, 1990.
8. J. Oncina y P. García. "Un Algoritmo de inferencia de Lenguajes Regulares Usando Datos Positivos y Negativos". IV Simposium Nacional de Reconocimiento de Formas y Análisis de Imagenes. Granada 1990.
9. J. Oncina and P. García. "Inferring Regular Languages in Polynomial Update Time". In *Pattern Reconition & Image Analysis. Word Scientific*, 1992.