Algorithms for Learning Regular Expressions

(Extended Abstract)

Henning Fernau[1,2]*

[1] University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, UK
[2] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany; email: henningfernau@yahoo.de

Abstract. We describe algorithms that *directly* infer regular expressions from positive data and characterize the regular language classes that can be learned this way.

1 Introduction

Over the last about forty years, many algorithms have been proposed and implemented that are capable to infer a regular language, given only a finite number of samples of the language. Probably most of them, especially the ones in actual use, are mere heuristics in the sense that there does not exist any concise characterization of the class of languages that can be learned in this way.

In more mathematical terms, the "generalization capability" sketched in the preceding paragraph is best captured by Gold's learning paradigm of *learning (identification) in the limit from positive samples* as proposed by Gold [10]. In this well-established model, a language class $\mathcal{L}$ (defined via a class of language describing devices $\mathcal{D}$ as, e.g., grammars or automata) is said to be *identifiable* if there is a so-called *inference machine* I to which as input an arbitrary language $L \in \mathcal{L}$ may be enumerated (possibly with repetitions) in an arbitrary order, i.e., I receives an infinite input stream of words $E(1)$, $E(2)$, ..., where $E : \mathbb{N} \to L$ is an enumeration of L, i.e., a surjection, and I reacts with an output device stream $D_i \in \mathcal{D}$ such that there is an $N(E)$ so that, for all $n \geq N(E)$, we have $D_n = D_{N(E)}$ and, moreover, the language defined by $D_{N(E)}$ equals L.

In practical applications of this learning scenario, learning regular languages often means to infer regular expressions (REs), because REs are arguably the more suitable model to specify regular languages, especially for human beings. Therefore they (or variants thereof) are probably used in well-known tools as `grep` in UNIX. Unfortunately, to our knowledge all learning algorithms with known characterizations of the learned language class are based on grammars

* Most of the work on this project has been done while the author was with The University of Newcastle, School of Electrical Engineering and Computer Science, University Drive, NSW 2308 Callaghan, Australia; the assistance by a New Staff grant from the University of Newcastle that made possible to employ Linda Buisman, aka Postniece, to implement the algorithms described in this paper, is gratefully acknowledged.

and/or automata.[3] The disadvantage is that when you finally turn your results into REs (as the "target structure" towards human beings) by standard algorithms as contained in any introductory book to automata theory (e.g., [11]), these are often quite clumsy and lengthy, containing lots of "nested loops" and seeming repetitions of subsequences. Quite easily a human who wants to somehow check the outcome of the automatic learning procedure is simply abhorred by such output. We made some computer experiments in the context of inferring DTDs for XML documents with the aid of known DFA learners that verified these considerations [8]. Hence, although the (syntactic) task of DTD inference basically boils down to learning regular expressions, most systems that are actually used for this purpose rely on heuristics, see [9] for the description of one such system, as well as further references.

In order to improve on the readability of the resulting REs and probably also giving a better intuitive "explanation" of the observed data (sample strings), we are going to propose a learning procedure which is only generating REs of star height one, i.e., starred expressions get never nested. Although the corresponding class of languages is that restricted, many cases that occur in practice are covered.

2 Blockwise grouping and alignment

The basic technique we are using can be best described as *blockwise grouping and alignment*. This means the following: Given some sample strings, say

$$ababb, aabb, ababa, abc,$$

we would first transform them into

$$[a][b][a][bb], [aa][bb], [a][b][a][b][a], [a][b][c],$$

where we use square brackets to denote the "block letters." Actually, and more technically, we will call any $[x^n]$ a *block letter* whenever x is a character in the alphabet over which the input words are formed. x is called the *basic letter* of $[x^n]$. When we only use block letters $[x]$ representing x^n for any $n \geq 1$, we also say that we *ignore multiplicities*. In our transformation, we also require that the basic letters of neighboring block letters are different. At first glance, this appears to mean that we are dealing with infinite alphabets. But since we will only use block letters as derived from input samples, there are at any moment only finitely many of them, so that they can be conveniently handled.

In a second step, we try to align the blocks from left to right:

$$\begin{array}{cccccc} [a] & [b] & [a] & [bb] & & \\ [aa] & [bb] & & & & \\ [a] & [b] & [a] & [b] & [a] & \\ [a] & [b] & [c] & & & \end{array}$$

[3] There are even only few learning algorithms that directly deal with regular expressions, see [4, 5, 12] and the literature quoted therein, but they are not addressing the text learning model we are using here.

As an aside, let us mention that committing and restricting ourselves to leftmost alignments seems to be quite suited to what humans are doing in such a case, as well. Namely, when given the task of describing the common nature of the given four strings, most people I dare to claim would describe this nature along the following lines: "each string starts with one or two a, then there are one or two b, possibly followed by..." But there other obvious alternatives, as well, e.g.: "each string *possibly* starts with ab, followed by one or two a, then one or two b, and finally (optionally) one a or one c." Observe that in this second description, the second and fourth sample was aligned starting with the third block of the other two samples, i.e.:

$$\begin{array}{cccccc} [a] & [b] & [a] & [bb] & & \\ & & [aa] & [bb] & & \\ [a] & [b] & [a] & [b] & [a] & \\ & & [a] & [b] & [c] & \end{array}$$

Due to the above "psychologic" argument and because the general problem of finding "best alignments" is NP hard (corresponding to the multiple sequence alignment problem), we will restrict our attention to leftmost alignments in what follows.

So, sticking to our first "generalization" we would end up with the language describable with the following RE:

$$(a|aa)(b|bb)(\lambda|a(b|bb)(\lambda|a)|c).$$

Since this is still denoting a finite language, we might wish to further generalize. Then, it is quite natural to consider "one or two repetitions" as a (very) special case of "one or more repetitions". Having this in mind, we might then arrive at:

$$a^+b^+(\lambda|ab^+(\lambda|a)|c),$$

which, moreover, gives a shorter "explanation" of the given "observations" and is hence preferable by the "minimum description length" principle, likewise known as "Occam's razor," see [9, 13, 14] for a discussion with respect to Grammatical Inference.

Note that we were writing down Kleene plus rather than Kleene star operations to be sure not to destroy the "blockwise readings" we originally provided.

There is still one issue we did not discuss up to now: how should we, in general terms, arrive at a suitable "explanation" when facing the block letters $[x^{k_1}], \ldots, [x^{k_j}]$ within one block (e.g., assuming that $\{x^{k_1}, \ldots, x^{k_j}\}$ was the given input sample)? Without loss of generality, assume that $0 < k_1 < k_2 < \cdots < k_j$. Of course, we could generalize every time directly to x^+, but this might override to quickly the "multiplicity information" contained in the sample. We will instead choose integer numbers $\ell \geq 0$ and $r \geq 1$ such that, for each $1 \leq i \leq j$, r divides $k_i - \ell$, and such that r is maximal with this property. So, $r = 1$ and $\ell = 0$ will always satisfy the first part of the property (and this would correspond to the generalization x^+ mentioned above), but larger r will reveal more information within the loop. More concretely, if $k_i = 2i + 1$, i.e., given $x^3, x^5, x^7, \ldots$, the

"explanation" $x(xx)^+$ might look better than simply guessing at x^+. In other words, we would have taken $\ell = 1$ and $r = 2$ in that case.

On the downside, let us mention that this way only so-called strictly bounded languages can be derived, i.e., subsets of $x_1^* x_2^* \dots x_k^*$, where the x_i are characters of the original alphabet. Especially, this means that the universal language Σ^* cannot be derived for any input alphabet Σ with more than one letter. We will therefore also discuss strategies how to overcome this sort of dilemma, see Sec. 4.

3 Working out details of RE learning strategies

We start analyzing a simplified version of our learning strategy.

3.1 Creating the start tree

This algorithm can be explained as follows for a basic alphabet Σ and an input sample $I_+ \subset \Sigma^+$, yielding the recursive procedure `create-tree`(I), which initially takes as input I the sample I_+.

Algorithm 1 (`create-tree`(I)*).*

1. Left-align the input words.
   Due to the recursion within this procedure, aligning according to the first symbols is sufficient, so that this step can be very efficiently implemented.
2. This gives, for each symbol $a \in \Sigma$, the set
$$I(a) = \{w \in I \mid w \text{ starts with } a\,\}.$$
   This yields a partition of I into the different sets $I(a)$.
3. For each $I(a)$, form the set
$$I^a = \{v \in b\Sigma^* \mid b \neq a \wedge \exists n > 0 (a^n v \in I(a))\}.$$
4. Recursively call, for all $a \in \Sigma$, `create-tree`(I^a).

Let us introduce some further notations: Given a set $I \subset \Sigma^+$, consider

$$I(a) = \{a^{n_1} v_1, a^{n_2} v_2, \dots, a^{n_m} v_m\}$$

with $v_i \in (\Sigma \setminus \{a\})\Sigma^* \cup \{\lambda\}$ and $n_j \leq n_{j+1}$; then, the $m = |I(a)|$-tuple $S(I(a)) = (n_1, n_2, \dots, n_m)$ is also called the *a-spectrum* of I.

Further denotations are: $\alpha(S(I(a))) = n_1$, $\omega(S(I(a))) = n_m$.

The set J^b with $J = I^a$ is also denoted I^{ab}. In accordance, $I^\lambda = I$.

It is obvious that we can associate to I_+ a rooted labeled directed tree, the root being labeled $I_+ = I_+^\lambda$, and the children of a node labeled I_+^x being labeled with the I_+^{xa} for $a \in \Sigma$ if $I_+^x(a)$ is non-empty. The arc from I_+^x to I_+^{xa} is then labeled a. This tree is in direct correspondence with the recursion tree of `create-tree`(I_+) and will be also called the *start tree* for I_+. Sometimes, it is more convenient to label non-root vertices of this tree $I^x(a)$ instead of I^{xa}. Then, the root will be labeled r.

Getting back to our start example, we illustrate these notions.

Example 1. We take

$$I_+ = \{ababb, aabb, ababa, abc\} \subset \Sigma^+ = \{a, b, c\}^+$$

Hence, $I_+(a) = I_+$, and moreover,

$$I_+^a = \{babb, bb, baba, bc\}.$$

Since $I_+^a(b) = I_+^a$, we get

$$I_+^{ab} = \{abb, aba, c\}.$$

Note that $bb \in I_+^a(b)$ did not leave any trace in I_+^{ab}. Now, $I_+^{ab}(a) = \{abb, aba\}$ and $I_+^{ab}(c) = \{c\}$. Since $I_+^{abc} = \emptyset$, we continue with aligning $I_+^{aba} = \{bb, ba\}$, finally giving $I_+^{abab} = \{a\}$.

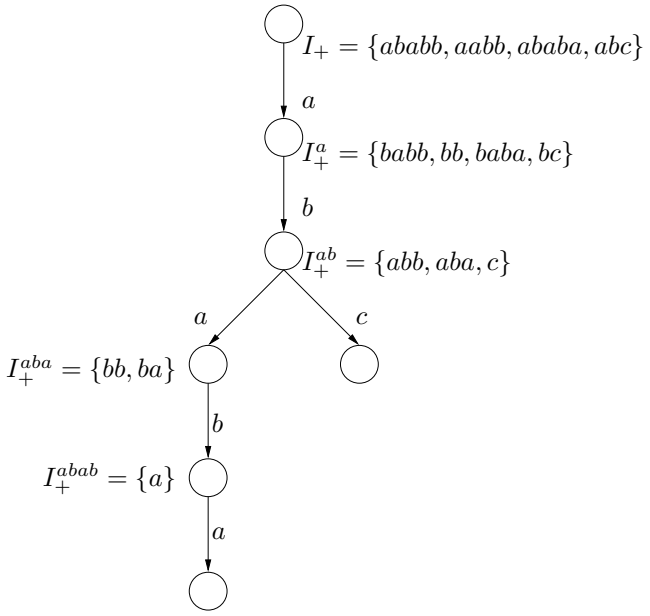


Fig. 1. A simple prefix block tree

In our example, the nodes of the start tree are labeled I_+, I_+^a, I_+^{ab}, I_+^{aba}, I_+^{abc}, I_+^{abab}, and I_+^{ababa}. Leaves in this tree are always labeled with the empty set. This gives the picture of Fig. 1. The alternative node labeling convention is displayed in Fig. 2.

3.2 Constructing NFAs by alignment

Recall now the notion of a *generalized nondeterministic finite automaton (NFA)*: arcs of such an NFA may be labeled with whole words (or even finite set of

words), not just single symbols. By way of contrast, a *generalized deterministic finite automaton (DFA)* is a generalized NFA which maintains the "determinism" property; therefore, we require that the labels of the arcs emanating from an arbitrary node form a prefix code.

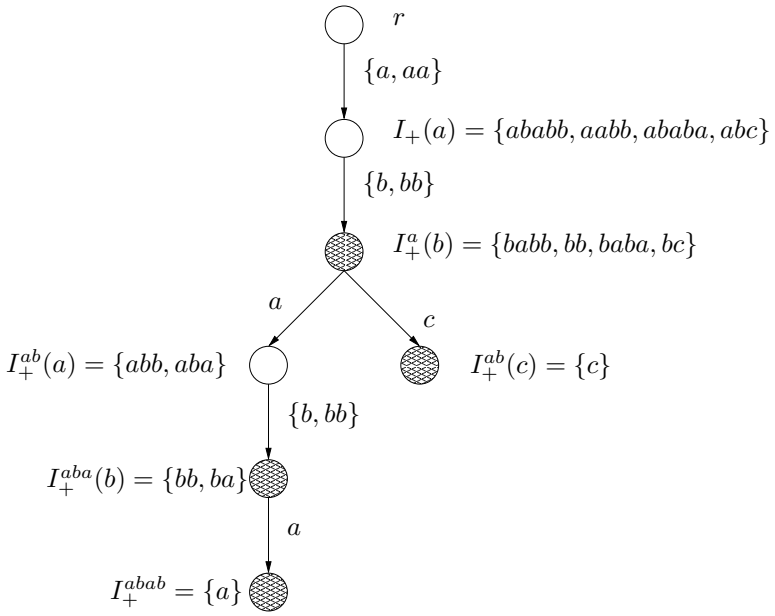


Fig. 2. The resulting NFA

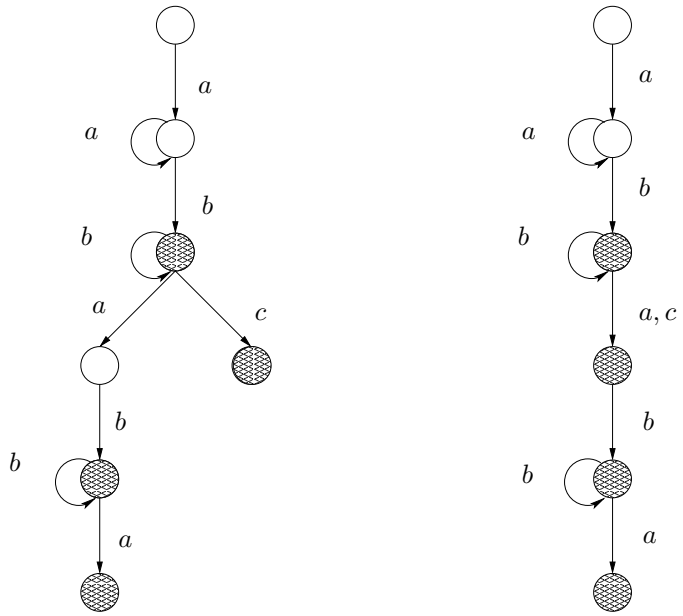
It is now easy to turn the start tree into a (generalized) NFA: If a node is carrying the label $I^x(a)$, then label the arc ending in $I^x(a)$ with the finite set $\{a^j \mid j \in S(I^x(a))\}$. The resulting NFA in our example is displayed in Fig. 2. The root obviously becomes the start state, and the terminal states are (as usual) indicated by double-circles. Observe that this leads to some generalization (e.g., now the word $ab \notin I_+$ will be accepted), according to the following *principle* which is a common characteristics *of alignment-based learning* algorithms [15]:

Aligned pieces of words are considered to be interchangeable.

However, this kind of generalization is very cautious, since it will always create only finite languages, given a finite sample. In order to arrive at infinite languages, loops must be introduced.

3.3 Introducing loops by determinization

We will introduce loops by turning the (generalized) NFA we obtained so far into a (generalized) DFA. To this end, if an arc A in the NFA is labeled with a finite



(a) The resulting DFA in $\mathcal{SL}$ (b) Another DFA proposal

Fig. 3. Two ways to generalize

set of words, say $\{a^{n_1}, \ldots, a^{a_m}\}$ containing more than one word, (i.e., $m > 1$) then only keep the shortest among these words as a label of A and enable the generation of the longer words by introducing a loop labeled a at the node A points to. The resulting generalized automaton is indeed deterministic, since in the situation sketched above, in the original generalized NFA there was no arc labeled a starting at the node A points to due to the alignment we performed in the very beginning. The resulting DFA for our example is depicted in Fig. 3(a). We can also summarize both generalization steps into one algorithm as follows:

Algorithm 2 (`generalize-simple`*(start tree)).*
If a node is carrying the label $J(a)$, then

- (re)label the arc ending in $J(a)$ with $a^{\alpha(S(J(a)))}$;
- moreover, if $\omega(S(J(a))) > \alpha(S(J(a)))$, then introduce a loop on the node labeled $J(a)$; this loop arc is labeled a.

Let us have a look at another example where we actually obtain generalized automata.

Example 2. Let $I_+ = \{aaa, aab\}$. The generalized NFA resulting from the start tree has three states (labeled r, $I_+(a)$ and $I_+^a(b)$, the first being the start state

and the latter two being final states). The arc from r to $I_+(a)$ is labeled $\{aa, aaa\}$ and the arc from $I_+(a)$ to $I_+^a(b)$ is labeled b. This yields a generalized DFA with the same states and the following three arcs:

- One arc labeled $aa = a^{\alpha(S(I_+(a)))}$ from r to $I_+(a)$.
- One loop arc at $I_+(a)$ labeled a.
- One arc labeled b from $I_+(a)$ to $I_+^a(b)$.

Obviously, it is the label of the arc from the root r to $I_+(a)$ which makes the automata "truely generalized."

3.4 The language class $\mathcal{SL}$

Which language class can we learn this way? In fact, this is an interesting question which can be posed for many "heuristics" which have been proposed in the literature (or merely presented as programs, nowadays on the web) for learning regular expressions (or more generally, certain classes of [descriptions of] languages). We now give two definitions, restricting the usual notion of a regular expressions and the usual notion of a DFA, which turn out to be two characterizations of the languages which can be learned by the algorithm `generalize-simple`.

Definition 1 (simple looping expressions). *Let us call two* union-free *regular expressions*

$$b_1^{y_1} b_2^{y_2} \dots b_i^{y_i}$$

and

$$c_1^{z_1} c_2^{z_2} \dots c_j^{z_j}$$

(where b_ℓ and c_ℓ are elements of the basic alphabet Σ and y_ℓ and z_ℓ equal either 1 or $*$; this should also cover the boundary case $i = 0$, i.e., $\beta = \lambda$ [the empty word]) left-aligned *if $y_k = z_k$ for $1 \leq k \leq K$, where $K \leq \min(i, j)$ is defined as the largest non-negative integer obeying $b_k = c_k$ for all $1 \leq k \leq K$ and $b_{K+1} \neq c_{K+1}$; here by convention let $b_{i+1} = c_{j+1} =$ \$ be a special symbol not contained in the basic alphabet Σ. Observe the special case that two identical union-free regular expressions are not left-aligned according to our definition.*

We call a regular expression simple looping *if it is either empty (denoting the empty language), or a single union-free RE, or the finite union of pairwisely left-aligned expressions α such that either $\alpha = \lambda$ or they can be written in the following* normal form*:*

$$\alpha = a_1^{k_1} a_1^{x_1} a_2^{k_2} a_2^{x_2} \dots a_K^{k_K} a_k^{x_K}$$

where

- *all a_i are single symbols from the basic alphabet Σ,*
- *each of the a_i are different from its (at most two) "neighbors" (i.e., a_{i-1} if $i > 1$, and a_{i+1} if $i < K$),*
- *each k_i is some positive integer, and*

– *each* x_i *equals either* 0 *or* $*$ *(if* $x_i = 0$*, the part* a_i^0 *of the expression will not be written down, since it denotes the empty word).*

Call a language L simple looping *if there is a simple looping regular expression describing* L*. The corresponding language class is denoted by* $\mathcal{SL}$*.*

Definition 2 (simple looping automata). *A deterministic finite automaton is called* simple looping *if*

– *when deleting all the loops in the automaton graph, the resulting (arc-labeled) so-called* skeleton graph *would be a directed tree without multiple arcs such that any node of outdegree larger than one in that tree has the property that all emanating arcs carry pairwisely different labels, and if*
– *the label* a *carried by some loop arc (at some state* s*) in the automaton graph is likewise carried by the necessarily existing other arc pointing to* s*.*

Observe that the first property of the previous definition ensures that the only cycles in the automaton graph are loops.

Lemma 1. *A simple looping DFA is not necessarily minimal. Conversely, if* L *is simple looping, then the minimal automaton* $A(L)$ *is not necessarily simple looping.*

Proof. If $a \neq b$ are two letters, then $A(\{a, b\})$ has two states, while this automaton is not simple looping, since it contains multiple arcs, namely two arcs (labeled a and b, respectively) from the start state to the final state. The corresponding simple looping DFA has three states (and is hence not minimal): one of its final states is reached via an a-transition from the start state, while the other final state is reached by a b-transition from the start state.

Theorem 1 (Characterization Theorem). *The following conditions are equivalent for a given language* $L \subseteq \Sigma^*$*:*

– L *is simple looping.*
– L *is generatable by a simple looping DFA.*

Proof. If L is simple looping, then there is a simple looping regular expression E describing L, consisting of pairwisely left-aligned union-free subexpressions E_i, $1 \leq i \leq m$. Each E_i can be straightforwardly turned into a DFA A_i which is simple looping. A part a^k of E_i is thereby turned into a path of length k within the automaton graph of A_i where each arc is labeled a; if a^* follows a^k in E_i, then the endpoint of the mentioned part will carry a loop labeled a. Since the expressions E_i are left-aligned, we can "overlay" the different A_i graphs to produce a DFA A which is simple looping.

Conversely, if L is generated by a simple looping DFA A, we can decompose A (due to the underlying "tree structure") into a collection of paths A_i (which may contain loops), such that each A_i contains only one final state, namely the last one on the path. (This means that "intermediate" final states are not considered to be final in "larger paths.") Each such A_i can be turned into a regular expression E_i whose collection yields the required simple looping regular expression E.

Since (as remarked upon introducing simple looping regular expressions) union-free "subexpressions" only turn up once, we can conclude:

Corollary 1. *Every simple looping language has a only one representation as a simple looping regular expression if we disregard the order in which the union-free components of the expression are listed.*

The following lemma relates $\mathcal{SL}$ with our learning algorithm. Recall that a generalized DFA can be turned into a "usual" DFA by possibly introducing "intermediate states" used for spelling the words labeling arcs. If A is a generalized DFA, let DFA(A) denote the DFA obtained in this way. The following lemma is immediate from the definition, yet crucial.

Lemma 2. *If A is the generalized DFA obtained as output of the learning algorithm* `generalize-simple`*, then DFA(A) is a simple looping DFA.*

3.5 $\mathcal{SL}$ can be learned from positive data

Let us now discuss the identifiability of $\mathcal{SL}$. To this end, let us first describe how to obtain a characteristic sample for a simple looping DFA A.

For every final state s, there is a unique path from the initial state s_0 of A to s in the skeleton graph (tree) associated to A: upon reading the labels of the arcs, this yields a unique word $w_s \in L(A)$. In fact, w_s is the shortest word in $L(A)$ that is accepted via state s. Observe that the collection L_F of all those w_s pass through all states of A, since A has no useless states. Therefore, we may extend our notation, so that for each state s (not only for final ones), w_s refers to some word from L_F such that w_s passes through s. Then, let u_s be the initial part of w_s that describes how to get from the initial state s_0 into s. Let v_s refer to the remaining part of w_s, i.e., $w_s = u_s v_s$. Now, if A has a loop labeled a at state s, let $w_s^\circ = u_s a v_s$. Let L° collect all such loop indicating words w_s°. Then, $\chi(L) = L_F \cup L^\circ$ is a characteristic sample of L.

Namely, upon getting the sample $\chi(L)$, L_F will provide the information to construct the skeleton graph (which corresponds to the prefix block tree discussed above, except that block words are spelled out). Then, the loops are reconstructed using the words from L°.

In our example from Fig. 3(a), the characteristic sample would be

$$\chi = \{ab, abc, abab, ababa, aab, abb, ababb\}.$$

Notice that this is larger than the input sample I_+ in Ex. 1 due to the following reasons: (i) ab and $abab$ are prefixes of other words in L_F and could hence be omitted; (ii) we introduce one loop-generating word per loop in our systematic construction of L°, ignoring the possibility of describing more than one loop in a (longer) word.

Finally notice that the algorithm we presented so far for learning $\mathcal{SL}$ languages is a "bulk algorithm" in the sense that it expects the whole sample at one gulp. From the point of view of practice, also incremental algorithms are

interesting. It is not too hard to convert the given algorithm into an incremental one. Firstly, we would scan a new word w with the automaton derived so far. If w is accepted, we continue with the next input word. If not, this can have two reasons: (i) either, we have reached a state that has not been marked final; then, marking this state final will cope with the situation; or (ii) we would have to use a yet non-existing transition in a certain state. In that case, we might (a) either introduce a loop in the state we are just dealing with, or (b) we will leave the state via a new arc to a new state, and the remainder of the word will create more new states. Notice that also the case (a) might in continuation lead to cases (i) or (ii)(b) upon further reading of w by means of the (modified) automaton.

We summarize our findings:

Theorem 2. *$\mathcal{SL}$ is identifiable in the limit (also with polynomial update time).*

4 Extensions of our approach

4.1 Dealing with multiplicities

In Sec. 2, we described another way of dealing with loops: namely, we described how to actually count multiple occurrences of a letter upon looping. It is not hard to see how this can be actually incorporated into the framework developed in the preceding section.

More precisely, we would now allow looping regular expressions that contain loops of the form $(a^n)^*$, or a^{n*} for short. This would have to be incorporated into the left-alignment definition (allowing y_ℓ/z_ℓ to be 1 or $n*$, not only 1 or $*$, in the notation of that definition). This way, we can describe a superclass $\mathcal{SL}'$ of $\mathcal{SL}$. This would naturally also entail that the corresponding looping DFA may contain "larger loops." The inference algorithm itself is again very similar, except that when it observes, e.g., $\{b, bbb\}$ as arc label of a resulting NFA, it would create a loop with arc label bb in the resulting (general) DFA. Keeping in mind the intention of the characteristic sample definition (to exercise each transition at least once), also that definition can be extended.

This reasoning allows us to conclude:

Theorem 3. *$\mathcal{SL}'$ is identifiable in the limit.*

4.2 Introducing wildcards

There is another type of generalization that one easily comes up with when thinking of *easy* regular expressions: the use of wildcards, maybe in the slightly general form of introducing character groups. It is exactly the generalizing capacity formalized in $\mathcal{SL}$ together with the ability of forming character groups that has been realized in the SWYN system of Blackwell [4].

Introducing wildcards (for simplicity) on top of the $\mathcal{SL}$-mechanism means that we are finally arriving at an identifiable language class different from $\mathcal{SL}$ (so not generalizing $\mathcal{SL}$ as we did in the preceding section). How can we find

places where we might wish to insert wildcards (denoted by the letter $'?'$ in what follows, assuming that this is not part of the usual alphabet we are dealing with)? The easiest way to find such places is to look at it in terms of preprocessing, given an input sample I_+:

Algorithm 3 (`preprocess-wildcards`(I_+)*).*

1. Form the set of blocks $[I_+]$ consisting of all sequences of block letters of all words from I_+ while ignoring multiplicity information.
2. Pairwisely left-align the blocks in $[I_+]$; whenever we find a pair (x, y) with
   - $x = [a_1] \dots [a_m][a][a_{m+2}] \dots [a_n]$ and
   - $y = [b_1] \dots [b_m][b][b_{m+2}] \dots [b_\ell]$

   such that, for all $i = 1, \dots, m, m+2, \dots, \min(n, \ell)$, either $a_i =?$ or $b_i =?$ or $a_i = b_i$; then we can replace both a and b by $'?'$. Moreover, we would replace a_i by $'?'$ if $b_i =?$ and (conversely) b_i by $'?'$ if $a_i =?$. Notice that in accordance with our usage of block letters, the actual interpretation of $'?'$ depends on its context: so, if we result in [?][?], then this refers to two different basic letters upon unfolding into the usual alphabet.
3. If the second step permits no further changes, we go back to the original input I_+ and replace letters by $'?'$ whenever such a change was indicated by the second step (working on block letters), yielding a new instance $I_+^?$.

Then, we run the procedure for $\mathcal{SL}$-learning (or say for $\mathcal{SL}'$-learning) on $I_+^?$.

Let us explain the wildcard usage by means of a little example. Consider $I_+ = \{ab, ac, bc\}$. Hence, $[I_+] = \{[a][b], [a][c], [b][c]\}$. Matching $[a][b], [a][c]$ gives $[a][?]$. Alternatively, we can match $[a][?], [b][c]$ yielding $[?][?]$. Since this last expression matches (subsumes) all original ones, we arrive at $I_+^? = \{??\}$.

Starting with the less pathological sample $I_+ = \{ababb, aabb, ababa, abc\}$ from Ex. 1, we get

$$[I_+] = \{[a][b][a][b], [a][b], [a][b][a][b][a], [a][b][c]\}.$$

Matching $[a][b][a][b]$ against $[a][b][c]$ yields $[a][b][?][b]$ and $[a][b][?]$. Similarly, we might match $[a][b][a][b][a]$ against $[a][b][c]$. Hence, $I_+^? = \{ab?bb, aabb, ab?ba, ab?\}$. The (finally) resulting DFA is depicted in Fig. 3(b).

How can we characterize the class of languages that is identifiable in this way? This can be easiest explained using simple looping expressions that may contain the special letter $'?'$. We can define a compatibility relation of two left-aligned, union-free expressions if we consider the "mergibility" of the corresponding block letter words as described above. Then, we would only allow looping expressions (that may contain $'?'$) that are decomposed as finite union of expressions which are incompatible in the sense sketched above. For reasons of space, we omit the corresponding technical details. These details become even more cumbersome if we aim to generalize $\mathcal{SL}'$ or if try to introduce *character groups* (as "lowercase letters", "alphanumerical symbols", etc., as known from certain application areas), thus enabling to have different sorts of wildcards. However, let us state that all these ways of introducing wildcards lead to identifiable language classes that can be syntactically characterized by certain forms of regular expressions.

5 A possible application: learning DTDs

XML. The expectations surrounding XML (eXtendible Markup Language) as universal syntax for data representation and exchange are huge, as underlined by the amount of effort being committed to XML by the World Wide Web Consortium (W3C) (see `www.w3.org/TR/REC-XML`), by the huge number of academics involved in the research of the backgrounds of XML, as well as by numerous private companies. Moreover, many applications arise which make use of XML, although they are not directly related to the world wide web. For example, nowadays XML plays an important role in the integration of manufacturing and management in highly automated fabrication processes such as in car companies [7]. For further information, refer to `www.oasis-open.org/cover/xmlIntro.html`.

XML grammars. The syntactic part of the XML language describes the relative position of pairs of corresponding *tags*. This description is done by means of a *document type definition* (DTD). Ignoring attributes of tags, a DTD is a special form of a context-free grammar. This sort of grammar formalism has been formalized and studied by Berstel and Boasson [3], defining *XML grammars*.

Three applications of grammatical inference. As already worked out by Ahonen, grammatical inference (GI) techniques can be very useful for automatic document processing, see [1, 2]. Building upon her work, we described [8] three possible applications of grammatical inference in the context of DTD inference:

- to assist designing grammars for (semi-) structured documents;
- to create views and sub-documents; and
- to optimize the performance of database queries based on XML by the help of adequate DTDs.

To underline the first of these applications, let us quote Tim Bray, one of the "fathers" of XML, who wrote (see `www.xml.com/axml/notes/Any1.html`):

Suppose you're given an existing well-formed XML document and you want to build a DTD for it. One way to do this is as follows:

1. Make a list of all the element types that actually appear in the document, and build a simple DTD which declares each and every one of them as ANY. Now you've got a DTD (not a very useful one) and a valid document.
2. Pick one of the elements, and work out how it's actually used in the document. Design a rule, and replace the ANY declaration with a ... content declaration. This, of course, is *the tricky part*, particularly in a large document.
3. Repeat step 2, working through the elements ..., until you have a useful DTD.

Instead of explaining these notions formally (as done in [8]), let us rather discuss a small but realistic example.

Fig. 4 shows the first lines of a short novel in somewhat simplified XML format. Disregarding the actual contents (i.e., the novel itself), we obtain the following possible samples of what a paragraph could be: it could consist in one sentence (see the headline etc.), of three sentences (first paragraph of the

novel itself) or two (due to cutting short the novel after the introductory part). Hence, our learner would generalize these examples to express that a paragraph could consist of one or more sentences. However, all learning algorithms that we proposed would insist in a chapter consisting of at least two paragraphs according to the examples seen up to now. Both ways of generalization are according to common sense: chapters with one paragraph are rarely observed.

```
<book>
  <part>
    <chapter>
      <paragraph>
        <sentence>Die Verwandlung</sentence>
      </paragraph>
      <paragraph>
        <sentence>von Franz Kafka</sentence>
      </paragraph>
    </chapter>
    <chapter>
      <paragraph>
        <sentence>I</sentence>
      </paragraph>
      <paragraph>
        <sentence>Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte,
                  fand er sich in seinem Bett zu einem ungeheueren Ungeziefer verwandelt.
        </sentence>
        <sentence>Er lag auf seinem panzerartig harten Rücken und sah, wenn er den Kopf ein wenig hob, seinen gewölbten,
                  braunen, von bogenförmigen Versteifungen geteilten Bauch, auf dessen Höhe sich die Bettdecke,
                  zum gänzlichen Niedergleiten bereit, kaum noch erhalten konnte.
        </sentence>
        <sentence>Seine vielen, im Vergleich zu seinem sonstigen Umfang kläglich dünnen Beine
                  flimmerten ihm hilflos vor den Augen.
        </sentence>
      </paragraph>
      <paragraph>
        <sentence>"Was ist mit mir geschehen?", dachte er. </sentence>
        <sentence>Es war kein Traum. </sentence>
     </paragraph>
    </chapter>
  </part>
</book>
```

Fig. 4. The first lines of Kafka's short novel "Verwandlung" in XML format.

6 Conclusions

We are aware of the fact that our proposed learners can only generate very simple kinds of regular expressions. However, first of all this is intrinsic to the learning model we used, since text learning (identification in the limit from positive samples) does not allow to learn all regular languages (be them encoded by automata or expressions); even the class of languages that definable by all REs (based on the operations union, catenation and star) without nested stars is *not* identifiable from positive data: ponder the sample $w_n = (ab)^n$ versus $(ab)^*$.

Secondly, there do exist applications for at least similarly simplistic versions of regular expressions. For example, the path expressions as discussed in [6] in the context of XML path queries are of a similar simplicity.

Moreover, many web browsers do only admit a limited form of regular expressions which pretty much resemble the subset we propose. We already mentioned the SWYN system [4] that employs practically the same sorts of expressions.[4] However, it would be nice to extend the results of this paper in a direction that allows longer strings (containing different sorts of letters) to be starred. Finding

[4] From the system description, it is not quite clear how the alignment is actually performed, so there might be some technical differences.

a reasonable subclass of regular languages that can be inferred in this way remains a topic of future study, obviously limited by the observed sample in the first paragraph.

We have also considered the (non-)closure properties of $\mathcal{SL}$ (all standard operations but intersection show non-closure behavior) and have derived an alternative, more compact characterization of $\mathcal{SL}$ in terms of REs. We skipped details here for reasons of space.

References

1. H. Ahonen. Disambiguation of SGML content models. In C. Nicholas and D. Wood, eds., *Principles of Document Processing PODP'96*, volume 1293 of *LNCS*, pp. 27–37. Springer, 1997.
2. H. Ahonen, H. Mannila, and E. Nikunen. Forming grammars for structured documents: an application of grammatical inference. In R. C. Carrasco and J. Oncina, eds., *International Colloquium on Grammatical Inference ICGI'94*, volume 862 of *LNCS/LNAI*, pp. 153–167. Springer, 1994.
3. J. Berstel and L. Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, August 2002.
4. A.F. Blackwell. SWYN: A visual representation for regular expressions. In H. Lieberman, ed., *Your wish is my command: Giving users the power to instruct their software*, pp. 245–270. Morgan Kaufmann, 2001.
5. A. Brazma. Efficient learning of regular expressions from approximate examples. In R. Greiner, T. Petsche, and S. J. Hanson, eds., *Computational Learning Theory and Natural Learning Systems, Vol. IV: Making Learning Systems Practical*, chapter 19, pp. 337–352. Cambridge (MA) USA: MIT Press, 1997.
6. Y. D. Chung, J. W. Kim, and M. H. Kim. Efficient preprocessing of XML queries using structured signatures. *Information Processing Letters*, 87:257–264, 2003.
7. CZ-Redaktion. Maschinenmenschen plaudern per XML mit der Unternehmens-IT. *Computer Zeitung*, (50):30, December 2000.
8. H. Fernau. Learning XML grammars. In P. Perner, ed., *Machine Learning and Data Mining in Pattern Recognition MLDM'01*, volume 2123 of *LNCS/LNAI*, pp. 73–87. Springer, 2001.
9. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: learning document type descriptors from XML document collections. *Data Mining and Knowledge Discovery*, 7:23–56, 2003.
10. E. M. Gold. Language identification in the limit. *Information and Control (now Information and Computation)*, 10:447–474, 1967.
11. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading (MA) USA: Addison-Wesley, 1979.
12. Ph. D. Laird. *Learning from Good and Bad Data*. Norwell (MA) USA: Kluwer Academic Publishers, 1988.
13. C. G. Nevill-Manning and I. H. Witten. Online and offline heuristics for inferring hierarchies of repetitions in sequences. *Proc. IEEE*, 88:1745–1755, 2000.
14. T. C. Smith, I. H. Witten, J. G. Cleary, and S. Legg. Objective evaluation of inferred context-free grammars. In *Proc. Australian and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia*, November 1994.
15. M. van Zaanen. *Bootstrapping Structure into Language: Alignment-Based Learning*. PhD, School of Computing, University of Leeds, UK, September 2001.