Information and Computation 207 (2009) 521–541

Contents lists available at ScienceDirect

Information and Computation

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

Algorithms for learning regular expressions from positive data☆

Henning Fernau

Universität Trier, FB 4, Abt. Informatik, Germany

ARTICLE INFO

Article history:
Received 26 October 2007
Revised 5 December 2008
Available online 24 January 2009

Keywords:
Regular expressions
Identification in the limit from positive data
Text learning

ABSTRACT

We describe algorithms that *directly* infer very simple forms of 1-unambiguous regular expressions from positive data. Thus, we characterize the regular language classes that can be learned this way, both in terms of regular expressions and in terms of (not necessarily minimal) deterministic finite automata.

1. Introduction

Over the last about forty years, many algorithms have been proposed and implemented that are capable of inferring 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, cf. the discussion in [28]. We are addressing this drawback by presenting learning algorithms for regular expressions that not only implement some of the features of learning algorithms used in practice, but also allow for characterizations of language classes that can be inferred 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* [30].

In practical applications of this learning scenario, learning regular languages often means to infer regular expressions (REs), because REs are arguably the most suitable model to specify regular languages, especially for human beings. Therefore, they (or variants thereof) are used in well-known tools such as `grep` in UNIX. A detailed discussion of regular expressions in different variants and contexts can be found in [51]. Unfortunately, to our knowledge all (but one) learning algorithms with known characterizations of the learned language class are based on grammars and/or automata. There are only few learning algorithms that directly deal with regular expressions, see [8,14,37,39] and the literature quoted therein, but they are not addressing the text learning model we are using here. Only recently (and after the publication of the conference version [25] of this paper), in the context of DTD inference for XML documents, another characterizable learning algorithm for regular expressions has been proposed in [7]. The identifiable language class discussed in [7] is, in a sense, even more restricted than the ones presented in this paper, because it is a subclass of testable languages and hence finite. In a broader sense, the

☆ Much 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 main algorithm described in this paper, is gratefully acknowledged. Part of the work has also been performed while the author was with Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, and with the University of Hertfordshire, Hatfield, UK.

E-mail address: fernau@uni-trier.de

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.12.008

transformation learning exhibited in [43] can also be interpreted as a learner for regular expressions, again in the context of XML.

The disadvantage of basing RE learning on the inference of deterministic finite automata (DFA) 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., [34]), these are often clumsy and lengthy, containing lots of "nested loops" and seeming repetitions of subsequences. This claim was empirically validated by Blackwell in [8], and there are several reasons for this blow-up:

1. Classical constructions turning DFAs into REs will introduce lots of repetitions and parenthesized sub-expressions, which makes it possible to get from an n-state DFA to a RE of length c^n (exponential blow-up), even for fixed (small) alphabet size, as shown in [31].
2. Even DFAs that do not contain loops (hence describing only finite languages) may exhibit a quadratic blow-up in size, due to repeated sub-structures; formally, this can be also seen by Fibonacci-subgraph structures that might be found in automata graphs, see [38].

Both scenarios can also occur within classes of regular languages that are identifiable from positive samples, like the 0-reversible languages introduced by Angluin [3].

Quite easily a human who wants to check the outcome of the automatic learning procedure might be 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 [23] for practically relevant examples, as well. 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 [28] 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 to give a better intuitive "explanation" of the observed data (sample strings), we are going to propose a learning procedure which only generates REs of star height one, i.e., starred expressions get never nested. Although the corresponding class of languages is thus restricted, many cases occurring in practice are covered. As it has been empirically verified in [7], also humans obviously tend to use only very simple regular expressions when specifying document type descriptions (DTD). Moreover, we avoid the basic reason for exponential-size of REs compared to DFA, namely unlimited star height, as proved in [31]. Hence, we can obtain one-to-one relations between DFAs and REs (for our restricted language class) in terms of their size.

Organization of the paper. In the next section, we will provide definitions of notions taken from the literature. The strategy of our learning algorithms is based on the concept of *alignment-based learning*, as explained in Section 3. We proceed by presenting a simple version of our learning algorithm, called $\mathcal{SL}$-infer, see Section 4. This (rather informal) discussion is made concise in Section 5, where we show that we can characterize the language class $\mathcal{SL}$ given by *simple looping regular expressions* that can be inferred by our learning algorithm both via a class of regular expressions and via specific DFA, called *simple looping automata*. There, also connections to minimal DFA are shown. Also, the main result of the whole paper is exhibited in that section, proving the learnability of $\mathcal{SL}$ (through the algorithm $\mathcal{SL}$-infer that has been presented before). In Section 6, we discuss several extensions of $\mathcal{SL}$-infer. More specifically, we show how multiplicity information and wildcards may be incorporated in that approach, leading to other learnable language classes. $\mathcal{SL}$-infer is further explained by applying it to the inference of DTD in Section 7. The reader primarily interested in that application could read this section before diving into the details of $\mathcal{SL}$-infer as put forward in the preceding sections. In Section 8, further variations and adaptations of our approach to similar learning tasks are sketched to conclude the paper.

2. Preliminaries

The purpose of this section is to fix some not(at)ions used in this paper and to discuss further relations with terminology of other papers on Learning Theory and on Grammar Induction. We will also make use of standard mathematical notations, as $|M|$ referring to the cardinality of set M.

2.1. Learning (identification/inference) in the limit from positive samples

In this well-established model, a language (target) 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 (*inductive*) *inference machine* (*IIM*) I (also called a *learner*) to which as input an arbitrary language $L \in \mathcal{L}$ may be completely enumerated (possibly with repetitions) in an arbitrary order, i.e., I receives an infinite input stream of words $E(1), E(2), \ldots$, where $E : \mathbb{N} \to L$ is an enumeration of L, i.e., a surjection, and I reacts with an output device stream $D_i \in \mathcal{D}$ (hypotheses stream) 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. We will also say that I is a learner for $\mathcal{L}$.

Notice that in the area of Learning Theory, usually the hypothesis space (i.e., the set from which the inference machine may select its hypotheses) is clearly distinguished from the target class. However, since we aim at presenting learning algorithms that are able to identify all languages from a concisely described language class, yielding normal forms thereof, we deal with so-called *exact learning* only, cf. [42]. More precisely, we will exhibit for the most important target class $\mathcal{SL}$ discussed in this paper a number of characterizations through two different forms of regular expressions, as well as two different forms of

finite automata; each of these forms of description actually yields a specific normal form for each $L \in \mathcal{SL}$ and can therefore be taken as concrete hypothesis space. Moreover, we mention in passing that some of the burdens of general Learning Theory are found easier with Grammar Induction; for example, we always deal with so-called indexable language classes. This makes several facts easier to phrase or to apply.

Further properties that all IIMs described in this paper satisfy (without further mentioning) are the following ones:

1. An IIM is called *iterative* (or sometimes also *incremental*) if its new hypothesis only depends on the previous hypothesis and the last input word.
2. A *conservative* IIM maintains its actual hypothesis at least as long as it has not seen data contradicting it.
3. A learner is called *consistent* if all its intermediate hypotheses do correctly reflect the data seen so far.
4. An IIM is *strong monotonic* if it always produces a stream of hypotheses describing an augmenting chain of languages, i.e., the new hypotesis language is always a superset of the previous one.[1]
5. An IIM is *rearrangement-independent* or *order-independent* if its hypothesis h obtained after having seen $E(1), \ldots, E(n)$ is the same as its hypothesis having seen $E(\pi_n(1)), \ldots, E(\pi_n(n))$, where π_n is an arbitrary permutation of $\{1, \ldots, n\}$.
6. An IIM is *set-driven*[2] if its hypothesis h obtained after having seen $E(1), \ldots, E(n)$ is the same as its hypothesis having seen $F(1), \ldots, F(m)$, with

$$\{E(1), \ldots, E(n)\} = \{F(1), \ldots, F(m)\}.$$

Notice that it is known that each of these properties alone (apart from the last one, see [32, Lemma 7.1.9]) presents a restriction of the learnable language classes, see [41,42,54,55]. However, from a practical point of view, it is quite desirable to have these properties. Therefore, we will present our IIM always as working in a batch mode, i.e., the IIM takes a finite language (a *sample*) and produces a hypothesis from this sample.

The claimed properties 1–6 are seen as follows: from a general perspective, the main reason why our IIM enjoys these properties is that IIM will follow a state-merging paradigm that can be paraphrased as follows: whenever inconsistencies with the envisaged form of automaton are detected, resolve these by state merging; this leads to a new hypothesis automaton that can be used to parse the next input example; if (and only if) that example cannot be accepted by the present hypothesis automaton, we extend the automaton by adding a new "acceptance path" for that particular new input word. This way, we ensure consistency with the data seen so far. Moreover, by the sketched construction, the learner is iterative and strong monotonic and hence conservative, see [55, Theorem 6]. Further discussions of these general properties of state-merging algorithms can be found in [21,22].

Given a concrete IIM I that is a learner for $\mathcal{L}$, we call a finite language $D_{L,I}$ a *characteristic sample* for $L \in \mathcal{L}$[3] if I outputs a hypothesis describing L, whenever $D_{L,I}$ has been enumerated to I (in any order, possibly with other words interleaved). It is known that, if $\mathcal{L}$ is identifiable, as witnessed by the IIM I, then each $L \in \mathcal{L}$ possesses a characteristic sample $D_{L,I}$, see [32, Section 7]. When dealing with consistent learners, we know that $D_{L,I} \subseteq L$. For state-merging algorithms, characteristic samples are often described by a set of words that exercises all nodes and arcs of a canonical finite automaton A accepting the target language L. By slight abuse of terminology, we will call a characteristic sample obtained this way a *characteristic sample for A*. Sometimes, such samples are also called (*structurally*) *complete* or *live complete* for A in the literature.

2.2. Regular expressions and regular languages

We include a formal definition of regular expressions, since several different notations are around, e.g., UNIX regular expressions differ from POSIX regular expressions, PERL regular expressions even allow non-regular languages to be described. A nice overview on different features of REs, including a discussion on the efficiency of parsing them, can be found in [19]. Also, textbooks use quite different notations. The situation seems to be more homogeneous with finite automata, so that we refrain from giving detailed definitions in that case, apart from Glushkov automata and generalized finite automata that are defined below.

Let Σ be an alphabet of symbols. Regular expressions over Σ are built from $\emptyset$ (indicating the empty language), λ (denoting the empty word), and the symbols in Σ using the binary operators $|$ (indicating union), $\cdot$ (indicating concatenation; this symbol is often suppressed for convenience), and the unary operator * (Kleene star). Sometimes, we also use the shorthand $^+$, where $x^+ = x \cdot x^*$. Likewise, i-fold iterations of the same sequence x may be abbreviated as x^i. At some places, it is also convenient to allow for an empty expression to denote the empty language. The language described by the regular expression E is denoted as $L(E)$.

To indicate different positions of the same symbol in a regular expression, we mark symbols with subscripts. For example, $(a_1|b_1)^* a_2 (a_3 b_2)^*$ is a marking of the expression $(a|b)^* a (ab)^*$. Marking is not unique, however, let $\mu(E)$ denote some marking of the expression E. Obviously, $\mu(E)$ is a regular expression over the alphabet $\mu(\Sigma)$ of marked symbols (derived from Σ by

[1] A different notion of monotonicity is discussed in [27].

[2] The latter two notions (again) seems to be used in a different manner in the Learning Theory and in the Grammar Induction communities; we mostly follow [32] here, but point the reader to different definitions, e.g., in [27], leading to results that contrast [32, Lemma 7.1.9], also confer [46].

[3] $D_{L,I}$ is also sometimes called *characteristic set* or *representative sample*; especially the latter notion is also used in a different meaning.

slightly overloading μ). The reverse of marking is the dropping of subscripts, indicated by σ. Hence, $E = \sigma(\mu(E))$ for any expression E over Σ, irrespectively of the chosen marking μ.

For each language L over Σ (and sometimes, to simplify notations, also for a regular expression E, referring to $L(E)$), we define:

1. $\text{first}(L) = \{b \in \Sigma \mid \exists w \in \Sigma^* : bw \in L\}$,
2. $\text{last}(L) = \{b \in \Sigma \mid \exists w \in \Sigma^* : wb \in L\}$, and
3. $\text{follow}(L, a) = \{b \in \Sigma \mid \exists v, w \in \Sigma^* : vabw \in L\}$.

These notions allow us to describe the *Glushkov automaton* G_E associated to a regular expression E over Σ along the lines of [17]: $G_E = (Q_E, \Sigma, \delta_E, q_I, F_E)$, where:

1. $Q_E = \mu(\Sigma) \cup \{q_I\}$, where $q_I \notin \mu(\Sigma)$ is the *initial state*.
2. For $a \in \Sigma$, $\delta_E(q_I, a) = \{x \in \text{first}(\mu(E)) \mid \sigma(x) = a\}$.
3. For $a \in \Sigma$, $x \in \mu(\Sigma)$, $\delta_E(x, a) = \{y \in \text{follow}(\mu(E), x) \mid \sigma(y) = a\}$.
4. $F_E = \text{last}(\mu(E))$ is the set of *final states*; add q_I to F_E iff $\lambda \in L(E)$.

A regular expression E is 1-*unambiguous* iff G_E is deterministic. A regular language L is 1-*unambiguous* iff $L = L(E)$ for some 1-unambiguous regular expression. As explained by Brüggemann-Klein and Wood in [17], 1-unambiguous regular expressions play an important role in the context of the ISO standard for the Standard Generalized Markup Language, SGML.

Recall now the notion of a *generalized nondeterministic finite automaton* (*NFA*): arcs of such an NFA may be labeled with words (or even finite set of words), not just single symbols. Accordingly, these words have to be found as subwords in the input upon scanning it. By way of contrast, a *generalized deterministic finite automaton* (*DFA*) is a generalized NFA which maintains the determinism property; this means that the labels of the arcs emanating from an arbitrary node form a prefix code, so that it is always clear where to continue upon scanning the input. Details can be found in [29]. Transitions will be sometimes denoted as $(s, w) \mapsto s'$, meaning that the automaton can transit from state s to state s' while reading the word w. Recall that a generalized DFA can be turned into a "usual" DFA by introducing, if necessary, "intermediate states" used for spelling the (longer) words labeling arcs. If A is a generalized DFA, let $\text{DFA}(A)$ denote the DFA obtained in this way.

For all types of finite automata considered in this paper, we will use notions like initial or final state as introduced for Glushkov automata. Likewise, when referring to the underlying automaton graph, we also speak of initial or final nodes. One note on deterministic finite automata (DFAs) should be in order here: in contrast to most textbooks on Formal Languages, but in line with the usual treatment within the Grammar Induction literature, the transition function of a DFA need not be completely defined, which in particular implies that a DFA with a two-letter input alphabet might possess a node with only one out-going arc. Naturally, such incomplete DFA can be made complete by adding one more trap state, whose only out-going arcs are loops. This peculiarity also applies to the notion of minimal DFA of a language L, denoted $A(L)$; again, a possibly contained trap state is left out.

3. 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 call any $[x^n]$ a *block letter* whenever x is a character in the alphabet over which the input words are formed; we call x the *basic letter* of $[x^n]$. When we only use *simplified 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:

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

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 are 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 were aligned starting with the third block of the other two samples, i.e.:

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

Due to the above psychologic argument and because the general problem of finding "best alignments" (in different senses) is NP-hard (since this is corresponding to a multiple sequence alignment problem, see [50]), 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* (MDL) principle, likewise known as *Occam's razor*, see [28,44,48] 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 be too simplistic, ignoring any type of multiplicity information.

Instead, one could choose integer numbers $\ell \geq 0$ and $r \geq 1$ such that, for each $1 \leq i \leq j$, r divides $k_i - \ell$, such that r is maximal with this property. Then, our generalization would be $x^\ell(x^r)^+$. So, $r = 1$ and $\ell = 0$ will always satisfy the first part of the property (and this would correspond to the generalization x^+ discussed above), but larger r will keep more information of the loop itself. 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.

Alternatively (and more simply), we might generalize, given $x^3, x^5, x^7, \ldots$, to x^3x^* (or, equivalently, x^2x^+), thereby ignoring the multiplicity information on the loops, but keeping multiplicity information of the part leading into the loop. In fact, this is the way our main algorithm $\mathcal{SL}$-infer will work (as detailed in the next section), although it is possible to modify it in order to cope with the kind of generalization strategy outlined in this paragraph; more details can be found at the end of the paper (Section 6).

On the downside, let us mention that any generalization strategy discussed in the preceding three paragraphs can (almost) only derive so-called strictly bounded languages, i.e., subsets of $x_1^*x_2^* \ldots x_k^*$, where the x_i are characters of the original alphabet, with x_{i+1} being distinct from x_i; see [9] as an early reference; strictly bounded languages originally require that $x_i = x_j$ implies $i = j$, while we may derive subsets of $a^*b^*a^*$. 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 Section 6.

4. Working out details of RE learning algorithms

We are going to describe and analyze our learning algorithm $\mathcal{SL}$-infer, which can be seen as a simplified version of the overall learning strategy proposed in this paper.

The following notions will be used throughout this section. For each symbol $a \in \Sigma$ and $I \subseteq \Sigma^*$, define the set

$$I(a) = \{w \in I \mid w \text{ starts with } a\}.$$

This yields a partition of I into the different sets $I(a)$ for $a \in \text{first}(I)$. Likewise, define

$$I^a = \{v \in b\Sigma^* \mid b \neq a \wedge \exists n > 0 (a^n v \in I(a))\}.$$

Given a set $I \subset \Sigma^+$, consider

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

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

4.1. Creating the start tree

The algorithm creating the *start tree* can be explained as follows for an alphabet Σ and an input sample $I_+ \subset \Sigma^+$, calling the recursive procedure `create-tree`(I_+). The tree that is created corresponds to the recursion tree of the procedure.

Algorithm 1. `create-tree`(I)

1. Compute $I(a)$, I^a for each symbol $a \in \Sigma$.
2. Recursively call, for all $a \in \Sigma$, `create-tree`(I^a).

It is obvious that we can associate to I_+ a rooted labeled directed tree (corresponding to the recursion tree), the root being labeled I_+ (with $I_+ = I_+^\lambda$), and the children of a node labeled I_+^x being labeled with the sets 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. 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. Notice that we can describe that resulting tree (alternatively) as the automaton graph of the prefix tree acceptor of $[I_+]$, which is obtained from I_+ by using simplified block letters after left-alignment of the words from I_+ as described in the previous sub-section.[4]

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 $I_+^{abab} = \{a\}$.

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, which are not displayed for simplicity. This gives the picture of Fig. 1. The alternative node labeling convention is displayed in Fig. 2.

4.2. Constructing NFAs by alignment

It is 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 initial state, and the final states are indicated by shading. 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 [53]:

> Aligned pieces of words are considered to be interchangeable.

However, this kind of generalization is not very aggressive, since it will always create only finite languages, given a finite sample. In order to arrive at infinite languages, loops must be introduced.

4.3. Introducing loops by determinization

We will introduce loops by turning the (generalized) NFA we obtained so far into a (generalized) DFA, where every arc carries exactly one label. To this end, if an arc α in the NFA is labeled with a finite set of $m > 1$ words, say $\{a^{n_1}, \ldots, a^{n_m}\}$, then only keep the shortest among these words as a label of α and enable the generation of the longer words by introducing a loop labeled a at the node α points to. The resulting generalized automaton is indeed deterministic, since in the situation sketched

[4] Prefix tree acceptors are the usual starting point of many inductive inference machines for subclasses of regular languages, see [3,32].

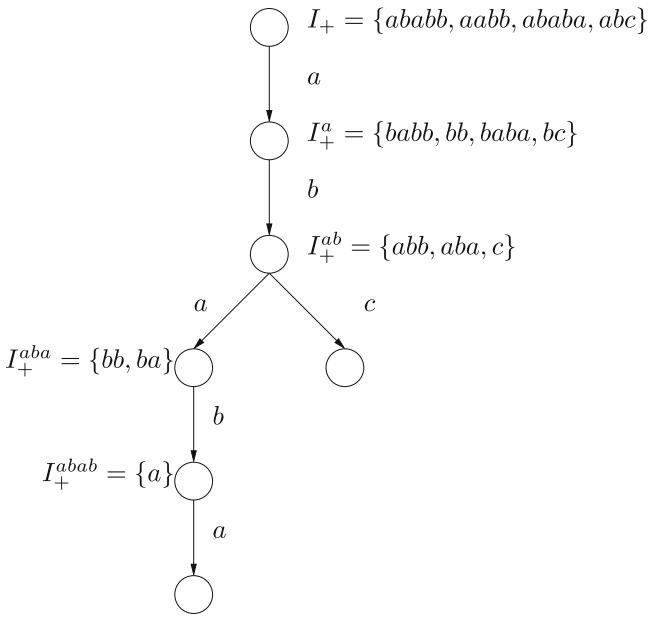


Fig. 1. A start tree; notice that we explicitly spelt out some of the sets which are used to label nodes.

above, in the original generalized NFA there was no arc labeled a starting at the node α 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.

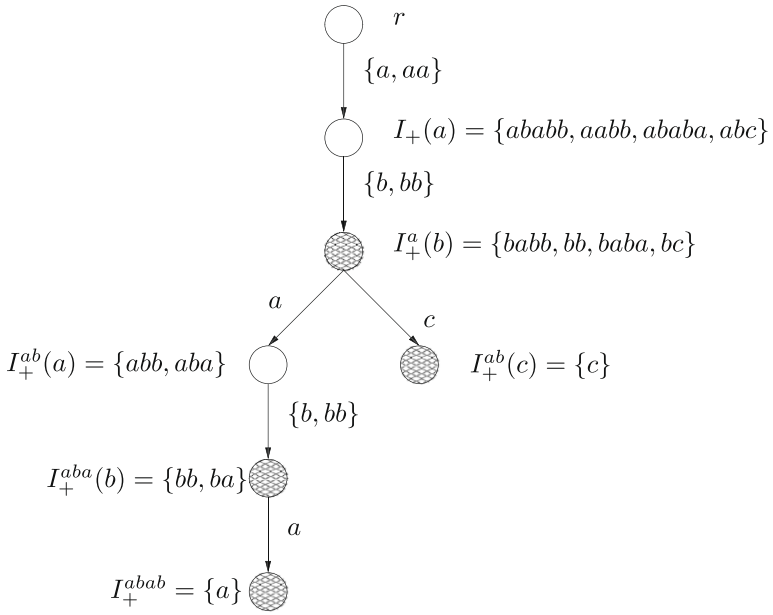
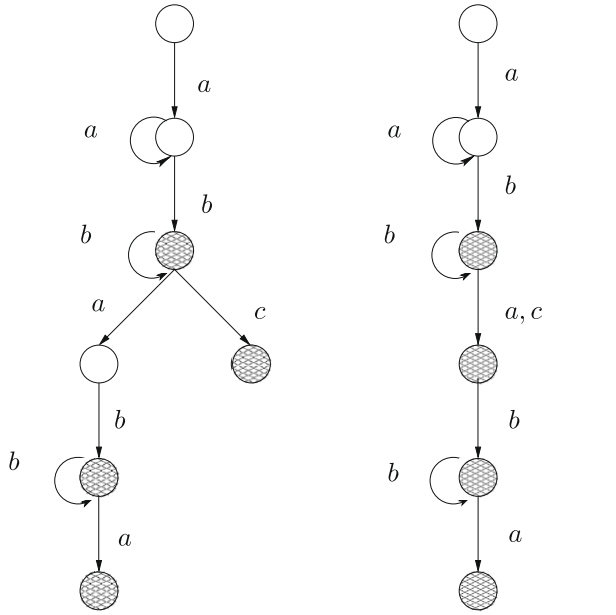


Fig. 2. The resulting generalized NFA.



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

Fig. 3. Two ways to generalize.

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 initial 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 "truly generalized."

`generate-tree` and `generalize-simple` together constitute our main learning algorithm, called $\mathcal{SL}$`-infer` for reasons becoming clear in the next section. Further examples showing how our learning algorithm works can be found at the end of Section 5.4. Variants of that algorithm will be discussed in Section 6, where we also explain how to obtain the generalization depicted in Fig. 3(b).

5. The language class $\mathcal{SL}$

5.1. Definitions and simple properties

Which language class can we learn with our algorithm $\mathcal{SL}$`-infer`? 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 and other classes of [descriptions of] languages. We now give two main definitions[5], restricting the usual notion of a regular expression and the usual notion of a DFA, which turn out to be two characterizations of the languages which can be learned by the algorithm $\mathcal{SL}$`-infer`.

5.1.1. Simple looping expressions

Definition 1 (*left-aligned expressions*). Consider two *union-free* regular expressions

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

and

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

[5] Notice that these definitions are in some respects different from the conference version [25] of this paper, since that version unfortunately contains some omissions.

(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]). By convention, let $b_{i+1} = c_{j+1} = \$$ be a special *endmarker* symbol not contained in the basic alphabet Σ. $K \leq \min(i,j)$ is defined as the largest non-negative integer obeying $b_k = c_k$ for all $1 \leq k \leq K$ and $b_{K+1} \neq c_{K+1}$. We call β and γ *left-aligned* iff $\beta \neq \gamma$ and $y_k = z_k$ for $1 \leq k \leq K$ and $b_K \notin \{b_{K+1}, c_{K+1}\}$.

Example 3. The language $L = \{a, aa\}$ cannot be represented by two left-aligned expressions. More specifically, it can be seen that we have to consider expressions for $\{a\}$ and $\{aa\}$, i.e., $\beta = a^1 = b_1^1$ and $\gamma = a^1a^1 = c_1^1c_2^1$. Hence, $K = 1$, but $b_K = b_1 = a \in \{b_2, c_2\} = \{a, \$\}$.

Definition 2 (*simple looping expressions*)**.** We call a regular expression *simple looping* if it is a finite union of pairwisely left-aligned expressions α such that either $\alpha = \lambda$ or they can be written in the following *normal form*:

$$\alpha = a_1^{k_1} a_1^{x_1} a_2^{k_2} a_2^{x_2} \ldots a_K^{k_K} a_K^{x_K}, \tag{1}$$

where

- all a_i are single symbols from the basic alphabet Σ,
- for all $1 \leq i < K$, $a_i \neq a_{i+1}$,
- 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 class of all simple looping languages is denoted by $\mathcal{SL}$.

Observe that any finite language can be written as a union of union-free expressions in the normal form of Eq. (1); however, in general union-free expressions in such an expression will not be left-aligned, e.g., $a \cup aa$, see Example 3.

5.1.2. Simple looping automata

Let us first introduce some auxiliary notions and operations on automata. A sequence $(s_1, s_2, \ldots, s_k)$ of $k > 2$ different states of a DFA A (with transition function δ and input alphabet Σ, $a \in \Sigma$) is called an *a-path* if:

1. $\delta(s_i, a) = s_{i+1}$ for $1 \leq i < k$.
2. The only outgoing arcs of s_i $(1 \leq i < k)$ in the automaton graph are given by item 1, except s_1 which might have an a-loop, with $b \neq a$.
3. The only ingoing arcs of s_i $(1 < i \leq k)$ are given by item 1., except s_k which might have an a-loop.
4. None of the states $s_2, \ldots, s_{k-1}$ is final.

An a-path is *maximal* if it is not part of a larger a-path. The (*path*) *contraction* $C(A)$ of an automaton A is obtained from A by replacing, for every input symbol a, every maximal a-path $(s_1, s_2, \ldots, s_k)$ by the transition $(s_1, a) \mapsto s_k$, and deleting all intermediate states $s_1, \ldots, s_{k-1}$.

For example, within the DFA given in Fig. 3(a), no$a$-paths exist, so that $C(A) = A$ in that case.

Since we only contract maximal a-paths, it does not matter in which order those contractions are undertaken, so we can state the following confluence property:

Lemma 1. *Given any finite automaton A, its contraction $C(A)$ is uniquely determined, i.e., it is independent of the contraction sequence of automata leading to $C(A)$.*

Definition 3 (*simple looping automata*)**.** A finite automaton A is called *simple looping* iff.

- when deleting all the loops and arc labels in $C(A)$, the resulting *skeleton graph* would be a directed tree without multiple arcs, also called the *underlying tree structure of A*, with one particular initial node with indegree zero (the root), and nodes of outdegree zero (leaves) being final states;
- all non-loop arcs to which a specific node of $C(A)$ is incident (independent of the direction of that arc) carry pairwisely different labels; and
- the label a carried by some loop arc (at some state node s) in the automaton graph is likewise carried by the necessarily existing other arc pointing to s.

For example, the DFA A displayed in Fig. 3(a) is simple looping, keeping in mind that $A = C(A)$.

5.1.3. Simple properties

Let us derive some simple properties of simple looping DFAs.

Lemma 2. *If A is simple looping, then the only cycles in the automaton graph of $C(A)$ are loops.*

Proof. If this were not the case, then $C(A)$ contains a larger cycle c. Not being loops, the arcs of c would not have been removed when constructing the skeleton graph of A (as described in the first item of the previous definition). Therefore, that skeleton graph would not be a directed tree. Hence, A is not simple looping, contradicting our assumption. □

As a direct consequence from the second and third item of the previous definition, we state:

Lemma 3. *Let s be a node of a simple looping automaton A that is not the initial node. Then, s has exactly one ingoing arc α (labeled, say, a) that does not originate in s. In addition, if there is a loop at s, it also carries the label a. Any other outgoing arc from s must carry a label different from a.*

The previous lemma immediately entails:

Corollary 1. *Every simple looping automaton is deterministic.*

Lemma 4. *If $L \subseteq \Sigma^*$ can be described by a simple looping DFA, then for any $a \in \Sigma$, L^a can be described by a simple looping DFA, as well.*

Proof. Let A be a simple looping DFA describing L, with initial state s_0. Let $a \in \Sigma$. Let $(s_0, s_1, \ldots, s_k)$ be the maximal a-path originating at s_0. Then, L^a is accepted by the DFA A^a obtained from A by

- removing the states $s_0, \ldots, s_{k-1}$ of A,
- declaring s_k to be the initial state s_0^a of A^a,
- removing the possibly existing loop labeled a at s_0^a, and
- removing all states (and incident arcs) that are not reachable from s_0^a.

Obviously, A^a is simple looping and $L(A^a) = L^a$. □

The idea of the construction in the previous lemma and the definition of simple looping automata itself yield:

Lemma 5. *Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ be the languages accepted by the simple loopings DFAs A and A', respectively. Then, $L = L'$ if and only if, for all $a \in \Sigma$, $L^a = (L')^a$ and $a^k L^a \subseteq L \iff a^k L^a \subseteq L'$ for all $k \geq 1$.*

Theorem 1. *Up to isomorphism, there is at most one simple looping DFA for a regular language L.*

Proof. Assume that A_1 and A_2 are two simple looping DFA accepting the same language L. Let $p_i = p(A_i)$ be the length of the longest path in the skeleton graph of A_i. If $p_1 > p_2$, then there would be a word $w \in a_1^+ a_2^+ \cdots a_{p_1}^+ \cap L$, where $a_i \in \Sigma$, $a_i \neq a_{i+1}$ for $1 \leq i < p_1$. Since $p_1 < p_2$, w cannot be accepted by A_2, i.e., $L(A_2) \neq L$, a contradiction. Hence, due to symmetry of A_1 and A_2, $p_1 = p_2$.

Clearly, if $L = \{\lambda\}$, then A_1 and A_2 are identical: they both consist of a single state which is both initial and final. In the following, let $L \neq \{\lambda\}$. Hence, we see that, for $p_1 = p_2 = 0$, A_1 and A_2 are isomorphic. This serves as the basis of our induction proof. Assume the claim is true for automata A_1 and A_2 with $p_1 = p_2 \leq k$. Consider automata A_1 and A_2 with $p_1 = p_2 = k + 1$ accepting $L \subseteq \Sigma^*$. Let $a \in \Sigma^*$. Due to the construction described in the proof of Lemma 4, L^a is accepted by the automata A_1^a and A_2^a, respectively. Clearly, $p_1^a = p(A_1^a) < p_1$ and $p_2^a = p(A_2^a) < p_2$. Moreover, the reasoning of the preceding paragraph yields $p_1^a = p_2^a$. By induction, A_1^a and A_2^a are isomorphic. In order to ensure that $L(A_1) = L(A_2)$, also the initial sequence of as (preceding a word from L^a) must be treated alike both in A_1 and in A_2, cf. the previous lemma. Since the argument holds for any such prefix symbol a, A_1 and A_2 are isomorphic.

The claim follows by the induction principle. □

Lemma 6. *A simple looping DFA is not necessarily minimal. Conversely, if L is simple looping, then the minimal DFA $A(L)$ of 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 initial 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 initial state, while the other final state is reached by a b-transition from the initial state. □

5.2. Characterization theorems

Theorem 2 (*Characterization Theorem*). *The following conditions are equivalent for an arbitrary 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, $1 \leq i \leq m$, i.e., $E = \bigcup_{i=1}^{m} E_i$, $L(E) = L$.

Each E_i can be turned into a DFA A_i which is simple looping, as detailed in the following. Namely, by Definition 2, $E_i = a_1^{k_1} a_1^{x_1} a_2^{k_2} a_2^{x_2} \ldots a_K^{k_K} a_K^{x_K}$, with k_j being natural numbers $k_j > 0$, and $x_j \in \{*, 0\}$. First, define a generalized finite automaton A accepting $L(E_i)$ by introducing the states $s_0, \ldots, s_{K+1}$, s_0 being initial and s_{K+1} being final, and the transitions $(s_j, a_j^{k_j}) \mapsto s_{j+1}$ (for $j = 1, \ldots, K$), plus loops $(s_{j+1}, a_j) \mapsto s_{j+1}$ (for $j = 1, \ldots, K$) if $x_j = *$. According to Definition 2, $a_j \neq a_{j+1}$ (for $j = 1, \ldots, K$), so that A is deterministic. A can be easily converted into a DFA A_i by introducing intermediate states. More specifically, if $k_j > 1$, introduce new states $q_{j,1}, \ldots, q_{j,k_j-1}$ and introduce transitions $(s_j, a_j) \mapsto q_{j,1}$, $(q_{j,1}, a_j) \mapsto q_{j,2}$, $\ldots$, $(q_{j,k_j-2}, a_j) \mapsto q_{j,k_j-2}$, $(q_{j,k_j-1}, a_j) \mapsto s_{j+1}$. These intermediate states are meant to "spell out" the words used as transition arc labels in the generalized DFA A. If $k_j = 1$, we take the transition $(s_j, a_j) \mapsto s_{j+1}$ within A_i (as within A). The loops are treated in A_i as in A. Since the contraction $C(A_i)$ can be obtained from A by replacing all arc labels of the form a^{k_j} by a, it is not hard to see that A_i is in fact simple looping.

Since the expressions E_i are left-aligned, we can "overlay" the different A_i automaton graphs to produce a DFA A which is simple looping. Observe how the left-alignment condition on simple looping expressions guarantees that the automaton obtained by overlaying two A_i, A_j is indeed deterministic, and that the conditions regarding the contractions for simple looping automata (see Definition 3) are maintained by a simple inductive argument.

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. The simple looping condition being valid for the A_i ensures that the corresponding expressions E_i are left-aligned. □

Since (as remarked upon introducing simple looping regular expressions) union-free "subexpressions" turn up only once, we can conclude with the help of Theorem 1.

Corollary 2. *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.*

Due to Lemma 6, it is also interesting to illuminate the relationship between minimal automata and $\mathcal{SL}$.

However, as seen by previous examples, minimal automata might have less states than simple looping automata for the same language. To get a characterization of $\mathcal{SL}$ in terms of minimal DFA, we need to "unmerge" certain states. Namely, assume that s is a node of a DFA A with m incoming arcs (not counting loops) with $m > 1$. Let $b_1, \ldots, b_m$ be the labels of the arcs connecting t_i with s (when carrying label b_i). Moreover, there is a certain number of outgoing arcs connecting s with u_i (carrying label c_i), $1 \leq i \leq n$. Then, we replace s by m copies $s_1, \ldots, s_m$ such that we have only one incoming arc into s_i, and this is coming from t_i and is labeled b_i; we have outgoing arcs connecting s_i with u_j labeled c_j ($j = 1, \ldots, m$); furthermore, if s is final, then all s_i are final states. The DFA $A' = T(A, s)$ obtained in this way accepts the same language as A. A' might again contain nodes with more than one incoming arc (disregarding loops). After repeated applications of the described procedure, starting from A we might arrive at an automaton $t(A)$ without multiple incoming arcs (not counting loops). We call $T(A)$ the *tree version* of A if $T(A)$ has the minimum number of states among all $t(A)$. For example, the classical prefix tree acceptor PTA(L) of a finite language L can be described as the tree version of the minimal DFA of L. For any DFA A, the multigraph obtained from A by removing the arc labels will be called the *skeleton graph of A*.

Lemma 7. *Let A be a DFA with the property that the removal of all loops from the skeleton graph of A results in a directed acyclic multigraph. Then, the tree version* $T(A)$ *exists and is unique up to renaming of states.*

Proof. The existence of a tree version can be seen as follows: consider some topological ordering $\prec$ of the nodes of the DFA A according to the arc relation of the skeleton graph (disregarding loops). This ordering exists due to the acyclicity assumption.

Now, we construct $A' = T(A, s)$ (as described above) for the first node with respect to the topological ordering $\prec$ with $m > 1$ incoming arcs (neglecting loops). The topological ordering $\prec'$ of the nodes in A' is inherited from $\prec$ as follows: we simply replace s by its m copies $s_1, \ldots, s_m$, i.e., whenever $a \prec s$ or $s \prec a$, respectively, then (for $1 \leq i \leq m$) $a \prec' s_i$ or $s_i \prec' a$, respectively. The order amongst the s_j is arbitrarily fixed. Notice that all s_j and all a with $a \prec' s_i$ have only one incoming arc (disregarding loops), since the skeleton graph of A is a directed acyclic multigraph by assumption. This acyclicity assumption

is also true for A', so that we can continue our argument with A' instead of A. Hence, after at most as many steps as there are states in A, we arrive at a tree version $t(A)$ of A.

We could have chosen a different sequence of states that should be split to avoid having more than one incoming arc. Observe that the number of copies of any state s in any tree version $t(A)$ equals

$$\sum_{t|s \text{ can be reached from } t \text{ in } A} (\delta^-(t) - 1),$$

where $\delta^-(t)$ denotes the indegree of t. Therefore, any way of obtaining some $t(A)$ yields a $T(A)$.

If $T(A)$ and $T'(A)$ are two tree versions of A, then an isomorphism between their state sets can be best defined "backwards" according to a topological sorting $\precsim$ ($\precsim'$, resp.) of the state set of $T(A)$ ($T'(A)$, resp.), starting by identifying the "last" corresponding states (where the correspondence is given by A) of $T(A)$ with those of $T'(A)$ and then moving backwards in the ordering. □

Example 4. The minimal DFA A accepting $a(aa)^*$ has no tree version; in fact, the procedure that exhaustively removes multiple incoming arcs by introducing new copies of nodes with multiple incoming arcs would never terminate.

Theorem 3 (*Characterization via minimal automata*)**.** *A language L is in $\mathcal{SL}$ iff its minimal automaton A satisfies the following conditions: removing all loops from the skeleton graph of A results in a directed acyclic multigraph, and the tree version $T(A)$ is simple looping.*

Proof. Assume first that L is described by a minimal automaton $A = A(L)$ satisfying the conditions above. Hence, $T(A)$ is simple looping, and $L(A) = L(T(A))$ according to our previous reasoning. Therefore, $L \in \mathcal{SL}$.

Conversely, if we perform the well-known stage construction for obtaining minimal DFAs, starting from some simple looping automaton A, then one can see by induction that the sequence of automata $A = A_0, A_1, \ldots, A_n = A(L(A))$ obtained in this way is (basically) reversing the construction for obtaining the tree version of a simple looping automaton described above. For example: first (induction base), final states will be merged when going from A_0 to A_1, and this will be the last bit that is "unmerged" in the construction described in detail above. This enables to show the following claim with the help of the previous lemma: $T(A(L(A)))$ is isomorphic to A. □

The last theorem (together with the previous Lemma) also shows that simple looping DFA can serve as canonical objects for $\mathcal{SL}$: up to renaming, two simple looping DFA recognizing the same language are identical. However, please notice.

Remark 1. As can be seen from our running example, the canonical simple looping automaton corresponding to a language from $\mathcal{SL}$ need not be a minimal-state DFA. In fact, since it is basically built from a prefix tree acceptor, often some states can be merged without changing the language. However, note that the blow-up that is obtained when going from a minimal automaton accepting a language from $\mathcal{SL}$ to the corresponding simple looping DFA is only polynomial in the number of states, since the number of edges stays unchanged.

The following lemma relates $\mathcal{SL}$ with our learning algorithm. It is immediate from the definition, yet crucial.

Lemma 8. *If A is the generalized DFA obtained as output of the learning algorithm* `generalize-simple`*, then DFA(A) is a simple looping DFA.*

5.3. $\mathcal{SL}$ 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 $T(A)$: 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 $L(A)$ being 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. Notice that there might be several words passing through s on their way to their specific final state; with the notation w_s, we refer to an arbitrary but fixed such word. Then, let u_s be the initial part of w_s describing 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°. Observe that due to our fixed choice of w_s, there is exactly one loop indicating word per loop in L°. Then, $\chi(L) = L_F \cup L^\circ$ is a characteristic sample of L.[6]

[6] Notice that the choice of w_s for a non-final state s may not be deterministic; this means that the notation $\chi(L)$ is strictly speaking not quite correct, since a number of languages is described, not a single one; however, since we do not care in the subsequent constructions which specific $\chi(L)$ is realized, we omit these technicalities.

In our example from Fig. 3(a), a characteristic sample for that particular language L would be $\chi(L) = L_F \cup L^\circ$, with

$L_F = \{ab, abc, abab, ababa\}$ and

$L^\circ = \{aab, abb, ababb\}$.

In this case, we always chose the shortest possible w_s to construct L°. Notice that this is larger than the input sample I_+ in Example 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 sample-oriented algorithm. From the point of view of practice, 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. Observe that the sketched incremental algorithm can be alternatively seen as an implementation of the guideline that adds any new word that cannot be scanned with the present hypothesis and then changes the resulting automaton, up to the point that it (again) satisfies the desired properties (to be simple looping in our case). Moreover, since the learning algorithm is surely strong monotonic, it can be viewed as an iterative learner by general results, see [40, Theorem 7].

We summarize our findings:

Theorem 4. *$\mathcal{SL}$ is identifiable in the limit from positive samples.*

Proof. If $L \in \mathcal{SL}$ is given, we can construct a characteristic sample $\chi(L)$. Observe that $L = \chi(L)$ whenever L is finite. Upon receiving $\chi(L)$, our algorithm will construct a simple looping expression E (or likewise a simple looping automaton A) whose language contains $\chi(L)$. This expression is unique as argued above (cf. Theorem 3). As the construction works, it is clear that $\chi(L) \subseteq L(E) = L(A)$. The algorithm can be also viewed as a merging state algorithm starting with the classical prefix tree acceptor, see [3,32].

Following the behavior of the learning algorithm closer, we observe: 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°. Hence, $L \subseteq L(A)$ is clear.

Conversely, if $w \in L(A) \backslash \chi(L)$, then w is exercising at least one loop in A $m > 0$ times, since $L(A)$ is infinite. Let w' be the word obtained from w by replacing the m "loopings" made by w by only one looping in w'; i.e., a subword a^m is replaced by a, such that $w = ua^m v$ and $w' = uav$. Obviously, $w' \in L(A)$. If $w' \in L(A) \backslash \chi(L)$, then there is another loop being executed by w'; we can completely cut out that loop to produce another word from $L(A)$, and we continue cutting out more and more loops until we finish up with a word $w'' = u'av'$ that is obtained from $w' = uav$ by cutting out a number of loops. Hence, $u'v' \in L(A)$ shows no loops, so that $u'v' \in L_F \subset \chi(L)$. By construction, all the loops in $L(A)$ are caused by exercising some loop of some automaton A' describing L (and hence they are testified in $\chi(L)$). Hence, we can reverse the process of cutting out loops, and we will always find a word in $L = L(A')$. Hence, $w \in L$, so that $L(A) \subseteq L$. □

5.3.1. Hypothesis spaces

In order to ensure convergence on the level of language descriptions, it is important to have canonical representations for the languages to be learned, i.e., any two descriptions of the same language are isormophic in a concise way. According to our previous language-theoretic assertions, we can provide a number of such well-suited hypothesis spaces, since all characterizations of $\mathcal{SL}$ that we describe also contain algorithmic transformations that allow to go from one representation of a language to another one. Hence, we can provide our IIM for $\mathcal{SL}$ with a "back-end" that produces output in any of the hypothesis spaces.

1. Simple looping expressions: They characterize the language class $\mathcal{SL}$ by definition. Due to Corollary 2, they offer a canonical representation.
2. Simple looping automata: Due to Theorem 2, they offer an alternative characterization of $\mathcal{SL}$. Moreover, Theorem 1 shows canonicity.
3. Minimal DFA: It is well-known that they describe all regular languages. The particular subclass of minimal DFA that actually characterize $\mathcal{SL}$ is described in Theorem 3. Clearly, they offer a canonical representation.
4. Compact simple looping expressions: They will be formally introduced in the section below. As will be seen, they offer a fourth characterization of $\mathcal{SL}$, see Lemma 12.

As already argued, the first and fourth descriptions (in terms of regular expressions) seem to be the most interesting ones.

5.3.2. Efficiency discussion

There are several ideas how to measure the efficiency of a Gold-style learner, see [32] for a more detailed discussion. The arguably weakest requirement for iterative learners is to look upon the time it needs to update its hypothesis. It is not hard to see from the previous exposition that only polynomial update time is needed for our learning algorithm $\mathcal{SL}$-infer.

Another complexity measure is the number of mind changes, i.e., revision of its hypothesis, that an inductive inference machine undergoes when learning a language. This complexity measure is not bounded in the case of $\mathcal{SL}$-infer, as can be seen from the example of the languages $L_k = \{(ab)^i \mid 1 \leq i \leq k\} \in \mathcal{SL}$.

5.4. Further properties of $\mathcal{SL}$

5.4.1. Closure and non-closure properties of the class $\mathcal{SL}$

Lemma 9. *$\mathcal{SL}$ is closed under intersection.*

Proof. Any words that might be contained in the intersection of two simple looping REs are in fact contained in the intersection of two union-free REs that are part of the given REs and whose corresponding block letter words (upon completely neglecting multiplicities) are identical. To obtain the intersection of such two union-free REs

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

and

$$\beta = a_1^{l_1} a_1^{y_1} a_2^{l_2} a_2^{y_2} \ldots a_K^{l_K} a_k^{y_K},$$

in normal form, we form

$$\gamma = a_1^{m_1} a_1^{z_1} a_2^{m_2} a_2^{z_2} \ldots a_K^{m_K} a_k^{z_K},$$

with $m_i = \min(k_i, l_i)$ and $z_i = 0$ if $0 \in \{x_i, y_i\}$ and $z_i = *$ if $x_i = y_i = *$. Now, γ describes the intersection of α and β iff we do not find an index j such that either $k_j < l_j$ and $x_j = 0$, or $l_j < k_j$ and $y_j = 0$. Otherwise, the intersection is empty. □

Lemma 10. *$\mathcal{SL}$ is not closed under union.*

Proof. Reconsider Example 3. □

Lemma 11. *The following provides a list of examples for languages that are not in $\mathcal{SL}$:*

- $\{a, b\}^+$, $\{a, b\}^*$;
- $(ab)^+$;
- $\{a, b\}^+ \setminus \{b\}^+$, $\{a, b\}^* \setminus \{b\}^+$.

Proof. The proof idea is always the same: observe how our learning algorithm would react upon seeing the sequence $w_n = (ab)^n$. Namely,

$$\bigcup_{j=1}^{n} (ab)^j$$

is a simple looping expression for each n. Hence, none of the above-listed languages will be ever hypothesized. □

These counter-examples immediately entail:

Corollary 3. *For $|\Sigma| > 1$, $\mathcal{SL}$ cannot describe the universal language Σ^*.*

Corollary 4. *$\mathcal{SL}$ is not closed under complement.*

Corollary 5. *$\mathcal{SL}$ is not closed under homomorphism.*

Proof. Consider the homomorphism $c \mapsto ab$ transferring the $\mathcal{SL}$-language c^+ into $(ab)^+$, the second counter-example. □

5.4.2. Compact simple looping expressions

We can also give an alternative characterization of $\mathcal{SL}$ in terms of regular expressions. Since the corresponding regular expressions tend to be more concise than the formerly given ones, we call them compact.

Definition 4 (*Compact simple looping expressions (CSLE)*)**.** A RE is called *compact simple looping* (over the alphabet Σ) if it is

- either *primitive*, i.e., of the form λ;
- or *atomic*, i.e., of the form a^k resp. $a^\ell a^+$ for some $k > 0$, $\ell \geq 0$, $a \in \Sigma$;
- or, if E, E' are CSLE starting with different letters, i.e., $\mathrm{first}(E) \cap \mathrm{first}(E') = \emptyset$, then $E \cup E'$ is a CSLE.
- or, if E, E' are CSLE with $\mathrm{first}(E) \neq \mathrm{first}(E')$ and if E' is atomic, then $E'(E)$, i.e., the concatenation of E' and (E) (where the parentheses are important to be put), is a CSLE.

Example 5. $bb^*(a)$ is CSLE, since bb^* is atomic, and $\mathrm{first}(bb^*) = \{b\} \neq \{a\} = \mathrm{first}(a)$. Since a is atomic, $a(bb^*(a))$ is CSLE, as well as c. Hence, $a(bb^*(a)) \cup c$ is CSLE. Therefore, $bb^*(a(bb^*(a)) \cup c)$ is CSLE (which explains why it is important to put parentheses in the last item of the definition). $aa^*(bb^*(a(bb^*(a)) \cup c))$ is a CSLE that exactly describes (or better, spells) the language accepted by the simple looping DFA from Fig. 3(a).

Let us state the characterization more formally:

Lemma 12. *A language $L \in \mathcal{SL}$ iff there is a CSLE describing that language. Moreover, up to permutations due to commutativity of the union operator, there is at most one CSLE for any regular language L.*

This lemma can be best understood when considering the characterization of $\mathcal{SL}$ via simple looping DFAs, keeping in mind their canonicity. Namely, CSLE describe exactly the branching structure of such DFAs. Therefore, they also tend to be more concise than simple looping REs explicitly spelling out all paths. More specifically, keeping the number of edges as the basic size measure of finite automata (which might be a bit unusual but actually reflecting the resources needed when storing the automata), the transformation between simple looping DFA and CSLE is linear (in both directions), contrasting the usual well-known descriptional complexity blow-up between these two description mechanisms. This property has been noted as very important when applying learning algorithms for regular expressions in a recent paper dedicated to learning DTDs for XML documents [7], an application being further sketched below. Hence, CSLE give a better (readable) target structure, at least as long as the nesting of parentheses is not too deep. With that particular application in mind, the following theorem is important.

Theorem 5. *$\mathcal{SL}$-languages are 1-unambiguous.*

Proof. Consider $L \in \mathcal{SL}$. Due to Lemma 12, there is a CSLE E for L. By the definition of CSLE, it is immediate that the corresponding Glushkov automaton G_E is deterministic. Hence, L is 1-unambiguous. □

Notice that we can compromise on the nesting depth of parentheses for regular expressions describing $\mathcal{SL}$-languages: For any given nesting depth (possibly user-defined), we can define a normal form that (as a first priority) tries to build up a parenthesised expression (from the final states/leaves of the corresponding DFA); however, if the upper bound on the parenthesis nesting depth of the expression is reached, then the path to the root (initial) node of the automaton is spelled out as with simple looping expressions.

5.4.3. Further remarks on looping expressions

Coming back to simple looping expressions, the reader might have wondered why we could not have started with a possibly simpler but more general definition, namely with arbitrary regular expressions where the use of the star operation is restricted to starring single symbols. However, this restriction is insufficient, since all languages from the superfinite class

$$\left\{ \{a^i \mid 1 \leq i \leq n\} \mid n \in \mathbb{N} \} \cup \{a\}^* \right\},$$

can be described this way, which entails that this language class is not identifiable in the limit according to Gold [30]. However, we can easily construct a simple looping expression E' for any expression E such that $L(E) \subseteq L(E')$ and such that there is no $L \in \mathcal{SL}$ with $L(E) \subseteq L \subsetneq L(E')$. Firstly, E can be transformed into the normal form $E = \bigcup E_i$ due to the valid distributive laws, where no E_i contains union and any two E_i are different. Then, we check if each E_i satisfies the normal form from Definition 2; a violation could be due to two different reasons:

- either E_i contains a part of the form a^*a^k; this can be equivalently replaced by a^ka^*;
- or E_i contains two loops, i.e., a part of the form a^*a^*; this can be equivalently replaced by one loop a^*.

Finally, we have to check if any two E_i and E_j are left-aligned. Whenever this condition is violated, we have to replace E_i and E_j by a common generalization, usually by introducing a star. More specifically, using the symbols β, γ and K as in Definition 1, whenever $y_k \neq z_k$ for $1 \leq k \leq K$, we will replace both by $*$; and if $b_K \in \{b_{K+1}, c_{K+1}\}$, then we set $y_K = z_K = *$ and possibly further change the newly obtained expressions according to the possible normal form violations as described above.

Notice that the described procedure can be seen as a variant of our learning algorithm that directly works on expressions; namely, we can view any sample $I_+ = \{w_i \mid 1 \leq i \leq J\}$ as the expression $\bigcup_{1 \leq i \leq J} w_i$ with which we could start the described procedure.

5.4.4. A final look at our IIM for $\mathcal{SL}$

Finally, it might be interesting to observe what happens if our algorithm is fed with languages it is not really meant to deal with.

Example 6. If $S = (a^n b^n \mid n \geq 1)$ is input sequence to our IIM for $\mathcal{SL}$, it will work as follows: consider the initial part $ab, aabb$. The IIM will find the blocks $[a][b]$. Since $[a]$ is representing a and aa, it will generalize to the sub-expression aa^*. Likewise, $[b]$ is generalized to the sub-expression bb^*. Hence, the hypothesis at this stage will be aa^*bb^*. This hypothesis will not be changed anymore upon reading further words from S. Hence, the IIM will over-generalize but converge in this non-regular case.

Example 7. If $S = ((ab)^n \mid n \geq 1)$ is input sequence to our IIM for $\mathcal{SL}$, it will work as follows: Consider the initial part $ab, abab, \ldots, (ab)^n$. The IIM will basically produce the canonical DFA accepting the language $\{(ab)^n\}$, together with several intermediate final states. Hence, the whole sequence would result in indefinite growth of the number of states and arcs of the hypothesis DFA. So, there are regular languages on which the algorithm does not converge.

Notice that the behaviour exemplified by the last two examples (possible convergence in case of some non-regular languages, possible divergence for some regular languages) is in contrast with what has been found with language learners akin to reversible languages, see [24]. There, even a well-definable sort of convergence can be observed for all regular languages.

6. Extensions of our approach

In this section, we briefly discuss two possible extensions of our approach: the first one contains a suggestion how a multiplicity count can be integrated in loops and the second one discusses a possibility how to deal with wildcards.

6.1. Dealing with multiplicities

In Section 3, we described another way of dealing with loops: namely, we described how to count multiple occurrences of a letter upon looping. It is not hard to see how this can be formally incorporated into the framework developed in the preceding section.

More precisely, we would now allow looping regular expressions containing loops of the form $(a^n)^*$, or a^{n*} for short. More specifically, a^{0*} denotes the empty word. This would have to be incorporated into the left-alignment Definition 1 (allowing y_ℓ/z_ℓ to be 1 or $n*$, not only 1 or *, in the notation of that definition), as well as in the simple looping Definition 2, regarding the x_ℓ.

This would naturally also entail that the corresponding looping DFA may contain longer cycles, but these could always be seen as simple loops in generalized DFA; more specifically, the only possible labels of loops would have the form a^n for some $n > 0$. More precisely, the following list of properties of generalized DFA describe the intended language class:

1. When removing all arc labels and all loops from the automaton graph, a tree will result.
2. All arc labels are of the form a^m for some input letter a and some $m > 0$.
3. If a^m and b^n are labels of two arcs incident to some node s, then either $a \neq b$, or $a = b$ and either a^m labels an incoming arc for s and a^n labels a loop on s, or vice versa.

This way, we can describe a superclass $\mathcal{SL}'$ of $\mathcal{SL}$. Namely, we can characterize $\mathcal{SL}$ by the generalized DFA for $\mathcal{SL}'$ by requiring that all loop labels have the form a^1. Therefore, $a(aa)^*$ describes a language from $\mathcal{SL}' \setminus \mathcal{SL}$.

The resulting inference algorithm $\mathcal{SL}'$-`infer` itself is again very similar to $\mathcal{SL}$-`infer`, except that when it gets $\{a, aaa\}$ as input, it would create a loop with arc label aa in the (generalized) DFA. This would lead to the expression $a(aa)^*$, while $\mathcal{SL}$-`infer` would produce aa^* as its hypothesis. 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 6. $\mathcal{SL} \subsetneq \mathcal{SL}'$.

Theorem 7. *$\mathcal{SL}'$ is identifiable in the limit.*

Alternatively, we might further simplify the permitted regular expressions by ignoring multiplicities. Then, union-free subexpressions would have the shape

$$a_1^{k_1} a_1^{x_1 *} a_2^{k_2} a_2^{x_2 *} \ldots a_m^{k_m} a_m^{x_m *},$$

with $k_i, x_i \in \{0,1\}$. Notice that such a restriction would also allow a simpler characterization via DFA, since we do not need to refer to the contraction operation. More specifically, we could use the characterization of $\mathcal{SL}'$ via generalized DFA, requiring in addition that the automata are non-generalized. Again, it is not hard to modify all definitions, assertions, and algorithms to show for a such-defined language class $\mathcal{SL}'' \subset \mathcal{SL}$ (a separating language being a^2a^*):

Theorem 8. *$\mathcal{SL}'' \subsetneq \mathcal{SL}$.*

Theorem 9. *$\mathcal{SL}''$ is identifiable in the limit.*

6.2. Introducing wildcards

There is another type of generalization one easily comes up with when thinking of *easy* regular expressions: the use of *wildcards*, i.e., placeholders for one arbitrary letter, maybe in the slightly more general form of introducing character groups. It is exactly the generalizing capacity formalized in $\mathcal{SL}$ together with the ability of forming character groups implemented in the SWYN system of Blackwell [8].

Introducing wildcards (for simplicity) on top of the $\mathcal{SL}$-mechanism means that we are finally arriving at an identifiable language class different from $\mathcal{SL}$ (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.
   Initially set $[I_+^?] = \emptyset$.
2. Pairwisely left-align the blocks in $[I_+]$; whenever we find a pair (x, y) with
   - $x = [a_1] \ldots [a_m][a][a_{m+2}] \ldots [a_n]$ and
   - $y = [b_1] \ldots [b_m][b][b_{m+2}] \ldots [b_\ell]$,

   such that for all $i = 1, \ldots, m, m+2, \ldots, \min(n, \ell)$, $a_i = b_i$. 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 $a_i =$?. Add the strings containing wildcard ? that can be obtained this way to $[I_+^?]$.
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 (on block letters in $[I_+^?]$), yielding a new instance $I_+^?$. Furthermore, add words from I_+ that are not (yet) described by words from $I_+^?$ to $I_+^?$.

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

Notice that, by our definition of block letters, the basic letters of subsequent block letters are different. This implies an according interpretation of the wildcard ?: It does not stand for any letter, but only for any letter different from the immediately preceding or following letters.

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 $[a][c], [b][c]$ yielding $[?][c]$. Hence, we arrive at $I_+^? = \{a?, ?c\}$. Since ab and ac can be described by $a?$, and bc is covered by $?c$, no further words have to be added to $I_+^?$.

Starting with the less pathological sample $I_+ = \{ababb, aabb, ababa, abc\}$ from Example 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, $I_+^? = \{ab?bb, aabb, ab?ba, ab?\}$. Notice that, in particular, $[a][b]$ (resulting from $aabb$) cannot be successfully matched against any other string from $[I_+]$ in the second step of the wildcard preprocessing algorithm to produce additional strings containing wildcards. Hence, $aabb$ is finally added to $I_+^?$. The DFA depicted in Fig. 3(b) is obtained by running the $\mathcal{SL}$-`infer`ence algorithm on $I_+^?$ and finally re-interpreting the wildcards in a proper way: Namely, since two consecutive block letters always designate different letters, the wildcard that would label the arc connecting the 3rd to the 4th node should be uniquely interpreted as a or c.

How can we characterize the class of languages being 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 into finite union of expressions which are incompatible in the sense sketched above. As with the other modifications of our main algorithm discussed so far, we omit the corresponding technical details. These details become even more cumbersome if we aim to generalize $\mathcal{SL}'$ or if we try to introduce *character groups* (as "lowercase letters", "alphanumerical symbols", etc., as known from certain application areas), thus allowing to have different sorts of wildcards. However, let us state (without formal verification) that all these ways of introducing wildcards lead to identifiable language classes that can be syntactically characterized by certain forms of regular expressions.

7. A possible application: learning DTDs

XML. The expectations surrounding XML (eXtendible Markup Language) as universal syntax for data representation and exchange are big, as underlined by the 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 WWW. 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 [20]. 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 [5], defining *XML grammars*, further recently studied in [6].

Three applications of grammatical inference. As already worked out by Ahonen and her co-authors, grammatical inference (GI) techniques can be very useful for automatic document processing, see [1,2]. Building upon her work, we described [23] 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 are 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 have 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 [23]), let us rather discuss a small but realistic example.

Fig. 4 shows the first lines of a short story 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 short story 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.

Also for XML documents, it is recommended to only use what is known as 1-unambiguous expressions, since the programs dealing with these documents might be based on SGML, see [4]. In the context of formalizing SGML content models, in particular Brüggemann-Klein and Wood studied such restricted REs, see [16,17]. As we have seen, compact simple looping regular expressions are indeed always 1-unambiguous. This offers a further advantage when applying our approach.

8. Prospects and future work

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)

```
<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 story "Verwandlung" in XML format.

does not allow to learn all regular languages (be these encoded by automata or expressions); even the class of languages that is definable by all REs (based on the operations union, catenation and star) without nested stars is *not* identifiable from positive data only: consider the sample $I_n = \{w_1, \ldots, w_n\}$ with $w_i = (ab)^i$ versus $(ab)^*$.

Secondly, there do exist applications for at least similarly simplistic versions of regular expressions. For example, the path expressions as discussed in [18] in the context of XML path queries are of a similar simplicity.

Linda Buisman implemented the main learning algorithm $\mathcal{SL}$-`infer` in C++ under our supervision. The author is happy to send the corresponding implementation to anyone interested. As output format, we chose CSLE due to improved readability. With the sketched XML application in mind, $\mathcal{SL}$-`infer` has also been integrated in a DTD learner. Notice that our output format ensures 1-unambiguity. The undertaken experiments look quite promising.

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 [8] that employs practically the same sorts of expressions.[7] 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 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 of this chapter.

- Sempere [47] discussed *linear expressions*, i.e., a variant of regular expressions characterizing the linear languages. In fact, the CONTROL operator discussed in [26] can be also re-interpreted for defining regular-like expressions characterizing other language classes.
  This way, the learning algorithms as described in this paper can also be used towards the learning of non-regular languages based on regular-like expressions.
- In order to generalize the learnable language classes related to reversible languages as introduced by Angluin in [3], we introduced the concept of function distinguishability in [24]. It might well be that such ideas are also helpful to broaden the approach sketched in the present paper in order to learn further, possibly more general regular language classes based on regular expressions. The basic idea is to help the learner by giving some additional bias information known as a distinguishing function. This remains a topic of future research.
- Regular expressions also exist for tree languages. More details can be found, e.g., in the chapter written by Gécseg and Steinby in [45]. in view of the many modern applications of tree languages, in particular in the area of XML processing, this could be a promising direction to follow.
- As also nicely argued in the slides of O. Hofmann, see [33], from applications motivated by computational biology, the question of inferring regular expressions in a probabilistic setting is worth exploring. From a theoretical point of view, this would first necessitate exploring "probabilistic regular expressions" as such. To our knowledge, such a concept does not yet exist.

As a further technical point, notice that as long as we are restricting ourselves to the text learning model, we are basically relying on the existence of normal forms for the language class under consideration. Even in the case of regular languages, this need not be the classical minimal finite automata. In fact, neither the canonical objects (simple looping automata) derived in

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

this paper are necessarily minimal (see Remark 1) nor the canonical objects defined in [24] are the classical minimal finite automata. However, in both cases, they are technically closely related, see Theorem 3 in the case of $\mathcal{SL}$. By way of contrast, there are no "classical" canonical minimal regular expressions. Therefore, to a certain extent one might also say that the algorithms presented in this paper rely on the existence of canonical minimal objects for DFA. This poses two (differently flavored) challenges:

- Find interesting classes of regular expressions that have genuine canonical objects that are not "somehow" related to DFA.
- Further extend DFA-based techniques by exhibiting classes of DFA that can be transformed into equivalent RE avoiding the "usual" exponential pay-off.

Acknowledgments

The author is grateful for all discussions and feedback received after the publication of the extended abstract of this paper at ALT 2005. Moreover, he is thankful to the anonymous referees of this journal who considerably helped improve the presentation of the material.

References

[1] H. Ahonen, Disambiguation of SGML content models, in: C. Nicholas, D. Wood (Eds.), Principles of Document Processing PODP'96, volume 1293 of LNCS, Springer, 1997, pp. 27–37.
[2] H. Ahonen, H. Mannila, E. Nikunen, Forming grammars for structured documents: an application of grammatical inference, in: R.C. Carrasco, J. Oncina (Eds.), Proceedings of the Second International Colloquium on Grammatical Inference (ICGI-94): Grammatical Inference and Applications, volume 862 of LNCS/LNAI, Springer, 1994, pp. 153–167.
[3] D. Angluin, Inference of reversible languages, Journal of the ACM 29 (1982) 741–765.
[4] H. Behme, S. Mintert (translators), Extensible Markup Language (XML) 1.0; Anhang E. Available from: <http://www.linkwerk.com/pub/xmlidp/2000/xml-spec-appe.
[5] J. Berstel, L. Boasson, Formal properties of XML grammars and languages, Acta Informatica 38 (2002) 649–671.
[6] A. Bertoni, C. Choffrut, B. Palano, Context-free grammars and XML languages, in: O.H. Ibarra, Z. Dang (Eds.), Developments in Language Theory DLT, volume 4036 of LNCS, Springer, 2006, pp. 108–119.
[7] G.J. Bex, F. Neven, T. Schwentick, K. Tuyls, Inference of concise DTDs from XML data, in: U. Dayal, K.-Y. Whang, D.B. Lomet, G. Alonso, G.M. Lohman, M.L. Kersten, S.K. Cha, Y.-K. Kim (Eds.), 32nd International Conference on Very Large Data Bases VLDB, ACM Press, 2006, pp. 115–126.
[8] 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, Morgan Kaufmann, 2001, pp. 245–270.
[9] M. Blattner, A. Cremers, Observations about bounded languages and developmental systems, Theory of Computing Systems 10 (1976) 253–258.
[10] A. Brazma, Inductive synthesis of dot expons, in: J. Barzdins, D. Bjørner (Eds.), Baltic Computer Science, volume 502 of LNCS, Springer, 1991, pp. 156–212.
[11] A. Brazma, Efficient identification of regular expressions from representative examples, in: Proc. of the Ann. Conf. on Computational Learning Theory COLT, ACM, 1993, pp. 236–242.
[12] A. Brazma, Learning a subclass of regular expressions by recognizing periodic repetitions, in: E. Sandewall, C.G. Jansson (Eds.), Proc. of the 4th Scandinavian Conference on Artificial Intelligence, IOS Press, 1993, pp. 137–146.
[13] A. Brazma, Learning of regular expressions by pattern matching, in: P.M.B. Vitányi, (Eds.), Computational Learning Theory, Second European Conference, EuroCOLT, volume 904 of LNCS/LNAI, Springer, 1995, pp. 392–403.
[14] A. Brazma, Efficient learning of regular expressions from approximate examples, in: R. Greiner, T. Petsche, S.J. Hanson (Eds.), Computational Learning Theory and Natural Learning Systems, volume IV of Making Learning Systems Practical, MIT Press, Cambridge, MA, 1997, pp. 337–352., chapter 19.
[15] A. Brazma, E. Kinber, Generalized regular expressions—a language for synthesis of programs with branching in loops, Theoretical Computer Science 46 (1986) 175–195.
[16] A. Brüggemann-Klein, Unambiguity of extended regular expressions in SGML document grammars, in: T. Lengauer (Ed.), European Symposium on Algorithms ESA, volume 726 of LNCS, Springer, 1993, pp. 73–84.
[17] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Information and Computation 142 (1998) 182–206.
[18] Y.D. Chung, J.W. Kim, M.H. Kim, Efficient preprocessing of XML queries using structured signatures, Information Processing Letters 87 (2003) 257–264.
[19] R. Cox, Regular expression matching can be simple and fast, January 2007. Available from <http://swtch.com/rsc/regexp/regexp1.html>.
[20] CZ-Redaktion, Maschinenmenschen plaudern per XML mit der Unternehmens-IT, Computer Zeitung, Heft 50:30, December 2000.
[21] P. Dupont, Incremental regular inference, in: L. Miclet, C. de la Higuera (Eds.), Proceedings of the Third International Colloquium on Grammatical Inference (ICGI-96): Learning Syntax from Sentences, volume 1147 of LNCS/LNAI, Springer, 1996, pp. 222–237.
[22] P. Dupont, L. Miclet, Inférence grammaticale régulière: fondements théoriques et principaux algorithmes, Technical Report RR-3449, INRIA, 1998.
[23] H. Fernau, Learning XML grammars, in: P. Perner (Ed.), Machine Learning and Data Mining in Pattern Recognition MLDM'01, volume 2123 of LNCS/LNAI, Springer, 2001, pp. 73–87.
[24] H. Fernau, Identification of function distinguishable languages, Theoretical Computer Science 290 (2003) 1679–1711.
[25] H. Fernau, Algorithms for learning regular expressions, in: S. Jain, H.-U. Simon, E. Tomita (Eds.), Algorithmic Learning Theory ALT 2005, volume 3734 of LNCS/LNAI, 2005, pp. 297–311.
[26] H. Fernau, J.M. Sempere, Permutations and control sets for learning non-regular language families, in: A.L. Oliveira (Ed.), Grammatical Inference: Algorithms and Applications, 5th International Colloquium (ICGI 2000), volume 1891 of LNCS/LNAI, Springer, 2000.
[27] M.A. Fulk, Prudence and other conditions on formal language learning, Information and Computation (formerly Information and Control) 85 (1990) 1–11.
[28] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim, XTRACT: learning document type descriptors from XML document collections, Data Mining and Knowledge Discovery 7 (2003) 23–56.
[29] D. Giammmarresi, R. Montalbano, Deterministic generalized automata, Theoretical Computer Science 215 (1999) 191–208.
[30] E.M. Gold, Language identification in the limit, Information and Control (now Information and Computation) 10 (1967) 447–474.
[31] H. Gruber, M. Holzer, Finite automata, digraph connectivity, and regular expression size (extended abstract), in: L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (Eds.), Automata, Languages and Programming ICALP (2), volume 5126 of LNCS, Springer, 2008, pp. 39–50.
[32] C. de la Higuera, Grammatical inference, July 2007. Available from: <http://labh-curien.univ-st-etienne.fr/cdlh/book/>.
[33] O. Hofmann, Regular expressions and other pattern discovery methods. Does it have to be so complicated? Bioinformatics Honours Course, 18th March 2005. Available from: <http://www.sanbi.ac.za/Honours/HonoursCourse05_DeterministicPatterns.pdf>.

[34] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading (MA), 1979.
[35] E. Kinber, Learning a class of regular expressions via restricted subset queries, in: K.P. Jantke (Ed.), Analogical and Inductive Inference AII, volume 642 of LNCS/LNAI, Springer, 1992, pp. 232–243.
[36] E. Kinber, J. Nessel, Learning regular expressions from examples and queries. Personal communication, 2005.
[37] E.B. Kinber, On learning regular expressions and patterns via membership and correction queries, in: A. Clark, F. Coste, L. Miclet (Eds.), Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI, volume 5278 of LNCS, Springer, 2008, pp. 125–138.
[38] M. Korenblit, V.E. Levit, On algebraic expressions of series-parallel and Fibonacci graphs, in: C. Calude, M.J. Dinneen, V. Vajnovszki (Eds.), Discrete Mathematics and Theoretical Computer Science DMTCS, volume 2731 of LNCS, Springer, 2003, pp. 215–224.
[39] Ph. D. Laird, Learning from Good and Bad Data, Kluwer Academic Publishers, Norwell, MA, USA, 1988.
[40] S. Lange, T. Zeugmann, Types of monotonic language learning and their characterization, in: Computational Learning Theory COLT, ACM Press, 1992, pp. 377–390.
[41] S. Lange, T. Zeugmann, Incremental learning from positive data, Journal of Computer and System Sciences 53 (1996) 88–103.
[42] S. Lange, T. Zeugmann, Set-driven and rearrangement-independent learning of recursive languages, Mathematical Systems Theory 29 (1996) 599–634.
[43] R. da Luz, M.F. Ferrari, M.A. Musicante, Regular expression transformations to extend regular languages (with applications to a Datalog XML schema validator), Journal of Algorithms 62 (2007) 148–167.
[44] C.G. Nevill-Manning, I.H. Witten, Online and offline heuristics for inferring hierarchies of repetitions in sequences. Proc. IEEE, 88 (2000) 1745–1755.
[45] G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. III, Springer, Berlin, 1997.
[46] G. Schäfer-Richter, Über Eingabeabhängigkeit und Komplexität von Inferenzstrategien. Rheinisch-Westfälische Technische Hochschule Aachen, Dissertation, 1984.
[47] J.M. Sempere, On a class of regular-like expressions for linear languages, Journal of Automata, Languages and Combinatorics 5 (2000) 343–354.
[48] T.C. Smith, I.H. Witten, J.G. Cleary, S. Legg, Objective evaluation of inferred context-free grammars, in: Proc. 2nd Australian and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia, Institute of Electrical and Electronics Engineers, November 1994, pp. 392–396.
[49] J. Vilo, A. Brazma, I. Jonassen, A.J. Robinson, E. Ukkonen, Mining for putative regulatory elements in the yeast genome using gene expression data, in: P.E. Bourne, M. Gribskov, R.B. Altman, N. Jensen, D. Hope, T. Lengauer, J.C. Mitchell, E.D. Scheeff, C. Smith, S. Strande, H. Weissig (Eds.), Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology ISMB, AAAI Press, 2000, pp. 384–394.
[50] L. Wang, T. Jiang, On the complexity of multiple sequence alignment, Journal of Computational Biology 1 (1994) 337–348.
[51] A. Watt, Beginning Regular Expressions, Wiley, 2005.
[52] R. Wiehagen, From inductive inference to algorithmic learning theory, New Generation Computing 12 (1994) 321–335.
[53] M. van Zaanen, Bootstrapping Structure into Language: Alignment-Based Learning. Ph.D, School of Computing, University of Leeds, UK, September 2001.
[54] T. Zeugmann, Can learning in the limit be done efficiently? in: R. Gavaldà, K.P. Jantke, E. Takimoto (Eds.), Algorithmic Learning Theory ALT, volume 2842 of LNCS, Springer, 2003, pp. 17–38.
[55] T. Zeugmann, S. Lange, A guided tour across the boundaries of learning recursive languages, in: K.P. Jantke, S. Lange (Eds.), Algorithmic Learning for Knowledge-Based Systems, GOSLER Final Report, volume 961 of LNCS, Springer, 1995, pp. 190–258.