# Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data

GEERT JAN BEX, WOUTER GELADE, FRANK NEVEN
Hasselt University and Transnational University of Limburg
and
STIJN VANSUMMEREN
Université Libre de Bruxelles

Inferring an appropriate DTD or XML Schema Definition (XSD) for a given collection of XML documents essentially reduces to learning *deterministic* regular expressions from sets of positive example words. Unfortunately, there is no algorithm capable of learning the complete class of deterministic regular expressions from positive examples only, as we will show. The regular expressions occurring in practical DTDs and XSDs, however, are such that every alphabet symbol occurs only a small number of times. As such, in practice it suffices to learn the subclass of deterministic regular expressions in which each alphabet symbol occurs at most $k$ times, for some small $k$. We refer to such expressions as $k$-occurrence regular expressions ($k$-OREs for short). Motivated by this observation, we provide a probabilistic algorithm that learns $k$-OREs for increasing values of $k$, and selects the deterministic one that best describes the sample based on a Minimum Description Length argument. The effectiveness of the method is empirically validated both on real world and synthetic data. Furthermore, the method is shown to be conservative over the simpler classes of expressions considered in previous work.

Categories and Subject Descriptors: F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages; I.2.6 [**Artificial Intelligence**]: Learning; I.7.2 [**Document and Text Processing**]: Document Preparation

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: regular expressions, schema inference, XML

## 1. INTRODUCTION

Recent studies stipulate that schemas accompanying collections of XML documents are sparse and erroneous in practice. Indeed, Barbosa et al. [2005] and Mignet et al. [2003] have shown that approximately half of the XML documents available on the web do not refer to a schema. In addition, Bex et al. [2004] and Martens et al. [2006] have noted that about two-thirds of XML Schema Definitions (XSDs) gathered from schema repositories and from the web at large are not valid with respect to the W3C XML Schema specification [Thompson et al. 2001], rendering them

A preliminary version of this article appeared in the 17th International World Wide Web Conference (WWW 2008).

```
<!ELEMENT store (order*, stock)>
<!ELEMENT order (customer, item+)>
<!ELEMENT customer (first, last, email*)>
<!ELEMENT item (id, price + (qty, (supplier + item+)))>
<!ELEMENT stock (item*)>
<!ELEMENT supplier (first, last, email*)>
```

Fig. 1. An example DTD.

essentially useless for immedidate application. A similar observation was made by Sahuguet [2000] concerning Document Type Definitions (DTDs). Nevertheless, the presence of a schema strongly facilitates optimization of XML processing (cf., e.g., [Benedikt et al. 2005; Che et al. 2006; Du et al. 2004; Freire et al. 2002; Koch et al. 2004; Manolescu et al. 2001; Neven and Schwentick 2006]) and various software development tools such as Castor [cas ] and SUN's JAXB [jax ] rely on schemas as well to perform object-relational mappings for persistence. Additionally, the existence of schemas is imperative when integrating (meta) data through schema matching [Rahm and Bernstein 2001] and in the area of generic model management [Bernstein 2003].

Based on the above described benefits of schemas and their unavailability in practice, it is essential to devise algorithms that can infer a DTD or XSD for a given collection of XML documents when none, or no syntactically correct one, is present. This is also acknowledged by Florescu [2005] who emphasizes that in the context of data integration

> *"We need to extract good-quality schemas automatically from existing data and perform incremental maintenance of the generated schemas."*

As illustrated in Figure 1, a DTD is essentially a mapping $d$ from element names to regular expressions over element names. An XML document is valid with respect to the DTD if for every occurrence of an element name $e$ in the document, the word formed by its children belongs to the language of the corresponding regular expression $d(e)$. For instance, the DTD in Figure 1 requires each `store` element to have zero or more `order` children, which must be followed by a `stock` element. Likewise, each order must have a `customer` child, which must be followed by one or more `item` elements.

To infer a DTD from a corpus of XML documents $\mathcal{C}$ it hence suffices to look, for each element name $e$ that occurs in a document in $\mathcal{C}$, at the set of element name words that occur below $e$ in $\mathcal{C}$, and to infer from this set the corresponding regular expression $d(e)$. As such, the inference of DTDs reduces to the inference of regular expressions from sets of positive example words. To illustrate, from the words `id price`, `id qty supplier`, and `id qty item item` appearing under `<item>` elements in a sample XML corpus, we could derive the rule

$$\texttt{item} \rightarrow (\texttt{id}, \texttt{price} + (\texttt{qty}, (\texttt{supplier} + \texttt{item}^+))).$$

Although XSDs are more expressive than DTDs, and although XSD inference is therefore more involved than DTD inference, derivation of regular expressions remains one of the main building blocks on which XSD inference algorithms are built.

In fact, apart from also inferring atomic data types, systems like Trang [Clark ] and XStruct [Hegewald et al. 2006] simply infer DTDs in XSD syntax. The more recent *i*XSD algorithm [Bex et al. 2007] does infer true XSD schemas by first deriving a regular expression for every *context* in which an element name appears, where the context is determined by the path from the root to that element, and subsequently reduces the number of contexts by merging similar ones.

So, the effectiveness of DTD or XSD schema inference algorithms is strongly determined by the accuracy of the employed regular expression inference method. The present article presents a method to reliably learn regular expressions that are far more complex than the classes of expressions previously considered in the literature.

### 1.1 Problem setting

In particular, let $\Sigma$ be a fixed set of alphabet symbols (also called element names), and let $\Sigma^*$ be the set of all words over $\Sigma$.

*Definition* 1.1 *(Regular Expressions).* Regular expressions are derived by the following grammar.

$$r, s ::= \emptyset \mid \varepsilon \mid a \mid r \,.\, s \mid r + s \mid r? \mid r^+$$

Here, parentheses may be added to avoid ambiguity; $\varepsilon$ denotes the empty word; $a$ ranges over symbols in $\Sigma$; $r \,.\, s$ denotes concatenation; $r + s$ denotes disjunction; $r^+$ denotes one-or-more repetitions; and $r?$ denotes the optional regular expression. That is, the language $\mathcal{L}(r)$ accepted by regular expression $r$ is given by:

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset & \mathcal{L}(\varepsilon) &= \{\varepsilon\} \\ \mathcal{L}(a) &= \{a\} & \mathcal{L}(r \,.\, s) &= \{vw \mid v \in \mathcal{L}(r), w \in \mathcal{L}(s)\} \\ \mathcal{L}(r + s) &= \mathcal{L}(r) \cup \mathcal{L}(s) & \mathcal{L}(r^+) &= \{v_1 \dots v_n \mid n \geq 1 \text{ and } v_1, \dots, v_n \in \mathcal{L}(r)\} \\ \mathcal{L}(r?) &= \mathcal{L}(r) \cup \{\varepsilon\}. \end{aligned}$$

□

Note that the Kleene star operator (denoting zero or more repititions as in $r^*$) is not allowed by the above syntax. This is not a restriction, since $r^*$ can always be represented as $(r^+)?$ or $(r?)^+$. Conversely, the latter can always be rewritten into the former for presentation to the user.

The class of *all* regular expressions is actually too large for our purposes, as both DTDs and XSDs require the regular expressions occurring in them to be *deterministic* (also sometimes called one-unambiguous [Brüggemann-Klein and Wood 1998]). Intuitively, a regular expression is deterministic if, without looking ahead in the input word, it allows to match each symbol of that word uniquely against a position in the expression when processing the input in one pass from left to right. For instance, $(a + b)^*a$ is not deterministic as already the first symbol in the word $aaa$ could be matched by either the first or the second $a$ in the expression. Without lookahead, it is impossible to know which one to choose. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic.

*Definition* 1.2. Formally, let $\overline{r}$ stand for the regular expression obtained from $r$ by replacing the $i$th occurrence of alphabet symbol $a$ in $r$ by $a^{(i)}$, for every $i$ and $a$. For example, for $r = b^+a(ba^+)?$ we have $\overline{r} = {b^{(1)}}^+ a^{(1)}(b^{(2)}{a^{(2)}}^+)?$. A regular

expression $r$ is *deterministic* if there are no words $wa^{(i)}v$ and $wa^{(j)}v'$ in $\mathcal{L}(\overline{r})$ such that $i \neq j$. □

Equivalently, an expression is deterministic if the Glushkov construction [Brüggeman-Klein 1993] translates it into a deterministic finite automaton rather than a non-deterministic one [Brüggemann-Klein and Wood 1998]. Not every non-deterministic regular expression is equivalent to a deterministic one [Brüggemann-Klein and Wood 1998]. Thus, semantically, the class of deterministic regular expressions forms a strict subclass of the class of all regular expressions.

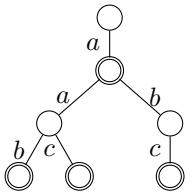
For the purpose of inferring DTDs and XSDs from XML data, we are hence in search of an algorithm that, given enough sample words of a target deterministic regular expression $r$, returns a deterministic expression $r'$ equivalent to $r$. In the framework of *learning in the limit* [Gold 1967], such an algorithm is said to learn the deterministic regular expressions from positive data.

*Definition* 1.3. Define a *sample* to be a finite subset of $\Sigma^*$ and let $\mathcal{R}$ be a subclass of the regular expressions. An algorithm $M$ mapping samples to expressions in $\mathcal{R}$ *learns* $\mathcal{R}$ *in the limit from positive data* if (1) $S \subseteq \mathcal{L}(M(S))$ for every sample $S$ and (2) to every $r \in \mathcal{R}$ we can associate a so-called *characteristic sample* $S_r \subseteq \mathcal{L}(r)$ such that, for each sample $S$ with $S_r \subseteq S \subseteq \mathcal{L}(r)$, $M(S)$ is equivalent to $r$. □

Intuitively, the first condition says that $M$ must be *sound*; the second that $M$ must be *complete*, given enough data. A class of regular expressions $\mathcal{R}$ is *learnable in the limit from positive data* if an algorithm exists that learns $\mathcal{R}$. For the class of all regular expressions, it was shown by Gold that no such algorithm exists [Gold 1967]. We extend this result to the class of deterministic expressions:

Theorem 1.4. *The class of deterministic regular expressions is not learnable in the limit from positive data.*

Proof. It was shown by Gold [1967, Theorem I.8], that any class of regular expressions that contains all non-empty finite languages as well as at least one infinite language is not learnable in the limit from positive data. Since deterministic regular expressions like $a^*$ define an infinite language, it suffices to show that every non-empty finite language is definable by a deterministic expression. Hereto, let $S$ be a finite, non-empty set of words. Now consider the prefix tree $T$ for $S$. For example, if $S = \{a, aab, abc, aac\}$, we have the following prefix tree:




Nodes for which the path from the root to that node forms a word in $S$ are marked by double circles. In particular, all leaf nodes are marked.

By viewing the internal nodes in $T$ with two or more children as disjunctions; internal nodes in $T$ with one child as conjunctions; and adding a question mark for every marked internal node in $T$, it is straightforward to transform $T$ into a regular

expression. For example, with $S$ and $T$ as above we get $r = a\,.(b\,.\,c + a\,.(b + c))?$. Clearly, $\mathcal{L}(r) = S$. Moreover, since no node in $T$ has two edges with the same label, $r$ must be deterministic. □

Theorem 1.4 immediately excludes the possibility for an algorithm to infer the full class of DTDs or XSDs. In practice, however, regular expressions occurring in DTDs and XSDs are *concise* rather than arbitrarily complex. Indeed, a study of 819 DTDs and XSDs gathered from the Cover Pages [Cover 2003] (including many high-quality XML standards) as well as from the web at large, reveals that regular expressions occurring in practical schemas are such that every alphabet symbol occurs only a small number of times [Martens et al. 2006]. In practice, therefore, it suffices to learn the subclass of deterministic regular expressions in which each alphabet symbol occurs at most $k$ times, for some small $k$. We refer to such expressions as *$k$-occurrence regular expressions*.

*Definition* 1.5. A regular expression is *$k$-occurrence* if every alphabet symbol occurs at most $k$ times in it. □

For example, the expressions $\texttt{customer}\,.\,\texttt{order}^+$ and $(\texttt{school} + \texttt{institute})^+$ are both 1-occurrence, while $\texttt{id}\,.(\texttt{qty}{+}\texttt{id})$ is 2-occurrence (as id occurs twice). Observe that if $r$ is $k$-occurrence, then it is also $l$-occurrence for every $l \geq k$. To simplify notation in what follows, we abbreviate '$k$-occurrence regular expression' by $k$-ORE and also refer to the 1-OREs as 'single occurrence regular expressions' or SOREs.

## 1.2 Outline and Contributions

Actually, the above mentioned examination shows that in the majority of the cases $k = 1$. Motivated by that observation, we have studied and suggested practical learning algorithms for the class of deterministic SOREs in a companion article [Bex et al. 2006]. These algorithms, however, can only output SOREs even when the target regular expression is not. In that case they always return an approximation of the target expressions. It is therefore desirable to also have learning algorithms for the class of deterministic $k$-OREs with $k \geq 2$. Furthermore, since the exact $k$-value for the target expression, although small, is unknown in a schema inference setting, we also require an algorithm capable of determining the best value of $k$ automatically.

We begin our study of this problem in Section 3 by showing that, for each fixed $k$, the class of deterministic $k$-OREs *is* learnable in the limit from positive examples only. We also argue, however, that this theoretical algorithm is unlikely to work well in practice as it does not provide a method to automatically determine the best value of $k$ and needs samples whose size can be exponential in the size of the alphabet to successfully learn some target expressions.

In view of these observations, we provide in Section 4 the practical algorithm $i$DRegEx. Given a sample of words $S$, $i$DRegEx derives corresponding deterministic $k$-OREs for increasing values of $k$ and selects from these candidate expressions the expression that describes $S$ best. To determine the "best" expression we propose two measures: (1) a Language Size measure and (2) a Minimum Description Length measure based on the work of Adriaans and Vitányi [2006]. The main technical contribution lies in the subroutine used to derive the actual $k$-OREs for $S$.

Indeed, while for the special case where $k = 1$ one can derive a $k$-ORE by first learning an automaton $A$ for $S$ using the inference algorithm of Garcia and Vidal [1990], and by subsequently translating $A$ into a 1-ORE (as shown in [Bex et al. 2006]), this approach does not work when $k \geq 2$. In particular, the algorithm of Garcia and Vidal only works when learning languages that are "$n$-testable" for some fixed natural number $n$ [Garcia and Vidal 1990]. Although every language definable by a 1-ORE is 2-testable [Bex et al. 2006], there are languages definable by a 2-ORE, for instance $a^*ba^*$, that are not $n$-testable for any $n$. We therefore use a probabilistic method based on Hidden Markov Models to learn an automaton for $S$, which is subsequently translated into a $k$-ORE.

The effectiveness of $i$DREGEX is empirically validated in Section 5 both on real world and synthetic data. We compare the results of $i$DREGEX with those of the algorithm presented in previous work [Bex et al. 2008], to which we refer as $i$DREGEX(RWR$^0$).

## 2. RELATED WORK

**Semi-structured data.** In the context of semi-structured data, the inference of schemas as defined in [Buneman et al. 1997; Quass et al. 1996] has been extensively studied [Goldman and Widom 1997; Nestorov et al. 1998]. No methods were provided to translate the inferred types to regular expressions, however.

**DTD and XSD inference.** In the context of DTD inference, Bex et al. [2006] gave in earlier work two inference algorithms: one for learning 1-OREs and one for learning the subclass of 1-OREs known as *chain regular expressions*. The latter class can also be learned using Trang [Clark ], state of the art software written by James Clark that is primarily intended as a translator between the schema languages DTD, Relax NG [Clark and Murata 2001], and XSD, but also infers a schema for a set of XML documents. In contrast, our goal in this article is to infer the more general class of deterministic expressions. XTRACT [Garofalakis et al. 2003] is another regular expression learning system with similar goals. We note that XTRACT also uses the Minimum Description Length principle to choose the best expression from a set of candidates.

Other relevant DTD inference research is [Sankey and Wong 2001] and [Chidlovskii 2001] that learn finite automata but do not consider the translation to deterministic regular expressions. Also, in [Young-Lai and Tompa 2000] a method is proposed to infer DTDs through stochastic grammars where right-hand sides of rules are represented by probabilistic automata. No method is provided to transform these into regular expressions. Although Ahonen [1996] proposes such a translation, the effectiveness of her algorithm is only illustrated by a single case study of a dictionary example; no experimental study is provided.

Also relevant are the XSD inference systems [Bex et al. 2007; Clark ; Hegewald et al. 2006] that, as already mentioned, rely on the same methods for learning regular expressions as DTD inference.

**Regular expression inference.** Most of the learning of regular languages from positive examples in the computational learning community is directed towards inference of automata as opposed to inference of regular expressions [Angluin and

Smith 1983; Pitt 1989; Sakakibara 1997]. However, these approaches learn strict subclasses of the regular languages which are incomparable to the subclasses considered here. Some approaches to inference of regular expressions for restricted cases have been considered. For instance, [Brāzma 1993] showed that regular expressions without union can be approximately learned in polynomial time from a set of examples satisfying some criteria. [Fernau 2005] provided a learning algorithm for regular expressions that are finite unions of pairwise left-aligned union-free regular expressions. The development is purely theoretical, no experimental validation has been performed.

**HMM learning.** Although there has been work on Hidden Markov Model structure induction [Rabiner 1989; Freitag and McCallum 2000], the requirement in our setting that the resulting automaton is deterministic is, to the best of our knowledge, unique.

## 3. BASIC RESULTS

In this section we establish that, in contrast to the class of all deterministic expressions, the subclass of deterministic $k$-OREs *can* theoretically be learned in the limit from positive data, for each fixed $k$. We also argue, however, that this theoretical algorithm is unlikely to work well in practice.

Let $\Sigma(r)$ denote the set of alphabet symbols that occur in a regular expression $r$, and let $\Sigma(S)$ be similarly defined for a sample $S$. Define the *length* of a regular expression $r$ as the length of it string representation, including operators and parenthesis. For example, the length of $(a\,.\,b)^+? + c$ is 9.

THEOREM 3.1. *For every $k$ there exists an algorithm $M$ that learns the class of deterministic $k$-OREs from positive data. Furthermore, on input $S$, $M$ runs in time polynomial in the size of $S$, yet exponential in $k$ and $|\Sigma(S)|$.*

PROOF. The algorithm $M$ is based on the following observations. First observe that every deterministic $k$-ORE $r$ over a finite alphabet $A \subseteq \Sigma$ can be simplified into an equivalent deterministic $k$-ORE $r'$ of length at most $10k|A|$ by rewriting $r$ according to the following system of rewrite rules until no more rule is applicable:

$$\begin{array}{rcl@{\qquad}rcl}
((s)) & \to & (s) & s?^+ & \to & s^+? \\
s?? & \to & s? & s^{++} & \to & s^+ \\
s + \varepsilon & \to & s? & \varepsilon + s & \to & s? \\
s\,.\,\varepsilon & \to & s & \varepsilon\,.\,s & \to & s \\
\varepsilon? & \to & \varepsilon & \varepsilon^+ & \to & \varepsilon \\
s + \emptyset & \to & s & \emptyset + s & \to & s \\
s\,.\,\emptyset & \to & \emptyset & \emptyset\,.\,s & \to & \emptyset \\
\emptyset? & \to & \emptyset & \emptyset^+ & \to & \emptyset
\end{array}$$

(The first rewrite rule removes redundant parenthesis in $r$.) Indeed, since each rewrite rule clearly preserves determinism and language equivalence, $r'$ must be a deterministic expression equivalent to $r$. Moreover, since none of the rewrite rules duplicates a subexpression and since $r$ is a $k$-ORE, so is $r'$. Now note that, since

no rewrite rule applies to it, $r'$ is either $\emptyset$, $\varepsilon$, or generated by the following grammar

$$\begin{aligned} t \ ::= \ & a \mid a? \mid a^+ \mid a^+? \mid (a) \mid (a)? \mid (a)^+ \mid (a)^+? \\ \mid \ & t_1 \,.\, t_2 \mid (t_1 \,.\, t_2) \mid (t_1 \,.\, t_2)? \mid (t_1 \,.\, t_2)^+ \mid (t_1 \,.\, t_2)^+? \\ \mid \ & t_1 + t_2 \mid (t_1 + t_2) \mid (t_1 + t_2)? \mid (t_1 + t_2)^+ \mid (t_1 + t_2)^+? \end{aligned}$$

It is not difficult to verify by structural induction that any expression $t$ produced by this grammar has length

$$|t| \leq -4 + 10 \sum_{a \in \Sigma(t)} \mathit{rep}(t, a),$$

where $\mathit{rep}(t, a)$ denotes the number of times alphabet symbol $a$ occurs in $t$. For instance, $\mathit{rep}(b\,.(b + c), a) = 0$ and $\mathit{rep}(b\,.(b + c), b) = 2$. Since $\mathit{rep}(r', a) \leq k$ for every $a \in \Sigma(r')$, it readily follows that $|r'| \leq 10k|A| - 4 \leq 10k|A|$.

Then observe that all possible regular expressions over $A$ of length at most $10k|A|$ can be enumerated in time exponential in $k|A|$. Since checking whether a regular expression is deterministic is decidable in polynomial time [Brüggemann-Klein and Wood 1998]; and since equivalence of deterministic expressions is decidable in polynomial time [Brüggemann-Klein and Wood 1998], it follows by the above observations that for each $k$ and each finite alphabet $A \subseteq \Sigma$ it is possible to compute in time exponential in $k|A|$ a finite set $\mathcal{R}_A$ of pairwise non-equivalent deterministic $k$-OREs over $A$ such that

—every $r \in \mathcal{R}_A$ is of size at most $10k|A|$; and

—for every deterministic $k$-ORE $r$ over $A$ there exists an equivalent expression $r' \in \mathcal{R}_A$.

(Note that since $\mathcal{R}_A$ is computable in time exponential in $k|A|$, it has at most an exponential number of elements in $k|A|$.) Now fix, for each finite $A \subseteq \Sigma$ an arbitrary order $\prec$ on $\mathcal{R}_A$, subject to the provision that $r \prec s$ only if $\mathcal{L}(s) - \mathcal{L}(r) \neq \emptyset$. Such an order always exists since $\mathcal{R}_A$ does not contain equivalent expressions.

Then let $M$ be the algorithm that, upon sample $S$, computes $\mathcal{R}_{\Sigma(S)}$ and outputs the first (according to $\prec$) expression $r \in \mathcal{R}_{\Sigma(S)}$ for which $S \subseteq L(r)$. Since $\mathcal{R}_{\Sigma(S)}$ can be computed in time exponential in $k|\Sigma(S)|$; since there are at most an exponential number of expressions in $\mathcal{R}_{\Sigma(S)}$; since each expression $r \in \mathcal{R}_{\Sigma(S)}$ has size at most $10k|\Sigma(S)|$; and since checking membership in $\mathcal{L}(r)$ of a single word $w \in S$ can be done in time polynomial in the size of $w$ and $r$, it follows that $M$ runs in time polynomial in $S$ and exponential in $k|\Sigma(S)|$.

Furthermore, we claim that $M$ learns the class of deterministic $k$-OREs. Clearly, $S \subseteq \mathcal{L}(M(S))$ by definition. Hence, it remains to show completeness, i.e., that we can associate to each deterministic $k$-ORE $r$ a sample $S_r \subseteq L(r)$ such that, for each sample $S$ with $S_r \subseteq S \subseteq L(r)$, $M(S)$ is equivalent to $r$. Note that, by definition of $\mathcal{R}_{\Sigma(r)}$, there exists a deterministic $k$-ORE $r' \in \mathcal{R}_{\Sigma(r)}$ equivalent to $r$. Initialize $S_r$ to an arbitrary finite subset of $\mathcal{L}(r) = \mathcal{L}(r')$ such that each alphabet symbol of $r$ occurs at least once in $S$, i.e., $\Sigma(S_r) = \Sigma(r)$. Let $r_1 \prec \cdots \prec r_n$ be all predecessors of $r'$ in $\mathcal{R}_{\Sigma(r)}$ according to $\prec$. By definition of $\prec$, there exists a word $w_i \in \mathcal{L}(r) - \mathcal{L}(r_i)$ for every $1 \leq i \leq n$. Add all of these words to $S_r$. Then clearly, for every sample $S$ with $S_r \subseteq S \subseteq \mathcal{L}(r)$ we have $\Sigma(S) = \Sigma(r)$ and $S \not\subseteq L(r_i)$ for every $1 \leq i \leq n$. Since

$M(S)$ is the first expression in $\mathcal{R}_{\Sigma(r)}$ with $S \subseteq L(r)$, we hence have $M(S) = r' \equiv r$, as desired. □

While Theorem 3.1 shows that the class of deterministic $k$-OREs is better suited for learning from positive data than the complete class of deterministic expressions, it does not provide a useful practical algorithm, for the following reasons.

(1) First and foremost, $M$ runs in time exponential in the size of the alphabet $\Sigma(S)$, which may be problematic for the inference of schema's with many element names.

(2) Second, while Theorem 3.1 shows that the class of deterministic $k$-OREs is learnable in the limit for each fixed $k$, the schema inference setting is such that we do not know $k$ a priori. If we overestimate $k$ then $M(S)$ risks being an under-approximation of the target expression $r$, especially when $S$ is incomplete. To illustrate, consider the 1-ORE target expression $r = a^+b^+$ and sample $S = \{ab, abbb, aabb\}$. If we overestimate $k$ to, say, 2 instead of 1, then $M$ is free to output $aa?b^+$ as a sound answer. On the other hand, if we underestimate $k$ then $M(S)$ risks being an over-approximation of $r$. Consider, for instance, the 2-ORE target expression $r = aa?b^+$ and the same sample $S = \{ab, abbb, aabb\}$. If we underestimate $k$ to be 1 instead of 2, then $M$ can only output 1-OREs, and needs to output at least $a^+b^+$ in order to be sound. In summary: we need a method to determine the most suitable value of $k$.

(3) Third, the notion of learning in the limit is a very liberal one: correct expressions need only be derived when sufficient data is provided, i.e., when the input sample is a superset of the characteristic sample for the target expression $r$. The following theorem shows that there are reasonably simple expressions $r$ such that characteristic sample $S_r$ of any sound and complete learning algorithm is at least exponential in the size of $r$. As such, it is unlikely for any sound and complete learning algorithm to behave well on real-world samples, which are typically incomplete and hence unlikely to contain all words of the characteristic sample.

THEOREM 3.2. *Let $A = \{a_1, \ldots, a_n\} \subseteq \Sigma$ consist of $n$ distinct element names. Let $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$, and let $r_2 = (a_2 + \cdots + a_n)^+a_1(a_2 + \cdots + a_n)^+$. For any algorithm that learns the class of deterministic $(2n + 3)$-OREs and any sample $S$ that is characteristic for $r_1$ or $r_2$ we have $|S| \geq \sum_{i=1}^{n}(n-2)^i$.*

PROOF. First consider $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$. Observe that there exist an exponential number of deterministic $(2n + 3)$-OREs that differ from $r_1$ in only a single word. Indeed, let $B = A - \{a_1, a_2\}$ and let $W$ consist of all non-empty words $w$ over $B$ of length at most $n$. Define, for every word $w = b_1 \ldots b_m \in W$ the deterministic $(2n + 3)$-ORE $r_w$ such that $\mathcal{L}(r_w) = \mathcal{L}(r_1) - \{w\}$ as follows. First, define, for every $1 \leq i \leq m$ the deterministic 2-ORE $r_w^i$ that accepts all words in $\mathcal{L}(r_1)$ that do not start with $b_i$:

$$r_w^i := (a_1a_2 + (B - \{b_i\})) .(a_1a_2 + a_3 + \cdots + a_n)^*$$

Clearly, $v \in \mathcal{L}(r_1) - \{w\}$ if, and only if, $v \in \mathcal{L}(r_1)$ and there is some $0 \leq i \leq m$ such that $v$ agrees with $w$ on the first $i$ letters, but differs in the $(i + 1)$-th letter.

Hence, it suffices to take

$$r_w := r_w^1 + b_1(\varepsilon + r_w^2 + b_2(\varepsilon + r_w^3 + b_3(\cdots + b_{m-1}(\varepsilon + r_w^m + b_m \,.\, r_1)\ldots)))$$

Now assume that algorithm $M$ learns the class of deterministic $(2n+3)$-OREs and suppose that $S_{r_1}$ is characteristic for $r_1$. In particular, $S_{r_1} \subseteq \mathcal{L}(r_1)$. By definition, $M(S)$ is equivalent to $r$ for every sample $S$ with $S_{r_1} \subseteq S \subseteq \mathcal{L}(r_1)$. We claim that in order for $M$ to have this property, $W$ must be a subset of $S_r$. Then, since $W$ contains all words over $B$ of length at most $n$, $|S_{r_1}| \geq \sum_{i=1}^{n}(n-2)^i$, as desired. The intuitive argument why $W$ must be a subset of $S_r$ is that if there exists $w$ in $W - S_r$, then $M$ cannot distinguish between $r_1$ and $r_w$. Indeed, suppose for the purpose of contradiction that there is some $w \in W$ with $w \notin S_{r_1}$. Then $S_{r_1}$ is a subset of $\mathcal{L}(r_w)$. Indeed, $S_{r_1} = S_{r_1} - \{w\} \subseteq \mathcal{L}(r_1) - \{w\} = \mathcal{L}(r_w)$. Furthermore, since $M$ learns the class of deterministic $(2n+3)$-OREs, there must be some characteristic sample $S_{r_w}$ for $r_w$. Now, consider the sample $S_{r_1} \cup S_{r_w}$. It is included in both $\mathcal{L}(r_1)$ and $\mathcal{L}(r_w)$ and is a superset of both $S_{r_1}$ and $S_{r_w}$. But then, by definition of characteristic samples, $M(S_{r_1} \cup S_{r_w})$ must be equivalent to both $r_1$ and $r_w$. This is absurd, however, since $\mathcal{L}(r_1) \neq \mathcal{L}(r_w)$ by construction.

A similar argument shows that the characteristic sample $S_{r_2}$ of $r_2 = (a_2 + \cdots + a_n)^+ a_1 (a_2 + \cdots + a_n)^+$ also requires $\sum_{i=1}^{n}(n-2)^i$ elements. In this case, we take $B = A - \{a_1\}$ and we take $W$ to be the set of all non-empty words over $B$ of length at most $n$. For each $w = b_1 \ldots b_m \in W$, we construct the deterministic $(2n+3)$-ORE $r_w$ such that $\mathcal{L}(r_w)$ accepts all words in $\mathcal{L}(r)$ that do not end with $a_1 w$, as follows. Let, for $1 \leq i \leq m$, $r_w^i$ be the 2-ORE that accepts all words in $B^+$ that do not start with $b_i$:

$$r_w^i := (B - \{b_i\}) \,.\, B^*$$

Then it suffices to take

$$r_w := B^+ a_1 (r_w^i + b_1(\varepsilon + r_w^2 + b_3(\cdots + b_{m-1}(\varepsilon + r_w^m + b_m B^+)\ldots))).$$

A similar argument as for $r_1$ then shows that the characteristic sample $S_{r_2}$ of $r_2$ needs to contain, for each $w \in W$, at least one word of the form $v a_1 w$ with $v \in B^+$. Therefore, $|S_{r_2}| \geq \sum_{i=1}^{n}(n-2)^i$, as desired. □

## 4. THE LEARNING ALGORITHM

In view of the observations made in Section 3, we present in this section a practical learning algorithm that (1) works well on incomplete data and (2) automatically determines the best value of $k$ (see Section 5 for an experimental evaluation). Specifically, given a sample $S$, the algorithm derives deterministic $k$-OREs for increasing values of $k$ and selects from these candidate expressions the $k$-ORE that describes $S$ best. To determine the "best" expression we propose two measures: (1) a Language Size measure and (2) a Minimum Description Length measure based on the work of Adriaans and Vitányi [2006].

Our algorithm does not derive deterministic $k$-OREs for $S$ directly, but uses, for each fixed $k$, a probabilistic method to first learn an automaton for $S$, which is subsequently translated into a $k$-ORE. The following section (Section 4.1) explains how the probabilistic method that learns an automaton from $S$ works. Section 4.2 explains how the learned automaton is translated into a $k$-ORE. Finally, Section 4.3,

introduces the whole algorithm, together with the two measures to determine the best candidate expression.

### 4.1 Probabilistically Learning a Deterministic Automaton

In particular, the algorithm first learns a *deterministic k-occurrence automaton* (deterministic $k$-OA) for $S$. This is a specific kind of finite state automaton in which each alphabet symbol can occur at most $k$ times. Figure 2(a) gives an example. Note that in contrast to the classical definition of an automaton, no edges are labeled: all incoming edges in a state $s$ are assumed to be labeled by the label of $s$. In other words, the 2-OA of Figure 2(a) accepts the same language as $aa?b^+$.

*Definition* 4.1 *($k$-OA).* An *automaton* is a node-labeled graph $G = (V, E, lab)$ where

—$V$ is a finite set of nodes (also called *states*) with a distinguished source $src \in V$ and sink $sink \in V$;
—the edge relation $E$ is such that $src$ has only outgoing edges; $sink$ has only incoming edges; and every state $v \in V - \{src, sink\}$ is reachable by a walk from $src$ to $sink$;
—$lab\colon V - \{src, sink\} \to \Sigma$ is the labeling function.

In this context, an *accepting run* for a word $a_1 \dots a_n$ is a walk $src\, s_1 \dots s_n\, sink$ from $src$ to $sink$ in $G$ such that $a_i = lab(s_i)$ for $1 \leq i \leq n$. As usual, we denote by $\mathcal{L}(G)$ the set of all words for which an accepting run exists. An automaton is *$k$-occurrence* (a $k$-OA) if there are at most $k$ states labeled by the same alphabet symbol. If $G$ uses only labels in $A \subseteq \Sigma$ then $G$ is *an automaton over $A$*. □

In what follows, we write $\mathrm{Succ}(s)$ for the set $\{t \mid (s,t) \in E\}$ of all direct successors of state $s$ in $G$, and $\mathrm{Pred}(s)$ for the set $\{t \mid (t,s) \in E\}$ of all direct predecessors of $s$ in $G$. Furthermore, we write $\mathrm{Succ}(s,a)$ and $\mathrm{Pred}(s,a)$ for the set of states in $\mathrm{Succ}(s)$ and $\mathrm{Pred}(s)$, respectively, that are labeled by $a$. As usual, an automaton $G$ is *deterministic* if $\mathrm{Succ}(s,a)$ contains at most one state, for every $s \in V$ and $a \in \Sigma$.

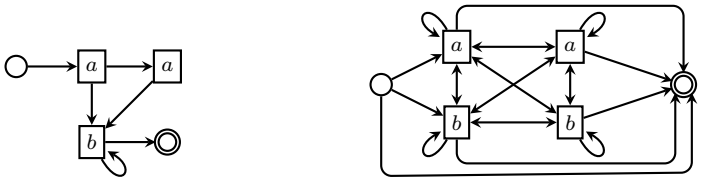
For convenience, we will also refer to the 1-OAs as "single occurence automata" or SOAs for short.

We learn a deterministic $k$-OA for a sample $S$ as follows. First, recall from Section 3 that $\Sigma(S)$ is the set of alphabet symbols occurring in words in $S$. We view $S$ as the result of a stochastic process that generates words from $\Sigma^*$ by performing random walks on the *complete* $k$-OA $C_k$ *over* $\Sigma(S)$.

*Definition* 4.2. Define the *complete* $k$-OA $C_k$ over $\Sigma(S)$ to be the $k$-OA $G = (V, E, lab)$ over $\Sigma(S)$ in which each $a \in \Sigma(S)$ labels exactly $k$ states such that

—there is an edge from $src$ to $sink$;
—$src$ is connected to exactly one state labeled by $a$, for every $a \in \Sigma(S)$; and
—every state $s \in V - \{src, sink\}$ has an outgoing edge to every other state except $src$. □

To illustrate, the complete 2-OA over $\{a, b\}$ is shown in Figure 2(b). Clearly, $\mathcal{L}(C_k) = \Sigma(S)^*$.




(a) An example 2-OA. It accepts the same language as $aa?b^+$

(b) The complete 2-OA over $\{a, b\}$.

Fig. 2. Two 2-OAs.

The stochastic process that generates words from $\Sigma^*$ by performing random walks on $C_k$ operates as follows. First, the process picks, among all states in $\mathrm{Succ}(src)$, a state $s_1$ with probability $\alpha(src, s_1)$ and emits $lab(s_1)$. Then it picks, among all states in $\mathrm{Succ}(s_1)$ a state $s_2$ with probability $\alpha(s_1, s_2)$ and emits $lab(s_2)$. The process continues moving to new states and emitting their labels until the final state is reached (which does not emit a symbol). Of course, $\alpha$ must be a true probability distribution, i.e.,

$$\alpha(s,t) \geq 0; \quad \text{and} \quad \sum_{t \in \mathrm{Succ}(s)} \alpha(s,t) = 1 \tag{1}$$

for all states $s \neq sink$ and all states $t$. The probability of generating a particular accepting run $\vec{s} = src\, s_1 s_2 \ldots s_n\, sink$ given the process $\mathcal{P} = (C_k, \alpha)$ in this setting is

$$P[\vec{s} \mid \mathcal{P}] = \alpha(src, s_1) \cdot \alpha(s_2, s_3) \cdot \alpha(s_2, s_3) \cdots \alpha(s_n, sink),$$

and the probability of generating the word $w = a_1 \ldots a_n$ is

$$P[w \mid \mathcal{P}] = \sum_{\text{all accepting runs } \vec{s} \text{ of } w \text{ in } C_k} P[\vec{s} \mid \mathcal{P}].$$

Assuming independence, the probability of obtaining all words in the sample $S$ is then

$$P[S \mid \mathcal{P}] = \prod_{w \in S} P[w \mid \mathcal{P}].$$

Clearly, the process that best explains the observation of $S$ is the one in which the probabilities $\alpha$ are such that they maximize $P[S \mid \mathcal{P}]$.

To learn a deterministic $k$-OA for $S$ we therefore first try to infer from $S$ the probability distribution $\alpha$ that maximizes $P[S \mid \mathcal{P}]$, and use this distribution to determine the topology of the desired deterministic $k$-OA. In particular, we remove from $C_k$ the non-deterministic edges with the lowest probability as these are the least likely to contribute to the generation of $S$, and are therefore the least likely to be necessary for the acceptance of $S$.

The problem of inferring $\alpha$ from $S$ is well-studied in Machine Learning, where our stochastic process $\mathcal{P}$ corresponds to a particular kind of Hidden Markov Model sometimes referred to as a Partially Observable Markov Model (POMM for short). (For the readers familiar with Hidden Markov Models we note that the initial state distribution $\pi$ usually considered in Hidden Markov Models is absorbed in

**Algorithm 1** *i*KOA

**Require:** a sample $S$, a value for $k$
**Ensure:** a deterministic $k$-OA $G$ with $S \subseteq \mathcal{L}(G)$
1: $\mathcal{P} \leftarrow \text{init}(k, S)$
2: $\mathcal{P} \leftarrow \text{BAUMWELSH}(\mathcal{P}, S)$
3: $G \leftarrow \text{DISAMBIGUATE}(\mathcal{P}, S)$
4: $G \leftarrow \text{PRUNE}(G, S)$
5: **return** $G$

**Algorithm 2** DISAMBIGUATE

**Require:** a POMM $\mathcal{P} = (G, \alpha)$ and sample $S$
**Ensure:** a deterministic $k$-OA
1: Initialize queue $Q$ to $\{s \in \text{Succ}(src) \mid \alpha(src, s) > 0\}$
2: Initialize set of marked states $D \leftarrow \emptyset$
3: **while** $Q$ is non-empty **do**
4: $s \leftarrow \text{first}(Q)$
5: **while** some $a \in \Sigma$ has $|\text{Succ}(s, a)| > 1$ **do**
6: pick $t \in \text{Succ}(s, a)$ with $\alpha(s, t) = \max\{\alpha(s, t') \mid t' \in \text{Succ}(s, a)\}$
7: set $\alpha(s, t) \leftarrow \sum\{\alpha(s, t') \mid t' \in \text{Succ}(s, a)\}$
8: **for** all $t'$ in $\text{Succ}(s, a) \setminus \{t\}$ **do**
9: delete edge $(s, t')$ from $G$
10: set $\alpha(s, t') \leftarrow 0$
11: $\mathcal{P} \leftarrow \text{BAUMWELSH}(\mathcal{P}, S)$
12: **if** $S \not\subseteq \mathcal{L}(G)$ **then Fail**
13: add $s$ to marked states $D$ and pop $s$ from $Q$
14: enqueue all states in $\text{Succ}(s) \setminus D$ to $Q$
15: **return** $G$

the state transition distribution $\alpha(src, \cdot)$ in our context.) Inference of $\alpha$ is generally accomplished by the well-known Baum-Welsh algorithm [Rabiner 1989] that adjusts initial values for $\alpha$ until a (possibly local) maximum is reached.

We use Baum-Welsh in our learning algorithm *i*KOA shown in Algorithm 1, which operates as follows. In line 1, *i*KOA initializes the stochastic process $\mathcal{P}$ to the tuple $(C_k, \alpha)$ where

—$C_k$ is the complete $k$-OA over $\Sigma(S)$;
—$\alpha(src, sink)$ is the fraction of empty words in $S$;
—$\alpha(src, s)$ is the fraction of words in $S$ that start with $lab(s)$, for every $s \in \text{Succ}(src)$; and
—$\alpha(s, t)$ is chosen randomly for $s \neq src$, subject to the constraints in equation (1).

It is important to emphasize that, since we are trying to model a stochastic process, multiple occurrences of the same word in $S$ *are* important. A sample should therefore not be considered as a set in Algorithm 1, but as a *bag*. Line 2 then optimizes the initial values of $\alpha$ using the Baum-Welsh algorithm.

With these probabilities in hand DISAMBIGUATE, shown in Algorithm 2, determines the topology of the desired deterministic $k$-OA for $S$. In a breadth-first

manner, it picks for each state $s$ and each symbol $a$ the state $t \in \text{Succ}(s, a)$ with the highest probability and deletes all other edges to states labeled by $a$. Line 7 merely ensures that $\alpha$ continues to be a probability distribution after this removal and line 11 adjusts $\alpha$ to the new topology. Line 12 is a sanity check that ensures that we have not removed edges necessary to accept all words in $S$; DISAMBIGUATE reports failure otherwise. The result of a successful run of DISAMBIGUATE is a deterministic $k$-OA which nevertheless may have edges $(s, t)$ for which there is no *witness* in $S$ (i.e., a word in $S$ whose unique accepting run traverses $(s, t)$). The function PRUNE in line 4 of $i$KOA removes all such edges. It also removes all states $s \in \text{Succ}(src)$ without a witness in $S$. Figure 3 illustrates a hypothetical run of $i$KOA.

It should be noted that BAUMWELSH, which iteratively refines $\alpha$ until a (possibly local) maximum is reached, is computationally quite expensive. For that reason, our implementation only executes a fixed number of refinement iterations of BAUMWELSH in Line 11. Rather surprisingly, this cut-off actually improves the precision of $i$DREGEX, as our experiments in Section 5 show, where it is discussed in more detail.

### 4.2 Translating $k$-OAs into $k$-OREs

Once we have learned a deterministic $k$-OA for a given sample $S$ using $i$KOA it remains to translate this $k$-OA into a deterministic $k$-ORE. An obvious approach in this respect would be to use the classical state elimination algorithm (cf., e.g., [Hopcroft and Ullman 2007]). Unfortunately, as already hinted upon by Fernau [2004; 2005] and as we illustrate below, it is very difficult to get *concise* regular expressions from an automaton representation. For instance, the classical state elimination algorithm applied to the SOA in Figure 4 yields the expression:[1]

$$(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))^*(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d)))(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))^*$$

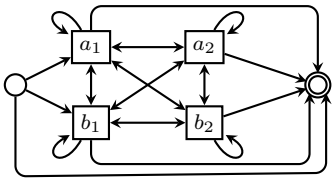
which is non-deterministic and differs quite a bit from the equivalent deterministic SORE

$$((b?(a + c))^+d)^+e.$$

Actually, results by Ehrenfeucht and Zeiger [1976]; Gelade and Neven [2008]; and Gruber and Holzer [2008] show that it is impossible in general to generate concise regular expressions from automata: there are $k$-OAs (even for $k = 1$) for which the number of occurrences of alphabet symbols in the smallest equivalent expression is exponential in the size of the automaton. For such automata, an equivalent $k$-ORE hence does not exist.

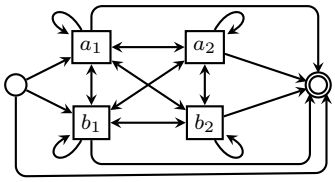
It is then natural to ask whether there is an algorithm that translates a given $k$-OA into an equivalent $k$-ORE when such a $k$-ORE exists, and returns a $k$-ORE super approximation of the input $k$-OA otherwise. Clearly, the above example shows that the classical state elimination algorithm does not suffice for this purpose.

[1] Transformation computed by JFLAP: `www.jflap.org`.



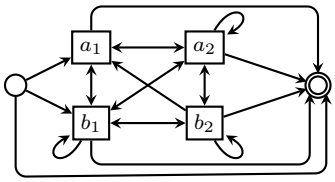

| $\alpha$ | $a_1$ | $a_2$ | $b_1$ | $b_2$ | $sink$ |
|---|---|---|---|---|---|
| $src$ | 1 | \ | 0 | \ | 0 |
| $a_1$ | 0.2 | 0.3 | 0.3 | 0.1 | 0.1 |
| $a_2$ | 0.4 | 0.1 | 0.2 | 0.1 | 0.2 |
| $b_1$ | 0.1 | 0.3 | 0.3 | 0.2 | 0.1 |
| $b_2$ | 0.1 | 0.1 | 0.2 | 0.5 | 0.1 |

(a) Process $\mathcal{P}$ returned by init with random values for $\alpha$.



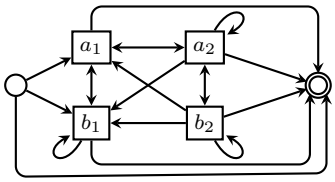

| $\alpha$ | $a_1$ | $a_2$ | $b_1$ | $b_2$ | $sink$ |
|---|---|---|---|---|---|
| $src$ | 1 | \ | 0 | \ | 0 |
| $a_1$ | 0.2 | 0.3 | 0.3 | 0.19 | 0.01 |
| $a_2$ | 0.01 | 0.01 | 0.6 | 0.37 | 0.01 |
| $b_1$ | 0.01 | 0.01 | 0.5 | 0.28 | 0.2 |
| $b_2$ | 0.01 | 0.01 | 0.33 | 0.5 | 0.15 |

(b) Process $\mathcal{P}$ after first training by BaumWelsh.



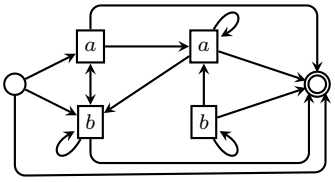

| $\alpha$ | $a_1$ | $a_2$ | $b_1$ | $b_2$ | $sink$ |
|---|---|---|---|---|---|
| $src$ | 1 | \ | 0 | \ | 0 |
| $a_1$ | 0 | 0.5 | 0.49 | 0 | 0.01 |
| $a_2$ | 0.01 | 0.01 | 0.6 | 0.37 | 0.01 |
| $b_1$ | 0.01 | 0.01 | 0.5 | 0.28 | 0.2 |
| $b_2$ | 0.01 | 0.01 | 0.33 | 0.5 | 0.15 |

(c) Process $\mathcal{P}$ after first disambiguation step (for $a_1$). Edges to $a_1$ and $b_2$ are removed.

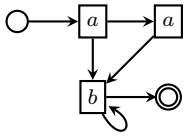



| $\alpha$ | $a_1$ | $a_2$ | $b_1$ | $b_2$ | $sink$ |
|---|---|---|---|---|---|
| $src$ | 1 | \ | 0 | \ | 0 |
| $a_1$ | 0 | 0.5 | 0.49 | 0 | 0.01 |
| $a_2$ | 0.01 | 0.01 | 0.6 | 0.37 | 0.01 |
| $b_1$ | 0.02 | 0 | 0.78 | 0 | 0.2 |
| $b_2$ | 0.01 | 0.01 | 0.38 | 0.4 | 0.2 |

(d) Process $\mathcal{P}$ after second disambiguation step (for $b_1$). Edges to $a_2$ and $b_2$ are removed.




(e) Automaton $A$ returned by Disambiguate.




(f) Automaton $A$ returned by Prune. It accepts the same language as $aa?b^+$.

Fig. 3. Example run of $i$Koa for $k = 2$ with target language $aa?b^+$. For the process $\mathcal{P}$ in (c)-(f), the $\alpha$ values are listed in table-form. To distinguish different states with the same label, we have indexed the labels.

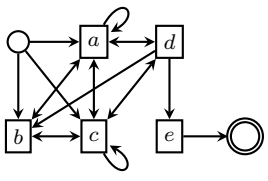



Fig. 4. A SOA on which the classical state elimination algorithm returns a complicated expression.

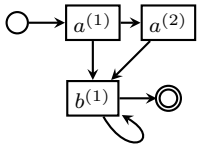



Fig. 5. An example marking

For that reason, we have proposed in a companion article [Bex et al. ] a family of algorithms $\{\text{RWR}, \text{RWR}_1^2, \text{RWR}_2^2, \text{RWR}_3^2, \dots\}$ that translate SOAs into SOREs and have exactly these properties:

THEOREM 4.3 ([BEX ET AL. ]). *Let $G$ be a SOA and let $T$ be any of the algorithms in the family $\{\text{RWR}, \text{RWR}_1^2, \text{RWR}_2^2, \text{RWR}_3^2, \dots\}$. If $G$ is equivalent to a SORE $r$, then $T(G)$ returns a SORE equivalent to $r$. Otherwise, $T(G)$ returns a SORE that is a super approximation of $G$, $\mathcal{L}(G) \subseteq \mathcal{L}(T(G))$.*

(Note that SOAs and SOREs are always deterministic by definition.)

These algorithms, in short, apply an inverse Glushkov translation. Starting from a $k$-OA where each state is labeled by a symbol, they iteratively rewrite subautomata into equivalent regular expressions. In the end only one state remains and the regular expression labeling this state is the output.

In this section, we show how the above algorithms can be used to translate $k$-OAs into $k$-OREs. For simplicity of exposition, we will focus our discussion on $\text{RWR}_1^2$ as it is the concrete translation algorithm used in our experiments in Section 5, but the same arguments apply to the other algorithms in the family.

*Definition* 4.4. First, let $\Sigma^{(k)}$ denote the alphabet that consists of $k$ copies of the symbols in $\Sigma$, where the first copy of $a \in \Sigma$ is denoted by $a^{(1)}$, the second by $a^{(2)}$, and so on:

$$\Sigma^{(k)} := \{a^{(i)} \mid a \in \Sigma, 1 \leq i \leq k\}.$$

Let *strip* be the function mapping copies to their original symbol, i.e., $\mathit{strip}(a^{(i)}) = a$. We extend *strip* pointwise to words, languages, and regular expressions over $\Sigma^{(k)}$. □

For example, $\mathit{strip}(\{a^{(1)}a^{(2)}b^{(1)}, a^{(2)}a^{(2)}c^{(2)}\}) = \{aab, aac\}$ and $\mathit{strip}(a^{(1)} \,.\, a^{(2)}? \,.\, {b^{(1)}}^{+}) = a \,.\, a? \,.\, b^{+}$.

To see how we can use $\text{RWR}_1^2$, which translates SOAs into SOREs, to translate a $k$-OA into a $k$-ORE, observe that we can always transform a $k$-OA $G$ over $\Sigma$ into a SOA $H$ over $\Sigma^{(k)}$ by processing the nodes of $G$ in an arbitrary order and replacing the $i$th occurrence of label $a \in \Sigma$ by $a^{(i)}$. To illustrate, the SOA over $\Sigma^{(2)}$ obtained in this way from the 2-OA in Figure 2(a) is shown in Figure 5. Clearly, $\mathcal{L}(G) = \mathit{strip}(\mathcal{L}(H))$.

*Definition* 4.5. We call a SOA $H$ over $\Sigma^{(k)}$ obtained from a $k$-OA $G$ in the above manner a *marking* of $G$. □

Note that, by Theorem 4.3, running $\text{RWR}_1^2$ on $H$ yields a SORE $r$ over $\Sigma^{(k)}$ with $\mathcal{L}(H) \subseteq \mathcal{L}(r)$. For instance, with $H$ as in Figure 5, $\text{RWR}^2(H)$ returns $r =$

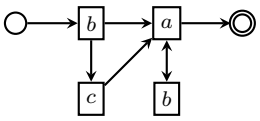
**Algorithm 3** RWR$^2$

**Require:** a $k$-OA $G$
**Ensure:** a $k$-ORE $r$ with $\mathcal{L}(G) \subseteq \mathcal{L}(r)$
1: compute a marking $H$ of $G$.
2: **return** $\mathit{strip}(\text{RWR}_1^2(H))$

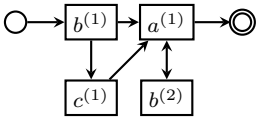
$a^{(1)} \,.\, a^{(2)}? \,.\, b^{(1)+}$. By subsequently stripping $r$, we always obtain a $k$-ORE over $\Sigma$. Moreover, $\mathcal{L}(G) = \mathit{strip}(\mathcal{L}(H)) \subseteq \mathit{strip}(\mathcal{L}(r)) = \mathcal{L}(\mathit{strip}(r))$, so the $k$-ORE $\mathit{strip}(r)$ is always a super approximation of $G$. Algorithm 3, called RWR$^2$ , summarizes the translation. By our discussion, RWR$^2$ is clearly sound:

PROPOSITION 4.6. RWR$^2(G)$ *is a (possibly non-deterministic)* $k$-ORE *with* $\mathcal{L}(G) \subseteq \mathcal{L}(\text{RWR}^2(G))$*, for every* $k$-OA $G$*.*

Note, however, that even when $G$ is deterministic and equivalent to a deterministic $k$-ORE $r$, RWR$^2(G)$ need not be deterministic, nor equivalent to $r$. For instance, consider the 2-OA $G$:




Clearly, $G$ is equivalent to the deterministic 2-ORE $bc?a(ba)^+?$. Now suppose for the purpose of illustration that RWR$^2$ constructs the following marking $H$ of $G$. (It does not matter which marking RWR$^2$ constructs, they all result in the same final expression.)




Since $H$ is not equivalent to a SORE over $\Sigma^{(k)}$, RWR$_1^2(H)$ need not be equivalent to $\mathcal{L}(H)$. In fact, RWR$_1^2(H)$ returns $((b^{(1)}c^{(1)}?a^{(1)})?b^{(2)}?)^+$, which yields the non-deterministic $((bc?a)?b?)^+$ after stripping. Nevertheless, $G$ is equivalent to the deterministic 2-ORE $bc?a(ba)^+?$.

So although RWR$^2$ is always guaranteed to return a $k$-ORE, it does not provide the same strong guarantees that RWR$_1^2$ provides (Theorem 4.3). The following theorem shows, however, that if we can obtain $G$ by applying the Glushkov construction on $r$ [Brüggeman-Klein 1993], RWR$^2(G)$ is always equivalent to $r$. Moreover, if $r$ is deterministic, then so is RWR$^2(G)$. So in this sense, RWR$^2$ applies an inverse Glushkov construction to $r$. Formally, the Glushkov construction is defined as follows.

*Definition* 4.7. Let $r$ be a $k$-ORE. Recall from Definition 1.2 that $\overline{r}$ is the regular expression obtained from $r$ by replacing the $i$th occurrence of alphabet symbol $a$ by $a^{(i)}$, for every $a \in \Sigma$ and every $1 \leq i \leq n$. Let $\mathit{pos}(\overline{r})$ denote the symbols in $\Sigma^{(k)}$ that actually appear in $\overline{r}$. Moreover, let the sets $\mathit{first}(\overline{r})$, $\mathit{last}(\overline{r})$, and $\mathit{follow}(\overline{r}, a^{(i)})$ be defined as shown in Figure 6. A $k$-OA $G$ is a *Glushkov translation* of $r$ if there exists a one-to-one onto mapping $\rho\colon (V(G) - \{\mathit{src}, \mathit{sink}\}) \to \mathit{pos}(\overline{r})$ such that

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{first}(\emptyset) & = & \emptyset & \mathit{first}(\varepsilon) & = & \emptyset \\
\mathit{first}(a^{(i)}) & = & \{a^{(i)}\} & \mathit{first}(\overline{r}?) & = & \mathit{first}(\overline{r}) \\
\mathit{first}(\overline{r}^{+}) & = & \mathit{first}(\overline{r}) & \mathit{first}(\overline{r}+\overline{s}) & = & \mathit{first}(\overline{r}) \cup \mathit{first}(\overline{s}) \\
\mathit{first}(\overline{r}\,.\,\overline{s}) & = & \begin{cases} \mathit{first}(\overline{r}) & \text{if } \varepsilon \notin \mathcal{L}(\overline{r}), \\ \mathit{first}(\overline{r}) \cup \mathit{first}(\overline{s}) & \text{otherwise.} \end{cases} & & &
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{last}(\emptyset) & = & \emptyset & \mathit{last}(\varepsilon) & = & \emptyset \\
\mathit{last}(a^{(i)}) & = & \{a^{(i)}\} & \mathit{last}(\overline{r}?) & = & \mathit{last}(\overline{r}) \\
\mathit{last}(\overline{r}^{+}) & = & \mathit{last}(\overline{r}) & \mathit{last}(\overline{r}+\overline{s}) & = & \mathit{last}(\overline{r}) \cup \mathit{last}(\overline{s}) \\
\mathit{last}(\overline{r}\,.\,\overline{s}) & = & \begin{cases} \mathit{last}(\overline{s}) & \text{if } \varepsilon \notin \mathcal{L}(\overline{s}), \\ \mathit{last}(\overline{r}) \cup \mathit{last}(\overline{s}) & \text{otherwise.} \end{cases} & & &
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{follow}(a^{(i)}, a^{(i)}) & = & \emptyset \\
\mathit{follow}(\overline{r}?, a^{(i)}) & = & \mathit{follow}(\overline{r}, a^{(i)}) \\
\mathit{follow}(\overline{r}^{+}, a^{(i)}) & = & \begin{cases} \mathit{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \notin \mathit{last}(\overline{r}), \\ \mathit{follow}(\overline{r}, a^{(i)}) \cup \mathit{first}(\overline{r}) & \text{otherwise.} \end{cases} \\
\mathit{follow}(\overline{r}+\overline{s}, a^{(i)}) & = & \begin{cases} \mathit{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}), \\ \mathit{follow}(\overline{s}, a^{(i)}) & \text{otherwise.} \end{cases} \\
\mathit{follow}(\overline{r}\,.\,\overline{s}, a^{(i)}) & = & \begin{cases} \mathit{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}), a^{(i)} \notin \mathit{last}(\overline{r}), \\ \mathit{follow}(\overline{r}, a^{(i)}) \cup \mathit{first}(\overline{s}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}), a^{(i)} \in \mathit{last}(\overline{r}), \\ \mathit{follow}(\overline{s}, a^{(i)}) & \text{otherwise.} \end{cases}
\end{array}
$$

Fig. 6. Definition of $\mathit{first}(\overline{r})$, $\mathit{last}(\overline{r})$, and $\mathit{follow}(\overline{r}, a^{(i)})$, for $a^{(i)} \in \mathit{pos}(\overline{r})$.

(1) $v \in \mathrm{Succ}(\mathit{src}) \Leftrightarrow \rho(v) \in \mathit{first}(\overline{r})$;

(2) $v \in \mathrm{Pred}(\mathit{sink}) \Leftrightarrow \rho(v) \in \mathit{last}(\overline{r})$;

(3) $v \in \mathrm{Succ}(w) \Leftrightarrow \rho(v) \in \mathit{follow}(\overline{r}, \rho(w))$; and

(4) $\mathit{strip}(\rho(v)) = \mathit{lab}(v)$,

for all $v, w \in V(G) - \{\mathit{src}, \mathit{sink}\}$. □

THEOREM 4.8. *If $k$-OA $G$ is a Glushkov representation of a target $k$-ORE $r$, then* $\textsc{rwr}^2(G)$ *is equivalent to $r$. Moreover, if $r$ is deterministic, then so is* $\textsc{rwr}^2(G)$.

PROOF. Since $\textsc{rwr}^2(G) = \mathit{strip}(\textsc{rwr}_1^2(H))$ for an arbitrarily chosen marking $H$ of $G$, it suffices to prove that $\mathit{strip}(\textsc{rwr}_1^2(H))$ is equivalent to $r$ and that $\mathit{strip}(\textsc{rwr}_1^2(H))$ is deterministic whenever $r$ is deterministic, for every marking $H$ of $G$. Hereto, let $H$ be an arbitrary but fixed marking of $G$. In particular, $G$ and $H$ have the same set of nodes $V$ and edges $E$, but differ in their labeling function. Let $\mathit{lab}_G$ be the labeling function of $G$ and let $\mathit{lab}_H$ the labeling function of $H$. Clearly, $\mathit{lab}_G(v) = \mathit{strip}(\mathit{lab}_H(v))$ for every $v \in V - \{\mathit{src}, \mathit{sink}\}$. Since $G$ is a Glushkov translation of $r$, there is a one-to-one, onto mapping $\rho\colon (V - \{\mathit{src}, \mathit{sink}\}) \to \mathit{pos}(\overline{r})$ satisfying properties (1)-(4) in Definition 4.7. Now let $\sigma\colon \mathit{pos}(\overline{r}) \to \Sigma^{(k)}$ be the function that maps $a^{(i)} \in \mathit{pos}(\overline{r})$ to $\mathit{lab}_H(\rho^{-1}(a^{(i)}))$. Since $\mathit{lab}_H$ assigns a distinct label to each state, $\sigma$ is one-to-one and onto the subset of $\Sigma^{(k)}$ symbols used as labels in $H$. Moreover, by property (4) and the fact that $\mathit{lab}_G(v) = \mathit{strip}(\mathit{lab}_H(v))$

we have,

$$strip(a^{(i)}) = lab_G(\rho^{-1}(a^{(i)})) = strip(lab_H(\rho^{-1}(a^{(i)}))) = strip(\sigma(a^{(i)})) \qquad (\star)$$

for each $a^{(i)} \in pos(\overline{r})$. In other words, $\sigma$ preserves (stripped) labels. Now let $\sigma(\overline{r})$ be the SORE obtained from $\overline{r}$ by replacing each $a^{(i)} \in pos(\overline{r})$ by $\sigma(a^{(i)})$. Since $\sigma$ is one-to-one and $\overline{r}$ is a SORE, so is $\sigma(\overline{r})$. Moreover, we claim that $\mathcal{L}(H) = \mathcal{L}(\sigma(\overline{r}))$.

Indeed, it is readily verified by induction on $\overline{r}$ that a word ${a_1}^{(i_1)} \dots {a_n}^{(i_n)} \in \mathcal{L}(\overline{r})$ if, and only if, (i) ${a_1}^{(i_1)} \in first(\overline{r})$; (ii) ${a_{p+1}}^{(i_{p+1})} \in follow(\overline{r}, {a_{p+1}}^{(i_{p+1})})$ for every $1 \leq p < n$; and (iii) ${a_n}^{(i_n)} \in last(\overline{r})$. By properties (1)-(4) of Definition 4.7 we hence obtain:

$$\begin{aligned}
& \sigma({a_1}^{(i_1)}) \dots \sigma({a_n}^{(i_n)}) \in \mathcal{L}(\sigma(\overline{r})) \\
\Leftrightarrow\ & {a_1}^{(i_1)} \dots {a_n}^{(i_n)} \in \mathcal{L}(\overline{r}) \\
\Leftrightarrow\ & src, \rho^{-1}({a_1}^{(i_1)}), \dots, \rho^{-1}({a_n}^{(i_n)}), sink \text{ is a walk in } G \\
\Leftrightarrow\ & src, \rho^{-1}({a_1}^{(i_1)}), \dots, \rho^{-1}({a_n}^{(i_n)}), sink \text{ is a walk in } H \\
\Leftrightarrow\ & lab_H(\rho^{-1}({a_1}^{(i_1)})) \dots, lab_H(\rho^{-1}({a_n}^{(i_n)})) \in \mathcal{L}(H) \\
\Leftrightarrow\ & \sigma({a_1}^{(i_1)}) \dots \sigma({a_n}^{(i_n)}) \in \mathcal{L}(H)
\end{aligned}$$

Therefore, $\mathcal{L}(H) = \mathcal{L}(\sigma(\overline{r}))$.

Hence, we have established that $H$ is a SOA over $\Sigma^{(k)}$ equivalent to the SORE $\sigma(\overline{r})$ over $\Sigma^{(k)}$. By Theorem 4.3, $\text{RWR}_1^2(H)$ is hence equivalent to $\sigma(\overline{r})$. Therefore, $strip(\text{RWR}_1^2(H))$ is equivalent to $strip(\sigma(\overline{r}))$, which by $(\star)$ above, is equivalent to $strip(\overline{r}) = r$, as desired.

Finally, to see that $strip(\text{RWR}_1^2(H))$ is deterministic if $r$ is deterministic, let $s := strip(\text{RWR}_1^2(H))$ and suppose for the purpose of contradiction that $s$ is not deterministic. Then there exists $wa^{(i)}v_1$ and $wa^{(j)}v_2$ in $\mathcal{L}(\overline{s})$ with $i \neq j$. It is not hard to see that this can happen only if there exist $w'a^{(i')}v'_1$ and $w'a^{(j')}v'_2$ in $\mathcal{L}(\text{RWR}_1^2(H))$ with $i' \neq j'$. Since $\mathcal{L}(\text{RWR}_1^2(H)) = \mathcal{L}(\sigma(\overline{r}))$ we know that hence $\sigma^{-1}(w'a^{(i')}v'_1) \in \mathcal{L}(\overline{r})$ and $\sigma^{-1}(w'a^{(j')}v'_2) \in \mathcal{L}(\overline{r})$. Let $w''a^{(i'')}v''_1 = \sigma^{-1}(w'a^{(i')}v'_1)$ and $w''a^{(j'')}v''_2 = \sigma^{-1}(w'a^{(i')}v'_2)$. Since $\sigma$ is one-to-one and $i' \neq j'$, also $i'' \neq j''$. Therefore, $r$ is not deterministic, which yields the desired contradiction. □

### 4.3 The whole Algorithm

Our deterministic regular expression inference algorithm $i$DRegEx combines $i$Koa and $\text{RWR}^2$ as shown in Algorithm 4. For increasing values of $k$ until a maximum $k_{\max}$ is reached, it first learns a deterministic $k$-OA $G$ from the given sample $S$, and subsequently translates that $k$-OA into a $k$-ORE using $\text{RWR}^2$. If the resulting $k$-ORE is deterministic then it is added to the set $C$ of deterministic candidate expressions for $S$, otherwise it is discarded. From this set of candidate expressions, $i$DRegEx returns the "best" regular expression best($C$), which is determined according to one of the measures introduced below. Since it is well-known that, depending on the initial value of $\alpha$, BaumWelsh (and therefore $i$Koa) may converge to a local maximum that is not necessarily global, we apply $i$Koa a number of times $N$ with independently chosen random seed values for $\alpha$ to increase the probability of correctly learning the target regular expression from $S$.

The observant reader may wonder whether we are always guaranteed to derive at least one deterministic expression such that best($C$) is defined. Indeed, Theorem 4.8 tells us that if we manage to learn from sample $S$ a $k$-OA which is the

**Algorithm 4** *i*DRegEx

**Require:** a sample $S$
**Ensure:** a $k$-ORE $r$
1: initialize candidate set $C \leftarrow \emptyset$
2: **for** $k = 1$ to $k_{\max}$ **do**
3:  **for** $n = 1$ to $N$ **do**
4:  $G \leftarrow i\text{Koa}(S, k)$
5:  **if** $\text{rwr}^2(G)$ is deterministic **then**
6:  add $\text{rwr}^2(G)$ to $C$
7: **return** best($C$)

Glushkov representation of the target expression $r$, then $\text{rwr}^2$ will always return a deterministic $k$-ORE equivalent to $r$. When $k > 1$, there can be several $k$-OAs representing the same language and we could therefore learn a non-Glushkov one. In that case, $\text{rwr}^2$ always returns a $k$-ORE which is a super approximation of the target expression. Although that approximation can be non-deterministic, since we derive $k$-OREs for increasing values of $k$ and since for $k = 1$ the result of $\text{rwr}^2$ is always deterministic (as every SORE is deterministic), we always infer at least one deterministic regular expression. In fact, in our experiments on 100 synthetic regular expressions, we derived for 96 of them a deterministic expression with $k > 1$, and only for 4 expressions had to resort to a 1-ORE approximation.

4.3.1 *A Language Size Measure for Determining the Best Candidate.* Intuitively, we want to select from $C$ the simplest deterministic expression that "best" describes $S$. Since each candidate expression in $C$ accepts all words in $S$ by construction, one way to interpret "the best" is to select the expression that accepts the least number of words (thereby adding the least number of words to $S$). Since an expression defines an infinite language in general, it is of course impossible to take all words into account. We therefore only consider the words up to a length $n$, where $n = 2m + 1$ with $m$ the length of the candidate expression, excluding regular expression operators, $\emptyset$, and $\varepsilon$. For instance, if the candidate expression is $a\,.(a + c^+)?$, then $m = 3$ and $n = 7$. Formally, for a language $L$, let $|L^{\leq n}|$ denote the number of words in $L$ of length at most $n$. Then the best candidate in $C$ is the one with the least value of $|\,\mathcal{L}(r)^{\leq n}|$. If there are multiple such candidates, we pick the shortest one (breaking ties arbitrarily). It turns out that $|\,\mathcal{L}(r)^{\leq n}|$ can be computed quite efficiently; see [Bex et al. ] for details.

4.3.2 *A Minimum Description Length Measure for Determining the Best Candidate.* An alternative measure to determine the best candidate is given by Adriaans and Vitányi [2006], who compare the size of $S$ with the size of the language of a candidate $r$. Specifically, Adriaans and Vitányi define the data encoding cost of $r$ to be:

$$\text{datacost}(r, S) := \sum_{i=0}^{n} \left( 2 \cdot \log_2 i + \log_2 \binom{|\,\mathcal{L}^{=i}(r)|}{|S^{=i}|} \right),$$

where $n = 2m + 1$ as before; $|S^{=i}|$ is the number of words in $S$ that have length $i$; and $|\,\mathcal{L}^{=i}(r)|$ is the number of words in $\mathcal{L}(r)$ that have exactly length $i$. Although

the above formula is numerically difficult to compute, there is an easier estimation procedure; see [Adriaans and Vitányi 2006] for details.

In this case, the model encoding cost is simply taken to be its length, thereby preferring shorter expressions over longer ones. The best regular expression in the candidate set $C$ is then the one that minimizes both model and data encoding cost (breaking ties arbitrarily).

We already mentioned that XTRACT [Garofalakis et al. 2003] also utilizes the Minimum Description Length principle. However, their measure for data encoding cost depends on the concrete structure of the regular expressions while ours only depends on the language defined by them and is independent of the representation. Therefore, in our setting, when two equivalent expressions are derived, the one with the smallest model cost, that is, the simplest one, will always be taken.

## 5. EXPERIMENTS

In this section we validate our approach by means of an experimental analysis. Throughout the section, we say that a target $k$-ORE $r$ *is successfully derived* when a $k$-ORE $s$ with $\mathcal{L}(r) = \mathcal{L}(s)$ is generated. The *success rate* of our experiments then is the percentage of successfully derived target regular expressions.

Our previous work [Bex et al. 2008] on this topic was based on a version of the RWR$^0$ algorithm [Bex et al. 2006], we refer to this algorithm as $i$DREGEX(RWR$^0$). Unfortunately, as detailed in [Bex et al. 2008], it is not known whether RWR$^0$ is complete on the class of all single occurrence regular expressions. Nevertheless, the experiments in [Bex et al. 2008] which are revisited below show a good and reliable performance. However, to obtain a theoretically complete algorithm, c.f.r. Theorem 4.8, we use the algorithm RWR$^2$ which is sound and complete on single occurrence regular expressions. In the remainder we focus on $i$DREGEX, but compare with the results for $i$DREGEX(RWR$^0$).

As mentioned in Section 4.3.1, another new aspect of the results presented here is the use of language size as an alternative measure over Minimum Description Length (MDL) to compare candidates. The $i$DREGEX(RWR$^0$) algorithm is only considered with the MDL criterion. We note that for alphabet size 5, the success rate of $i$DREGEX with the MDL criterion was only 21 %, while that of the language size criterion is 98 %. The corpus used in this experiment is described in Section 5.3. Therefore in the remainder of this section we only consider $i$DREGEX with the language size criterion.

For all the experiments described below we take $k_{\max} = 4$ and $N = 10$ in Algorithm 4.

### 5.1 Running times

All experiments were performed using a prototype implementation of $i$DREGEX and $i$DREGEX(RWR$^0$) written in Java executed on Pentium M 2.0 GHz class machines equipped with 1GB RAM. For the BAUMWELSH subroutine we have gratefully used Jean-Marc François' *Jahmm* library [François 2006], which is a faithful implementation of the algorithms described in Rabiner's Hidden Markov Model tutorial [Rabiner 1989]. Since Jahmm strives for clarity rather than performance and since only limited precautions are taken against underflows, our prototype should be seen as a proof of concept rather than a polished product. In particular, under-

flows currently limit us to target regular expressions whose total number of symbol occurrences is at most 40. Here, the total number of symbol occurrences $occ(r)$ of a regular expression $r$ is its length excluding the regular expression operators and parenthesis. To illustrate, the total number of symbol occurrences in $aa?b^+$ is 3. Furthermore, the lack of optimization in Jahmm leads to average running times ranging from 4 minutes for target expressions $r$ with $|\Sigma(r)| = 5$ and $occ(r) = 6$ to 9 hours for targets expression with $|\Sigma(r)| = 15$ and $occ(r) = 30$. Running times for $i$DREGEX and $i$DREGEX(RWR$^0$) are similar.

As already mentioned in Section 4.3, one of the bottlenecks of $i$DREGEX is the application of BAUMWELSH in Line 11 of DISAMBIGUATE (Algorithm 2). BAUMWELSH is an iterative procedure that is typically run until convergence, i.e., until the computed probability distribution no longer change significantly. To improve the running time, we only apply a fixed number $\ell$ of iteration steps when calling BAUMWELSH in Line 11 of DISAMBIGUATE. Experiments show that the running time performance scales linear with $\ell$ as one expects, but, perhaps surprisingly, the success rate improves as well for an optimal value of $\ell$. This optimal value for $\ell$ depends on the alphabet size. These improved results can be explained as follows: applying BAUMWELSH in each disambiguation step until it converges guarantees that the probability distribution for that step will have reached a local optimum. However, we know that the search space for the algorithm contains many local optima, and that BAUMWELSH is a local optimization algorithm, i.e., it will converge to one of the local optima it can reach from its starting point by hill climbing. The disambiguation procedure proceeds state by state, so fine tuning the probability distribution for a disambiguation step may transform the search space so that certain local optima for the next iteration can no longer be reached by a local search algorithm such as BAUMWELSH. Table I shows the performance of the algorithm for various number of BAUMWELSH iterations $\ell$ for expressions of alphabet size 5, 10 and 15. These expressions are those described in Section 5.3. In this Table, $\ell = \infty$ denotes the case where BAUMWELSH is ran until convergence after each disambiguation step. The Table illustrates that the success rate is actually higher for small values of $\ell$. The running time performance gains increase rapidly with the expressions' alphabet size: for $|\Sigma| = 5$, we gain a factor of 3.5 ($\ell = 2$), for $|\Sigma| = 10$, it is already a factor of 10 ($\ell = 3$) and for $|\Sigma| = 15$, we gain a factor of 25 ($\ell = 3$). This brings the running time for the largest expressions we tested down to 22 minutes, in contrast with 9 hours mentioned for $i$DREGEX(RWR$^0$) and $i$DREGEX. The algorithm with the optimal number of BAUMWELSH steps in the disambiguation process will be referred to as $i$DREGEX$^{\text{fixed}}$. In particular for small alphabet sizes ($|\Sigma| \leq 7$) we use $\ell = 2$, for large alphabet size $\ell = 3$ ($|\Sigma| > 7$). We note that the alphabet size can easily be determined from the sample.

We should also note that Experience with Hidden Markov Model learning in bioinformatics [Finn et al. 2006] suggests that both the running time and the maximum number of symbol occurrences that can be handled can be significantly improved by moving to an industrial-strength BAUMWELSH implementation. Our focus for the rest of the section will therefore be on the precision of $i$DREGEX.

| $\ell$ | rate $\lvert\Sigma\rvert = 5$ | rate $\lvert\Sigma\rvert = 10$ | rate $\lvert\Sigma\rvert = 15$ |
|---|---|---|---|
| 1 | 95 % | 80 % | 40 % |
| 2 | **100 %** | 75 % | 50 % |
| 3 | 95 % | **84 %** | **60 %** |
| 4 | 95 % | 77 % | 50 % |
| $\infty$ | 98 % | 75 % | 50 % |

Table I. Success rate for a limited number of BaumWelsh iterations in the disambiguation procedure, $\ell = \infty$ corresponds to $i$DRegEx, for $\ell = 1, \ldots, 4$ correspond to $i$DRegEx$^{\text{fixed}}$.

### 5.2 Real-world target expressions and real-world samples

We want to test how $i$DRegEx performs on real-world data. Since the number of publicly available XML corpora with valid schemas is rather limited, we have used as target expressions the 49 content models occurring in the XSD for XML Schema Definitions [Thompson et al. 2001] and have drawn multiset samples for these expressions from a large corpus of real-world XSDs harvested from the Cover Pages [Cover 2003]. In other words, the goal of our first experiment is to derive, from a corpus of XSD definitions, the regular expression content models in the schema for XML Schema Definitions[2]. As it turns out, the XSD regular expressions are all single occurrence regular expressions.

The $i$DRegEx(rwr$^0$) algorithm infers all these expressions correctly, showing that it is conservative with respect to $k$ since, as mentioned above, the algorithm considers $k$ values ranging from 1 to 4. In this setting, $i$DRegEx performs not as well, deriving only 73 % of the regular expressions correctly. We note that for each expression that was not derived exactly, always an expression was obtained describing the input sample and which in addition is more specific than the target expression. $i$DRegEx therefore seems to favor more specific regular expressions, based on the available examples.

### 5.3 Synthetic target expressions

Although the successful inference of the real-world expressions in Section 5.2 suggests that $i$DRegEx is applicable in real-world scenarios, we further test its behavior on a sizable and diverse set of regular expressions. Due to the lack of real-world data, we have developed a synthetic regular expression generator that is parameterized for flexibility.

**Synthetic expression generation.** In particular, the occurrence of the regular expression operators concatenation, disjunction (+), zero-or-one (?), zero-or-more ($^*$), and one-or-more ($^+$) in the generated expressions is determined by a user-defined probability distribution. We found that typical values yielding realistic expressions are 1/10 for the unary operators and 7/20 for others. The alphabet can be specified, as well as the number of times that each individual symbol should occur. The maximum of these numbers determines the value $k$ of the generated $k$-ORE.

To ensure the validity of our experiments, we want to generate a wide range of different expressions. To this end, we measure how much the language of a generated

[2]This corpus was also used in [Bex et al. 2007] for XSD inference.

$((debab) + c)^* a$
$((((c + b)b) + a)ca) + e + d$
$(((ea)^* db) + b + a + c)^+$
$((b^+ + c + e + d)aab)^+$
$((((eabh) + d + j + c + b)^+ f) + a + g + i)?$
$((((aa) + e)^+ + c)b) + b + d$
$((((d + a)^* eabcb) + c)a)?$
$((((ac) + b + d)eab) + c)^*$
$(((((bab) + c)^+ + e)?a) + d)^+$
$((((ecb)^+ a) + b)^+ + d + a)?$
$((bagbfeid) + c + a + j + h)^*$
$((gdab) + a + i + c + j + e + f)^+ hb$
$((h^* cdfa) + j + e + g + b + i)^* ab$
$((g + b + e + f + i + d)^* aba) + h + j + c$
$((((h + b + c + j + f)^+ + e)?aaidb) + g)?$
$((((((dbe)^* cf) + j)hac) + b + i)^* gad$
$((((((ihaaj) + d)^+ + g)b) + e + b + f + c)^+$
$(((ecgecd) + b + d + a + j + f)^* ihaba)^*$
$(l + c + d + m + n)^* aojahbegcbfidke$
$(((c + b)ab) + d + i + a)^+ + j + g + f + e + h$
$(((a?clfhabgd) + b + n + o)iedjcem)^* k$
$((a + k + f + c + m + e)^+ bdieclbonjgda)^* h$
$(((k?jghadfcelifcjbhom) +$
$b + g + a + e + i + n)^+ + d)?$
$(((aedoadenhdbci) + h + k + m + j + g + b)^*$
$fccgelbifja)$
$((a^+ + f + d + o + g + n + h + c + b + j + i + e)$
$keacdlbm)$
$(((k + f + o + a + j)?edhldfhngicjmab)?cie)^* bg$
$((((a?d)^+ ba) + h + g + e + c)^+ + j + i + b)?f$

Fig. 7. A snapshot of the 100 generated expressions.

expression overlaps with $\Sigma^*$. The larger the overlap, the greater its language size as defined in Section 4.3.1.

To ensure that the generated expressions do not impede readability by containing redundant subexpressions (as in e.g., $(a^+)^+$), the final step of our generator is to syntactically simplify the generated expressions using the following straightforward equivalences:

$$\begin{aligned}
r^* &\rightarrow r^+? \\
r?? &\rightarrow r? \\
(r^+)^+ &\rightarrow r^+ \\
(r?)^+ &\rightarrow r^+? \\
(r_1 \cdot r_2) \cdot r_3 &\rightarrow r_1 \cdot (r_2 \cdot r_3) \\
r_1 \cdot (r_2 \cdot r_3) &\rightarrow r_1 \cdot r_2 \cdot r_3 \\
(r_1? \cdot r_2?)? &\rightarrow r_1? \cdot r_2? \\
(r_1 + r_2) + r_3 &\rightarrow r_1 + (r_2 + r_3) \\
r_1 + (r_2 + r_3) &\rightarrow r_1 + r_2 + r_3 \\
(r_1 + r_2^+)^+ &\rightarrow (r_1 + r_2)^+ \\
(r_1^+ + r_2^+) &\rightarrow (r_1 + r_2)^+ \\
r_1 + r_2? &\rightarrow (r_1 + r_2)?
\end{aligned}$$

Of course, the resulting expression is rejected if it is non-deterministic.

To obtain a diverse target set, we synthesized expressions with alphabet size 5 (45 expressions), 10 (45 expressions), and 15 (10 expressions) with a variety of symbol occurrences ($k = 1, 2, 3$). For each of the alphabet sizes, the expressions were selected to cover language size ranging from 0 to 1. All in all, this yielded a set of 100 deterministic target expressions. A snapshot is given in Figure 7.

**Synthetic sample generation.** For each of those 100 target expressions, we generated synthetic samples by transforming the target expressions into stochastic processes that perform random walks on the automata representing the expressions (cf. Section 4). The probability distributions of these processes are derived from the structure of the originating expression. In particular, each operand in a disjunction

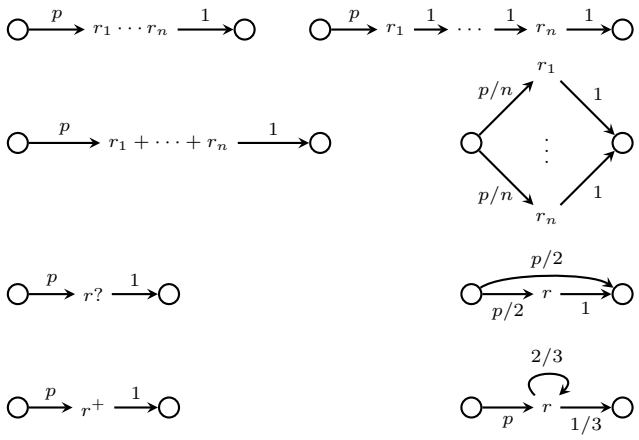
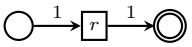



Fig. 8. From a regular expression to a probabilistic automaton.

is equally likely and the probability to have zero or one occurrences for the zero-or-one operator ? is 1/2 for each option. The probability to have $n$ repetitions in a one-or-more or zero-or-more operator ($^*$ and $^+$) is determined by the probability that we choose to continue looping (2/3) or choose to leave the loop (1/3). The latter values are based on observations of real-world corpora. Figure 8 illustrates how we construct the desired stochastic process from a regular expression $r$: starting from the following initial graph,




we continue applying the rewrite rules shown until each internal node is an individual alphabet symbol.

**Experiments on covering samples.** Our first experiment is designed to test how *i*DRegEx performs on samples that are at least large enough to *cover* the target regular expression, in the following sense.

*Definition* 5.1. A sample $S$ *covers* a deterministic automaton $G$ if for every edge $(s, t)$ in $G$ there is a word $w \in S$ whose unique accepting run in $G$ traverses $(s, t)$. Such a word $w$ is called a *witness* for $(s, t)$. A sample $S$ *covers* a deterministic regular expression $r$ if it covers the automaton obtained from $S$ using the Glushkov construction for translating regular expressions into automata as defined in Definition 4.7.

Intuitively, if a sample does not cover a target regular expression $r$ then there will be parts of $r$ that cannot be learned from $S$. In this sense, covering samples are the minimal samples necessary to learn $r$. Note that such samples are far from "complete" or "characteristic" in the sense of the theoretical framework of learning in the limit, as some characteristic samples are bound to be of size exponential in the size of $r$ by Theorem 3.2, while samples of size at most quadratic in $r$ suffice to cover $r$. Indeed, the Glushkov construction always yields an automaton whose number of states is bounded by the size of $r$. Therefore, this automaton can have

at most $|r|^2$ edges, and hence $|r|^2$ witness words suffice to cover $r$.

Table II shows how $i$DREGEX performs on covering samples, broken up by alphabet size of the target expressions. The size of the sample used is depicted as well. The table demonstrates a remarkable precision. Out of a total of 100 expressions, 82 are derived exactly for $i$DREGEX. Although $i$DREGEX(RWR$^0$) outperforms $i$DREGEX with a success rate of 87 %, overall $i$DREGEX$^{\text{fixed}}$ performs best with 89 %. The performance decreases with the alphabet size of the target expressions: this is to be expected since the inference task's complexity increases. It should be emphasized that even if $i$DREGEX$^{\text{fixed}}$ does not derive the target expression exactly, it always yields an over-approximation, i.e., its language is a superset of the target language.

Table III shows an alternative view on the results. It shows the success rate as a function of the target expression's language size, grouped in intervals. In particular, it demonstrates that the method works well for all language sizes.

A final perspective is offered in Table IV which shows the success rate in function of the average states per symbol $\kappa$ for an expression. The latter quantity is defined as the length of the regular expression excluding operators, divided by the alphabet size. For instance, for the expression $a(a+b)^+cab$, $\kappa = 6/3$ since its length excluding operators is 6 and $|\Sigma| = 3$. It is clear that the learning task is harder for increasing values of $\kappa$. To verify the latter, a few extra expressions with large $\kappa$ values were added to the target expressions. For the algorithm $i$DREGEX$^{\text{fixed}}$ the success rate is quite high for target expressions with a large value of $\kappa$. Conversely, $i$DREGEX(RWR$^0$) yields better results for $\kappa < 1.6$, while its success rate drops to around 50 % for larger values of $\kappa$. This illustrates that neither $i$DREGEX(RWR$^0$) nor $i$DREGEX$^{\text{fixed}}$ outperforms the other in all situations.

| $\|\Sigma\|$ | #regex | $i$DREGEX(RWR$^0$) | $i$DREGEX | $i$DREGEX$^{\text{fixed}}$ | $\|S\|$ |
|---|---|---|---|---|---|
| 5 | 45 | 86 % | 97 % | 100 % | 300 |
| 10 | 45 | 93 % | 75 % | 84 % | 1000 |
| 15 | 10 | 70 % | 50 % | 60 % | 1500 |
| total | 100 | 87 % | 82 % | **89 %** | |

Table II. Success rate on the target regular expressions and the sample size used per alphabet size for the various algorithms.

| Density($r$) | #regex | $i$DREGEX(RWR$^0$) | $i$DREGEX | $i$DREGEX$^{\text{fixed}}$ |
|---|---|---|---|---|
| $[0.0, 0.2[$ | 24 | 100 % | 87 % | 96 % |
| $[0.2, 0.4[$ | 22 | 82 % | 91 % | 91 % |
| $[0.4, 0.6[$ | 20 | 90 % | 75 % | 85 % |
| $[0.6, 0.8[$ | 22 | 95 % | 72 % | 83 % |
| $[0.8, 1.0]$ | 12 | 83 % | 78 % | 78 % |

Table III. Success rate on the target regular expressions, grouped by language size.

It is also interesting to note that $i$DREGEX successfully derived the regular expression $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$ of Theorem 3.2 for $n = 8$, $n = 10$, and $n = 12$ from covering samples of size 500, 800, and 1100, respectively. This is quite surprising considering that the characteristic samples for these expressions was proven to

| $\kappa$ | #regex | $i$DREGEX(RWR$^0$) | $i$DREGEX | $i$DREGEX$^{\text{fixed}}$ |
|---|---|---|---|---|
| [1.2, 1.4[ | 29 | 96 % | 72 % | 83 % |
| [1.4, 1.6[ | 37 | 100 % | 89 % | 89 % |
| [1.6, 1.8[ | 24 | 91 % | 92 % | 100 % |
| [1.8, 2.0[ | 11 | 54 % | 91 % | 100 % |
| [2.0, 2.5[ | 12 | 41 % | 50 % | 50 % |
| [2.5, 3.0] | 18 | 66 % | 71 % | 78 % |

Table IV. Success rate on the target regular expressions, grouped by $\kappa$, the average number of states per symbol.

be of size at least $(n-2)!$, i.e., 720, 40320, and 3628800 respectively. The regular expression $r_2 = (\Sigma \setminus a_1)^+ a_1 (\Sigma \setminus a_1)^+$, in contrast, was not derivable by $i$DREGEX from small samples.

**Experiments on partially covering samples.** Unfortunately, samples to learn regular expressions from are often smaller than one would prefer. In an extreme, but not uncommon case, the sample does not even entirely cover the target expression. In this section we therefore test how $i$DREGEX performs on such samples.

*Definition* 5.2. The *coverage* of a target regular expression $r$ by a sample $S$ is defined as the fraction of transitions in the corresponding Glushkov automaton for $r$ that have at least one witness in $S$.

Note that to successfully learn $r$ from a partially covering sample, $i$DREGEX needs to "guess" the edges for which there is no witness in $S$. This guessing capability is built into $i$DREGEX(RWR$^0$) and $i$DREGEX in the form of repair rules [Bex et al. 2006; Bex et al. 2008]. Our experiments show that for target expressions with alphabet size $|\Sigma| = 10$, this is highly effective for $i$DREGEX(RWR$^0$): even at a coverage of 70%, half the target expressions can still be learned correctly as Table V shows. The algorithm $i$DREGEX is performing very poorly in this setting, being only successful occasionally for coverages close to 100 %. $i$DREGEX$^{\text{fixed}}$ performs better, although not as well as $i$DREGEX(RWR$^0$). This again illustrates that both algorithms have their merits.

| coverage | $i$DREGEX(RWR$^0$) | $i$DREGEX | $i$DREGEX$^{\text{fixed}}$ |
|---|---|---|---|
| 1.0 | 100 % | 80 % | 80 % |
| 0.9 | 64 % | 20 % | 60 % |
| 0.8 | 60 % | 0 % | 40 % |
| 0.7 | 52 % | 0 % | 0 % |
| 0.6 | 0 % | 0 % | 0 % |

Table V. Success rate for 25 target expressions for $|\Sigma| = 10$ for samples that provide partial coverage of the target expressions.

We also experimented with target expressions with alphabet size $|\Sigma| = 5$. In this case, the results were not very promising for $i$DREGEX(RWR$^0$), but as Table VI illustrates, $i$DREGEX and $i$DREGEX$^{\text{fixed}}$ performs better, on par with the target expressions for $|\Sigma| = 10$ in the case of $i$DREGEX$^{\text{fixed}}$. This is interesting since the absolute amount of information missing for smaller regular expressions is larger than in the case of larger expressions.

| coverage | $i$DRegEx(rwr$^0$) | $i$DRegEx | $i$DRegEx$^{\text{fixed}}$ |
|---|---|---|---|
| 1.0 | 100 % | 100 % | 100 % |
| 0.9 | 25 % | 75 % | 66 % |
| 0.8 | 16 % | 75 % | 41 % |
| 0.7 | 8 % | 25 % | 33 % |
| 0.6 | 8 % | 25 % | 17 % |
| 0.5 | 0 % | 8 % | 17 % |

Table VI. Success rate for 12 target expressions for $|\Sigma| = 5$ with partially covering samples.

## 6. CONCLUSIONS

We presented the algorithm $i$DRegEx for inferring a deterministic regular expression from a sample of words. Motivated by regular expressions occurring in practice, we use a novel measure based on the number $k$ of occurrences of the same alphabet symbol and derive expressions for increasing values of $k$. We demonstrated the remarkable effectiveness of $i$DRegEx on a large corpus of real-world and synthetic regular expressions of different densities.

Our experiments show that $i$DRegEx(rwr$^0$) performs better than $i$DRegEx for target expressions with a $\kappa < 1.6$ and vice versa for larger values of $\kappa$. For partially covering samples, $i$DRegEx(rwr$^0$) is more robust than $i$DRegEx. As $\kappa$ values and sample coverage are not known in advance, it makes sense to run both algorithms and select the smallest expression or the one with the smallest language size, depending on the application at hand.

Some questions need further attention. First, in our experiments, $i$DRegEx always derived the correct expression or a super-approximation of the target expression. It remains to investigate for which kind of input samples this behavior can be formally proved. Second, it would also be interesting to characterize precisely which classes of expressions can be learned with our method. Although the parameter $\kappa$ explains this to some extend, we probably need more fine grained measures. A last and obvious goal for future work is to speed up the inference of the probabilistic automaton which forms the bottleneck of the proposed algorithm. A possibility is to use an industrial strength implementation of the Baum-Welsh algorithm as in [Finn et al. 2006] rather than a straightforward one or to explore different methods for learning probabilistic automata.

Although $i$DRegEx can be directly plugged into the XSD inference engine $i$XSD of [Bex et al. 2007], it would be interesting to investigate how to extend these techniques to the more robust class of Relax NG schemas [Clark and Murata 2001].

## REFERENCES

Castor. `www.castor.org`.

SUN Microsystems JAXB. `java.sun.com/webservices/jaxb`.

Adriaans, P. and Vitányi, P. 2006. The Power and Perils of MDL.

Ahonen, H. 1996. Generating Grammars for structured documents using grammatical inference methods. Report A-1996-4, Department of Computer Science, University of Finland.

Angluin, D. and Smith, C. H. 1983. Inductive Inference: Theory and Methods. *ACM Computing Surveys 15,* 3, 237–269.

Barbosa, D., Mignet, L., and Veltri, P. 2005. Studying the XML Web: gathering statistics from an XML sample. *World Wide Web 8,* 4, 413–438.

Benedikt, M., Fan, W., and Geerts, F. 2005. XPath satisfiability in the presence of DTDs. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 25–36.

Bernstein, P. A. 2003. Applying Model Management to Classical Meta Data Problems. In *First Biennial Conference on Innovative Data Systems Research*.

Bex, G., Neven, F., Schwentick, T., and Vansummeren, S. Inference of Concise Regular Expressions and DTDs. *ACM TODS*. To Appear.

Bex, G. J., Gelade, W., Neven, F., and Vansummeren, S. 2008. Learning deterministic regular expressions for the inference of schemas from XML data. In *WWW*. Beijing, China, 825–834. Accepted for WWW 2008.

Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. 2006. Inference of concise DTDs from XML data. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 115–126.

Bex, G. J., Neven, F., Schwentick, T., and Vansummeren, S. 2008. Inference of Concise Regular Expressions and DTDs. submitted to VLDB Journal.

Bex, G. J., Neven, F., and Van den Bussche, J. 2004. DTDs versus XML Schema: a practical study. In *Proceedings of the 7th International Workshop on the Web and Databases*. 79–84.

Bex, G. J., Neven, F., and Vansummeren, S. 2007. Inferring XML Schema Definitions from XML data. In *Proceedings of the 33rd International Conference on Very Large Databases*. 998–1009.

Brāzma, A. 1993. Efficient identification of regular expressions from representative examples. In *Proceedings of the 6th Annual ACM Conference on Computational Learning Theory*. ACM Press, 236–242.

Brüggeman-Klein, A. 1993. Regular expressions into finite automata. *Theoretical Computer Science 120,* 2, 197–213.

Brüggemann-Klein, A. and Wood, D. 1998. One-unambiguous regular languages. *Information and computation 140,* 2, 229–253.

Buneman, P., Davidson, S. B., Fernandez, M. F., and Suciu, D. 1997. Adding structure to unstructured data. In *Database Theory - ICDT '97, 6th International Conference*, F. N. Afrati and P. G. Kolaitis, Eds. Lecture Notes in Computer Science, vol. 1186. Springer, 336–350.

Che, D., Aberer, K., and Özsu, M. T. 2006. Query optimization in XML structured-document databases. *VLDB Journal 15,* 3, 263–289.

Chidlovskii, B. 2001. Schema extraction from XML: a grammatical inference approach. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases*.

Clark, J. Trang: Multi-format schema converter based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`.

Clark, J. and Murata, M. 2001. *RELAX NG Specification*. OASIS.

Cover, R. 2003. The Cover Pages. http://xml.coverpages.org/.

Du, F., Amer-Yahia, S., and Freire, J. 2004. ShreX: Managing XML Documents in Relational Databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 1297–1300.

Ehrenfeucht, A. and Zeiger, P. 1976. Complexity measures for regular expressions. *Journal of computer and system sciences 12*, 134–146.

Fernau, H. 2004. Extracting minimum length Document Type Definitions is NP-hard. In *ICGI*. 277–278.

Fernau, H. 2005. Algorithms for Learning Regular Expressions. In *Algorithmic Learning Theory, 16th International Conference*. 297–311.

Finn, R., Mistry, J., Schuster-Bckler, B., Griffiths-Jones, S., et al. 2006. Pfam: clans, web tools and services. *Nucleic Acids Research 34*, D247–D251.

Florescu, D. 2005. Managing semi-structured data. *ACM Queue 3,* 8 (October).

François, J.-M. 2006. Jahmm. `http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/`.

Freire, J., Haritsa, J. R., Ramanath, M., Roy, P., and Siméon, J. 2002. StatiX: making XML count. In *SIGMOD Conference*. 181–191.

Freitag, D. and McCallum, A. 2000. Information Extraction with HMM Structures Learned by Stochastic Optimization. In *AAAI/IAAI*. AAAI Press / The MIT Press, 584–589.

Garcia, P. and Vidal, E. 1990. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12,* 9 (September), 920–925.

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., and Shim, K. 2003. XTRACT: learning document type descriptors from XML document collections. *Data mining and knowledge discovery 7*, 23–56.

Gelade, W. and Neven, F. 2008. Succinctness of the Complement and Intersection of Regular Expressions. In *STACS*. 325–336.

Gold, E. 1967. Language identification in the limit. *Information and Control 10,* 5 (May), 447–474.

Goldman, R. and Widom, J. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*. 436–445.

Gruber, H. and Holzer, M. 2008. Finite Automata, Digraph Connectivity, and Regular Expression Size. In *ICALP (2)*. 39–50.

Hegewald, J., Naumann, F., and Weis, M. 2006. XStruct: efficient schema extraction from multiple and large XML documents. In *ICDE Workshops*. 81.

Hopcroft, J. and Ullman, J. 2007. *Introduction to automata theory, languages and computation*. Addison-Wesley, Reading, MA.

Koch, C., Scherzinger, S., Schweikardt, N., and Stegmaier, B. 2004. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 228–239.

Manolescu, I., Florescu, D., and Kossmann, D. 2001. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of 27th International Conference on Very Large Data Bases*. 241–250.

Martens, W., Neven, F., Schwentick, T., and Bex, G. J. 2006. Expressiveness and Complexity of XML Schema. *ACM Transactions on Database Systems 31,* 3, 770–813.

Mignet, L., Barbosa, D., and Veltri, P. 2003. The XML web: a first study. In *Proceedings of the 12th International World Wide Web Conference*. Budapest, Hungary, 500–510.

Nestorov, S., Abiteboul, S., and Motwani, R. 1998. Extracting Schema from Semistructured Data. In *International Conference on Management of Data*. ACM Press, 295–306.

Neven, F. and Schwentick, T. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science 2,* 3.

Pitt, L. 1989. Inductive Inference, DFAs, and Computational Complexity. In *Proceedings of the International Workshop on Analogical and Inductive Inference*, K. P. Jantke, Ed. Lecture Notes in Computer Science, vol. 397. Springer-Verlag, 18–44.

Quass, D., Widom, J., Goldman, R., et al. 1996. LORE: a Lightweight Object REpository for semistructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 549.

Rabiner, L. 1989. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proc. IEEE 77,* 2, 257–286.

Rahm, E. and Bernstein, P. A. 2001. A survey of approaches to automatic schema matching. *VLDB Journal 10,* 4, 334–350.

Sahuguet, A. 2000. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask (Extended Abstract). In *The World Wide Web and Databases, 3rd International Workshop*, D. Suciu and G. Vossen, Eds. Lecture Notes in Computer Science, vol. 1997. Springer, 171–183.

Sakakibara, Y. 1997. Recent advances of grammatical inference. *Theoretical Computer Science 185,* 1, 15–45.

Sankey, J. and Wong, R. K. 2001. Structural inference for semistructured data. In *Proceedings of the 10th international conference on Information and knowledge management*. ACM Press, 159–166.

Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. 2001. *XML Schema part 1: structures*. W3C.

Young-Lai, M. and Tompa, F. W. 2000. Stochastic Grammatical Inference of Text Database Structure. *Machine Learning 40,* 2, 111–137.

Received Month Year; revised Month Year; accepted Month Year