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

Geert Jan Bex Wouter Gelade[*] Frank Neven Stijn Vansummeren[†]

Hasselt University/Transnational University of Limburg, Belgium
firstname.lastname@uhasselt.be

ABSTRACT

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 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 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

Keywords

XML, regular expressions, schema inference

1. INTRODUCTION

Recent studies stipulate that schemas accompanying collections of XML documents are sparse and erroneous in practice. Indeed, Barbosa et al. [6, 39] have shown that approximately half of the XML documents available on the web do not refer to a schema. In addition, Bex et al. [11, 38] 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 [50], rendering them essentially useless for immedidate application. A similar observation was made by Sahuguet [47] concerning Document Type Definitions (DTDs). Nevertheless, the presence of a schema strongly facilitates optimization of XML processing (cf., e.g., [7, 18, 23, 28, 36, 37, 42]) and various software development tools such as Castor [1] and SUN's JAXB [2] 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 [46] and in the area of generic model management [8]. Based on the above described benefits of schemas and their unavailability in practice, it is essential to devise algorithms that can infer a schema for a given collection of XML documents when no (valid) one is present. The latter problem is also acknowledged by Florescu [26] 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*".

Both DTDs and XSDs can be conveniently abstracted by grammar rules with regular expressions on the right-hand side [38, 40]. Schema inference then reduces to learning regular expressions from a set of example strings [10, 12, 31]. To infer a DTD, for example, it suffices to derive for every element name n a regular expression describing the strings of element names allowed to occur below n. To illustrate, from the strings `author title`, `author title year`, and `author author title year` appearing under `<book>` elements in a sample XML corpus, we could derive the rule

$$\texttt{book} \rightarrow \texttt{author}^{+}\ \texttt{title}\ \texttt{year}?$$

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 [20] and XStruct [34] simply infer DTDs in XSD syntax. The more recent *i*XSD algorithm [12] 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 learning al-

[*] Research Assistant of the Research Foundation - Flanders (FWO).

[†] Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

gorithms is strongly determined by the accuracy of the employed regular expression learning method. The present paper presents a method to reliably learn regular expressions that are far more complex than the classes of expressions previously considered in the literature.

The W3C specification of both DTDs and XSDs requires the regular expressions occurring in them to be *deterministic* (also known as the Unique Particle Attribution in [50]). We extend Gold's [32] seminal result, stating the impossibility to learn the complete class of regular expressions from positive examples only, to the subclass of deterministic regular expressions. This means that no matter how many example strings are provided, no algorithm can learn every target deterministic regular expression. By the dependence of schema inference on regular expression learning, this negative result also excludes the possibility of a sound and complete inference algorithm for the complete classes of DTDs and XSDs.

An examination of 819 DTDs and XSDs gathered form the Cover Pages [22] (including many high-quality XML standards) as well as from the web at large reveals, however, that the regular expressions occurring in practical schemas are such that every alphabet symbol occurs only a small number of times [38]. 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* (k-OREs for short). For example, $((b?(a+c))^+d)^+)e$ is a 1-ORE while $a(a+b)^*$ is a 2-ORE.

Actually, the above examination showed that in the majority of the cases $k = 1$. Motivated by this observation, Bex et al [10] provided a practical learning algorithm for the class of deterministic 1-OREs. Unfortunately, this algorithm can only output 1-OREs even when the target regular expression is not and will in that case only return an approximation of the target expression. It is therefore desirable to also have learning algorithms for the class of deterministic k-OREs with $k \geq 2$. Furthermore, as 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 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 iDRegEx. Given a sample of strings S, iDRegEx derives corresponding deterministic k-OREs for increasing values of k and utilizes a *minimum description length* argument to choose the best one. The main technical contribution lies in the subroutine iKore 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 [30], and by subsequently translating A into a 1-ORE [10], 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 [30]. Although every language definable by a 1-ORE is 2-testable [10], there are languages definable by a 2-ORE, like a^*ba^*, that are not n-testable for any n. iKore therefore uses 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 iDRegEx is empirically validated in Section 5 both on real world and synthetic data. In particular, we show that the algorithm is conservative over the real-world corpus of 1-OREs in [12]. That is, iDRegEx chooses each time the target 1-ORE from among all generated candidate expressions (including some 4-OREs). We also tested iDRegEx on a set of 100 synthetic expressions of diverse densities. From these 88 where derived exactly, for the remaining 12 a super-approximation was obtained.

2. RELATED WORK

Semi-structured data. In the context of semi-structured data, the inference of schemas as defined in [16, 44] has been extensively studied [33, 41]. 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 [10] 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 [20], state of the art software written by James Clark that is primarily intended as a translator between the schema languages DTD, Relax NG [21], and XSD, but also infers a schema for a set of XML documents. In contrast, our goal in this paper is to infer the more general class of deterministic expressions. xtract [31] is another regular expression learning system with similar goals. We note that xtract also uses the MDL principle to choose the best expression from a set of candidates. We compare these approaches in Section 5.

Other relevant DTD inference research is [49] and [19] that learn finite automata but do not consider the translation to deterministic regular expressions. Also, in [51] 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 [4] 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 [12, 20, 34] 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 [5, 43, 48]. 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 [13] showed that regular expressions without union can be approximately learned in polynomial time from a set of examples satisfying some criteria. Fernau [24] 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 [45, 29], the requirement in our setting that the resulting automaton is deterministic is, to the best of our knowledge, unique.

3. DEFINITIONS AND BASIC RESULTS

Deterministic regular expressions. In this article we are interested in learning regular expressions r of the form

$$r ::= \varepsilon \mid \sigma \mid r\,r \mid r + r \mid r? \mid r^+ \mid r^*,$$

where ε denotes the empty word and σ ranges over symbols in a finite alphabet Σ. Note that the empty language ($\emptyset$) is not allowed as basic expression. As usual, we write $L(r)$ for the language defined by regular expression r.

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 [15]). 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. Formally, let $\overline{r}$ stand for the regular expression obtained from r by replacing the ith occurrence of alphabet symbol σ in r by σ_i, for every i and σ. For example, for $r = b^*a(b^*a)^*$ we have $\overline{r} = b_1^*a_1(b_2^*a_2)^*$.

Definition 1. *A regular expression r is* deterministic *if there are no words $w\sigma_iv$ and $w\sigma_jv'$ in $L(\overline{r})$ such that $i \neq j$.*

Equivalently, an expression is deterministic if the Glushkov-construction translates it into a deterministic finite automaton rather than a non-deterministic one [15]. Not every non-deterministic regular expression is equivalent to a deterministic one [15]. Thus, semantically, the class of deterministic regular expressions forms a strict subclass of the class of all regular expressions.

Learning in the limit. 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* [32], such an algorithm is said to learn the deterministic regular expressions from positive data.

Definition 2. *Define a* sample *to be a finite set of words and let $\mathcal{R}$ be a subclass of the regular expressions. An algorithm M mapping samples to expressions in $\mathcal{R}$ is said to* learn $\mathcal{R}$ from positive data *if (1) $S \subseteq L(M(S))$ for every sample S and (2) to every $r \in \mathcal{R}$ we can associate a so-called* characteristic sample $S_r \subseteq L(r)$ *such that, for each sample S with $S_r \subseteq S \subseteq L(r)$, $M(S)$ 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 [32]. In fact, he showed that every class of regular expressions that contains all non-empty finite languages and at least one infinite language is not learnable in the limit from positive data. Since deterministic regular expressions like a^* define infinite languages, and since every non-empty finite language can be defined by a deterministic expression (as we show in the full version of this paper [9]), it follows that also the class of deterministic regular expressions is not learnable in the limit.

Theorem 1. *The class of deterministic regular expressions is not learnable in the limit from positive data.*

Bounded number of symbol occurrences. Theorem 1 immediately excludes the possibility for an algorithm to infer the full class of DTDs, and therefore also the even larger class of 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 [22] (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 at most k times, with k small. Actually, in the majority of the cases $k = 1$.

Definition 3. *A regular expression is* k-occurrence *if every alphabet symbol occurs at most k times in it.*

For example, $(b?(a + c))^+d$ is a 1-ORE, $a(a + b)^*$ is a 2-ORE, and $a(ba)^*a$ is 3-ORE. We abbreviate 'k-occurrence regular expression' by k-ORE and denote the class of all deterministic k-OREs over Σ by Dk-ORE(Σ). We prove in the full version of this paper [9]:

Theorem 2. *For every k and Σ there exists an algorithm M that learns* Dk-ORE(Σ) *from positive data. Furthermore, M runs in time polynomial in the size of the input sample, and in time exponential in k and the size of Σ.*

While this 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.

First and foremost, M runs in time exponential in the size of the alphabet. Second 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 expressions for which the characteristic sample of any sound and complete learning algorithm is at least exponential in the size of the alphabet.

Theorem 3. *Let $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ be an alphabet of size n, let $r_1 = (\sigma_1\sigma_2 + \sigma_3 + \cdots + \sigma_n)^+$, and let $r_2 = \Sigma^+\sigma_1\Sigma^+$. For any algorithm that learns* D$(2n+3)$-ORE(Σ) *and any sample S that is characteristic for r_1 or r_2 we have $|S| \geq \sum_{i=1}^{n}(n-2)^i$.*

The proof is in [9]. Since $\sum_{i=1}^{n}(n-2)^i > (n-2)!$, it is unlikely for any sound and complete learning algorithm to behave well on real-world samples, which are typically incomplete and unlikely to contain all words of the characteristic sample.

Third, while the study referred to above reveals that all expressions in practical schemas are k-OREs for some small k, the schema inference setting is such that we do not know the exact value of k when given a sample. If we overestimate k then $M(S)$ risks being an under-approximation of the target regular expression r especially when S is incomplete. To illustrate, consider the 1-ORE target expression $r = a^+b^+$ and the 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 $M(S)$ risk 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 hence need a method to determine the most suitable value of k.

4. THE LEARNING ALGORITHM

In view of the observations made in the previous section, we present in this section a practical learning algorithm that works well on incomplete data and automatically determines the best value of k. Specifically, given a sample S, the algorithm derives deterministic k-OREs for increasing values of k and selects from these candidate expressions the one which best describes S based on a Minimum Description Length argument. The 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.

Translating automata into k-OREs. It is notoriously difficult, however, to translate an *arbitrary* automaton into a k-ORE when k is fixed. The standard state elimination algorithm [35], for instance, often duplicates subexpressions whenever it eliminates a state, and therefore almost always returns an expression of size exponential in the number of states of the automaton.

Our algorithm therefore does not learn an arbitrary automaton, but a *deterministic* k-OA, a specific kind of deterministic automaton with (1) labels placed on states rather than on edges in which (2) each alphabet symbol occurs at most k times. Figure 2(f) gives an example. The underlying idea is that every edge carries the label of the state it points to. In this context, an *accepting run* for a word $\sigma_1 \dots \sigma_n$ is a path $s_1 \dots s_n s_{\text{out}}$ that starts at an initial state and stops at the unique final state s_{out} such that s_1 is labeled by σ_1, s_2 by σ_2, ..., and s_n by σ_n. In other words, the 2-OA of Figure 2(f) accepts the same language as $aa?b^+$.

Definition 4. *Formally, an* automaton over Σ *is a tuple* (V, E, I, s_{out}, lab) *where* V *is a finite set of states,* $E \subseteq (V \setminus \{s_{out}\}) \times V$ *is the edge relation,* $I \subseteq V$ *are the initial states,* s_{out} *is the final state, and* $lab\colon V \setminus \{s_{out}\} \to \Sigma$ *is the labeling function. A* k-occurrence automaton (k-OA) *is an automaton in which every* Σ*-symbol labels at most* k *states.*

We write $L(A)$ for the language defined by A; $\text{out}(s)$ for the set $\{s' \mid (s, s') \in E\}$ of all states reachable by an outgoing edge from s in automaton A; and $\text{in}(s)$ for the set $\{s' \mid (s', s) \in E\}$ of all states for which s has an incoming edge in A. Furthermore, we write $\text{out}_\sigma(s)$ and $\text{in}_\sigma(s)$ for the set of states in $\text{out}(s)$ and $\text{in}(s)$, respectively, that are labeled by σ. As usual, an automaton A is *deterministic* if $\text{out}_\sigma(s)$ contains at most one state, for every s and σ.

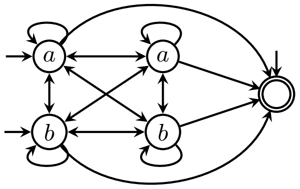


Figure 1: The complete 2-OA **over** $\Sigma = \{a, b\}$**.**

It is sufficient to learn only deterministic k-OAs since every deterministic k-ORE can be represented as one. Indeed, the Glushkov construction [15] for translating regular expressions into automata always outputs a deterministic k-OA on deterministic k-ORE inputs. Moreover, in previous work Bex et al. [10] have already proposed an algorithm for translating k-OAs into k-OREs. This algorithm, which we will refer to as KoaToKore in this paper[1], essentially applies an *inverse* Glushkov construction to its input. It is an appealing algorithm due to the following properties, which we prove in the full version of this paper [9]:

Theorem 4. KoaToKore *is sound in the sense that, for every* k-OA A*, it outputs a (possibly non-deterministic when* $k > 1$*)* k-ORE r *with* $L(A) \subseteq L(r)$*. It is also complete in the sense that if* A *is the Glushkov translation of a deterministic* k-ORE r'*, then* r *is deterministic and equivalent to* r'*.*

In other words, if we manage to learn a k-OA that is the Glushkov representation of the target expression r', then KoaToKore 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, KoaToKore 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 KoaToKore is always deterministic (as every 1-ORE is deterministic), we always infer at least one deterministic regular expression. Moreover, to determine the best regular expression from the set of generated expressions, only the deterministic ones are considered. 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.

Learning k-OAs probabilistically. Define the *complete* k-OA C_k over alphabet Σ to be the fully connected automaton over Σ that has exactly k states per alphabet symbol and whose initial states consist of the final state plus one representative state for each symbol. To illustrate, the complete 2-OA over $\{a, b\}$ is shown in Figure 1. Clearly, $L(C_k) = \Sigma^*$.

We use a probabilistic method to learn a deterministic k-OA for a sample S. Specifically, we view S as the re-

[1] Unfortunately, KoaToKore is called iDTD in [10], although its sole purpose is to transform k-OAs into k-OREs, not DTDs. We call it KoaToKore in this paper to avoid confusion. Also note that KoaToKore is only illustrated and shown sound and complete on 1-OAs in [10], although its literal definition applies to all k-OAs.

sult of a stochastic process that generates words from Σ^* by performing random walks over C_k. First, the process picks, among all initial states, an initial state s_1 with probability $\pi(s_1)$ and emits $lab(s_1)$. Then it picks, among all states in $\text{out}(s_1)$ 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, π and α must be true probability distributions, i.e., $\pi(s)$ and $\alpha(s,t)$ are greater or equal than zero for all s, t;

$$\sum_{\text{all initial states } s} \pi(s) = 1; \quad \text{and} \quad \sum_{t \in \text{out}(s)} \alpha(s,t) = 1 \qquad (1)$$

for all states s. The probability of generating a particular accepting run $\vec{s} = s_1 s_2 \dots s_n s_{\text{out}}$ given the process $\mathcal{P} = (C_k, \pi, \alpha)$ in this setting is

$$P[\vec{s} \mid \mathcal{P}] = \pi(s_1) \cdot \alpha(s_1, s_2) \cdot \alpha(s_2, s_3) \cdots \alpha(s_n, s_{\text{out}}),$$

and the probability of generating the word $w = \sigma_1 \dots \sigma_n$ 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 π and α 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 probabilities π and α that maximize $P[S \mid \mathcal{P}]$, and use these probabilities 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 π and α 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)[2]. Inference of π and α is generally accomplished by the well-known Baum-Welsh algorithm [45] that adjusts initial values for π and α until a (possibly local) maximum is reached.

We use Baum-Welsh in our learning algorithm *i*KORE as shown in Algorithm 1. In line 1, *i*KORE initializes the stochastic process $\mathcal{P}$ to the triple (C_k, π, α) where the alphabet of C_k is taken to consist of all symbols occurring in words in S; α is chosen randomly (subject to the constraints in (1)); $\pi(s_{\text{out}})$ is the fraction of empty strings in S; and for every other initial state s with $lab(s) = \sigma$, $\pi(s)$ is the fraction of words in S that start with σ. 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 no longer be considered as a set, but as a *bag*. Line 2 then optimizes the initial values for π and α using the Baum-Welsh algorithm.

Algorithm 1 *i*KORE

```
Require: a sample $S$, a value for $k$
Ensure: a $k$-ORE $r$ with $S \subseteq L(r)$
 1: $\mathcal{P} \leftarrow \text{init}(k, S)$
 2: $\mathcal{P} \leftarrow$ BAUMWELSH$(\mathcal{P}, S)$
 3: $A \leftarrow$ DISAMBIGUATE$(\mathcal{P}, S)$
 4: $A \leftarrow$ PRUNE$(A, S)$
 5: return KOATOKORE$(A)$
```

Algorithm 2 DISAMBIGUATE

```
Require: a POMM $\mathcal{P} = (A, \pi, \alpha)$ and sample $S$
Ensure: a deterministic $k$-OA
 1: Initialize queue $Q$ to the initial states of $A$ with $\pi(s) > 0$
 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 $\sigma \in \Sigma$ has $|\text{out}_\sigma(s)| > 1$ do
 6:       pick $t \in \text{out}_\sigma(s)$ with $\alpha(s,t) = \max_{t' \in \text{out}(s)} \alpha(s,t')$
 7:       set $\alpha(s,t) \leftarrow \sum_{t' \in \text{out}_\sigma(s)} \alpha(s,t')$
 8:       for all $t'$ in $\text{out}_\sigma(s) \setminus \{t\}$ do
 9:          delete edge $(s,t')$ from $A$
10:          set $\alpha(s,t') \leftarrow 0$
11:       $\mathcal{P} \leftarrow$ BAUMWELSH$(\mathcal{P}, S)$
12:       if $S \not\subseteq L(A)$ then Fail
13:    add $s$ to marked states $D$ and pop $s$ from $Q$
14:    enqueue all states in $\text{out}(s) \setminus D$ to $Q$
15: return $A$
```

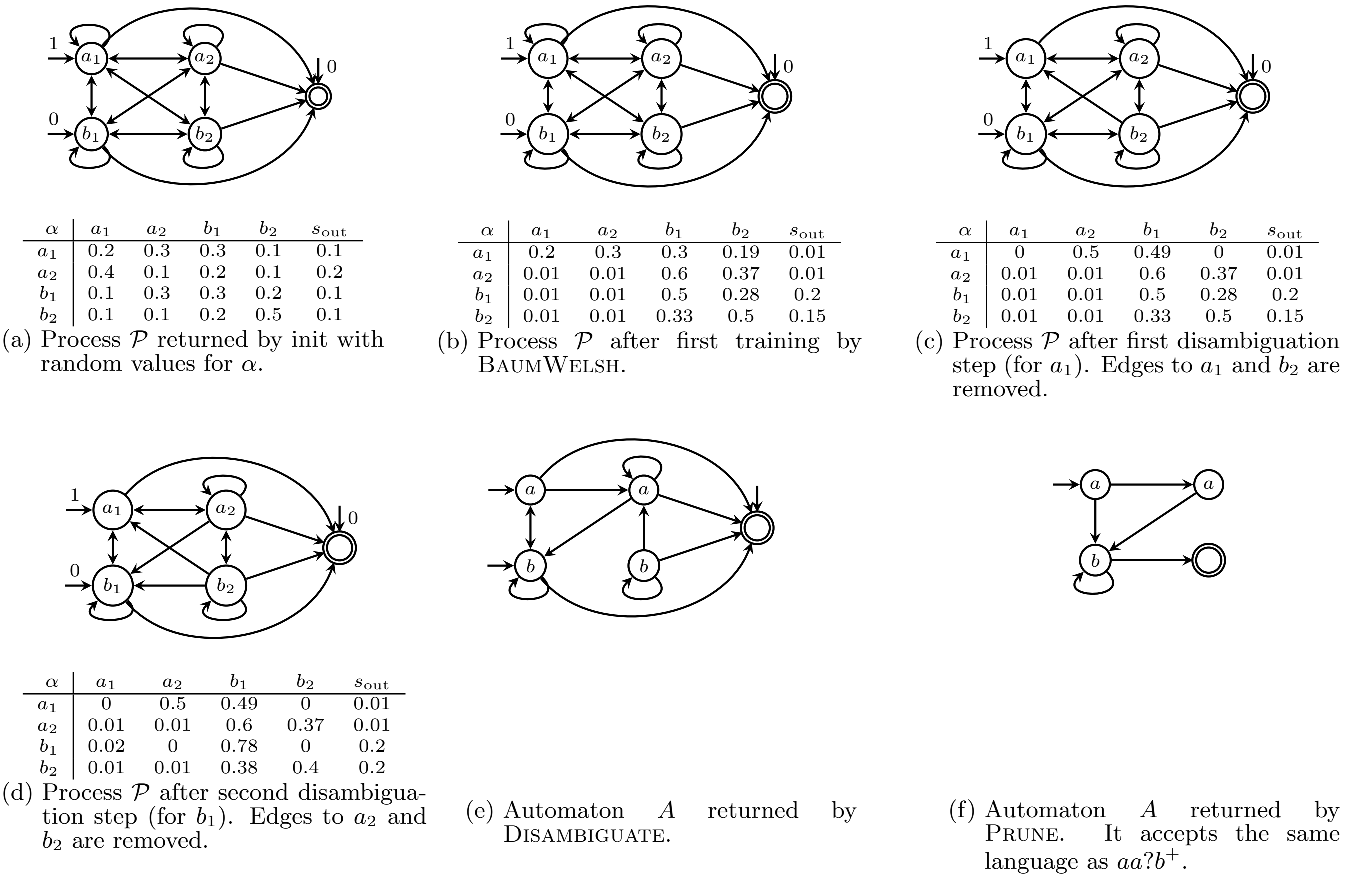
With these probabilities in hand DISAMBIGUATE, shown in Algorithm 2, determines the topology of the desired deterministic k-OA for S. In a breadth-first manner, it picks for each state s and each symbol σ the state $t \in \text{out}_\sigma(s)$ with the highest probability and deletes all other edges to states labeled by σ. Line 7 merely ensures that α continues to be a probability distribution after this removal and line 11 adjusts π and α 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*KORE removes all such edges. It also removes all inital states without a witness in S. Finally, line 5 transforms the resulting deterministic k-OA into a k-ORE. Figure 2 illustrates a hypothetical run of *i*KORE.

Denote by *i*1-ORE(S) the earlier algorithm of [10] applied to sample S which derives a 1-ORE by first learning a 1-OA through the algorithm 2T-INF [30] and then applies KOATOKORE to obtain a 1-ORE. The next Theorem says that *i*KORE is a *conservative* extension of 1-ORE as for $k = 1$, it produces the same output as *i*1-ORE (though in a very inefficient manner).

THEOREM 5. *For every sample S, iKORE$(S, 1)$ does not fail and returns the same* 1-ORE *as* i1-ORE(S).

For the proof it suffices to note that on complete 1-OAs the DISAMBIGUATE step does not remove any edges and that the PRUNE step hence produces the same deterministic automaton as 2T-INF.

[2] In contrast to the general Hidden Markov Model, a POMM can emit only one alphabet symbol per state.



α	a_1	a_2	b_1	b_2	s_{out}
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 α.

α	a_1	a_2	b_1	b_2	s_{out}
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.

α	a_1	a_2	b_1	b_2	s_{out}
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.

α	a_1	a_2	b_1	b_2	s_{out}
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 $aa?b^+$.

Figure 2: Example run of iKore for $k = 2$ with target language $aa?b^+$. For the process $\mathcal{P}$ in (a)-(d), π is indicated on the edges of the initial states whereas α is listed in table-form. To distinguish different states with the same label, we have indexed the labels.

The whole algorithm. Unfortunately, it is well-known that, depending on the initial probabilities for π and α, BaumWelsh may converge to a local maximum that is not necessarily global. To increase the probability of finding a global maximum and hence correctly learning the target regular expression from S, we apply iKore a number of times N with independently chosen random seed values for α. The whole learning algorithm iDRegEx is then shown in Algorithm 3 where the function best(C) selects the "best" regular expression from the candidate set of expressions C according to the following measure.

The Minimum Description Length measure. Intuitively, we want to select from C the simplest deterministic expression that describes S the best, i.e., the expression that overgeneralizes S the least. To measure the degree of generalisation, we employ the data encoding cost of Adriaans and Vitányi [3] that compares the size of S with the size of the *language* defined by an inferred k-ORE r. Specifically, Adriaans and Vitányi define:

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

where $L^{\max} = 2k|\Sigma| + 1$, S^ℓ is the subset of words in S that have length ℓ, and $L^\ell(r)$ is the subset of words in $L(r)$ that have length ℓ. Although the above formula is numerically difficult to compute, there is an easier estimation procedure; see [3] for details.

The complexity of a regular expression, i.e., its 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 now the *deterministic* one that minimizes both model and data encoding cost.

We already mentioned that xtract [31] 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.

Algorithm 3 iDRegEx

```
Require: a sample $S$
Ensure: a $k$-ORE $r$
initialize candidate set $C \leftarrow \emptyset$
for $k = 1$ to $k_{\max}$ do
    for $n = 1$ to $N$ do
        add $i$Kore$(S, k)$ to $C$
return best$(C)$
```

5. EXPERIMENTS

In this section we validate our approach by means of an experimental analysis. All experiments were performed using a prototype implementation of *i*DRegEx written in Java executed on a Pentium M 1.73 GHz with 1GB RAM. For the BaumWelsh subroutine we have gratefully used Jean-Marc François' *Jahmm* library [27], which is a faithful implementation of the algorithms described in Rabiner's Hidden Markov Model tutorial [45]. 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, underflows currently limit us to regular expressions whose total number of symbol occurrences is at most 40. 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 expressions with alphabet size $|\Sigma| = 5$ to 9 hours for expressions with alphabet size 15, details are shown in Table 1. Experience with Hidden Markov Model learning in bio-informatics [25], for instance, suggests that both the running time and number of symbol occurrences 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.

$\|\Sigma\|$	mean	median	$\|S\|$
5	239	154	300
10	7246	3585	1000
15	31881	19537	1000

Table 1: Mean and median runtime in seconds for learning expressions of varying alphabet size.

For all the experiments described below we take $k_{\max} = 4$ and $N = 10$ in Algorithm 3.

5.1 Real-world target expressions and real-world samples

First, 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 [50] and have drawn multiset samples for these expressions from a large corpus of real-world XSDs harvested from the Cover Pages [22]. 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[3]. All 49 regular expressions were successfully derived by *i*DRegEx. Since these expressions are in fact 1-OREs and $k_{\max}$ is set to 4, this illustrates that *i*DRegEx is *conservative* over the learning algorithm presented in [10] and does not incorrectly infer a k-ORE with $k > 1$ when its target is a 1-ORE.

The number of publicly available XML corpora with non-1-ORE regular expressions in their schema is even more restricted, and in previous work we have only been able to identify two such real-world target expressions [10]. Because of the limitations of our BaumWelsh implementation described at the beginning of this section, we have reduced the alphabet size of the second expression yielding

$$a_1(a_2 + a_3)^*(a_4(a_2 + a_3 + a_5)^*)^* \text{ and}$$
$$a_1?a_2a_3?a_4?(a_5^+ + ((a_6 + \cdots + a_{10})^+a_5^*)).$$

*i*DRegEx successfully derived both expressions.

5.2 Synthetic target expressions

Although the successful inference of the real-world expressions in Section 5.1 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 expression overlaps with Σ^*. The larger the overlap, the greater its *density*. Specifically, we define

$$\text{Density}(r) = 1 + \frac{1}{\sum_{\ell=1}^{L_{\max}} \log(1 + |\Sigma^\ell|)} \sum_{\ell=1}^{L_{\max}} \log \frac{1 + |L^\ell(r)|}{1 + |\Sigma^\ell|}.$$

Here Σ^ℓ is the subset of words in Σ^* with length exactly ℓ; $L^\ell(r)$ is the subset of words in $L(r)$ with length exactly ℓ; and $L_{\max} = 2k|\Sigma|$. As such, the measure depends on the fraction of words over Σ with length ℓ that are in $L(r)$, for each ℓ up to $L_{\max}$. In particular, it is 1 iff $L(r) = \Sigma^*$ and is 0 iff $L(r) = \emptyset$. The measure can be used to ascertain the diversity of our set of generated regular expressions by ensuring that it covers the spectrum $[0, 1]$.

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 some straightforward equivalences. For instance, $(a^+)^+$ is rewritten into a^+. The entire set of rewrite rules can be found in [9]. 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 densities ranging from 0 to 1. All in all, this yielded a set of 100 deterministic target expressions. A snapshot is given in Figure 3. In [9] the complete list of expressions is listed.

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 disjunc-

[3] This corpus was also used in [12] for XSD inference.

$((debab) + c)^*a$
$((((c + b)b) + a)ca) + e + d$
$(((ea)^*db) + b + a + c)^+$
$((b^+ + c + e + d)aab)^+$
$((((aa) + e)^+ + c)b) + b + d$
$((((d + a)^*eabcb) + c)a)?$
$((((ac) + b + d)eab) + c)^*$
$(((((bab) + c)^+ + e)?a) + d)^+$
$((((ecb)^+a) + b)^+ + d + a)?$
$((bagbfeid) + c + a + j + h)^*$

$((gdab) + a + i + c + j + e + f)^+hb$
$((h^*cdfa) + j + e + g + b + i)^*ab$
$((g + b + e + f + i + d)^*aba) + h + j + c$
$((((h + b + c + j + f)^+ + e)?aaidb) + g)?$
$(((((dbe)^*cf) + j)hac) + b + i)^*gad$
$(((((ihaaj) + d)^+ + g)b) + e + b + f + c)^+$
$((((eabh) + d + j + c + b)^+f) + a + g + i)?$
$(((ecgecd) + b + d + a + j + f)^*ihaba)^*$
$(l + c + d + m + n)^*aojahbegcbfidke$
$(((c + b)ab) + d + i + a)^+ + j + g + f + e + h$

$(((a?clfhabgd) + b + n + o)iedjcem)^*k$
$((a + k + f + c + m + e)^+bdieclbonjgda)^*h$
$(((k?jghadfcelifcjbhom)+$
$b + g + a + e + i + n)^+ + d)?$
$(((aedoadenhdbci) + h + k + m + j + g + b)^*$
$fccgelbifja)$
$((a^+ + f + d + o + g + n + h + c + b + j + i + e)$
$keacdlbm)$
$(((k + f + o + a + j)?edhldfhngicjmab)?cie)^*bg$
$((((a?d)^+ba) + h + g + e + c)^+ + j + i + b)?f$

Figure 3: A snapshot of the 100 generated expressions.

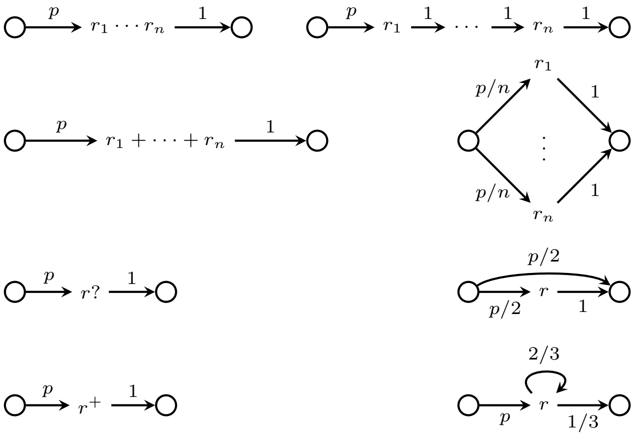
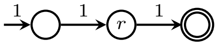


Figure 4: From a regular expression to a probabilistic automaton.

tion 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 4 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. *A sample S covers a deterministic automaton A if for every edge (s, t) in A there is a word $w \in S$ whose unique accepting run in A 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 [14].*

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, 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 2 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, 88 are derived exactly. For the remaining 12 expressions that could not be derived exactly, an over-approximation of the target expression was derived, i.e., if the sample covered r then $L(r)$ was a subset of the language of the derived expression.

Table 3 shows an alternative view on the results. It shows the success rate as a function of the density of the target expression, grouped in intervals. In particular, it demonstrates that the method works well for every density.

A final perspective is offered in Table 4 which shows the success rate in function of the average states per symbol κ for an expression. The latter quantity is defined as the length of the regular expression excluding operators, divided by its k-value. For instance, for the expression $a(a+b)^+cab$, $\kappa = 6/3$ since its length excluding operators is 6 and $k = 3$. It is clear that the learning task is harder for increasing values of κ. Up to values of $\kappa \approx 1.7$, the success rate is quite high, while it drops to around 50 % for larger values of κ. To verify the latter, a few extra expressions with large κ values were added to the target expressions.

$\|\Sigma\|$	#regex	successes	rate	$\|S\|$
5	45	39	86 %	300
10	45	42	93 %	1000
15	10	7	70 %	1000

Table 2: Success rate on the target regular expressions and the sample size used per alphabet size.

It is also interesting to note that *i*DRegEx successfully derived the regular expression $r_1 = (\sigma_1\sigma_2 + \sigma_3 + \cdots + \sigma_n)^+$ of Theorem 3 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 be of size at least $(n - 2)!$, i.e., 720, 40320, and 3628800 respectively. The regular expression $r_2 = \Sigma^+\sigma_1\Sigma^+$, in contrast, was not derivable by *i*DRegEx from small samples.

Density(r)	#regex	successes	rate
$[0.0, 0.2[$	24	24	100 %
$[0.2, 0.4[$	22	18	82 %
$[0.4, 0.6[$	20	18	90 %
$[0.6, 0.8[$	22	21	95 %
$[0.8, 1.0]$	12	10	83 %

Table 3: Success rate on the target regular expressions, grouped by expression density.

κ	#regex	successes	rate
$[1.2, 1.4[$	29	28	96 %
$[1.4, 1.6[$	37	37	100 %
$[1.6, 1.8[$	24	22	91 %
$[1.8, 2.0[$	11	5	54 %
$[2.0, 2.5[$	12	5	41 %
$[2.5, 3.0]$	18	12	66 %

Table 4: Success rate on the target regular expressions, grouped by κ, the average number of states per symbol.

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 iDRegEx performs on such samples.

Definition 6. *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, iDRegEx needs to "guess" the edges for which there is no witness in S. This guessing capability is built into KoaToKore in the form of repair rules [10]. Our experiments show that for target expressions with alphabet size $|\Sigma| = 10$, this is highly effective: even at a coverage of 70%, half the target expressions can still be learned correctly as Table 5 shows. This clearly illustrates that iDRegEx is robust with respect to the quality of the samples.

coverage	successes	rate
1.0	25	100 %
0.9	16	64 %
0.8	15	60 %
0.7	13	52 %
0.6	0	0 %

Table 5: 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 are less promising as Table 6 illustrates. Only approximately 25% of the target regular expressions can be learned correctly given a partially covering sample. This is to be expected since the absolute amount of information missing for smaller regular expressions is larger than in the case of larger expressions.

coverage	successes	rate
1.0	12	100 %
0.9	3	25 %
0.8	2	16 %
0.7	1	8 %
0.6	1	8 %
0.5	0	0 %

Table 6: Success rate for 12 target expressions for $|\Sigma| = 5$ with partially covering samples.

5.3 Comparison to XTRACT

The xtract algorithm of Garofalakis et al. [31] is a regular expression learning system focusing on deriving general regular expressions. Its precision on 1-OREs has already been tested in [10, Table 1 and 2], where it was outperformed by our earlier algorithm i1-ORE. As shown in Section 5.1 and Theorem 5, iDRegEx is conservative over i1-ORE, and therefore also outperforms xtract over these expressions.

The expressions derived by xtract on non-1-OREs of alphabet size $|\Sigma| = 5$ with a sample of 300 words show an over-generalization to Σ^* whereas our algorithm derives the target regular expression flawlessly. For instance, for the target expression $(bcda + a + b + e)^+$ xtract returns $(a + b + c + d + e)^*$.

6. CONCLUSIONS

We presented the algorithm iDRegEx for inferring 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 iDRegEx on a large corpus of real-world and synthetic regular expressions of different densities.

Some questions need further attention. First, in our experiments, iDRegEx 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 κ 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 [25] rather than a straightforward one or to explore different methods for learning probabilistic automata.

Although iDRegEx can be directly plugged in into the XSD inference engine iXSD of [12], it would be interesting to investigate how to extend these techniques to the more robust class of Relax NG schemas [21].

7. REFERENCES

[1] Castor. `www.castor.org`.

[2] SUN Microsystems JAXB. `java.sun.com/webservices/jaxb`.

[3] P. Adriaans and P. Vitanyi. The Power and Perils of MDL, 2006.

[4] H. Ahonen. Generating Grammars for Structured Documents using Grammatical Inference Methods. Report A-1996-4, Dept. of Comp. Sci., Univ. of Finland, 1996.

[5] D. Angluin and C. Smith. Inductive Inference: Theory and Methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

[6] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 8(4):413–438, 2005.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath Satisfiability in the Presence of DTDs. In *PODS*, pages 25–36, 2005.

[8] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[9] G. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data, 2007. `http://www.uhasselt.be/~lucg5855/papers/infkore.pdf`

[10] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of Concise DTDs from XML Data. In *VLDB*, 2006.

[11] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: a Practical Study. In *WebDB*, pages 79–84, 2004.

[12] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML data. In *VLDB*, pages 998–1009, 2007.

[13] A. Brāzma. Efficient Identification of Regular Expressions from Representative Examples. In *COLT*, pages 236–242. ACM Press, 1993.

[14] A. Brüggeman-Klein. Regular Expressions into Finite Automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[15] A. Brüggemann-Klein and D. Wood. One-unambiguous Regular Languages. *Information and computation*, 140(2):229–253, 1998.

[16] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *ICDT*, pages 336–350, 1997.

[17] P. Caron and D. Ziadi. Characterization of Glushkov Automata. *Theo. Comp. Sc.*, 233(1–2):75–90, 2000.

[18] D. Che, K. Aberer, and M. T. Özsu. Query Optimization in XML Structured-Document Databases. *VLDB J.*, 15(3):263–289, 2006.

[19] B. Chidlovskii. Schema Extraction from XML: a Grammatical Inference Approach. In *KRDB*, 2001.

[20] J. Clark. Trang: Multi-format Schema Converter. `http://www.thaiopensource.com/relaxng/trang.html`.

[21] J. Clark and M. Murata. *RELAX NG Specification.* OASIS, December 2001.

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

[23] J. F. Fang Du, Sihem Amer-Yahia. ShreX: Managing XML Documents in Relational Databases. In *VLDB*, pages 1297–1300, 2004.

[24] H. Fernau. Algorithms for Learning Regular Expressions. In *ALT*, pages 297–311, 2005.

[25] R. Finn, J. Mistry, B. Schuster-Böckler, S. Griffiths-Jones, et al. Pfam: Clans, Web Tools and Services. *Nucleic Acids Research*, 34:D247–D251, 2006.

[26] D. Florescu. Managing Semi-structured Data. *ACM Queue*, 3(8), October 2005.

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

[28] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *SIGMOD*, pages 181–191, 2002.

[29] D. Freitag and A. McCallum. Information Extraction with HMM Structures Learned by Stochastic Optimization. In *AAAI/IAAI*, pages 584–589, 2000.

[30] P. Garcia and E. Vidal. Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.

[31] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data Mining and Knowledge Discovery*, 7:23–56, 2003.

[32] E. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, May 1967.

[33] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.

[34] J. Hegewald, F. Naumann, and M. Weis. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *ICDE Workshops*, page 81, 2006.

[35] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 2007.

[36] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB*, pages 228–239, 2004.

[37] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*, pages 241–250, 2001.

[38] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and Complexity of XML Schema. *ACM TODS*, 31(3), 2006.

[39] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: A First Study. In *WWW*, pages 500–510, 2003.

[40] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[41] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In *ICDM*, pages 295–306. 1998.

[42] F. Neven and T. Schwentick. On the Complexity of XPath Containment in the Presence of Disjunction, DTDs, and Variables. *Logical Methods in Computer Science*, 2(3), 2006.

[43] L. Pitt. Inductive Inference, DFAs, and Computational Complexity. In *AII*, pages 18–44, 1989.

[44] D. Quass, J. Widom, R. Goldman, et al. LORE: a Lightweight Object REpository for Semistructured Data. In *SIGMOD*, page 549, 1996.

[45] L. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE*, 77(2):257–286, 1989.

[46] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB J.*, 10(4):334–350, 2001.

[47] A. Sahuguet. Everything You Ever Wanted to Know about DTDs, but Were Afraid to Ask. In *WebDB*, 2000.

[48] Y. Sakakibara. Recent Advances of Grammatical Inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[49] J. Sankey and R. K. Wong. Structural Inference for Semistructured Data. In *CIKM*, pages 159–166. 2001.

[50] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures.* W3C, May 2001.

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