Learning Restricted Deterministic Regular Expressions with Counting

Xiaofan Wang[1,2] and Haiming Chen[1(✉)]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{wangxf,chm}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. Regular expressions are widely used in various fields. Learning regular expressions from sequence data is still a popular topic. Since many XML documents are not accompanied by a schema, or a valid schema, learning regular expressions from XML documents becomes an essential work. In this paper, we propose a restricted subclass of single-occurrence regular expressions with counting (RCsores) and give a learning algorithm of RCsores. First, we learn a single-occurrence regular expressions (SORE). Then, we construct an equivalent *countable finite automaton* (CFA). Next, the CFA runs on the given finite sample to obtain an updated CFA, which contains counting operators occurring in an RCsore. Finally we transform the updated CFA to an RCsore. Moreover, our algorithm can ensure the result is a *minimal* generalization (such generalization is called *descriptive*) of the given finite sample.

Keywords: Schema inference · Regular expressions · Counting · Descriptive generalization

1 Introduction

Regular expression are widely used in information extraction, network security, database management, programming languages, etc. Nowadays, mining potential knowledge from sequence data has become a common task in many research areas and application scenarios [9,20,24,27]. The technologies of learning regular expressions have also obtained more and more attention and development. For example, many XML documents are not accompanied by a schema, or a valid schema [1,4,5,23], learning regular expressions from XML documents will facilitate the diverse applications of XML Schema, such as data processing, automatic data integration, and static analysis of transformations [10,21,22]. In this paper, we focus on learning regular expressions from XML documents.

For any given positive data, Gold specified that the class of regular expressions cannot be learned [15]. Even Bex et al. claimed that the class of

Work supported by National Natural Science Foundation of China under Grant Nos. 61872339, 61472405.

R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 98–114, 2019.
https://doi.org/10.1007/978-3-030-34223-4_7

deterministic regular expressions cannot be learned [3]. Therefore, there are many works focusing on learning subclasses of deterministic regular expressions [2,3,6,7,11,12]. Deterministic regular expressions [8] require that each symbol in the input word can be unambiguously matched to a position in the regular expression without looking ahead in the word. Single-occurrence regular expressions (SOREs) [6,7] are classic subclass of deterministic regular expressions (standard). However, SOREs do not support counting, which is an extension of standard regular expressions used in XML Schema [14,16–19,25,26]. Then, we propose a restricted subclass of single-occurrence regular expressions with counting (RCsores). Our experiments (see Table 3) showed that the proportion of RCsores is 89.45% for 425,275 regular expressions extracted from XSD files, which were grabbed from Open Geospatial Consortium (OGC) XML Schema repository[1]. I.e., the majority of schemas in above real-world XSD files use RCsores. Therefore, it is necessary to study a learning algorithm for RCsore. Compared with Gold-style learning [15], the descriptive generalization [12,13] does not require to learn an exact representation of the target language, but can lead to a compact and powerful model [13]. Thus, our learning algorithm is based on the descriptive generalization [12,13].

For learning algorithms of SOREs, Bex et al. [7] proposed RWR and RWR^2_ℓ [7]. Freydenberger et al. [12] presented the learning algorithm *Soa*2*Sore* [12]. Additionally, [7] (resp. [12]) mentioned the future work, which is that SOREs extended with counting can be learnt by an additional post-processing step following the algorithm RWR (resp. *Soa*2*Sore*). However, the additional post-processing may result in the problem of overgeneralization [25]. For solving this problem, Wang et al. [25] proposed the class ECsores (see Definition 2), and the corresponding learning algorithm *InfECsore* [25]. However, although the ECsore learnt by *InfECsore* is descriptive of any given finite sample, the recall of *InfECsore* is lower[2]. Additionally, every possibly repeated subexpression of the ECsore can be extended with counting, then the algorithm *InfECsore* needs plenty of accurate counting such that it is not efficient to process larger samples. Wang et al. [26] also proposed a subclass cSOREs, which are a subclass of ECsore, and the corresponding learning algorithm *InfcSORE* [26], but the learnt cSORE is not descriptive of any given finite sample[3]. Therefore, we propose a new subclass RCsore and the corresponding method for learning RCsore. Although RCsores are also subclass of ECsores, for any given finite sample, our algorithm not only can ensure the learnt RCsore is descriptive of the given finite sample (w.r.t. the class of RCsores), but also can ensure that the recall for the expression derived by our algorithm can be higher than that for the expression learnt by

[1] http://schemas.opengis.net/.

[2] For instance, the original expression in XSD can be denoted by $r_0 = (a|b)^{[1,6]}$, given sample $\{ba, aa, abaa, aabaa\}$, the ECsore learnt by *InfECsore* is $r_1 = (b?a^{[1,2]})^{[1,2]}$. However, the learnt RCsore can be $r_2 = (b?a)^{[1,4]}$. Let $S_1 = \{s | s \in \mathcal{L}(r_0), s \in \mathcal{L}(r_1)\}$ and $S_2 = \{s | s \in \mathcal{L}(r_0), s \in \mathcal{L}(r_2)\}$. Then, $|S_1| = 14$ and $|S_2| = 25$. Thus, $\frac{|S_1|}{|\mathcal{L}(r_0)|} < \frac{|S_2|}{|\mathcal{L}(r_0)|}$.

[3] Let $S = \{b, abd, ad, cddcdd\}$, the cSORE learnt by *InfcSORE* is $r_3 = ((a?b?|c)d?)^{[1,4]}$, however, there is a cSORE $r_4 = (a?b?|c?(d^{[1,2]})?)^{[1,2]}$ such that $\mathcal{L}(r_3) \supset \mathcal{L}(r_4) \supseteq S$.

InfECsore. Moreover, for a smaller sample, the learnt RCsore has better generalization ability (higher precision and recall) than the learnt ECsore. And the learning algorithm of RCsore is more efficient than that of ECsore for processing larger samples.

The main contributions of this paper are as follows.

- We infer a SORE and construct an equivalent countable finite automaton (CFA) [25].
- The CFA runs on the given finite sample to obtain an updated CFA, which has updated the counting operators that will occur in an RCsore.
- We convert the updated CFA to an RCsore and prove that the generated RCsore is descriptive of any given finite language.

The paper is structured as follows. Section 2 gives the basic definitions. Section 3 presents the learning algorithm of the RCsore, and proves the RCsore generated by our algorithm is descriptive of any given finite language. Section 4 presents experiments. Section 5 concludes the paper.

2 Preliminaries

2.1 Regular Expression with Counting

Let Σ be a finite alphabet of symbols. $\mathcal{R}_c$ is a set (non-empty) of regular expressions with counting over Σ. ε, $a \in \Sigma$ are regular expressions in $\mathcal{R}_c$. For regular expressions $r_1, r_2 \in \mathcal{R}_c$, the disjunction $(r_1|r_2)$, the concatenate $(r_1 \cdot r_2)$, the Kleene-star r_1^*, and counting (*numerical occurrence constraints* [14]) $r_1^{[m,n]}$ are also regular expressions in $\mathcal{R}_c$. $m \in \mathbb{N}$, $n \in \mathbb{N}_{/1}$, $\mathbb{N} = \{1, 2, 3, \cdots\}$, $\mathbb{N}_{/1} = \{2, 3, 4, ...\} \cup \{+\infty\}$, and $m \leq n$. For a regular expression $r \in \mathcal{R}_c$, $\mathcal{L}(r^{[m,n]}) = \{w_1 \cdots w_i | w_1, \cdots, w_i \in \mathcal{L}(r), m \leq i \leq n\}$. Note that r^+, $r?$, and r^* are used as abbreviations of $r^{[1,+\infty]}$, $r|\varepsilon$, and $r^{[1,+\infty]}|\varepsilon$, respectively. Usually, we omit concatenation operators in examples. $|r|$ denotes the length of r, which is the number of symbols and operators occurring in r plus the sizes of the binary representations of the integers [14]. For a finite sample S, $|S|$ denotes the number of strings in S. $\varnothing$ denotes the empty set. For space consideration, all omitted proofs can be found at http://github.com/GraceFun/InfRCsore.

2.2 SORE, ECsore and RCsore

SORE is defined as follows.

Definition 1 (SORE [6,7]). *Let Σ be a finite alphabet. A single-occurrence regular expression (SORE) is a standard regular expression over Σ in which every terminal symbol occurs at most once.*

Example 1. $(ab)^+$ is a SORE, while $(ab)^+a$ is not.

Definition 2 (ECsore [25]). *Let Σ be a finite alphabet. An ECsore is a regular expression with counting over Σ in which every terminal symbol occurs at most once. For a regular expression r, an ECsore forbids immediately nested counters, expressions of form $(r?)?$ and $(r?)^{[m,n]}$.*

ECsore does not use the Kleene-star and the iteration operations. And ECsores are deterministic by definition.

Definition 3 (RCsore). *Let Σ be a finite alphabet. An RCsore is an ECsore over Σ. For regular expressions r_1, r_2 and r_3, an RCsore forbids expressions of form $(r_1r_2r_3)^{[m_1,n_1]}$ where $\varepsilon \in \mathcal{L}(r_1)$, $\varepsilon \in \mathcal{L}(r_3)$ and $r_2 \in \{e^{[m_2,n_2]}, e?\}$ for regular expression e ($\varepsilon \notin \mathcal{L}(e)$).*

According to the definition, RCsores are a subclass of ECsores. ECsores are deterministic regular expressions, so are the RCsores.

Example 2. $(a|b^{[1,2]})^{[3,4]}(c?d)^{[1,+\infty]}$, $(a^{[3,4]}b)^{[1,2]}$, and $((a?b?|c)(d^{[2,3]})?)^{[1,2]}$ are RCsores, also ECsores, while $a?b^+a$ is not a SORE, therefore neither an RCsore nor an ECsore. However, the expressions $(a?b^{[1,2]}c?)^{[1,2]}$ and $(a?b?c?)^{[1,2]}$ are ECsores, not RCsores. $(a^{[1,2]})^{[1,2]}$, $((a^{[1,2]})?)^{[1,2]}$ and $((a^{[1,2]})?)?$ are forbidden.

2.3 Descriptivity

We give the notion of descriptive expressions and automata.

Definition 4 (Descriptivity [12]). *Let $\mathcal{D}$ be a class of regular expressions or finite automata over some alphabet Σ. A $\delta \in \mathcal{D}$ is called $\mathcal{D}$-descriptive of a non-empty language $S \subseteq \Sigma^*$ if $\mathcal{L}(\delta) \supseteq S$, and there is no $\gamma \in \mathcal{D}$ such that $\mathcal{L}(\delta) \supset \mathcal{L}(\gamma) \supseteq S$.*

If a class $\mathcal{D}$ is clear from the context, we simply write *descriptive* instead of $\mathcal{D}$-descriptive.

Proposition 1. *Let Σ be a finite alphabet. There exists an RCsore-descriptive RCsore r for every language $\mathcal{L} \subseteq \Sigma^*$.*

2.4 Countable Finite Automaton

Definition 5 (Countable Finite Automaton [25]). *A Countable Finite Automaton (CFA) is a tuple $(Q, Q_c, \Sigma, \mathcal{C}, q_0, q_f, \Phi, \mathsf{U}, \mathsf{L})$. The members of the tuple are described as follows:*

- *Σ is a finite and non-empty alphabet.*
- *q_0 and q_f : q_0 is the initial state, q_f is the unique final state.*
- *Q is a finite set of states. $Q = \Sigma \cup \{q_0, q_f\} \cup \{+_i\}_{i\in\mathbb{N}}$.*
- *$Q_c \subset Q$ is a finite set of counter states. Counter state is a state q ($q \in \Sigma$) that can directly transit to itself, or a state $+_i$. For each subexpression (excluding single symbol $a \in \Sigma$) under the iteration operator, we associate a unique counter state $+_i$ to count the minimum and maximum number of repetitions of the subexpression, respectively.*

- *$\mathcal{C}$ is finite set of counter variables that are used for counting the number of repetitions of the subexpressions under the iteration operators. $\mathcal{C} = \{c_q | q \in Q_c\}$, for each counter state q, we also associate a counter variable c_q.*
- $\mathsf{U} = \{u(q) | q \in Q_c\}$, $\mathsf{L} = \{l(q) | q \in Q_c\}$. *For each subexpression under the iteration operator, we associate a unique counter state q such that $l(q)$ and $u(q)$ are the minimum and maximum number of repetitions of the subexpression, respectively.*
- *Φ maps each state $q \in Q$ to a set of tuples consisting of a state $p \in Q$ and two update instructions. Φ: $Q \mapsto \wp(Q \times ((\mathsf{L} \times \mathsf{U} \mapsto (\mathbf{Min}(\mathsf{L} \times \mathcal{C}), \mathbf{Max}(\mathsf{U} \times \mathcal{C}))) \cup \{\emptyset\}) \times ((\mathcal{C} \mapsto \{\mathbf{res}, \mathbf{inc}\}) \cup \{\emptyset\}))$. ($\emptyset$ denotes empty instruction.)*

Definition 6 (Transition Function of a CFA [25]). *The transition function δ of a CFA $(Q, Q_c, \Sigma, \mathcal{C}, q_0, q_f, \Phi, \mathsf{U}, \mathsf{L})$ is defined for any configuration (q, γ, θ) and the letter $y \in \Sigma \cup \{\dashv\}$*

(1) $y \in \Sigma$: $\delta((q, \gamma, \theta), y) = \{(z, f_\alpha(\gamma, \theta), g_\beta(\theta)) | (z, \alpha, \beta) \in \Phi(q) \wedge (z = y \vee ((y, \alpha, \beta) \notin \Phi(q) \wedge z \in \{+_i\}_{i \in \mathbb{N}}))\}$.
(2) $y = \dashv$: $\delta((q, \gamma, \theta), \dashv) = \{(z, f_\alpha(\gamma, \theta), g_\beta(\theta)) | (z, \alpha, \beta) \in \Phi(q) \wedge (z = q_f \vee z \in \{+_i\}_{i \in \mathbb{N}})\}$.

3 Inference of RCsores

Our learning algorithm works in the following steps.

(1) We infer a SORE for a given finite sample. (2) A CFA is equivalently transformed from the SORE obtained from (1). (3) The CFA transformed from step (2) runs on the same finite sample used in step (1) to obtain an updated CFA, which has updated the counting operators that will occur in an RCsore. (4) We convert the updated CFA in step (3) to an RCsore.

Algorithm 1. *InfRCsore*

Input: a finite sample S;
Output: an RCsore-descriptive RCsore;
1: A SORE r_s =*InfSore*($SOA(S)$);
2: CFA $\mathcal{A}$ = *ConsCFA*(r_s);
3: CFA $\mathcal{A}'$=*Counting*($\mathcal{A}, S$);
4: $r = GenRCsore(A')$;
5: **return** r;

Algorithm 1 is the framework of our learning algorithm. Algorithm *SOA* [12] constructs the single-occurrence automaton (SOA) [7,12] for the given finite sample S. Algorithm *InfSore* is described in Sect. 3.1, algorithm *ConsCFA* is given in Sect. 3.2, algorithm *Counting* is showed in [25], algorithm *GenRCsore* is presented in Sect. 3.4.

3.1 Inferring Standard Deterministic Regular Expression: SORE

The problem of learning SORE was solved by Bex et al. and Freydenberger et al. Bex et al. proposed the learning algorithm RWR [7] and its variants. Freydenberger et al. [12] proved the results of RWR with its variants are not descriptive of any given finite sample, and then presented the learning algorithm *Soa2Sore* [12]. However, the SORE learnt by *Soa2Sore* is descriptive of the language, which is the set of the strings accepted by the SOA that is built for the given finite sample [12]. Despite of that, we still can infer a SORE such that an RCsore, which is descriptive of the given finite sample, can be derived from the obtained SORE.

Algorithm 2 learns a SORE from the given finite sample. First, a SORE is inferred by *Soa*2*Sore*. Then, the SORE is converted to a normal form (SORE). Theorem 1 demonstrates that the normal form is more approximate to the given finite sample than the SORE learnt by *Soa*2*Sore*.

Algorithm 2. *InfSore*

Input: a finite sample S;
Output: a SORE r_s;
1: A SORE $r_0 = Soa2Sore(SOA(S))$;
2: Let $r_{f_1} = (r_1? \cdots r_k?)^+$ $(k \geq 2)$;//r_i $(1 \leq i \leq k)$ is a regular expression
3: Let $r_{f_2} = (r_1|\cdots|r_k)^+$ where $r_i \in \{e_i^+, e_i\}$ $(k \geq i \geq 1)$;//e_i is a regular expression
4: Let $r_{f_3} = (r_1 r_2^+ r_3)^+$ where $\varepsilon \in \mathcal{L}(r_1)$ and $\varepsilon \in \mathcal{L}(r_3)$;
5: **if** Case (1): r_0 contains the expression of the form r_{f_1} **then**
6:     for all expressions of form r_{f_1}: r_{f_1} is converted to $r'_{f_1} = (r_1|\cdots|r_k)^+$;
7: **if** Case (2): r_0 contains the expression of the form r_{f_2}, where $r_i = e_i$ **then**
8:     for all expressions of form r_{f_2}: r_{f_2} is converted to $r'_{f_2} = (e_1^+|\cdots|e_k^+)^+$;
9: **if** Case (3): r_0 contains the expression of the form r_{f_3} **then**
10:     for all expressions of form r_{f_3}: r_{f_3} is converted to $r'_{f_3} = (r_1 r_2 r_3)^+$;
11: Let $r_s = r_0$; **return** r_s;

In Algorithm 2, if the SORE r_0 does not contain any one expression of the forms r_{f_1}, r_{f_2} and r_{f_3} (which are specified in lines 4, 2 and 3, respectively), then *InfSore* directly outputs r_0, i.e., $r_s = r_0$. Note that, except for case (1) (in line 5), other cases are equivalent conversions for r_0. The conversion in case (2) (in line 7) is mainly used to easily construct a CFA in the next section and track as many subexpressions as possible (which can be repeated) in a SORE. For processing r_0 to a normal form r_s, it takes $\mathcal{O}(|r_0|)$ time. Let the built SOA in line 1 contain n_s nodes and t_s transitions. *Soa*2*Sore* takes $\mathcal{O}(n_s t_s)$ time to infer a SORE. Thus, the time complexity of algorithm *InfSore* is $\mathcal{O}(n_s t_s)$ $(n_s t_s > |r_0|)$.

Example 3. For sample $S = \{a, acc, acbb, bab\}$, the result of algorithm *Soa*2*Sore* is $r_0 = ((a(c^+)?)|b)^+$. Let the SORE r_s :=*InfSore*($SOA(S)$), then the SORE $r_s = ((a(c^+)?)^+|b^+)^+$.

Theorem 1. *For any given finite sample S, let $r_0 = Soa2Sore(SOA(S))$, and let r_s :=$InfSore(SOA(S))$, then $\mathcal{L}(r_0) \supseteq \mathcal{L}(r_s) \supseteq S$.*

According to Theorem 1, $\mathcal{L}(r_s)$ is more approximate to the given finite sample than $\mathcal{L}(r_0)$. Therefore, we can obtain a descriptive RCsore, which is extended from the expression of form r_s.

3.2 Translating SORE to CFA

To avoid plenty of accurate counting in a CFA, the CFA should be constructed from a specific structure, instead of being learnt from a given finite sample [25]. Therefore, in this section, we present how to translate a SORE to a CFA. First, we construct the state-transition diagram of a CFA by traversing the syntax tree of the SORE, which is obtained from Sect. 3.1. Then, the detailed descriptions of the CFA are similar with that described in [25]. Theorem 2 shows that an equivalent CFA can be transformed from an RCsore.

Algorithm 3 first constructs the state-transition diagram of a CFA by using Algorithm 4, then presents the detailed descriptions of the CFA. The state-transition diagram of a CFA is a finite directed graph, denoted by G. Algorithm 4 constructs a directed graph G by traversing a syntax tree. The entire process is similar to the preorder traversal of the binary tree. For a syntax tree T, $T.L$ and $T.R$ denote the left subtree and the right subtree of T, respectively. For a graph G, $G.\prec(v)$ denotes the set of all immediate predecessors of v in G, $G.\succ(v)$ denotes the set of all immediate successors of v in G. Some subroutines in Algorithm 4 are as follows.

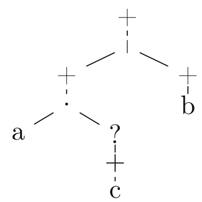


Fig. 1. The syntax tree of expression $((a(c^+)?)^+|b^+)^+$.

$Conn_G(t, G_1, G_2)$. According to label t, a new graph G is constructed by connecting graphs G_1 and G_2. If t=‘·’, then add edges $\{(v_1, v_2)|v_1 \in G_1.\prec(q_f), v_2 \in G_2.\succ(q_0)\}$; remove nodes $G_1.q_f$, $G_2.q_0$ and their associated edges; let $G.q_0 = G_1.q_0$. If t=‘|’, then add new nodes q_0, q_f; add edges $\{(q_0, v_1)|v_1 \in G_1.\succ(q_0) \cup G_2.\succ(q_0)\}$, and $\{(v_2, q_f)|v_2 \in G_1.\prec(q_f) \cup G_2.\prec(q_f)\}$; remove nodes $G_1.q_0$, $G_1.q_f$, $G_2.q_0$, $G_2.q_f$ and their associated edges; let $G.q_0 = q_0$.

Algorithm 3. *ConsCFA*

Input: a syntax tree T;
Output: a CFA $\mathcal{A}$;
1: $G = Cons_G(T)$;
2: CFA $\mathcal{A} = (Q, Q_c, \Sigma, \mathcal{C}, G.q_0, G.q_f, \Phi(\mathcal{R}), \mathsf{U}, \mathsf{L})$;
3: **return** $\mathcal{A}$;

$Add^{+}(G, +_i)$. G is a graph, and $+_i$ (a counter state in CFA) is a node. Add^{+} adds node $+_i$ (initially, $i = 1$) into the graph G. Add new node q_f; let $\mathcal{R}_{+_i} = \{v | v \in G. \succ (q_0)\}$; add edges $\{(+_i, v_1) | v_1 \in G. \succ (q_0)\}$; add edges $\{(v_2, +_i) | v_2 \in G. \prec (q_f)\}$; remove node $G.q_f$ and its associated edges; add edge $(+_i, q_f)$. The set of $\mathcal{R}_{+_i}$ is established to specify the transition entrances for state $+_i$ to count the minimum and maximum number of repetitions of the subexpression under the iteration operator. Each $\mathcal{R}_{+_i}$ is a global variable. Let $\mathcal{R} = \{\mathcal{R}_{+_i}\}_{i \in \mathbb{N}}$.

Algorithm 4. $Cons_G$

```
Input: a syntax tree T;
Output: a directed graph G(V, E);
if T = ∅ return ∅;
if T.label ∈ Σ then
    Add new nodes q = T.label, q_0, and q_f;
    return G({q_0, q, q_f}, {(q_0, q), (q, q_f)});
if T.label =‘·’ then
    G_1 = Cons_G(T.L); G_2 = Cons_G(T.R);
    return Conn_G(T.label, G_1, G_2);
if T.label ∈ {+, ?} then
    G = Cons_G(T.L);
    if T.label = ‘+’ then
        if T.L.label ∈ Σ then
            add edge (G.T.L.label, G.T.L.label);
        else G = Add^+(G, +_i); inc i;
    if T.label =‘?’ then
        add edge (G.q_0, G.q_f);
    return G
if T.label =‘|’ then
    G_1 = Cons_G(T.L); G_2 = Cons_G(T.R);
    G = Conn_G(T.label, G_1, G_2);
    return G;
```

In Algorithm 3, after the state-transition diagram G of a CFA is constructed, the CFA $\mathcal{A}$ is then obtained. In line 2, [25] shows the detailed descriptions of the CFA $\mathcal{A}$. Note that, $\Phi(\mathcal{R})$ denotes that $\mathcal{R}$ is a parameter in Φ.

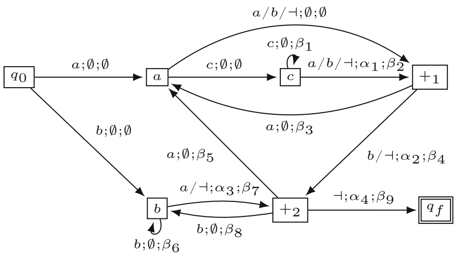
For any SORE r obtained in Sect. 3.1, the time complexity of constructing the corresponding syntax tree is $\mathcal{O}(|r|)$, and the preorder traversal of the syntax tree used to construct the state-transition diagram of a CFA also requires $\mathcal{O}(|r|)$ time. Therefore, the time complexity of constructing a CFA is $\mathcal{O}(|r|)$.

Example 4. For the expression $((a(c^{+})?)^{+}|b^{+})^{+}$, the syntax tree can be seen in Fig. 1. The corresponding state-transition diagram can be seen in Fig. 2(a).

Theorem 2. *For any given SORE r, there is a CFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(r)$.*

3.3 Counting with CFA

The constructed CFA in Sect. 3.2 runs on the given finite sample, which is the same set of strings used to generate the SORE in Sect. 3.1. The CFA counts the minimum and maximum number of repetitions of the subexpressions under



(a) CFA $\mathcal{A}$

$\alpha_1 : (l(c), u(c)) \mapsto (\mathbf{Min}(l(c), c_c), \mathbf{Max}(u(c), c_c))$; $\alpha_2 : (l(+_1), u(+_1)) \mapsto (\mathbf{Min}(l(+_1), c_{+_1}), \mathbf{Max}(u(+_1), c_{+_1}))$. $\alpha_3 : (l(b), u(b)) \mapsto (\mathbf{Min}(l(b), c_b), \mathbf{Max}(u(b), c_b))$; $\alpha_4 : (l(+_2), u(+_2)) \mapsto (\mathbf{Min}(l(+_2), c_{+_2}), \mathbf{Max}(u(+_2), c_{+_2}))$;
$\beta_1 : c_c \mapsto \mathbf{inc}$; $\beta_2 : c_c \mapsto \mathbf{res}$; $\beta_3 : c_{+_1} \mapsto \mathbf{inc}$;
$\beta_4 : c_{+_1} \mapsto \mathbf{res}$; $\beta_5, \beta_8 : c_{+_2} \mapsto \mathbf{inc}$; $\beta_6 : c_b \mapsto \mathbf{inc}$;
$\beta_7 : c_b \mapsto \mathbf{res}$; $\beta_9 : c_{+_2} \mapsto \mathbf{res}$.

(b) Update instructions

Fig. 2. (a) is the CFA $\mathcal{A}$ for regular language $\mathcal{L}(((a(c^+)?)^+|b^+)^+)$. The label of the transition edge is $(y;\alpha_i;\beta_j)$ $(i,j \in \mathbb{N})$, y $(y \in \Sigma \cup \{\dashv\})$ is a current letter; (b) specifies that, α_i is an update instruction for the lower bound and upper bound variables, and β_j is an update instruction for the counter variable.

the iteration operators. Counting rules are given by transition functions of the CFA. We use the algorithm *Counting* proposed in [25] to run the CFA. Let $\mathcal{A}$ denote the constructed CFA and S denote the given finite sample. After the CFA $\mathcal{A}$ recognized the sample S, let $\mathcal{A}'$ denote the CFA $\mathcal{A}$ which has updated the the minimum and maximum number of repetitions of the subexpressions under the iteration operators. Let $\mathcal{A}' = Counting(\mathcal{A}, S)$, and $\mathsf{C} = \{(l(q), u(q))|l(q) = \mathcal{A}'.\mathsf{L}.l(q), u(q) = \mathcal{A}'.\mathsf{U}.u(q), q \in \mathcal{A}'.Q_c\}$. The elements in C are counting operators, which will be introduced into an RCsore. The time complexity of *Counting* is $\mathcal{O}(N\overline{L})$ time, where $N = |S|$ and $\overline{L}$ is the average length of the strings in S [25].

Example 5. For the sample $S = \{a, acc, acbb, bab\}$, $r_s = ((a(c^+)?)^+|b^+)^+$ is the SORE obtained from Sect. 3.1, the CFA $\mathcal{A}$ showed in Fig. 2 runs on the sample S. Then, the tuples in C are listed as follows: $(l(c), u(c)) = (1, 2)$, $(l(b), u(b)) = (1, 1)$[4], $(l(+_1), u(+_1)) = (1, 1)$, $(l(+_2), u(+_2)) = (1, 3)$. $l(+_1)$ and $u(+_1)$ (resp. $l(+_2)$ and $u(+_2)$) are the minimum and maximum number of repetitions of the subexpression $(a(c^+)?)$ (resp. $(a(c^+)?|b)$), respectively. Note that the minimum numbers of repetitions of symbol c are both 0 in strings a and bab. In Sect. 3.4, we will convert expression $c^{[1,2]}$ to $(c^{[1,2]})?$.

3.4 Generating RCsore

In this section, we transform the updated CFA $\mathcal{A}'$ obtained in Sect. 3.3 to an RCsore. Since the algorithm *GenECsore* can convert a CFA to an descriptive ECsore (w.r.t. the class of ECsores). We still can use the algorithm *GenECsore*

[4] Note that, the CFA $\mathcal{A}$ runs on S, the direct counting result for b is $(l(b), u(b)) = (1, 2)$. However, $(l(b), u(b))$ is subsequently updated by *Counting* that b can be repeated by using the counting operator $[l(+_2), u(+_2)] = [1, 3]$.

to derive an RCsore, the constructed CFA in this paper is equivalent to an RCsore, not an equivalent representation of an ECsore. Then, for an updated CFA $\mathcal{A}'$, the algorithm *GenECsore* can convert the CFA $\mathcal{A}'$ to an descriptive RCsore (w.r.t. the class of RCsores).

Algorithm 5 converts the updated CFA to an RCsore. Theorem 3 demonstrates the finally obtained RCsore is descriptive of any given finite sample. Assume that the updated CFA contains n_c nodes and t_c transitions. *GenECsore* takes $\mathcal{O}(n_c t_c)$ time to infer an ECsore [25]. Then, the time complexity of generating RCsore is $\mathcal{O}(n_c t_c)$.

Algorithm 5. *GenRCsore*

Input: the updated CFA $\mathcal{A}'$
Output: an RCsore r;
1: $r =$*GenECsore*$(\mathcal{A}')$;
2: **return** r;

Example 6. The tuples in C obtained from algorithm *Counting* are as follows. $(l(c), u(c)) = (1, 2)$, $(l(b), u(b)) = (1, 1)$, $(l(+_1), u(+_1)) = (1, 1)$ and $(l(+_2), u(+_2)) = (1, 3)$. For the updated CFA $\mathcal{A}'$, the generated RCsore is $((a(c^{[1,2]})?)|b)^{[1,3]}$.

Theorem 3. *For any given finite language S, let $r :=$InfRCsore(S), the time complexity of algorithm InfRCsore is $\mathcal{O}(n_c t_c + N\overline{L})$ and r is an RCsore-descriptive RCsore for S.*

Let $\mathcal{A}_c$ and $\mathcal{A}_g$ denote the CFAs constructed in this paper and in literature [25], respectively. Assume that the CFA $\mathcal{A}_g$ contains n_g nodes and t_g transitions. The time complexity of *InfECsore* is $\mathcal{O}(n_g t_g + N\overline{L})$ [25]. $\mathcal{A}_c$ and $\mathcal{A}_g$ are equivalent representations of RCsore and ECsore, respectively. The CFA $\mathcal{A}_g$ can contain more nodes labeled $+_i$ ($i \in \mathbb{N}$) than the CFA $\mathcal{A}_c$. And the transitions in $\mathcal{A}_g$ can be also more than that in $\mathcal{A}_c$. Thus, $n_c t_c \leq n_g t_g$.

4 Experiments

In this section, we validate our algorithm on real-world XML data and generated XML data. We also provide evaluations of our algorithm in terms of generalization ability and time performance.

4.1 Data and Experiments

Table 3 demonstrates the practicability of RCsores, then we evaluate our algorithm on XML data. We obtained XML documents (*dblp-2018-04-01.xml*) conforming to DTD from DBLP Computer Science Bibliography corpus[5], from which we extracted the elements: inproc(eedings), article, phdth(esis), incolle(ction), and procee(dings). We obtained XML documents conforming to

[5] http://dblp.org/xml/release/.

XSD form Mondial corpus[6], from which the elements count(ry), provin (ce) and city are extracted. In order to validate on diverse XSDs, a number of real-world XSDs listed in Table 2 are searched from Google. However, we do not find the corresponding XML data, so we randomly generated them by using ToXgene[7]. The samples employed in the experiments are available at http://github.com/GraceFun/InfRCsore.

Table 1 lists the results of the learning algorithms *Soa2Sore*, *InfECsore* and *InfRCsore* on real-world XML data. Note that, based on descriptive generalization, *Soa2Sore* is the first algorithm being used to infer a SORE [12], and *InfECsore* is the algorithm being applied to learn a most practical subclass of deterministic regular expressions with counting: ECsore [25]. For each of the elements inproc(eedings), article and procee(dings), the corresponding expression learnt by *InfRCsore* is not only more precise than the corresponding expression in original DTD, but also more precise than the corresponding expression computed by *Soa2Sore*. Also, the result of *InfRCsore* is more general than the result of *InfECsore*, such that the learnt RCsore covers more XML data satisfying the corresponding original DTD than the learnt ECsore. For phdth(esis) and incolle(ction), the learnt RCsores are identical to the corresponding expressions computed by *InfECsore*. For each of elements count(ry), provin(ce) and city, the result of *InfRCsore* and the result of *InfECsore* are the same, and the corresponding RCsore and ECsore both are more precise than the corresponding expression generated by *Soa2Sore* and the corresponding expression in original XSD.

Table 2 lists a number of the expressions extracted from real-world XSDs and the results of the learning algorithms *Soa2Sore*, *InfECsore* and *InfRCsore* on generated XML data. For ep1, the learnt RCsore is identical to the learnt ECsore, they both indicate that more symbols or subexpressions can have numerical occurrence constraints, but are allowed to occur more times by the nested counters. For ep2, the learnt RCsore is identical to the learnt ECsore, they both are identical to the corresponding original XSD. This implies the original XSDs such as shown by ep2 could be precisely learnt by *InfRCsore* and *InfECsore*. For ep3 and ep4, although the learnt RCsores forbid the expressions learnt by *InfECsore*, which are more precise than the corresponding original XSD, even are identical to the corresponding original XSD for ep3, the learnt RCsores are more general than the learnt ECsores. Especially, for ep4, the learnt RCsore covers more XML data satisfying the corresponding original XSD than the learnt ECsore. For ep5, the learnt RCsore has the same higher nesting depth of counting operators with the learnt ECsore.

[6] http://www.dbis.informatik.uni-goettingen.de/Mondial/#XML.
[7] http://www.cs.toronto.edu/tox/toxgene/.

Table 1. Results of *Soa2Sore*, *InfECsore* and *InfRCsore* on real-world XML data. The left column gives element names, sample size for *Soa2Sore*, *InfECsore* and *InfRCsore*, respectively. The right column lists original DTD/XSD, the results of *Soa2Sore*, the results of *InfECsore* and the results of *InfRCsore*, respectively.

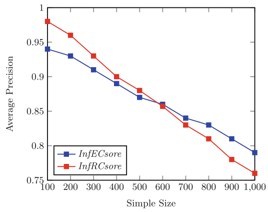
Element **Sample size**	Original segment of DTD/XSD Result of *Soa2Sore* Result of *InfECsore* Result of *InfRCsore*
inproc.	$(a\mid b\mid \cdots \mid v)^*$
2153167	$(b^*(ck?)?(r\mid a\mid m)?(o\mid (dj?)\mid f\mid n\mid q\mid e\mid l)^*)^+$
2153167	$(b?(ck?)?((r\mid a^{[1,45]}\mid m^{[1,3]})^{[1,3]})?((o^{[1,87]}\mid (dj?)\mid f\mid n\mid q\mid e\mid l)^{[1,6]})?)^{[1,5]}$
2153167	$((b\mid (ck?)\mid r\mid a^{[1,34]}\mid m^{[1,3]}\mid o^{[1,51]}\mid (dj?)\mid f\mid n^{[1,2]}\mid q\mid e\mid l\mid)^{[5,11]})?$
article	$(a\mid b\mid \cdots \mid v)^*$
1796920	$(b^*(((a^*(c\mid e)?)\mid m\mid n\mid q)(((j\mid ((f\mid r)d?)\mid h\mid i)k?)\mid p\mid l)^*o^*)^+)$
1796920	$((b^{[1,5]})?((((a^{[1,69]})?(c\mid e)?)^{[1,2]}\mid m\mid n\mid q)^{[1,3]}((((j\mid ((f\mid r)d?)\mid h\mid i)k?)^{[1,3]}\mid p\mid l)^{[1,3]})?$ $(o^{[1,116]})?)^{[1,3]})$
1796920	$(((b^{[1,5]})?(a^{[1,69]}\mid c\mid e\mid q\mid m^{[1,2]}\mid n\mid ((j\mid ((f\mid r)d?)\mid h\mid i)k?)^{[1,4]}\mid p\mid l\mid o^{[1,116]})^{[1,9]})?$
phdth.	$(a\mid b\mid \cdots \mid v)^*$
64943	$(a^*c((((p\mid (fk?)\mid u)t?j?)\mid e)(i\mid l\mid m\mid s)^*)^+q?)$
64943	$((a^{[1,3]})?c((e\mid ((u\mid (fk?)\mid p)t?j?))((s^{[1,3]}\mid m^{[1,5]}\mid l\mid i)^{[1,3]})?)^{[1,5]}q?)$
64943	$((a^{[1,3]})?c((e\mid ((u\mid (fk?)\mid p)t?j?))((s^{[1,3]}\mid m^{[1,5]}\mid l\mid i)^{[1,3]})?)^{[1,5]}q?)$
procee.	$(a\mid b\mid \cdots \mid v)^*$
58959	$(((a?(b\mid c))^+h?)?(i\mid s\mid d)?(j\mid q\mid l\mid (fr?)\mid t\mid e\mid (pg?)\mid m)^*)^*$
58959	$((((a?(b\mid c))^{[1,32]}h?)?((i\mid s^{[1,2]}\mid d)^{[1,2]})?((j\mid q\mid l\mid (fr?)\mid t\mid e\mid (pg?)\mid m^{[1,3]})^{[1,5]})?)^{[1,4]})?$
58959	$(((a?(b\mid c))^{[1,32]}h?)\mid i\mid s^{[1,2]}\mid d\mid j\mid q\mid l\mid (fr?)\mid t\mid e\mid (pg?)^{[1,2]}\mid (m^{[1,2]})^{[3,9]})?$
incolle.	$(a\mid b\mid \cdots \mid v)^*$
46750	$(a^*c((d(j\mid p)?)\mid f\mid r\mid (ev?)\mid l\mid m)^*(o^+\mid n\mid q)?)$
46750	$((a^{[1,49]})?c(((d(j\mid p)?)\mid f\mid r\mid (ev?)\mid l\mid m)^{[3,6]})?(o^{[2,104]}\mid n\mid q)?)$
46750	$((a^{[1,49]})?c(((d(j\mid p)?)\mid f\mid r\mid (ev?)\mid l\mid m)^{[3,6]})?(o^{[2,104]}\mid n\mid q)?)$
count.	$(a^+b?c^*d?\cdots k?(l?\mid m?)n?o^+p^*\cdots s^*(t^*\mid u^*))$
244	$(ab?c^+(de?)?(f(g(hi)?)?j?k?)?(m?\mid l)n?o^+p^*\cdots t^*u^+)$
244	$(ab?c^{[1,25]}(de?)?(f(g(hi)?)?j?k?)?(l\mid m?)n?o^{[1,2]}(p^{[1,12]})?(q^{[1,8]})?(r^{[1,8]})?(s^{[1,16]})?$ $(t^{[1,2]})?u^{[1,306]})$
244	$(ab?c^{[1,25]}(de?)?(f(g(hi)?)?j?k?)?(l\mid m?)n?o^{[1,2]}(p^{[1,12]})?(q^{[1,8]})?(r^{[1,8]})?(s^{[1,16]})?$ $(t^{[1,2]})?u^{[1,306]})$
provin.	$(a^+b?c?d^*e^*)$
1443	$(a^+b?c?d^*e^*)$
1443	$(a^{[1,4]}b?c?(d^{[1,6]})?(e^{[1,5]})?)$
1443	$(a^{[1,4]}b?c?(d^{[1,6]})?(e^{[1,5]})?)$
city	$(a^+b?c?d?e?f^*g^*h^*)$
3383	$(a^+b?(cde?)?f^*g^*h^*)$
3383	$(a^{[1,5]}b?(cde?)?(f^{[1,10]})?(g^{[1,4]})?(h^{[1,3]})?)$
3383	$(a^{[1,5]}b?(cde?)?(f^{[1,10]})?(g^{[1,4]})?(h^{[1,3]})?)$

Table 2. Results of *Soa2Sore*, *InfECsore* and *InfRCsore* on generated XML data.

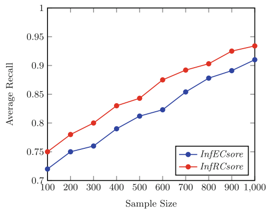
Element / **Sample size**	Original segment of XSD / Result of *Soa2Sore* / Result of *InfECsore* / Result of *InfRCsore*
ep1	$((a\|b\|c\|d\|e\|f)^{[1,10]})?$
941	$(a\|b\|c\|d\|e\|f)^{+}$
941	$(a^{[1,3]}\|b^{[1,4]}\|c^{[1,3]}\|d^{[1,4]}\|e^{[1,3]}\|f^{[1,4]})^{[2,6]}$
941	$(a^{[1,3]}\|b^{[1,4]}\|c^{[1,3]}\|d^{[1,4]}\|e^{[1,3]}\|f^{[1,4]})^{[2,6]}$
ep2	$(a^{[10,20]}\|b^{[30,40]})^{[3,5]}$
188	$(a\|b)^{+}$
188	$(a^{[10,20]}\|b^{[30,40]})^{[3,5]}$
188	$(a^{[10,20]}\|b^{[30,40]})^{[3,5]}$
ep3	$(((a\|b)?c?(d\|e)?)^{[2,48]})$
988	$((a\|b)?(d\|e)?c?)^{+}$
988	$(((a\|b)?(d\|e)?c?)^{[2,48]})$
988	$(a\|b\|c\|d\|e)^{[6,45]}$
ep4	$(a?b?c?def?g?h?)^{[1,1000]}$
500	$(a?b?c?def?g?h?)^{+}$
500	$(a?b?c?(de)^{[1,10]}f?g?h?)^{[1,100]}$
500	$(a?b?c?def?g?h?)^{[1,597]}$
ep5	*None*
48	$(a\|(b(c\|d)^{+}))^{+}$
48	$((b(d^{[1,2]}\|c^{[1,2]})^{[1,8]})^{[1,2]}\|a^{[1,3]})^{[1,9]}$
48	$((b(d^{[1,2]}\|c^{[1,2]})^{[1,8]})^{[1,2]}\|a^{[1,3]})^{[1,9]}$

Table 3. Proportions of SOREs, ECsores, and RCsores.

Subclasses	% of XSDs
SOREs	80.74
ECsore	93.53
RCsore	89.45



(a)



(b)

Fig. 3. (a) is average precision as a function of the sample size for each of *InfECsore* and *InfRCsore*. (b) is average recall as a function of the sample size for each of *InfECsore* and *InfRCsore*.

4.2 Performance

Generalization Abilities. Since the corresponding results of the algorithms *InfECsore* and *InfRCsore* have different generalization abilities for the same sample (such as `ep3` and `ep4` showed in Table 2), we evaluate the algorithms *InfECsore* and *InfRCsore* by computing the precision and recall. We specify that, the learnt expression with higher precision and recall has better generalization ability. The average precision and average recall, which are as functions of sample size, respectively, are the average values over 1000 expressions.

We randomly extracted the 1000 expressions from XSDs, which were grabbed from OGC XML Schema repository[8]. Each one of the 1000 expressions contains the counters, where the upper bounds are less than 100. To learn each extracted expression e_0, we randomly generated corresponding XML data by using ToXgene, the samples are extracted from the XML data, each sample size is that

[8] http://schemas.opengis.net/.

listed in Fig. 3. And we define precision (p) and recall (r). Let positive sample (S_+) be the set of the all strings accepted by e_0, and let negative sample (S_-) be the set of the all strings not accepted by e_0. Let e_1 be the expression derived by *InfECsore* or *InfRCsore*. A true positive sample (S_{tp}) is the set of the strings, which are in S_+ and accepted by e_1. While a false negative sample (S_{fn}) is the set of the strings, which are in S_+ and rejected by e_1. Similarly, a false positive sample (S_{fp}) is the set of the strings, which are in S_- and accepted by e_1. While a true negative sample (S_{tn}) is the set of the strings, which are in S_- and rejected by e_1. Then, let $p = \frac{|S_{tp}|}{|S_{tp}|+|S_{fp}|}$ and $r = \frac{|S_{tp}|}{|S_{tp}|+|S_{fn}|}$. Note that, for an RCsore, we can construct an equivalent counter automata [14]. The constructed counter automata can decide whether the samples S_+ and S_- can be recognized or not, then we can obtain $|S_{tp}|$, $|S_{fp}|$ and $|S_{fn}|$.

As the sample size increases, compared with the results of *InfECsore*, the plots in Fig. 3(a) demonstrate that the precision for the expression learnt by *InfRCsore* is higher for a smaller sample, but is lower for a larger sample. However, the plots in Fig. 3(b) illustrate that, for any given sample, the recall for the expression learnt by *InfRCsore* is higher than that for the expression derived by *InfECsore*. The reason is that, for the same sample, the learnt RCsore can have more constrains than the learnt ECsore such that some subexpressions without counting operators. This will reduce that the learnt RCsore is expressive enough to cover more XML data. In summary, *InfRCsore* has better generalization ability for a smaller sample.

Time Performance. Although Theorem 3 implies that, for learning a RCsore, the algorithm *InfRCsore* can be faster than the algorithm *InfECsore*, the quantitative analyses of time performance about the algorithms *InfRCsore* and *InfECsore* should be given. Then, we present the evaluation about running time in different size of samples and different size of alphabets. Our experiments were conducted on a ThinkCentre M8600t-D065 with an Intel core i7-6700 CPU (3.4GHz) and 8G memory. And all codes were written in C++.

Table 4(a) shows the average running times in seconds for *InfRCsore* and *InfECsore* as a function of sample size, respectively. Table 4(b) shows the average running times in seconds for *InfRCsore* and *InfECsore* as a function of alphabet size, respectively. We still randomly extracted expressions from XSDs according to the above mentioned method. 1000 expressions of alphabet size 15 are chosen that, to learn each one of them, we randomly generated corresponding XML data by using ToXgene, the samples are extracted from the XML data, each sample size is that listed in Table 4(a). The running times listed in Table 4(b) are averaged over 1000 expressions of that sample size. Another 1000 expressions with distinct alphabet size listed in Table 4(b) are chosen that, to learn each one of them, we also randomly generated corresponding XML data by using ToXgene, the samples are extracted from the XML data, but the corresponding sample size is 1000. The running times listed in Table 4(a) are averaged over 1000 expressions of that alphabet size.

The running times of *InfRCsore* as compared with that of *InfECsore* are reported in Table 4(a). They show that *InfRCsore* is more efficient than *InfECsore* on large samples. However, Table 4(b) illustrates that the speed of *InfRCsore* varies widely when the alphabet size is over 20. Thus, the time performances of *InfRCsore* and *InfECsore* demonstrate that the algorithm *InfRCsore* is more efficient for processing large data sets.

Table 4. (a) and (b) are average running times in seconds for *InfRCsore* and *InfECsore* as the functions of sample size and alphabet size, respectively.

(a)

	time(s) ($\lvert\Sigma\rvert = 15$)	
sample size	*InfRCsore*	*InfECsore*
100	0.044	0.043
1000	0.052	0.079
10000	0.142	0.394
100000	0.989	3.488
1000000	11.389	21.22

(b)

	time(s) ($\lvert S\rvert = 1000$)	
alphabet size	*InfRCsore*	*InfECsore*
5	0.049	0.071
10	0.054	0.075
20	0.067	0.141
50	0.631	0.280
100	1.711	1.269

5 Conclusion

This paper proposed a restricted subclass of deterministic regular expressions with counting: RCsores and the corresponding learning algorithm. The main steps include learning a SORE, constructing an equivalent CFA, running the CFA to obtain an updated CFA, and converting the updated CFA to an RCsore. Compared with previous work, for any given finite language, our algorithm not only can learn a descriptive RCsore, which has higher recall for any sample, but also has better generalization ability for smaller sample, and is more efficient for processing larger sample. A future work is extending the SORE with counting, interleaving, and unorder concatenation, studying the practical issues and the learning algorithms.

References

1. Barbosa, D., Mignet, L., Veltri, P.: Studying the XML Web: gathering statistics from an XML sample. World Wide Web **9**(2), 187–212 (2006)
2. Bex, G.J., Gelade, W., Neven, F., Vansummeren, S.: Learning deterministic regular expressions for the inference of schemas from XML data. In: Proceedings of the 17th International Conference on World Wide Web, pp. 825–834. ACM (2008)
3. Bex, G.J., Gelade, W., Neven, F., Vansummeren, S.: Learning deterministic regular expressions for the inference of schemas from XML data. ACM Trans. Web **4**(4), 1–32 (2010)
4. Bex, G.J., Martens, W., Neven, F., Schwentick, T.: Expressiveness of XSDs: from practice to theory, there and back again. In: Proceedings of the 14th International Conference on World Wide Web, pp. 712–721. ACM (2005)
5. Bex, G.J., Neven, F., Van den Bussche, J.: DTDs versus XML Schema: a practical study. In: Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004, pp. 79–84. ACM (2004)

6. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In: International Conference on Very Large Data Bases, Seoul, Korea, pp. 115–126, September 2006
7. Bex, G.J., Neven, F., Schwentick, T., Vansummeren, S.: Inference of concise regular expressions and DTDs. ACM Trans. Database Syst. **35**(2), 1–47 (2010)
8. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Inf. Comput. **142**(2), 182–206 (1998)
9. Bui, D.D.A., Zeng-Treitler, Q.: Learning regular expressions for clinical text classification. J. Am. Med. Inform. Assoc. **21**(5), 850–857 (2014)
10. Che, D., Aberer, K., Özsu, M.T.: Query optimization in XML structured-document databases. VLDB J. **15**(3), 263–289 (2006)
11. Freydenberger, D.D., Kötzing, T.: Fast learning of restricted regular expressions and DTDs. In: Proceedings of the 16th International Conference on Database Theory, pp. 45–56. ACM (2013)
12. Freydenberger, D.D., Kötzing, T.: Fast learning of restricted regular expressions and DTDs. Theory Comput. Syst. **57**(4), 1114–1158 (2015)
13. Freydenberger, D.D., Reidenbach, D.: Inferring descriptive generalisations of formal languages. J. Comput. Syst. Sci. **79**(5), 622–639 (2013)
14. Gelade, W., Gyssens, M., Martens, W.: Regular expressions with counting: weak versus strong determinism. SIAM J. Comput. **41**(1), 160–190 (2012)
15. Gold, E.M.: Language identification in the limit. Inf. Control **10**(5), 447–474 (1967)
16. Hovland, D.: Regular expressions with numerical constraints and automata with counters. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 231–245. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03466-4_15
17. Kilpeläinen, P., Tuhkanen, R.: Towards efficient implementation of XML Schema content models. In: Proceedings of the 2004 ACM Symposium on Document Engineering, pp. 239–241. ACM (2004)
18. Kilpeläinen, P., Tuhkanen, R.: One-unambiguity of regular expressions with numeric occurrence indicators. Inf. Comput. **205**(6), 890–916 (2007)
19. Latte, M., Niewerth, M.: Definability by weakly deterministic regular expressions with counters is decidable. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 369–381. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_29
20. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: ACM SIGPLAN Notices, vol. 52, pp. 70–80. ACM (2016)
21. Manolescu, I., Florescu, D., Kossmann, D.: Answering XML queries on heterogeneous data sources. In: VLDB, vol. 1, pp. 241–250 (2001)
22. Martens, W., Neven, F.: Typechecking top-down uniform unranked tree transducers. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) ICDT 2003. LNCS, vol. 2572, pp. 64–78. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36285-1_5
23. Mignet, L., Barbosa, D., Veltri, P.: The XML Web: a first study. In: Proceedings of the 12th International Conference on World Wide Web, pp. 500–510. ACM (2003)
24. Moreo, A., Eisman, E.M., Castro, J.L., Zurita, J.M.: Learning regular expressions to template-based FAQ retrieval systems. Knowl.-Based Syst. **53**, 108–128 (2013)
25. Wang, X., Chen, H.: Inferring deterministic regular expression with counting. In: Trujillo, J.C., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 184–199. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_15

26. Wang, X., Chen, H.: Learning a subclass of deterministic regular expression with counting. In: Douligeris, C., Karagiannis, D., Apostolou, D. (eds.) KSEM 2019. LNCS (LNAI), vol. 11775, pp. 341–348. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29551-6_29
27. Xie, Y., Yu, F., Achan, K., Panigrahy, R., Hulten, G., Osipkov, I.: Spamming botnets: signatures and characteristics. ACM SIGCOMM Comput. Commun. Rev. **38**(4), 171–182 (2008)