Inferring Restricted Regular Expressions with Interleaving from Positive and Negative Samples

Yeting Li[1,2], Haiming Chen[1(✉)], Lingqi Zhang[3], Bo Huang[4], and Jianzhao Zhang[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{liyt,chm,zhagjz}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Beijing University of Technology, Beijing, China
zhanglingqisteve@gmail.com
[4] Northwestern Polytechnical University, Xi'an, China
HBruomeng@outlook.com

Abstract. The presence of a schema for XML documents has numerous advantages. Unfortunately, many XML documents in practice are not accompanied by a schema or a valid schema. Therefore, it is essential to devise algorithms to infer schemas. The fundamental task in XML schema inference is to learn regular expressions. In this paper, we focus on learning the subclass of RE(&) called SIREs (the subclass of regular expressions with interleaving). Previous work in this direction lacks inference algorithms that support inference from positive and negative examples. We provide an algorithm to learn SIREs from positive and negative examples based on genetic algorithms and parallel techniques. Our algorithm also has better expansibility, which means that our algorithm not only supports learning with positive and negative examples, but also supports learning with positive or negative examples only. Experimental results demonstrate the effectiveness of our algorithm.

Keywords: XML · Schema inference · Learning expressions · Interleaving · Positive and negative examples

1 Introduction

A classical problem in grammatical inference is to identify a language from positive examples and negative examples. We study learning regular expressions (REs) with *interleaving* (*shuffle*), denoted by RE(&). Since RE(&) are widely used in various areas of computer science [1], including XML database systems [5,12,26], complex event processing [24], system verification [4,13,15], plan recognition [18] and natural language processing [21,28].

H. Chen—Work supported by the National Natural Science Foundation of China under Grant Nos. 61872339 and 61472405.

H. W. Lauw et al. (Eds.): PAKDD 2020, LNAI 12085, pp. 769–781, 2020.
https://doi.org/10.1007/978-3-030-47436-2_58

Studying the inference of RE(&) has several practical motivations, such as *schema inference*. The presence of a schema for XML documents has many advantages, such as for query processing and optimization, data integration and exchange [11,30]. However, many XML documents in practice are not accompanied by a valid schema [16], making schema inference an attractive research topic [2,3,10,14,31]. Learning Relax NG schemas is an important research problem for schema inference, since it is more powerful than other XML schema languages, such as DTD or XSD [5] and has unrestricted supports for the interleaving operator. It is known that the essential task in Relax NG schema inference is learning RE(&) from a set of given sample [23,31].

Previously, RE(&) learning has been studied from positive examples only [23,29,31]. However, negative examples might be useful in some applications. For instance, the *schema evolution* [8,9] can be done incrementally, with little feedback needed from the user, when we also allow negative examples. Learning RE(&) from positive and negative examples may have other crucial applications, such as *mining scientific workflows*. REs have already been used in the literature as a well-suited mechanism for inter-workflow coordination [17]. The user labeled some sequences of modules from a set of available workflows as positive or negative examples. So such algorithms can be thus applied to infer the workflow pattern that the user has in mind.

Such kinds of applications motivate us to investigate the problem of learning RE(&) from positive and negative examples. Most researchers have studied subclasses of REs, which are expressive enough to cover the vast majority of real-world applications [6,7,22] and perform better on several decision problems than general ones [6,7,19,20,25,27]. Bex et al. [3] proposed learning algorithms for two subclasses of REs: SOREs and CHAREs, which capture many practical DTDs/XSDs and are both single occurrence REs. Bex et al. [2] also studied learning algorithms, based on the Hidden Markov Model, for the subclass of REs in which each alphabet symbol occurs at most k times (k-OREs). More recently, Freydenberger and Kötzing [10] proposed more efficient algorithms for the above-mentioned SOREs and CHAREs. Existing work on RE(&) learning mentioned above [23,29,31] are all working on specific subclasses of REs. The aim of these approaches is to infer restricted subclasses of single occurrence REs with interleaving starting from a positive set of words representing XML documents based on *maximum clique* or *maximum independent set*.

In this paper, we focus on learning the subclass of RE(&), called SIREs (see Definition 1) [29]. It has been proved that the problem of learning SIREs is NP-hard [29]. Here, we solve this problem by using genetic algorithms and parallel techniques. Genetic algorithms have been used to solve NP problems, and parallel techniques can make programs more efficient. As a result, when given both positive and negative examples, we can effectively learn a SIRE.

The main contributions of this paper are listed as follows.

– We design algorithm *i*SIRE based on genetic algorithm, which can learn SIREs from both positive and negative examples. To the best of our knowledge, our work is the first one to infer the subclass of RE(&) from positive and negative examples. We hope our work may shed some light on further research.

– Our algorithm has better expansibility. Algorithm *i*SIRE not only supports learning with positive and negative examples, but also supports learning with positive or negative examples only.
– We conduct a series of experiments with alphabets of different sizes. The results reveal the effectiveness of *i*SIRE, show the high accuracy and preciseness of our work.

2 Preliminaries

Regular Expression with Interleaving. Let Σ be a finite alphabet of symbols. The set of all words over Σ is denoted by Σ^*. The empty word is denoted by ε. A RE with interleaving over Σ is defined inductively as follows: ε or $a \in \Sigma$ is a RE, for REs r_1 and r_2, the disjunction $r_1|r_2$, the concatenation $r_1 \cdot r_2$, the interleaving $r_1 \& r_2$, or the Kleene-Star r_1^* is also a RE. $r^?$ and r^+ are abbreviations of $r|\varepsilon$ and $r \cdot r^*$, respectively. They are denoted as RE(&).

The size of a RE r, denoted by $|r|$, is the total number of symbols and operators occurred in r. The language $L(r)$ of a RE r is defined as follows: $L(\varnothing) = \varnothing$; $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L(r_1^*) = L(r_1)^*$; $L(r_1 \cdot r_2) = L(r_1)L(r_2)$; $L(r_1|r_2) = L(r_1) \cup L(r_2)$; $L(r_1 \& r_2) = L(r_1) \& L(r_2)$. Let $u = au'$ and $v = bv'$ where $a, b \in \Sigma$ and $u', v' \in \Sigma^*$, then $u \& \varepsilon = \varepsilon \& u = u$ and $u \& v = a(u' \& v) \cup b(u \& v')$. For example, $L(ab\ cd) = \{cdab, cadb, cabd, acdb, acbd, abcd\}$.

A RE with interleaving r is **SOIRE**, if every alphabet symbol occurs at most once in r. We consider the subclass of REs with interleaving (SIREs) defined by the following grammar.

Definition 1. *The subclass of REs with interleaving (SIREs) are SOIREs over Σ defined by the following grammar:*

$$S ::= T \& S | T$$
$$T ::= \varepsilon | a | a^* | TT, \text{where } a \in \Sigma$$

For instance, $a^* b^? \& cd^+$ is a SIRE, but $a^+ b \& c^+ a$ is not because a appears twice.

Definition 2 ***Candidate Region (CR).*** *We use candidate region to define the skeleton structure of a SIRE. Let $\mathbb{N}$ ={0,1,2,...}, $\mathbb{N}_0 = \mathbb{N} \backslash \{0\}$ (0 is excluded). For a SIRE $r := D_1 \& \cdots \& D_n$ where $D_i \in \Sigma^*$, $1 \le i \le n$, $1 \le n \le \mathbb{N}_0$, it belongs to the candidate region $|D_1| \& \cdots \& |D_n|$. The size of D_i, denoted by $|D_i|$, is the total number of alphabet symbols occurred in D_i.*

For a given alphabet $|\Sigma| = n$, it is easy to see there are 2^{n-1} CRs. For example, consider Σ={a, b, c, d, e} and $|\Sigma|$=5. As is shown in Fig. 1, we can get 16 CRs. The number of squares with the same color represents the $|D_i|$, e.g., the 6th CR denotes 1&1&3 and the 12th CR denotes 1&1&1&2. So, the SIRE $r_1 = a^+ \& b \& c^* d^+ e^?$ belongs to the 6th CR 1&1&3 and the SIRE $r_2 = a^+ \& b \& c^* \& d^+ e^?$ belongs to the 12th CR 1&1&1&2.

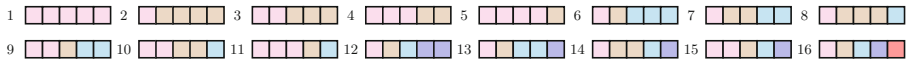

Fig. 1. All the candidate regions of $|\Sigma| = 5$

3 Learning Algorithm

Our algorithm aims to obtain an accurate and precise SIRE, which should accept as many positive samples as possible and reject as many negative samples as possible. We show the major technical details of our algorithm in this section. The main algorithm is presented in Sect. 3.1. Initializing all the simplified candidate regions (SCRs) is introduced in Sect. 3.2. Selecting the best candidate SIRE from each SCR is given in Sect. 3.3.

3.1 The Main Algorithm

The algorithm *i*SIRE first figures out the SCRs of the expression to be learned, then for each SCR, employs genetic algorithms to learn character sequence and multiplicity sequence in parallel, and decodes each learned sequence to a SIRE according to its SCR. After multi-generation evolution and iteration, the best SIRE is selected by function *bestRE()*. The main procedures of the algorithm are presented in Algorithm 1, and are illustrated as follows.

- Scan positive examples S_+ and negative examples S_- to get the alphabet Σ, then call function *getSCRs()* to initialize all the SCRs based on $|\Sigma|$.
- In parallel, call algorithm candSIRE to select the best SIRE from each SCR, and put them in the candidate set C.
- Call function *bestRE()* to select the best SIRE from C and output it.

Function *bestRE()* is designed to select the best SIRE. It measures two metrics of SIREs: $K(r)$ for *accuracy* and $CC(r)$ [23] for *preciseness*. For a SIRE r, $K(r) = \frac{|T_P|+|T_N|-|F_P|-|F_N|}{|S_+|+|S_-|}$, $T_P = \{w \in S_+ | w \in L(r)\}$, $T_N = \{w \in S_- | w \notin L(r)\}$, $F_P = \{w \in S_+ | w \notin L(r)\}$, $F_N = \{w \in S_- | w \in L(r)\}$, S_+ is the set of positive examples and S_- is the set of negative examples. The Combinatorial Cardinality ($CC(r)$, introduced in [23]) of r can be computed as follows: $CC(r) = \prod_{i=1}^{n-1} \binom{\sum_{j=1}^{i+1} |D_j|}{|D_{i+1}|}$. Note that $K(r)$ has a higher priority than $CC(r)$ when selecting the best SIRE. If the value of $K(r)$ is larger, then it means r can accept more positive examples and reject more negative examples. Smaller the $CC(r)$ is, the more precise the SIRE will be. In the rest of this section, we will discuss the implementations of lines 3, 4 and 5 in detail.

3.2 Initializing All the Simplified Candidate Regions (SCRs)

Next, we will give a detailed explanation of Line 3 of Algorithm 1 (initializing all the SCRs) in this section.

Algorithm 1: *i*SIRE

Input: positive examples S_+, negative examples S_-
Output: a SIRE r
1 initialize candidate set $C \leftarrow \varnothing$
2 $\Sigma \leftarrow$ getAlphabet(S_+, S_-)
3 $SCRs \leftarrow$ getSCRs($|\Sigma|$)
4 **foreach** $scr \in SCRs$ **in parallel do**
5 add candSIRE(S_+, S_-, scr) to C
6 **return** $r \leftarrow$ bestRE(C)

From Definition 2, when $|\Sigma| = n$, there are 2^{n-1} CRs. Because of the unorder features of SIREs, we can easily find that for a SIRE $r = D_1\& \cdots \&D_n$, the order of D_i can be arbitrary, where $1 \leq i \leq n$. Hence, we can merge some equivalent CRs and get the SCRs. For instance, in Fig. 1, we can merge the 6th CR 1&1&3, the 8th CR 1&3&1 and the 11th CR 3&1&1 together. After the merger of some equivalent CRs, we get the SCRs shown in Fig. 2.

When the $|\Sigma| = n$, how many SCRs are there? This problem is equivalent to Integer Partition, e.g., when $|\Sigma| = 5$, there are 7 SCRs, including 5, 4&1, 3&2, 3&1&1, 2&2&1, 2&1&1&1 and 1&1&1&1&1. Meanwhile, the 7 partitions of 5 are: $5=5$, $5=4+1$, $5=3+2$, $5=3+1+1$, $5=2+2+1$, $5=2+1+1+1$ and $5=1+1+1+1+1$. In general, approximation formulas exist that can calculate the number of partitions. For $n \in \mathbb{N}$, the number of partitions of n $p(n) \approx \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$ as $n \to \infty$. As n increases, $\frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$ is far less than 2^{n-1}. It can be seen from Table 1 intuitively that the number of SCRs is far less than CRs. So we use function *getSCRs()* to get SCRs instead of CRs in our algorithm *i*SIRE.

Fig. 2. All the simplified candidate regions of $|\Sigma| = 5$

Table 1. The number of CRs and SCRs of varying alphabet size.

$\lvert\Sigma\rvert$	5	10	15	20	25	30	35	40
CRs	16	512	16384	524288	16777216	536870912	17179869184	549755813888
SCRs	7	42	176	627	1958	5604	14883	37338

3.3 Selecting the Best Candidate SIRE from Each SCR

For each SCR obtained in first step, we employ the algorithm candSIRE (shown in Algorithm 2) to find the best candidate SIRE. Because each SCR is independent of each other and does not interfere with each other, we use multi-thread on our multi-core processor to run the candSIRE algorithm in parallel. By using

parallel processing, we can infer the best candidate SIRE for each SCR with numerous SIREs simultaneously, which makes a huge difference when there are often hundreds of SIREs to evaluate per SCR.

Algorithm 2: candSIRE

Input: positive examples S_+, negative examples S_-, an SCR scr
Output: a SIRE r

```
initialize character population C_POP
for g = 1 to C_G_max do
    initialize candidate list SIREs ← ∅
    foreach cs ∈ C_POP in parallel do
        add decode(cs,selectMuls(cs, scr),scr) to SIREs
    C_POP ← select(C_POP,calcValues(SIREs, S+, S-))
    charCrossover()
    charMutate()
return r ← bestRE(SIREs)
```

The algorithm candSIRE uses a number of genetic operators. Using the alphabet $\Sigma = \{a, b, c, d, e\}$ as an example, we introduce some of them as follows.

- *character crossover:* in the character population, we randomly select two character sequences $p_1 = ebdac$ and $p_2 = eabdc$ as parents (in Fig. 3). First, we select the genetic information (bd) and (ab) of the parent p_1 and p_2 at the same position, and put them into the children c_1 and c_2 at the corresponding position respectively. Then we explain how to add the genetic information of the parent p_2 into the child c_1. Our method starts from the ending position of genetic information of p_2, and then passes each gene of parent p_2, which has not appeared in c_1, to c_1. In this example, we start with the gene d of p_2, because d is already in c_1, we skip it and move to c. Since c is not in c_1, we can put c to the first available location of c_1. As we arrived at the end of p_2, we moved to the first gene e. This time, e is not in c_1, so we can add e to the next available location of c_1. Continue the process we get $c_1 = cbdea$. In the same way, we generate $c_2 = cabed$, and they are shown in Fig. 3.
- *character mutation:* the principle of character mutation is to traverse each gene and determine the mutation according to the mutation rate. If the selected gene mutates, the method randomly selects another gene and exchange their positions. For example, for a character sequence $p_1 = abcde$, we assume the selected gene a mutates, then we select gene c and exchange the position of a and c, thus finally get $p_1^{'} = ebcda$ shown in Fig. 4.
- *chromosome encoding:* as is shown in Fig. 5, when we encode SIRE $r = d^?b^+c^*\&e^+\&a$, we can extract a SCR 3&1&1, a multiplicity sequence "?+∗+1" and a character sequence $dbcea$.
- *chromosome decoding:* we decode a SCR 2&2&1, a multiplicity sequence "∗?? + ∗" and a character sequence $abcde$ to get a SIRE $r = a^*b^?\&c^?d^+\&e^*$. The example is shown in Fig. 6.

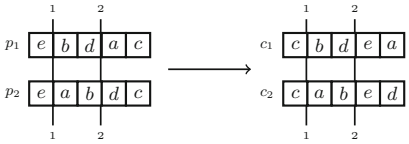

Fig. 3. Character crossover

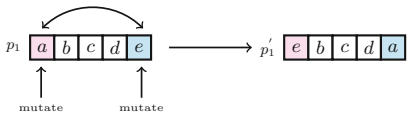

Fig. 4. Character mutation

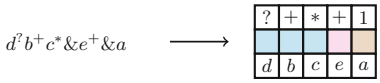

Fig. 5. Chromosome encoding

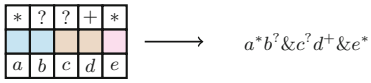

Fig. 6. Chromosome decoding

In the algorithm candSIRE, for a given SCR, positive examples S_+ and negative examples S_-, we select the best candidate SIRE that accepts as many positive samples as possible, rejects as many negative samples as possible, and as precise as possible. The main procedures are as follows.

1. Initialize the population of candidate character sequences. Here we set the population size to 500.
2. Select the best multiplicity sequence for each character sequence using algorithm selectMuls in parallel. The pseudocode of selectMuls is presented in Algorithm 3, we will explain its details later.
3. Decode each pair of character sequences, corresponding best multiplicity sequences and the given SCR to get the population of candidate SIREs by calling function *decode()*.
4. Call function *calcValues()*, calculate fitness value $f(r)$ for each SIRE r. The fitness value $f(r)$ of r is defined as follows.

$$f(r) = (K(r), CC(r)),$$

   For the detailed definitions of $K(r)$ and $CC(r)$, see Sect. 3.1. Our fitness function gives priority to $K(r)$, and then compare the $CC(r)$, that is, on the basis of selecting the SIRE that can accept more positive examples and reject more negative examples, then consider the more precise ones.
5. Call function *select()* to generate the next generation SIREs. The method first retains the best 20% of SIREs by the fitness $f(r)$ in the current population unchanged, and then applies roulette-wheel selection to the remaining 80% to get the next generation SIREs. Meanwhile, it is also important to note that $K(r)$ is the top priority when choosing SIREs. When the values of $K(r)$ are the same, we choose the SIRE which $CC(r)$ is minimum.
6. Call function *charCrossover()*, select some pairs of character sequences according to the crossover rate (0.8), and construct new pairs of character sequences by applying the character crossover.

7. Call function *charMutate()*, select some character sequences according to the mutation rate (0.03), and modify the selected sequences by applying the character mutation.
8. Iterate $2-8$ steps until the number of generations reaches the given threshold C_G_{max}. Here we set C_G_{max}=300. Finally, we call function *bestRE()* (see Sect. 3.1) to select the best SIRE from the last generation of SIREs.

In order to improve the efficiency of evolution, we adopt two tricks to optimize algorithm candSIRE. In the second step of candSIRE, it needs to select the best multiplicity sequence for each character sequence. Obviously, this process can be executed in parallel because these character sequences are independent of each other when finding the best multiplicity sequence. Besides, as is well known, the fitness function is usually the most computationally expensive component of the genetic algorithm. Thus, we use value hashing to reduce the amount of time spent on calculating fitness values by storing previously computed fitness values in a hash table. During execution, solutions found previously will be revisited due to the random mutations and recombinations of SIREs, then we just revisit its fitness value directly from the hash table instead of recalculation. Inevitably, the storage of the hash table consumes memory usage.

Now we introduce the algorithm selectMuls used in the second step of algorithm candSIRE (shown in Algorithm 3), it aims to select best multiplicity sequence for each character sequence. Before introducing the details of selectMuls, we illustrate its genetic operators as follows.

- *multiplicity crossover:* randomly select crossover points of two parents, and then exchange the selected genes to get children. In the multiplicity population, e.g., we randomly select two multiplicity sequences $p_1 =$ "$*+1?+$" and $p_2 =$ "$*?? + +$" as parents. Then we exchange "$+1$" of p_1 and "$??$" of p_2 to get children $c_1 =$ "$*???+$" and $c_2 =$ "$*+1++$" in Fig. 7.
- *multiplicity mutation:* replacing the mutated gene with an element of the set $\{*, +, ?, 1\}$. The principle of character mutation is to traverse each gene of the chromosome and determine the mutation according to the mutation rate. The example is shown in Fig. 8.

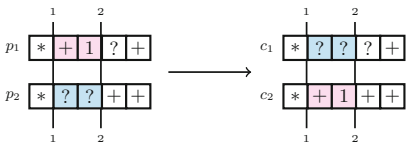

Fig. 7. Multiplicity crossover

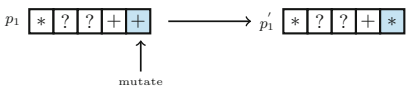

Fig. 8. Multiplicity mutation

The main procedures of selectMuls are illustrated as follows.

1. Initialize the population of candidate multiplicity sequences. Here we set the population size to 200.

2. Call function *decode()*, decode each group of multiplicity sequences, the character sequences and the SCR to get the population of candidate SIREs.
3. Call function *calcValues()*, calculate fitness value $f(r)$ for each SIRE r.
4. Call function *select()*, use roulette-wheel selection to generate a next generation from the current population according to fitness values.
5. Call function *mulCrossover()*, select some pairs of multiplicity sequences according to the crossover rate (0.8), and construct new pairs of multiplicity sequences by multiplicity crossover.
6. Call function *mulMutate()*, select some multiplicity sequences according to the mutation rate (0.03), and modify the sequences by applying the multiplicity mutation.
7. Iterate 2–8 steps until the number of generations reaches the given threshold M_G_{max}. Here we set M_G_{max}=100. Then, we call function *bestRE()* to select the best SIRE r from the last generation of SIREs in the given SCR. Finally, we call function *encode(r).muls* to get a multiplicity sequence.

Algorithm 3: selectMuls

Input: positive examples S_+, negative examples $S\_$, character sequence cs, an SCR scr
Output: a multiplicity sequence ms

```
initialize multiplicity population M_POP
for g = 1 to M_G_max do
    initialize candidate list SIREs ← ∅
    foreach ms ∈ M_POP in parallel do
        add decode(cs, ms, scr) to SIREs
    M_POP ← select(M_POP,calcValues(SIREs, S+, S-))
    mulCrossover()
    mulMutate()
SIREs ← ModifyMuls(SIREs)
r = bestRE(SIREs)
return ms ← encode(r).ms
```

4 Experiments

In this section, we validate our algorithm by means of experimental analysis. All experiments were performed using a prototype implementation of *i*SIRE written in Python 3.6 executed on a machine with sixteen-core Intel Xeon CPU E5620@2.4 GHz, 24 GB memory.

4.1 Learning SIREs from Positive Examples

To compare the algorithms *Exact Minimal* [29], *conMiner* [29], *conDAG* [29] and *iSIRE*, we generate 9 datasets of positive examples with alphabet size $|\Sigma| = \{5, 10, 15\}$ and example size $|S| = \{100, 500, 1000\}$. Table 2 presents the $K(r)$ values and $CC(r)$ values of the learned SIREs. From Table 2 we can see that for all the 9 datasets, the $K(r)$ values of learned SIREs with both algorithms are all 100%, which means both algorithms can guarantee the learned SIREs to cover all positive examples[1]. According to the $CC(r)$ values of SIREs learned by the four algorithms in Table 2, we observe that when the alphabet size $|\Sigma|$ is smaller ($|\Sigma| = 5$), the learned SIREs by *Exact Minimal*, *conMiner*, *conDAG* and *iSIRE* have the same smaller $CC(r)$ values. However, as the alphabet size $|\Sigma|$ grows larger ($|\Sigma| = 15$), the $CC(r)$ values of learned SIREs by *Exact Minimal*, *conMiner* or *conDAG* is much larger than that of *i*SIRE, that means the results learned by *i*SIRE is more precise than the other 3 algorithms.

Table 2. Result of learning SIREs from positive examples.

$\|\Sigma\|$	$\|S\|$	Exact minimal		conMiner		conDAG		*i*SIRE	
		$K(r)$	$CC(r)$	$K(r)$	$CC(r)$	$K(r)$	$CC(r)$	$K(r)$	$CC(r)$
5	100	100%	20	100%	20	100%	20	100%	20
	500	100%	30	100%	30	100%	30	100%	30
	1000	100%	60	100%	60	100%	60	100%	60
10	100	100%	840	100%	840	100%	840	100%	840
	500	100%	37800	100%	50400	100%	50400	100%	**16800**
	1000	100%	5040	100%	6300	100%	5040	100%	**3150**
15	100	100%	1801800	100%	2522520	100%	2162160	100%	**1261260**
	500	100%	8108100	100%	15135120	100%	10810800	100%	**7207200**
	1000	100%	300300	100%	360360	100%	900900	100%	**270270**

4.2 Learning SIREs from Positive and Negative Examples

In order to evaluate the effectiveness of our learning algorithm on learning examples of both positive and negative cases, we would have liked to compare *i*SIRE with other approaches, but this was impossible, since we found no other tools or algorithms supporting learning SIREs from both positive and negative examples. Thus we only conducted experiment with our own algorithm. According to the alphabet size, example size, and the proportion of positive and negative examples, we made 27 datasets of examples and conducted experiment on these examples (shown in Table 3). As Table 3 shows, more than 74% $K(r)$ values

[1] This is also very easy to prove, because the worst case is that the learned SIRE is $a_1^* \& a_2^* \& \cdots \& a_n^*$, which can guarantee that accept all the positive examples.

of inferred SIREs are above 75%, that is, majority of SIREs learned by *i*SIRE accept most of positive examples and reject most of negative examples, which demonstrates the high effectiveness of our algorithms.

Table 3. Results of learning SIREs from positive and negative examples.

$\|\Sigma\|$	$\|S_+\|$	$\|S_-\|$	$K(r)$	$CC(r)$	$\|\Sigma\|$	$\|S_+\|$	$\|S_-\|$	$K(r)$	$CC(r)$	$\|\Sigma\|$	$\|S_+\|$	$\|S_-\|$	$K(r)$	$CC(r)$
5	25	75	92%	10	10	25	75	82%	360	15	25	75	82%	756756
	50	50	74%	30		50	50	72%	840		50	50	74%	360360
	75	25	90%	5		75	25	78%	7560		75	25	82%	25225200
	300	200	87.6%	10		300	200	82.4%	5040		300	200	82.4%	300300
	250	250	88.8%	20		250	250	81.2%	2520		250	250	70.4%	50450400
	200	300	81.2%	60		200	300	83.2%	4200		200	300	75.6%	900900
	750	250	69.8%	20		750	250	85.6%	75600		750	250	72.2%	37837800
	500	500	84%	30		500	500	91.6%	30240		500	500	73.4%	4054050
	250	750	77.6%	60		250	750	84.8%	25200		250	750	82.6%	378378000

5 Conclusions

In this paper, we provided algorithm *i*SIRE to learn a SIRE from positive and negative examples based on genetic algorithms and parallel techniques. Then we conducted experiments with alphabets of different sizes, and results showed that with only positive examples, our learning results are more precise compared with the state-of-the-art algorithms, and when given both positive and negative examples, we can learn SIREs with high accuracy.

References

1. Berglund, M., Björklund, H., Björklund, J.: Shuffled languages - representation and recognition. Theor. Comput. Sci. **489–490**, 1–20 (2013)
2. Bex, G.J., Gelade, W., Neven, F., Vansummeren, S.: Learning deterministic regular expressions for the inference of schemas from XML data. TWEB **4**(4), 14:1–14:32 (2010)
3. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In: Proceedings of the 32nd VLDB, pp. 115–126 (2006)
4. Boja'nczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: Proceedings of the 21st LICS, pp. 7–16 (2006)
5. Clark, J., Makoto, M.: RELAX NG Tutorial (2003). https://relaxng.org/tutorial-20030326.html
6. Colazzo, D., Ghelli, G., Pardini, L., Sartiani, C.: Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. Theor. Comput. Sci. **492**, 88–116 (2013)
7. Colazzo, D., Ghelli, G., Sartiani, C.: Linear time membership in a class of regular expressions with counting, interleaving, and unordered concatenation. ACM Trans. Database Syst. **42**(4), 24:1–24:44 (2017)

8. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. PVLDB **4**(2), 117–128 (2010)
9. Florescu, D.: Managing semi-structured data. ACM Queue **3**(8), 18–24 (2005)
10. Freydenberger, D.D., Kötzing, T.: Fast learning of restricted regular expressions and DTDs. Theory Comput. Syst. **57**(4), 1114–1158 (2015)
11. Gallinucci, E., Golfarelli, M., Rizzi, S.: Schema profiling of document-oriented databases. Inf. Syst. **75**, 13–25 (2018)
12. Gao, S., Sperberg-McQueen, C.M., Thompson, H.S.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures (2012). https://www.w3.org/TR/xmlschema11-1/
13. Garg, V.K., Ragunath, M.T.: Concurrent regular expressions and their relationship to petri nets. Theor. Comput. Sci. **96**(2), 285–304 (1992)
14. Garofalakis, M., Gionis, A., Shim, K., Shim, K., Shim, K.: XTRACT: learning document type descriptors from XML document collections. Data Min. Knowl. Disc. **7**(1), 23–56 (2003)
15. Gischer, J.L.: Shuffle languages, petri nets, and context-sensitive grammars. Commun. ACM **24**(9), 597–605 (1981)
16. Grijzenhout, S., Marx, M.: The quality of the XML Web. J. Web Semant. **19**, 59–68 (2013)
17. Heinlein, C.: Workflow and process synchronization with interaction expressions and graphs. In: Proceedings of the 17th ICDE, pp. 243–252 (2001)
18. Högberg, J., Kaati, L.: Weighted unranked tree automata as a framework for plan recognition. In: Proceedings of the 13th FUSION, pp. 1–8 (2010)
19. Hovland, D.: The inclusion problem for regular expressions. J. Comput. Syst. Sci. **78**(6), 1795–1813 (2012)
20. Hovland, D.: The membership problem for regular expressions with unordered concatenation and numerical constraints. In: Proceedings of the 6th LATA, pp. 313–324 (2012)
21. Kuhlmann, M., Satta, G.: Treebank grammar techniques for non-projective dependency parsing. In: Proceedings of the 12th EACL, pp. 478–486 (2009)
22. Li, Y., Chu, X., Mou, X., Dong, C., Chen, H.: Practical study of deterministic regular expressions from large-scale XML and schema data. In: Proceedings of the 22nd IDEAS, pp. 45–53 (2018)
23. Li, Y., Mou, X., Chen, H.: Learning concise Relax NG schemas supporting interleaving from XML documents. In: Proceedings of the 14th ADMA, pp. 303–317 (2018)
24. Li, Z., Ge, T.: PIE: approximate interleaving event matching over sequences. In: Proceedings of the 31st ICDE, pp. 747–758 (2015)
25. Losemann, K., Martens, W., Niewerth, M.: Closure properties and descriptional complexity of deterministic regular expressions. Theor. Comput. Sci. **627**, 54–70 (2016)
26. Martens, W., Neven, F., Niewerth, M., Schwentick, T.: BonXai: combining the simplicity of DTD with the expressiveness of XML schema. In: Proceedings of the 34th PODS, pp. 145–156 (2015)
27. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for XML Schemas and chain regular expressions. SIAM J. Comput. **39**(4), 1486–1530 (2009)
28. Nivre, J.: Non-projective dependency parsing in expected linear time. In: Proceedings of the 47th ACL, pp. 351–359 (2009)

29. Peng, F., Chen, H.: Discovering restricted regular expressions with interleaving. In: Cheng, R., Cui, B., Zhang, Z., Cai, R., Xu, J. (eds.) APWeb 2015. LNCS, vol. 9313, pp. 104–115. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25255-1_9
30. Wang, L., et al.: Schema management for document stores. PVLDB **8**(9), 922–933 (2015)
31. Zhang, X., Li, Y., Cui, F., Dong, C., Chen, H.: Inference of a concise regular expression considering interleaving from XML documents. In: Proceedings of the 22nd PAKDD, pp. 389–401 (2018)