Inference of Grammars for Trees of Variable Arity

Neetha Maria Sebastian
Guide: Prof Kamala Krithivasan
Dept. of CSE, IITM
1. **Introduction**

2. **Problem**

3. **Regular Expression Inference**

4. **Other Results**

5. **Conclusion**
OUTLINE

1. INTRODUCTION
   - Fundamentals

2. Problem

3. Regular Expression Inference

4. Other Results

5. Conclusion
What is Grammatical Inference?

- Find a grammar \((G)\) that describes a language \(L\).
- Sample data \((D)\) is given as input.
- Finds common substructures /generalisations

Data can be

**Positive** - samples \(\in L\)

**Negative** - samples \(\notin L\)

Inference Process

- Use positive data alone
- Use positive and negative data
Grammatical Inference assumes that there exists a grammar $G_0$ which can generate data $D$.

The task is to find $G$ as close to $G_0$ as possible.
**Outline**

1. Introduction
2. **Problem**
3. Regular Expression Inference
4. Other Results
5. Conclusion
Motivation

- Well formed XML documents do not always have schema associated with it.
- Validation of the structure of automatically generated documents.
- Other uses include designing of schemas for complex documents.
- Finding a common schema for a heterogeneous collection of documents.

A good schema inferencing algorithm would be useful.
Existing Solutions

**XTRACT**  A regular expression inference engine.

**BEX**  Similar (but better) regular expression inference engine.

Both methods seem to have their own drawbacks.
OUTLINE

1. Introduction

2. Problem

3. Regular Expression Inference
   - Theoretical Results on Regular Expression Inference
   - An example of an inference Algorithm
   - A Special Case
   - Our Solution

4. Other Results

5. Conclusion
Theoretical Results on Regular Expression Inference

Gold's Result

Superfinite All finite cardinality languages and at least one infinite cardinality language.

Inference Superfinite languages can be identified in the limit from complete presentation only. They cannot be identified from positive samples alone.

Regular Languages The class of regular languages is superfinite.

Theorem

Regular Languages cannot be identified in the limit from positive samples alone.


**Angluin’s Result**

**Gold**  It is not possible to identify the class of regular languages from positive samples alone.

**Subclasses**  But it is possible to identify subclasses of regular languages from positive samples

**K Testable**  $k$ testable languages can be identified in the limit from positive samples alone.
K TESTABLE LANGUAGES IN THE STRICT SENSE

$k$-TSSL — finite set of substrings of length $k$ allowed to appear in strings of the language.

\[ Z_k = (\Sigma, I_k, F_k, T_k), \]

$\Sigma$ is a finite alphabet,

$I_k, F_k \subseteq \bigcup_{i=1}^{k-1} \Sigma^i$ are two sets of initial and final segments, respectively,

$T_k \subseteq \Sigma^k$ is a set of forbidden segments of length $k$.

**Intuition**

- start with segments in $I_k$, end with segments in $F_k$
- no segments of length $k$ which is in $T_k$
**K TESTABLE LANGUAGES IN THE STRICT SENSE**

$k$-TSSL — finite set of substrings of length $k$ allowed to appear in strings of the language.

$$Z_k = (\Sigma, I_k, F_k, T_k),$$

where $\Sigma$ is a finite alphabet,

$$I_k, F_k \subseteq \bigcup_{i=1}^{k-1} \Sigma^i$$

are two sets of initial and final segments, respectively,

$$T_k \subseteq \Sigma^k$$

is a set of forbidden segments of length $k$.

**K TSSL**

A $k$ testable language in the strict sense is $(kTLSS)$ is defined by the regular expression $I(Z_k) = I_k \Sigma^* \cap \Sigma^* F_k - \Sigma^* T_k \Sigma^*$
An example of an inference Algorithm

**Learning of k testable languages**

\[ R = \text{learning set (positive samples)} \text{ and } k \geq 1. \text{ We can associate a } \]

\[ Z_k(R) = (\Sigma(R), I_k(R), F_k(R), T_k(R)) \text{ where } \]

\[ I_k(R) = \{ u | uv \in R, |u| = k - 1, v \in \Sigma(R)^* \} \cup \{ x \in R | |x| < k - 1 \} \]

\[ F_k(R) = \{ v | uv \in R, |v| = k - 1, u \in \Sigma(R)^* \} \cup \{ x \in R | |x| \leq k - 1 \} \]

\[ T_k(R) = \Sigma(R)^k - \{ v | uvw \in R, |v| = k, u, w \in \Sigma(R)^* \} \]
**Inference Algorithm for k Testable Languages - k TSSI**

**Input:** $k \geq 2$ and $R$, the set of training strings. **Output:** DFA $A_k = (Q, \Sigma, \delta, q_0, Q_f)$

- $(\Sigma, I, F, T) = (\Sigma(R), l_k(R), F_k(R), T_k(R))$
- $Q = \epsilon, \delta = \phi, q_0 = \epsilon$
- For all $a_1...a_m \in I$ for $j = 1tom$
  $$Q = Q \cup (a_1...a_j)$$
  $$\delta = \delta \cup ((a_1...a_{j-1}, a_j, a_1...a_j))$$
- For all $a_1...a_k \in (\Sigma^k - T)$
  $$Q = Q \cup (a_2...a_k)$$
  $$\delta = \delta \cup ((a_1...a_{k-1}, a_k, a_2...a_k))$$
- $Q_f = F, A_k = (Q, \Sigma, \delta, q_0, Q_f)$
AN EXAMPLE OF AN INFESSION ALGORITHM

EXAMPLE FOR THE WORKING OF k TSSI

\[ R = \{aa, aba, abba, abba\}. \]
Example for the working of $k$ TSSI

$R = \{aa, aba, abba, abbb\}.$
AN EXAMPLE OF AN INFERENCE ALGORITHM

EXAMPLE FOR THE WORKING OF K TSSI

\[ R = \{aa, aba, abba, abbb\a\} \].

\[ \begin{array}{c}
  a \\
  a \\
 \end{array} \rightarrow \begin{array}{c}
  b \\
  b \\
 \end{array} \]
Example for the working of k TSSI

\[ R = \{ aa, aba, abba, abbbba \} . \]
Example for the working of k TSSI

\[ R = \{aa, aba, abba, abbbba\} \]
Example for the working of $k$ TSSI

$$R = \{aa, aba, abba, abbba\}.$$
**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbbba\} \]
An example of an inference Algorithm

**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbb\} \].
An example of an inference Algorithm

**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbbba\}. \]
AN EXAMPLE OF AN INFERENCE ALGORITHM

EXAMPLE FOR THE WORKING OF \textit{k TSSI}

\[ R = \{ aa, aba, abba, abbb \}. \]
EXAMPLE FOR THE WORKING OF k TSSI

\[ R = \{aa, aba, abba, abbbba\}. \]
**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbbba\} \]
\[ R = \{aa, aba, abba, abbbba\}. \]
An example of an inference Algorithm

**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbbba\}. \]
An example of an inference Algorithm

**Example for the working of k TSSI**

\[ R = \{aa, aba, abba, abbbba\} \]
Example for the working of k TSSI

\[ R = \{aa, aba, abba, abbbba\} \.]
AN EXAMPLE OF AN INFEERENCE ALGORITHM

HEIRARCHY OF THE INFERRED LANGUAGES

give a figure showing the hierarchy.
A Special Case

**The case of** $k = 2$ (BEX’s Algorithm)

$k = 2$ method for inference uses $k$ TSSI algorithm for the case $k = 2$.

**SOA** Constructs a single object automaton - one state for each letter in $\Sigma$.

**Transitions** from one state to the other happens on reading a letter.

**Reduction** The SOA is converted into a concise regular expression.

**Drawback**

Each symbol in the regular expression can appear only once.
ALTERNATIVE SOLUTION

- Use larger $k$!
- Allows tighter representation.

**DRAWBACK**

$Q \subseteq \bigcup_{i=0}^{k-1} \Sigma^i$. Thus the number of states grows exponentially with increasing $k$.

Solution: Instead of creating one state for each substring of length $k$, create a state only for frequently occurring patterns.
Inference of Regular Expressions using a Pattern Automaton

Find Patterns Identify commonly occurring patterns which can cover all the strings

Pattern Automaton Create an automaton that can represent all the strings in terms of the identified patterns

Regular Expression Reduce this pattern automaton into a regular expression using a set of rules.
Our Solution

Notations

**Pattern**: sequence of symbols that occur in the set repeatedly.

**Maximal**: A maximal repeating pattern is one that occurs maximum number of times in the set of input strings.

**Covered**: if each of the input strings can be completely represented as a sequence of patterns.

**Sequence**: obtained by replacing the input strings with its set of maximal repeating patterns.

**Successor**: For $p_1 p_2 \ldots p_n$ a "successor relation" is: $p_j$ is a successor of $p_i$ if $p_j$ occurs after $p_i$ in any input string.
Example

Strings are *aabaa*, *aabaabaaba* and *abaabaaabaaabaabaa*.

<table>
<thead>
<tr>
<th>Parent</th>
<th>Pattern</th>
<th>No of Occurrences</th>
<th>Chosen</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>24</td>
<td>yes</td>
<td>first pattern</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>9</td>
<td>yes</td>
<td>first pattern</td>
</tr>
<tr>
<td>a</td>
<td>ba</td>
<td>10</td>
<td>no</td>
<td>10 &lt; 24</td>
</tr>
<tr>
<td>a</td>
<td>ab</td>
<td>9</td>
<td>no</td>
<td>9 &lt; 24</td>
</tr>
<tr>
<td>a</td>
<td>aa</td>
<td>9</td>
<td>no</td>
<td>9 &lt; 24</td>
</tr>
<tr>
<td>b</td>
<td>ba</td>
<td>9</td>
<td>yes</td>
<td>9 &gt;= 9</td>
</tr>
<tr>
<td>b</td>
<td>ab</td>
<td>9</td>
<td>yes</td>
<td>9 &gt;= 9</td>
</tr>
<tr>
<td>ba</td>
<td>baa</td>
<td>8</td>
<td>no</td>
<td>8 &lt; 9</td>
</tr>
<tr>
<td>ba</td>
<td>aba</td>
<td>9</td>
<td>yes</td>
<td>9 &gt;= 9</td>
</tr>
<tr>
<td>ab</td>
<td>aba</td>
<td>9</td>
<td>yes</td>
<td>9 &gt;= 9</td>
</tr>
<tr>
<td>ab</td>
<td>aab</td>
<td>8</td>
<td>no</td>
<td>8 &lt; 9</td>
</tr>
<tr>
<td>aba</td>
<td>abaa</td>
<td>6</td>
<td>no</td>
<td>6 &lt; 9</td>
</tr>
<tr>
<td>aba</td>
<td>aaba</td>
<td>6</td>
<td>no</td>
<td>8 &lt; 9</td>
</tr>
</tbody>
</table>
Our Solution

**Pattern Automaton**

The patterns identified are: $p_1 : aba$ and $p_2 : a$

The pattern sequences formed are: $p_2.p_1.p_2.p_2$, $p_2.p_1.p_1.p_1$, $p_1.p_1.p_2.p_1.p_2.p_2.p_1.p_1.p_1.p_2$

A Pattern Automaton is a four tuple $Q, \Sigma, \delta, F$

- $Q$ - set of states in the automaton, each state corresponding to a pattern,
- $\Sigma$ is the input alphabet,
- $\delta$ is the set of transitions
- $F$ is the set of final states.

Here $\delta : Q \times \Sigma^* \rightarrow Q$. 

Neetha Maria Sebastian Guide: Prof Kamala Krithivasan

Inference of Grammars for Trees of Variable Arity
Our Solution

Constructing a Pattern Automaton

\[ p_1 \cdot p_1 \cdot p_2 \cdot p_2, \]
\[ p_2 \cdot p_1 \cdot p_1 \cdot p_1, \]
\[ p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 \]
Our Solution

Constructing a Pattern Automaton

\[ p_1 \cdot p_1 \cdot p_2 \cdot p_2, \]
\[ p_2 \cdot p_1 \cdot p_1 \cdot p_1, \]
\[ p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 \]
CONSTRUCTING A PATTERN AUTOMATON

$p_2.p_1.p_2.p_2,$

$p_2.p_1.p_1.p_1,$

$p_1.p_1.p_2.p_1.p_2.p_1.p_1.p_2$
Our Solution

**CONSTRUCTING A PATTERN AUTOMATON**

\[ I \]

\[ p_1 \]

\[ p_1 \]

\[ p_2 \]

\[ p_2 \]

\[ F \]

\[ p_2 \cdot p_1 \cdot p_2 \cdot p_2, \]

\[ p_2 \cdot p_1 \cdot p_1 \cdot p_1, \]

\[ p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 \]
Our Solution

**CONSTRUCTING A PATTERN AUTOMATON**

\[ \begin{align*}
I & \xrightarrow{p_1} p_1 \xrightarrow{p_2} p_2 \\
F & \xrightarrow{p_1} p_1 \xrightarrow{p_2} p_2 \xrightarrow{p_1} p_2 \xrightarrow{p_1} p_1 \xrightarrow{p_2} p_1 \xrightarrow{p_1} p_2 \\
\end{align*} \]
Our Solution

**CONSTRUCTING A PATTERN AUTOMATON**

\[ p_2 \cdot p_1 \cdot p_2 \cdot p_2, \]
\[ p_2 \cdot p_1 \cdot p_1 \cdot p_1, \]
\[ p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 \]
Our Solution

Constructing a Pattern Automaton

\[
p_2 \cdot p_1 \cdot p_2 \cdot p_2, \\
p_2 \cdot p_1 \cdot p_1 \cdot p_1, \\
p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2
\]
Our Solution

**Constructing a Pattern Automaton**

\[
p_2 \cdot p_1 \cdot p_2 \cdot p_2,
\]

\[
p_2 \cdot p_1 \cdot p_1 \cdot p_1,
\]

\[
p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2
\]
Our Solution

**CONSTRUCTING A PATTERN AUTOMATON**

\[
p_2 \cdot p_1 \cdot p_2 \cdot p_2, \\
p_2 \cdot p_1 \cdot p_1 \cdot p_1, \\
p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2
\]
Our Solution

**Constructing a Pattern Automaton**

\[ p_2 \cdot p_1 \cdot p_2 \cdot p_2, \]
\[ p_2 \cdot p_1 \cdot p_1 \cdot p_1, \]
\[ p_1 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 \]
Reducing a Pattern Automaton

A set of rules are used to reduce the constructed automaton into a regular expression.

**Self Loops** remove all self loops \((p_r,p_r)\) and label the corresponding state as \(p_r^*\)

**Concat** States: \(p_r\) and \(p_q\), edge \((p_r,p_q)\) and no other outgoing edge from \(p_r\) and no other incoming edge for \(p_q\)

![Diagram](q)
DISJUNCTION $p_r$ and $p_q$ which occur in alternate paths with same predecessor and successor nodes, create $p_r + p_q$. 

Inference of Grammars for Trees of Variable Arity
Our Solution

...CONTINUED

**OPTIONAL**: A path from $p_r$ to $p_q$ through $p_s$ and an alternate direct path from $p_r$ to $p_q$, eliminate the direct path and label $p_s$ with $p_s$?
Our Solution

...continued

CYCLES

For every \((\text{entry}, \text{exit}) \rightarrow (x, y)\) create a node \((p_r p_{r+1} \cdots p_m p_1 p_2 \cdots p_{r-1})^* p_r p_{r+1} \cdots p_s\) with an arc from \(x\) to this node and another from this node to \(y\).
Our Solution

REDUCTION
Our Solution

**Additional Reduction rules**

- \((p_1^*p_2^*p_3^*\ldots p_n^*)^* \rightarrow (p_1 + p_2 + p_3)^*\)
- \(\beta_1 \delta_1 \gamma_1 + \beta_1 \delta_2 \gamma_1 + \ldots \rightarrow \beta_1 (\delta_1 + \delta_2 \ldots) \gamma_1\)

- Cycle rule generates redundancy. To eliminate this: If there is a part of the regular expression of the form \((R)^* Q\) where \(R\) is a set of patterns that are "ORed" and \(Q\) is a list of the same set of patterns, we can reduce the expression to \((R)^*\).
Our Solution

Language Inferred in the case of our example

Reduction of the cycle gives:
\[(p_1^* p_2^*)^* p_1 + (p_1^* p_2^*)^* p_1 p_2 + (p_2^* p_1^*)^* p_2 (p_2^* p_1^*)^* p_2 p_1).\]

Using the first rule for reducing the regular expressions,
\[(p_1 + p_2)^* p_1 + (p_1 + p_2)^* p_1 p_2 + (p_2 + p_1)^* p_2 (p_2 + p_1)^* p_2 p_1).\]

After using the second rule we get
\[(p_1 + p_2)^* (p_1 + p_2 + p_2 p_1 + p_1 p_2).\]
And using the third one, we get
\[(p_1 + p_2)^*\]
Our Solution

Language Inferred in the case of our example

- Reduction of the cycle gives:
  \[(p_1^* p_2^*)^* p_1 + (p_1^* p_2^*)^* p_1 p_2 + (p_2^* p_1^*)^* p_2 (p_2^* p_1^*)^* p_2 p_1).\]

- Using the first rule for reducing the regular expressions,
  \[(p_1 + p_2)^* p_1 + (p_1 + p_2)^* p_1 p_2 + (p_2 + p_1)^* p_2 (p_2 + p_1)^* p_2 p_1).\]

- After using the second rule we get
  \[(p_1 + p_2)^* (p_1 + p_2 + p_2 p_1 + p_1 p_2).\]
  And using the third one, we get
  \[(p_1 + p_2)^* \]
Our Solution

**LANGUAGE INFERRED IN THE CASE OF OUR EXAMPLE**

Reduction of the cycle gives:

\[(p_1^* p_2^*)^* p_1 + (p_1^* p_2^*)^* p_1 p_2 + (p_2^* p_1^*)^* p_2 (p_2^* p_1^*)^* p_2 p_1).\]

Using the first rule for reducing the regular expressions,

\[(p_1 + p_2)^* p_1 + (p_1 + p_2)^* p_1 p_2 + (p_2 + p_1)^* p_2 (p_2 + p_1)^* p_2 p_1).\]

After using the second rule we get \((p_1 + p_2)^*(p_1 + p_2 + p_2 p_1 + p_1 p_2).\)

And using the third one, we get \((p_1 + p_2)^*\)
LEMMA

Observation:

- The input strings can be written as a sequence of patterns.
- Pattern Automaton represents all successor relationships in the input strings.

LEMMA

Applying each rule in the procedure on a pattern automaton $A$ yields a new automaton $B$ which accepts all strings that are accepted by $A$.

The converse i.e., the other way inclusion need not be true.
Our Solution

**LANGUAGE INFERRED BY THE PATTERN AUTOMATON**

- $k_{\text{small}}$ length of the smallest identified pattern
- $k_{\text{large}}$ length of the largest identified pattern
- $R_{k_{\text{small}}}$ automaton constructed by $k$ TSSI for $k = k_{\text{small}}$
- $R_{k_{\text{large}}}$ automaton constructed by $k$ TSSI for $k = k_{\text{large}}$
- $L(R_{k_{\text{small}}})$ language accepted by $R_{k_{\text{small}}}$
- $L(R_{k_{\text{large}}})$ language accepted by $R_{k_{\text{large}}}$

**THEOREM**

$L(R_{k_{\text{large}}}) \subseteq L(P) \subseteq L(R_{k_{\text{small}}})$
Our Solution

LANGUAGE INFERRED BY THE PATTERN AUTOMATON

\[ L(R_{k_{\text{large}}}) \subseteq L(P) \]

Figure above shows the pattern automaton and the corresponding states in \( R_{k_{\text{large}}} \)
**Language Inferred by the Pattern Automaton**

\[ L(P) \subseteq L(R_{k_{small}}) \]

Figure above shows the pattern automaton and the corresponding states in \( R_{k_{small}} \)
**OUTLINE**

1. Introduction
2. Problem
3. Regular Expression Inference
4. **Other Results**
5. Conclusion
Language Represented by a Pattern
Automaton is Regular

Converting into the original alphabet we get

\[ I \rightarrow p_1 P_1 \]
\[ I \rightarrow p_2 P_2 \]
\[ P_1 \rightarrow p_1 P_1 \]
\[ P_2 \rightarrow p_2 P_2 \]
\[ P_1 \rightarrow p_2 P_2 \]
\[ P_2 \rightarrow p_1 P_1 \]
\[ P_1 \rightarrow \varepsilon F \]
\[ P_2 \rightarrow \varepsilon F \]

\[ I \rightarrow aba P_1 \]
\[ I \rightarrow a P_2 \]
\[ P_1 \rightarrow aba P_1 \]
\[ P_2 \rightarrow a P_2 \]
\[ P_1 \rightarrow a P_2 \]
\[ P_2 \rightarrow aba P_1 \]
\[ P_1 \rightarrow \varepsilon \]
\[ P_2 \rightarrow \varepsilon \]
Language Represented by a Pattern Automaton is not always $k$ testable

Regular expression $= (aaa + bbb)^*$.

If $k = 3$, $I_k = \{a, aa, b, bb\}$, $F_k = \{aaa, bbb\}$. $T_k = \{aba, aab\}$. Generates strings of the form $abbabb$, $\notin (aaa + bbb)^*$.
**Outline**

1. Introduction
2. Problem
3. Regular Expression Inference
4. Other Results
5. **Conclusion**
   - References
We have shown here how to infer a regular expression using a pattern automaton.

We have shown properties of language represented by the pattern automaton in relation to the k testable languages. We also showed that the language represented by this automaton is always regular and that they need not always be k testable.
REFERENCES


● G J Bex, F Neven, T Schwentick and K Tuyls. Inference of concise DTDs from XML data. VLDB 2006.


Thank You