

Binary Queries

Author:

- First Name: Alexandru
- Family Name: Berlea
- Job Title: Research Assistant
- Address:
 - Affiliation: University of Trier
 - Sub-affiliation: Department of Computer Science
 - City: Trier
 - Country: Germany
 - E-mail: aberlea@psi.uni-trier.de
 - Web: <http://www.informatik.uni-trier.de/~aberlea/>
- Biography:

Alexandru Berlea is a Ph.D. Student working with Helmut Seidl at the University of Trier, Germany. He has graduated in 1999 from the Computer Science and Engineering Department of the Politehnica University of Bucharest, Romania.

Author:

- First Name: Helmut
- Family Name: Seidl
- Job Title: Professor
- Address:
 - Affiliation: University of Trier
 - Sub-affiliation: Department of Computer Science
 - City: Trier
 - Country: Germany
 - E-mail: aberlea@psi.uni-trier.de
 - Web: <http://www.informatik.uni-trier.de/~seidl/>
- Biography:

Helmut Seidl is a Professor with the University of Trier, Germany.

Keywords: XML querying, XML pattern matching, Binary queries, XML pattern matching with regular expressions, Forest grammars, Forest automata, XML transformations, XSLT, XPath, fxgrep.

Abstract:

Querying XML documents is a basic task for many XML applications. Typically, matches of queries are located individually, i.e. a match is a single node in a queried document. *k-ary* queries, in contrast, simultaneously identify *k* locations in the document, which are connected via some specified relation. For example, in rule-based transformations (like XSLT transformations), a node to which a rule is applied and another node which is selected by the rule in the context of the first node could be identified together by a binary query defining the relation between these two nodes. Such a binary query would replace the match and the select pattern used to identify the first and the second node respectively. By eliminating the need for select patterns, binary queries would not only reduce the number of patterns in a transformation, but would also allow that all the pattern matching is done statically, i.e. before the transformation actually begins. We use the formalism of regular forest grammars to define queries of arbitrary arities. Pushdown forest automata have been used in the past to efficiently implement unary queries expressed by using forest grammars. The main contribution of this paper is an algorithm based on pushdown forest automata which evaluates binary queries. In the worst theoretical case the algorithm evaluates binary queries in time proportional to the square of the size of the input document. In practice however, the complexity is mostly rather linear in the input size. Queries for which the right context does not need to be checked can be evaluated even along with the parsing of the input document. Rather than using forest grammars, queries can be also expressed in a more intuitive pattern language using a syntax similar to XPath.

1. Introduction

Searching in documents for components with specific properties, is one of the most fundamental tasks in XML document processing applications. Various pattern languages have been designed for identifying properties of document components, see e.g. [FSW99][XPath1][fxgrep]. Besides the basic use of retrieving parts of XML documents, querying has been also used in XML transformation languages for referring to parts of XML documents manipulated by the transformation languages [XSLT1][fxt]. Clearly, in order to cover the wide range of possible applications, the language of XML query patterns should be both very expressive and also efficiently implementable.

Perhaps the most widely known language of query patterns is XPath [XPath1][XPath2]. The XPath patterns describe *unary queries*, i.e. they locate individual nodes in the input. The same is also true for the query language supported by fxgrep [fxgrep]. In this paper, we are going to lift this limitation. Generalizing the regular approach to querying of fxgrep, we propose a concept of *recognizable k-ary queries* and present techniques for the efficient implementation for the special case of *binary queries*. Recognizable *k-ary* queries simultaneously locate *k-tuples* of nodes in the input document which simultaneously satisfy a specific property.

In particular, the result of a binary query can be interpreted as a tabulation of relative links between parts of documents. This may be exploited in rule-based XML transformation languages. XSLT transformations for example [XSLT1][XSLT2], use unary queries, expressed through XPath patterns, for two purposes. The first purpose is to differentiate components of the input document requiring different processing, i.e. given a node, to identify the applicable rule for it. Patterns used for this purpose are called *match patterns*. Secondly, within a rule, patterns are used for selecting nodes relative to the node to which the rule applies for further processing. Patterns used for this purpose are called *select patterns*. The match patterns can be queried once before the actual transformation. In contrast, select patterns must be repeatedly queried during the transformation phase. Note also that the class of select patterns is stronger (and more difficult to implement) as these allow for arbitrary navigation in the document.

Our key application for binary queries is to eliminate the dynamic select queries by combining the unary match pattern of a rule and a select pattern within this rule into a (statically tabulated) binary query.

Example: Consider the following XML input document:

```
<company>
  <url>spice.girls</url>
  <empl><name>Mel A.</name></empl>
  <empl><name>Mel B.</name></empl>
  <empl><name>Mel C.</name></empl>
</company>
```

The following XSLT rule produces a `homepage` element for each employee:

```
<xsl:template match="company[url]/empl">
  <homepage>
    <body>Under construction.
      See the company's page:<link><xsl:copy-of select="../url"/></link>
    </body>
  </homepage>
</xsl:template>
```

A binary match could simultaneously locate an employee and the `url` of her company. Let us suppose that binary queries were possible in XPath. Let the second element of a binary match be specified by preceding the corresponding node in the pattern with the `%` symbol, and referred within the rule by using the same symbol. The rule above could then be expressed as follows:

```
<xsl:template match="company[%url]/empl">
  <homepage>
    <body>Under construction.
      See the company's page:<link>%</link>
    </body>
  </homepage>
</xsl:template>
```

In the remainder of this paper we show that an expressive class of binary queries can be evaluated efficiently. Many select queries relative to the dynamic current context can be collected into one binary query whose evaluation can be performed statically, i.e., preceding the transformation of the input document.

XSLT keys contribute one special case of binary matches. Basically, a key is a pair consisting of the node which has the key and the value of the key (a string). The

node is identified using a match pattern, while the value is given by a select pattern evaluated in the context of the node. Thus, the binary matching algorithm presented here can also be used to implement the XSLT keys.

The latest drafts of XPath [XPath2] and XQuery [XQuery1] provide a *for* and a *FLWR* expression, respectively, which allow variables to be bound to nodes which are matches of unary queries. These nodes can be used in the scope of the expressions as context for the evaluation of further unary queries. This use of *for* expressions also qualifies for an implementation which uses binary queries to subsume the two unary queries.

2. Preliminaries

XML documents are textual representations of sequences of trees which we call forests. As usual, the *trees* t and *forests* f over an alphabet Σ are defined as:

- $t ::= \langle a \rangle f \langle /a \rangle, a \in \Sigma$
- $f ::= t_1 \dots t_n, n \geq 0$

The definition naturally captures XML elements without attributes and having as sub-trees only element nodes. The definition captures, though, also realistic XML elements. Text nodes can be represented as special elements containing an empty sub-element whose name is the respective text. Processing instructions can be represented as special elements containing two text nodes, one for the processor part and one for the instruction part. Finally, attributes can be collected under a special element as pairs of text nodes containing the name and the value for each attribute.

As in XML, we abbreviate $\langle a \rangle \langle /a \rangle$ to $\langle a / \rangle$. Given a tree $t = \langle a \rangle f \langle /a \rangle$, we say that a is the *label* of t or that t is labeled by a . We write this as $label(t) = a$.

An important task in document processing consists in verifying a structural property of a document tree. For XML, various schema languages have been proposed for this purpose. Besides the document-type declaration of a document, people have proposed, e.g., XML-Schema [XSchema], DSD [DSD], RelaxNG [RelaxNG]. In essence, all these formalisms specify *regular forest languages*. For a discussion on this see [Taxonomy]. Regular forest languages constitute a very expressive and theoretically robust formalism for specifying properties of forests [BW98][Mur95][BKW01]. Validating against one of these XML schema languages is therefore a test for membership in a forest regular language. Forest regular languages can be specified using *forest grammars*.

Definition: A *forest grammar* over Σ is a tuple $G = (X, r_0, R)$, where X is a set of *variables*, r_0 , a regular expression over variables, is the *start expression*, and R is a finite set of *rules* of the form $x \rightarrow \langle a \rangle r \langle /a \rangle$ with $x \in X$, $a \in \Sigma$ and r a regular

expression over variables.

Example: Consider the grammar $G = (\{x_T, x_1, x_a, x_b, x_c\}, (x_1|x_a), R)$ over $\{a, b, c\}$ with the following rules:

1. $x_T \rightarrow \langle a \rangle x_T^* \langle /a \rangle$
2. $x_T \rightarrow \langle b \rangle x_T^* \langle /b \rangle$
3. $x_T \rightarrow \langle c \rangle x_T^* \langle /c \rangle$
4. $x_1 \rightarrow \langle a \rangle x_T^* (x_1|x_a) x_T^* \langle /a \rangle$
5. $x_a \rightarrow \langle a \rangle x_b x_c \langle /a \rangle$
6. $x_b \rightarrow \langle b \rangle x_T^* \langle /b \rangle$
7. $x_c \rightarrow \langle c \rangle x_T^* \langle /c \rangle$

This grammar describes documents in which there is a path from the root to an a node whose children are a b and a c node, and all the nodes on this path are labeled a .

The first three rules make x_T account for trees with arbitrary content. As specified by rule number 5, x_a stands for the a element with a b and a c child. Rules 6 and 7 say that these children can have arbitrary content. Finally, rule 4 specifies that the a node specified by 5 can be at arbitrary depth in the input, and all its ancestors must be labeled a .

A grammar is the formal specification of the set of forests which can be *derived* from a variable string conforming to r_0 , by using rules from R in a number of steps. In each step, a variable symbol x for which a rule $x \rightarrow \langle a \rangle r \langle /a \rangle$ exists is replaced by $\langle a \rangle w \langle /a \rangle$, where w is a variable string conforming to r .

In the examples throughout the rest of the paper we will use the grammar G from above and the following input document:

```
<a>
  <a>
    <b/>
    <c/>
```

```
</a>  
<a>  
  <b/>  
</a>  
<a>  
  <b/>  
  <c/>  
</a>  
</a>
```

The tree-like representation of the input document is depicted in [\[tree\]](#)

The tree representation of the input document

The possible derivations of the input under G are presented in [\[derivations\]](#). Note that a derivation annotates each node of the input with a variable x from X .

Derivations of the input document

3. Queries

3.1. Unary queries

Patterns for unary queries, conceptually, consist of two parts. The contextual part constrains the surrounding context of the sub-trees of interest, whereas the structural part describes properties of the sub-trees themselves. In [\[NeumannThesis\]](#), forest grammars have been used to specify both the contextual and the structural part of unary queries. The idea is that a variable x of a grammar is used to characterize the set of trees which can be generated from x . Thus, given a grammar G , we may specify the structural part of a pattern simply through a variable x . In order to capture the context, we use the grammar itself which (given a variable for the whole document) constrains the places where the variable x may occur in derivations.

Thus, a unary query is a pair $Q = (G, X_{targets})$ consisting of a forest grammar $G = (X, r_0, R)$ and a set of target variables $X_{targets} \subseteq X$. A sub-tree t of an input forest f is a match for Q if and only if there is a derivation of f with respect to G in which the root of t is labeled with a symbol from $X_{targets}$.

Example: The query $Q_1 = (G, \{x_a\})$ locates the a nodes having only a ancestors and

as children, a b followed by a c . In our input document, these are the leftmost and the rightmost children of the root element. One can see that both are indeed labeled with x_a by the first and the second derivation of the input document, respectively.

In [NeumannSeidl], a new class of automata, pushdown forest automata, has been introduced which allows the efficient implementation of such unary queries [NeumannThesis][NeumannSeidl] (see below).

3.2. k -ary Queries

In this paper we extend the notion of unary queries as presented in the last sub-section to k -ary queries for $k \geq 1$. The idea is very simple. Instead of using a set of target variables, we now have a set of k -tuples of variables. Thus, a k -ary query is a tuple $Q = (G, T)$ consisting of a forest grammar $G = (X, r_0, R)$ and T a set of k -tuples of variables $(x_{target1}, \dots, x_{targetk}) \in X^k$.

Given a query Q and an input forest f , the matches of the query Q are the k -tuples of trees from f , for which a tuple of variables exists in T , such that there is one derivation which labels the root of each tree in the tuple of trees with the corresponding variable in the tuple of variables.

Example: The query $Q_2 = (G, \{(x_b, x_c)\})$ locates the pairs of b and c nodes having as father the same node a , and only a ancestors. The leftmost b and c nodes of our input form a match, as they are labeled with x_b and x_c , respectively, by the first derivation of the input (see [derivations]). Similarly, the rightmost b and c form a match, as it can be seen in the second derivation.

4. Accepting Regular Forest Languages - Forest automata

Various classes of automata have been introduced for the implementation of forest grammars and querying.

4.1. Bottom-up forest automata

Classically, the bottom-up finite-state forest automata have been used for accepting regular forest languages. The processing model of these automata is a bottom-up forest traversal as illustrated in [bottomup]. A bottom-up forest automaton has two types of states, *tree states* and *forest states*, and two types of transitions, *Side* and

Up. In order to assign a state to a tree $t = \langle a \rangle t_1 \dots t_n \langle /a \rangle$, a tree state p_i is first assigned to each t_i . Then an initial forest state q_1 is chosen from the set of all possible initial forest states. By traversing the word $p_1 \dots p_n$ from left to right and performing a side-transition at each step, a forest state q_{n+1} is obtained from q_1 . An up-transition for q_{n+1} and a yields a tree state p for t .

The processing model of a bottom-up forest automaton

Given a forest grammar G , a non-deterministic bottom-up automaton can be constructed which accepts the forest language defined by the grammar. Note that the implementation of a forest automaton however, requires that the automaton is made deterministic. Indeed, for every non-deterministic bottom-up forest automaton, an equivalent deterministic automaton can be obtained by a subset construction. This construction, however, may lead to an exponential blowup of the number of states of the automaton.

4.2. Pushdown Forest Automata

A class of forest automata equally expressive as the bottom-up forest automata, but much more concise and efficient to implement are the pushdown forest automata as introduced by Neumann and Seidl [NeumannSeidl]. These automata combine features of bottom-up and top-down automata. Besides *Side* and *Up* transitions as in the case of bottom-up automata, pushdown automata also have a transition relation *Down*. The processing model of a pushdown forest automaton is depicted in [pushdown]. A pushdown automaton visits the nodes of a document in the same order in which they are encountered by an XML parser. When arriving at some node a with a forest state q , the *Down* transition is used to select a *sensible* initial forest state q_1 for the traversal of the children of the node. q_1 is chosen only if it can lead to a successful *Up* transition followed by a successful *Side* transition from the node a .

The processing model of a pushdown forest automaton

A left-to-right pushdown forest automaton $A = (P, Q, I, F, Down, Up, Side)$ consists of a set of tree states P , a set of forest states Q , a set of start states $I \subseteq Q$, a set of final states $F \subseteq Q$, an up-relation $Up \subset Q \times \Sigma \times P$, a side-relation $Side \subset Q \times P \times Q$ and a down-relation $Down \subset Q \times \Sigma \times Q$.

Based on $Down$, Up and $Side$, the behavior of A is described by the relations $\delta_F \subset Q \times F \Sigma \times Q$ and $\delta_T \subset Q \times T \Sigma \times P$ as follows:

- $(q, \varepsilon, q) \in \delta_F$ for all $q \in Q$
- $(q_1, ft, q_2) \in \delta_F \Leftrightarrow (q_1, f, q) \in \delta_F$, $(q, t, p) \in \delta_T$ and $(q, p, q_2) \in Side$ for some $q \in Q$, $p \in P$
- $(q, a\langle f \rangle, p) \in \delta_T \Leftrightarrow (q, a, q_1) \in Down$, $(q_1, f, q_2) \in \delta_F$ and $(q_2, a, p) \in Up$ for some $q_1, q_2 \in Q$

The language accepted by an automaton A is $L_A = \{f \mid (q_1, f, q_2) \in \delta_F \text{ for some } q_1 \in I, q_2 \in F\}$.

[NeumannThesis] shows that deterministic and non-deterministic pushdown forest automata are equally expressive. Given a grammar G , with the notations from the previous sections, a deterministic pushdown forest automaton A_G , accepting the set of forests specified by G can be constructed as follows: Let $G = (X, r_0, R)$ and $\{r_1, \dots, r_l\}$ be the set of regular expressions different from r_0 occurring on the right-hand sides of rules in R . For each $j \in \{0, \dots, l\}$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the language of r_j as obtained by a Berry-Sethi construction [BS86]. Y_j is the set of NFA states, $y_{0,j} \in Y_j$ the initial state, $F_j \subseteq Y_j$ the set of final states, and $\delta_j \subseteq Y_j \times \Sigma \times Y_j$ the transition relation of A_j . Let $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \dots \cup Y_l$ and $\delta = \delta_0 \cup \dots \cup \delta_l$. Then $A_G = (2^X, 2^Y, \{q_0\}, F, Down, Up, Side)$ where:

- $q_0 = \{y_{0,0}\}$

- $F = \{q \mid q \cap F_0 \neq \emptyset\}$
- $Down(a, q) = \{y_{0,j} \mid y \in q, (y, x, y_1) \in \delta, x \rightarrow \langle a \rangle r_j \langle /a \rangle \text{ for some } x, y_1\}$
- $Up(a, q) = \{x \mid x \rightarrow \langle a \rangle r_j \langle /a \rangle \text{ and } q \cap F_j \neq \emptyset\}$
- $Side(q, p) = \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\}$.

Example: [\[berry\]](#) depicts the finite automata obtained by Berry-Sethi constructions for the regular expressions occurring in G . The start state of each automaton is marked with a thick point. The final states are depicted in grey. Note that the Berry-Sethi construction always yields an automaton in which for every state y , all the incoming transitions are labeled with the same symbol x .

The finite automata for the regular expressions in grammar G

The run of the A_G automaton on our input document is depicted in [\[firstpass\]](#)

The run of A_G on the input document

5. Querying XML Using Pushdown Forest Automata

The forest automata presented in the previous section can be used to test the membership of a forest to a language. Testing membership to a forest language essentially means testing the conformance to a schema or, in other words, verifying the structure of trees. For querying however, we need to be able to locate sub-trees which not only have a specified structure, but also occur in a specified context. Neumann and Seidl have successfully applied pushdown forest automata for answering unary queries [\[NeumannSeidl\]](#)[\[NeumannThesis\]](#)[\[fxgrep\]](#).

In order to explain their technique of query evaluation we need to relate the run of a pushdown automaton over a forest with the set of possible derivations of the forest. By comparing the run of the pushdown automaton in [\[firstpass\]](#) and the derivations in [\[derivations\]](#), the following observation can be made. If a derivation annotates a node with a variable symbol x , then x is contained in the tree state p for that node. The converse however does not hold. There are nodes whose tree state p contains some x , but such that no derivation exists in which the node is annotated with x . Consider for example the node b in the middle of the input document. Its tree state contains x_b , but no derivation labels this node with x_b . A tree state assigned by A_G to a node is thus a superset of the set of symbols x with which any possible derivations labels the node. The reason why these two sets may not be equal is that by the time the

pushdown automaton finishes visiting a node, it has not seen all the context of the node. More precisely, it has not seen the right context. In order to account for the right context, a second automaton must proceed from right to left.

5.1. Right-to-left Pushdown Automata

A deterministic right-to-left pushdown automaton $B_{\mathcal{G}}^{\leftarrow}$ runs on the input forest in which every node is annotated by the run of the first automaton $A_{\mathcal{G}}$ with the tree state $p \rightarrow$ synthesized for the node and the forest state $q \rightarrow$ obtained by the side transition at that node. $B_{\mathcal{G}}^{\leftarrow}$ operates on the same sets of NFA states as $A_{\mathcal{G}}$. The processing model of $B_{\mathcal{G}}^{\leftarrow}$ is depicted in [\[rtlpushdown\]](#). Note that we use the superscript \rightarrow to refer to the states of the left-to-right automaton $A_{\mathcal{G}}$.

The processing model of $B_{\mathcal{G}}^{\leftarrow}$

In contrast to $A_{\mathcal{G}}$, when descending to the children of a node, $B_{\mathcal{G}}^{\leftarrow}$ selects final instead of initial NFA states, and follows the NFA transitions in reverse, while traversing the children from right to left. Furthermore, when executing its *Down*, *Side* and *Up* transitions, $B_{\mathcal{G}}^{\leftarrow}$ only takes into account NFA states that were also reached by $A_{\mathcal{G}}$. *Up* computes a tree state for a node from the forest state q_1 computed for its children and, additionally as compared to $A_{\mathcal{G}}$, from the forest state q by which $B_{\mathcal{G}}^{\leftarrow}$ reaches the node.

$B_{\mathcal{G}}^{\leftarrow}$ is defined as $(P, Q, F_0, \{\}, \text{Down}, \text{Up}, \text{Side})$, where P , Q and F_0 are as in the definition of $A_{\mathcal{G}}$ and:

- $\text{Down}((a, p \rightarrow, q \rightarrow), q) = \{y_2 \mid y \in q \cap q \rightarrow, (y_1, x, y) \in \delta, x \rightarrow \langle a \rangle r_j \langle /a \rangle \text{ and } y_2 \in F_j\}$
- $\text{Up}((a, p \rightarrow, q \rightarrow), (q, q_1)) = \{x \mid x \rightarrow \langle a \rangle r_j \langle /a \rangle, y \in q_1, y = y_{0,j}, y_1 \in q \cap q \leftarrow, (y_2, x, y_1) \in \delta\}$.

- $Side_{\leftarrow}((a,p\rightarrow,q\rightarrow),(q,p)) = \{y_1 \mid (y_1,x,y) \in \delta, y \in q \cap q\rightarrow, x \in p\}$

The right-to-left automaton constructed above assigns to each node in the input a tree state p such that $x \in p$ if and only if there is a derivation which labels the node with x . Thus, accordingly to the definition of a unary query $(G, X_{targets})$, a node is a match if and only if $\exists x \in p$ and $x \in X_{targets}$.

Example: [secondpass] illustrates the run of $B_{G\leftarrow}$ on the input annotated by A_G . Note that the tree state assigned by $B_{G\leftarrow}$ to the middle node b contains now only x_T , the only symbol by which this node is labeled in both of the possible derivations.

The run of $B_{G\leftarrow}$ on the input document

6. Binary Queries

Let us now consider a binary query (G, T) . For simplicity, let us first assume that T equals the Cartesian product $X_{targets1} \times X_{targets2}$. Given a binary match (n_1, n_2) , we call n_1 *primary match*, and n_2 *secondary match*. The tree states of $B_{G\leftarrow}$ can be used to find all primary and all secondary matches. Primary matches are the nodes annotated with a tree state p such that $\exists x \in p, x \in X_{targets1}$. Similarly, secondary matches are the nodes annotated with some p such that $\exists x \in p$ and $x \in X_{targets2}$.

However, finding all the primary and all the secondary matches is not enough for binary query answering. In order to report a binary match we must be able to tell whether a primary match and a secondary match are defined with respect to the same derivation.

Example: Consider the XML input and the query Q_2 as above. By looking at the tree states of $B_{G\leftarrow}$ in [secondpass], one can not tell which primary match b node and which secondary match c node belong together to a binary match. More information is needed in the annotation made by the pushdown automata.

We enhance the second pushdown forest automaton as follows:

1. To each element y of a forest state q , we attach two lists L_1 and L_2 of primary and secondary matches, respectively. Whenever we reach a node in a forest state q and $(y, L_1, L_2) \in q$, then L_1 contains all primary matches seen so far which may occur in a derivation which annotates the present node with x , where x is the label of the incoming NFA transitions to the y state. Analogously for L_2 .
2. Also, to each element x of a tree state p , we attach two lists L_1 and L_2 of primary and secondary matches, respectively. Whenever we reach a node in a tree state p and $(x, L_1, L_2) \in p$, then L_1 contains all primary matches seen below (or at) the node, which may occur in a derivation which annotates the present node with x . Analogously for L_2 .

The new pushdown automaton $B_G^{2\leftarrow}$ over $\Sigma \times P \times Q$ is defined as follows: $B_G^{2\leftarrow} = (P, Q, I, \downarrow, \uparrow, \text{Down}, \text{Up}, \text{Side})$, where, with the same notations as in the definition of the ordinary right-to-left automaton $B_{g\leftarrow}$:

- Initially, all lists are empty:
 $I = \{(y_0, [], []) \mid y_0 \in F_0\}$
- When going down, we initialize the lists for matches below as empty lists:
 $\text{Down}((a, p \rightarrow, q \rightarrow), q) = \{(y_2, [], []) \mid (y, L_1, L_2) \in q, y \in q \rightarrow, (y_1, x, y) \in \delta, x \rightarrow \langle a \rangle r_j \langle /a \rangle, \text{ and } y_2 \in F_j\}$
- When going up, we propagate the found matches possibly adding a match of the

current node n

$$Up_n((a,p \rightarrow, q \rightarrow), (q, q_1)) = \{(x, L_1, L_2) \mid$$

1. $x \rightarrow \langle a \rangle r_j \langle /a \rangle, (y, l_1, l_2) \in q_1, y = y_{0,j}, (y_1, l_1', l_2') \in q, y_1 \in q \rightarrow, (y_2, x, y_1) \in \delta$
2. if $x \in X_{targets1}$ then $L_1 = l_1 + [n]$ else $L_1 = l_1$; if $x \in X_{targets2}$ then $L_2 = l_2 + [n]$ else $L_2 = l_2$ }

- At side transitions, we combine the lists from the subtree below with the matches from the already visited part to the right:

$$Side((a,p \rightarrow, q \rightarrow), (q, p)) = \{(y_1, L_1, L_2) \mid$$

1. $(y_1, x, y) \in \delta, y \in q \rightarrow, (y, l_1, l_2) \in q, (x, l_1', l_2') \in p$
2. $L_1 = l_1 + l_1'; L_2 = l_2 + l_2'$ }

Given the construction above, matches are detected at side transitions of the automaton. For every NFA transition (y_1, x, y) considered at a side transition, every primary match associated with x (from list l_1') is paired with every match associated with y (from list l_2). Every such pair is a binary match. We proceed similarly for secondaries at x (from list l_2') and primaries at y (from list l_1). Binary matches of the form (n, n) are detected at Up transitions for $x \in X_{targets1} \cap X_{targets2}$.

Algorithm: Answering binary queries

- Input: binary query $Q = (G, X_{targets1} \times X_{targets2})$, input forest f
- Output: set of binary matches M defined by the binary query in f

```
function match(Q, f) {
  f1 := annotation_of_f_by_the_left-to-right_pushdown_automaton(Q, f)
  (q, M) := deltaF(f1, I, {})
  return M
}

function deltaF(f, q, M) {
  case f of
    emptyForest => return (q, M)

  | forest f1 followed by tree <(a, pl, ql)>f2</(a, pl, ql)> =>
    /*pl and ql are the annotations of the left-to-right automaton*/
    (p, M) := deltaT (<(a, pl, ql)>f2</(a, pl, ql)>, q, M)
    foreach (y, l1, l2) in q with y in ql {
      foreach (x, l1', l2') in p with (y1, x, y) in delta {
```

```

    foreach m1 in l1 {
      foreach m2 in l2' {
        M := M ∪ (m1,m2)
      }
    }
    foreach m1 in l1' {
      foreach m2 in l2 {
        M := M ∪ (m1,m2)
      }
    }
  }
}
q' := Side((a,p1,q1),(q,p))
return deltaF(f1,q',M)
}

function deltaT(<(a,p1,q1)>f</(a,p1,q1)>,q,M) {
  qn := Down((a,p1,q1),(a,q))
  (q1,M) := deltaF(f,qn,M)
  p := Up((a,p1,q1),(q,q1))
  if ∃ (x,l1,l2) ∈ p and x ∈ Xtargets1 ∩ Xtargets2 then
    M := M ∪ (<a>f</a>,<a>f</a>)
  return (p,M)
}

```

Example: [\[matching\]](#) shows the run of $B_G^{2\leftarrow}$ on our input. The nodes have been numbered in order to be able to refer them in the list of matches. Elements (x,l_1,l_2) and (y,l_1,l_2) are represented as and , respectively. $(x,[],[])$ and $(y,[],[])$ are represented as `x` and `y`, respectively.

The run of $B_G^{2\leftarrow}$ on the input document

The proof of the correctness of the algorithm is sketched in the Appendix section.

In order to assess the complexity of our algorithm given some input document t , let $|t|$ be the size of the input, s the maximum of the number of primary and the number of secondary matches in t , and r the number of binary matches. The annotation of the input by the left-to-right automaton has the complexity $O(|t|)$. The second automaton visits each node of the document once. For each node, it recomputes for each component of the state the two lists L_1 and L_2 . This amounts to time $O(s)$. Overall this contributes time $O(s \times |t|)$. Furthermore, the second pass has to dump the binary matches. As each match pair is considered only a constant number of times, the overall cost for dumping amounts to $O(r)$. Altogether this gives the time complexity $O(r + s \times |t|)$.

Note that $r \leq |t|^2$ and $s \leq |t|$. The theoretical worst therefore is $O(|t|^2)$. In practice however, it is likely that in most cases the number of primary, secondary and binary matches is unimportant as compared to the input size. Then the complexity of the algorithm is rather linear than quadratic.

7. Extensions

7.1. General Binary Queries

In the algorithm from the last section, we assumed that the relation T of the binary

query $Q = (G, T)$ is a Cartesian product. Let us briefly sketch how the algorithm can be generalized also to arbitrary binary relations T .

In this general case we simply decompose T into the union of binary Cartesian products T_1, \dots, T_m . Now we may successively run our algorithm for each of the T_i and collect all the matches. Or, more efficiently, we just perform one left-to-right pass and then execute the m right-to-left passes corresponding to the T_i strictly in parallel. Note that this extension increases the run time only by a constant factor.

7.2. Extension to Larger k -s

The idea of our algorithm for binary queries can be generalized to k -ary queries. For that, we modify the second pass by maintaining not just lists for primary and secondary matches. Instead, we maintain separate lists for each non-empty subset $A \subseteq \{1, \dots, k\}$ of at most $k-1$ elements. The list for A then contains all tuples of subtrees which form a partial match corresponding to the indices in A .

As the complexity of the resulting algorithm grows exponentially in k , we have refrained from implementing this generalization.

7.3. One-pass Matching

A large class of queries can be answered even more efficiently than by the algorithm presented so far. For many queries all matches can be found already during the first left-to-right pass. In particular, this holds for queries for which the right-context of the matches does not have to be verified. This can be checked prior to the query evaluation by analyzing the query [NeumannThesis]. Right-context-ignoring queries, both unary and binary, can therefore be answered in one single pass. Besides saving the second pass, this allows to search a document without constructing the document tree in memory. Moreover, the querying can be done along with the parsing of the document via an event-based parser.

8. Practical Aspects

We have implemented the algorithm presented above and used it to extend the `fxgrep` querying tool (<http://www.informatik.uni-trier.de/~aberlea/Fxgrep/>) to allow for XML binary queries.

The formalism of forest grammars as presented in this paper ignores many XML specific aspects. In order to obtain a practical querying tool, various practical aspects must be considered.

8.1. Text Nodes

XML elements can contain besides other elements also character data. In order to

deal with this, each segment of character data is seen as a single node. To allow text pattern-matching, the forest automata are enhanced with an external predicate for each text pattern. A special *Down* relation selects the text patterns to be checked for a text node, as those which might contribute to a succeeding side transition. The tree state for the text nodes are determined from the set of fulfilled predicates using a special *Up* relation.

8.2. Avoiding State Explosion

The pushdown automata are efficiently implemented by computing their transitions only as they are needed. Transitions which are not required for the traversal of the input are not computed. This avoids the computation of possibly exponentially large transition tables. The number of transitions that are actually computed is at most linear in the size of the input document. In practice only few transitions need to be computed even for large XML documents.

8.3. The Pattern Language

The queries specified using forest grammars can be very expressive, though their power is not easily exploitable by non-expert users. Therefore, our querying tool `fxgrep` allows to specify queries also by using a more intuitive pattern language.

The pattern language of `fxgrep` resembles in its syntax to XPath, but allows more precise specification of paths as well as of the context of nodes. Structural conditions for a node may be expressed by using regular expressions over the children of the node. Structural conditions are given between brackets following the node to which they refer. For example, the pattern $a[b^*c[d^*]]$ is fulfilled by an a element that has one or more b children followed by a c child that itself has only d children. Contextual conditions for a node may be specified as structural conditions for nodes lying on the path from the root to that node. For example $//a[\#c]/b$ identifies b nodes which have only one right sibling c and whose father is an a node. The $\#$ here is a placeholder for the tree in which the matches are to be found. Furthermore, paths can be also specified as regular expressions. For example, $(a/)^+b$ identifies a b node, where each ancestor (at least one) is an a node. The unary example query Q_1 from above can be specified as $(a/)^+a[\#c]/b$.

In our enhancement of `fxgrep`, we also extended this base pattern language to allow the specification of binary queries. In order to make this extension as simple and

intuitive as possible, we just provide one extra symbol $\%$ which may be placed anywhere inside the pattern to indicate the secondary match position. Thus, the binary query Q_2 can be expressed as $(a/)+a[\#\%c]/b$.

Consider the unary query `//book[_ (author/"escu$") _]/title` ($_$ is a wild-card matching an arbitrary forest). The query locates all the book titles whose author's names end in "escu". The binary query to simultaneously report the titles as above and their authors is: `//book[_ (%author/"escu$") _]/title`.

9. Final Remarks

Queries on trees can be also specified as logical formulas. In [NS00] a unary query is specified as a formula in a fragment of the monadic second order logic. It is shown that an algorithm exists that evaluates the query in time linear in the input size. [Sch00] defines an extension of the first order logic in which queries of arbitrary arity can be expressed. It is shown that unary queries can be also theoretically evaluated in linear time. Binary queries however need evaluation time cubic in the input size.

In this paper we showed how regular forest grammars can be used to specify XML queries of arbitrary arities. We presented how unary queries can be efficiently implemented using pushdown forest automata. The main contribution of the paper is an efficient algorithm for evaluating binary queries. Furthermore, we indicated how binary queries can increase expressivity and efficiency of XML transformation and querying languages.

10. Appendix

Proof of correctness of the algorithm: (sketch)

Let us consider an NFA transition (y_1, x, y) tracked by the right-to-left pushdown automaton at one of its *Side* transitions at a node n . Let L_1 be the list of primary matches associated with x and L_2 be the list of secondaries associated with y .

According to the invariants maintained by the list constructions (see the section [binary]), for every $n_1 \in L_1$ there exists a derivation D_1 such that

- (1): n_1 is a primary match with respect to D_1
- (2): D_1 labels n with x

Similarly, for every $n_2 \in L_2$ there exists a derivation D_2 such that

- (3): n_2 is a secondary match with respect to D_2
- (4): D_2 labels n with x

From the definition of derivations, (2) and (4) we can construct a derivation D_3 which is obtained from D_2 by replacing the derivation steps for n from D_2 with those from D_1 .

Because of (1), n_1 is a primary match with respect to D_3 . Because of (3), n_2 is a secondary match with respect to D_3 . Thus (n_1, n_2) is a binary match with respect to D_3 .

Bibliography

- Fxg02 Andreas Neumann, Alexandru Berlea. *fxgrep 1.4.5*. Source Code, 2002. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxgrep>.
- Fxp01 Andreas Neumann. *fxp 1.4.4*. Source Code, 2001. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxp>
- Fxt01 Alexandru Berlea. *fxt 2.0*. Online Documentation, 2001. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxt>.
- NS98 Andreas Neumann and Helmut Seidl. *Locating Matches of Tree Patterns in Forests* In V.Arvind and R.Ramamujan ,editors, Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS), volume 1530 of Lecture Notes in Computer Science, pages 134-145. Springer, Heidelberg, 1998
- NS00 Frank Neven and Thomas Schwentick. *Expressive and efficient pattern languages for tree-structured data*. In Proc. 19th Symposium on Principles of Database Systems (PODS 2000), pages 145-156, 2000.
- Sch00 Thomas Schwentick. *On diving in trees*. In Proc. 25th Symposium on Mathematical Foundations of Computer Science (MFCS 2000), pages 660-669, 2000.
- FSW99 M.Fernandez, J.Simeon and P.Wadler. *XML Query Languages: Experiences and exemplars*. Available online at <http://www-db.research.bell-labs.com/user/simeon/xquery.html>.
- Fxt02 Alexandru Berlea. *fxt 2.0*. Online Documentation, 2002. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxt>.
- Neu00 Andreas Neumann. *Parsing and Querying XML Documents in SML*. Ph.D. Thesis, University of Trier, Trier, 2000. Available online at <http://www.informatik.uni-trier.de/~neumann/Papers/thesis.ps.gz>.
- W3C99a James Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at <http://www.w3.org/TR/xslt>.
- W3C99b Michael Kay, editor. *XSL Transformations (XSLT) Version 2.0*. W3C Working Draft, World Wide Web Consortium, December 2001. Available online at <http://www.w3.org/TR/xslt20/>.

- W3C99c James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at <http://www.w3.org/TR/xpath>.
- W3C99d A.Berglund, S.Boag, D.Chamberlin, M.F.Fernandez, M.Kay, J.Robie, J.Sim#on, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at <http://www.w3.org/TR/xpath20/>.
- W3C99e S.Boag, D.Chamberlin, M.F.Fernandez, D.Florescu, J.Robie, J.Sim#on, M.Stefanescu, editors. *XQuery 1.0: An XML Query Language* W3C Working Draft, World Wide Web Consortium, December 2001. Available online at <http://www.w3.org/TR/xquery/>.
- W3C99f H.S.Thompson, D.Beech, M.Maloney and N.Mendelsohn, editors *XML Schema Part 1: Structures*. W3C Recommendation, May 2001. Available online at <http://www.w3.org/TR/xmlschema-1/>.
- KMS00 N.Klarlund, A.Moller and M.I.Schwartzbach. *DSD: A Schema Language for XML*. In ACM SIGSOFT Workshop on Formal Methods in Software Practice, Portland, USA, August 2000.
- Oas01 James Clark and Makoto Murata, editors. *RelaxNG Specification*, December 2001. Available online at <http://www.oasis-open.org/committees/relax-ng/>.
- MLM01 Makoto Murata, Dongwon Lee and Murali Mani. *Taxonomy of XML Schema Languages Using Formal Language Theory*, August 2001. In Extreme Markup Languages 2001.
- BW98 Anne-Bruggemann-Klein and Derick Wood. *Regular Tree Languages over Non-Ranked Alphabets*. Unpublished draft, April 1998. Available online at <http://www.oasis-open.org/cover/regTreeLanguages-ps.gz>.
- BW98 Anne-Bruggemann-Klein, Makoto Murata and Derick Wood. *Regular Tree and Regular Hedge Languages over Non-Ranked Alphabets*. Unfinished Research Report HKUST-2001-05, HKUST Theoretical Computer Science Center, April 2001. Available online at <http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz>.
- Mur95 Makoto Murata. *Forest Regular Languages and Tree Regular Languages*. Unpublished manuscript, 1995.
- BS86 Gerard Berry and Ravi Sethi. *From Regular Expressions to Deterministic Automata*. Theoretical Computer Science, 48, pages 117-126, 1986

Content

Introduction	2
Preliminaries	4
Queries	7
Unary queries	7
k-ary Queries	8
Accepting Regular Forest Languages - Forest automata	8
Bottom-up forest automata	8
Pushdown Forest Automata	9
Querying XML Using Pushdown Forest Automata	12
Right-to-left Pushdown Automata	13
Binary Queries	14
Extensions	18
General Binary Queries	18
Extension to Larger k-s	19
One-pass Matching	19
Practical Aspects	19
Text Nodes	19
Avoiding State Explosion	20
The Pattern Language	20
Final Remarks	21
Appendix	21