Journal of Universal Computer Science, vol. 16, no. 5 (2010), 833-851
submitted: 27/6/09, accepted: 12/2/10, appeared: 1/3/10 © J.UCS

Evaluating Linear XPath Expressions by Pattern-Matching Automata

Panu Silvasti
(Helsinki University of Technology, Finland
psilvast@cs.hut.fi)

Seppo Sippu
(University of Helsinki, Finland
sippu@cs.helsinki.fi)

Eljas Soisalon-Soininen
(Helsinki University of Technology, Finland
ess@cs.hut.fi)

Abstract: We consider the problem of efficiently evaluating a large number of XPath expressions, especially in the case when they define subscriber profiles for filtering of XML documents. For each document in an XML document stream, the task is to determine those profiles that match the document. In this article we present a new general method for filtering with profiles expressed by linear XPath expressions with child operators (/), descendant operators (//), and wildcards (∗). This new filtering algorithm is based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick pattern-matching automaton. This automaton has a size linear in the sum of the sizes of the XPath filters, and the worst-case time bound of the algorithm is much less than the time bound of the simulation of linear-size nondeterministic automata.
Our new algorithm has a predecessor that can handle child and descendant operators but not wildcards, and has been shown to be extremely efficient when a document-type definition (DTD) has been used to prune out all the wildcards and most of the descendant operators. But in some cases, such as when the DTD is highly recursive, it may not be possible to prune out all wildcards without producing a too large set of filters. Then it is important to have the full generality of an evaluation algorithm, as presented in this article, that can also handle wildcards.
Key Words: filtering of streams of XML documents, linear XPath expressions
Category: H.3.3

1 Introduction

In a publish-subscribe system based on XML filtering, subscribers usually specify their profiles by filters written in the XPath language. The system processes a stream of published XML documents and delivers to subscribers those documents that match the corresponding profiles. The number of subscribers can be large, thousands or even millions; thus the scalability of the filtering system is critical.

The primary problem addressed in this article is the *filtering problem* for XML streams: given a set of XPath expressions and a stream of XML documents, determine for each document in the stream those expressions that match the document. More specifically, we study the filtering problem for linear XPath expressions, that is, XPath expressions that do not have branches in their parse tree. Linear XPath expressions without predicates are defined by the following grammar:

$$path \rightarrow /\ step \mid //\ step \mid path\ path,$$
$$step \rightarrow label \mid *,$$

where *label* denotes an XML-element label.

Several approaches to XML filtering with XPath-defined profiles use a finite automaton as a basis of the filtering algorithm [Altinel and Franklin 2000, Diao et al. 2003, Green et al. 2004, Gupta and Suciu 2003, Onizuka 2003]. Diao et al. [2003] report an evaluation method called YFilter that evaluates nondeterministic finite automata (NFAs) constructed from the filters. YFilter uses a single NFA that combines the effect of the individual NFAs and achieves considerable improvements in performance by path sharing. In other words, YFilter merges states that correspond to common prefixes in different query paths, while still retaining the linear size of the NFA with respect to the filter descriptions.

Contrary to YFilter, which uses an NFA of a size linear in the total size of the filters, the algorithm of Green et al. [2004] is based on a deterministic finite automaton (DFA). The state explosion of the DFA is avoided by constructing the DFA lazily, as needed, while input documents are being filtered: if in processing the stream of XML documents, no next state is defined on the current input symbol, the corresponding new state will be computed and the process is continued at this new state. While exponential in the worst case, this approach works well in many cases, when the incoming XML documents obey a schema or DTD (see e.g. Kilpeläinen and Wood [2001]) that is non-recursive or contains only simple cycles (a cycle is simple if its nodes are not contained in other cycles).

The filtering methods outlined above are general in the sense that input documents do not need to obey a specific schema or DTD, but are only required to conform with the XML syntax. If it is known that the documents to be filtered are produced according to a DTD, a natural question is to ask whether or not the filtering process can be speeded up by using this knowledge. This question has been answered positively both in the case of YFilter [Silvasti et al. 2009a] and in the case of the lazy DFA construction [Silvasti et al. 2009b].

The optimization method of Silvasti et al. [2009a], called *filter pruning*, takes as input a DTD and a set of linear XPath filters and produces a set of "pruned" linear XPath filters. In the pruned filters, wildcards ($*$) and descendant operators ($//$) are replaced (or *pruned*) by single symbols and symbol strings, respectively, that are allowed in place of "$*$" and "$//$".

In this article we present yet another automaton-based general algorithm for filtering with profiles expressed by linear XPath expressions. This new filtering algorithm is based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick pattern-matching automaton [Aho and Corasick 1975]; it is called the *pattern-matching-automaton-based* (or *PMA-based*) *filtering algorithm*. The size of the PMA is linear in the sum of the sizes of the filters, and the worst-case time bound of the algorithm is much less than the time needed for simulating nondeterministic automata, which is the worst-case time bound of YFilter [Diao et al. 2003]. Thus, when the worst-case time bound is concerned, the PMA-based filtering is superior both to YFilter and to the lazy DFA algorithm [Green et al. 2004]; the latter has the worst-case space and time bound $\Omega(2^k)$, where k denotes the number of filters.

Our new algorithm and its predecessor [Silvasti et al. 2009a] work in the same way if only child and descendant operators are present. This implies that our new PMA-based filtering algorithm will be very efficient, in the same way as its predecessor [Silvasti et al. 2009a], when wildcards and descendant operators can be pruned out.

This article is organized as follows. In Section 2 we present the basics of automata-based filtering, and prove the optimal time complexity of the PMA-based filtering algorithm, when only the child operators in addition to a leading descendant operator are allowed. Section 3 is devoted to the analysis of a new algorithm of multiple-pattern matching when wildcards are allowed in the patterns. Algorithms are presented both for pattern matching of linear text and for filtering of XML documents. In Section 4 we present and analyze our algorithm that, using the results of the previous sections, solves the XML filtering problem for full linear XPath expressions (without predicates).

2 Automaton-Based Filtering

A linear XPath expression (without predicates) is defined as a sequence of XML-element labels or wildcards separated by child (/) or descendant (//) operators; in other words a sequence of the form

$$S = op_1\ l_1\ op_2\ \ldots\ l_{n-1}\ op_n\ l_n, \tag{1}$$

where each op_i is "/" or "//", and l_i is an XML-element label or a wildcard "$*$". A *filter* is a union of sequences of form (1).

The filtering problem can now be expressed as a language recognition problem over alphabet Σ, the set of XML element labels that possibly occur in the input documents. Let $F = \{f_1, \ldots, f_k\}$ be a set of filters f_i each of which is a union of sequences of form (1). Moreover, let $\hat{f}_i$ denote the union of languages $\hat{S}$ obtained from sequences S of the form (1) as follows: (i) substitute Σ^* for occurrences of

"//"; (ii) substitute Σ for occurrences of "$*$"; (iii) interprete occurrences of "/" as string-catenation operators (and hence omit); (iv) finally append Σ^* at the end of $\hat{S}$.

Then construct a DFA D that accepts the language $\hat{F} = \hat{f}_1 \cup \ldots \cup \hat{f}_k$ in such a way that, for each w in $\hat{F}$, D also reports all those filters f_i for which the language $\hat{f}_i$ contains w. To construct such a DFA is possible by simply marking final state q by index i whenever q was set as a final state because of the acceptance of a string in $\hat{f}_i$. However, such a DFA containing different types of final states becomes easily exponential in size because in the worst case there is a final state for each different subset of $\{\hat{f}_1, \ldots, \hat{f}_k\}$.

When processing an XML stream the input document will first be parsed by a SAX parser, which produces a parse tree from the document. In filtering, each path from the root to a leaf in this parse tree is fed to the DFA D, and at the end of the path the accepted filters are reported. It is not necessary to completely construct the parse trees, but we can perform the evaluation in conjunction of parsing. A stack of states need to be maintained in order to backtrack correctly when several paths have the same prefix.

SAX events are processed as follows. (See e.g. the article by Green et al. [2004] for a more detailed description.) When the beginning of an XML element is encountered, the current state q will be pushed onto the stack and the new current state will be the one accessed from q on the element label. When the end of an XML element is encountered, the state on top of the stack is popped and set as the new current state.

As an example, consider the set of three filters $f_1 = \{//a_1\}, f_2 = \{//a_2\}, f_3 = \{//a_3\}$. When considered as a language recognition problem, filtering with respect to these three filters can be done by a DFA that accepts the language $L = L_1 \cup L_2 \cup L_3$, where L_i denotes the language $\Sigma^* a_i \Sigma^*$, and decides by final states which of the three filters are matched. The minimized DFA is shown in Fig. 1.

At final state $i, 1 \leq i \leq 3$, only language L_i is recognized, at final states 4, 5 and 6, the sets $\{L_1, L_2\}$, $\{L_1, L_3\}$, and $\{L_2, L_3\}$, respectively, and at final state 7, the set $\{L_1, L_2, L_3\}$. Note that the DFA of Fig. 1 cannot be further minimized, because the final states all accept different subsets of $\{L_1, L_2, L_3\}$.

In general, $\Omega(2^k)$ states are included in the minimal DFA that solves the filtering problem for k (different) keyword sets, that is, each filter is a union of sequences of the form $//a_1/a_2/\ldots/a_p$, where $a_1 a_2 \ldots a_p$ is called a *keyword*. When filters are composed of keyword sets, the filtering problem can be solved efficiently by using the pattern-matching automaton by Aho and Corasick [1975].

The use of this automaton is based on the acceptance by an *output function*: If in a state q filter f_i is recognized (that is, string $\hat{f}_i$ is accepted at q), then *output*(q) is defined to contain i. An array *result*, indexed by filter numbers, is

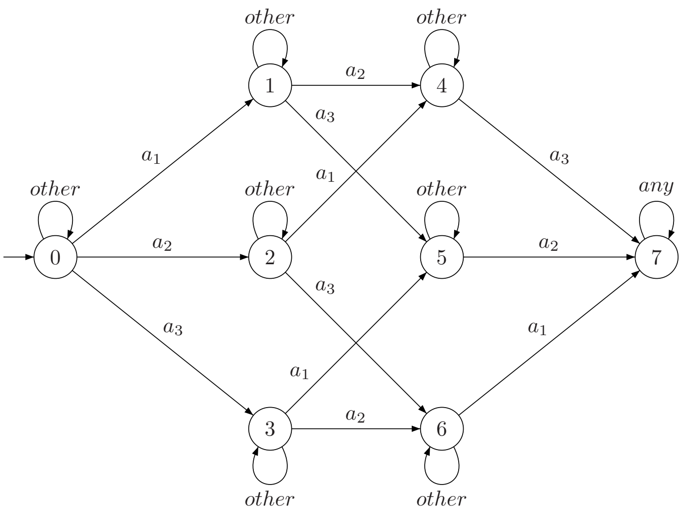

Figure 1: Minimal DFA that solves the filtering problem for filters $f_1 = \{//a_1\},\ f_2 = \{//a_2\},\ f_3 = \{//a_3\}$.

used to store information about matched keywords in the filters. If the pattern-matching automaton has recognized a keyword in filter f_i, then $\mathit{result}[i]$ is set to 1 (initially $\mathit{result}[i] = 0$ for all filters f_i). When the input document has been processed, the result of the filtering can be read from the array *result*.

The basic version of the pattern-matching automaton (PMA) as defined by Aho and Corasick [1975] is composed of the *goto* and *failure* functions that dictate the next-state transitions when processing the input. For each prefix y of some keyword, the PMA has a unique state, denoted $\mathit{state}(y)$, different from all $\mathit{state}(y')$, where $y' \neq y$. The state $\mathit{state}(\epsilon)$, where ϵ is the empty string, is the *initial state* of the PMA. Clearly, the number of states in the PMA is at most $||F|| + 1$, where $||F||$ denotes the size of the filter set F (composed of keyword sets), that is, the sum of the lengths of all keywords in F.

The *goto* function of the PMA is defined by the equation $\mathit{goto}(\mathit{state}(y), a) = \mathit{state}(ya)$, where ya is a prefix of some keyword and a is an element in Σ. The *fail* function of the PMA is defined by the equation $\mathit{fail}(\mathit{state}(uv)) = \mathit{state}(v)$, where uv is a prefix of some keyword and v is the longest proper suffix of uv such that v is also a prefix of some keyword. For nonnegative integer k, we denote by fail^k the fail function applied k times: $\mathit{fail}^0(q) = q$, and $\mathit{fail}^{k+1}(q) = \mathit{fail}(\mathit{fail}^k(q))$.

For any state q we denote by $\mathit{string}(q)$ the unique string y with $\mathit{state}(y) = q$. In our filtering application, the *output* function of the PMA is defined by setting for each state q:

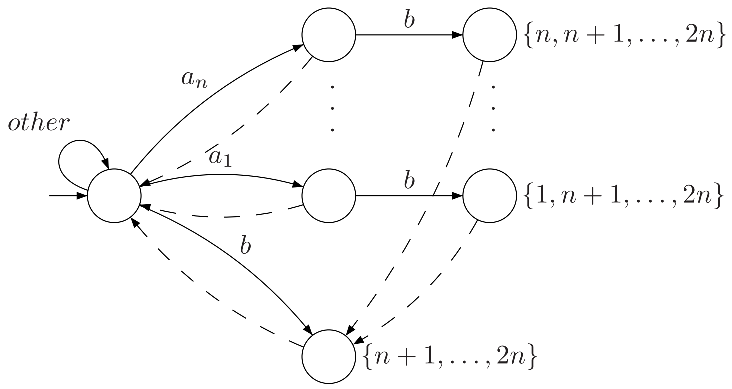

Figure 2: Pattern-matching automaton for the filters $f_1 = //a_1/b,\ \ldots,\ f_n = //a_n/b,\ f_{n+1} = //b,\ \ldots,\ f_{2n} = //b$.

$$\mathit{output}(q) = \{i \mid \text{a suffix of } \mathit{string}(q) \text{ is a keyword in filter } f_i\}. \quad (2)$$

A straightforward explicit implementation of the output function would make its size quadratic. Consider for example the filters

$$f_1 = //a_1/b,\ \ldots,\ f_n = //a_n/b,\ f_{n+1} = //b,\ \ldots,\ f_{2n} = //b.$$

The PMA for this filter set is depicted in Fig. 2. The size of the filter set is $3n$, but the size of the output function is $n^2 + 2n + 1$.

To circumvent this undesirable size growth we store the output sets as linked lists in the following way. First we define

$$\mathit{direct\_output}(q) = \{i \mid \mathit{string}(q) \in f_i\}, \quad (3)$$

and a way to reach all nonempty direct output sets without passing also the empty ones: The function *output_fail* is defined by setting for state q:

$$\mathit{output\_fail}(q) = \mathit{fail}^k(q),$$

if k is the greatest integer less than or equal to the length of $\mathit{string}(q)$ such that $\mathit{direct\_output}(\mathit{fail}^m(q))$ is empty for all $m = 1, \ldots, k-1$. We have:

Lemma 1.

$$\mathit{output}(q) = \cup\{\mathit{direct\_output}(q') \mid q' \in \mathit{output\_fail}^*(q)\},$$

and the representation of the output function as linked lists of direct output sets is of size $O(||F||)$.

Proof. A straightforward induction on the length of the path from state q consisting of the *output_fail* arcs. The representation of the output function as linked lists is obtained by linking the direct output sets together in the prescribed order. □

The PMA as constructed above can be used as a filtering machine exactly as a DFA. Because of the *fail* arcs (but only at most one per state), it must be prescribed that *fail* arcs are applied only when *goto* arcs cannot be applied. For input of size n at most $2n$ transitions can be performed.

When processing a stream of XML documents, the current state will be pushed onto the stack if a start-element tag is encountered and a *goto* arc on the element can be applied but not when a *fail* arc is used. When an end-element tag is encountered, it is discarded, the next element is scanned, and the state on top of the stack is taken as the new current state.

Theorem 1. Let $f_1, f_2, \ldots, f_k$ be filters each of which is composed of a set of keywords. Then the filtering problem for filters $f_1, f_2, \ldots, f_k$ can be solved in time $O(n+m)$, where n is the length of the input document (counted as the number of XML elements) and m is the sum of the sizes of the filters.

Proof. We construct the PMA from the keywords in $F = f_1 \cup \ldots \cup f_k$ as described above. If a keyword of f_i leads from the initial state to state q, the set *direct_output*(q) is set to include i. By the results of Aho and Corasick [1975] and by Lemma 1, this construction takes time linear in the size of F.

Whenever, during the processing of an input, state q with a nonempty output set is entered, *result*$[i] \leftarrow 1$ for all $i \in$ *output*(q), after which *output*(q) is set to empty. Once processed *output*(q) can be set to empty, because in the filtering application we do not want to determine all occurrences of keywords but only the first. Setting *output*(q) to empty is necessary, because otherwise we would not obtain the desired time bound $O(n+m)$ but we should add a term denoting the number of keyword occurrences.

When the process has finished, the question of whether or not the input x matches with filter f_i (that is, $x \in \hat{f}_i$) is answered by checking *result*$[i]$: $x \in \hat{f}_i$ if and only if *result*$[i] = 1$. □

As an example, Fig. 3 (a) shows the PMA obtained from the keywords of the filters $f_1 = //a_1$, $f_2 = //a_2$, $f_3 = //a_3$. This PMA is augmented with the output function *output*(q) that contains i if a_i has been read at q.

The failure transitions of the PMA can be eliminated by using the *next-move* function of a DFA in place of the *goto* and *fail* functions:

$$\textit{next-move}(q, a) = \textit{goto}(\textit{fail}^k(q), a), \tag{4}$$

where k is the least nonnegative integer for which $\textit{goto}(\textit{fail}^k(q), a)$ is defined, that is, the state $\textit{fail}^k(q)$ has a *goto* transition on element a.

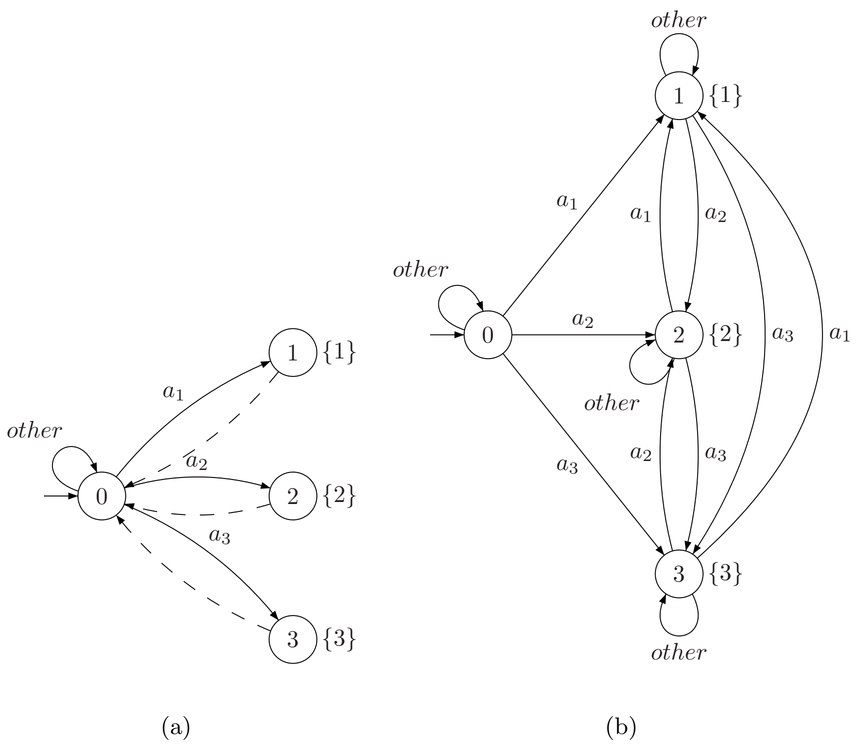

Figure 3: The PMA (a) and the corresponding DFA (b) that solve the filtering problem for $f_1 = //a_1$, $f_2 = //a_2$, and $f_3 = //a_3$, when filters are matched using the output function: $output(1) = \{1\}$, $output(2) = \{2\}$, and $output(3) = \{3\}$.

Fig. 3 (b) shows the DFA obtained by eliminating failure transitions from the PMA of Fig. 3 (a). Even though the DFA has exactly as many states as the PMA, the number of its (*next-move*) transitions is quadratic in the size of the keyword set.

If filters are composed of sequences of several keywords, so that they may also contain non-leading descendant operators "//", it is still possible to base the filtering algorithm on matching of keywords. For filter i that consists of a sequence of l keywords, a possible match has been found if $j \in result[i]$ for all $j = 1, \ldots, l$. Such a possible match is not always an exact match, because the matched keywords may not appear on the same path, nor in the specified order.

To find out which possible matches of multi-keyword filters are true matches, the input document must be filtered through some more general filtering algorithm, as through the one that will be presented in the following sections.

Another problem to be solved is that the multi-pattern matching algorithm of Aho and Corasick [1975] cannot handle keywords with wildcards. In the next section, we will present a new algorithm for multiple-pattern string matching when wildcards can appear in the patterns.

3 Multiple-Pattern Matching with Wildcards

It is typical that XPath expressions describing XML filters contain wildcards ($*$). As our intention is to derive a backtracking version of the multiple-pattern matching algorithm of Aho and Corasick [1975] for XML filtering, we first need to extend the basic algorithm to handle patterns with wildcards.

In Section 2 we considered the problem of matching filters that are keyword sets, but now we allow the filters to be composed of sets of sequences, called *patterns* (*with wildcards*), of the form

$$*^{n_1} w_1 *^{n_2} w_2 \ldots *^{n_k} w_k, \tag{5}$$

where $w_1, w_2, \ldots, w_k$ are keywords in Σ^+, $k \geq 1$, n_1 is a nonnegative integer, and $n_2, \ldots, n_k$ are positive integers.

In our algorithms, the length m_i of pattern P_i is denoted by *length*(i), and the number of keywords of pattern P_i is denoted by *#keywords*(i). For pattern P_i and the kth keyword w_k in P_i, *length*(i, k) gives the length of the keyword, and *distance*(i, k) gives the distance of the keyword from the beginning of P_i, that is,

$$\mathit{distance}(i,k) = \sum_{j=1}^{k-1} \mathit{length}(i,j) + \sum_{j=1}^{k} n_j,$$

where n_j is the length of the wildcard string preceding the jth keyword in P_i.

For example, for patterns

$P_1 = *ab{*}{*}{*}c$,
$P_2 = abc{*}{*}bab{*}b{*}b$

we have:

$\mathit{length}(1,1) = 2,\ \mathit{distance}(1,1) = 1,\ \mathit{length}(1,2) = 1,\ \mathit{distance}(1,2) = 6,$
$\mathit{length}(2,1) = 3,\ \mathit{distance}(2,1) = 0,\ \mathit{length}(2,2) = 3,\ \mathit{distance}(2,2) = 5,$
$\mathit{length}(2,3) = 1,\ \mathit{distance}(2,3) = 9,\ \mathit{length}(2,4) = 1,\ \mathit{distance}(2,4) = 11,$
$\mathit{length}(1) = 7,\ \mathit{length}(2) = 12.$

For solving the multiple-pattern matching problem with wildcards we construct the PMA as defined in Section 2 from the set of all keywords that appear in the patterns. This idea was previously used to solve the single-pattern matching problem [Pinter 1985] (also see the articles by Rahman et al. [2006] and Rahman and Iliopoulos [2007]). However, the algorithms presented in these articles cannot be directly extended to yield an efficient solution to the multiple-patterns matching problem.

The *output function* of the PMA is now defined as:

$$\mathit{output}(q) = \{(i, k) \mid \text{a suffix of } \mathit{string}(q) \text{ is the } k\text{th keyword of pattern } P_i\}.$$

A pair (i, k) in $\mathit{output}(q)$ is called an *output tuple* for state q; the tuple signals the recognition at state q of the kth keyword of pattern P_i; this keyword is a suffix of $\mathit{string}(q)$.

The basic idea of the algorithm is to collect, for each pattern P_i having more than one keyword, *partial matches* of P_i that represent matches of maximal prefixes of P_i found thus far. When a match up to and including the last keyword of P_i has been found, we have a full match of the pattern.

Partial matches of pattern P_i are stored in a set $\mathit{partial\_matches}(i)$ that contains pairs of the form (p, k) meaning that, starting at element position p in the input text, an occurrence of the prefix of P_i up to and including the kth keyword has been found. The algorithm simulates the PMA, and at state q, for all output tuples (i, k) with $\#\mathit{keywords}(i) > 1$, the set $\mathit{partial\_matches}(i)$ is updated:

(i) If $k = 1$, a new possible start of P_i is recorded by inserting $(p, 1)$ into $\mathit{partial\_matches}(i)$, where p is the element position of the start of the wildcard string preceding the keyword $\mathit{string}(q)$ recognized at state q, in other words,

$$p = \mathit{element\_count} - \mathit{length}(i, 1) - \mathit{distance}(i, 1),$$

where $\mathit{element\_count}$ gives the number of elements scanned thus far.

(ii) If $k > 1$ and $\mathit{partial\_matches}(i)$ contains $(p, k - 1)$ with

$$p = \mathit{element\_count} - \mathit{length}(i, k) - \mathit{distance}(i, k),$$

then this partial match $(p, k - 1)$ obtained thus far can be extended to also include the kth keyword. This is done by replacing $(p, k - 1)$ by (p, k) in the set $\mathit{partial\_matches}(i)$ (when $k < \#\mathit{keywords}(i)$) or by deleting $(p, k - 1)$ (when $k = \#\mathit{keywords}(i)$)).

When k reaches $\#\mathit{keywords}(i)$, then, instead of recording a partial match (p, k) in $\mathit{partial\_matches}(i)$, we record a full match of pattern P_i by inserting p into the set $\mathit{matches}(i)$.

In order to maintain efficiently the sets $\mathit{partial\_matches}(i)$, we store them as balanced binary search trees. In each set $\mathit{partial\_matches}(i)$ there is at most one pair (p, k) for any given p, and thus we may use p as a unique search key for the structure. Moreover, we will also keep a separate finger always pointing to the smallest element of the structure.

When searching the structure in order to replace pair $(p, k-1)$ by pair (p, k) we also check the element (p', k') pointed to by the finger. If here $p' < \mathit{element\_count} - \mathit{length}(i)$, we know that (p', k') can never be extended to a full match and can therefore be deleted. The deletion is repeated until the finger points to an element (p', k'), where $p' \geq \mathit{element\_count} - \mathit{length}(i)$. In this way we can keep the sets $\mathit{partial\_matches}(i)$ small, namely, containing at most $\mathit{length}(i)$ elements.

The operation cycle of the PMA is presented as Algorithm 1, and the procedure *check_output* that maintains the sets $\mathit{partial\_matches}(i)$ is presented as Algorithm 2.

Theorem 2. Let $S = \{P_1, \ldots, P_r\}$ be a set of patterns with wildcards and let T be a linear input text of length n. The set of all occurrences of the patterns of S in the text T can be computed in time

$$O(n + m + \sum_{i=1}^{r} \alpha_i \log |P_i|),$$

where m is the sum of the lengths $|P_i|$ of the patterns P_i in S, α_i denotes the number of occurrences of keywords of P_i in T, and the term $O(m)$ represents the time spent on preprocessing the patterns. Here the occurrences of pattern P_i are represented by element positions p, and the occurrences of the kth keyword of P_i by pairs (p, k), where p is the position in T of the first element of the occurrence.

The occurrences of the same patterns in any new text T' of length n' can be found in time

$$O(n' + \sum_{i=1}^{r} \alpha'_i \log |P_i|),$$

where α'_i denotes the number of occurrences of keywords of P_i in T'.

Proof. We construct the PMA for the set of all keywords that appear in the patterns in S as described above. By construction, this PMA is of size $O(m)$ and can be constructed in time $O(m)$.

We use Algorithm 1 to simulate the PMA. At each state q all output tuples (i, k) such that the kth keyword of P_i is a suffix of $\mathit{string}(q)$ will be checked for partial matches: if the set $\mathit{partial\_matches}(i)$ contains $(p, k-1)$ meaning that the prefix of P_i including the $(k-1)$th keyword has been found, $(p, k-1)$ is replaced by (p, k). Now if a whole match of P_i is obtained, p is inserted into the set $\mathit{matches}(i)$.

Each set $\mathit{partial\_matches}(i)$ can contain at most $|P_i|$ elements, because for each p there can only be one pair (p, k) and because all elements (p', k') with $p' < \mathit{element\_count} - |P_i|$ will be deleted in conjunction with possible updating of the set. When $\mathit{partial\_matches}(i)$ is organized as a balanced binary search tree indexed by unique key p, we conclude the bound $O(\alpha_i \log |P_i|)$, where α_i denotes the number of occurrences of keywords of P_i, for the total number of checks in set $partial\_matches(i)$.

Altogether we conclude the claimed time bound $O(n+m+\sum_{i=1}^{r}\alpha_i \log|P_i|)$. The time bound $O(n' + \sum_{i=1}^{r}\alpha'_i \log|P_i|)$ for any new text T' comes from the fact that the $O(m)$ time for constructing the PMA is not needed. □

Algorithm 1 Operating cycle of the PMA for matching linear text with sets of patterns containing wildcards.

for all patterns P_i **do**
  $matches(i) \leftarrow$ empty
  $partial\_matches(i) \leftarrow$ empty
end for
$state \leftarrow initial\_state$
$element\_count \leftarrow 0$
$scan\_next(element)$
while *element* was found **do**
  $element\_count \leftarrow element\_count + 1$
  while $goto(state,\ element) = fail$ **do**
    $state \leftarrow fail(state)$
  end while
  $state \leftarrow goto(state,\ element)$
  $check\_output(state)$
  $scan\_next(element)$
end while

Assume then that instead of linear input text we are processing a stream of XML documents, and that the filters are composed of patterns with wildcards. That is, each filter f_i is of the form (5) with a leading descendant operator "//". We can use the construction of Theorem 2 yielding a PMA for all keywords appearing in the filters with an output function defined by

$$output(q) = \{(i,k) \mid \text{a suffix of } string(q) \text{ is the } k\text{th keyword of filter } f_i\}.$$

In this case backtracking is not as simple as in the case when filters do not contain wildcards (Theorem 1). When upon encountering a start-element tag of an element a *goto* arc is applied on the element, the current state will be pushed onto the stack, but also changes in the sets of partial matches computed at that state must be logged onto the stack. More specifically, depending on the statement used to change $partial\_matches(i)$, a tuple containing enough information for performing the reversal of the change will be pushed onto the stack.

Statements of the forms "insert $(p,1)$ into $partial\_matches(i)$" and "delete (p,k) from $partial\_matches(i)$" cause tuples $inserted\langle i,p,1\rangle$ and $deleted\langle i,p,k\rangle$,

Algorithm 2 Procedure *check_output(state)* for matching linear text with sets of patterns containing wildcards.

for all $(i,k) \in$ *output(state)* **do**
  $p \leftarrow \mathit{element\_count} - \mathit{length}(i,k) - \mathit{distance}(i,k)$
  if $k = 1 = \#\mathit{keywords}(i)$ **then**
    if $p \geq 0$ **then**
      insert p into $\mathit{matches}(i)$
    end if
  else if $k = 1 < \#\mathit{keywords}(i)$ **then**
    if $p \geq 0$ **then**
      insert $(p,1)$ into $\mathit{partial\_matches}(i)$
    end if
  else if $k = \#\mathit{keywords}(i)$ and $(p, k-1) \in \mathit{partial\_matches}(i)$ **then**
    delete $(p, k-1)$ from $\mathit{partial\_matches}(i)$
    insert p into $\mathit{matches}(i)$
  else if $k < \#\mathit{keywords}(i)$ and $(p, k-1) \in \mathit{partial\_matches}(i)$ **then**
    replace $(p, k-1)$ by (p,k) in $\mathit{partial\_matches}(i)$
  end if
  for all $(p', k') \in \mathit{partial\_matches}(i)$ with $p' < \mathit{element\_count} - \mathit{length}(i)$ **do**
    delete (p', k') from $\mathit{partial\_matches}(i)$
  end for
end for

respectively, to be logged, and a statement of the form "replace $(p, k-1)$ by (p,k) in $\mathit{partial\_matches}(i)$" causes tuple $\mathit{replaced}\langle i,p,k\rangle$ to be logged.

Then when an end-element tag is encountered the topmost state in the stack will be the new state, but before continuing processing at that state the reversal operations based on the logged tuples above the topmost state must be performed. For tuple $\mathit{inserted}\langle i,p,1\rangle$, the statement "delete $(p,1)$ from $\mathit{partial\_matches}(i)$" is performed. For tuple $\mathit{deleted}\langle i,p,k\rangle$, the statement "insert (p,k) into $\mathit{partial\_matches}(i)$" is performed. For tuple $\mathit{replaced}\langle i,p,k\rangle$, the statement "replace (p,k) by $(p,k-1)$ in $\mathit{partial\_matches}(i)$" is performed.

There is still one important point that must be taken into account when processing paths in a tree instead of linear text. This is that we must also correctly update the variable *element_count* such that its value is the number of elements in the current path. But this is accomplished by simply decrementing the counter by one whenever the current state is obtained from the stack.

It is clear that all reversal operations caused by backtracking can be performed as efficiently as the original operations, and the number of reversal operations cannot be greater than the number of original operations. Thus we have:

Theorem 3. Let $f_1, f_2, \ldots, f_r$ be filters each of which is of the form $//P_i$, where P_i is a pattern composed of XML element names, child operators (/), and wildcards ($*$). Then the XML filtering problem for document T of length n (counted as the number of XML elements) can be solved in time $O(n + m + \sum_{i=1}^{r} \alpha_i \log |P_i|)$, where m is the sum of the lengths of the filters and α_i denotes the number of occurrences of keywords of P_i in T.

Proof. The claim follows from the result stated in Theorem 2 when we observe that each reversal operation has the same cost as the operation to be reversed. Moreover, altogether the number of reversals cannot be more than $\sum_{i=1}^{r} \alpha_i$, because each of them either inserts or deletes a keyword occurrence, and never more than once. □

4 Matching linear XPath expressions

In this section we are finally ready to present our main result which states how multiple-pattern matching for streams of XML documents can be done efficiently when for the filters the full generality of linear XPath filters (without predicates) are allowed. More specifically, we consider matching with linear XPath filters that contain descendant operators (//) also in non-leading positions, that is, filters of the form

$$f_i = //P_{i,1}//P_{i,2}// \ldots //P_{i,m_i}, \tag{6}$$

where each subsequence $P_{i,j}$ is a pattern composed of XML element names, child operators (/), and wildcards ($*$). The special case in which the filter begins with "/" instead of "//" is handled by requiring that the SAX parser surrounds each document by tags $\langle\#\rangle$ and $\langle/\#\rangle$, where # is new element name. Every filter of the form $/g$ is transformed into $//\#/g$, thus being of the form (6).

We call the subsequences $P_{i,j}$ *segments* of the filter. Each segment is partitioned into one or more *keywords* as are the patterns considered in the previous section.

For example, the filter

$//a/b/*/*/c//*/*/d$

has two segments, namely $a/b/*/*/c$ and $*/*/d$, where the keywords of the first segment are ab and c, and the only keyword of the second segment is d.

The number of segments of filter i is given by $\#segments(i)$, and the number of keywords of segment j of filter i is given by $\#keywords(i, j)$. For filter number i, segment number j and keyword number k, $length(i, j, k)$ gives the length of the kth keyword of segment j of filter i, $distance(i, j, k)$ gives the distance of the kth keyword from the beginning of the segment, and $length(i, j)$ gives the length of segment j of filter i.

If the number of the above example filter is i, we have:

$length(i, 1, 1) = 2,\ distance(i, 1, 1) = 0,$
$length(i, 1, 2) = 1,\ distance(i, 1, 2) = 4,$
$length(i, 2, 1) = 1,\ distance(i, 2, 1) = 2,$
$length(i, 1) = 5,\ length(i, 2) = 3.$

As in the previous sections, the PMA is constructed from the set of all keywords that appear in the filters.

Output tuples in the output sets of states now take the form (i, j, k), where i is a filter number, j is a number of a segment of filter i, and k is a number of a keyword of segment j.

Partial matches of segment j of filter i are recorded in set $partial\_matches(i, j)$ containing pairs (p, k), where p is an element position in the input document denoting the possible start of a match of segment j of filter i, and k is the number of the keyword of segment j up to which the match has been found. As with the sets $partial\_matches(i)$ in the previous section, there is at most one pair (p, k) in $partial\_matches(i, j)$ for any given element position p. Thus we can store $partial\_matches(i, j)$ in the same way as $partial\_matches(i)$ as balanced binary trees with key p.

A partial match (p, k) in $partial\_matches(i, j)$ is a *full match* of segment j of filter i if $k = \#keywords(i, j)$. The sets $partial\_matches(i, j)$ are actually maintained in the same way as the sets $partial\_matches(i)$ of the previous section, the only difference being that a new partial match of segment $j > 1$ at position p can only be started if a match of segment $j - 1$ has been found at some previous position p', and far enough from p. In other words, $p' \leq p - length(i, j - 1)$. Because of this condition, a full match of segment j actually signals full matches of all segments of the entire filter from segment 1 upto and including segment j.

Full matches of segment j of filter i are collected into the set $matches(i, j)$; a match of an entire filter i has been found when a full match of the last segment of the filter has been found, that is, when the set $matches(i, j)$ with $j = \#segments(i)$ becomes nonempty. It is easy to see that sets $partial\_matches(i, j)$ can be maintained in time $O(\alpha_{i,j} \log |P_{i,j}|)$ time, where $\alpha_{i,j}$ denotes the number of occurrences of keywords of $P_{i,j}$ in the input XML document being filtered (cf. Section 3). Additionally, we need be able to test whether or not $p' \leq p - length(i, j - 1)$, for some p' in $matches(i, j - 1)$. This can be simply accomplished by maintaining the sets $matches(i, j)$ as balanced binary trees, and checking the condition for the smallest element in the tree. The total time needed for each $matches(i, j)$ is $O(\beta_{i,j} \log \beta_{i,j})$, where $\beta_{i,j}$ denotes the number of occurrences of $P_{i,1}//P_{i,2}// \ldots //P_{i,j}$.

We have:

Theorem 4. Let $f_1, f_2, \ldots, f_r$ be filters each of which is of the form

$$f_i = //P_{i,1}//P_{i,2}// \ldots //P_{i,m_i},$$

Algorithm 3 Operating cycle of a backtracking PMA for matching XML documents with filters containing descendant operators and wildcards.

```
for all filters i and their segments j do
  matches(i, j) ← empty
  partial_matches(i, j) ← empty
end for
state ← initial_state
element_count ← 0
scan_next(token)
while token was found do
  if token is the start-element tag of element e then
    element_count ← element_count +1
    stack.push(state)
    while goto(state, e) = fail do
      state ← fail(state)
    end while
    state ← goto(state, e)
    check_output(state)
  else if token is an end-element tag then
    state ← backtrack()
    element_count ← element_count −1
  end if
  scan_next(token)
end while
```

where $P_{i,j}$ is a pattern composed of XML element names, child operators (/), and wildcards ($*$). Then the XML filtering problem for document T of length n (counted as the number of XML elements) can be solved in time

$$O(n + m + \sum_{i=1}^{r} \sum_{j=1}^{m_i} \alpha_{i,j} \log |P_{i,j}| + \sum_{i=1}^{r} \sum_{j=1}^{m_i} \beta_{i,j} \log \beta_{i,j}),$$

where m is the sum of the lengths of the filters, $\alpha_{i,j}$ denotes the number of occurrences in T of keywords of $P_{i,j}$, and $\beta_{i,j}$ the number of occurrences in T of $P_{i,1}//P_{i,2}//\ldots//P_{i,j}$.

Proof. The Algorithms 3–6 give a solution that has the stated time bound, when we observe the above discussion of how the sets $partial\_matches(i, j)$ and $matches(i, j)$ are maintained. Backtracking is needed in the same way as for the result stated as Theorem 3, but now we have explicitly put all operations in the algorithms: logging is performed in Algorithms 4 and 5, and Algorithm 6 performs the backtracking. Again (cf. the proof of Theorem 3) the number of

Algorithm 4 Procedure *check_output*(*state*) for matching XML documents with filters containing descendant operators and wildcards.

```
for all (i, j, k) ∈ output(state) do
  p ← element_count − length(i, j, k) − distance(i, j, k)
  if k = 1 then
    if j = 1 then
      if p ≥ 0 then
        advance_match(i, j, p, 1)
      end if
    else if matches(i, j − 1) contains some p′ ≤ p − length(i, j − 1) then
      advance_match(i, j, p, 1)
    end if
  else if (p, k − 1) ∈ partial_matches(i, j) then
    advance_match(i, j, p, k)
  end if
  for all (p′, k′) ∈ partial_matches(i, j) with p′ < element_count − length(i, j)
  do
    delete (p′, k′) from partial_matches(i, j)
    stack.push(deleted⟨i, j, p′, k′⟩)
  end for
end for
```

Algorithm 5 Procedure *advance_match*(i, j, p, k).

```
if k = 1 then
  insert (p, 1) into partial_matches(i, j)
  stack.push(inserted⟨i, j, p, 1⟩)
else
  replace (p, k − 1) by (p, k) in partial_matches(i, j)
  stack.push(replaced⟨i, j, p, k⟩)
end if
if k = #keywords(i, j) then
  insert p into matches(i, j)
end if
```

reversals performed cannot be more than the number of all occurrences of any keywords in T. □

A slightly less tight but much simpler formulation of Theorem 4 can be stated as follows:

Theorem 5. Let $f_1, f_2, \ldots, f_r$ be filters as in Theorem 4. Then the XML filtering problem for document T of length n (counted as the number of XML

Algorithm 6 Procedure *backtrack*().

$s \leftarrow stack.pop()$
while s is not a state **do**
  if $s = inserted\langle i,j,p,1\rangle$ for some i, j, p **then**
    delete $(p,1)$ from $partial\_matches(i,j)$
  else if $s = replaced\langle i,j,p,k\rangle$ for some i, j, p, k **then**
    replace (p,k) by $(p,k-1)$ in $partial\_matches(i,j)$
  else if $s = deleted\langle i,j,p,k\rangle$ for some i, j, p, k **then**
    insert (p,k) into $partial\_matches(i,j)$
  end if
  $s \leftarrow stack.pop()$
end while
$return(s)$

elements) can be solved in time

$$O(n + m + \alpha \log \alpha),$$

where m is the sum of the lengths of the filters and α denotes the number of all occurrences in T of any keywords of the filters. □

5 Conclusion

In this article we have presented a new algorithm for matching linear XPath expressions (without predicates) with XML documents. The application we had in mind was filtering with filters expressed as linear XPath expressions; that is, for each document in an XML document stream, the task is to determine those filters that match the document. The basic building block in this algorithm is the Aho-Corasick multiple-pattern-matching algorithm, which we have extended in two novel ways. First, we showed how it can be efficiently applied when wildcards are present in the patterns, and second, we showed how tree patterns can be matched with tree-like text. Specifically, we used as patterns linear XPath expressions and XML documents as tree-like text.

Our main results (Theorem 4 and Theorem 5) are contributions in the sense that they state new worst-case bounds for XML filtering. Our future work aims at improvements by dynamically using the information of matched prefixes of patterns, without explicitly trying to match portions of patterns for which no corresponding matched prefix exists.

Acknowledgement

The financial support of the Academy of Finland is gratefully acknowledged.

References

[Aho and Corasick 1975] Aho, A. V., Corasick, M. J.: “Efficient string matching: an aid to bibliographic search”; Communications of the ACM, 18, 6 (1975), 333–340.

[Altinel and Franklin 2000] Altinel, M., Franklin, M. J.: “Efficient filtering of XML documents for selective dissemination of information”; VLDB 2000, Proc. of 26th Internat. Conf. on Very Large Data Bases, 53–64.

[Avila Campillo et al.] Avila-Campillo, I., Raven, D., Green, T., Gupta, A., Kadiyska, Y., Onizuka, M., D. Suciu, D.: “An XML toolkit for light-weight XML stream processing”; `http://www.cs.washington.edu/homes/suciu/XMLTK/`.

[Diao et al. 2003] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., Fischer, P.: “Path sharing and predicate evaluation for high-performance XML filtering”; ACM Trans. Database Syst., 28, 4 (2003), 467–516.

[Fernandez and Suciu 1998] Fernandez, M. F., Suciu, D.: “Optimizing regular path expressions using graph schemas”; Proc. of the 14th IEEE Internat. Conf. on Data Engineering (1998), 14–23.

[Green et al. 2004] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: “Processing XML streams with deterministic automata and stream indexes”; ACM Trans. Database Syst., 29, 4 (2004), 752–788.

[Gupta and Suciu 2003] Gupta, A. K., Suciu, D.: “Stream processing of XPath queries with predicates”; Proc. of the 2003 ACM SIGMOD Internat. Conf. on Management of Data, 419–430.

[Kalai 2002] Kalai, A.: “Efficient pattern-matching with don’t cares”; Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms, 655–656.

[Kilpeläinen and Wood 2001] Kilpeläinen, P., Wood, D.: “SGML and XML document grammars and exceptions”; Information and Computation, 169 (2001), 230–251.

[Lee et al. 2007] Lee, D., Shin, H., Kwon, J., Yang, W., Lee, S.: “SFilter: schema based filtering system for XML streams”; MUE 2007, Internat. Conf. on Multimedia and Ubiquitous Engineering, Seoul, Korea, 266–271.

[Onizuka 2003] Onizuka, M.: “Light-weight xPath processing of XML stream with deterministic automata”; Proc. of the 2003 ACM CIKM Internat. Conf. on Information and Knowledge Management, 342–349.

[Pinter 1985] Pinter, R. Y.: “Efficient string matching”; Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO Advanced Science Institute Series F: Computer and System Sciences, vol. 12, 11–29.

[Rahman et al. 2006] Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: “Finding patterns with variable length gaps or don’t cares”; Chen, D.Z., Lee, D.T. (eds.) Computing and Combinatorics, 12th Annual Internat. Conf., COCOON 2006, Proceedings. LNCS, vol. 4112, 146–155.

[Rahman and Iliopoulos 2007] Rahman, M.S., Iliopoulos, C.S.: “Pattern matching algorithms with don’t cares”; SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conf. on Current Trends in Theory and Practice of Comp. Sci., 116–126.

[Silvasti et al. 2009a] Silvasti, P., Sippu, S., Soisalon-Soininen, E.: “Schema-conscious filtering of XML documents”; EDBT 2009, Proc. of the 12th Internat. Conf. on Extending Database Technology, 970–981.

[Silvasti et al. 2009b] Silvasti, P., Sippu, S., Soisalon-Soininen, E.: “Processing schema-optimized XPath filters by deterministic automata”; SEDE 2009, Proc. of the 18th Internat. Conf. on Software Engineering and Data Engineering, 55–60.

[Suciu 2006] Suciu, D.: “XML data repository”; The Database Research Group of University of Washington, 2006; `http://www.cs.washington.edu/research/xmldatasets/`.