Online Dictionary Matching for Streams of XML Documents

Panu Silvasti[1], Seppo Sippu[2], and Eljas Soisalon-Soininen[1]

[1] Aalto University, School of Science and Technology, Finland
{ess,psilvast}@cs.hut.fi
[2] University of Helsinki, Finland
sippu@cs.helsinki.fi

Abstract. We consider the online multiple-pattern matching problem for streams of XML documents, when the patterns are expressed as linear XPath expressions containing child operators (/), descendant operators (//) and wildcards ($*$) but no predicates. For each document in the stream, the task is to determine all occurrences in the document of all the patterns. We present a general multiple-pattern-matching algorithm that is based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick pattern-matching automaton. This automaton is of size linear in the sum of the sizes of the XPath patterns, and the worst-case time bound of the algorithm is better than the time bound of the simulation of linear-size nondeterministic automata. In addition to the worst-case-efficient general solution we present an algorithm with a simple backtracking mechanism that works extremely well for cases in which the backtracking stack remains low. Our experiments show that, when applied to filtering, this simple algorithm scales well as regards the number of patterns (or filters) and is competitive with YFilter, a widely accepted software for XML filtering.

1 Introduction

String-pattern matching with wildcards has been considered in various contexts and for various types of wildcards in the pattern [1–6]. The simplest approach is to use the single-character wildcard ($*$), a character that can appear in any position of the string pattern and matches any character of the input alphabet Σ. Generalizations of this are the various ways in which "variable-length gaps" in the patterns are allowed, implemented, for example, by the single-character wildcard and by Σ^* that denotes an unlimited gap [4, 5].

Our goal in this article is to transfer the methodology of efficient string-pattern matching with wildcards and gaps to matching tree-structured text, especially one composed of a stream of XML documents. The patterns are given as linear XPath expressions with child operators (/), descendant operators (//), and wildcards ($*$), defined on paths of XML-document trees. Here the child and descendant operators correspond, respectively, to the concatenation and unlimited-gap operators of linear text.

Given a set of linear XPath expressions without predicates (the *patterns*) and a stream of XML documents (the *input documents*), we determine, for each document in the stream, all occurrences of all the patterns in an online fashion. Each matched occurrence is identified by the pattern and its last element position in the document. Because of the descendant operator "//", there can be more than one, actually an exponential number of occurrences of the same pattern at the same element position, but we avoid this possible explosion of the number of occurrences by reporting only one occurrence in such situations.

Our pattern-matching algorithm performs a single left-to-right scan of each document and reports each pattern occurrence once its end position is reached. Our strategy here is that we decompose each pattern into "keywords" that are maximal substrings of XML elements only containing child operators "/". Then we are able to consider the patterns as sequences composed of keywords and gaps that are maximal sequences of wildcards "$*$" and descendant operators "//". The basic idea of our algorithm is to recognize the keywords of the patterns using the Aho–Corasick pattern-matching automaton [7], and to build matches of the patterns upon these occurrences. The key feature of our algorithm is that we do not recognize those keyword occurrences about which we know that there cannot be a matching prefix with this keyword occurrence. Moreover, because matching of the patterns is performed against paths of the trees induced by XML documents, we need a backtracking mechanism in order to avoid repeated traversals of common path prefixes.

Our algorithm can also be used for filtering, in which only the first occurrence of each pattern needs to be reported. Related approaches to filtering are those of the NFA-based XFilter by Altinel and Franklin [8] and YFilter by Diao et al. [9, 10], and the lazy DFA construction by Green et al. [11], only to mention the best-known ones. Compared to these methods our algorithm has a superior worst-case time bound. However, for practical XML documents, the tree paths remain quite short implying that our general approach with a worst-case-efficient backtracking strategy is slower than the above mentioned filtering algorithms. We noticed this when we performed our experimental comparisons with YFilter, which is a widely accepted software for filtering and transformation for high-volume XML-message brokering [9]; YFilter has been programmed with Java, as is our system.

Because of the inefficiency of the general approach we also designed a simple backtracking strategy, which allows the matching algorithm to return to a previous state at the end of the current path by just popping a state from the backtracking stack. This approach implies a higher worst-case time bound, but indeed makes our approach competitive with YFilter. Moreover, our algorithm outperforms YFilter in the case of data whose XML schema or DTD is not heavily recursive.

The contributions of the present paper are essential extensions of our previous results [12], where a worst-case-efficient algorithm was presented in the case in which descendant operators, but no wildcards, were allowed in the patterns.

2 Linear XML Patterns with Gaps and Wildcards

The problem of online dictionary matching of XML is stated as follows: Given a set of linear XPath expressions without predicates (the *patterns*) and a stream of XML documents (the *input documents*), the task is to determine, for each document in the stream, all occurrences of all the patterns. An occurrence is reported by the pattern number and the element position of its last element in the current document. Different instances of the same pattern at the same position are reported as a single occurrence.

We decompose each pattern into keywords and gaps as follows. First, we remove all child operators "/" from the pattern. Then we define the *keywords* of the pattern to be maximal substrings consisting of XML elements only. The *gaps* of the patterns are defined to be maximal substrings consisting of descendant operators "//" and wildcards "$*$". If the pattern ends at a nonempty gap, then we assume that the last keyword of the pattern is the empty string ϵ. Each pattern is considered to begin with a gap, which thus may be ϵ.

We number the patterns and their gaps and keywords consecutively, so that the ith pattern P_i can be represented as

$$P_i = \mathit{gap}(i,1)\mathit{keyword}(i,1)\ldots\mathit{gap}(i,m_i)\mathit{keyword}(i,m_i),$$

where $\mathit{gap}(i,j)$ denotes the jth gap and $\mathit{keyword}(i,j)$ denotes the jth keyword of pattern P_i. For example, the pattern $//a/b/*/c//*/d/*/*$ consists of four gaps, namely $//$, $*$, $//*$, and $**$, and of four keywords, namely ab, c, d, and ϵ. The pattern $/a/b/*/c//*/d$ consists of three gaps, namely ϵ, $*$, and $//*$, and three keywords, namely ab, c, and d.

For pattern P_i, we denote by $\mathit{mingap}(i,j)$ and $\mathit{maxgap}(i,j)$, respectively, the minimum and maximum lengths of element strings that can be matched by $\mathit{gap}(i,j)$. The length of the jth keyword of pattern P_i is denoted by $\mathit{length}(i,j)$. We also assume that $\#\mathit{keywords}(i)$ gives m_i, the number of keywords in pattern P_i, and that $\#\mathit{keywords}$ denotes the number of keywords altogether.

For example, if the pattern $//a/b/*/c//*/d/*/*$ is the ith pattern, we have

$\mathit{mingap}(i,1) = 0$, $\mathit{maxgap}(i,1) = \infty$, $\mathit{length}(i,1) = 2$,
$\mathit{mingap}(i,2) = 1$, $\mathit{maxgap}(i,2) = 1$, $\mathit{length}(i,2) = 1$,
$\mathit{mingap}(i,3) = 1$, $\mathit{maxgap}(i,3) = \infty$, $\mathit{length}(i,3) = 1$,
$\mathit{mingap}(i,4) = 2$, $\mathit{maxgap}(i,4) = 2$, $\mathit{length}(i,4) = 0$.

If the pattern $/a/b/*/c//*/d$ is the ith pattern, we have

$\mathit{mingap}(i,1) = 0$, $\mathit{maxgap}(i,1) = 0$, $\mathit{length}(i,1) = 2$,
$\mathit{mingap}(i,2) = 1$, $\mathit{maxgap}(i,2) = 1$, $\mathit{length}(i,2) = 1$,
$\mathit{mingap}(i,3) = 1$, $\mathit{maxgap}(i,3) = \infty$, $\mathit{length}(i,3) = 1$.

In the case of XPath patterns, we have, for all i and j, either $\mathit{mingap}(i,j) = \mathit{maxgap}(i,j)$ or $\mathit{maxgap}(i,j) = \infty$, because in any XPath expression the number of wildcards is fixed. However, as will be evident from the presentation below, our algorithm can also handle any variable-length gaps with $\mathit{mingap}(i,j) < \mathit{maxgap}(i,j) < \infty$.

3 The Matching Algorithm

For the set of all keywords in the patterns, we construct a backtracking Aho–Corasick pattern-matching automaton [7] with a dynamically changing output set *current-output* containing tuples of the form (q, i, j, b, e), where $q = \mathit{state}(\mathit{keyword}(i, j))$, the state reached from the initial state upon reading the jth keyword of pattern P_i, and b and e are the earliest and latest element positions on a path in the input document at which some partial match of pattern P_i up to and including the jth keyword can possibly be found. The latest possible element position e may be ∞, meaning the end of the path.

Initially, the set *current-output* contains all output tuples for the first keywords in the patterns, that is, tuples $(q, i, 1, b, e)$, where q is the state reached from the initial state upon reading the first keyword of pattern P_i,

$b = \mathit{mingap}(i, 1) + \mathit{length}(i, 1)$, and
$e = \mathit{maxgap}(i, 1) + \mathit{length}(i, 1)$

Here $e = \infty$ if $\mathit{maxgap}(i, 1) = \infty$.

The operating cycle of the PMA is given as Alg. 1. The SAX-parser [13] call *scan-next*(*token*) returns the next XML token from the input stream. The functions *goto* and *fail* are the goto and fail functions of the standard Aho–Corasick PMA, that is, $\mathit{goto}(\mathit{state}(y), a) = \mathit{state}(ya)$, where ya is a prefix of some keyword and a is an XML element, and $\mathit{fail}(\mathit{state}(uv)) = \mathit{state}(v)$, 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.

Denote by $\mathit{string}(q)$ the unique element string y with $\mathit{state}(y) = q$. The function $\mathit{output\text{-}fail}(q)$ used in Alg. 3 to traverse the *output path* for state q is defined by: $\mathit{output\text{-}fail}(q) = \mathit{fail}^k(q)$, where k is the greatest integer less than or equal to the length of $\mathit{string}(q)$ such that $\mathit{string}(\mathit{fail}^m(q))$ is not a keyword for any $m = 1, \ldots, k-1$. Here fail^m denotes the *fail* function applied m times. Thus, the output path for state q includes those states in the fail path from q that have a nonempty set of output tuples.

The *backtracking stack* contains information about states visited and output tuples inserted into and deleted from the current output during traversing a root-to-leaf path in the current input document. The PMA backtracks when an end-element tag is scanned; then elements from the stack are popped, insertions and deletions of output tuples are undone, and the control of the PMA is returned to the state that was entered when scanning the previous start-element tag (see the procedure *backtrack* given as Alg. 4).

When visiting state q, the current output of the PMA is checked for possible matches of keywords in the procedure call $\mathit{traverse\text{-}output\text{-}path}(q)$ (see Alg. 3). A current output tuple (q, i, j, b, e) is found to represent a match of the jth keyword of pattern P_i, if $b \leq \mathit{path\text{-}length} \leq e$, where *path-length* is a global variable that maintains the number of elements scanned from the current path in the input document. Now if the jth keyword is the last one in pattern P_i, then this indicates a match of the entire pattern P_i. Otherwise, an output tuple $(q', i, j+1, b', e')$ for the $(j+1)$st keyword of pattern P_i is inserted into the set *current-output*, where q' is the state reached from the intitial state upon reading the $(j+1)$st keyword of pattern P_i,

$b' = \mathit{path\text{-}length} + \mathit{mingap}(i, j+1) + \mathit{length}(i, j+1)$, and
$e' = \mathit{path\text{-}length} + \mathit{maxgap}(i, j+1) + \mathit{length}(i, j+1)$.

Here $e' = \infty$ if $\mathit{maxgap}(i, j+1) = \infty$. If $e' = \infty$, we could delete from *current-output* all output tuples (q'', i, j'', b'', e'') with $j'' \leq j$. However, since tuples (q'', i, j'', b'', e'') with $e'' < \mathit{path\text{-}length}$ are eventually deleted by the procedure *clean-current-output* (see Alg. 2), we only delete here tuples $(q'', i, j'', b'', \infty)$ with $j'' \leq j$ (which can be done efficiently, see below).

The set of current output tuples is organized as two balanced binary search trees, both containing exactly the same information, namely, the current output tuples (q, i, j, b, e). One of the search trees is indexed by ordered triples (e, i, j), so that the node for key value (e, i, j) contains a pointer to a list of tuples (q, b). The other search tree is indexed by ordered pairs (q, b), so that the node for key value (q, b) contains a pointer to a list of tuples (e, i, j).

In the procedure *clean-current-output*, outdated output tuples (q, i, j, b, e) with $e < \mathit{path\text{-}length}$ are first located and deleted from the former search tree, and then sorted by (q, b) and deleted from the latter search tree, in both cases using a bulk-deletion algorithm.

In the procedure *traverse-output-path*, when visiting state q, the latter search tree is used to locate the output tuples (q, i, j, b, e) with $b \leq \mathit{path\text{-}length} \leq e$. Deletions of tuples $(q'', i, j'', b'', \infty)$ with $j'' \leq j$ are first performed on the latter search tree, and then sorted by (q'', b'') and deleted from the former search tree, again using a bulk-deletion algorithm. Every insertion into *current-output* goes to both search trees.

For pattern P_i, *#maxmatches*(i) denotes the maximum number of matches searched for i; this is specified by the user and may be a positive integer or ∞. For example, if we wish to solve only the filtering problem, we set *#maxmatches*$(i) = 1$ for all patterns P_i. The counter *#matches*(i) records the number of matches found for pattern P_i.

4 Correctness and Complexity

Whenever a prefix $P_{i,j}$ of P_i that ends with *keyword*(i, j) has been recognized at some element position (indicated by the variable *element-count* in the algorithm), a new output tuple $(q', i, j+1, b', e')$ will be inserted into the current output in Alg. 3. We have:

Lemma 1. At the point when a new tuple $(q', i, j+1, b', e')$ will be inserted into the current output in Alg. 3, an occurrence of a pattern prefix $P_{i,j}$ has been recognized. Conversely, for each occurrence of $P_{i,j}$, $j < m_i$, a tuple (q, i, j, b, e) with $b \leq \mathit{path\text{-}length} \leq e$ is found in the current output and a new tuple $(q', i, j+1, b', e')$ is inserted into the current output.

For a set S of keywords or pattern prefixes, denote by $\mathit{occ}(S)$ the number of occurrences in the input document of elements of S. Lemma 1 implies that

Algorithm 1 Operating cycle of the backtracking PMA.

```
$document\text{-}count \leftarrow 0$
$element\text{-}count \leftarrow 0$
$scan\text{-}next(token)$
while $token$ was found do
  $element\text{-}count \leftarrow element\text{-}count + 1$
  if $token$ is a start-document tag then
    $document\text{-}count \leftarrow document\text{-}count + 1$
    $initialize()$
    $path\text{-}length \leftarrow 0$
    $state \leftarrow initial\text{-}state$
    $push\text{-}onto\text{-}stack(state)$
    $traverse\text{-}output\text{-}path(state)$
  else if $token$ is an end-document tag then
    $element\text{-}count \leftarrow 0$
  else if $token$ is the start-element tag of element $E$ then
    $path\text{-}length \leftarrow path\text{-}length + 1$
    $push\text{-}onto\text{-}stack(state)$
    while $goto(state, E) = fail$ do
      $state \leftarrow fail(state)$
    end while
    $state \leftarrow goto(state, E)$
    $traverse\text{-}output\text{-}path(state)$
    $clean\text{-}current\text{-}output()$
  else if $token$ is an end-element tag then
    $backtrack()$
    $path\text{-}length \leftarrow path\text{-}length - 1$
  end if
  $scan\text{-}next(token)$
end while
```

Algorithm 2 Procedure *clean-current-output*().

```
for all tuples $(q, i, j, b, e) \in current\text{-}output$ with $e < path\text{-}length$ do
  delete $(q, i, j, b, e)$ from $current\text{-}output$
  if $\#matches(i) < \#maxmatches(i)$ then
    $push\text{-}onto\text{-}stack(deleted\langle q, i, j, b, e\rangle)$
  end if
end for
```

Algorithm 3 Procedure *traverse-output-path*(*state*).

```
$q \leftarrow state$
$traversed \leftarrow$ false
while not $traversed$ do
  for all $(q, i, j, b, e) \in current\text{-}output$ with $b \leq path\text{-}length \leq e$ do
    if $\#matches(i) = \#maxmatches(i)$ then
      delete $(q, i, j, b, e)$ from $current\text{-}output$
    else if $j = \#keywords(i)$ then
      report a match of pattern $P_i$ at position $element\text{-}count$ in document $document\text{-}count$
      $\#matches(i) \leftarrow \#matches(i) + 1$
      if $\#matches(i) = \#maxmatches(i)$ then
        delete $(q, i, j, b, e)$ from $current\text{-}output$
        for all $(q', i, j', b', \infty) \in current\text{-}output$ do
          delete $(q', i, j', b', \infty)$ from $current\text{-}output$
        end for
      end if
    else
      $q' \leftarrow state(keyword(i, j + 1))$
      $b' \leftarrow path\text{-}length + mingap(i, j + 1) + length(i, j + 1)$
      $e' \leftarrow path\text{-}length + maxgap(i, j + 1) + length(i, j + 1)$
      insert $(q', i, j + 1, b', e')$ into $current\text{-}output$
      $push\text{-}onto\text{-}stack(inserted\langle q', i, j + 1, b', e' \rangle)$
      if $e' = \infty$ then
        for all $(q'', i, j'', b'', \infty) \in current\text{-}output$ with $j'' \leq j$ do
          delete $(q'', i, j'', b'', \infty)$ from $current\text{-}output$
          $push\text{-}onto\text{-}stack(deleted\langle q'', i, j'', b'', \infty \rangle)$
        end for
      end if
    end if
  end for
  if $q = initial\text{-}state$ then
    $traversed \leftarrow$ true
  else
    $q \leftarrow output\text{-}fail(q)$
  end if
end while
```

Algorithm 4 Procedure *backtrack*().

```
pop the topmost element $s$ from the stack
while $s$ is not a state do
  if $s$ is $inserted\langle q, i, j, b, e \rangle$ then
    delete $(q, i, j, b, e)$ from $current\text{-}output$
  else if $s$ is $deleted\langle q, i, j, b, e \rangle$ then
    insert $(q, i, j, b, e)$ into $current\text{-}output$
  end if
  pop the topmost element $s$ from the stack
end while
$state \leftarrow s$
```

the algorithm has the lower time bound $\Omega(occ(\{P_{i,j} \mid i \geq 1, j \geq 1\}))$. Moreover, processing the input document requires additionally at most $O(K \cdot n)$ time, where n is the length of the input document and K denotes the maximum number of proper suffixes of any keyword that are also keywords. The multiplier K is due to the fact that tuples to be inserted into the current output are created only for states *state*(*keyword*(i, j)) and thus all states in the output path must be traversed in order to check all possibilities to continue the currently matched pattern prefix. Observe that the backtracking approach using a stack allows to relate the processing time to input length even though patterns are matched against tree paths. Some extra cost is due to maintaining the set *current-output*. Lemma 1 implies:

Lemma 2. The outermost **for** loop of Alg. 3 will be performed as many times as there are different occurrences of nonempty prefixes $P_{i,j}$ of pattern P_i, $j < m_i$, for all i. Moreover, for each iteration of the **while** loop, when performing the **for** loop of Alg. 3, the condition of the **for** loop will be tested unsuccessfully only once.

Output tuples (q, i, j, b, e) created in Alg. 3 become outdated when *path-length* advances beyond e, and should then be deleted. This is performed by the procedure *clean-current-output* (see Alg. 2). Also we have:

Lemma 3. After inserting tuple $(q', i, j+1, b', \infty)$ into the current output it is correct to delete all tuples of the form $(q'', i, j'', b'', \infty)$, where $j'' \leq j$.

Lemmas 1 and 3 imply:

Theorem 1. The multiple-pattern-matching algorithm given as Algs. 1–4 correctly reports all occurrences in the input document of all patterns P_i.

By Lemmas 1 and 2, and the discussion above we conclude:

Theorem 2. Excluding preprocessing that includes the construction of the Aho–Corasick PMA and only takes time linear in the total size of the patterns, the multiple-pattern-matching algorithm given in the previous section as Algs. 1–4 runs in time

$$O(K \times n + \log(L \times \#keywords) \times occ(\{P_{i,j} \mid i \geq 1, j \geq 1\})),$$

where n denotes the number of XML elements in the input document, K is the maximum number of proper suffixes of a keyword that are also keywords, and L is the depth of input document, that is, the maximum length of a path in the input document.

The logarithm term in the time bound of Theorem 2 is due to maintaining the binary trees as defined at the end of Sec. 3. The extreme worst case of the time bound occurs when all keywords that appear in the patterns are the same and all pattern prefixes match at every position in the input text. However, when the number of patterns is large we can safely assume that such a situation is very rare. If, on the contrary, $\#keywords = cN$, where N denotes the number of different keywords and c is a constant, the bound takes the form $O(Kn \times \log(L \times \#keywords))$.

5 Fast Backtracking

Our experiments have shown that in practice the tasks involved in backtracking the PMA constitute a performance bottleneck of our general algorithm presented in Sec. 3. The paths in XML documents tend to be quite short, so that backtracking happens often, and output tuples inserted into the current output and recorded into the backtracking stack are soon deleted from the current output because the path ends and backtracking must be performed.

We now present an organization for the current output tuples that allows for very fast backtracking. The current output set is stored in a *stack of blocks*, where each block is an array of *#states* entries, one for each state. The stack grows and shrinks in parallel with a stack used to store the states entered when reading start-element tags from the input document. The stack may grow up to a height of *maxdepth* blocks, where *maxdepth* is the length of the longest path in any input document in the stream. The block at height h stores the output tuples inserted when $h = \textit{path-length}$. Memory for the stack of blocks is allocated dynamically, so that *maxdepth* need not be known beforehand. Backtracking now involves only popping a state from the stack of states and forgetting the topmost block of the stack of blocks of output tuples.

The stack of blocks is implemented as a single dynamically growing array *current-output* of at most $O(\#\textit{states} \times \textit{maxdepth})$ entries, so that the index of the entry for state q in block h is obtained as $k = (h-1) \times \#\textit{states} + q$ (states q are numbered consecutively $1, 2, \ldots, \#\textit{states}$). The array entry *current-output*$[k]$ stores a tuple (t, d, v), where t is (a pointer to) a balanced binary search tree (a red-black tree) of output tuples (q, i, j, b, e) inserted into the current output when $\textit{path-length} = h$, $\textit{document-count} = d$, and $\textit{element-count} = v$. The binary search tree is indexed by the element positions b.

The pairs (d, v) act as version numbers of the entries in the array *current-output* and they relieve us from the need to deallocate an entire block when backtracking and from the need to reinitialize a block whose space is reused. When inserting a new output tuple (q, i, j, b, e) into the binary search tree t given in the entry *current-output*$[k] = (t, d, v)$, we first check whether or not $d = \textit{document-count}$ and $v = \textit{element-count}$; if not, the entry contains outdated information and hence must be reinitialized: the tree rooted at t is forwarded to a garbage collector, t is initialized as empty, and d and v are set to the current values of *document-count* and *element-count*. When traversing an output path and finding out which output tuples for state q stored in block h match, we first check whether or not $d = \textit{document-count}$ and $v = \textit{element-count}$ for the entry in *current-output*; if so, the entry is current and the output tuples (q, i, j, b, e) stored in the search tree of the entry are checked for the condition $b \leq \textit{path-length} \leq e$.

The backtracking stack that in the algorithm of Sec. 3 contained, besides states pushed there when reading start-element tags, also logging information about output tuples inserted or deleted from the current output, is now reduced to a stack of pairs (q, v), where q is the state and d is the value of *element-count* that were current at the time the pair was pushed onto the stack.

A downside of this algorithm is that the current output for state q is now dispersed in h blocks, where h is *path-length*, the length of the current path. The traversal of the output path for a state involves searches on $h \times K$ different search trees, where K is the length of the output path. This means that the term $K \times n$ in the complexity bound stated in Theorem 2 is changed to $\mathit{maxdepth} \times K \times n$.

6 Experimental Analysis

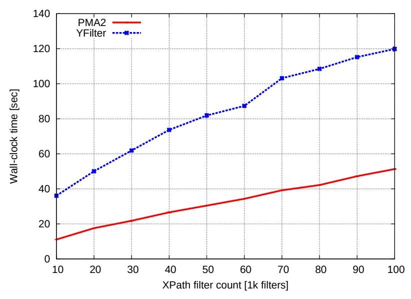
We have implemented (in Java) various versions of our pattern-matching algorithm, including the one described in Sec. 5 and denoted by "PMA2" in Fig. 1. The performances of PMA2 and YFilter [10] were evaluated with sets of patterns generated for two publicly available data sets: the slightly recursive NASA data set [14] and the highly recursive NewsML [15] data set. We also experimented with the basic version of our algorithm described in Sec. 3, whose asymptotic complexity is lower than that of the PMA2 version, but as its performance turned out to be inferior to that of PMA2, we only report results for PMA2 here.

Workloads of 10 000 to 100 000 linear XPath patterns without predicates were generated using the XPath query generator described by Diao et al. [10], parameterized with the maximum depth of XPath patterns and with the probabilities of the occurrences of the descendant operator ($\mathit{prob}(//)$) and of the wildcard ($\mathit{prob}(*)$). For each pattern workload the maximum depth of XPath patterns was set to the depth of the XML input document, that is, 8 for the 23.8 MB NASA document and 10 for the 2.6 MB NewsML document. For 10 000 patterns, our PMA has 671 states in the case of NASA and 1576 states in the case of NewsML.

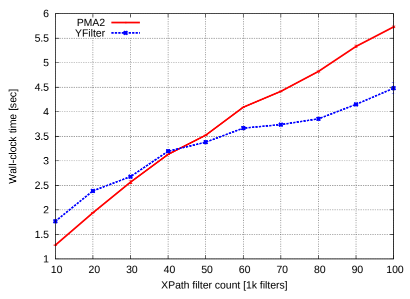
All our tests were run on a Dell PowerEdge SC430 server with 2.8 GHz Pentium 4 processor, 3 GB of main memory, and 1 MB of on-chip cache. The computer was running the Debian Linux 2.6.18 operating system with the Sun Java virtual machine 1.6.0_16 installed. In the tests the input document was read from the disk, but the overhead of the disk operations should be fairly small. The disk-read speed of the test hardware is more than 50 MB/sec. The throughput of the Java JAXP SAX parser (run in non-validating mode) on the input documents was 25–28 MB/sec.

Fig. 1 shows the running times of PMA2 and YFilter on the NASA and NewsML data sets for the filtering problem (that is, $\#\mathit{maxmatches}(i) = 1$ for each pattern P_i). The workloads of 10 000 to 100 000 linear XPath patterns without predicates were generated with $\mathit{prob}(*) = \mathit{prob}(//) = 0.2$. As is seen from the graphs, our algorithm clearly outperforms YFilter in the case of the workloads for the slightly recursive NASA data set, but for NewsML workloads greater than 50 000 filters, our algorithm is slightly inferior to YFilter.

Besides these filtering tests we also run with our algorithm some tests in which all occurrences of all patterns were determined. Tests with the two data sets and with 10 000 or 20 000 patterns show that the running time of PMA2 is 5.7- to 6.2-fold for the NASA data and 1.7-fold for the NewsML data when compared to the time spent on determining only the first occurrences. When



(a) NASA



(b) NewsML

Fig. 1: Filtering times for two XML data sets.

all the occurrences are stored, the space consumption is 290-fold and 7-fold, respectively. The high space consumption and speed degradation in the case of the NASA data set is mainly due to the great number of pattern occurrences.

7 Conclusion

Our main contribution is a new algorithm for online multiple-pattern matching of tree-structured text, where the patterns are given as path expressions composed of keywords and variable-length gaps. When applied to streams of XML documents, the patterns are linear XPath expressions without predicates, the keywords are maximal substrings of XML elements and child operators "/", and the gaps are maximal substrings of descendant operators "//" and wildcards "$*$".

Our algorithm is based on methodology previously applied to dictionary matching of linear text. We construct the Aho–Corasick pattern-matching automaton for the set of all keywords in the patterns, and we use this automaton for recognizing the occurrences of the keywords. From these we build occurrences of prefixes of patterns by checking that a candidate continuation of an already found prefix yields a new longer prefix, until an occurrence of a complete pattern is found. Our algorithm avoids recognizing occurrences of keywords that do not yield a proper continuation of any already found occurrence of a prefix of a pattern.

The use of the designed algorithm for matching patterns on paths of XML-document trees implies that a backtracking mechanism must be included, so that common prefixes of paths need not be traversed several time. However, worst-case-efficient backtracking, when tree paths are short as in typical XML documents, did not give good performance in practice. Therefore, we also designed a very simple backtracking strategy, which is not as good in the worst case, but allows for a considerable performance gain when tree paths remain short. We compared this simplified algorithm, when applied to filtering (in which only the first occurrences of the patterns are determined) with YFilter [10]. Our conclusion was that our method is better than YFilter, when the XML documents are not, as is usual in practice, deeply recursive.

Acknowledgement

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

References

1. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. of the 36th Annual ACM Symposium on Theory of Computing. (2004) 90–100
2. Fischer, M., Paterson, M.: String matching and other products. In: Proc. of the 7th SIAM-AMS Complexity of Computation. (1974) 113–125
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
4. Kucherov, G., Rusinowitch, M.: Matching a set of strings with variable length don't cares. Theor. Comput. Sci. **178** (1997) 129–154
5. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings. Cambridge University Press (2002)
6. Pinter, R.Y.: Efficient string matching. In Apostolico, A., Galil, Z., eds.: Combinatorial Algorithms on Words, NATO Advanced Science Institute Series F: Computer and System Sciences, vol. 12. (1985) 11–29
7. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communcations of the ACM **18** (1975) 333–340
8. Altinel, M., Franklin, M.J.: Efficient filtering of XML documents for selective dissemination of information. In: VLDB 2000, Proc. of 26th Internat. Conf. on Very Large Data Bases. (2000) 53–64
9. YFilter: Filtering and transformation for high-volume XML message brokering. (Department of Computer Science, University of Massachusetts, Amherst) `yfilter.cs.umass.edu`.
10. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst. **28** (2003) 467–516
11. 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** (2004) 752–788
12. Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Schema-conscious filtering of XML documents. In: EDBT 2009, Proc. of the 12th Internat. Conf. on Extending Database Technology. (2009) 970–981
13. Sax Project Organization: Simple API for XML (2001) `www.saxproject.org`.
14. Suciu, D.: XML data repository. The Database Research Group, University of Washington (2006) `www.cs.washington.edu/research/xmldatasets/`.
15. NewsML: News exchange format. (International Press Telecommunications Council) `www.newsml.org`.