Department of Computer Science and Engineering

Algorithms for XML Filtering

Panu Silvasti

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 85/2011

Algorithms for XML Filtering

Panu Silvasti

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the the Faculty of Information and Natural Sciences for public examination and debate in Auditorium T2 at the Aalto University School of of Science (Espoo, Finland) on the 28th of October 2011 at 12 noon.

Aalto University
School of of Science
Department of Computer Science and Engineering

Supervisor
Prof. Eljas Soisalon-Soininen

Instructor
Prof. Seppo Sippu

Preliminary examiners
Prof. Pekka Kilpeläinen, University of Kuopio, Finland
Prof. Veli Mäkinen, University of Helsinki, Finland

Opponent
Prof. Dr. Alexander Markowetz, University of Bonn, Germany

Aalto University publication series
DOCTORAL DISSERTATIONS 85/2011

ISBN 978-952-60-4286-2 (pdf)
ISBN 978-952-60-4285-5 (printed)
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934 (printed)

Aalto Print
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Panu Silvasti

Name of the doctoral dissertation
Algorithms for XML Filtering

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 85/2011

Field of research Software Systems

Manuscript submitted 6 June 2011 **Manuscript revised** 21 September 2011

Date of the defence 28 October 2011 **Language** English

☒ **Monograph** ☐ **Article dissertation (summary + original articles)**

Abstract

In a publish-subscribe system based on XML filtering, the subscriber profiles are usually specified by filters written in the XPath language. The system processes the stream of XML documents and delivers to subscribers a notification or the content of those documents that match the filters. The number of interested subscribers and their stored profiles can be very large, thousands or even millions. In this case, the scalability of the system is critical.

In this thesis, we develop several algorithms for XML filtering with linear XPath expressions. The algorithms are based on a backtracking Aho-Corasick pattern-matching automaton (PMA) built from ``keywords'' extracted from the filters, where a keyword is a maximal substring consisting only of XML element names. The output function of the PMA indicates which keyword occurrences of which filter are recognized at a given state. Our best results have been obtained by using a dynamically changing output function, which is dynamically updated during the processing of the input document.

We have conducted an extensive performance study in which we compared our filtering algorithms with YFilter and the lazy DFA, two well-known automata-based filtering methods. With a non-recursive XML data set, PMA-based filtering is tens of times more efficient than YFilter and also significantly more efficient than the lazy DFA. With a slightly recursive data set PMA-based filtering has the same performance as the lazy DFA and it is significantly more efficient than YFilter.

We have also developed an optimization method called filter pruning. This method improves the performance of filtering by utilizing knowledge about the XML document type definition (DTD) to simplify the filters. The optimization algorithm takes as input a DTD and a set of linear XPath filters and produces a set of pruned linear XPath filters that contain as few wildcards and descendant operators as possible. With a non-recursive data set and with a slightly recursive data set the filter-pruning method yielded a tenfold increase in the filtering speed of the PMA-based algorithms and a hundredfold increase with YFilter and the lazy DFA. Filter pruning can also increase the filtering speed in the case of highly recursive data sets.

Keywords XML stream processing, string processing, XPath, automata-based filtering

ISBN (printed) 978-952-60-4285-5 **ISBN (pdf)** 978-952-60-4286-2

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 147 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Panu Silvasti

Väitöskirjan nimi
Algoritmeja XML-dokumenttien suodattamiseen

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 85/2011

Tutkimusala Ohjelmistojärjestelmät

Käsikirjoituksen pvm 06.06.2011 **Korjatun käsikirjoituksen pvm** 21.09.2011

Väitöspäivä 28.10.2011 **Kieli** Englanti

☒ **Monografia** ☐ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

XML-dokumenttien suodattamiseen perustuvassa julkaisu-tilaus -järjestelmässä suodattimet esitetään useimmiten XPath-kielellä. Järjestelmä käsittelee XML-dokumenttien virtaa ja lähettää tilaajille tiedon suodattimien läpäisemistä dokumenteista. Tilaajia ja heidän tallentamiaan suodattimia voi olla paljon, jopa tuhansia tai miljoonia, jolloin järjestelmän skaalautuvuus suhteessa suodattimien lukumäärään muodostuu hyvin tärkeäksi ominaisuudeksi.

Tässä väitöskirjassa esitetään useita algoritmeja XML-dokumenttien suodattamiseen lineaarisilla XPath-lausekkeilla. Algoritmit pohjautuvat peruuttavaan Aho-Corasick-automaattiin (pattern matching automaton, PMA). Automaatti on muodostettu suodattimissa esiintyvistä avainsanoista, missä avainsana on pisin vain XML-elementtien nimiä sisältävä osajono. Automaatin tulostusfunktio määrittelee, mitkä avainsanat tunnistetaan missäkin tilassa. Parhaat koetulokset on saavutettu algoritmilla, jossa tulostusfunktiota muutetaan dynaamisesti syötedokumentin prosessoinnin aikana.

Väitöskirjassa on suoritettu laaja kokeellinen tutkimus, jossa on vertailtu PMA-pohjaisia suodatusalgoritmeja kahteen tunnettuun XML-suodatusalgoritmiin, YFilteriin ja lazy DFA:han. Ei-rekursiivisella XML-aineistolla PMA-pohjainen suodattaminen on kymmeniä kertoja tehokkaampi menetelmä kuin YFilter ja selvästi tehokkaampi kuin lazy DFA. Lievästi rekursiivisella aineistolla PMA-pohjainen suodattaminen on edelleen selvästi tehokkaampi menetelmä kuin YFilter ja yhtä tehokas kuin lazy DFA.

Väitöskirjassa esitetään myös suodattimien operaattoreiden karsimismenetelmä, jossa suodattamista tehostetaan käyttämällä hyväksi dokumenttien tiedossa olevaan syntaksia (DTD:tä). Menetelmässä lineaarinen XPath-suodatin korvataan joukolla karsittuja lineaarisia suodattimia, joissa on mahdollisimman vähän jokerimerkkejä (*) ja esi-isä-jälkeläis-operaattoreita (//). Ei-rekursiivisella ja lievästi rekursiivisella XML-aineistolla karsimismenetelmä kasvattaa PMA-pohjaisen suodattamisen tehokkuutta yli kymmenkertaiseksi ja YFilterin sekä lazy DFA:n tehokkuutta yli satakertaiseksi. Kokeissa havaittiin, että karsimismenetelmä voi tehostaa suodattamisen tehokkuutta myös erittäin rekursiivistenkin aineistojen kanssa.

Avainsanat XML-tietovirran käsittely, merkkijononkäsittely, XPath, automaattipohjainen suodattaminen

ISBN (painettu) 978-952-60-4285-5 **ISBN (pdf)** 978-952-60-4286-2

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2011

Sivumäärä 147 **Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/

Preface

I wish to thank my supervisor, Professor Eljas Soisalon-Soininen, for introducing the research topic of XML filtering to me five years ago. Since then we have developed several new algorithms in this research field with him and with my instructor, Professor Seppo Sippu from the University of Helsinki. This work has been interesting and satisfying. I wish to thank them for always having time for me, for answering all my questions, and for giving constructive feedback on my work. I am grateful for having had the opportunity to work in their guidance. I have learned so much during the doctoral research.

I would also like to thank Professors Pekka Kilpeläinen (University of Kuopio) and Veli Mäkinen (University of Helsinki) for their detailed and valuable comments on the manuscript, and Professor Alexander Markowetz (Rheinische Friedrich-Wilhelms-Universität, Bonn) for kindly agreeing to be the opponent.

The research presented in this dissertation was carried out at the Department of Computer Science and Engineering at the Aalto University School of Science. The research was partly funded by the Academy of Finland. I am grateful for the possibility to have been primarily able to concentrate on the research work and for being able to visit several distinguished scientific conferences around the world.

I wish to thank all my colleagues at the computer science laboratory for all the discussions we have had about doing computer science research. Especially I would like to thank Dr. Riku Saikkonen, who gave me important comments in the early stages of the research, and who, together with Dr. Tuukka Haapasalo, helped with the LaTeX word processor.

Finally, I would like to thank my family for their love and support. This dissertation would not have been completed without their encouragement.

Jyväskylä, September 2011

Panu Silvasti

Contents

CHAPTER 1

Introduction

A *publish-subscribe system* consists of one or more publishers and many subscribers, where the publishers provide a stream of documents and the subscribers specify their interests with filters that match some of those documents. The system processes the stream of documents and delivers to subscribers a notification or the content of those documents that match the filters. Publish-subscribe systems have emerged in everyday use on the Internet; examples include *Google Alerts* [1] and *Yahoo! Alerts* [3].

The document stream is a potentially unbounded sequence of documents. In a traditional database management system the data is persistent and queries volatile, but in a data stream management system, such as a publish-subscribe system, the queries (or filters) are persistent and the incoming data stream volatile. Saving the document stream onto disk and indexing it for querying with the filters is not feasible: the stream must be processed online. This dissertation presents new algorithms that can be used for filtering the data stream online.

In a publish-subscribe system based on XML filtering, the filters are usually specified in the XPath language [11]. Designing efficient techniques for the filtering of XML documents has received much attention [6, 8, 15, 19, 21, 24, 28, 30, 33, 37–40, 49, 51, 72, 73], and XML filtering techniques have been applied in areas such as routing real-time air traffic control data [68].

1.1 The Filtering Problem

The primary problem addressed in this dissertation is defined as the *XML filtering problem*: given a set of XPath expressions and an input stream of XML documents, identify the filters that match the documents in the stream. More specifically, for each document in the input stream, we must determine the set of filters that match the document.

The goal is to build a system that is able to process the input stream at a sustained throughput even when the number of filters is large. The number of filters can be very large, thousands or even millions. The scalability of the system with respect to the number of filters is critical.

Two other problems related to the filtering problem are the *language recognition problem* and the *online dictionary-matching problem*. In language recognition the task is to determine whether or not the input document belongs to the language described by the union of the filters. This problem is easier to solve than the filtering problem, because it is not necessary to identify which of the filters match. In online dictionary matching for XML document streams the problem is to determine, for each XML document in the stream, all occurrences of all the XPath patterns in an online fashion.

This dissertation presents new algorithms for solving the filtering problem for linear XPath filters without predicates. Linear XPath filters do not have branches in their query trees. We also show how these algorithms can be extended to solve the online dictionary-matching problem.

1.2 Automata-based Filtering

Several approaches to XML filtering use a finite automaton as a basis of the filtering algorithm [6, 21, 24, 28, 30, 49, 51, 72]. Diao et al. [24] report an XPath filtering method, called YFilter, that applies nondeterministic finite automata (NFAs). YFilter is an improvement upon its predecessor, called XFilter [6], which uses a separate NFA for each filter but executes them simultaneously in processing the input document. YFilter uses a single NFA that combines the effect of the individual NFAs and achieves considerable improvements in performance by path-sharing; that is, by merging states that correspond to common prefixes in different query paths.

The lazy DFA algorithm by Green et al. [28] is based on a single deterministic finite automaton (DFA). The state explosion of the DFA is tackled by constructing the DFA lazily. In other words, the DFA is constructed at runtime, on demand: 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 extremely well in many cases, when the incoming XML documents obey a schema or a document type definition (DTD) that is nonrecursive or contains only simple cycles (a cycle is simple if its nodes do not occur

in other cycles).

The new XML filtering algorithms presented in this dissertation are based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick [5] pattern-matching automaton (PMA). The PMA is constructed from the set of filters in the preprocessing phase of the algorithm. The automaton has a size linear in the sum of the sizes of the filters.

The classic Aho–Corasick PMA solves the online dictionary-matching problem for linear text, where for a set of keywords and an input text, the task is to identify all occurrences of the keywords in the text. The starting point of our filtering algorithms is a simple backtracking PMA that can efficiently process linear XPath filters without wildcards ("*") and non-leading descendant operators ("//") [63]. An extension of this algorithm can also process linear XPath filters having wildcards and descendant operators [66]. These algorithms have a static output function, as is the case with the classic Aho–Corasick PMA. We also present a new Aho–Corasick-based filtering algorithm (later referred to as the *dynamic PMA*) that has a dynamically changing output function; the output function is dynamically updated during processing of the input document. A preliminary version of our dynamic PMA has been published in a conference article [65]. This algorithm can process linear XPath filters that have descendant operators, but not wildcards. Recently we have further developed this algorithm so as to also handle wildcards [67].

Our PMA-based filtering algorithms have been experimentally benchmarked with YFilter [24] and the lazy DFA [28] using XML data sets obtained from the XML Data Repository of the University of Washington [71]. With a non-recursive data set PMA-based filtering is 40 times more efficient than YFilter and 5 times more efficient than the lazy DFA. With a slightly recursive data set the performance of PMA-based filtering is comparable to that of the lazy DFA and three times better than that of YFilter. However, with a highly complex data set YFilter was found to be more scalable. The lazy DFA is inapplicable with this data set.

1.3 Schema-Conscious Filtering

The above-mentioned filtering methods are general in that the input documents to be filtered are not required to comply with any predefined schema or DTD; they are only expected to obey the generic syntax of XML. However, in practice the documents published by a specific site may very well be restricted to a few different topics and described by a

specific XML schema. This raises the question of whether filtering speed or throughput could be improved by utilizing knowledge about a DTD in building the filtering automaton. Fernández and Suciu [25] have presented a technique called *query pruning* for optimizing regular path expressions with graph schemas. Inspired by their work, we have developed an optimization method, called *filter pruning* [63–65], that takes as input a DTD and a set of linear XPath filters and produces a set of *pruned* linear XPath filters that contain as few wildcards and descendant operators as possible without increasing the number of different filters too much. The set of pruned filters is equivalent to the set of original filters in that each original filter is represented by the union of a set of pruned filters that matches the same set of XML documents, provided that the documents obey the DTD.

Our experiments with data sets obtained from the XML Data Repository [71] show that filter pruning can significantly accelerate automaton-based filtering. In two recent conference articles [64, 65] we have published experimental results of the effect of filter pruning on the performance of YFilter [24] and the lazy DFA [28]. However, our recent experiments show even better performance increase than presented in these articles. With a non-recursive data set and with a slightly recursive data set the filter-pruning method yielded an increase in the filtering speed of the PMA-based algorithms by a factor of 10–70. With the same data sets the filtering performance of YFilter and the lazy DFA increased even more than hundredfold.

The DTDs of the above-mentioned data sets are simple and tree-like. They have only a small number of nodes with many incoming edges. In addition to these data sets, experiments have been run with complex and highly recursive data sets [35, 71]. Imposing some simple conditions stating when an operator may be eliminated, a polynomial bound can be guaranteed on the total size of the pruned filters. Our experiments show that pruning can increase the filtering speed of PMA-based filtering and YFilter also in the case of complex and highly recursive data sets.

In this dissertation we give a thorough overview of results obtained by experimenting with the filter-pruning method and with our PMA-based algorithms, YFilter, and the lazy DFA.

1.4 Organization of the Dissertation

This dissertation is organized as follows. The next chapter gives an overview of existing XML filtering algorithms. In Chapter 3 we present

the new PMA-based XML filtering algorithms. Chapter 4 contains an experimental study of the PMA-based filtering, benchmarking the PMA-based algorithms with YFilter [24] and the lazy DFA [28]. Chapter 5 describes the filter-pruning optimization, and in Chapter 6 we present an experimental study of filter pruning. Chapter 7 discusses the evaluation of branching filters (twig filters) and filters with value-based predicates. Chapter 8 concludes the dissertation.

CHAPTER 2

XML Filtering

This chapter gives an overview of XML filtering literature. The XML filtering algorithms can be classified into automata-based algorithms, index-based algorithms, sequence-based algorithms, and other approaches. As the focus of this dissertation is on automata-based filtering, a more thorough overview of the automata-based algorithms is given. The research area is fairly new; the first XML filtering algorithm [6] was published in 2000.

2.1 XPath Filters and XML Streams

The XML filtering systems presented in the literature usually support some subset of the XPath [11] language in expressing the profiles. Some systems can process only *linear* XPath expressions (e.g. the lazy DFA algorithm [28]), for which the query tree is a linear list. An example of such a linear query is

$$/a/b/c[text() = 'C1']$$

that finds all c elements occurring exactly on a path `<a><b><c>` and having a text node child '$C1$'. The query tree for the expression is presented in Figure 2.1(a).

One popular research subject in XML filtering has been efficient processing of twig expressions [6, 24, 30, 38]. A *twig expression* can have many nested paths and the query is a branching tree. An example of such an expression is

$$/a/b[c[text() = 'C1']]/e[text() = 'C2']$$

that finds XML trees whose c element on a path `<a><b><c>` has a text node child '$C1$' and `e` element on a path `<a><b><e>` has text node child

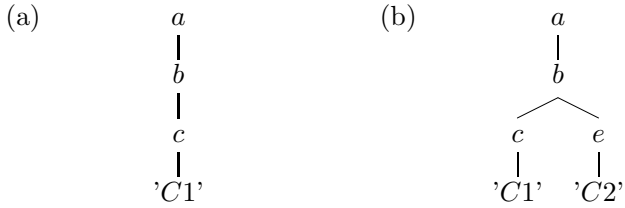


Figure 2.1. Query tree of a linear XPath expression (a) and query tree of a twig XPath expression (b).

'$C2$'. The prefix path `<a><b>` has to be the same for both paths. The query tree of the expression is presented in Figure 2.1(b). As can be seen, the branching node of the twig expression is b. Evaluation of twig filters will be discussed in more detail in Section 7.1.

XML filtering systems usually evaluate XPath filters in two phases: structure navigation and evaluation of value-based predicates. The structure part of the filter $/a/b/c[text() =$'$C1$'$]$ is $/a/b/c/$ and the value-based predicate is $text() =$'$C1$'. In this chapter the emphasis will be on methods for structure navigation. Different methods for predicate evaluation will be discussed in Section 7.2.

The "/" operator between nodes denotes a parent-child relationship, as in a/b. The "//" denotes an ancestor-descendant relationship; for example $a//c$ locates all c elements occurring somewhere under an a element. The wildcard "$*$" denotes any element name. For example, the query $/a/b/*[text() =$'$C1$'$]$ finds all elements occurring under a child element of a b element and having text node child with value '$C1$'. The descendant and wildcard operators in the XPath filters complicate XML filtering. Most of the algorithms presented in this chapter can handle these operators. In Section 3.1 it is shown how linear XPath filters without wildcards and non-leading descendant operators can be processed with a simple and efficient algorithm.

XPath allows several comparison operators in value-based predicates, such as $=$, $<$, $\leq$, $>$, $\geq$, and $\neq$. Logical operators such as AND, OR and NOT can also be specified in the filters. Some XML filtering systems support only the equality operator [28], and some a wider range of operators [30].

XPath also has aggregate functions for counting the sum, average, minimum or maximum of element values, or the count of elements. Only few of the algorithms presented in the literature of XML filtering with XPath filters, however, support evaluation of such aggregate functions [54].

Most XML filtering systems use an event-based parser, such as a SAX parser [62], for processing the input XML documents. The input for the

filtering algorithm is then the event stream produced by the parser. The parser generates the following types of events: *startDocument*, *startElement*, *characters*, *endElement*, and *endDocument*. For example, the XML document `<a><b>4</b></a>` is converted by the parser into the following sequence of events:

```
startDocument()
startElement(a)
startElement(b)
characters("4")
endElement(b)
endElement(a)
endDocument()
```

In filtering of XML documents, the task is to identify matching XPath filters for each XML document in the input stream. For example, given a filter workload consisting of the filters $Q_1 = /a/b/c$, $Q_2 = /e/f$, $Q_3 = /a//c$, $Q_4 = /a//d$, $Q_5 = /a/*/c$, and $Q_6 = //c$, and an XML input document `<a><b><c></c></b></a>`, then filters Q_1, Q_3, Q_5, and Q_6 are found to match the document.

2.2 Automata-based Filtering

Several approaches to XML filtering with XPath filters use an automaton as a basis of the filtering algorithm. A linear XPath filter can be represented as a nondeterministic finite automaton (NFA) [32]. Figure 2.2 shows example filters and corresponding NFAs. Building an NFA of a linear XPath filter is straightforward. If the filter begins with a descendant operator, then the NFA will have a "$*$" loop in the initial state, where "$*$" denotes any element (filter Q_3). A descendant operator appearing in the middle of a filter is represented as an "ϵ" transition (i.e., a transition on the empty string) to a new state that has a "$*$" loop (filters Q_2 and Q_4).

Several XML filtering algorithms build a single combined NFA for the set of all the XPath filters [21, 24, 49, 72]. Simulating a single combined NFA is more efficient than simulating several separate NFAs. A combined NFA is build by using the *path sharing* principle; it means that for a set of common prefixes of filters are merged into a single path in the filtering automaton [24]. Figure 2.3 shows the combined NFA of the example filters of Figure 2.2; it can be seen that filters Q_1 and Q_2 share the states for the prefix $/a/b$.

Benefits of an NFA-based algorithm are that its size is linear in the size of the XPath filters and that it is easy to insert and delete filters, which are important properties for an XML filtering system.

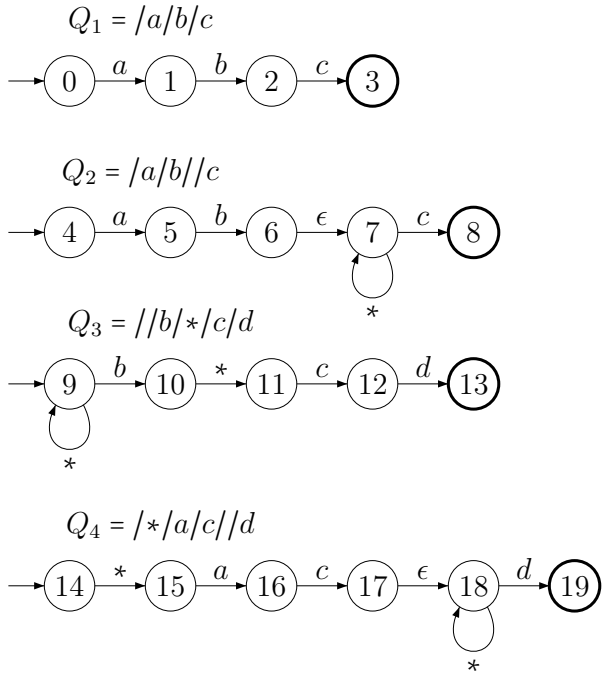


Figure 2.2. Example filters and corresponding NFAs.

An NFA can be converted into a deterministic finite automaton (DFA) [32], which is the most efficient machine for language recognition. A DFA can process one input symbol (or SAX event) in $O(1)$ time. However, the size of a DFA can be exponential in the number and size of the XPath filters.

As an example, consider the set of filters $Q_1 = //a_1$, $Q_2 = //a_2, \ldots, Q_n = //a_n$. Filtering with respect to these filters can be done by a DFA that accepts the language $L = a_1 \cup a_2 \cup \ldots \cup a_n$, and decides by final states which of the n filters are matched. The minimized DFA for the case when $n = 3$ is shown in Fig 2.4. At final state $i, 1 \leq i \leq 3$, only filter Q_i is recognized, at final states 4, 5 and 6, the filter sets $\{Q_1, Q_2\}$, $\{Q_1, Q_3\}$, and $\{Q_2, Q_3\}$, respectively, are recognized, and at final state 7, the set $\{Q_1, Q_2, Q_3\}$ is recognized. Note that the DFA of Figure 2.4 cannot be further minimized, because the final states all accept different subsets of $\{Q_1, Q_2, Q_3\}$.

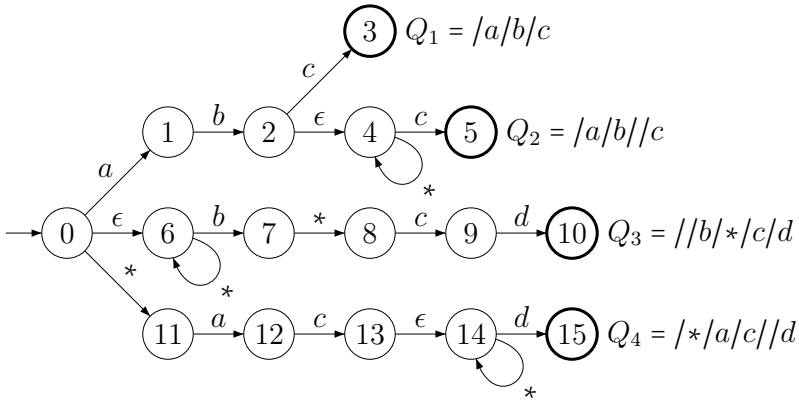


Figure 2.3. A combined NFA built with the path-sharing principle.

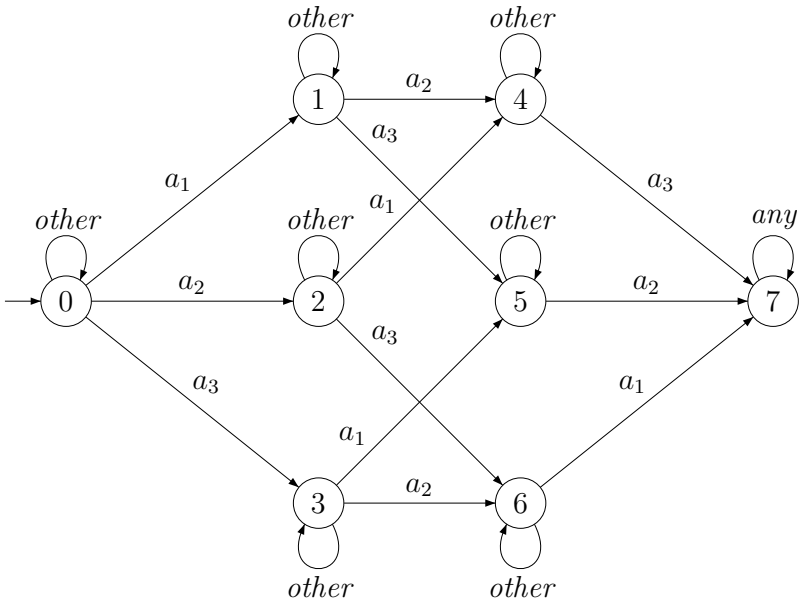


Figure 2.4. Minimal DFA that solves the filtering problem for filters $Q_1 = //a_1$, $Q_2 = //a_2$, and $Q_3 = //a_3$.

2.3 NFA-based Filtering

2.3.1 XFilter

XFilter [6] by Altinel and Franklin is a pioneering automata-based XML filtering algorithm. It is based on building a separate NFA for each XPath filter and executing them simultaneously during processing of the input document. XFilter can process linear and twig filters having value-based predicates with comparison operators.

With XFilter matching of linear XPath filters without predicates works as follows. The workload of example filters Q_1 = $/a/b/c$, Q_2 = $/a/b//c$, Q_3 = $//b/*/c/d$, and Q_4 = $/*/a/c//d$ is represented as a machine similar to Figure 2.2. For an XML document `<a><b></b></a>`, when the SAX parser encounters the start-tag of element `a`, states 1, 5, 9, and 15 will be accessed. When processing the start-tag of element `b`, states 2, 6, 10 will be accessed. When an accepting state of an NFA is reached, then the corresponding filter matches the XML document. Processing an element end-tag causes backtracking of NFAs; the states of the automata are changed into the states that were active before processing of the corresponding start-tag started. For example, when processing the end-tag of element `b`, states 1, 5, 9, and 15 will be selected again.

In the worst case, one SAX event can trigger n state transitions, where n is the number of XPath filters.

2.3.2 YFilter

YFilter by Diao et al. [24] is a successor of XFilter. YFilter builds a single combined NFA for the set of all the XPath filters and it achieves considerable improvements in performance by path sharing.

With YFilter the SAX events are processed as follows. When the beginning of an XML element is encountered, the set of current states will be pushed onto the stack and new states will be the ones accessed from current states by the element label (current states are the active states of the NFA). For example, with XML document `<a><b><c></c></b></a>`, when processing the start-tag of element `a` with the NFA of Figure 2.3, states 1, 6 and 11 will be accessed and state 0 will be pushed onto the stack. When processing the start-tag of element `b` states 2, 4, 6, and 7 will be reached and the set $\{1, 6, 11\}$ will be pushed onto the stack. On the start-tag of element `c` states 3, 4, 5, 6, and 8 will be set as the current states; now we find that filters Q_1 and Q_2 have matched the input document.

When the end of an XML element is encountered, the automaton has to be backtracked into the state it was before processing the beginning of the element. This is done by setting the set of states on top of the stack as the current state of the automaton, and popping the stack. For example, when processing the end-tag of element `c`, the set $\{1, 6, 11\}$ will be set as the current states and removed from the top of the stack.

With YFilter, one SAX event can also trigger n state transitions, where n is the number of XPath filters. However, in practice path sharing works effectively and YFilter is much more efficient than XFilter [24].

2.3.3 Other NFA-based Algorithms

The BUFF algorithm by Moro et al. [49] is also based on representing XPath filters by a combined NFA. However, BUFF evaluates XML trees bottom-up. The XML document tree is thus evaluated in post-order instead of the pre-order traversal used in top-down approaches. For this the XPath filters are reversed before building the NFA. For example, filter $//a/b/c//d$ is turned into $//d//c/b/a$ before inserting it into the NFA. While YFilter's NFA applies path-sharing, BUFF thus groups filters according to their common suffixes. In the experiments [49] BUFF was found to be even 20% more efficient than YFilter's top-down NFA.

Chen et al. [21] present the GFilter system, which also evaluates XML paths bottom-up. GFilter builds a combined NFA for the set of all filters and applies both prefix- and suffix sharing. GFilter supports complex twig filters and value-based predicates. It can also process GTP (generalized tree pattern) [22] queries. In the experiments [21] GFilter was found to be more than twice as fast as YFilter when processing linear XPath filters. With twig filters GFilter was more than 50% more efficient than YFilter.

Byun et al. [17] propose an automata-based PFilter algorithm that is based on YFilter. The structure matching is done by a combined NFA as in YFilter, but the method for predicate evaluation is different (see Section 7.2).

2.4 DFA-based Filtering

2.4.1 Lazy DFA

The *lazy DFA* filtering algorithm by Green et al. [28, 29] is based on a deterministic finite automaton (DFA). The state explosion of the DFA is tackled by constructing the DFA lazily during processing of the XML

document stream. The lazy DFA can process linear XPath filters having value-based predicates of the form *text*() ='S', where S is a string constant.

The lazy DFA constructs the DFA in the following way. Each XPath filter is first compiled into an NFA. These NFAs are then combined into a single NFA similarly to YFilter (see Figure 2.3). Then, during processing of an input document, a DFA is constructed from the combined NFA. The construction of the DFA is done lazily. Initially the DFA has only the initial state, and whenever a transition is being performed into a non-existing state, this state is constructed. The main difference between the lazy DFA and the NFA-based algorithms is that the lazy DFA stores constructed states into memory whereas the NFA-based algorithms do not.

SAX events are processed as follows. 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 by 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.

Assume processing of an XML document <a$_1$><a$_3$></a$_3$></a$_1$><a$_2$></a$_2$> with the DFA of Figure 2.4. When the DFA is constructed lazily, the transitions constructed are only those that are reachable upon reading a$_1$a$_3$ or a$_2$ from the initial state and lead to states 5 and 2, respectively.

The lazy DFA is very efficient in processing XML data that has a simple tree-like DTD, such as the protein and NASA datasets [71]. However, with complex datasets (such as the treebank dataset [71]) too many states may be created and the system can run out of memory.

Onizuka [51] proposes several techniques for improving the memory usage of the DFA. He also extends the lazy DFA algorithm for processing twig filters. Onizuka has developed a clustering algorithm that divides the XPath filters into clusters according to axis types ("/", "//") at each depth level. The observation behind the clustering method is that the number of states in the DFA grows exponentially with the number of "//" operators. As the "/" axis is much more common than "//", it is reasonable to group filters containing "//" into their own clusters. A separate filtering DFA will then be constructed from each cluster, so if there are n clusters, then n DFAs will be created. In spite of the fact that a SAX event will have to be processed with each of the n DFAs, the clustering increases filtering performance. In the experiments the best performance was acquired with $n = 8$. In this case the memory usage of the DFA was reduced 40-fold when compared to $n = 1$ [51].

Altinel et al. [6] show how a prefiltering technique can enhance the

performance of the XFilter algorithm. In prefiltering each XML document is parsed twice, and XPath filters having elements not present in the XML document are not considered in filtering. Chen et al. [20] have applied prefiltering in the context of the lazy DFA algorithm. They also show how the DTD can be used to check the validity of XPath filters; filters not consistent with the DTD can be ignored in filtering. Moreover, they present techniques for optimizing the construction of lazy DFA states.

2.4.2 XPush

The XPush algorithm [30] is also based on automata; it builds a single deterministic pushdown automaton (PDA) [32] of the XPath filters. Whereas a DFA makes a state transition based on its current state and the input symbol, a transition of a PDA is based on its state, input symbol and contents of a pushdown stack. The transition function of a PDA determines when symbols are added onto the stack and when the stack is popped.

With XPush the PDA is constructed lazily. XPush builds NFA representations of the XPath filters in a way similar to YFilter and the lazy DFA algorithms. YFilter combines these NFAs into a single NFA. The lazy DFA builds a single DFA from the NFAs; as explained above, to avoid the exponential state growth the DFA is constructed lazily during processing of the XML input stream. XPush, on the other hand, combines the NFAs into a single PDA that is also constructed lazily. The NFA construction phase of YFilter, lazy DFA and XPush are similar, but the XML trees are evaluated bottom-up in XPush and top-down in YFilter and lazy DFA. YFilter and the lazy DFA share the processing of common prefixes in the structure navigation part of XPath filters. XPush focuses on eliminating redundant work in the predicate evaluation part.

In has been experimentally shown that a fully computed XPush machine processes each SAX event in constant time. However, in practice the XPush machine becomes too large and it needs to be computed lazily. It was shown that the running time and memory requirements of the lazy XPush machine are moderate. The hit ratio of successful lookups from the state transition tables not requiring construction of states versus the total number of lookups was over 90 %. It was also shown that the several optimization techniques improved the running time and reduced the number of states of the XPush machine considerably [30].

All automata-based XML filtering algorithms have characteristics of a PDA, since they all use a stack in order to backtrack the automaton on an element end-tag. However, with these algorithms the state transitions

are based only on the current state and the input symbol read, not on the contents of the stack (except when backtracking).

2.5 Hybrid Finite Automaton

The benefit of NFA-based filtering is that it consumes only a small amount of memory. The lazy DFA, on the other hand, is very efficient, but it may run out of memory in filtering complex XML data. When the lazy DFA runs out of memory, the DFA is restarted and so far constructed states are cleared from memory. With complex XML data this restarting may occur often, which consumes much processing time.

The *two-tier hybrid finite automaton* (HFA) [72] is a combination of lazy DFA and NFA. HFA uses a lazy DFA to process XML elements under a given depth and an NFA to process XML elements beyond that depth. Thus the HFA keeps in memory only the frequently accessed states.

When the two-tier HFA runs out of memory, it needs to be restarted. The cost of restarting the HFA can be reduced by a *three-tier* HFA [72]. The first tier of the HFA is a pre-expanded DFA, the second tier is the lazy DFA and the third tier is the NFA. When a memory overflow happens, only the second tier needs to be cleared to release the memory. Two parameters are specified: D_{pre} and D_{exp}. In processing XML documents (with *depth* denoting the current depth in the XML document tree), when $depth \leq D_{pre}$ the fully computed DFA is used, when $D_{pre} < depth \leq D_{exp}$ the lazy DFA tier is used, and when $depth > D_{exp}$ the NFA tier is used. Parameters D_{pre} and D_{exp} can be adjusted according to the available physical memory and the input XML stream. Setting higher values of D_{pre} decreases the time needed to restart the algorithm in case of memory overflow. Increasing the D_{exp} consumes more memory, but can increase the filtering speed.

In their experiments Sun et al. [72] concluded that when the average depth of XML documents increases, the performance of the lazy DFA decreases quickly, but the performance of the two- and three-tier HFA decrease only somewhat. On average, the performance of the hybrid approach was 30 % better than non-hybrid algorithms (YFilter [24] and lazy DFA [28]).

2.6 Other Automata-based Algorithms

He et al. [31] propose a filtering algorithm that is based on the lazy DFA. The algorithm is optimized to utilize the hardware cache of modern processors. Their basic idea is that the most frequently used states of the automaton are stored into the cache. Their experiments show that cache-consciousness can improve the performance of automata-based filtering by 16–46%. Yin et al. [76] have developed a similar technique for improving the performance of the lazy DFA; their method is called *frequent access technology*.

The XSQ [54] system uses an NFA augmented with a buffer. XSQ supports value-based predicates and aggregations. In a recent article, Onizuka [52] also proposes an NFA-based algorithm for evaluating XPath queries over XML streams. However, these algorithms can evaluate only one XPath query at a time.

Miliaraki et al. [45, 46] present a method for distributing the execution of YFilter by using distributed hash tables [69]. They show how to balance the filtering load across several servers and in this way enable efficient parallel processing of millions of XPath filters. Uchiyama et al. [74] have developed a method for distributing the execution of the lazy DFA algorithm.

2.7 Other Approaches

Some XML filtering algorithms are based on index structures. XTrie [19] builds a trie-based index structure of the keywords occurring in XPath filters, where a keyword is a maximal substring consisting only of XML element names. The Hybrid approach described by Diao et al. [24] uses a similar idea of a trie index. Bruno et al. [15] propose an algorithm called Index-Filter that indexes the XML input stream to be filtered. The main difference between Index-Filter and the trie-based algorithms is that Index-Filter creates a B-tree index of the element occurrences in the XML documents whereas the trie-based algorithms create an index structure of the XPath filters. An open question with Index-Filter is how it works with very large input documents. Its index structure may become too large to fit into the main memory.

The AFilter system by Candan et al. [18] can evaluate linear XPath filters without predicates. It is also based on a trie structure and it exploits both prefix and suffix commonalities in XPath filters. In the experiments, AFilter was found to be more than twice as fast as YFilter.

FiST [38] and its successors iFiST [40] and pFiST [39] are based on encoding the XML input stream and the XPath filters into Prüfer [56] sequences. The matching is then performed bottom-up by using a subsequence matching algorithm. BoXFilter [49] is a similar sequence-based algorithm that differs from the FiST family in that it groups filters according to the similarity of their Prüfer encodings. XFIS system by Antonellis and Makris [8] is yet another sequence-based XML filtering algorithm.

Tian et al. [73] propose an XML filtering system in which the XPath filters are stored into the tables of a relational database. The system supports processing of value-based predicates and twig filters. Using a relational database provides good scalability and is memory-efficient. The matching algorithm exploits commonalities in both XPath filters and also in the XML paths to be matched.

Hou and Jacobsen [33] present a predicate-based XML filtering algorithm. In their system each XPath filter is represented as an ordered set of predicates. A predicate is a triple (a, o, v) that determines the relation of adjacent elements in the XPath filter. In the triple, a is either a tag name or a pair of tag names, o is $=$ or $\geq$ operator, and v is a distance value. For example, the XPath expression $/*/a$ is represented as an absolute predicate $(a, =, 2)$, and expression $/*//a$ as $(a, \geq, 2)$. A relation $a/*/b$ is represented as predicate $(d(a, b), =, 2)$ and $a//b$ as $(d(a, b), \geq, 1)$. The predicate in this sense has a meaning different from that of a value-based predicate in an XPath filter. However, Hou and Jacobsen also show how value-based predicates and twig filters can be processed with the algorithm.

In the algorithm of Hou and Jacobsen [33] the XML stream is processed with a SAX parser, but each path in the document tree is considered separately. An XML path is decomposed into a set of attribute-value pairs and these pairs are matched against the predicates. The algorithm stores and evaluates only distinct predicates. In the experiments, the predicate-based filtering algorithm was found to have filtering performance comparable to YFilter and Index-Filter.

Mitra et al. [48] show how XML filtering can be done with field-programmable gate arrays (FPGAs). The filtering algorithm is implemented in VHDL (very high speed integrated circuit hardware description language) and deployed on the FGPA board. The idea is to represent the XPath filters as regular expressions, which can be translated into VHDL [47]. This architecture has a high degree of parallelism and it is very efficient; in the experiments the hardware implementation was found to be around 100 times more efficient than YFilter. However, extending the FPGA architecture for processing twig filters has been left for future

work.

Gong et al. [27] present an XML filtering algorithm that is based on Bloom filters [12]. A Bloom filter is a data structure that can be used to efficiently test whether or not an element is member of a set. In this case linear XPath expressions are encoded into a Bloom filter. The filtering result is approximate, since there is a small probability that a Bloom filter produces a false positive match. The system is more efficient than YFilter with shallow XML data, but when the depth of the XML documents increases, the performance of the Bloom-filter-based algorithm decreases quickly. One benefit of the algorithm is that inserting and deleting filters is fast.

Some recent systems for XML filtering use web ontology languages (OWL) [10] for XPath filter matching. M-Filter [34] can process twig filters and the system by Saigaonkar et al. [60] twig filters having value-based predicates. In the experiments M-Filter was found to perform better than YFilter. Saigaonkar et al. did not yet present a performance study. Qeli and Freislebeln [57] present an idea of using aspect-oriented programming for implementing the filtering algorithm.

A problem related to XML filtering with XPath filters is the case when filters are described in the XQuery [13] language. The XSM filtering system presented by Ludäscher et al. [44] is based on building transducers of XQuery filters. The filter workload is compiled into a C program that simulates the execution of a transducer. The XSM does not support the descendant axis in a query.

Koch et al. [37] also consider evaluation of XQuery and XPath filters over XML streams. They propose a prefiltering technique that is based on the Boyer-Moore [14] and Commentz-Walter [23] string-matching algorithms. Instead of tokenizing the XML input documents by using a SAX parser they process the documents as an unparsed character stream. Their prefiltering technique can significantly speed up the QizX [2] XQuery processor and the SPEX [50] XPath engine. However, they only consider the case of evaluating one filter at a time.

The TurboXPath algorithm by Josifovski et al. [36] is another system for evaluating XQuery queries over XML streams. The algorithm compiles the input query into a single parse tree and matches the input XML document against the parse tree nodes. TurboXPath also evaluates only one query at a time and it is not known how it would apply to XML filtering, when the number of boolean queries (or filters) can be large.

2.8 Summary

Table 2.1 presents a summary of the above-described XML filtering algorithms. The table describes the basic idea of the filtering engine, XML parsing scheme used, and whether or not predicates and twigs are supported. Most of the algorithms use a SAX parsing scheme either with top-down or bottom-up evaluation. If bottom-up parsing is used, buffering of SAX events is required. Tian et al. [73] and Jacobsen et al. [33] consider processing each path in the XML document separately. Algorithms by Koch et al. [37] and Mitra et al. [48] do not use SAX, but process XML documents as character stream.

Algorithms capable of handling twig filters [8, 15, 19, 24, 38, 40, 49] are discussed in more detail in Section 7.1, where the evaluation of twig filters is discussed. In Section 7.2 we discuss predicate evaluation and review techniques for predicate evaluation presented with YFilter [24], XPush [30], and PFilter [17].

System	Filtering engine	Input stream parsing	Predicates	Twigs	Notes
XFilter [6]	builds an FSM for each filter	SAX, top-down	yes	yes	
YFilter [24]	single NFA for all the filters	SAX, top-down	yes	yes	
BUFF [49]	single NFA for all the filters	SAX, bottom-up	yes	yes	
GFilter [21]	single NFA for all the filters	SAX, bottom-up	yes	yes	GTP filters
PFilter [17]	single NFA for all the filters	SAX, top-down	yes	yes	built upon YFilter
Lazy DFA [28]	single DFA that is constructed lazily	SAX, top-down	yes	-	
XPush [30]	single PDA that is constructed lazily	SAX, bottom-up	yes	yes	
HFilter [72]	combination of DFA and NFA	SAX, top-down	-	-	
cache-conscious automata [31]	utilizes the processor cache	SAX, top-down	-	-	
XSQ [54]	uses an NFA augmented with a buffer	SAX, top-down	yes	yes	only one filter at a time
DHT [45]	distibutes processing of YFilter	SAX, top-down	yes	-	built upon YFilter
Uchiyama et al. [74]	distibutes processing of lazy DFA	SAX, top-down	yes	-	built upon lazy DFA
Lazy XTrie [19]	trie-based indexing of the profiles	SAX, bottom-up	yes	yes	
Index-Filter [15]	indexes the XML stream to be filtered	SAX, bottom-up	yes	yes	
AFilter [18]	trie-based indexing of the profiles	SAX, top-down	-	-	
FiST family [38–40]	subsequence matching	SAX, bottom-up	yes	yes	
BoXFilter [49]	subsequence matching	SAX, bottom-up	yes	yes	
XFIS [8]	subsequence matching	SAX, bottom-up	-	yes	
Tian et al. [73]	stores profiles into relational database	SAX, each path separately	yes	yes	
Hou and Jacobsen [33]	filters are represented as ordered sets of predicates	SAX, each path separately	yes	yes	
Mitra et al. [48]	FPGA-based filtering	character stream	-	-	
Gong et al. [27]	uses Bloom filters for approximate filtering	SAX, top-down	-	-	
M-Filter [34]	uses a web ontology language	SAX, top-down	-	yes	
Koch et al. [37]	string matching approach	character stream	-	-	XQuery filters
TurboXPath [36]	compiles the input query into a parse tree	SAX, top-down	yes	yes	one XQuery filter at a time

Table 2.1. XML filtering systems for filtering with XPath and XQuery filters.

CHAPTER 3

XML Filtering by Pattern-Matching-Automata

In this chapter we present new algorithms for solving the filtering problem for linear XPath filters without predicates. The new algorithms are based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick [5] pattern-matching automaton (PMA). The PMA is constructed from the set of filters in the preprocessing phase of the algorithm. The automaton has a size linear in the sum of the sizes of the filters.

In Section 3.1 we describe a simple and efficient version of the PMA-based algorithm that can process linear XPath filters without wildcards and non-leading descendant operators [63]. We call this algorithm the *bare AC*. Then in Section 3.2 we extend the bare AC so as to handle also wildcards and non-leading descendant operators [66]. This algorithm is called the *static PMA*. In Section 3.3 we present a more efficient version of the algorithm that uses dynamic modification of the output function of the PMA during processing of the input document. We have published this algorithm in a recent conference article [67], and it is called the *dynamic PMA*. In that article we also outlined an optimization called "fast backtracking" (*PMA FB*); this optimization is described in detail in Section 3.4.

A problem related to XML filtering is the online dictionary-matching problem for XML document streams, where the task is to locate all occurrences of all XPath patterns in the input XML document. In Section 3.5 we extend the algorithm of Section 3.4 for online dictionary matching of XML documents.

3.1 Backtracking Aho–Corasick Pattern-Matching Automaton

In this section we describe the simple and efficient bare AC algorithm for matching linear XPath filters without wildcards and non-leading descendant operators.

3.1.1 The Algorithm

The classical Aho–Corasick PMA [5] for a finite set W of nonempty strings called *keywords* over a finite alphabet Σ is a deterministic linear-time finite-state recognizer of the regular language $\Sigma^* W \Sigma^*$. The size of the PMA is $O(|W|)$, where $|W|$ denotes the sum of the lengths of all keywords in W. In processing input string x, the PMA makes at most $2|x|$ moves.

For each prefix y of some keyword in W, the Aho–Corasick PMA has a unique state, denoted by $state(y)$, different from all $state(y')$ where $y' \neq y$. The state $state(\epsilon)$, where ϵ is the empty string, is the *initial state* of the PMA. The number of states in the PMA is at most $|W|+1$ and the states are numbered with positive integers.

The *goto function* of the PMA is defined by the equation $goto(state(y), a) = state(ya)$, where ya is a prefix of some keyword and a is a symbol in Σ. For any state q we denote by $string(q)$ the unique string y with $state(y) = q$. Thus, $string(q)$ is the string upon which state q is reached from the initial state via the goto function. We denote by $depth(q)$ the length of $string(q)$.

The *fail function* of the PMA is defined by the equation $fail(state(uv)) = state(v)$, 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. The fail function is organized as an array indexed by state numbers.

We modify the standard PMA so that, upon recognition of a keyword, the PMA is able to report exactly which of the filters match. Each keyword w is given a unique identifier, $id(w)$, and the *output function* of the PMA maps each accepting state q to $id(string(q))$. The output function is constructed in conjunction with calculation of the goto function, but it is not completed while constructing the fail function, as with the original algorithm by Aho and Corasick [5]. This means that $output(q)$ contains only the integer $id(string(q))$ and not the identifiers of the keywords found in the fail path from q. The *fail path* of state q includes those states that are found by traversing the fail arcs from q to the initial state. In our

construction the output function is of linear size instead of quadratic size, which is the case with the traditional Aho–Corasick PMA. Algorithms 3.1 and 3.2 present the pseudo code for the construction of the *goto* and *fail* functions of the automaton.

We use the function *output-fail*(q) to traverse the *output path* for state q. The function is defined by: *output-fail*(q) = $fail^k(q)$, where k is the greatest integer less than or equal to the length of *string*(q) such that *string*$(fail^m(q))$ 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 for which *output*(q) is not empty.

procedure *build-goto*(W):

```
1  newstate ← 0
2  for each keyword w in W  do
3     enter(w)
4  end for
5  for each a in the alphabet such that goto(0, a) is undefined  do
6     goto(0, a) ← 0
7  end for
```

procedure *enter*$(a_1 a_2 \ldots a_n)$:

```
1  state ← 0
2  j ← 1
3  while goto(state, a_j) is defined do
4     state ← goto(state, a_j)
5     j ← j + 1
6  end while
7  for p ← j to n  do
8     newstate ← newstate + 1
9     goto(state, a_p) ← newstate
10    state ← newstate
11 end for
12 output(state) ← id({a_1 a_2 ... a_n})
```

Algorithm 3.1. Construction of the *goto* function for the set of keywords W [5].

In our setting, the alphabet Σ contains the set of elements occurring in the DTD plus an additional symbol # denoting the beginning of any XML document. The set W of keywords is derived from the XPath filters

procedure *build-fail*():

```
1  queue ← empty
2  for each a such that goto(0, a) = s ≠ 0 do
3     queue.enqueue(s)
4     fail(s) ← 0
5  end for
6  while queue ≠ empty do
7     r ← queue.dequeue()
8     for each a such that goto(r, a) = s do
9        queue.enqueue(s)
10       state ← fail(r)
11       while goto(state, a) is not defined do
12          state ← fail(state)
13       end while
14       fail(s) ← goto(state, a)
15    end for
16 end while
```

Algorithm 3.2. Construction of the *fail* function differs from the original algorithm of Aho–Corasick [5] in that the output function is not completed.

provided by the subscribers. We decompose each filter into keywords as follows. First, we remove all child operators "/" from the filter. Then we define the *keyword* of the filter to be the string consisting of XML elements only. Here XML element names are represented as unique lexical symbols in our alphabet Σ, rather than as strings of characters. If the filter begins with an "/", then the symbol # is added in front of the keyword. Now a linear XPath filter containing no wildcards "$*$" nor non-leading descendant operators "//" gives rise to a single keyword w in Σ^*.

For example, the filters $Q_1 = /a/b/f$, $Q_2 = //b/f$, $Q_3 = /a/c/f$, and $Q_4 = /a/d/e/f$ are represented as keywords $w_1 = \#abf$, $w_2 = bf$, $w_3 = \#acf$, and $w_4 = \#adef$. The Aho–Corasick PMA is constructed for the set of all keywords thus obtained. Figure 3.1 shows the PMA for matching the example keywords.

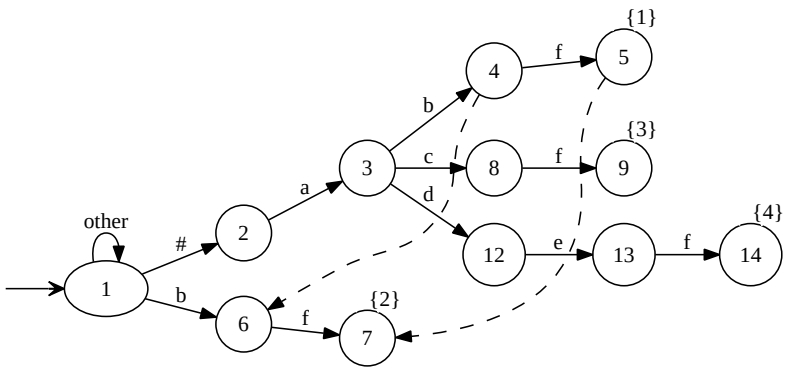


Figure 3.1. The Aho–Corasick automaton for matching keywords $w_1 = \#abf, w_2 = bf, w_3 = \#acf$ and $w_4 = \#adef$. The dashed lines denote fail arcs. There is also a fail arc to the initial state 1 from states 2, 3, 6, 7, 8, 9, 12, 13 and 14. The output function maps states 5, 7, 9, and 14 to the identifiers of the keywords recognized at those states.

To turn a PMA that recognizes linear text to a PMA that filters tree-structured text such as XML, we must do the following. A backtracking facility must be added so that the state of the automaton can be restored after scanning a subtree of the input document. The state is restored into the state that was active just before processing the element start-tag of the subtree started. This makes it possible to continue the matching process with the next sibling sub-element.

The input stream for the PMA consists of tokens produced by a SAX parser. When the SAX parser encounters an element start-tag, the current state of the automaton is pushed onto a stack, and the symbol correspond-

ing to the element name is consumed by the automaton. The automaton changes its state according to the PMA's *goto* and *fail* functions, and keeps track of the matching filters. On each element end-tag the automaton is backtracked: the state on top of stack is set as the current state and the stack is popped. When the input document has been processed, the algorithm reports the filters that match the input document.

The algorithm uses the following data structures, where *#keywords* is the number of distinct keywords, *#filters* is the number of filters and *#states* is the number of states.

- *goto*$[1 \ldots \#states]$: an array representing the *goto* function of the Aho–Corasick PMA. For state q, the entry *goto*$[q]$ is a hash table of pairs (a, q') indexed by input symbols a.
- *fail*$[1 \ldots \#states]$: an array representing the *fail* function of the PMA.
- *output*$[1 \ldots \#states]$: an array representing the *output* function of the PMA.
- *filters*$[1 \ldots \#keywords]$ is an array where *filters*$[id(w)]$ contains the numbers of those filters that contain the given keyword w.
- *result*$[1 \ldots \#filters]$ is a boolean array where *result*$[i]$ is *true* if i'th filter matches the input document.
- *output-visited*$[1 \ldots \#states]$ is a boolean array where *output-visited*$[s]$ is *true* if state s has been visited during the processing of the input document. When state s has been visited, we set *output-visited*$[s]$ to *true* and do not scan the same output path again for the input document.
- *symbol-table*$[1 \ldots |\Sigma|]$ is a hash table indexed by XML element names. The table contains the unique input symbol in Σ for the given XML element name.

The operating cycle of the algorithm is presented in Algorithm 3.3. The input for the procedure are the tokens produced by the SAX parser. When the input document has been processed, the filtering result can be read from the boolean array *result*. The procedure *print-result* prints the result to the user.

Figure 3.2 presents an example of processing an XML input document `<a><b><f></f></b><c><f></f></c></a>` with the PMA of Figure 3.1.

```
procedure operating-cycle():
 1  scan-next(token)
 2  while token was found do
 3     if token is a document start-tag then
 4        initialize()
 5        stack.push(state)
 6        sym ← #
 7        state ← goto(state, sym)
 8     else if token is a document end-tag then
 9        print-result()
10     else if token is a start-tag of element E then
11        stack.push(state)
12        sym ← symbol-table[E]
13        while goto(state, sym) = fail do
14           state ← fail(state)
15        end while
16        state ← goto(state, sym)
17        report-output(state)
18     else if token is an element end-tag then
19        state ← stack.pop()
20     end if
21     scan-next(token)
22  end while
```

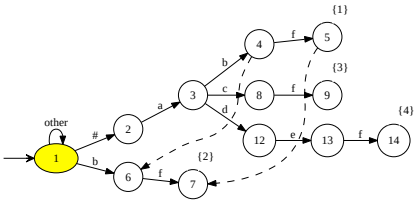
Algorithm 3.3. Operating cycle of the backtracking PMA.

```
procedure initialize():
1  state ← initial_state
2  for i = 1 to #states do
3     output-visited[i] ← false
4  end for
5  for i = 1 to #filters do
6     result[i] ← false
7  end for
```

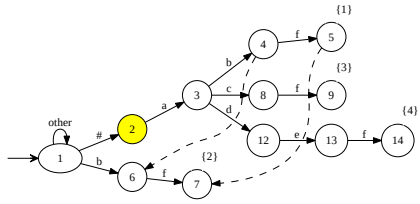
Algorithm 3.4. Procedure *initialize*.

```
procedure report-output(state):
 1  q ← state
 2  traversed ← false
 3  while not traversed do
 4     if output-visited[q] = false then
 5        id ← output[q]
 6        for each i in filters[id] do
 7           result[i] ← true
 8        end for
 9        output-visited[q] ← true
10     else
11        traversed ← true
12     end if
13     if q = initial-state then
14        traversed ← true
15     else
16        q ← output-fail(q)
17     end if
18  end while
```

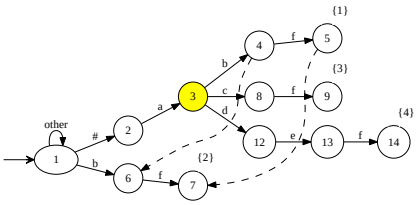
Algorithm 3.5. Procedure *report-output*(*state*).



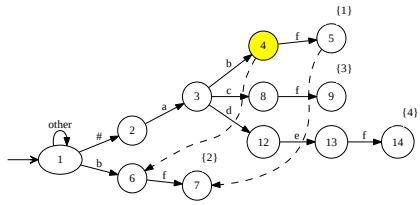
(a) Processing of the XML document starts from the initial state.



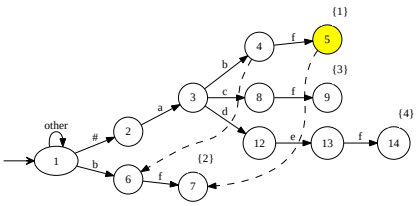
(b) Processed start of document. $S = \langle 1 \rangle$.



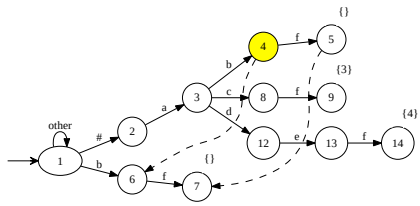
(c) Processed start of element `a`. $S = \langle 1, 2 \rangle$.



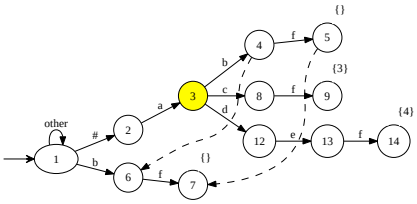
(d) Processed start of element `b`. $S = \langle 1, 2, 3 \rangle$.



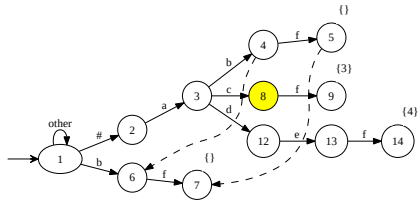
(e) Processed start of element `f`. $S = \langle 1, 2, 3, 4 \rangle$.



(f) Processed end of element `f`. Backtracking to state 4. $S = \langle 1, 2, 3 \rangle$.



(g) Processed end of element `b`. Backtracking to state 3. $S = \langle 1, 2 \rangle$.



(h) Processed start of element `c`. $S = \langle 1, 2, 3 \rangle$.

Figure 3.2. Processing XML document `<a><b><f></f></b><c><f></f></c></a>` with the backtracking PMA. S denotes the contents of the stack.

The processing starts from the initial state (Figure 3.2(a)). As we read the beginning of the XML document, the PMA enters state 2, and the previous state is pushed onto stack (Figure 3.2(b)). As we read the start-tag of element `a`, the PMA enters state 3, and the previous state is pushed onto stack (Figure 3.2(c)); similarly for elements `b` (Figure 3.2(d)) and `c` (Figure 3.2(e)). At state 5 we notice that filters Q_1 and Q_2 have matched. The numbers of matched filters are collected by traversing the output fail arc starting from state 5 down to state 7. This is called the output-path traversal. Next the output sets of states 5 and 7 can be cleared (by setting the *output-visited* flag for these states to true), because the filters have matched. As we process the end-tag of element `f`, the automaton returns to the state that was on top of the stack, and the stack is popped (Figures 3.2(f) and 3.2(g)). As we process the start-tag of element `c`, the PMA enters state 8. On the next `f` element filter Q_3 matches, concluding the example.

3.1.2 Complexity Analysis

Processing an element start-tag requires a lookup from the *goto*$[q]$ hash table, which takes $O(1)$ time. Here we assume the size of the alphabet Σ (or the number of distinct XML elements in the filters) is a constant. The output path of each state is traversed at most once, as can been seen in the while-loop of procedure *report-output* (Algorithm 3.5), where the *output-visited* flag is set for each visited state. Processing an element end-tag is also an $O(1)$ operation. Reporting each matched filter in the *report-output* procedure adds the term k to the total complexity result, where k is the number of matched filters. Preprocessing that includes the construction of the Aho–Corasick PMA takes time linear in the total size of the filters. The time bound of filtering is stated by the following theorem.

Theorem 3.1. The time complexity of filtering with XPath filters that do not contain wildcards or non-leading descendant operators is

$$O(|x| + k),$$

where $|x|$ is the size of the input x, that is the number of XML elements in the input document, and k is the number of matched filters.

3.2 Filtering with Wildcards and Descendant Operators

In this section we present an extension of the Aho–Corasick based [5] filtering algorithm of the previous section. This *static PMA* algorithm can process linear XPath filters having wildcards and descendant operators. The supported XPath fragment is described by the following grammar:

```
P := /E | //E | PP
E := label | *
```

where `label` denotes an XML-element label. The static PMA has been published in our recent journal article [66].

For solving the filtering problem with wildcards and descendant operators we construct the PMA as defined in the previous section from the set of all keywords that appear in the filters. This idea was previously used to solve the single-pattern matching problem with wildcards [55] (see also articles by Rahman et al. [59] and Rahman and Iliopoulos [58]). In single-pattern matching we collect all occurrences of a single pattern in the input text. However, the algorithms presented in these articles cannot be directly extended to yield an efficient solution to the case when there are multiple patterns (or filters). Also the algorithms are designed for processing only linear text; tree-structured input is not considered.

3.2.1 The Algorithm

As a preprocessing task, we construct an Aho–Corasick PMA from the filters as follows. Again the alphabet Σ contains the set of elements occurring in the DTD. Now we decompose each filter into keywords and gaps as follows. First, we remove all child operators "/" from the filter. Then we define the *keywords* of the filter to be maximal substrings consisting of XML elements only. The *gaps* of the filters are defined to be maximal substrings consisting of descendant operators "//" and wildcards "$*$". If the filter ends at a nonempty gap, then we assume that the last keyword of the filter is the empty string ϵ. Each filter is considered to begin with a gap, which thus may be ϵ.

We number the filters and their gaps and keywords consecutively, so that the ith filter Q_i can be represented as

$$Q_i = gap(i,1)keyword(i,1)\ldots gap(i,m_i)keyword(i,m_i),$$

where $gap(i,j)$ denotes the jth gap and $keyword(i,j)$ denotes the jth

keyword of filter Q_i. For example, the filter $//a/b/*/c//*/d/*/*$ consists of four gaps, namely $//$, $*$, $//*$, and $**$, and of four keywords, namely ab, c, d, and ϵ. The filter $/a/b/*/c//*/d$ consists of three gaps, namely ϵ, $*$, and $//*$, and three keywords, namely ab, c, and d.

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

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

$$\begin{aligned}
&mingap(i,1) = 0,\ maxgap(i,1) = \infty,\ length(i,1) = 2,\\
&mingap(i,2) = 1,\ maxgap(i,2) = 1,\ length(i,2) = 1,\\
&mingap(i,3) = 1,\ maxgap(i,3) = \infty,\ length(i,3) = 1,\\
&mingap(i,4) = 2,\ maxgap(i,4) = 2,\ length(i,4) = 0.
\end{aligned}$$

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

$$\begin{aligned}
&mingap(i,1) = 0,\ maxgap(i,1) = 0,\ length(i,1) = 2,\\
&mingap(i,2) = 1,\ maxgap(i,2) = 1,\ length(i,2) = 1,\\
&mingap(i,3) = 1,\ maxgap(i,3) = \infty,\ length(i,3) = 1.
\end{aligned}$$

In the case of XPath filters, we have, for all i and j, either $mingap(i,j) = maxgap(i,j)$ or $maxgap(i,j) = \infty$, because in any XPath expression the number of wildcards is fixed.

For the set of all keywords in the filters, we construct a backtracking Aho–Corasick PMA with output sets containing *output tuples* of the form (i,j). A tuple (i,j) is attached to state q, where $q = state(keyword(i,j))$, the state reached from the initial state upon reading the jth keyword of filter Q_i. The output sets are organized as an array $output[1 \ldots \#states]$, where entry $output[q]$ contains a doubly linked list of output tuples.

The basic idea of the algorithm is to collect, for each filter Q_i, *partial matches* of Q_i that represent matches of maximal prefixes of Q_i found thus far. When a match up to and including the last keyword of Q_i has been found, we have a full match of the pattern. Partial matches of keyword j of filter i are recorded in set $partial\text{-}matches(i,j)$ containing values p, where p is the current path length in the input document that matches the filter prefix $Q_{i,j}$. Here

$$Q_{i,j} = gap(i,1)keyword(i,1) \ldots gap(i,j)keyword(i,j).$$

Each set $partial\text{-}matches(i,j)$ is organized as a balanced binary search tree (red-black tree) indexed by p.

Algorithm 3.6 gives the operating cycle of the program. When visiting state q, its output set is checked for possible matches of keywords in the procedure call *traverse-output-path*(q) (see Algorithm 3.8). An output tuple (i, j) represents a match of the jth keyword of filter Q_i. The algorithm checks if a partial match (a match of a filter prefix) is found and stores the possible match into the set *partial-matches*(i, j). Now if the jth keyword is the last one in filter Q_i, then this indicates a match of the entire filter Q_i.

When the filter Q_i has matched, we may delete all output tuples for i (line 6 in Algorithm 3.8). This can be done efficiently, since the deletion of an output tuple from a doubly-linked list is an $O(1)$ operation. Deleted output tuples are stored into list *deleted-output*. This list is used to restore the output sets when the PMA is initialized for a new incoming XML document in the procedure *initialize*.

Because keywords of length zero may have to be recognized, the operating cycle of the backtracking PMA also contains the call *traverse-output-path*(*state*) for the initial state, and the procedure *traverse-output-path* has been structured so that it also observes possible output tuples for the initial state. The global variable *path-length* is incremented whenever an element start-tag is scanned, and decremented whenever an element end-tag is scanned. The variables *document-count* and *element-count* are used in the algorithm of Section 3.4.

The *backtracking stack* now contains information about states visited and partial matches inserted into the *partial-matches* structure during traversing a root-to-leaf path in the current input document. The PMA backtracks when an element end-tag is scanned; then elements from the stack are popped, insertions of partial matches are undone, and the control of the PMA is returned to the state that was entered when scanning the previous element start-tag (see the procedure *backtrack* given as Algorithm 3.10).

Figure 3.3 shows how the XML document `<a><b><c></c></b></a>` is processed with the PMA. The PMA has been constructed from filters $Q_1 = //a/b//c$ and $Q_2 = //b/*/*/c$. In this case there are three keywords: ab, b, and c, leading to states 3, 4, and 5. After the initialization of the PMA, the output set of state 3 contains the tuple $(1, 1)$ representing keyword ab of filter Q_1, the output set of state 4 contains the tuple $(2, 1)$ representing the first keyword of filter Q_2, and the output set of state 5 the tuples $(1, 2)$ and $(2, 2)$ representing the second keyword of filter Q_1 and the second keyword of filter Q_2.

The processing of the XML document starts from the initial state (Figure 3.3(a)). After processing the start-tags of elements `a` and `b`, the

```
procedure operating-cycle():
 1  document-count ← 0
 2  deleted-output ← ∅
 3  scan-next(token)
 4  while token was found do
 5    if token is a document start-tag then
 6      document-count ← document-count + 1
 7      element-count ← 0
 8      path-length ← 0
 9      initialize()
10      state ← initial-state
11      push-onto-stack(state)
12      traverse-output-path(state)
13    else if token is a document end-tag then
14      print-result()
15    else if token is a start-tag of element E then
16      element-count ← element-count + 1
17      path-length ← path-length + 1
18      push-onto-stack(state)
19      sym ← symbol-table[E]
20      while goto(state, sym) = fail do
21        state ← fail(state)
22      end while
23      state ← goto(state, sym)
24      traverse-output-path(state)
25    else if token is an element end-tag then
26      backtrack()
27      path-length ← path-length − 1
28    end if
29    scan-next(token)
30  end while
```

Algorithm 3.6. Operating cycle of the filtering PMA that can handle filters with wildcards and descendant operators.

procedure *initialize*():
1 **for** all keywords (i, j) in filters Q **do**
2 $\quad$ *partial-matches*$(i, j) \leftarrow \varnothing$
3 **end for**
4 **for** all filters Q_i **do**
5 $\quad$ *result*$[i] \leftarrow$ *false*
6 **end for**
7 **for** all $(i, j) \in$ *deleted-output* **do**
8 $\quad$ insert (i, j) into *output*(*state*(*keyword*$((i, j))$))
9 **end for**
10 *deleted-output* $\leftarrow \varnothing$

Algorithm 3.7. Procedure *initialize*().

PMA has entered state 3 and stored visited states onto the stack. In state 3, keywords *ab* and *b* have matched (Figure 3.3(c)). Current path length 2 is stored into sets *partial-matches*$(1, 1)$ and *partial-matches*$(2, 1)$ (Figure 3.3(d)). Modifications to *partial-matches* are also stored onto the stack. Here $i(i, j, p)$ denotes the insertion of a value p into *partial-matches*(i, j).

The processing of the start-tag of element `c` leads to state 5 (Figure 3.3(e)). Now filter Q_1 has matched, since *partial-matches*$(1, 1) \leq$ *path-length* – *length*$(1, 2)$ – *mingap*$(1, 2)$. When we process the end-tag of element `c`, the automaton enters state 3 and the stack is popped (Figure 3.3(f)). Processing the end-tag of element *b* causes the undoing of operations stored into the stack before the next state change operation. As we process the end-tag of element `a`, the automaton enters state 1 (Figure 3.3(h)), concluding this example.

3.2.2 Complexity Analysis

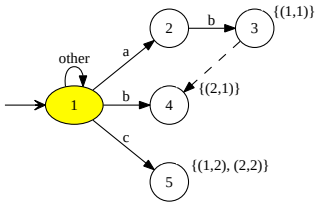
Each filter Q_i is of the form

$$Q_i = gap(i, 1)keyword(i, 1) \ldots gap(i, m_i)keyword(i, m_i).$$

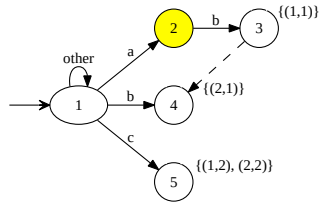
Whenever a keyword *keyword*(i, j) has been recognized at some element position in the algorithm, it can trigger a lookup and a possible insertion into the set *partial-matches*(i, j) (in Algorithm 3.8), which is a balanced binary search tree. The maximum size of the tree (the number of keys) is the depth of the input document (the largest value that the *path-length* variable can have). When we denote this by L, we get a time bound

```
procedure traverse-output-path(state):
 1  q ← state
 2  traversed ← false
 3  while not traversed do
 4    for all (i, j) ∈ output(q) do
 5      if result[i] = true then
 6        delete (i, j) from output(q)
 7        insert (i, j) into deleted-output
 8      else
 9        p ← path-length − length(i, j) − mingap(i, j)
10        if j = 1 then
11          if maxgap(i, j) = ∞ and p ≥ 0 then
12            advance-match(i, j)
13          else if p = 0 then
14            advance-match(i, j)
15          end if
16        else
17          if maxgap(i, j) = ∞ and partial-matches(i, j − 1) contains
            some p′ ≤ p then
18            advance-match(i, j)
19          else if maxgap(i, j) < ∞ and partial-matches(i, j − 1) con-
            tains p then
20            advance-match(i, j)
21          end if
22        end if
23      end if
24    end for
25    if q = initial-state then
26      traversed ← true
27    else
28      q ← output-fail(q)
29    end if
30  end while
```

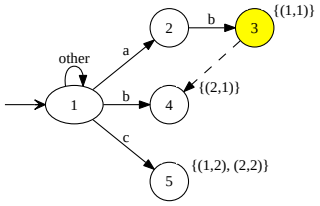
Algorithm 3.8. Procedure *traverse-output-path*(*state*).



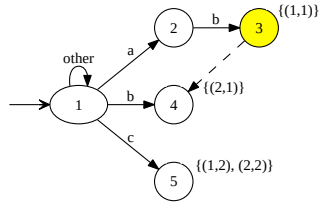
(a) Processing of the XML document starts from the initial state.



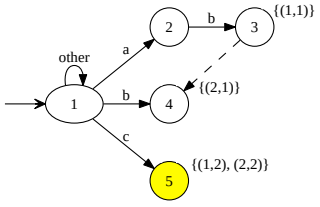
(b) Processed start of element a. $S = \langle 1 \rangle$.



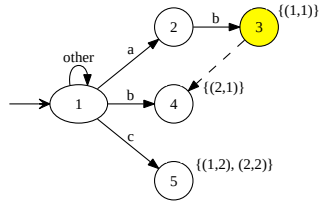
(c) Processed start of element `b`. $S = \langle 1, 2 \rangle$.



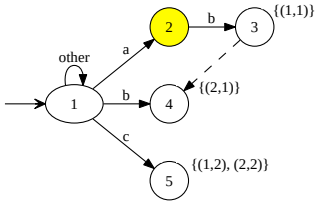
(d) Updated *partial-matches*. $S = \langle 1, 2, i(1,1,2), i(2,1,2) \rangle$. $P(1,1) = P(2,1) = \{2\}$.



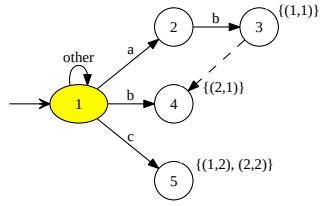
(e) Processed start of element `c`. Filter Q_1 matched. $S = \langle 1, 2, i(1,1,2), i(2,1,2), 3 \rangle$. $P(1,1) = P(2,1) = \{2\}$.



(f) Processed end of element `c`. Backtracked to state 3. $S = \langle 1, 2, i(1,1,2), i(2,1,2) \rangle$. $P(1,1) = P(2,1) = \{2\}$.



(g) Processed end of element `b`. Backtracked to state 2. $S = \langle 1 \rangle$.



(h) Processed end of element `a`. Backtracked to state 1. $S = \langle \rangle$.

Figure 3.3. Processing XML document `<a><b><c></c></b></a>` with the backtracking PMA. The filter workload consists of two filters: $Q_1 = //a/b//c$ and $Q_2 = //b/*/*/c$. As before, S denotes the contents of the stack. $P(i,j)$ denotes the contents of the set *partial-matches*(i,j).

procedure *advance-match*(i, j):

```
1  if j = keywords(i) then
2     result[i] ← true
3  else
4     insert path-length into partial-matches(i, j)
5     push-onto-stack(inserted⟨i, j, path-length⟩)
6  end if
```

Algorithm 3.9. Procedure *advance-match*(i, j).

procedure *backtrack*():

```
1  pop the topmost element s from the stack
2  while s is not a state do
3     if s is inserted⟨i, j, p, ⟩ then
4        delete p from partial-matches(i, j)
5     end if
6     pop the topmost element s from the stack
7  end while
8  state ← s
```

Algorithm 3.10. Procedure *backtrack*().

$O(\log L)$ for a lookup from or an insertion into the search tree. For a set of filters and an input document x, denote by *occ*(*keywords*) the number of occurrences in x of all keyword instances in the filter workload. This adds the term $O(\log L \times occ(keywords))$ to the time complexity of the filtering algorithm.

Processing the input document requires additionally at most $O(K \times |x|)$ time, where $|x|$ is the length of the input document (the number of XML elements in the document) and K denotes the maximum number of proper suffixes of one keyword that are also keywords. The multiplier K is due to the fact that all states on the output path must be traversed (by using the function *output-fail*) in order to check all possibilities to continue the currently matched filter prefix. This gives the time bound $O(K \times |x| + \log L \times occ(keywords))$ for filtering. However, because $occ(keywords) \geq K \times |x|$, the multiplier K can be removed. The term $|x|$ is needed, because in some cases there might be no keyword occurrences.

Now we can state the time complexity of the filtering algorithm with the following theorem.

Theorem 3.2. The static PMA filtering algorithm that can process linear XPath filters having wildcards and descendant operators runs in

time

$$O(|x| + \log L \times occ(keywords)),$$

where $|x|$ denotes the number of XML elements in the input document x, L is the depth of x, and $occ(keywords)$ denotes the number of occurrences in x of all keyword instances in the filter workload.

3.3 Using a Dynamic Output Function

In this section we present an enhanced version of the PMA-based filtering algorithm of the previous section. This dynamic PMA algorithm has dynamically changing output sets; the output function is dynamically updated during processing of the input document. With this algorithm we recognize only those keyword occurrences that have a matching filter prefix ending with this keyword occurrence. The dynamic PMA has been published in our recent conference article [67]. With practical data sets the dynamic PMA is more efficient than the static PMA.

3.3.1 The Algorithm

For the set of all keywords in the filters, we construct a backtracking Aho–Corasick pattern-matching automaton with a dynamically changing output set *current-output* containing tuples of the form

$$(q, i, j, b, e),$$

where $q = state(keyword(i, j))$, the state reached from the initial state upon reading the jth keyword of filter Q_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 filter Q_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 filters, that is, tuples $(q, i, 1, b, e)$, where q is the state reached from the initial state upon reading the first keyword of filter Q_i,

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

(see the procedure *initialize* given as Algorithm 3.11). Here $e = \infty$ if $maxgap(i, 1) = \infty$.

In the case of XPath patterns, we have, for all i and j, either $mingap(i,j) = maxgap(i,j)$ or $maxgap(i,j) = \infty$, 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 $mingap(i,j) < maxgap(i,j) < \infty$.

The *backtracking stack* now contains information about states visited and output tuples inserted into and deleted from the current output when traversing a root-to-leaf path in the current input document (see the procedure *backtrack* given as Algorithm 3.13).

The operating cycle of the algorithm is the same as with the previous algorithm (see Algorithm 3.6). When visiting state q, the current output of the PMA is checked for possible matches of keywords in the procedure call *traverse-output-path*(q) (see Algorithm 3.12). A current output tuple (q,i,j,b,e) is found to represent a match of the jth keyword of filter Q_i if $b \le$ *path-length* $\le e$, 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 filter Q_i, then this indicates a match of the entire filter Q_i. Otherwise, an output tuple $(q',i,j+1,b',e')$ for the $(j+1)$st keyword of filter Q_i is inserted into the set *current-output*, where q' is the state reached from the initial state upon reading the $(j+1)$st keyword of filter Q_i,

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

Here $e' = \infty$ if $maxgap(i,j+1) = \infty$. If $e' = \infty$, we could delete from *current-output* all output tuples (q'',i,j'',b'',e'') with $j'' \le j$. In this case we are no longer waiting keywords of filter Q_i with $j'' \le j$ to be recognized and the output tuples for these keywords can be removed from the current output. However, since tuples (q'',i,j'',b'',e'') with $e'' <$ *path-length* are deleted later by the procedure *traverse-output-path* (lines 7–9 in Algorithm 3.12), we only delete here tuples (q'',i,j'',b'',∞) with $j'' \le j$ (which can be done efficiently, see below).

The set of current output tuples is organized as an array *current-output*$[1 \ldots \#states]$ containing balanced binary search trees (red-black trees). Each such tree is indexed by b and a node contains a pointer to a doubly-linked list of current output tuples (q,i,j,b,e). In the procedure *traverse-output-path*, when visiting state q, the search tree *current-output*$[q]$ is used to locate the output tuples (q,i,j,b,e) with $b \le$ *path-length*. Figure 3.4 exemplifies the structure.

Current output tuples having $e = \infty$ are also stored into an array *current-output2*$[1 \ldots \#filters]$, where entry *current-output2*$[i]$ holds a poin-

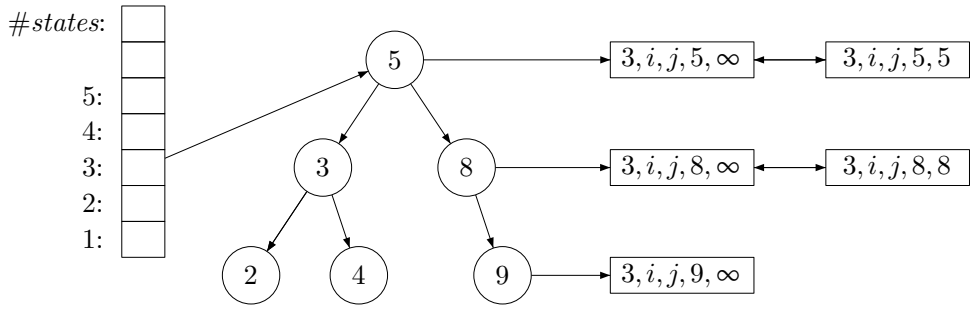


Figure 3.4. The set of current output tuples is organized as an array indexed by states and containing balanced binary search trees. Each such tree is indexed by b and a node contains a pointer to a doubly-linked list of current output tuples (q, i, j, b, e).

ter to a linked list of current output tuples (q, i, j, b, ∞). Array *current-output2* is used to locate tuples $(q'', i, j'', b'', \infty)$ for deletion in procedure *traverse-output-path* (lines 20–23 in Algorithm 3.12). Triple (i, j, b) defines a unique key for each output tuple and we also use a three-dimensional array *current-output3*$[i, j, b]$ for storing the current output tuples. This array is used to prevent the creation of duplicate tuples when backtracking.

When the filter Q_i has matched, we may delete all output tuples for i permanently, that is, without recording the deletions onto the stack. This will be also done in the *traverse-output-path* procedure (lines 5–6 in Algorithm 3.12). When the doubly-linked list pointed by key b in the search tree *current-output*$[q]$ becomes empty (because of deletions of tuples), then b can be removed from the tree.

Figure 3.5 shows how the example document `<a><b><c></c></b></a>` is processed with the dynamic PMA. Again, the PMA has been constructed from filters Q_1 = //*a*/*b*//*c* and Q_2 = //*b*/∗/∗/*c*. After the initialization of the PMA, the output set of state 3 contains the tuple $(3, 1, 1, 2, \infty)$ representing keyword *ab* of filter Q_1 and the output set of state 4 the tuple $(4, 2, 1, 1, \infty)$ representing the first keyword of filter Q_2.

The processing of the XML document starts from the initial state (Figure 3.5(a)). After processing the start-tags of elements `a` and `b`, the PMA has entered state 3 and stored visited states onto the stack (Figure 3.5(c)). In state 3, keywords *ab* and *b* have matched. The output tuple $(3, 1, 1, 2, \infty)$ is removed from the output set of state 3 and the out-

procedure *initialize*():

```
1 current-output ← ∅
2 for all filters Q_i do
3    result[i] ← false
4    q ← state(keyword(i, 1))
5    b ← mingap(i, 1) + length(i, 1)
6    e ← maxgap(i, 1) + length(i, 1)
7    insert (q, i, 1, b, e) into current-output
8    push-onto-stack(inserted(q, i, 1, b, e))
9 end for
```

Algorithm 3.11. Procedure *initialize*().

put tuple $(5, 1, 2, 3, \infty)$ of the next keyword of filter Q_1 is inserted into the output set of state 5. The output tuple $(5, 2, 2, 4, 4)$ of the next keyword of filter Q_2 is inserted into the output set of state 5 (Figure 3.5(d)). Modifications to the current output are stored onto the stack. Here $i(q, i, j, b, e)$ denotes the insertion of an output tuple, and $d(q, i, j, b, e)$ the deletion of a tuple.

The processing of the start-tag of element `c` leads to state 5. Now filter Q_1 has matched, since $b \leq$ *path-length* (b = *path-length* = 3). The output tuple $(5, 1, 2, 3, \infty)$ can be removed from the current output. This modification is not recorded. When we process the end-tag of element `c`, the automaton enters state 3 and the stack is popped (Figure 3.5(g)). Processing the end-tag of element `b` causes the undoing of operations stored into the stack before the next state change operation. However, the operations regarding the matched filter Q_1 do not need to be reversed. The automaton enters state 2 (Figure 3.5(h)), concluding this example.

3.3.2 Complexity Analysis

Each filter Q_i is of the form

$$Q_i = gap(i, 1)keyword(i, 1) \ldots gap(i, m_i)keyword(i, m_i),$$

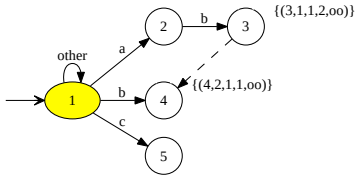
and whenever a prefix $Q_{i,j}$ of Q_i,

$$Q_{i,j} = gap(i, 1)keyword(i, 1) \ldots gap(i, j)keyword(i, j),$$

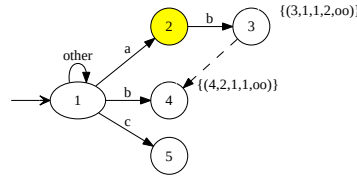
has been recognized at some element position in the algorithm, a new output tuple $(q', i, j + 1, b', e')$ will be inserted into the current output in Algorithm 3.12. The recognition of prefix $Q_{i,j}$ may also trigger a deletion

```
procedure traverse-output-path(state):
 1  q ← state
 2  traversed ← false
 3  while not traversed do
 4    for all (q, i, j, b, e) ∈ current-output with b ≤ path-length do
 5      if result[i] = true then
 6        delete (q, i, j, b, e) from current-output
 7      else if e < path_length then
 8        delete (q, i, j, b, e) from current-output
 9        push-onto-stack(deleted⟨q, i, j, b, e⟩)
10      else if j = #keywords(i) then
11        result[i] ← true
12        delete (q, i, j, b, e) from current-output
13      else
14        q′ ← state(keyword(i, j + 1))
15        b′ ← path-length + mingap(i, j + 1) + length(i, j + 1)
16        e′ ← path-length + maxgap(i, j + 1) + length(i, j + 1)
17        insert (q′, i, j + 1, b′, e′) into current-output
18        push-onto-stack(inserted⟨q′, i, j + 1, b′, e′⟩)
19        if e′ = ∞ then
20          for all (q″, i, j″, b″, ∞) ∈ current-output with j″ ≤ j do
21            delete (q″, i, j″, b″, ∞) from current-output
22            push-onto-stack(deleted⟨q″, i, j″, b″, ∞⟩)
23          end for
24        end if
25      end if
26    end for
27    if q = initial-state then
28      traversed ← true
29    else
30      q ← output-fail(q)
31    end if
32  end while
```

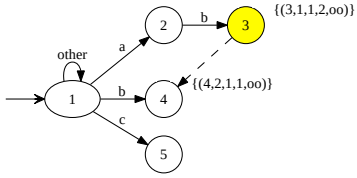
Algorithm 3.12. Procedure *traverse-output-path*(*state*).



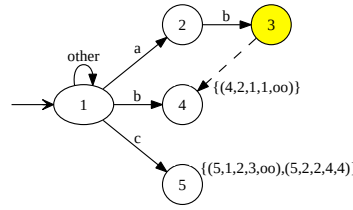
(a) Processing of the XML document starts from the initial state.



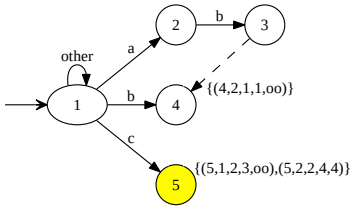
(b) Processed start of element `a`. $S = \langle 1 \rangle$.



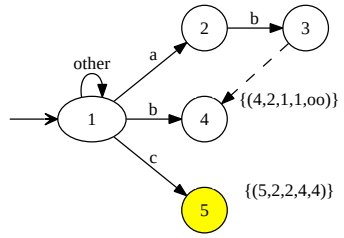
(c) Processed start of element `b`. $S = \langle 1, 2 \rangle$.



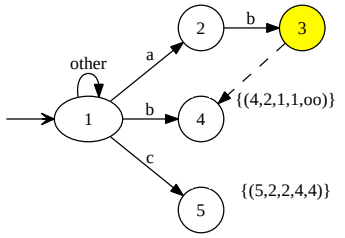
(d) Updated output sets of states 3 and 5. $S = \langle 1, 2, d(3,1,1,2,\infty), i(5,1,2,3,\infty), i(5,2,2,4,4) \rangle$.



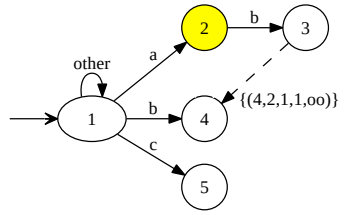
(e) Processed start of element `c`. Filter Q_1 matched. $S = \langle 1, 2, i(5,1,2,3,\infty), i(5,2,2,4,4), 3 \rangle$.



(f) Updated output set of state 5. $S = \langle 1, 2, d(3,1,1,2,\infty), i(5,1,2,3,\infty), i(5,2,2,4,4), 3 \rangle$.



(g) Processed end of element `c`. $S = \langle 1, 2, d(3,1,1,2,\infty), i(5,1,2,3,\infty), i(5,2,2,4,4) \rangle$.



(h) Processed end of element `b`. Updated output set of state 4. $S = \langle 1 \rangle$.

Figure 3.5. Processing XML document `<a><b><c></c></b></a>` with the dynamic PMA. The filter workload consists of filters: $Q_1 = //a/b//c$ and $Q_2 = //b/*/*/c$. As before, S denotes the contents of the stack.

procedure *backtrack*():

```
1  pop the topmost element s from the stack
2  while s is not a state do
3    if s is inserted⟨q, i, j, b, e⟩ then
4      delete (q, i, j, b, e) from current-output
5    else if s is deleted⟨q, i, j, b, e⟩ and result[i] = false then
6      insert (q, i, j, b, e) into current-output
7    end if
8    pop the topmost element s from the stack
9  end while
10 state ← s
```

Algorithm 3.13. Procedure *backtrack*().

of an output tuple.

An insertion into the current output requires a lookup and a possible insertion of a key into the balanced binary search tree. The maximum size of the tree (the number of keys) is the largest value that the b' component of the new output tuple can have. This is bounded by $L + M + N$, where L is the depth of the input document, that is, the maximum length of a path in the input document, M is the maximum number of consecutive wildcards in any filter in the filter workload Q, and N the maximum length of a keyword. For example, if the filter workload consists of a filter $//a//b/*/*/*/*/*/c/d$, then $M = 5$ and $N = 2$. Thus an insertion into the current output takes $O(\log(L+M+N))$ time. The deletion of a tuple from the doubly-linked lists is an $O(1)$ operation, but a deletion can also cause the deletion of a key from the search tree, which takes $O(\log(L+M+N))$ time.

For a set of filters and an input document x, denote by $occ(pref(Q))$ the number of occurrences in x of all filter prefixes ending with a whole keyword, that is, all prefixes of the form $Q_{i,j}$ above. An occurrence of such a filter prefix can cause the insertion of a tuple into the current output or the deletion of a tuple. This adds the term $O(\log(L+M+N) \times occ(pref(Q)))$ to the time complexity of the filtering algorithm.

With this algorithm processing the input document requires additionally at most $O(K \times |x|)$ time, where $|x|$ is the length of the input document (the number of XML elements in the document) and K denotes the maximum number of proper suffixes of one keyword that are also keywords. This is due to the fact that in some cases $occ(pref(Q)) \leq K \times |x|$, that is, the output path of some state can contain states with empty output sets. We can now state the time complexity as follows.

Theorem 3.3. The dynamic PMA filtering algorithm that can process linear XPath filters having wildcards and descendant operators runs in time

$$O(K \times |x| + \log(L + M + N) \times occ(pref(Q))),$$

where $|x|$ 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, L is the depth of the input document, M is the maximum number of consecutive wildcards in any filter in the filter workload Q, N is the maximum length of a keyword, and $occ(pref(Q))$ denotes the number of occurrences in x of all filter prefixes ending with a whole keyword.

When comparing this time bound with the time bound of the static algorithm (Theorem 3.2) the following remarks can be made. First, the number of all keyword occurrences $occ(keywords)$ is always greater than or equal to the number of occurrences of all filter prefixes $occ(pref(Q))$. This is due to the fact that for each filter prefix only at most one occurrence is recognized at each element position. For example, for the XML document `<c><a><b><c></c></b></a></c>` and filter $Q_1 = //a//c$, we can see that $occ(keywords) = 3$, $occ(pref(Q_1)) = 2$, and $occ(Q_1) = 1$.

Also practical XML documents are shallow; the depth of a highly complex XML data set is only 36 (see Section 4.1.3) and the depth of most data sets is much less. Usually M, the maximum number of wildcards, and N, the length of the keyword, are small. For this reason it can be assumed that the term $\log L \times occ(keywords)$ is greater than $\log(L + M + N) \times occ(pref(Q))$. With these arguments it can be expected that with practical XML data sets the dynamic algorithm of this section is more efficient than the algorithm of Section 3.2. This will be experimentally verified in the next chapter.

3.4 Optimization by Fast Backtracking

In practical XML documents the paths tend to be quite short, so that backtracking happens often. 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. In this section we present an organization of current output tuples that allows very efficient backtracking. We have outlined this *PMA FB* algorithm in our recent conference article [67].

3.4.1 The Algorithm

In the revised algorithm the current output 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 element start-tags from the input document. The stack may grow up to a height of *maxdepth* + 1 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 = *path-length*+1. 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 $O(\#states \times maxdepth)$ entries, so that the index of the entry for state q in block h is obtained as

$$k = (h - 1) \times \#states + q$$

(states q are numbered consecutively $1, 2, \ldots, \#states$). The 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 *path-length* + 1 = h, *document-count* = d, and *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 = *document-count* and v = *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* (see Algorithm 3.18).

When traversing an output path (in Algorithm 3.14) and finding out which output tuples for state q stored in block h match (in Algorithm 3.15), we first check whether or not d = *document-count* and v = *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 $b \leq$ *path-length* $\leq e$. Those output tuples are selected into

set S that satisfy this condition. We must collect output tuples from each block from 1 to *path-length*+1. Finally, the set of selected output tuples is returned for further processing. Figure 3.6 illustrates the data structure for the current output set for the fast backtracking optimization.

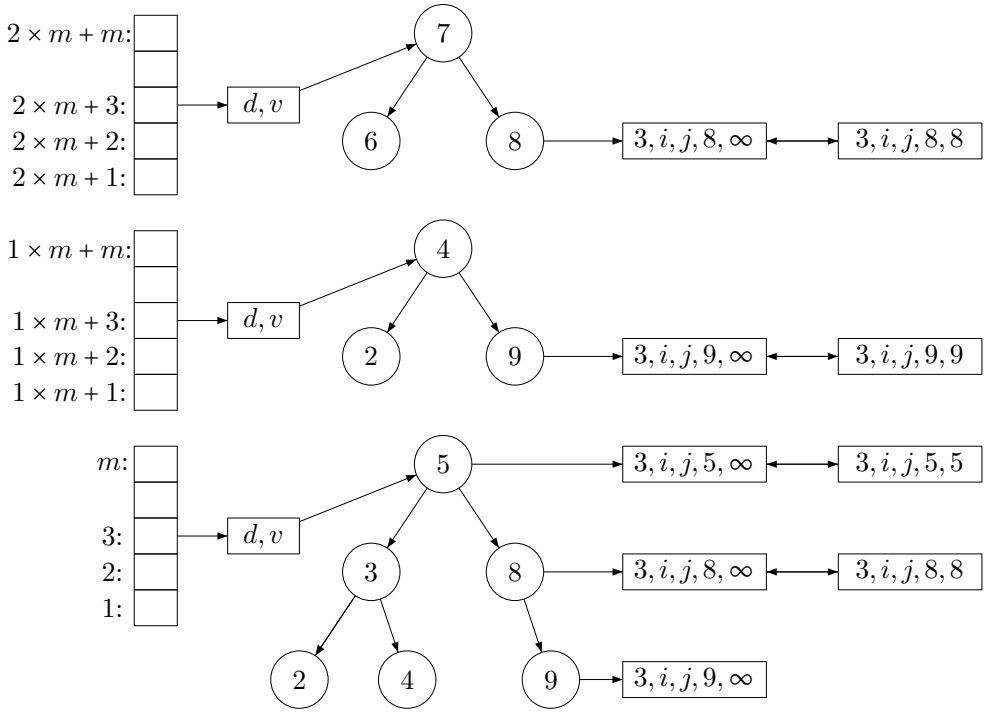


Figure 3.6. With the fast backtracking optimization the array holding balanced search trees is dispersed into blocks. The index of the entry for state q in block h is obtained as $k = (h - 1) \times m + q$, where m denotes the number of states.

The backtracking stack that in the algorithm of Section 3.3 contained, besides states pushed there when reading element start-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 v is the value of *element-count* that were current at the time the pair was pushed onto the stack. This stack is implemented as an array *stack* of size $O(maxdepth)$; the index of the topmost element of the stack is given by the variable *path-length* + 1 (see Algorithm 3.16).

Backtracking can now be done in a single step. All the output tuples contained in the search trees referenced in a block of the *current-output* array can be set outdated in a single operation. The set of output tuples

```
procedure traverse-output-path(state):
 1  q ← state
 2  traversed ← false
 3  for all (q, i, j, b, e) ∈ output(state, path-length) do
 4     if result[i] = true then
 5        delete (q, i, j, b, e) from current-output
 6     else if j = #keywords(i) then
 7        result[i] ← true
 8     else
 9        q′ ← state(keyword(i, j + 1))
10        b′ ← path-length + mingap(i, j + 1) + length(i, j + 1)
11        e′ ← path-length + maxgap(i, j + 1) + length(i, j + 1)
12        insert-output(q′, i, j + 1, b′, e′)
13     end if
14     if q = initial-state then
15        traversed ← true
16     else
17        q ← output-fail(q)
18     end if
19  end for
```

Algorithm 3.14. Procedure *traverse-output-path(state)*.

```
function output(state, path-length):
 1  S ← ∅
 2  for h = 1 to path-length + 1 do
 3     k ← (h − 1) × #states + state
 4     (t, d, v) ← current-output[k]
 5     (q, v′) ← stack[h]
 6     if d = document-count and v = v′ then
 7        for all (q, i, j, b, e) ∈ t with b ≤ path-length ≤ e do
 8           S ← S ∪ {(q, i, j, b, e)}
 9        end for
10     end if
11  end for
12  return S
```

Algorithm 3.15. Function *output*(*state*, *path-length*).

procedure *push-onto-stack*(*state*):
1 $stack[path\text{-}length + 1] \leftarrow (state, element\text{-}count)$

Algorithm 3.16. Procedure *push-onto-stack*(*state*).

procedure *initialize*():
1 **for** all filters Q_i **do**
2 $result[i] \leftarrow false$
3 $q \leftarrow state(keyword(i, 1))$
4 $b \leftarrow mingap(i, 1) + length(i, 1)$
5 $e \leftarrow maxgap(i, 1) + length(i, 1)$
6 $insert\text{-}output(q, i, 1, b, e)$
7 **end for**

Algorithm 3.17. Procedure *initialize*().

procedure *insert-output*(q, i, j, b, e):
1 $k \leftarrow path\text{-}length \times \#states + q$
2 $(t, d, v) \leftarrow current\text{-}output[k]$
3 **if** $d \neq document\text{-}count$ or $v \neq element\text{-}count$ **then**
4 $d \leftarrow document\text{-}count$
5 $v \leftarrow element\text{-}count$
6 initialize search tree t as empty
7 $current\text{-}output[k] \leftarrow (t, d, v)$
8 **end if**
9 insert (q, i, j, b, e) into tree t

Algorithm 3.18. Procedure *insert-output*(q, i, j, b, e).

procedure *backtrack*():
1 $(q, v) \leftarrow stack[path\text{-}length + 1]$
2 $state \leftarrow q$

Algorithm 3.19. Procedure *backtrack*().

contained in the topmost block become outdated when an element end-tag is encountered. This method of backtracking is more efficient than that used in the dynamic PMA that removes the outdated tuples from the doubly-linked lists one-by-one. With a workload of multiple filters there are usually many output tuples that become outdated when processing an element end-tag.

3.4.2 Complexity Analysis

The current output for state q is now dispersed in h blocks, where h is *path-length* + 1, the length of the current path plus one. 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 $K \times |x|$ in the complexity bound stated in Section 3.3.2 is changed to $L \times K \times |x|$. Here L is the depth of input document x, that is, the maximum length of a path in x. An insertion into the current output takes $O(\log(L + M + N))$ time as with the previous algorithm. The time bound for the algorithm is given in the following theorem.

Theorem 3.4. The dynamic filtering algorithm with the fast backtracking optimization runs in time

$$O(L \times K \times |x| + \log(L + M + N) \times occ(pref(Q))),$$

where $|x|$ denotes the number of XML elements in the input document x, K is the maximum number of proper suffixes of a keyword that are also keywords, L is the depth of the input document, M is the maximum number of consecutive wildcards in the filter workload Q, N is the maximum length of a keyword and $occ(pref(Q))$ denotes the number of occurrences in x of all filter prefixes ending with a whole keyword.

The time bound is worse than the one of the dynamic algorithm of the Section 3.3 (Theorem 3.3). It is clear that when the depth of the XML data increases, the performance of this algorithm decreases quickly. However, in practice the paths in the XML documents are short and L can be assumed to be a small constant. In the next chapter it will be experimentally verified that with real XML data sets this algorithm outperforms the previous algorithm.

3.5 Online Dictionary Matching of XML Documents

A problem related to XML filtering is the task of locating all occurrences of all XPath patterns. The problem is to determine, for each XML 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 can avoid this possible explosion of the number of occurrences by reporting only one occurrence in such situations.

All our PMA-based filtering algorithms (described in Sections 3.1–3.4) can easily be modified to solve this problem. In this section we present a modification to the algorithm of the previous section that locates all occurrences of all XPath patterns in the given XML document.

3.5.1 The Algorithm

To modify the algorithm of Section 3.4 so as to collect all occurrences of all XPath patterns only the procedure *traverse-output-path* needs to be altered. Now we do not delete the last output tuple of a filter (or a pattern) from the current output set once a match has been found, but leave the tuple in place and collect all the positions of each match. Algorithm 3.20 gives the new *traverse-output-path* procedure.

3.5.2 Complexity Analysis

The time bounds of the algorithms of Sections 3.2–3.4 are derived without taking into account the optimization that with these filtering algorithms the output tuples for a matched filter are removed from the output sets permanently. Thus the time bound of this algorithm is the same as presented in Section 3.4.2. Also with this algorithm, in the worst case, a lookup and possibly an insertion of a key into the search tree of size $L+M+K$ has to be performed on each occurrence of a filter prefix, where L is the document depth, M the maximum number of consecutive wildcards in the filter workload, and N the maximum length of a keyword. However, with practical XML data sets and filter workloads filtering is more efficient than online dictionary matching, because in the filtering case we collect only the first occurrence of each XPath filter.

When the bare AC algorithm (described in Section 3.1) is modified for collecting all occurrences of all XPath filters, its time bound will be $O(|x|+$

procedure *traverse-output-path*(*state*):

```
 1  q ← state
 2  traversed ← false
 3  for all (q, i, j, b, e) ∈ output(state, path-length) do
 4     if j = #keywords(i) then
 5        report a match of pattern Q_i at position element-count in docu-
           ment document-count
 6     else
 7        q' ← state(keyword(i, j + 1))
 8        b' ← path-length + mingap(i, j + 1) + length(i, j + 1)
 9        e' ← path-length + maxgap(i, j + 1) + length(i, j + 1)
10        insert-output(q', i, j + 1, b', e')
11     end if
12     if q = initial-state then
13        traversed ← true
14     else
15        q ← output-fail(q)
16     end if
17  end for
```

Algorithm 3.20. Procedure *traverse-output-path(state)* modified for reporting all occurrences of all XPath filters.

$occ(keywords)$), where $|x|$ is the size of the input x and $occ(keywords)$ denotes the number of occurrences in x of all keyword instances in the filter workload. In this case the *output-visited* flag is not used and the position of each keyword occurrence will be collected in the algorithm.

CHAPTER 4

Experiments on PMA-based Filtering

In this chapter we evaluate the filtering performance and memory usage of the PMA-based filtering algorithms of the previous chapter and those of YFilter [24] and the lazy DFA [28]. In the following the algorithm of Section 3.1 is denoted by "bare AC", the algorithm of Section 3.2 by "static PMA", the algorithm of Section 3.3 by "dynamic PMA", and the algorithm of Section 3.4 by "PMA FB".

We begin by describing the hardware and software environment, statistical tools, and characteristics of the XML data sets used in the experiments. This is followed by the results and analysis of our experiments.

4.1 Description of the Test Environment

4.1.1 Hardware and Software

The PMA-based filtering algorithms and YFilter are implemented in the Java language and compiled using JDK version 6. YFilter's implementation was acquired from the YFilter web site [4]. Execution times of the Java-based algorithms were measured using the wall-clock time given by the system time. Memory consumption was measured by examining the currently used heap of the Java virtual machine. Before each measurement all non-referenced objects are garbage-collected from the heap. The specification of the Java virtual machine is available in the work by Lindholm and Yellin [43].

The implementation of the lazy DFA algorithm was downloaded from the XML toolkit site [9]. The matching engine was implemented in the xdetector module in version 1.11 of the XML toolkit. The algorithm, written in C++ [70], was compiled in Linux by GCC 4.1. Wall-clock time was calculated by the C++'s `clock()` function. This setup is beneficial for the lazy DFA, since C++ is generally faster than Java. The mem-

ory usage of the lazy DFA process (data resident size) was measured by Linux's `ps` utility.

All 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 32-bit Debian Linux 2.6.18 operating system with the Sun Java virtual machine 1.6.0_16 installed. The server was dedicated to testing, and only necessary operating system processes were active during the test runs.

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 and the throughput of the C++ SAX parser was 33–42 MB/sec.

4.1.2 Statistical Analysis

For each measurement, the results are averages of five independent test runs. Error bars in the plots denote standard deviation. Standard deviation was calculated by using the biased method:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2},$$

where $x_1, x_2, \ldots x_N$ are random values from a finite data set and $\bar{x}$ is the average value.

4.1.3 Data Sets Used in the Experiments

To experiment with the filtering algorithms we used four data sets. Table 4.1 summarizes the characteristics of these data sets and their DTDs. The *document depth* denotes the XML document depth and the *avg. path length* the average path length of all XML elements in the document. The number of root-to-leaf paths is counted by enumerating all non-cyclic paths from the root node of the DTD to the leaf nodes (in this case we define the root node as the root element of the corresponding XML data set).

The protein-sequence database (Figure 4.1) is non-recursive. The DTD of the astronomical NASA data (Figure 4.2) has one cycle that makes the data recursive (the NASA DTD was generated from the data). Both these DTDs are fairly simple and tree-like; they have only few nodes

	File size (MB)	Document depth	Avg. path length	#elements (DTD)	#edges (DTD)	Recursive	#root-to-leaf paths (DTD)
Protein	24.0	7	5.1	66	83	No	66
NASA	23.8	8	5.5	61	82	Yes	67
NewsML	2.6	10	6.8	114	633	Yes	88100
Treebank	82.5	36	7.9	250	1765	Yes	$> 10^{11}$

Table 4.1. The DTDs and data sets used in the experiments.

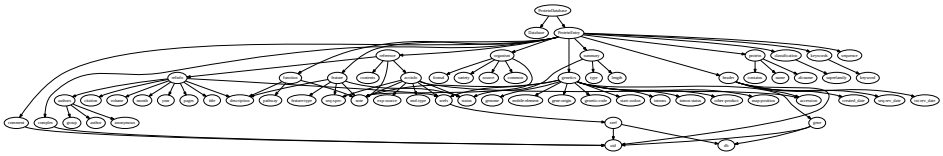


Figure 4.1. The graph schema of a 66-element non-recursive DTD for protein-sequence data.

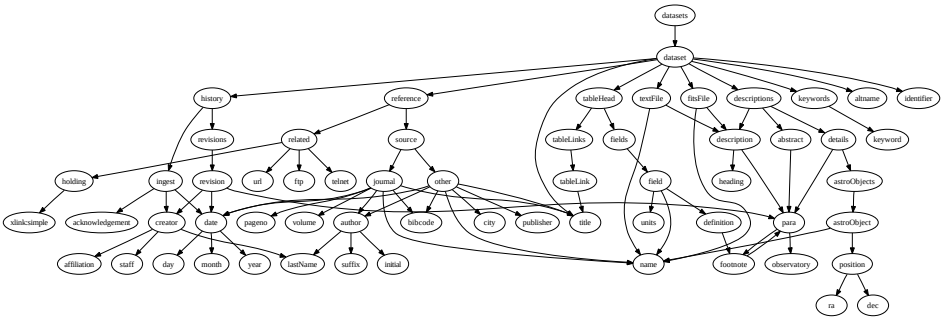


Figure 4.2. The graph schema of a 61-element slightly recursive DTD for NASA data.

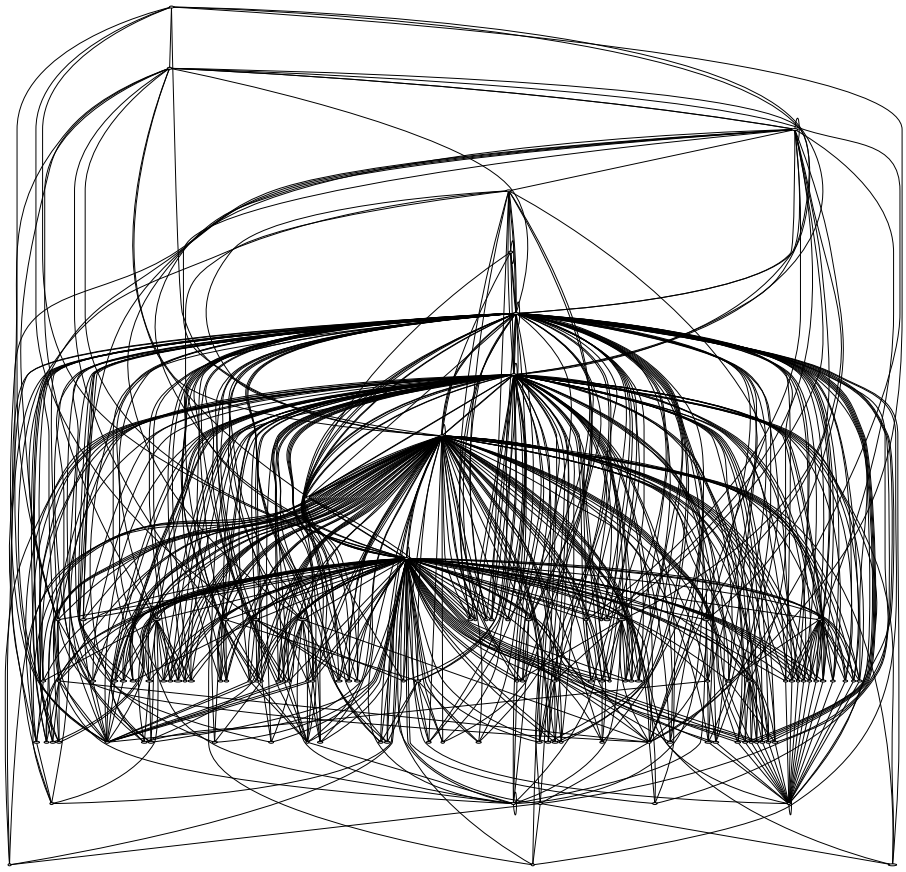


Figure 4.3. The graph schema of a 114-element recursive DTD for NewsML data.

```
/ProteinDatabase/ProteinEntry/summary/status
//uid
/*/ProteinEntry/feature/note
/ProteinDatabase/ProteinEntry/function/pathway
/*/ProteinEntry/*/accinfo//seq-spec
/ProteinDatabase//keyword
/ProteinDatabase/ProteinEntry//superfamily
/*/ProteinEntry/organism//formal
/ProteinDatabase/*/*/note
/ProteinDatabase/ProteinEntry/organism/variety
//ProteinDatabase/ProteinEntry/*/*
/*/*/*
```

Figure 4.4. Part of a filter workload generated for the protein-sequence DTD with $prob(//) = prob(*) = 0.2$.

with many incoming edges and a small number of root-to-leaf paths. The protein and NASA data sets were obtained from the XML Data Repository of the University of Washington [71]. The size of the protein data set is 683 MB, but we have used an extract of 24 MB in our experiments.

The NewsML (Figure 4.3) and treebank data sets represent more complex XML data. NewsML is a structural framework for multi-media news. The NewsML DTD and data were obtained from the NewsML web site [35]. The data consists of news from the Reuters news agency. The treebank database of English sentences was obtained from the XML Data Repository [71] (the treebank DTD was generated from the data). These DTDs have a much higher number of edges than nodes, as can be seen from Table 4.1 and from Figure 4.3. Thus the number of root-to-leaf paths is high for both DTDs. The NewsML DTD has 88 100 acyclic root-to-leaf paths. The depth-first search counted more than 10^{11} acyclic root-to-leaf paths for the treebank DTD until the calculation was aborted.

We used the XPath query generator obtained with the YFilter release [4] to generate workloads of linear XPath filters without predicates. The generated filters are consistent with the given DTD. The generation of filters can be parameterized by the number and maximum nesting depth of filters and by the following parameters: $prob(//)$, the probability of "//" being the operator at a location step, and $prob(*)$, the probability of "*" occurring at a location step. The generator can be set to generate a filter workload that can contain duplicate filters, and one that contains only distinct filters. Figure 4.4 shows a set of XPath filters generated using the protein DTD with $prob(//) = prob(*) = 0.2$.

4.2 Initialization of the PMA

We measured the times needed for reading the XPath filters from disk and parsing them and preprocessing the PMA. Also the number of states in the PMA and its initial size in memory were measured. The PMA is built only at system startup or when new filters are added into the system. The XPath parser has been programmed by using the ANTLR parser generator [53]. Preprocessing the automaton includes building the goto and fail functions (see Algorithms 3.1 and 3.2), but not the initialization of the output function.

Table 4.2 gives the results for this experiment. The workloads of 1 000 and 10 000 distinct filters were generated with $prob(*) = prob(//) = 0.2$. The measurements were done with the dynamic PMA algorithm, but the PMA FB gives the same results for this experiment. The standard deviation was less than 1.5 % for each result. It can be seen from Table 4.2 that parsing the XPath filters and preprocessing of the PMA is fast; even with 10 000 complex treebank filters parsing and preprocessing took less than two seconds altogether. The number of states of the PMA is moderate as well as its initial size in the memory. The size of the PMA was measured after building the goto and failure functions, but the procedure *initialize* had not been called yet, so current output was empty. In Section 4.4 we discuss memory allocation during processing of the input document, when output sets are dynamically updated.

	protein		NASA		NewsML		treebank	
# XPath filters	1000	10000	1000	10000	1000	10000	1000	10000
Parsing time (sec)	0.243	0.831	0.254	0.878	0.271	0.922	0.305	1.09
PMA build time (sec)	0.118	0.393	0.121	0.490	0.118	0.464	0.156	0.717
# states in the PMA	312	443	400	679	375	1474	1136	8698
Size in memory (MB)	0.857	7.91	0.972	8.32	1.07	8.65	1.73	12.7

Table 4.2. Times spent on constructing the filtering PMA, the number of states, and the size of the PMA in memory. The filter workloads were generated with $prob(*) = prob(//) = 0.2$.

4.3 Filtering Performance

4.3.1 Filtering with the bare AC

We begin by measuring the filtering performance of the bare AC algorithm. In this case the XPath workload will be generated with settings $prob(//) = prob(*) = 0.0$. When the maximum depth of the filters was set to the XML document depth, the filter generator managed to generate only 66 different filters for the protein DTD, 72 for the NASA DTD and, 1363 for the NewsML DTD. Thus in order to get meaningful results we measure the performance of the bare AC algorithm only with the most complex treebank data set. With the treebank DTD the XPath filter generator can easily generate up to 100 000 distinct linear filters without wildcards and descendant operators.

Because of the complexity of the treebank DTD, only less than 1 % of the filters actually match the XML input document. This kind of setting can be realistic in information filtering, even though with simpler DTDs or workloads having wildcards and descendant operators the proportion of matching filters is usually much bigger.

In our measurements the filtering times exclude the time needed for parsing the XPath filters and any preprocessing tasks for the algorithms. For the PMA-based algorithms and for the lazy DFA the filtering times include SAX parsing the XML input document, but for YFilter the document parsing time was not included.

Figure 4.5 shows the filtering times spent on filtering the 82.5 MB treebank data set when the number of distinct linear XPath filters (generated with $prob(//) = prob(*) = 0.0$) varies from 10 000 to 100 000. It can be seen from the figure that the lazy DFA algorithm is the most efficient with 10 000 filters, but when the number of filters increases, bare AC is faster. The performance of the bare AC and YFilter is nearly constant regardless of the number of filters.

When the filters do not have wildcards or descendant operators, the filtering automaton for YFilter becomes close to a DFA. In fact, an NFA without epsilon transitions is a DFA in this case. Also the performance of the bare AC is also nearly linear in the length of the input; it has been proven that the Aho-Corasick PMA makes at most $2|x|$ transitions for the input string of length $|x|$ [5].

Because the bare AC algorithm can handle leading descendant operators, we also ran tests with workloads having "//" operators at the beginning of the filters. In the workloads created by our modified filter generator about 80 % of the filters begin with "//". Figure 4.6 shows the

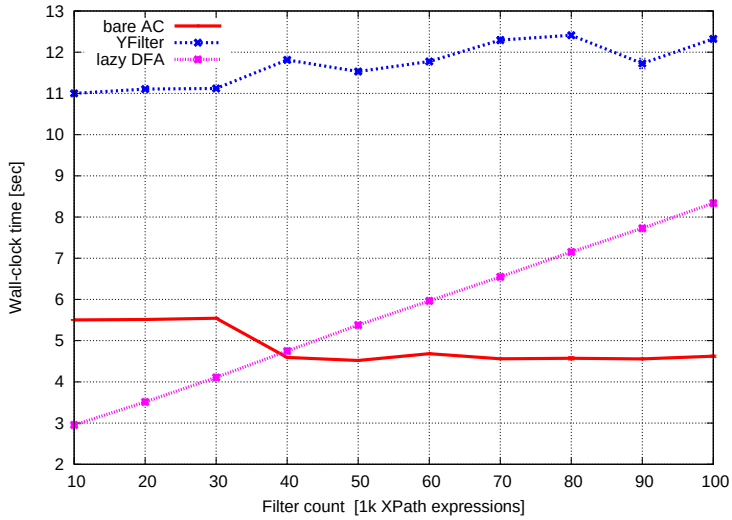


Figure 4.5. Filtering times of the 82.5 MB XML treebank data set, using the bare AC, YFilter and lazy DFA algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.0$.

filtering times spent on filtering the treebank data set with these workloads. Only graphs for the bare AC and YFilter are shown, because the lazy DFA was significantly slower with these workloads. With 10 000 filters the lazy DFA took 112 seconds and with 50 000 filters 988 seconds to process the input document. In this experiment the bare AC showed 3.7–3.9 times better performance than YFilter.

4.3.2 Filtering with the dynamic PMA and PMA FB

To measure the filtering performance of the dynamic PMA and PMA FB we created workloads of distinct filters having also wildcards and non-leading descendant operators. Figures 4.7–4.10 show the filtering times with the protein, NASA, NewsML, and treebank data sets when the number of filters vary from 2 000 to 20 000 with the protein data set and from 10 000 to 100 000 with the other data sets. The workloads of distinct linear filters were generated with $prob(//) = prob(*) = 0.2$ and with the maximum depth set as the depth of the corresponding XML document.

With the protein data set the proportion of matching filters is 90 % with 2 000 filters and 97 % with 20 000 filters. The protein DTD is so simple that is difficult to generate a large number of filters that do not match the input document. It can be seen from Figure 4.7 that PMA FB is the most scalable algorithm with the protein data set. It is somewhat

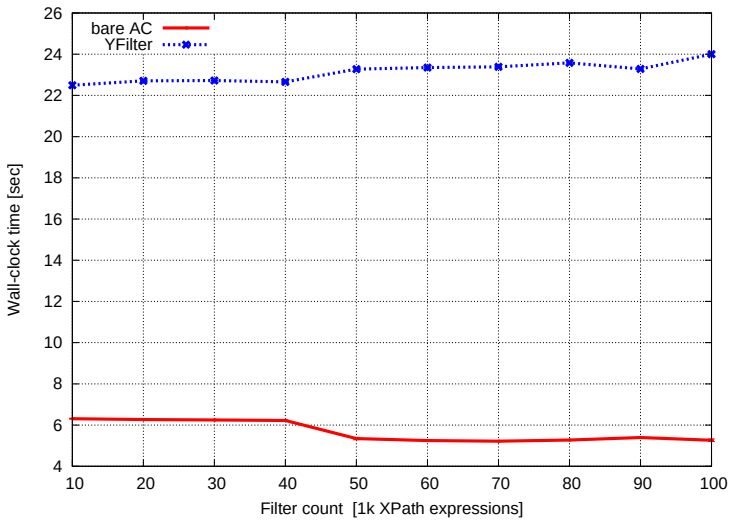


Figure 4.6. Filtering times of a 82.5 MB XML treebank data set, using the bare AC and YFilter algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.0$, but the filter workload was modified so that 80 % of the filters began with a descendant operator.

surprising that the performance of the lazy DFA was behind PMA FB, since a DFA should be very efficient with this kind of simple data and the lazy DFA should not construct too many states. However, with this relatively shallow data PMA FB's stack-based approach for storing the current output sets seems to work very efficiently.

With the NASA data set the proportion of matching filters was 65 % with 10 000 filters and 52 % with 100 000 filters. With this data set the proportion of matching filters decreases as the number of distinct filters increases. One reason for this is that when the number of filters increase, the XPath filter generator is forced to generate more filters of the maximum depth, but the probability of an occurrence of this kind of path in the XML data is low. As is seen from Figure 4.8, with the NASA data set the lazy DFA performs best.

Figure 4.9 shows the filtering times with the recursive and fairly complex NewsML data set. In this case the proportion of matching filters was 19 % with 10 000 filters and 10 % with 100 000 filters. When the complexity of the XML data increases, YFilter starts to show the best performance. With 100 000 filters the filtering time is 33.3 sec with YFilter, 36.3 sec with the lazy DFA, and 45.3 sec with PMA FB.

Figure 4.10 shows the filtering times with the highly recursive and complex treebank data set. In this case the proportion of matching filters

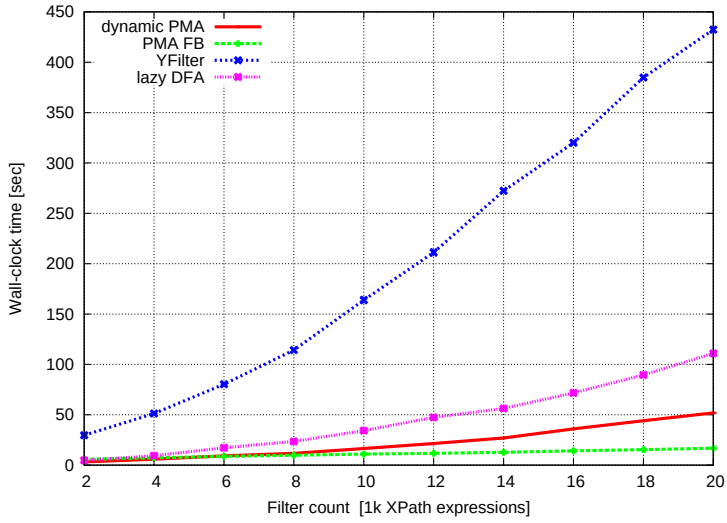


Figure 4.7. Filtering times of the 24 MB XML protein-sequence data set, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

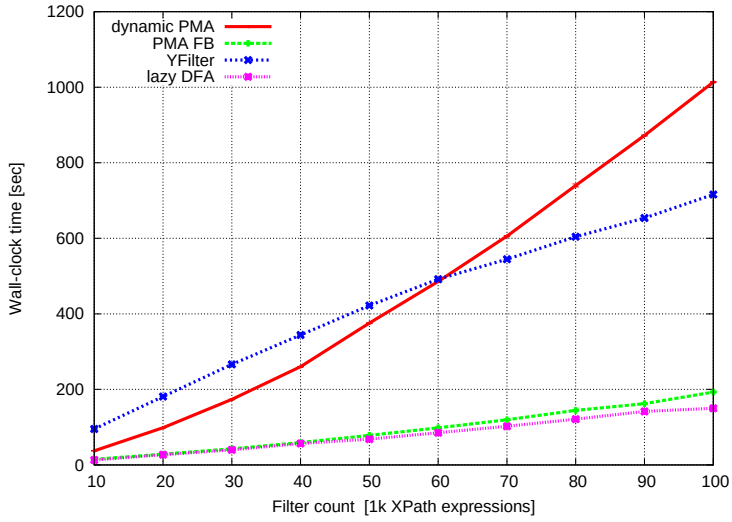


Figure 4.8. Filtering times of the 24 MB XML NASA data set, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

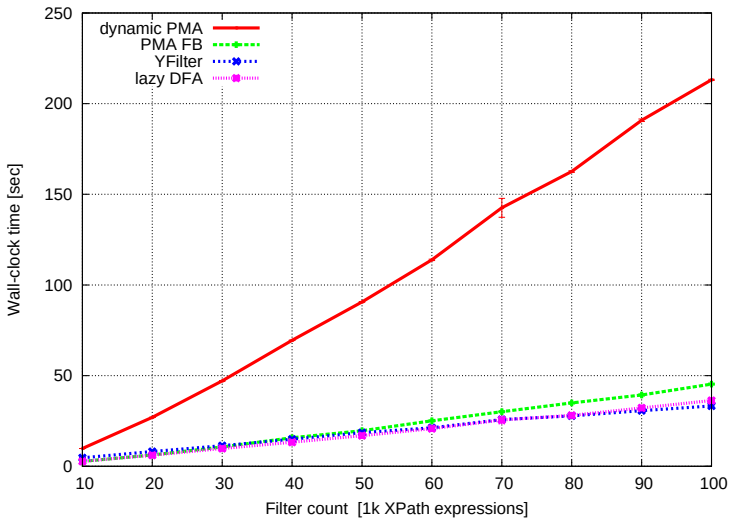


Figure 4.9. Filtering times of the 2.6 MB XML NewsML data set, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

was 15 % with 10 000 filters and 7 % with 100 000 filters. The lazy DFA is excluded from this experiment, because its performance decreases quickly as the complexity of the XML data increases. With 10 000 filters it took 756 sec for the lazy DFA to filter the 82.5 MB treebank data set. It can be seen from the figure that with this data set YFilter is the most scalable algorithm. With 10 000 filters PMA FB is 2.1 times more efficient than YFilter, but with 100 000 filters YFilter is 1.7 times more efficient than PMA FB.

Figures 4.11–4.14 show the filtering times of the algorithms with respect to $prob(*)$, when $prob(//)$ has been set to 0.2, the number of filters to 4 000, and the maximum depth of the filters to the depth of the corresponding data set. With the protein and NASA data sets the filtering speeds of YFilter and lazy DFA decrease a bit as the number of wildcards increases. The PMA-based filtering algorithms do not seem to be sensitive to the number of wildcards. With the NewsML data set the filtering speed of the lazy DFA is not sensitive to $prob(*)$, but the other algorithms slow down somewhat as the number of wildcards increases. With the treebank data set all the algorithms (the lazy DFA is excluded from this experiment) decelerate as $prob(*)$ increases.

Figures 4.15–4.18 show the filtering times of the algorithms with respect to $prob(//)$, when $prob(*)$ has been set to 0.2. The results indicate that with the protein data set the filtering speed of the dynamic PMA

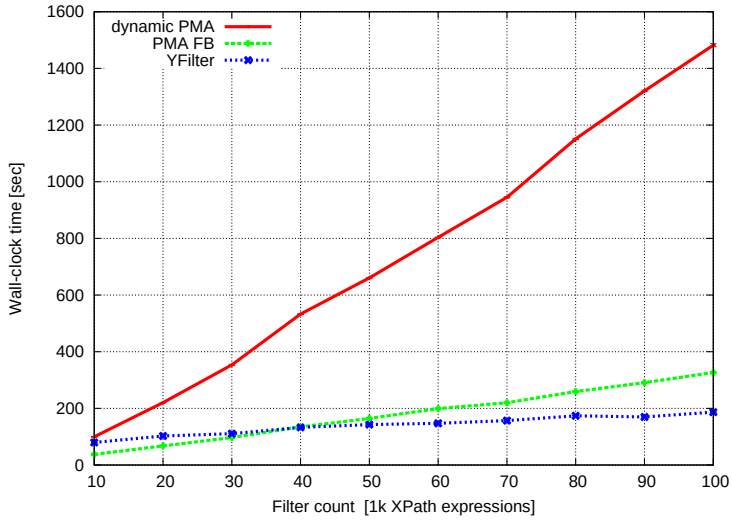


Figure 4.10. Filtering times of the 82.5 MB XML treebank data set, using dynamic PMA, PMA FB, and YFilter algorithms. The distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

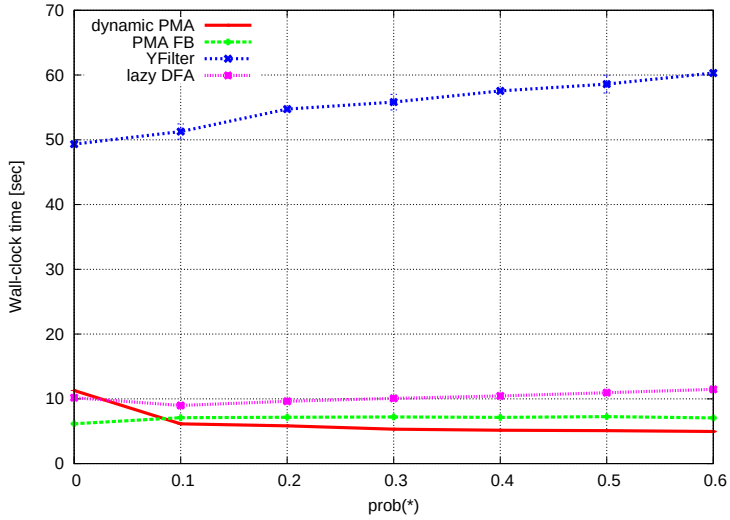


Figure 4.11. Filtering times of the 24 MB XML protein-sequence data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

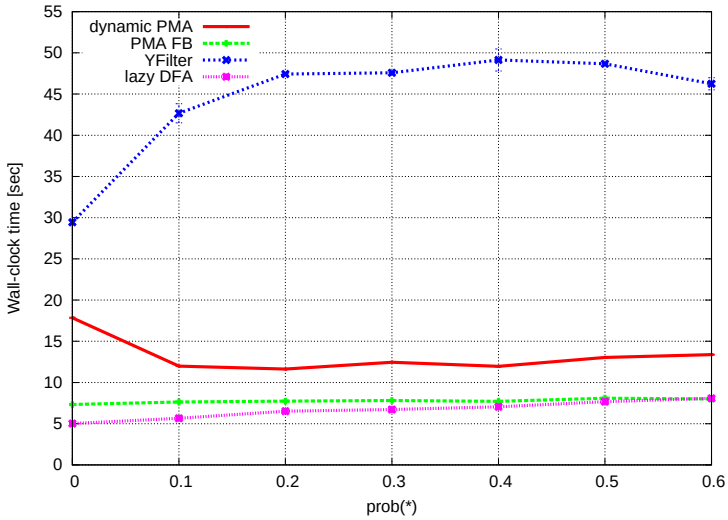


Figure 4.12. Filtering times of the 24 MB XML NASA data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

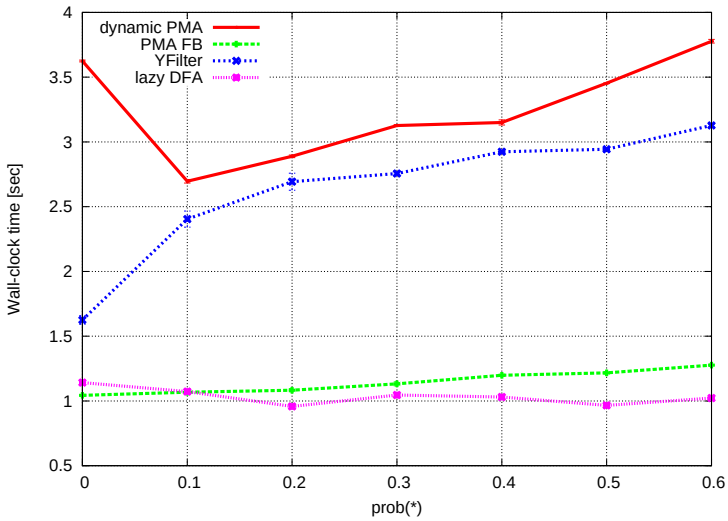


Figure 4.13. Filtering times of the 2.6 MB XML NewsML data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

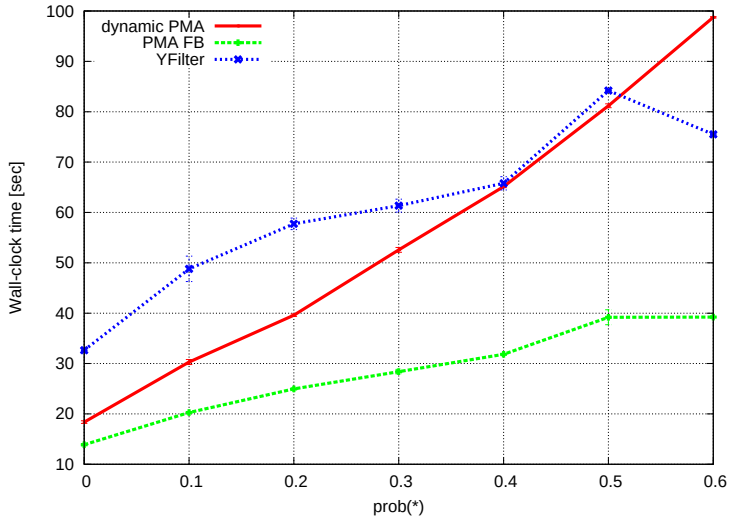


Figure 4.14. Filtering times of the 82.5 MB XML treebank data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, and YFilter algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

and YFilter somewhat decrease as the number of descendant operators increases. PMA FB and lazy DFA are not so sensitive to the number of descendant operators with the protein data set. However, with the other data sets the filtering speed of all the algorithms decrease as the number of descendant operators in the filters increases.

We also measured the filtering speed when the maximum depth of the filters varies. In this case we used the recursive NASA data set and workloads of 4 000 distinct filters generated with $prob(//) = prob(*) = 0.2$. We used values $5, 6, \ldots, 15$ as the maximum depth. As the depth of the XML data is 8, the proportion of matching filters decreases as the depth of the filters increases beyond that value. When the maximum depth was 5, the proportion of matching filters was 95 % and for depth 15, 64 % of the filters matched. Figure 4.19 shows the results of this experiment. It can be seen that the performance of our PMA-based algorithms and YFilter decreases as the depth of filters increases. The lazy DFA is not sensitive to filter depth.

Our initial assumption was that PMA FB is more efficient than the dynamic PMA; our experiments verify this. The experiments also show that the PMA FB algorithm performs well with both simple and complex XML data. PMA FB is significantly more efficient than YFilter with the simple protein and NASA data sets, but with complex and recursive

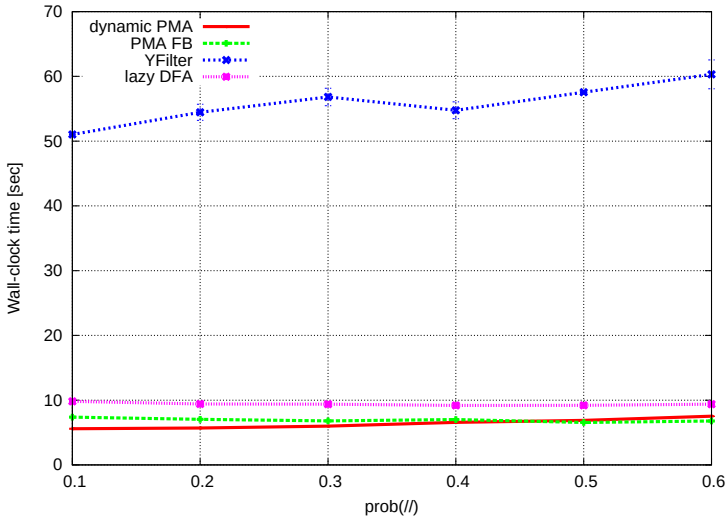


Figure 4.15. Filtering times of the 24 MB XML protein-sequence data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

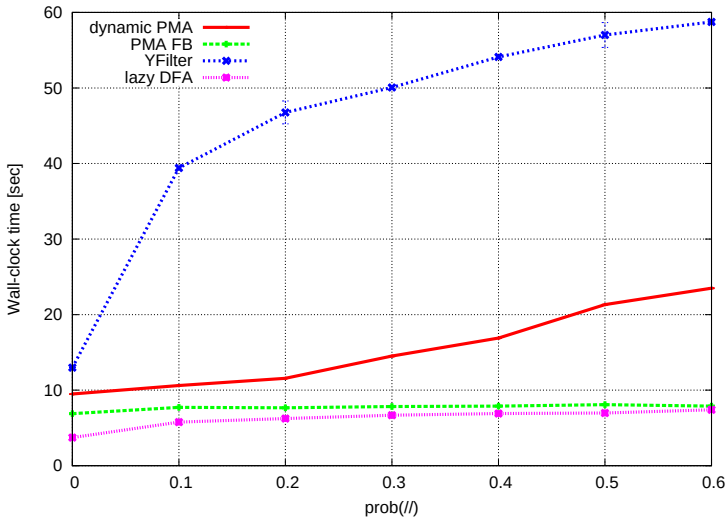


Figure 4.16. Filtering times of the 24 MB XML NASA data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

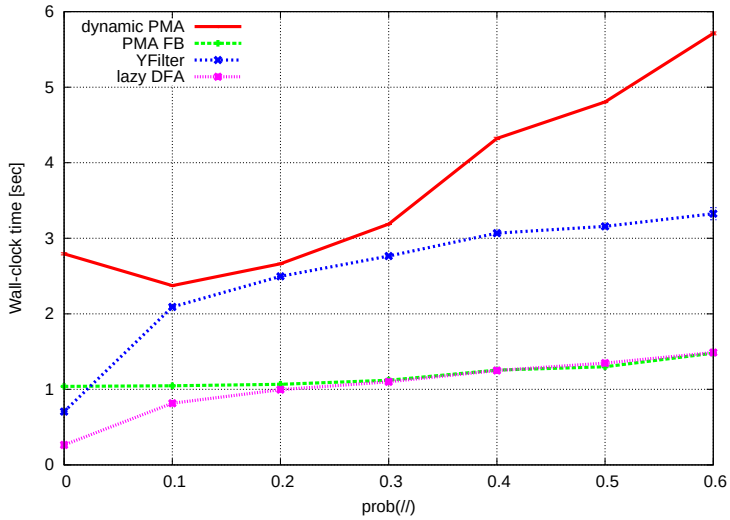


Figure 4.17. Filtering times of the 2.6 MB XML NewsML data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

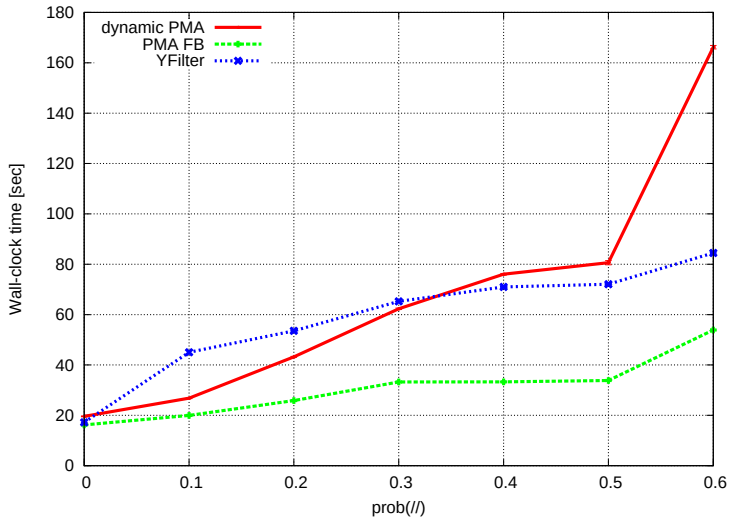


Figure 4.18. Filtering times of the 82.5 MB XML treebank data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, and YFilter algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

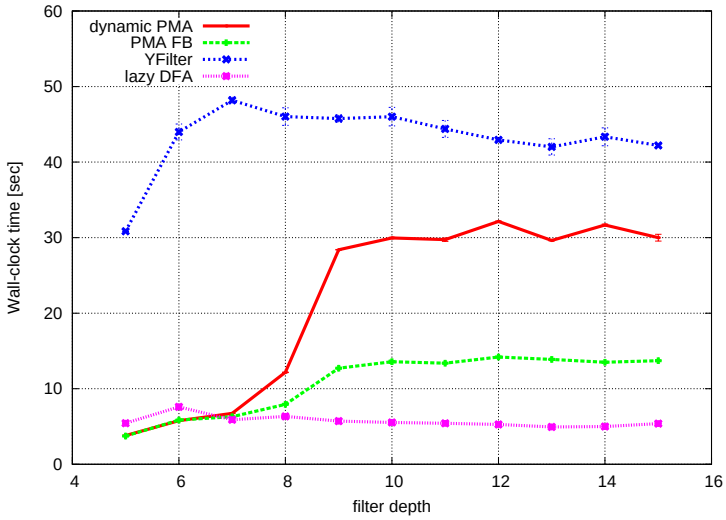


Figure 4.19. Filtering times of the 24 MB XML NASA data set with respect to maximum depth of filters, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = prob(*) = 0.2$.

NewsML and treebank data sets YFilter performs better. The lazy DFA can be used to efficiently filter XML data having a simple or a moderately complex DTD, but it is inapplicable with highly complex and recursive XML data.

4.3.3 Filtering with the static PMA

We also measured the filtering performance of the static PMA algorithm with the four data sets. As stated in Section 3.3.2, with practical XML data sets the filtering performance of the static PMA is likely to be worse than that of the dynamic PMA. The reason for this is that the number of occurrences of all keyword instances in the filter workload $occ(keywords)$ is greater than the number of occurrences of all filter prefixes $occ(pref(Q))$. Thus the factor $O(\log L \times occ(keywords))$ in Theorem 3.2 becomes greater than the factor $O(\log(L + M + N) \times occ(pref(Q)))$ in Theorem 3.3.

Table 4.3 shows the number keyword instance occurrences, the number of filter prefix occurrences, and the number of output tuples actually traversed by the static PMA, dynamic PMA, and PMA FB algorithms. The workloads of distinct linear filters were generated with $prob(//) = prob(*) = 0.2$ and with the maximum depth set as the depth of the corresponding XML document. For the static PMA, the number of

	protein		NASA	
# XPath filters	1000	10000	1000	10000
occ(*keywords*)	1.00×10^8	1.26×10^9	1.06×10^8	1.10×10^9
occ(*pref*(*Q*))	1.58×10^7	3.06×10^8	1.41×10^7	1.35×10^8
static PMA				
traversed tuples	1.15×10^7	4.15×10^7	1.35×10^7	2.70×10^8
filtering time (sec)	3.78	13.9	4.05	73.7
dynamic PMA				
traversed tuples	2.01×10^5	4.66×10^6	6.86×10^5	1.33×10^7
filtering time (sec)	2.39	13.5	3.10	31.5
PMA FB				
traversed tuples	9.12×10^5	5.37×10^6	1.16×10^6	1.37×10^7
selected tuples	2.00×10^5	4.65×10^6	6.72×10^5	1.30×10^7
filtering time (sec)	4.18	10.77	3.98	12.6

	NewsML		treebank	
# XPath filters	1000	10000	1000	10000
occ(*keywords*)	5.69×10^6	6.69×10^7	4.51×10^8	4.46×10^9
occ(*pref*(*Q*))	3.33×10^5	4.46×10^6	5.56×10^6	5.02×10^7
static PMA				
traversed tuples	3.79×10^6	5.33×10^7	3.46×10^8	3.92×10^9
filtering time (sec)	0.78	26.5	51.6	1973
dynamic PMA				
traversed tuples	1.67×10^5	2.66×10^6	2.10×10^6	4.21×10^7
filtering time (sec)	0.59	8.67	11.1	88.4
PMA FB				
traversed tuples	2.12×10^5	2.74×10^6	4.73×10^6	5.27×10^7
selected tuples	1.20×10^5	1.96×10^6	1.29×10^6	2.47×10^7
filtering time (sec)	0.43	2.29	12.2	32.0

Table 4.3. The number of occurrences in the input document of keywords and filter prefixes, the number traversed output tuples, and the filtering times with the static PMA, dynamic PMA, and PMA FB algorithms. The filter workloads were generated with $prob(*) = prob(//) = 0.2$.

traversed output tuples is less than the number of all keyword occurrences, because in the filtering case we are only collecting the first occurrence of each filter (see lines 5–7 in Algorithm 3.8). For the same reason, with the dynamic PMA algorithm the number of traversed tuples is lower than the number of filter prefix occurrences.

For the PMA FB algorithm, the number of traversed output tuples can in some cases be higher than the number of filter prefix occurrences, because in the inner for-loop of Algorithm 3.15 we have to traverse all output tuples of state q having $b \leq$ *path-length*, but only those tuples are selected that have $e \geq$ *path-length*. Selected tuples are collected into the set S (line of 8 Algorithm 3.15). Only selected tuples cause insertions of new output tuples into the current output and thus a lookup from the search tree of Figure 3.4 and a possible insertion of a key into the tree. Table 4.3 shows the number of selected output tuples for the PMA FB.

Table 4.3 also shows the filtering times for the algorithms. The results are averages of five test runs and the standard deviation was less than 2 % for each result. With the protein data set the static PMA was only slightly slower than the dynamic PMA. This can be explained by the high number of matching filters in this case; more than 90–95 % of the filters generated from the protein DTD matched the input document. When a filter matches, its output tuples are deleted from the output sets and these tuples are not traversed again during the processing of the rest of the input document. When most of the output tuples are deleted from the output sets quite in the beginning of the processing of the input document, the behavior of the static PMA becomes close to that of the dynamic PMA. With the other more complex data sets the number of matching filters is smaller; with the NASA data set 65–75 %, with the NewsML data set 20–25 %, and with the treebank data set 15–34 % of the filters matched the input document.

It can be seen in Table 4.3 that with the NASA, NewsML and treebank data sets the dynamic PMA is clearly faster than the static PMA. With the treebank data set and with 10 000 filters the dynamic PMA is even more than 20 times more efficient than the static PMA. However, as is evident from our experiments, the PMA FB is the fastest of the PMA-based filtering algorithms in the case of filters containing wildcards and descendant operators.

4.4 Memory Usage

We measured the memory consumption of the algorithms during processing the XML input stream. From the PMA-based algorithms we examined only the behavior of the dynamic PMA and PMA FB. For the PMA-based algorithms, the state transition tables do not change once constructed, but the dynamic modification of the output sets causes allocation of memory during filtering. For YFilter, the state transition tables are also fixed, but the number of active states can become large when the input document is complex. For the lazy DFA, during filtering memory is allocated for the state transition tables of the lazily constructed DFA.

Figures 4.20–4.23 show the memory usage of the algorithms with the four data sets as the processing of the input document proceeds. With the protein, NASA and NewsML data sets workloads of 10 000 distinct filters were used. Because the memory consumption of the lazy DFA was high with the treebank data set, a workload of only 1 000 filters was used in that case. For each data set the distinct filters were generated with $prob(*) = prob(//) = 0.2$.

The x-axis in the figures shows the number of processed element start-tags and the y-axis the amount of allocated memory (in megabytes) for the data structures at that moment. The measurements show that the memory usage of the algorithms does not grow during processing of the XML input stream. An exception of this is the lazy DFA algorithm with the complex treebank data set; in that case the size of the DFA grows relatively fast as processing of the document proceeds. It can also be seen that with each data set the lazy DFA uses more memory than the NFA- and PMA-based algorithms.

The initial sizes of the dynamic PMA and PMA FB algorithms are the same (both algorithms have identical goto and failure transitions), but when the processing starts the dynamic PMA uses more memory with the shallow data sets (protein, NASA and NewsML). The reason for this is that the dynamic PMA maintains three access paths to current output tuples whereas PMA FB uses only one (see Sections 3.3 and 3.4). But when the depth of the XML data increases, PMA FB starts to consume more memory, because the size of the stack for storing the output tuples grows. This behavior is exemplified by the experiment in Figure 4.23, where the memory consumptions of the PMA-based algorithms are more or less the same. With very deep XML data, the PMA FB should consume more memory than the dynamic PMA.

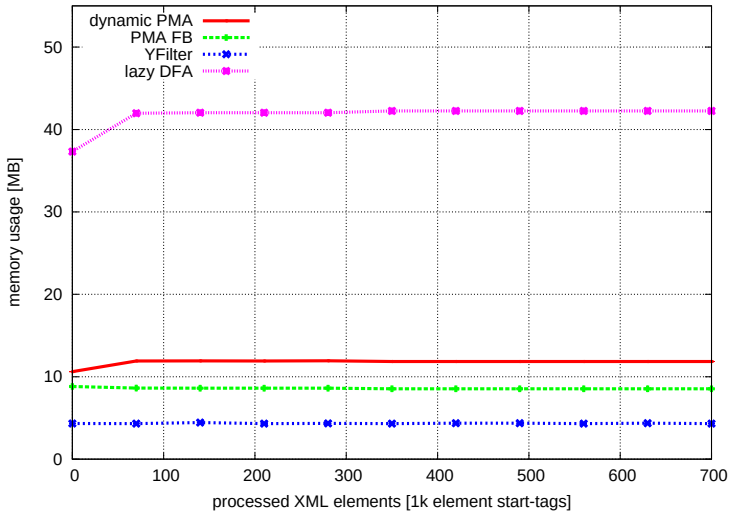


Figure 4.20. Memory usage of the algorithms during processing of the 24 MB XML protein-sequence data set. The workload of 10 000 distinct XPath filters was generated with $prob(//) = prob(*) = 0.2$.

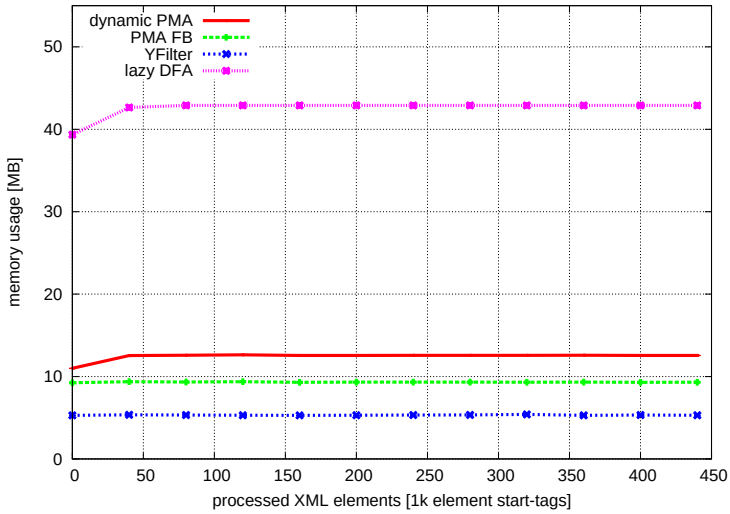


Figure 4.21. Memory usage of the algorithms during processing of the 24 MB XML NASA data set. The workload of 10 000 distinct XPath filters was generated with $prob(//) = prob(*) = 0.2$.

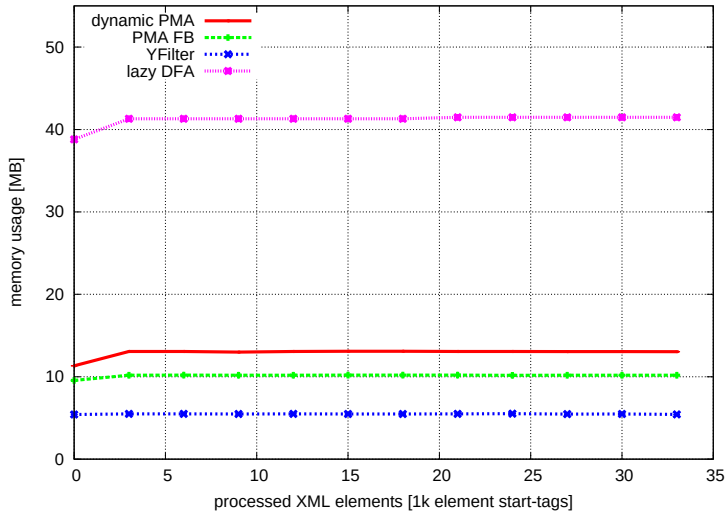


Figure 4.22. Memory usage of the algorithms during processing of the 2.6 MB NewsML XML data set. The workload of 10 000 distinct XPath filters was generated with $prob(//) = prob(*) = 0.2$.

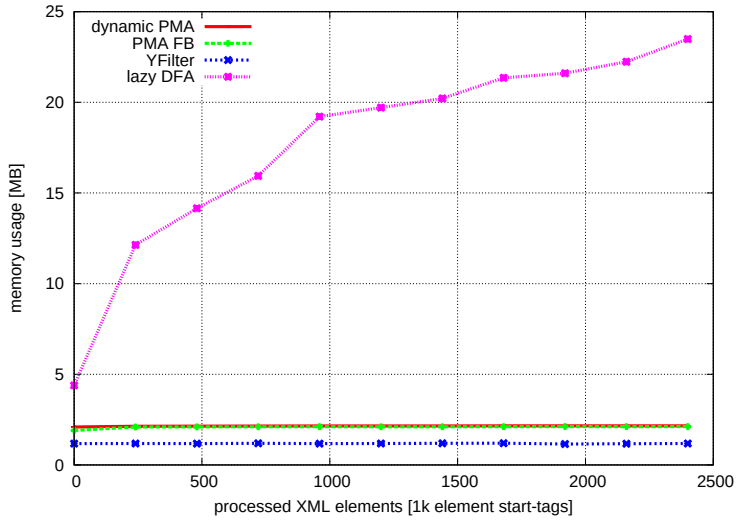


Figure 4.23. Memory usage of the algorithms during processing of the 82.5 MB treebank XML data set. The workload of 1 000 distinct XPath filters was generated with $prob(//) = prob(*) = 0.2$.

CHAPTER 5

Optimization by Filter Pruning

In this chapter we present a new optimization method, called *filter pruning*, that takes as input a DTD and a set of linear XPath filters and produces a set of "pruned" linear XPath filters that contain as few descendant operators "//" and wildcards "*" as possible. Our experimental results show that filter pruning can increase the performance of automata-based filtering significantly (see Chapter 6 and our articles [63–65]).

The filter pruning method was inspired by the *query pruning* technique for optimizing regular path expressions with graph schemas [25] and by the article by Green et al. [28], in which the idea of using query pruning in XPath processing was suggested.

Our ultimate goal is a set of pruned filters in which all wildcards and all non-leading descendant operators have been eliminated, but this may not always be possible because of recursion in the DTD or because the pruning may result in too many or in too large pruned filters. Imposing some simple conditions stating when an operator may be eliminated we can guarantee a polynomial bound on the total size of the pruned filters.

In Section 5.1 we review the concepts of a DTD and a graph schema. Section 5.2 presents the filter pruning algorithm for tree-like DTDs and Section 5.3 for complex and recursive DTDs. In Section 5.4 we show how real XML data sets can be pruned. Section 5.5 reviews some earlier algorithms that utilize the DTD in XML processing.

5.1 DTD and Graph Schema

Given a DTD or schema, let G be its *graph schema* [16], that is, the directed graph whose set of nodes is the set of XML elements in the DTD and that contains a directed edge from node a to node b if and only if b is a child element of a. There is a distinguished node, labeled with # and representing the root element of an XML document that has no incoming

edges. Figure 5.1 shows a sample DTD and its graph schema. This DTD is non-recursive, and hence its graph schema is acyclic.

```
<!ELEMENT a (b|c|d)>
<!ELEMENT b (f*)>
<!ELEMENT c (f*)>
<!ELEMENT d (e*)>
<!ELEMENT e (f*)>
<!ELEMENT f (i|j)*>
<!ELEMENT i (k*)>
<!ELEMENT j (k*)>
```

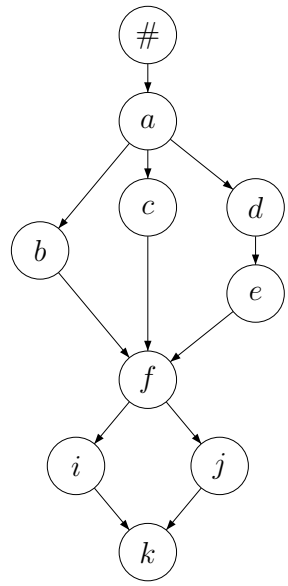


Figure 5.1. A non-recursive DTD and its graph schema.

We say that a linear XPath filter Q is *consistent* with the DTD, if it represents at least one path in the graph schema G of the DTD; in other words, we can find XML elements substituting for the occurrences of the wildcard "$*$" in Q, and strings of XML elements separated by child operators "/" substituting for the occurrences of the descendant operator "//" in Q, such that the resulting string (of XML elements and child operators "/"), when stripped of all child operators, is a path in G from the root element down to some element.

As with YFilter [24], we assume that each subscriber-defined XPath filter is first rewritten into an equivalent form in which (1) a substring of wildcards and one or more descendant operators is replaced by a string containing the wildcards and one descendant operator at the end of the string (e.g., "$//*/*//*$" is replaced by "$/*/*/*//$"), and (2) "//" is removed from "$/*//$" occurring at the end of the filter. For example, by using these rewriting rules the filter "$//*/*//*//a/*//$" is turned into "$/*/*/*//a/*$".

Algorithm 5.1 checks a filter Q for consistency with a DTD, using the graph schema of the DTD. For an element set E, the function *children*(E)

procedure *check-consistency*(Q):

```
1  S ← Q
2  Accessible ← {#}
3  while Accessible is nonempty and S is nonempty do
4    if S is of the form /bS′ where b is an element then
5      if b ∈ children(Accessible) then
6        Accessible ← {b}
7      else
8        Accessible ← empty
9      end if
10   else if S is of the form /∗S′ then
11     Accessible ← children(Accessible)
12   else if S is of the form //bS′ where b is an element then
13     if b ∈ descendant(Accessible) then
14       Accessible ← {b}
15     else
16       Accessible ← empty
17     end if
18   end if
19   S ← S′
20 end while
21 return(Accessible is nonempty and S is empty)
```

Algorithm 5.1. Checking a linear XPath filter Q for consistency with a DTD.

returns the set of all children of all elements in E, and the function *descendant*(E) returns the set of all descendants of all elements in E. The algorithm processes filter Q from left to right, extracting step by step a prefix of Q, and maintaining a set *Accessible* that contains the set of elements accessible in the graph schema from the root element upon reading the so-far-extracted prefix of Q. Filter Q is consistent with the DTD if and only if the algorithm reaches the end of Q with the set *Accessible* nonempty. The variable S stores the non-yet-processed suffix of Q.

5.2 Filter Pruning Algorithm for Tree-Like DTDs

Pruning a filter Q with respect to a DTD is just finding all combinations of substituting elements for as many occurrences of "$*$" as possible, and all substituting element strings for as many occurrences of "//" as possible, such that the pruned filters $Q_1, \ldots, Q_n$ obtained by those substitutions are consistent with the DTD and that their union, denoted by $Q_1 \cup \ldots \cup Q_n$, is *equivalent* to the original filter Q. That is, any XML document that conforms to the DTD matches with Q if and only if it matches with one of the filters Q_i, $i = 1, \ldots, n$.

Original filter	Pruned filter
$/a//f$	$/a/b/f \cup /a/c/f \cup /a/d/e/f$
$//c/f//k$	$/a/c/f/i/k \cup /a/c/f/j/k$
$/*/b$	$/a/b$
$/a/*$	$/a/b \cup /a/c \cup /a/d$
$/a/*/f$	$/a/b/f \cup /a/c/f$
$/*/*/*/*$	$/a/b/f/i \cup /a/b/f/j \cup /a/c/f/i \cup /a/c/f/j \cup /a/d/e/f$

Table 5.1. Original XPath filters and corresponding pruned filters obtained by pruning with the DTD of Figure 5.1. All wildcards "$*$" and descendant operators "//" were eliminated.

Table 5.1 shows a set of filters and the result of pruning them with respect to the non-recursive DTD of Figure 5.1 when all the "$*$" operators and "//" operators are eliminated. For example, in the case of the original filter $/a//f$, we find that the graph schema of the DTD contains three paths from element a, the child of root element #, to element f, namely *abf*, *acf*, and *adef*. Thus the element strings to be substituted for the occurrence of "//" are $/b$, $/c$, and $/d/e$, resulting in the pruned filter $/a/b/f \cup /a/c/f \cup /a/d/e/f$.

For pruning a set of filters with respect to a DTD, we precompute a two-dimensional array, *substitutes*, indexed by pairs of elements in the DTD. For an element pair (a, b), the entry $substitutes[a, b]$ will contain the set of all strings $/c_1/c_2/\ldots/c_n$ such that $ac_1c_2\ldots c_nb$ is a path in the graph schema of the DTD, if the number of such strings is finite and the sum of their lengths falls below a preset limit; otherwise, the entry will be set empty.

procedure *prune-filter*(Q):
1 Rewrite Q
2 **procedure** *prune*$(Q, \#, \epsilon)$

Algorithm 5.2. Pruning a linear XPath filter Q.

procedure *prune*(S, a, P):
1 **if** S is empty **then**
2 output P
3 **else if** S is of the form $/bS'$ where b is an element **then**
4 **if** $b \in$ *children(a)* **then**
5 *prune*$(S', b, P/b)$
6 **end if**
7 **else if** S is of the form $/*S'$ **then**
8 **for** all $b \in children(a)$ **do**
9 *prune*$(S', b, P/b)$
10 **end for**
11 **else if** S is of the form $//bS'$ where b is an element **then**
12 **if** $substitutes[a, b]$ is nonempty **then**
13 **for** all $x \in substitutes[a, b]$ **do**
14 *prune*(S', b, Px)
15 **end for**
16 **else**
17 *prune*$(S', b, P//b)$
18 **end if**
19 **end if**

Algorithm 5.3. Procedure *prune*(S, a, P).

Algorithms 5.2 and 5.3 represent a recursive formulation of a pruning algorithm in which all "$*$" operators are eliminated exhaustively, while the elimination of "//" operators is controlled by the precomputed array *substitutes*. The algorithm can be used to prune with recursive DTDs, but

substrings $a//b$ are, naturally, left unpruned if the DTD contains cycles on paths from element a to element b, or if there are simply too many paths from a to b in the DTD so that the entry $substitutes[a, b]$ has been set empty. All paths between two nodes of a tree-like graph can be calculated with a simple depth-first or breadth-first search procedure (an algorithm for calculating the substitute paths in a complex schema is presented in Section 5.3).

A recursive call $prune(S, a, P)$ takes as arguments a suffix S of the filter Q to be pruned, the corresponding pruned prefix P of Q (pruned in ancestor calls), and the last element a in P. The call extracts a prefix, $/b$, $/*$, or $//b$, from S, prunes it if needed and possible, and concatenates the result to P, to be used as an argument to further recursive calls of the procedure. A recursion path terminates when the suffix S becomes empty; then the argument P represents a complete pruned filter (i.e., one disjunct in the final union of pruned filters) and is written to the output of the algorithm. The pruning of a filter Q in the main program is started by the call $prune(Q, \#, \epsilon)$, where $\#$ denotes the root element and the empty string ϵ indicates a so-far-empty pruned prefix of Q.

Algorithm 5.3 is formulated so that it can also eliminate leading occurrences of the descendant operator "//". Some filtering methods gain from eliminating leading occurrences (see Section 6.2.1). To prevent leading occurrences of "//" from being eliminated, it is sufficient to set entries $substitutes[\#, b]$ empty for all elements b.

If the graph schema of a DTD is acyclic, then the maximum number of pruned filters for a linear XPath filter is bounded by the number of different paths in the schema. For a path, there is only one possible way to replace the wildcards and descendant operators. Because of the XPath filter rewriting rules described in the previous section, we do not need to take into account the case of a descendant operator followed by a wildcard, such as in $//*//*/a$, where the operators can be replaced in more than one way, while in the rewritten filter $/*/*//a$ the operators can be replaced in one way at most, for a given path. For this reason, there can be only one pruned filter for each path in the schema. The maximum number of pruned filters for a linear XPath filter for an acyclic graph schema is given by by the following theorem.

Theorem 5.1. For an acyclic graph schema G and a linear XPath filter Q, the filter pruning algorithm of Algorithm 5.2 produces at most $O(n)$ pruned filters, where n is the number of different paths in G. If G is a tree, then the number of pruned filters is bounded by $O(|G|)$, where $|G|$ is the size of G.

In fact many real DTDs are tree-like, composed of mostly non-recursive elements and having only few elements with many incoming edges. These properties are exemplified by the non-recursive DTD for the protein-sequence database, one of the DTDs from the XML Data Repository at the University of Washington [71] that we have used in our experiments (see the graph schema in Figure 4.1), and by the slightly recursive NASA DTD (see Figure 4.2).

Even in the case of a recursive DTD we may be able to eliminate all "//" operators if we happen to know the maximum nesting depth d of the input documents, or that the recursive rules are never applied in any input document more than d times, where d is a small constant. However, for highly recursive DTDs, such as that for the treebank database [71], this will usually not help to avoid the explosion of the number of generated pruned filters.

5.3 Filter Pruning Algorithm for Complex DTDs

The ultimate goal of exhaustive elimination of "//" operators naturally cannot be achieved when the DTD is recursive, that is, when the graph schema is cyclic, but exhaustive elimination of "//" or "$*$" may also be infeasible with non-recursive DTDs. A simple example of a case in which exhaustive elimination results in an exponential increase in the size of filters with respect to the combined size of the DTD and the filters is the DTD having elements $a_1, \ldots, a_k, a_{k+1}$, $b_1, \ldots, b_k$, $c_1, \ldots, c_k$, where b_i and c_i are children of a_i, and a_{i+1} is a child of both b_i and c_i, $i = 1, \ldots, k$ (see Figure 5.2). Eliminating "//" from the filter $/a_1//a_{k+1}$ results in a union of 2^k pruned filters of size $\Theta(k)$, although the original filter is of size $O(1)$ and the DTD is of size $O(k)$. The same union of pruned filters is also the result of exhaustive elimination of "$*$" from the filter $/a_1/*/a_2/*/\ldots/*/a_{k+1}$, which is of size $O(k)$. Obviously, we must control the number and size of pruned filters to be created.

It is possible to adjust Algorithm 5.3 for a specific DTD so that it will produce a set of sufficiently pruned reasonable-sized filters. Algorithm 5.4 is a pruning algorithm that is suitable for handling complex DTDs, such as the NewsML [35] and treebank [71] DTDs. With this algorithm there is a polynomial upper bound to the size of the pruned filters, regardless of the complexity of the DTD. The pruning of filter Q is started by the call *prune2*$(Q, \#, \epsilon, 0)$. The parameter *max-substitutes* regulates the number entries in the set *substitutes*$[a, b]$. If there are more than *max-substitutes* different paths from element a to element b in the

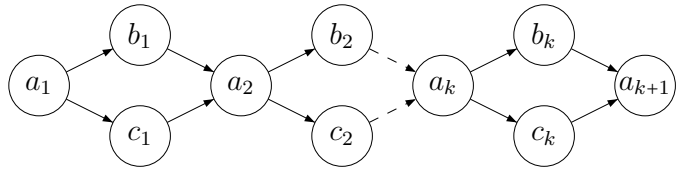


Figure 5.2. A non-recursive DTD that causes exponential increase in the size of the filter when pruning filter $/a_1//a_{k+1}$.

graph schema of the DTD, then entry *substitutes*$[a, b]$ is set empty. The *max-substitutes* threshold is also used to regulate pruning of wildcard operators. Function *numchildren*(E) returns the number of children for element E. If *numchildren*(E) > *max-substitutes*, then the wildcard operator following element E is left unpruned. As an example, for filter $/a/*$, if *numchildren*(a) > *max-substitutes*, then the wildcard operator is left unpruned. The parameter *pruning-count* regulates how many $*$ or // operators from the beginning of a filter Q can be pruned. For example for the DTD in Figure 5.1, if *pruning-count* = 2, then pruning the filter $/*/*/*/*$ results in the pruned filter $/a/b/*/* \cup /a/c/*/* \cup /a/d/*/*$. The following theorem states that our algorithm produces a reasonable-sized filter set regardless of the complexity of the DTD.

Theorem 5.2. When n is the number of "$*$" and "//" operators in a filter Q, then Algorithm 5.4 is guaranteed to produce at most

$$\textit{max-substitutes}^{min(n,\textit{pruning-count})}$$

pruned filters for filter Q.

Algorithm 5.5 is a breadth-first-search-based algorithm for calculating all possible paths between two elements a and b in the DTD graph G: the set *substitutes*$[a, b]$. The parameter *max-substitutes* regulates the number of entries in *substitutes*$[a, b]$. If G contains cycles on paths from element a to element b or if there are more than *max-substitutes* different paths from element a to element b, then *calc-substitutes*(a, b, G) returns an empty set.

Algorithm 5.5 starts by constructing a breadth-first search tree of height 2 starting from node a. Algorithm 5.6 builds the breadth-first tree and it works as follows. An edge (u, v) is added into the tree if v is a successor of u in the DTD graph G, there exists a path from v to b in G and u is not a descendant of v in the breadth-first tree constructed so far. The auxiliary procedure *is-descendant*$(tree, v, u)$ returns *true* if node u is a descendant of node v in the breadth-first tree. The procedure

procedure *prune2*(S, a, P, n):

```
1  if S is empty then
2    output P
3  else if n ≥ pruning-count then
4    output PS
5  else if S is of the form ∘bS′ where ∘ ∈ {/, //} and a = ∗ then
6    prune2(S′, b, P ∘ b, n)
7  else if S is of the form /bS′ where b is an element then
8    if b ∈ children(a) then
9      prune2(S′, b, P/b, n)
10   end if
11 else if S is of the form /∗S′ then
12   if numchildren(a) ≤ max-substitutes then
13     for all b ∈ children(a) do
14       prune2(S′, b, P/b, n + 1)
15     end for
16   else
17     prune2(S′, ∗, P/∗, n)
18   end if
19 else if S is of the form //bS′ where b is an element then
20   if substitutes[a, b] is nonempty then
21     for all x ∈ substitutes[a, b] do
22       prune2(S′, b, Px, n + 1)
23     end for
24   else
25     prune2(S′, b, P//b, n)
26   end if
27 end if
```

Algorithm 5.4. Procedure *prune2*(S, a, P, n).

```
function calc-substitutes(a, b, G):
 1  lastsize ← 0
 2  R ← ∅
 3  for i = 2 to number of nodes in G do
 4     tree ← bfstree(a, b, i)
 5     if tree = empty then
 6        return empty
 7     else if lastsize = size(tree) then
 8        return R
 9     end if
10     lastsize ← size(tree)
11     P ← enumerate-paths(tree, b)
12     R ← R ∪ P
13     if size(R) > max-substitutes then
14        return empty
15     end if
16  end for
17  return R
```

Algorithm 5.5. Procedure *calc-substitutes*(a, b, G).

is-connected(G, v, b) returns *true* if there exists a path from node v to node b in the DTD graph G. Calculation of *is-descendant* is fast, because in the tree structure all nodes contain a pointer to the parent node. We also maintain a hash table accessed by node names and containing sets of tree nodes for the keys. For example, the hash table entry for key f contains a set of tree nodes containing key f. In this way we can quickly locate all possible paths from the root of the tree to the given key; we just need to locate all paths from f to the root node and reverse those paths. The procedure *enumerate-paths*$(tree, v)$ calculates all paths from the root element of the *tree* into the node v. Figure 5.3 shows an example of a breadth-first tree of height 4 constructed by applying Algorithm 5.6 to the DTD graph of Figure 5.1.

If Algorithm 5.5 finds that the beginning of some path contains a cycle, then the calculation of substitute paths is aborted and the entry *substitutes*$[a, b]$ will be set empty. A path $v_1 v_2 \ldots v_n$ contains a cycle, if some node v_i has an edge pointing to node v_j, where $j \leq i$.

The procedure *calc-substitutes*(a, b, G) (Algorithm 5.5) continues to construct breadth-first trees of height i until i is the number of nodes in the DTD graph G or the call *bfstree*(a, b, i) (Algorithm 5.6) returns exactly the same tree than *bfstree*$(a, b, i - 1)$. Also calculation is aborted

function *bfstree*$(a, b, height)$:

```
1  tree ← empty
2  queue ← empty
3  queue.enqueue(a)
4  while queue is not empty and height(tree) ≤ height do
5    u ← queue.dequeue()
6    for each v in children(u) do
7      if v = b or (is-descendant(tree, u, v) = false and
       is-connected(G, v, b) = true) then
8        queue.enqueue(v)
9        add edge (u, v) into tree
10       P ← enumerate-paths(tree, v)
11       if P contains a cyclic path then
12         return empty
13       end if
14     end if
15   end for
16 end while
17 return tree
```

Algorithm 5.6. Procedure *bfstree*$(a, b, height)$.

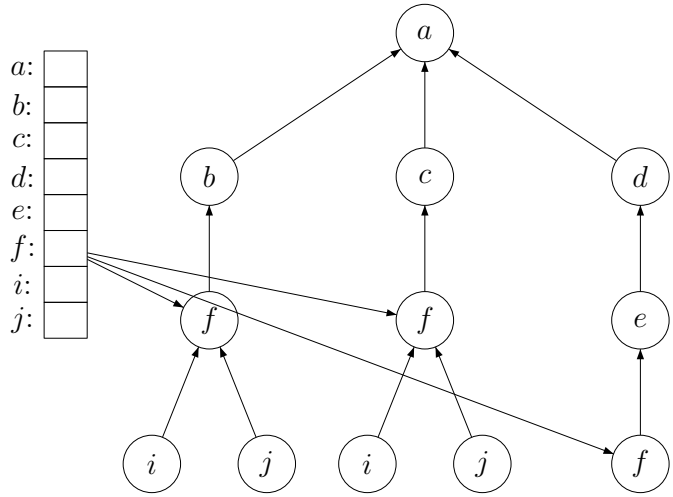


Figure 5.3. A breadth-first tree of height 4 constructed by applying Algorithm 5.6 to the DTD graph of Figure 5.1, starting from node a. A hash table entry f has pointers to all tree nodes containing key f (pointers from other table entries have not been drawn).

if $bfstree(a, b, i)$ returns an empty object. In the worst-case the size of the whole breadth-first tree is exponential in the size of the DTD graph. However, in practice this algorithm works well even with the highly complex and recursive treebank schema, as is demonstrated in the next sections. The reason for this is that the breadth-first tree for the whole graph is never constructed for the given filter workloads, but the possible non-cyclic connecting paths between given elements can be found with breadth-first trees of only moderate height.

5.4 Characteristics of Pruned Workloads

To experiment with the pruning algorithms we used the four data sets and the XPath query generator described in Section 4.1.3. We generated workloads of 10 000 distinct filters for the protein and NASA DTD with different values of $prob(//)$ and $prob(*)$. These workloads were pruned by using the Algorithm 5.3. We measured the number of pruned filters, that is, the total number of pruned XPath expressions in the union filters produced for the 10 000 original filters. Also leading occurrences of descendant operators were eliminated in pruning. Table 5.2 shows the characteristics of the pruned filters. In the case of the protein-sequence

workloads, the number of pruned filters varied between 15 699 and 16 816 when $prob(*)$ varied between 0.2 and 0.6 and $prob(//)$ between 0.2 and 0.6. For the NASA workloads, the number of pruned filters varied between 17 639 and 25 498 respectively.

In pruning the protein-sequence workloads, both "$*$" and "//" operators were eliminated exhaustively. In pruning the NASA workload, the wildcards "$*$" were eliminated exhaustively while not all descendant operators "//" could be eliminated because of the recursion; about 23%–48% of descendant operators remained in the pruned filter workloads. For the NASA workloads, the pruning was regulated by setting $substitutes[a, b]$ empty for element pairs (a, b) involved in a recursion cycle.

It is likely that any filtering algorithm can also gain from the fact that filter pruning can reduce the number of distinct filters. The idea is to perform filtering only for the distinct filters in the pruned workload, and to maintain a mapping from pruned distinct filters to original filters, so that matching original filters can be reported correctly. The mapping is stored in an array *original-filters*, where an entry $original\text{-}filters[i]$, for pruned filter number i, contains the numbers of original filters from which filter i was pruned.

We measured the numbers of distinct filters in the pruned workloads and noticed that they were surprisingly few. The pruned protein workload had only 90 distinct filters in each case. The number of distinct filters in the NASA workloads varied between 486 and 532. The tree-like shape of the DTD of these data sets is the reason for the fact that the number of distinct pruned filters stays within moderate limits even for high values of $prob(*)$ and $prob(//)$.

Workload parameters		protein		NASA	
$prob(*)$	$prob(//)$	# pruned	# pruned distinct	# pruned	# pruned distinct
0.2	0.2	15699	90	17670	486
0.2	0.4	15752	90	17657	522
0.2	0.6	15852	90	17639	532
0.4	0.2	16459	90	22004	469
0.6	0.2	16816	90	25498	504

Table 5.2. Characteristics of pruned filters for workloads of 10 000 distinct filters generated for protein and NASA DTD.

For experimentation with complex and recursive DTDs we used the NewsML and treebank data sets. We searched the optimal values for *max-substitutes* and *pruning-count* parameters for a workload of 10 000

distinct filters generated from the NewsML DTD having $prob(//) = prob(*) = 0.2$. The maximum depth of filters was set to 10, which is the maximum depth of the NewsML data. We measured the number of distinct filters in the pruned workload when *max-substitutes* had values $10, 20, \ldots, 100$ and *pruning-count* had values $1, 2, \ldots, 10$. Figure 5.4 shows the number of distinct pruned filters in relation to *max-substitutes* and *pruning-count* parameters. The smallest number of distinct filters 4 793 was calculated with values *max-substitutes*= 10 and *pruning-count*= 10. With all configurations the number of distinct pruned filters was less than 10 000.

The previous experiment was repeated with the treebank DTD. In this case the maximum depth of filters was set to 36, which is the maximum depth of the treebank data. Figure 5.5 shows the results of this experiment. The number of distinct filters varied between 12 697 and 121 766. The smallest number of distinct filters was acquired with values *max-substitutes* = 10 and *pruning-count* = 1. Thus with a highly complex data set we were not able to reduce the number of distinct filters by filter pruning.

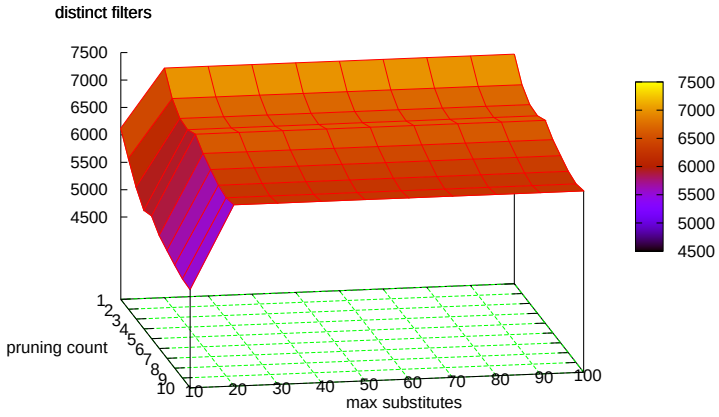


Figure 5.4. The number of distinct pruned filters in relation to the *max-substitutes* and *pruning-count* parameters. The original workload of 10 000 distinct filters was generated from the NewsML DTD with $prob(//) = prob(*) = 0.2$.

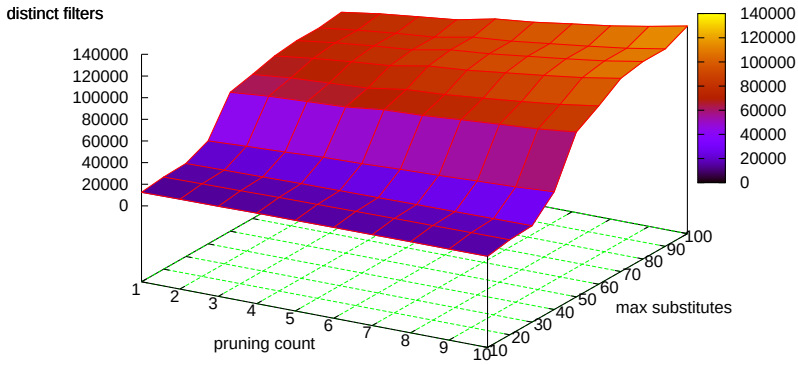


Figure 5.5. The number of distinct pruned filters in relation to the *max-substitutes* and *pruning-count* parameters. The original workload of 10 000 distinct filters was generated from the treebank DTD with $prob(//) = prob(*) = 0.2$.

5.5 Related Work

The filter pruning optimization was inspired by recent automata-based methods for XML filtering with XPath expressions [24, 28, 51], and by the query-pruning technique of Fernández and Suciu [25] who used graph schemas to optimize regular path expressions. The idea of query pruning is that the selectivity of the schema is embedded into the queries. The technique of Fernández and Suciu takes a user-provided query (a path expression) and a graph schema as input and constructs the product automaton of two NFAs: one that accepts the paths denoted by the query and another that accepts the paths denoted by the graph schema; from this product NFA, a pruned query is constructed by taking into account only those paths in the NFA that lead from the initial state to one of the final states.

Green et al. [28] present the idea of speeding up XPath processing on XML streams by removing the //'s and $*$'s based on the DTD, but they did not research that any further or develop any algorithms based on the idea. Florescu et al. [26] present the BEA XQuery evaluation engine. Their system exploits schema information for removing descendant operators from XQuery queries.

The SFilter system by Lee et al. [41, 42] utilizes the schema informa-

tion in XML filtering with XPath expressions. SFilter builds a product automaton from an automaton representing a (regular) XPath expression and an automaton representing the DTD. The difference of this method when compared to our filter-pruning method is that in our method the goal is to prune the filters such that the pruned form has the same syntax as the original expressions, only with fewer //'s and $*$'s, thus contributing to path sharing in automata-based filtering [24, 28]. Also the method of Lee et al. is not applicable with complex schemas, such as the NewsML and treebank DTDs.

Xiao-Ling and Min [75] have also developed a method similar to filter pruning, where //'s and $*$'s are eliminated based on the DTD. They have experimentally shown that the lazy DFA gains from their method. However, it is unclear how their algorithm can be applied in the case of complex and recursive schemas.

CHAPTER 6

Performance Gain of Filter Pruning

In this chapter we examine how filter pruning improves the filtering performance of our PMA-based filtering algorithms (described in Chapter 3), and of YFilter [24] and the lazy DFA [28] algorithms. First we measure the time needed to prune given workloads of linear XPath filters. Then we can execute the actual filtering algorithms with original unpruned and pruned workloads to see the performance gain achieved from filter pruning.

The hardware and software environment and the XML data sets used in the experiments of this chapter are the same than described in Section 4.1. Again, our own algorithms (Algorithms 5.1–5.6) have been programmed in the Java language.

6.1 Performance of the Pruning Algorithm

We measured the execution time of Algorithm 5.3 with various workloads. We generated 10 000 distinct filters for the protein and NASA DTDs when $prob(*)$ and $prob(//)$ varied between 0.2 and 0.6 and measured the pruning times of these workloads. The pruning time included reading the XPath filters from disk and parsing and pruning the filters. The XPath parser has been programmed in Java by using the ANTLR parser generator [53]. Table 6.1 shows the results of the experiment. It can be seen that in all cases the pruning time was small, less than 2 seconds in each setting. For both workloads, the $prob(//)$ does not seem to have much effect to the pruning time. However, the pruning time increases slightly when there are more wildcards in the filters.

The previous experiment was repeated with Algorithm 5.4 and with NewsML and treebank DTDs. We generated 10 000 distinct filters for both DTDs having $prob(//) = prob(*) = 0.2$ and measured the pruning time with *max-substitutes* values $10, 20, \ldots, 100$ and *pruning-count* values

Workload parameters		protein	NASA
$prob(*)$	$prob(//)$	time (sec.)	time (sec.)
0.2	0.2	1.33	1.38
0.2	0.4	1.29	1.32
0.2	0.6	1.20	1.28
0.4	0.2	1.37	1.48
0.6	0.2	1.40	1.59

Table 6.1. Running time of the pruning algorithm for workloads of 10 000 distinct filters generated for the protein and NASA DTDs.

$1, 2, \ldots, 10$. Figure 6.1 shows the pruning times of the NewsML workload and Figure 6.2 that of the treebank workload. With the NewsML workload the pruning time varied between 2.17 and 2.63 seconds, and with the treebank workload between 10.43 and 23.33 seconds. Pruning complex treebank filters is expensive, but some performance gain can be achieved by precomputing the $substitutes[a, b]$ entries (Algorithm 5.5) for each combination of a, b and *max-substitutes*.

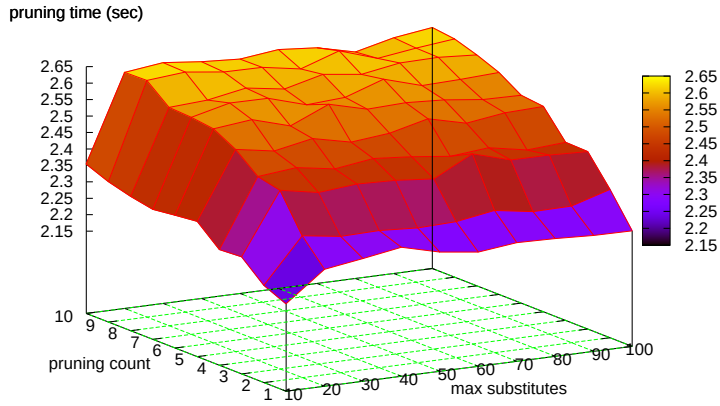


Figure 6.1. Pruning time of a filter workload of 10 000 distinct filters in relation to the *max-substitutes* and *pruning-count* parameters. The filter workload was generated from the NewsML DTD with $prob(*)$ = $prob(//)$ = 0.2.

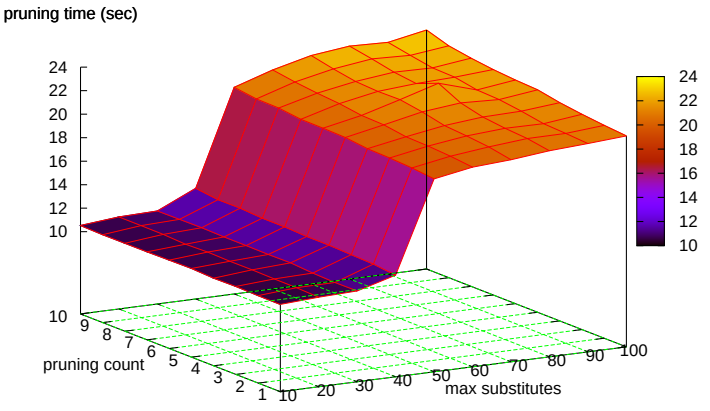


Figure 6.2. Pruning time of a filter workload of 10 000 distinct filters in relation to the *max-substitutes* and *pruning-count* parameters. The filter workload was generated from the treebank DTD with $prob(*) = prob(//) = 0.2$.

6.2 Performance Gain in Filtering

6.2.1 Data Sets Having a Simple DTD

We studied how filter pruning affects the filtering speed of our PMA-based algorithms, YFilter [24], and the lazy-DFA [28]. In our first experiments we used two sets of workloads, one generated from the protein DTD and the other from the NASA DTD. The speed of filtering was measured using as input documents the entire 24 MB NASA data set and a 24 MB extract from the protein-sequence data set.

Figures 6.3 and 6.4 show the performance gain achieved by pruning the filter workloads. The original workloads of distinct linear XPath filters without predicates were generated with $prob(*) = prob(//) = 0.2$. With the protein data set the number of filters varied from 2 000 to 20 000 and with the NASA data set from 10 000 to 100 000. With the protein data set 90–97 % and with the NASA data set 52–65 % of the filters matched the input document. It can be seen that with the protein data set the lazy DFA gains most from pruning. In that case the performance increases by a factor of 102 with a workload of 20 000 filters. With the same workload the performance increase was by a factor of 30 with the dynamic PMA, 9 with the PMA FB, and 78 with YFilter. Measurements with the NASA

data set produce similar results. The performance increase of the lazy DFA is by a factor of 155 with a workload of 100 000 distinct filters and that of YFilter by a factor of 134.

Figures 6.5 and 6.6 show the absolute filtering times of the algorithms with the pruned workloads. With the protein data set all wildcards and descendant operators were eliminated by pruning, but with the NASA data set some descendant operators remain in workloads because of the cycle in the DTD. It can be seen that the lazy DFA is the most efficient algorithm for processing pruned workloads in the case of these data sets having tree-like DTDs. With the protein data set the filtering speed of PMA FB is worse than that of the other PMA-based algorithms. Because the protein workloads do not contain wildcards or descendant operators, the bare AC algorithm can be used. However, in Figure 6.5 it can be seen that the performances of the bare AC and the dynamic PMA are more or less the same. In this case the dynamic PMA behaves very similar to the bare AC. With the NASA data set, however, PMA FB clearly outperforms the dynamic PMA.

Figures 6.7 and 6.8 show the performance gain achieved by pruning with respect to $prob(*)$, when $prob(//)$ has been set to 0.2, the number of distinct filters to 4 000, and the maximum depth of the filters to the depth of the corresponding data set. In the figures it can be seen that with the YFilter and lazy DFA algorithms the performance gain of pruning increases as the number of wildcards increases. However, the PMA-based filtering algorithms seem not to gain more from pruning when $prob(*)$ increases. Figures 6.9 and 6.10 show that the filtering speeds of the algorithms with the pruning optimization are nearly constant when $prob(*)$ varies.

We repeated the previous experiment by fixing $prob(*)$=0.2 and varying $prob(//)$. Figures 6.11 and 6.12 show the performance gain achieved by pruning and Figures 6.13 and 6.14 the filtering speeds with the pruned workloads. It can be seen that with the protein data set YFilter and the dynamic PMA benefit from pruning more as the number of descendant operators increases. For the PMA FB and lazy DFA algorithms the performance gains are more or less the same regardless of $prob(//)$. Measurements with the NASA data set indicate that the performance gains of YFilter and the lazy DFA increase with $prob(//)$, but somewhat decrease with the PMA-based algorithms. With both data sets the lazy DFA is the most efficient algorithm. With the protein data set pruning is able to remove all descendant operators and the filtering speed with the pruned workloads is nearly constant for all algorithms. However, with the NASA data set some descendant operators remain in the filters because of the

cyclic DTD, and in this case the filtering speed of the PMA-based algorithms decreases as the number of filters increase. With the NASA data set the filtering speeds of YFilter and the lazy DFA are not sensitive to the number of descendant operators, when the filter workloads have been pruned.

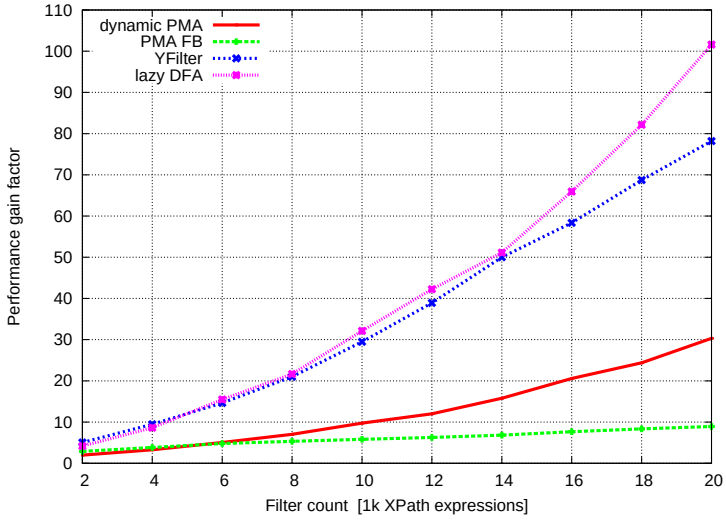


Figure 6.3. Performance gain of filter pruning with the protein-sequence data set. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

We also ran tests on pruned workloads in which the elimination of descendant operators was restricted to non-leading occurrences. We generated a workload of 10 000 distinct filters with $prob(*) = prob(//) = 0.2$ for the protein and NASA DTDs. For the protein data set the performance gain from eliminating also leading descendant operators was 20 % for the dynamic PMA, 47 % for PMA FB, 20 % for YFilter, and 31 % for the lazy DFA. For the NASA data set the performance gains were respectively 23 %, 40 %, 42 %, and 45 %. Thus all the algorithms gain from eliminating also the leading descendant operator.

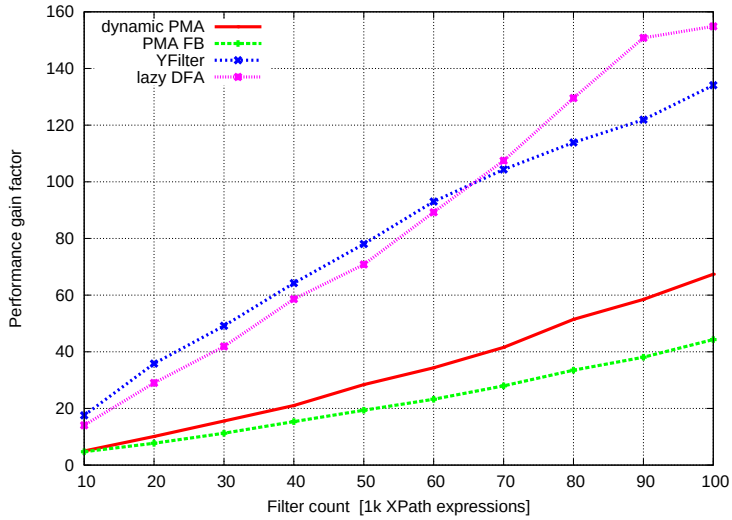


Figure 6.4. Performance gain of filter pruning with the NASA data set. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

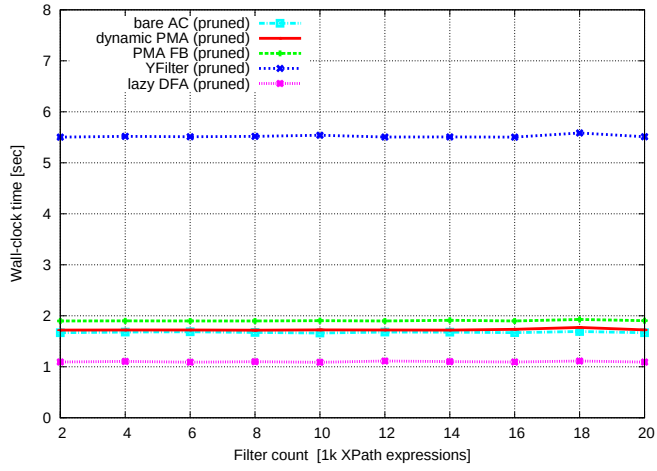


Figure 6.5. Filtering times of the 24 MB XML protein-sequence data set, using the bare AC, dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

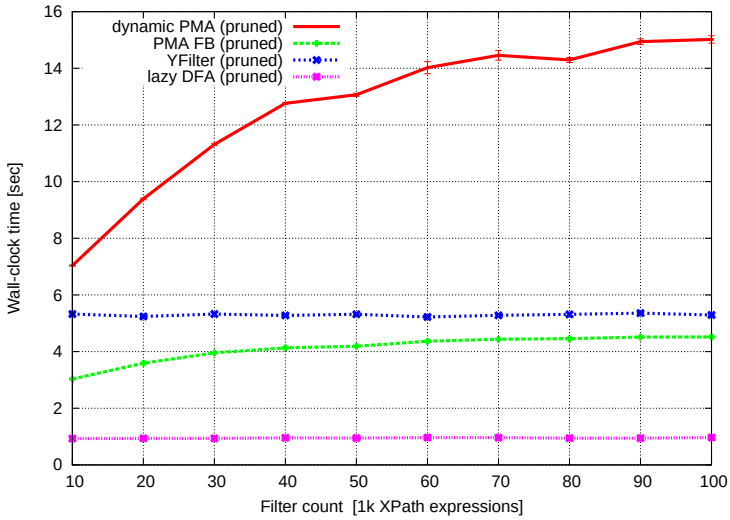


Figure 6.6. Filtering times of the 24 MB XML NASA data set, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

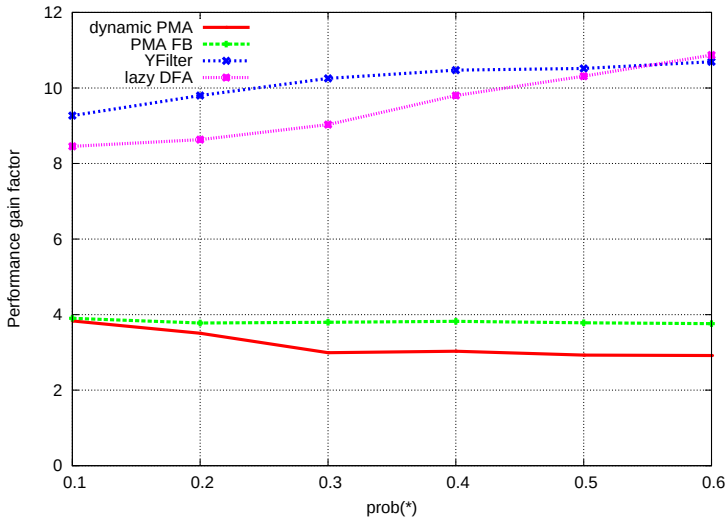


Figure 6.7. Performance gain of filter pruning with the protein-sequence data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

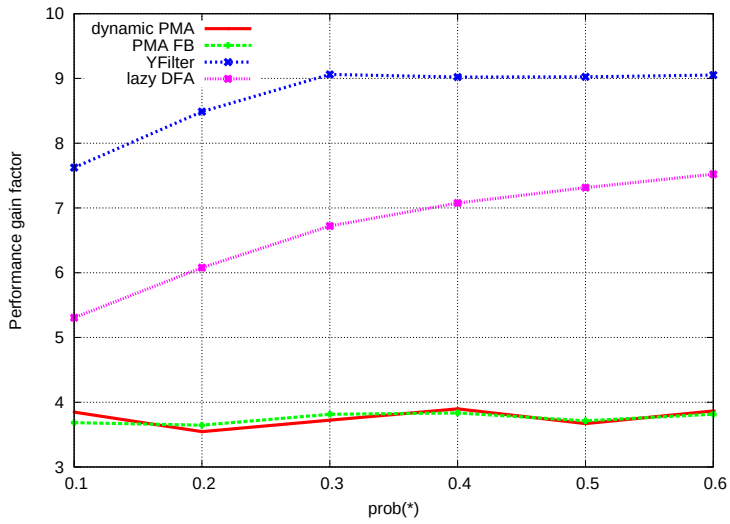


Figure 6.8. Performance gain of filter pruning with the NASA data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(//) = 0.2$.

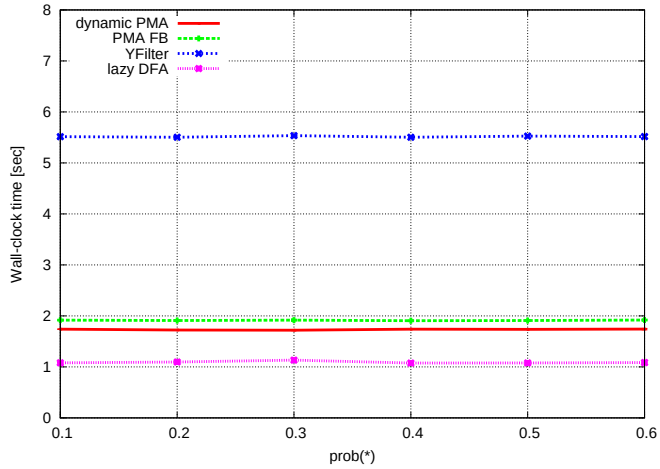


Figure 6.9. Filtering times of the 24 MB XML protein-sequence data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(//) = 0.2$.

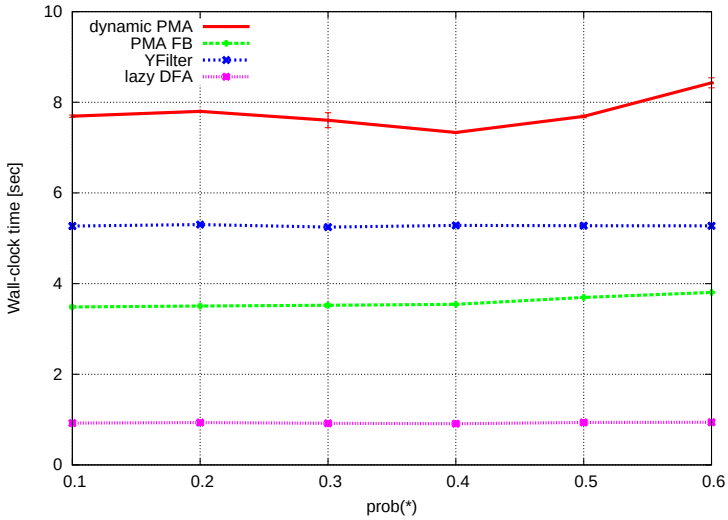


Figure 6.10. Filtering times of the 24 MB XML NASA data set with respect to $prob(*)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The distinct XPath workloads were generated with $prob(//) = 0.2$.

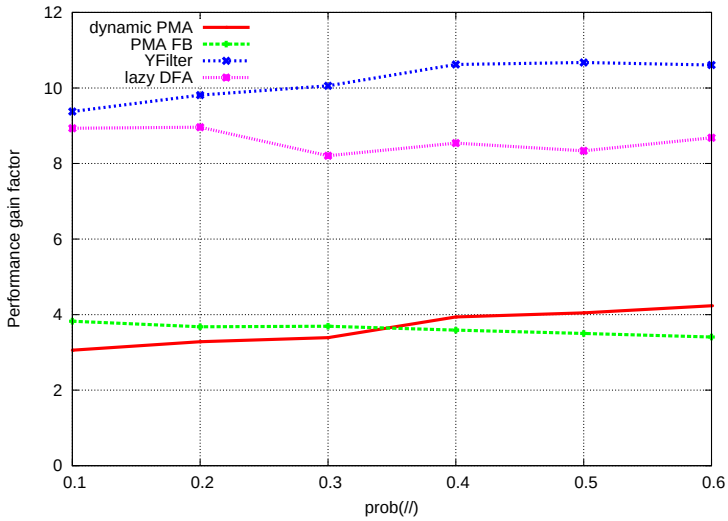


Figure 6.11. Performance gain of filter pruning with the protein-sequence data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

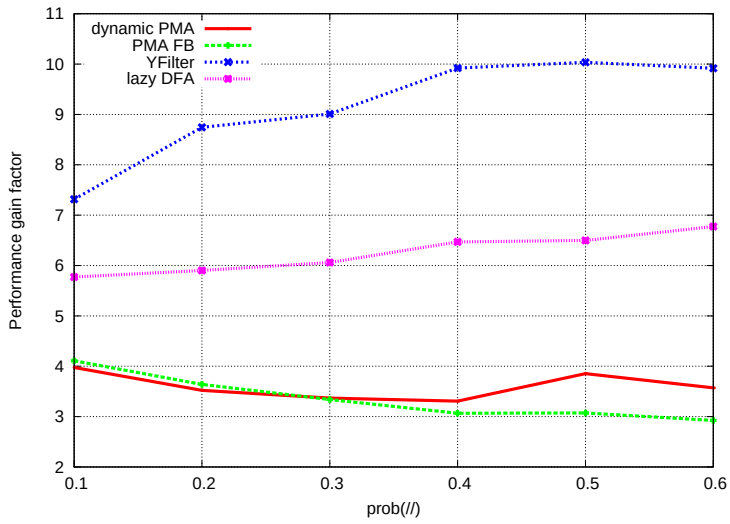


Figure 6.12. Performance gain of filter pruning with the NASA data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms. The 4 000 distinct XPath filters were generated with $prob(*) = 0.2$.

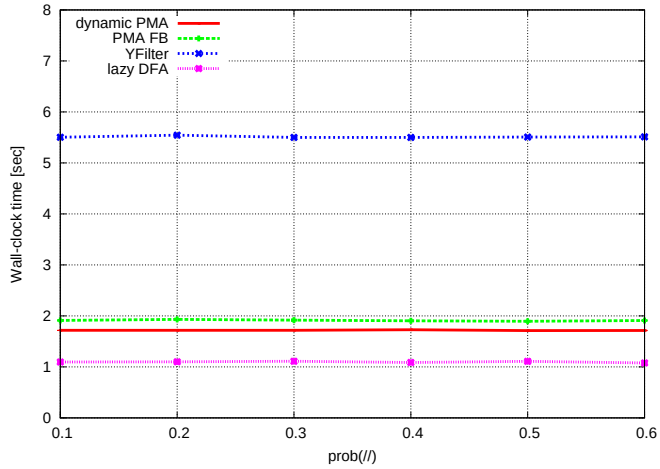


Figure 6.13. Filtering times of the 24 MB XML protein-sequence data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = 0.2$.

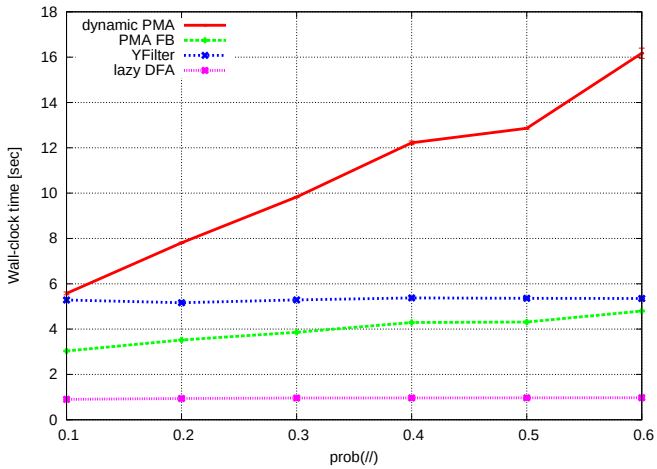


Figure 6.14. Filtering times of the 24 MB XML NASA data set with respect to $prob(//)$, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = 0.2$.

6.2.2 Data Sets Having a Complex DTD

The effect of filter pruning on the filtering performance was also studied with the complex and recursive NewsML and treebank data sets. We generated 10 000 distinct filters for the NewsML and treebank DTDs with $prob(//) = prob(*) = 0.2$. We measured the filtering time for the original workload and for the pruned workloads with *max-substitutes* values $10, 20, \ldots, 100$ and *pruning-count* values $1, 2, \ldots, 10$. For each measurement the results are averages of five test runs. The standard deviation was less than 5 % for each result. Table 6.2 shows the performance gains achieved by pruning the NewsML and treebank data sets and the configuration of *max-substitutes* and *pruning-count* that produced the best performance for each algorithm.

Original workload of 10 000 distinct NewsML filters				
	dynamic PMA	PMA FB	YFilter	lazy DFA
performance gain factor	3.05	2.38	3.06	1.96
pruning-count	7	9	10	10
max-substitutes	10	10	80	10
#distinct pruned filters	4 878	4 719	6 264	4 705
Original workload of 10 000 distinct treebank filters				
	dynamic PMA	PMA FB	YFilter	
performance gain factor	1.59	1.74	1.45	
pruning-count	3	1	2	
max-substitutes	10	10	50	
#distinct pruned filters	13 225	12 668	62 439	

Table 6.2. Performance gains achieved by pruning NewsML and treebank data sets. The original workloads were generated with $prob(*) = prob(//) = 0.2$.

With the NewsML data set each algorithm did benefit from fairly high values of *pruning-count*; the best performance was acquired with values nearly as great as the document depth. It was sufficient to set *max-substitutes*= 10 with the PMA-based algorithms and the lazy DFA, but YFilter gained from still higher values of *max-substitutes*.

As can be seen in Table 6.2, filter pruning increased the filtering performance also with the treebank data set. The lazy DFA is excluded from this experiment, because with the treebank data set its performance decreases quickly as the number of filters increases beyond a few thousand. The best performance gain was achieved when only a few first wildcards or descendant operators were pruned. The PMA-based algo-

rithms performed best when *max-substitutes* = 10, but with YFilter the best performance was achieved with a higher value of *max-substitutes* as with the NewsML data set.

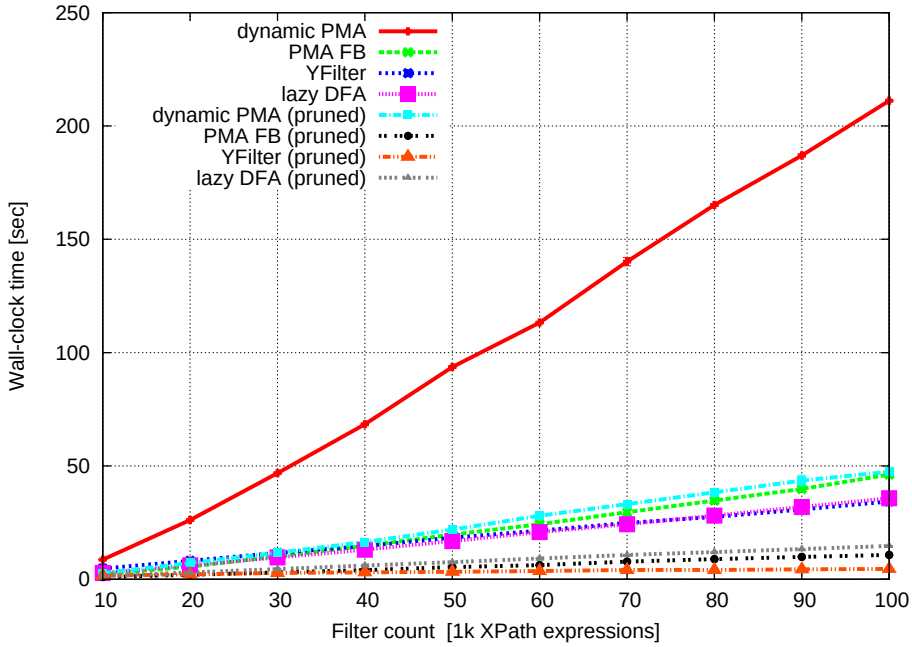


Figure 6.15. Filtering times of the 2.6 MB XML NewsML data set, using the dynamic PMA, PMA FB, YFilter, and lazy DFA algorithms with and without the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

Figure 6.15 shows the filtering time of the dynamic PMA, PMA FB, YFilter and the lazy DFA algorithms with NewsML data set. The workloads of 10 000 to 100 000 distinct XPath filters without predicates were generated with $prob(*) = prob(//) = 0.2$. The filter workloads for each algorithm were pruned with the optimal configuration of *max-substitutes* and *pruning-count* found in Table 6.2. It can be seen that without the filter pruning optimization YFilter and the lazy DFA are as efficient, but with the optimization YFilter performs better and is the most efficient algorithm for this data set. The lazy DFA is faster than PMA FB with the unpruned workloads, but PMA FB gains more from pruning and is better than the lazy DFA when filter pruning is used.

Figure 6.16 repeats the previous experiment with the treebank data set (the lazy DFA is excluded from this experiment). Even though the PMA-

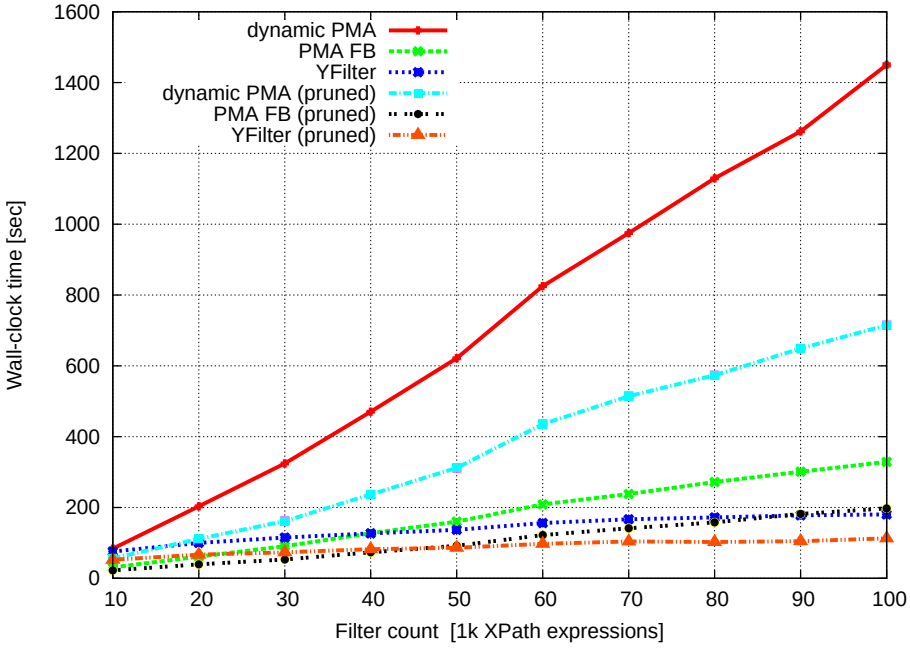


Figure 6.16. Filtering times of the 82.5 MB XML treebank data set, using the dynamic PMA, PMA FB, and YFilter algorithms with and without the filter pruning optimization. The workloads of distinct XPath filters were generated with $prob(*) = prob(//) = 0.2$.

based algorithms gain more from filter pruning than YFilter (Table 6.2), YFilter is still the most efficient algorithm with the treebank data set, when filter pruning is used.

6.3 Summary

The performance increase of filter pruning with data sets having a tree-like DTD is by order of magnitude; NFA-based YFilter becomes even 130 times more efficient and DFA-based lazy DFA 150 times more efficient. Also filter pruning increases the performance of our PMA-based algorithms significantly; by a factor of 10–70.

One reason for the performance increase is that filter pruning can reduce the number of distinct filters in the filter workload. With the simple protein and NASA data sets filter pruning reduces the numbers of distinct filters to 1 %–5 % of the original workload, and with the moderately complex NewsML data set to 47 % of the original workload. Clearly, all filtering algorithms perform matching only for workloads consisting of distinct filters.

However, with the most complex treebank data set the filtering performance of PMA-based algorithms and YFilter increased even though the number of distinct filters increased. This indicates that the method of filter pruning contributes to path sharing, an important optimization aspect used in automata-based filtering. YFilter applies path sharing in building the NFA, and our PMA-based algorithms apply path sharing in recognizing path fragments in filters that are separated by wildcards or descendant operators.

CHAPTER 7

General XPath Filters

The previous chapters examine evaluation of linear XPath filters without predicates. In this chapter the evaluation of more general XPath filters is discussed. Section 7.1 presents different methods for processing twig filters and Section 7.2 methods for predicate evaluation.

7.1 Nested XPath Filters

Nested XPath filters, or *twig filters*, are described by the following grammar:

```
P := /E | //E | PP
E := label | * | E[P]
```

where `label` denotes an XML-element label. An example of such a filter is $/a[d]/b[e/f]/c$ that matches, say, XML document `<a><d></d><b><e><f></f></e><c></c></b></a>`. Figure 7.1 illustrates the query tree of the filter. The main path of the expression is $/a/b/c$ and the nested subpaths d and e/f are surrounded by brackets ([,]) in the XPath notation. In this section we consider twig filters having only one level of nesting.

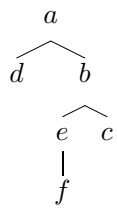


Figure 7.1. Query tree of a twig XPath expression.

This section presents two different methods for matching twig filters; evaluation as post-processing and holistic evaluation. Both methods can

be used to implement matching of twig filters with our PMA-based algorithms described in Chapter 3.

7.1.1 Evaluation as Post-Processing: YFilter

The evaluation of twig filters in YFilter [24] is done as a post-processing step. The filter workload is preprocessed and twig filters are decomposed into linear filters. These linear filters are then matched using the filtering NFA of YFilter. In processing the XML document, the parser gives a unique id for each path in the XML document tree. Whenever a linear filter matches a path, the path is stored for that filter. When the whole XML document has been processed, the matching twig filters are calculated from the collected information.

For example, the twig filter $Q = /a[d]//b[e/f]/c$ is decomposed into linear filters $L_1 = /a//b/c$, $L_2 = /a/d$, and $L_3 = /a//b/e/f$. Filter L_1 denotes the main path and L_2 and L_3 the nested subpaths. Figure 7.2 shows an example XML document, where the elements have been numbered in preorder. During the processing of the input document, the matching paths for each linear filter are collected: for filter L_1 the path $a_1b_4c_7$, for filter L_2 the paths a_1d_2 and a_1d_3, and for filter L_3 the path $a_1b_4e_5f_6$. It can be seen that all the linear filters match under the same `a` element (id 1) and filters L_1 and L_3 under the same `b` element (id 4). Thus the twig filter matches the input document. This calculation is done as a post-processing step after the whole XML document has been processed. A downside of this method is that if the XML document is very large, then collecting all matched paths can consume a large amount of memory.

7.1.2 Holistic Evaluation

Holistic evaluation of twig filters means that the filters are evaluated during processing of the XML input document. With this method there is no need to collect a large amount of information into memory while processing of the possibly large input document.

FiST [38] and its successor iFiST [40] are sequence-based algorithms that process ordered twig patterns holistically. *Ordered matching* means that the nested paths in the XML document must be in the same order as the nested paths in the XPath twig filter in order for the filter to match. For example, the filter $/a[b]/c$ matches XML document `<a><b></b><c></c></a>`, but not document `<a><c></c><b></b></a>`, when ordered matching of twig filters is applied.

The difference between iFiST and FiST is that iFiST merges common

segments in the filters thus applying a form of path sharing, while FiST does not use path sharing.

FiST and iFiST build Prüfer codes [56] of the XML documents while SAX parsing the documents. The algorithms encode the XML document into an extended form of Prüfer sequence, namely Labeled Prüfer Sequence (LPS). The Prüfer sequence of a tree is computed by first enumerating the tree by using some tree-traversal method; in the case of XML documents it is convenient to use preorder traversal. Then the leaf node v with smallest number is selected and the number of v's father is added into the code. Node v is deleted and the algorithm continues from the leaf node with smallest number. The original algorithm by Prüfer continues until there are two nodes left, thus the size of the Prüfer sequence of a tree with n nodes is $n-2$. However, FiST and iFiST continue until there is only one node left in the tree, producing Prüfer sequence of size $n-1$. A labeled encoding contains the values of the nodes instead of their numbers. With LPS encoding, the leaf nodes of the document tree are extended with dummy child nodes. As a result the LPS encoding also contains the node labels of the leaf nodes of the original tree. Figure 7.2 shows the example XML document and corresponding Prüfer and LPS sequences.

The XPath filters are also encoded into LPS form. The encoded filters are then stored into an index structure. Then during SAX processing of the XML input stream the profiles are searched from the LPS encoded stream by using a subsequence matching algorithm. Subsequence matching means that an occurrence of a pattern is located from the text, so that the letters of the pattern occur in the same order in the text, but not necessarily consecutively. Those LPS encodings of filters are located that are subsequences of the sequence encoding of the document. If a filter encoding is not a subsequence of the document encoding, then the filter is not a subtree of the original document tree. This way a set of candidate filters is identified and further checking is performed for this set of filters.

For the example filter $Q = /a[d]//b[e/f]/c$, the LPS sequence is $LPS(Q) =$ `dafebcba`. The LPS sequence of the document of Figure 7.2 is $LPS(T) =$ `dadafebccba`. It can be seen that $LPS(Q)$ is a subsequence of $LPS(T)$, thus the filter is a possible match for the document. The matching is performed bottom-up as with XPush (see Section 2.4.2) and BUFF (see Section 2.3.3).

BoXFilter [49] also translates filters and XML documents into Prüfer sequences (or into LPS form) and performs filtering by subsequence matching. BoXFilter groups filter sequences into structures called sequence en-

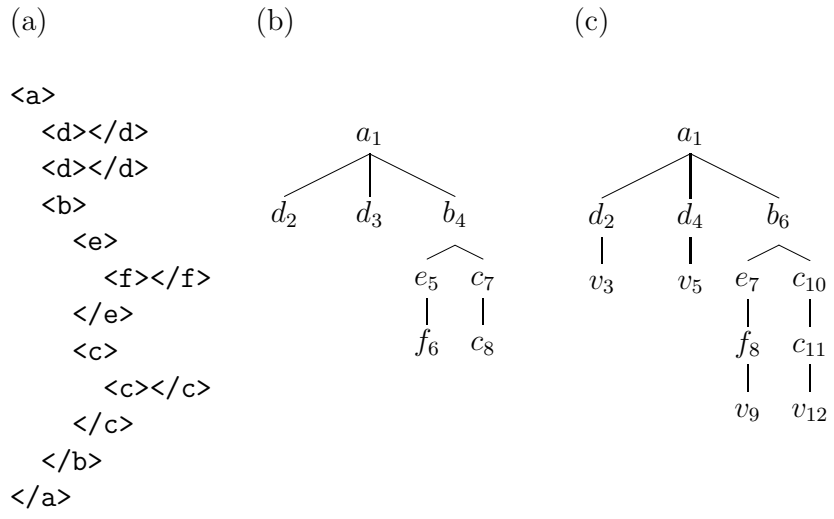


Figure 7.2. An example XML document (a) and its tree representation (b), where the nodes have been numbered in preorder. In tree (c) the leaf nodes of the original tree have been extended with dummy child nodes. The Prüfer code of the document tree (b) is $\{1, 1, 5, 4, 7, 4, 1\}$ and the corresponding labeled sequence $\{a, a, e, b, c, b, a\}$. The LPS sequence is calculated from tree (c), thus we get $\{d, a, d, a, f, e, b, c, c, b, a\}$.

velopes. This grouping makes pruning candidate filters more efficient. In the experiments [49] BoXFilter was found to be 80% more efficient than NFA-based BUFF. However, direct comparison with FiST or iFiST was not made.

7.1.3 Other Methods

XTrie [19] builds a trie-based index structure of the keywords occurring in XPath filters. The structure is similar to Aho–Corasick trie (see e.g. Section 3.1), but it does not have any fail arcs. The keyword index is searched in processing SAX events and matching filters are identified.

Bruno et al. [15] propose an algorithm called Index-Filter that indexes the XML stream by a classic inverted index data structure used in information retrieval [61]. The position of an element occurrence in the XML document is represented as pair $(L : R, D)$, where L is the position of the start-tag of the element, R the position of the corresponding end-tag, and D the nesting depth of the element. The example document in Figure 7.3 illustrates the indexing scheme. Structural relationships between nodes can easily be determined by using this scheme. Index-Filter requires that

the whole XML input document is read into memory.

```
<a(1:16,1)>
  <d(2:3,2)></d>
  <d(4,5,2></d>
  <b(6:15,2)>
    <e(7:10,3)>
      <f(8:9,4)></f>
    </e>
    <c(11:14,3)>
      <c(12,13,4)</c>
    </c>
  </b>
</a>
```

Figure 7.3. The indexing scheme of XML documents used with the Index-Filter algorithm [15].

7.1.4 Evaluation with the PMA-based Algorithms

With our PMA-based filtering algorithms described in Chapter 3, the method of evaluating twig filters as a post-processing task can be applied. Two modifications must be done into the SAX parser: (1) a counter for giving a unique identifier for each XML element and (2) a stack for holding the current path in the XML document need to be added. Then the algorithm of Section 3.5 can be modified so as to collect all matched paths for each linear filter. From this information the matched twig filters can be calculated as with YFilter.

Holistic evaluation could also be applied in the case of PMA-based filtering. We can modify the dynamic PMA algorithm of Section 3.3 to handle twig filters in an ordered fashion. The idea is to decompose a twig filter into linear filters as in Section 7.1.1, but these linear filters would then be matched holistically. A twig filter without wildcards and descendant operators is decomposed into keywords so that each root-to-leaf path of the query tree of the filter forms a keyword for the PMA. For example, with twig filter $Q_1 = /a[d]/b$ we would have

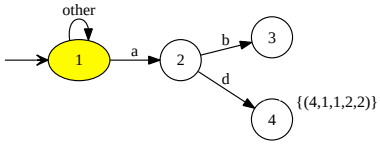
$$keyword(1,1) = ad,\ mingap(1,1) = maxgap(1,1) = 0,$$
$$keyword(1,2) = ab,\ mingap(1,2) = maxgap(1,2) = 0.$$

Again, the output sets of the PMA contain tuples of the form (q, i, j, b, e). The input document is processed in way similar to the algorithm of Section 3.3, but the method of backtracking is different. Now the backtracking stack is an array that may grow up to document depth. An entry for the array holds the previous state and a linked list of output tuples inserted into and deleted from the current output. The memory for the array is allocated dynamically. When a modification to the output sets is done, information of the modification is not necessarily stored into the list on top of the stack, but in some cases into the list that is somewhere in the middle of the stack. When an element end-tag is processed, the operations stored into the list that is on top of the stack are reversed, the state of the automaton is restored, and the stack is popped.

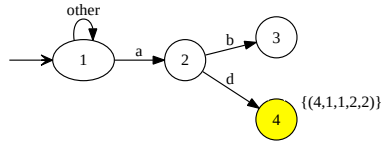
Figure 7.4 shows how an example document fragment `<a><d></d></a> <a><d></d><b></b></a>` can be processed with the dynamic PMA that is modified so as to handle twig filters. The PMA has been constructed from the above-mentioned filter Q_1. After the initialization of the PMA, the output set of state 4 contains the tuple $(4, 1, 1, 2, 2)$ representing keyword *ad*. The processing starts from the initial state (Figure 7.4(a)). After processing the start-tags of elements `a` and `d` the automaton has entered state 4 (Figure 7.4(b)). The visited states are stored onto the stack. Now the tuple $(4, 1, 1, 2, 2)$ can be removed from the output set of state 4 and the tuple $(3, 1, 2, 2, 2)$ inserted into the output set of state 3 (Figure 7.4(c)). Modifications to output sets are stored into the stack, not top of the stack but as part of the element at position 1. When we read the end-tag of element `d` we backtrack to the state 2 that was found on top of the stack, and pop the stack (Figure 7.4(d)). Next we read the end-tag of element `a`, and backtrack to state 1 (Figure 7.4(e)). The modifications made to the output sets are undone at this point (Figure 7.4(f)). As we read the start-tags of elements `a` and `d` the automaton enters state 4 again (Figure 7.4(g)). The output sets are again updated dynamically (Figure 7.4(h)). As we read the end-tag of element `d` the automaton backtracks to state 2 (Figure 7.4(i)). As we read the start-tag of element `b`, the automaton enters state 3, and we find that the filter has matched, concluding this example (Figure 7.4(j)).

The example illustrates how a twig filter can be processed with a PMA-based algorithm in an ordered fashion as with the FiST algorithms. Unordered matching is possible only if all order combinations of the twig filter are handled. For example, to find a match for filter $/a[d]/b$ we need to process filter $/a[b]/d$ as well.

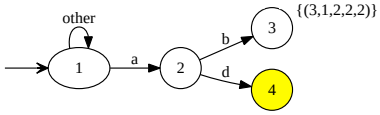
Handling wildcard and descendant operators in twig filters with the above-described method complicates the processing somewhat. For ex-



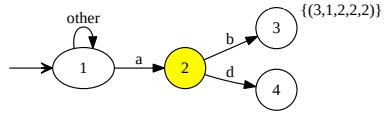
(a) Processing of the XML document starts from the initial state.



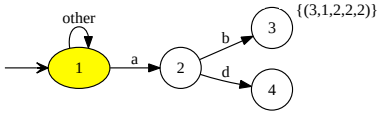
(b) Processed start of elements `a` and `d`. $S = \langle\{1\},\{2\}\rangle$.



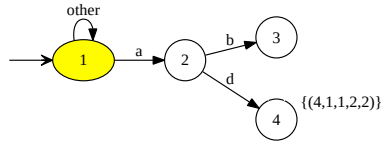
(c) Updated output sets of states 3 and 4. $S = \langle\{i(3,1,1,2,2), d(4,1,2,2,2), 1\},\{2\}\rangle$.



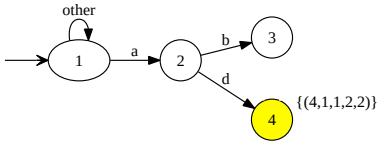
(d) Processed end of element `d`. $S = \langle\{\{i(3,1,1,2,2), d(4,1,2,2,2), 1\}\}\rangle$.



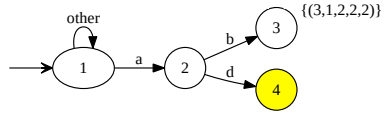
(e) Processed end of element `a`. $S = \langle\{\{i(3,1,1,2,2), d(4,1,2,2,2)\}\}\rangle$.



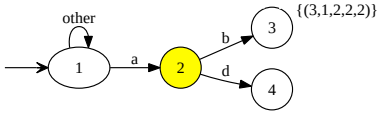
(f) Updated output set of states 3 and 4. $S = \langle\rangle$.



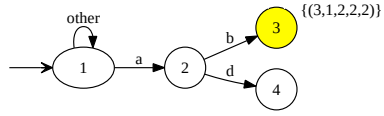
(g) Processed start of elements `a` and `d`. $S = \langle\{1\},\{2\}\rangle$.



(h) Updated output sets of states 3 and 4. $S = \langle\{i(3,1,1,2,2), d(4,1,2,2,2), 1\},\{2\}\rangle$.



(i) Processed end of element `d`. $S = \langle\{i(3,1,1,2,2), d(4,1,2,2,2), 1\}\rangle$.



(j) Processed start of element `b`. The filter has matched. $S = \langle\{i(3,1,1,2,2), d(4,1,2,2,2), 1\},\{2\}\rangle$.

Figure 7.4. Ordered matching of XML document fragment `<a><d></d></a><a><d></d><b></b></a>` with the dynamic PMA that is modified so as to handle twig filters. The filter workload consists of filter $Q_1 = /a[d]/b$. As before, S denotes the contents of the stack.

ample, to find an ordered match for filter the $/a[.//b]/c$, the filter would need to be decomposed into keywords a, b and ac. When processing an input document, after finding a the tuple representing a is deleted and the tuple for b is inserted into the output set. After locating element b occurring somewhere under a, the tuple representing b can be deleted and the tuple for ac inserted into the output set. These modifications to output sets are backtracked if a has not c as a child and the end of element a is encountered. Thus the backtracking mechanism would need to be modified so as to take into account the properties of each keyword: whether or not the keyword is a part of a branch of a twig filter.

7.1.5 Pruning Nested XPath Filters

The filter pruning method can also be used to eliminate wildcards and descendant operators from twig filters. For example, pruning the filter $/a[*/e]/b$ with the DTD of Figure 5.1 results in the pruned filter $/a[d/e]/b$.

SFilter [41, 42] uses the DTD to optimize twig filters also in other ways. If an XML element has a child constraint, i.e., element a must have a child b, then a twig filter $/a[b]/c$ can be simplified to $/a/c$. In a similar way information of a descendant constraint can be used to optimize certain filters. For example, a filter $/a[.//f]/c$ can be simplified to $/a/c$, if there is a constraint stating that element a must have f as a descendant. If it is known that elements i and j are always siblings, this knowledge can be used to simplify filter $//f[i]/j$ into $//f/j$.

7.2 Predicate Evaluation

This section presents different methods for the evaluation of *value-based predicates*. Predicates are used to test values of attributes or text data of XML elements. The predicate part of a filter $/a[text() = 'C1']$ is $text() = 'C1'$ and the filter matches, say, the XML document `<a>C1</a>`. The predicate part of a filter $/a[@attr = 'C2']$ is $@attr = 'C2'$ and the filter matches, say, the XML document `<a attr='C2'></a>`. More complex predicates can contain relational operators, conjunction, disjunction or negation. Examples of filters with such predicates include $Q_1 = //a[b/text() = 1 \text{ and } .//a[@c > 2]]$ and $Q_2 = //a[@c > 2 \text{ and } b/text() = 1]$.

7.2.1 Inline and Selection Postponed: YFilter

With YFilter [24] two alternative ways to evaluate predicates are presented: *inline* and *selection postponed*.

The inline method evaluates predicates at the time when corresponding states in the NFA are reached. For example, with filter Q_1 = $/a/b[a =' S_1']/c$, the predicate would be evaluated when state 2 is reached in the NFA of Figure 7.5. For each filter, bookkeeping information must be maintained of predicates evaluated to true. When a final state of the NFA is reached (e.g., state 3 in Figure 7.5), the predicate bookkeeping information for filters whose structure part matches at that state is checked and matching filters are identified. With this method the backtracking mechanism must undo changes made to predicate bookkeeping for each filter.

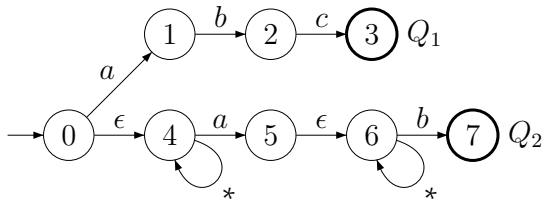


Figure 7.5. YFilter's NFA for filters Q_1 = $/a/b[a =' S_1']/c$ and Q_2 = $//a[@a =' S_2']//b$.

The other method, selection postponed, evaluates all the predicates of a filter after the structure part of the whole filter has been matched. In this case the predicate of filter Q_1 would be evaluated when the automaton enters state 3. In order to evaluate the correct predicates for each filter, the visited states of the NFA are stored for each XML path as well as the attribute and element values at those states. From each final state we can traverse backwards to find the states visited that lead to the final state. For example, with filter Q_2 = $//a[@a =' S_2']//b$ leading to state 7 in Figure 7.5 and XML document `<a a='S1'><a a='S2'><b></b></a></a>`, the sequences of visited states for the path `aab` are $\langle 0, 4, 5, 7\rangle$ and $\langle 0, 5, 6, 7\rangle$. At final state 7 both state sequences must be taken into account when evaluating the predicate. The attribute a had value S_1 in state 4 and value S_2 in state 5. With the latter sequence the predicate evaluates to true and the filter matches the document.

7.2.2 Automata-based Evaluation: lazy DFA and PFilter

As explained in Section 2.4, the lazy DFA algorithm builds a combined NFA from the linear XPath filters, and from the NFA it constructs a DFA lazily during processing of the XML input document. The lazy DFA can process linear XPath filters having value-based predicates of the form *text*() ='S', where S is a string constant without any wildcards. The evaluation of an XPath filter ending with such a predicate is accomplished in the lazy DFA as state transitions in the automaton. A sink state reachable on the symbol $text(S)$ is added to the NFA from which the DFA is constructed, where $text(S)$ calculates a unique symbol for the string constant S. If an XPath filter contains nested predicates (predicates appearing in the middle of the filter), the filter is decomposed into linear XPath filters. For example, filter Q_1 = /a[*text*() ='S']/b is decomposed into filters Q_2 = /a[*text*() ='S'] and Q_3 = /a/b. Filters Q_2 and Q_3 are then matched by using the lazy DFA's matching engine, and the result (if Q_1 matches or not) is calculated from these results. This method is similar to the way YFilter evaluates twig filters (see Section 7.1.1). The method has the drawback that it considerably reduces path sharing of filters.

Byun et al. [17] propose an automata-based algorithm called PFilter that is based on YFilter. The structure matching is done by a combined NFA as in YFilter, but the method for predicate evaluation is different. An Aho–Corasick [5] PMA is constructed from the string constants appearing in the value-based predicates. Evaluation of predicates is performed by the PMA; when the SAX parser encounters characters in the XML data, the characters are fed one by one to the PMA, which calculates the satisfied predicates. A filter matches if the structure part produces a match and all the predicates of the filter evaluate to true. Since XPath filters can have same string constants in the predicates, this method shares processing of predicates. The idea of shared processing of value-based predicates is the same as with the XPush algorithm, but PFilter applies path-sharing also in processing the structure parts of the filters.

PFilter supports only the equality operator (=) in predicates and no logical operators. However, PFilter introduces an extension to the XPath language, namely an % operator in the operand string of the XPath text() function. The operator % has a meaning similar to the LIKE operator in SQL [7]. For example, the filter /a[*text*() ='%bc%'] matches, say, XML document `<a>abcd</a>`. Thus the operator % can be used to implement XPath's *starts-with*, *ends-with* and *contains* functions.

7.2.3 Pushdown-automata-based Evaluation

The XPush algorithm [30] uses a pushdown automaton (PDA) in evaluating predicates. With XPush, predicates in XPath filters can be combined with *and*, *or* and *not*, and can be interleaved arbitrarily with the navigation. Supported operators are: $=$, $<$, $\leq$, $>$, $\geq$, and $\neq$. Examples of such XPath filters are: Q_1 = $//a[b/text() = 1$ *and* $.//a[@c > 2]]$ and Q_2 = $//a[@c > 2$ *and* $b/text() = 1]$. When processing these filters, XPush will evaluate predicates $@c > 2$ and $b/text() = 1$ only once.

The XPush machine is constructed by first compiling the XPath expressions into Alternating Finite Automata (AFAs). The AFAs are then compiled into a single XPush machine. An AFA is a nondeterministic finite automaton, where each state is labeled with AND, OR or NOT. Figure 7.6 shows the AFAs constructed from the example queries Q_1 and Q_2.

The XPush machine is constructed lazily by traversing the AFA states starting from the bottom of the query tree. For example, when the bottom-up SAX parser encounters value 1 during parsing of the document, the PDA state $q_0 = \{4, 13\}$ is constructed. Next, if XML element `b` is encountered, the state transition $t(q_0, b)$ is set by traversing AFA's edges backwards from q_0. Those states are selected that lead to AFA's states 4 and 13 with transition b, thus $t(q_0, b) = \{3, 12\} = q_1$.

7.2.4 Evaluation with the PMA-based Algorithms

In this section we discuss how value-based predicates can be evaluated with the filtering PMAs described in Sections 3.3 and 3.4.

The PMA-based filtering algorithms can be modified to use YFilter's selection-postponed method as follows. The SAX parser will need to be modified so as to store the XML element values for the current path. The values can be stored into a stack. An entry in the stack contains the XML attribute and element values at that height in the current path. Attribute values can be stored into a hash table accessed by attribute names. The predicates of a filter are evaluated when the structure part of the filter has matched and the possibly following `characters` event is encountered. For example, for the XML document

```
<a e='1'><b e='1' f='2'><c>test</c></b></a>
```

the parser produces the following events:

```
startDocument()
startElement(a(e='1'))
```

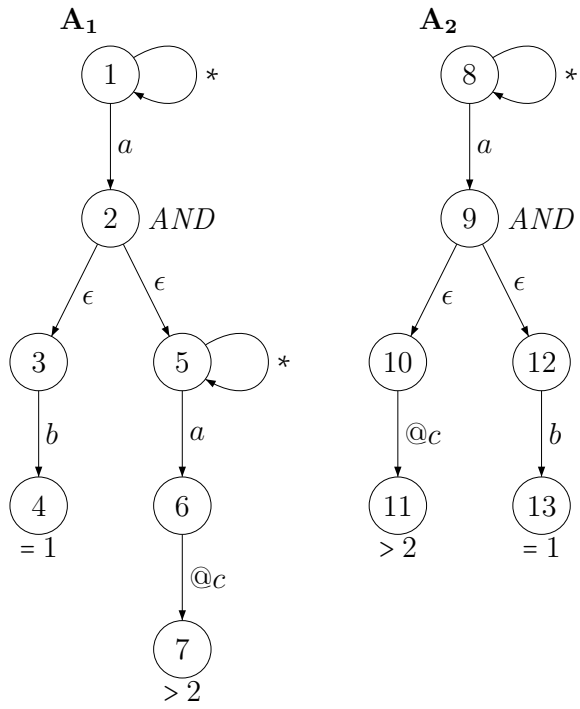


Figure 7.6. AFAs constructed from filters Q_1 = //a[b/*text*() = 1 *and* .//a[@c > 2]] and Q_2 = //a[@c > 2 *and* b/*text*() = 1] [30]. States 2 and 9 are AND states, and all other states are OR states.

```
startElement(b(e='1', f='2'))
startElement(c)
characters('test')
endElement(c)
endElement(b)
endElement(a)
endDocument()
```

When processing an XPath filter Q_1 = /a[@e ='1']/b/c[*text*() ='*test*'], after the structure part /a/b/c of the filter has matched and the following *characters* event has been processed, the contents of the stack at height 3 would be $S = \langle(e = \text{'1'}), (e = \text{'1'}, f = \text{'2'}), (\text{'}test\text{'})\rangle$. Now the predicates appearing in the filter would be evaluated against the contents of the stack. The value for the predicate @e ='1' will be checked from first entry of the stack and the value for the predicate *text*() ='*test*' from third entry of the stack. Both predicates evaluate to true, thus the filter matches the document.

The case when a filter has an ancestor-descendant relationship can be handled as follows. For each keyword in the filters, we store the depth at which the keyword has matched. The depth of the first element in the XML document is 1. For example, with the above XML document and an XPath filter Q_2 = //b[@e ='1']//c, the keyword b is found at depth 2 and keyword c at depth 3. We set $depth[2,1] = 2$ and $depth[2,2] = 3$. When the structure part //b//c has matched and we are evaluating the predicate, the value of the attribute `e` can be looked from the stack at entry $depth[2,1]$ corresponding to the keyword b, thus from the second entry of the stack. The predicate evaluates to true and the filter matches the document.

The evaluation of predicates can also be implemented with a method similar to YFilter's inline method. In this case the predicates pertaining to a keyword would be evaluated as soon as the corresponding keyword has been found. With the above filter Q_2, the predicate @e ='1' pertaining to keyword b would be evaluated when the start-tag of element `b` of the above XML document is processed. The output tuple of keyword c would be inserted into the current output only if the predicate pertaining to keyword b is evaluated to true. In general, when processing an output tuple of the jth keyword of a filter in the *traverse-output-path* procedure, the output tuple of the $(j+1)$th keyword is inserted into the current output only if all the predicates pertaining to the jth keyword are evaluated to true.

7.2.5 Filter Pruning with Filters Having Predicates

We tested the effect of filter pruning with workloads of linear XPath filters having value-based predicates of the forms *text()=s* and *@attr=s*, where s is a string constant (not containing any wildcards). We used a modified version of the YFilter's XPath filter generator [24] to generate the filter workloads. We modified the generator to produce string constants s appearing in the XML data set instead of randomly generated strings. This is a more realistic scenario and increases the probability of a filter having a value-based predicate to match the XML data set.

We experimented with YFilter [24] using the selection-postponed method for predicate evaluation. We generated workloads of 10 000 filters having one predicate per filter (generated with $prob(*) = prob(//) = 0.2$). Our experiments show that the performance gain from filter pruning is still evident: the speed-up of YFilter was 1.3 for the protein-sequence data set and 2.0 for the NASA data set. The speed-up is not so impressive as with filters without predicates, because much of the total filtering time is

spent on evaluating the predicates.

In our experiments with the lazy DFA [28], we observed that filter pruning impairs the performance when predicates are present: the filtering time for the protein data set almost doubled. This phenomenon might be explained by the fact that as the string constants s in predicates of different filters are usually different, we may no longer merge two filters even if they have an identical structure. When 10 000 distinct original filters with predicates were pruned, the number of distinct pruned filters almost tripled for the protein data set and doubled for the NASA data set.

The results obtained with the lazy DFA algorithm for filters with predicates are in contrast to the results we obtained with YFilter: pruning increased the performance of YFilter also in the case of filters having value-based predicates. YFilter accepts more general predicates than the lazy DFA algorithm and predicate evaluation is not part of the structure matching of the XPath filters: the evaluation of predicates is postponed until an accepting state of the NFA is reached during structure matching. As explained by Diao et al. [24], in this way the principle of path sharing of XPath filters is not destroyed.

CHAPTER 8

Conclusions

We have investigated the filtering problem of XML documents when the filters are expressed as XPath expressions. Several approaches to XML filtering use a finite automaton as a basis of the filtering algorithm, such as the widely-accepted YFilter [24] and lazy DFA [28] algorithms. Our novel solution for XML filtering is also based on a finite automaton: we build an Aho–Corasick [5] pattern-matching automaton (PMA) from the set of XPath filters. Our PMA-based algorithms have a small memory footprint and their performance is in many cases better than that of the YFilter and lazy DFA algorithms.

The bare AC algorithm presented in Section 3.1 is a simple and efficient PMA-based algorithm for processing linear XPath filters without wildcards and non-leading descendant operators. The set of keywords for the PMA is derived from the XPath filters by removing all child operators "/" from the filters and defining the keywords of the filters to be the strings consisting of XML elements only. The bare AC differs from the classic Aho–Corasick PMA in that it has a backtracking facility needed for processing tree-structured text. Also the construction of the fail function is different; the output sets are not completed while constructing the fail function. This means that the collection of the output sets is of linear size instead of quadratic size as with the classic Aho–Corasick.

Our experiments with the bare AC algorithm presented in Chapter 4 show nearly constant throughput of filtering regardless of the number of filters. This is consistent with our analytical results presented in Theorem 3.1. The bare AC also shows better performance than YFilter and the lazy DFA: with 100 000 distinct filters, the bare AC is 2.5 times more efficient than YFilter and 1.7 times more efficient than the lazy DFA.

In Section 3.2 we presented the static PMA algorithm, which is an extension of the bare AC. The static PMA can process linear XPath filters that have descendant and wildcard operators.

However, in our research we found out that a more efficient version of a PMA-based filtering algorithm exists. With this dynamic PMA algorithm we make the output function change dynamically during the processing of the input document. The basic idea of the 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. The modifications of the output sets performed when processing an element start-tag are recorded into a backtracking stack and reversed when the corresponding end-tag is encountered. We presented the dynamic PMA in Section 3.3. The analytical time bound of the dynamic PMA is better than that of the static PMA, and in our experiments the dynamic PMA was also found to be significantly more efficient.

In Section 3.4 we presented an optimized version of the dynamic PMA algorithm that has a different organization for the data structure for storing the output sets. The idea of this fast backtracking optimization (PMA FB) is that backtracking modifications to the output sets can be done in a single step. The time bound of PMA FB (Theorem 3.4) is worse than the time bound of the dynamic PMA (Theorem 3.3), when the XML data is very deep. However, in practice the paths in the XML documents are short and the document depth shallow; in our experiments PMA FB clearly outperformed dynamic PMA.

Our filtering experiments with linear XPath filters that have wildcards and descendant operators showed that the PMA FB algorithm performs well with both simple and complex XML data. With a non-recursive data set we measured the filtering speed of PMA FB to be 40 times faster than that of YFilter and 5 times faster than that of the lazy DFA. With a slightly recursive data set the PMA FB had the same performance than the lazy DFA and it was 3 times more efficient than YFilter. However, with a highly complex data set YFilter was found to be more scalable than PMA FB. The lazy DFA ran out of memory with this data set.

We also developed an optimization method called *filter pruning*. This method improves the performance of filtering by utilizing knowledge about a DTD to simplify the filters. The optimization algorithm takes as input a DTD and a set of linear XPath filters and produces a set of pruned linear XPath filters that contain as few wildcards and descendant operators as possible. The filter pruning method was inspired by the *query pruning* technique for optimizing regular path expressions with graph schemas [25] and by the article by Green et al. [28], in which the idea of using query pruning in XPath processing was suggested.

With a non-recursive data set and with a slightly recursive data set the filter-pruning method reduced the numbers of distinct filters to 1 %–5 %

of the original workload. When matching is performed only for the distinct filters in the workload, it is clear that the performance of any filtering algorithm increases. The increase in the filtering speed of the PMA-based algorithms was by a factor of 10–70. With the same data sets the filtering performance of YFilter and the lazy DFA increased even more than hundredfold.

In addition to these data sets, we ran experiments with complex and highly recursive data sets. By imposing some simple conditions stating when an operator may be eliminated a polynomial bound can be guaranteed on the total size of the pruned filters. Our experiments showed that filter pruning increased the filtering speed of PMA-based algorithms and YFilter also with these data sets. With PMA-based filtering the performance increase was by a factor of 1.6–2.4 and with YFilter by a factor of 1.5–3.1. With the most complex data set the filtering performance increased even though the number of distinct filters increased. This indicates that the filter-pruning method contributes to path sharing, an important optimization aspect used in automata-based filtering.

In Chapter 7 we discussed the evaluation of twig filters and filters with value-based predicates. As regards twig filters, we argue that the method presented by Diao et al. [24] can also be used with PMA-based filtering. Holistic evaluation [38, 40] of twig filters with PMA-based filtering algorithms could also be implemented. One benefit of holistic evaluation when compared to the method by Diao et al. is its smaller memory consumption. We studied different methods for the evaluation of value-based predicates and conclude that predicate evaluation could also be implemented with PMA-based filtering. We found out that the filter-pruning method can be used to improve the performance of YFilter also in the case of filters that have value-based predicates.

In this research we learned that the Aho-Corasick PMA can be modified into a fast and memory-efficient XML filtering machine. Processing linear XPath filters without wildcards and non-leading descendant operators with a PMA-based algorithm is fairly straightforward. Handling these operators can also be done, but requires some non-trivial modifications into the PMA. We also learned that the filter pruning method can be used to improve the performance of automata-based filtering significantly.

Bibliography

[1] Google Alerts. http://www.google.com/alerts/.

[2] Qizx/open: An open source implementation of XML query in Java. http://www.xfra.net/qizxopen/.

[3] Yahoo! Alerts. http://alerts.yahoo.com/.

[4] YFilter: Filtering and transformation for high-volume XML message brokering. http://yfilter.cs.umass.edu/.

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

[6] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB '00: Proc. of 26th Internat. Conf. on Very Large Databases*, pages 53–64, 2000.

[7] American National Standard for Information Systems. Database Language—SQL, 1992. ANSI X3. 135-1992.

[8] P. Antonellis and C. Makris. XFIS: an XML filtering system based on string representation and matching. *Internat. J. Web Eng. Technol.*, 4(1):70–94, 2008.

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

[10] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference - W3C recommendation, 2004. http://www.w3.org/TR/owl-ref/.

[11] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML path language (XPath) 2.0 W3C recommendation. Technical report, World Wide Web Consortium, 2010. `http://www.w3.org/TR/2010/REC-xpath20-20101214/`.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, 2007. `http://www.w3.org/TR/xquery/`.

[14] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[15] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *ICDE '03: Proc. of the Internat. Conf. on Data Engineering*, pages 139–150, 2003.

[16] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT '97: Proc. of the 6th Internat. Conf. on Database Theory*, pages 336–350, 1997.

[17] C. Byun, K. Lee, and S. Park. A keyword-based filtering technique of document-centric XML using NFA representation. *Internat. J. Appl. Math. Comput. Sci.*, 4(3):136–143, 2008.

[18] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. Afilter: adaptable XML filtering with prefix-caching suffix-clustering. In *VLDB '06: Proc. of the 32nd Internat. Conf. on Very Large Data Bases*, pages 559–570, 2006.

[19] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[20] D. Chen and R. K. Wong. Optimizing the lazy DFA approach for XML stream processing. In *CRPIT '04: Proc. of the 15th Conf. on Australasian Database*, pages 131–140, 2004.

[21] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Scalable filtering of multiple generalized-tree-pattern queries over XML streams. *IEEE Trans. on Knowl. and Data Eng.*, 20(12):1627–1640, 2008.

[22] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In *VLDB '03: Proc. of the 29th Internat. Conf. on Very Large Data Bases*, pages 237–248, 2003.

[23] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, 1979.

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

[25] M. F. Fernández and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE '98: Proc. of the 14th Internat. Conf. on Data Engineering*, pages 14–23, 1998.

[26] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13:294–315, 2004.

[27] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based XML packets filtering for millions of path queries. In *ICDE '05: Proc. of the 21st Internat. Conf. on Data Engineering*, pages 890–901, 2005.

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

[29] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *ICDT '03: Proc. of the 9th Internat. Conf. on Database Theory*, pages 173–189, 2002.

[30] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD '03: Proc. of the 2003 ACM SIGMOD Internat. Conf. on Management of Data*, pages 419–430, 2003.

[31] B. He, Q. Luo, and B. Choi. Cache-conscious automata for XML filtering. *IEEE Trans. on Knowl. and Data Eng.*, 18(12):1629–1644, 2006.

[32] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability (2nd edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[33] S. Hou and H.-A. Jacobsen. Predicate-based filtering of XPath expressions. In *ICDE '06: Proc. of the 22nd Internat. Conf. on Data Engineering*, page 53, 2006.

[34] E. C. Htoon and T. T. S. Nyunt. M-Filter: Semantic XML data filtering system for multiple queries. In *ICIS '09: Proc. of the 8th IEEE/ACIS Internat. Conf. on Computer and Information Science*, pages 1167–1171, 2009.

[35] Internat. Press Telecommunications Council. NewsML: News Exchange Format. `http://www.newsml.org/`.

[36] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005.

[37] C. Koch, S. Scherzinger, and M. Schmidt. XML prefiltering as a string matching problem. In *ICDE '08: Proc. of the 24th Internat. Conf. on Data Engineering*, pages 626–635, 2008.

[38] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *VLDB '05: Proc. of the 31st Internat. Conf. on Very Large Data Bases*, pages 217–228, 2005.

[39] J. Kwon, P. Rao, B. Moon, and S. Lee. Value-based predicate filtering of XML documents. *Data Knowl. Eng.*, 67(1):51–73, 2008.

[40] J. Kwon, P. Rao, B. Moon, and S. Lee. Fast XML document filtering by sequencing twig patterns. *ACM Trans. Internet Technol.*, 9(4):1–51, 2009.

[41] D. Lee, J. Kwon, W. Yang, H. Shin, J.-m. Kwak, and S. Lee. Schema-aware XPath filtering on XML document streams. *Journal of Intelligent Manufacturing*, 20(3):273–282, 2009.

[42] D. Lee, H. Shin, J. Kwon, W. Yang, and S. Lee. SFilter: Schema based filtering system for XML streams. In *MUE '07: Internat. Conf. on Multimedia and Ubiquitous Engineering*, pages 266–271, 2007.

[43] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd edition)*. Sun Microsystems, Palo Alto, California, USA, 1999.

[44] B. Ludäscher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB '02: Proc. of 28th Internat. Conf. on Very Large Data Bases*, 2002.

[45] I. Miliaraki, Z. Kaoudi, and M. Koubarakis. XML data dissemination using automata on top of structured overlay networks. In *WWW '08: Proc. of the 17th Internat. Conf on World Wide Web*, pages 865–874, 2008.

[46] I. Miliaraki and M. Koubarakis. Distributed structural and value XML filtering. In *DEBS '10: Proc. of the 4th ACM Internat. Conf. on Distributed Event-Based Systems*, pages 2–13, 2010.

[47] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT ids. In *ANCS '07: Proc. of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, 2007.

[48] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR '09: Proc. of the 4th Biennial Conf. on Innovative Data Systems Research*, 2009.

[49] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on XML-aware publish-subscribe systems. In *VLDB '07: Proc. of the 33rd Internat. Conf. on Very Large Data Bases*, pages 866–877, 2007.

[50] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Knowl. and Data Eng.*, 19(7):934–949, 2007.

[51] M. Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *CIKM '03: Proc. of the 12th Internat. Conf. on Information and Knowledge Management*, pages 342–349, 2003.

[52] M. Onizuka. Processing XPath queries with forward and downward axes over XML streams. In *EDBT '10: Proc. of the 13th Internat. Conf. on Extending Database Technology*, pages 27–38, 2010.

[53] T. Parr. The ANTLR Parser Generator, 2009. `http://www.antlr.org/`.

[54] F. Peng and S. S. Chawathe. XSQ: a streaming XPath engine. *ACM Trans. Database Syst.*, 30(2):577–623, 2005.

[55] R. Y. Pinter. Efficient string matching. *Combinatorial Algorithms on Words. NATO Advanced Science Institute Series F: Computer and System Sciences*, 12:11–29, 1985.

[56] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys*, 27:742–744, 1918.

[57] E. Qeli and B. Freisleben. Filtering XML documents using XPath expressions and aspect-oriented programming. In *DocEng '06: Proc. of the 2006 ACM Symposium on Document engineering*, pages 85–87, 2006.

[58] I. C. Rahman, M.S. Pattern matching algorithms with don't cares. In *SOFSEM '07: Theory and Practice of Computer Science, 33rd Conf. on Current Trends in Theory and Practice of Comp. Sci*, pages 116–126, 2007.

[59] M. Rahman, C. Iliopoulos, I. Lee, M. Mohamed, and W. Smyth. Finding patterns with variable length gaps or don't cares. In D. Chen and D. Lee, editors, *Computing and Combinatorics*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. 2006.

[60] S. Saigaonkar and M. Rao. XML filtering system based on ontology. In *A2CWiC '10: Proc. of the 1st Amrita ACM-W Celebration on Women in Computing in India*, pages 1–6, 2010.

[61] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[62] Sax Project Organization. Simple API for XML, 2001. `http://www.saxproject.org`.

[63] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. XML-document-filtering automaton. *Proc. VLDB Endowment*, 1(2):1666–1671, 2008.

[64] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. Processing schema-optimized XPath filters by deterministic automata. In *SEDE '09: Proc. of the 18th Internat. Conf. on Software Engineering and Data Engineering*, pages 55–60, 2009.

[65] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. Schema-conscious filtering of XML documents. In *EDBT '09: Proc. of the 12th Internat. Conf. on Extending Database Technology*, pages 970–981, 2009.

[66] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. Evaluating linear XPath expressions by pattern-matching automata. *Journal of Universal Computer Science*, 16(5):833–851, 2010.

[67] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. Online dictionary matching for streams of XML documents. In *IFIP TCS '10: Proc. of the 7th IFIP Internat. Conf. on Theoretical Computer Science*, pages 153–164, 2010.

[68] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. In *SOSP '01: Proc. of the 18th ACM symposium on Operating systems principles*, pages 160–173, 2001.

[69] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proc. of the Conf. on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[70] B. Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000.

[71] D. Suciu. XML Data Repository – The Database Research Group of University of Washington. `http://www.cs.washington.edu/research/xmldatasets/`.

[72] W. Sun, Y. Qin, P. Yu, Z. Zhang, and Z. He. HFilter: Hybrid finite automaton based stream filtering for deep and recursive XML data. In *DEXA '08: Proc. of the 19th Internat. Conf. on Database and Expert Systems Applications*, pages 566–580, 2008.

[73] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymäki. Implementing a scalable XML publish/subscribe system using relational database systems. In *SIGMOD '04: Proc. of the 2004 ACM SIGMOD Internat. Conf. on Management of data*, pages 479–490, 2004.

[74] H. Uchiyama, M. Onizuka, and T. Honishi. Distributed XML stream filtering system with high scalability. In *ICDE '05: Proc. of the 21st Internat. Conf. on Data Engineering*, pages 968–977, 2005.

[75] Z. Xiao-Lin and C. Min. DTD based LazyDFA query optimized algorithm over XML data stream. In *CSSE '08: Proc. of the Internat. Conf. on Computer Science and Software Engineering*, pages 516–518, 2008.

[76] G. Yin, J. Shen, and X. Wang. The improvement of XML filtering based on DFA. In *ICICSE '09: 4th Internat. Conf. Internet Computing for Science and Engineering*, pages 255 –259, 2009.

Aalto-DD 85/2011

ISBN 978-952-60-4286-2 (pdf)
ISBN 978-952-60-4285-5
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS