Indexing XML Documents Using Tree Paths Automaton*

Eliška Šestáková[1] and Jan Janoušek[2]

1 **Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic**
Eliska.Sestakova@fit.cvut.cz

2 **Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic**
Jan.Janousek@fit.cvut.cz

Abstract

An XML document can be viewed as a tree in a natural way. Processing tree data structures usually requires a pushdown automaton as a model of computation. Therefore, it is interesting that a finite automaton can be used to solve the XML index problem. In this paper, we attempt to support a significant fragment of XPath queries which may use any combination of child (i.e., `/`) and descendant-or-self (i.e., `//`) axis. A systematic approach to the construction of such XML index, which is a finite automaton called Tree Paths Automaton, is presented. Given an XML tree model T, the tree is first of all preprocessed by means of its linear fragments called string paths. Since only path queries are considered, the branching structure of the XML tree model can be omitted. For individual string paths, smaller Tree Paths Automata are built, and they are afterwards combined to form the index. The searching phase uses the index, reads an input query Q of size m, and computes the list of positions of all occurrences of Q in the tree T. The searching is performed in time $\mathcal{O}(m)$ and does not depend on the size of the XML document. Although the number of queries is clearly exponential in the number of nodes of the XML tree model, the size of the index seems to be, according to our experimental results, usually only about 2.5 times larger than the size of the original document.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases XML, XPath, index, tree, finite automaton

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.10

1 Introduction

XML plays an important role in many aspects of software development, often to simplify data storage and sharing. Therefore, efficient storing and querying XML data are key tasks which have been extensively studied during the past years. XML data is stored in a plain text format. This provides a software- and hardware-independent way of storing data. To be able to retrieve data from XML documents, various query languages such as XPath [2], XPointer [5], and XLink [6] have been designed.

However, without a structural summary, query processing can be quite inefficient due to an exhaustive traversal on XML data. To achieve fast searching and efficient processing of

* This research has been partially supported by grant CTU in Prague as project No. SGS17/209/OHK3/3T/18.

queries, we can preprocess the data subject and construct an index. This allows us to answer number of queries with low requirements for time complexity.

An XML document can be simply treated as a stream of plain text. Thus, stringology algorithms [3, 4] are applicable in this field. The theory of text indexing is well-researched and uses many sophisticated data structures, such as suffix tree, suffix array, or factor automaton.

However, the internal structure of XML documents can be also viewed as a tree in a natural way. The algorithmic discipline interested in processing tree data structures is called arbology, and it was officially introduced at London Stringology Days 2009 conference. Arbology solves problems such as tree pattern matching, tree indexing, or finding repeats in trees. For its algorithms, arbology uses a standard pushdown automaton as the basic model of computation, unlike stringology where a finite automaton is used.

Nowadays, many methods for indexing XML documents exist, but most of them lack clear references to a systematic approach of the standard theory of formal languages and automata. XML indexes usually work with the tree structure of an XML document, and according to their approaches, we can divide them as follows:

Graph-based methods construct a structural path summary; used to improve especially single path queries. See DataGuides [7], 1-Index [12], PP-Index [17], F&B-Index [8], or MTree [13].

Sequence-based methods transform both the source data and query into sequences. Therefore, querying XML data is equivalent to finding subsequence matches. See ViST [18], PRIX [14].

Node coding methods design codes for each node in order to evaluate the relationship among nodes by computation. See XISS [9].

Adaptive methods adapt their structure to suit the query workload. Therefore, adaptive methods index only the frequently used queries. See APEX Index [1].

Each of the methods listed above has its own advantages and disadvantages. Graph-based methods often do not support complex queries; sequence-based methods are likely to generate approximate solutions; node coding methods are difficult to be applied to ever changing data source; and adaptive methods perform low efficiency on non-frequent queries.

In [16], we discussed the automata-based approach for solving the XML index problem and presented Tree String Path Subsequences Automaton (TSPSA), an index for all linear XPath queries using descendant-or-self axis (i.e., `//`) only. In this paper, we introduce Tree Paths Automaton (TPA) which is designed to process more significant fragment of XPath queries. It is able to answer all queries with any combination of child (i.e., `/`) and descendant-or-self (i.e., `//`) axis, noted as $XP^{\{/,//,name-test\}}$.

Given an XML document D with its corresponding XML tree model $T(D)$, the searching phase uses the index, reads an input query Q of size m, and computes the list of positions of all occurrences of Q in the tree $T(D)$. The searching is performed in time $\mathcal{O}(m)$ and does not depend on the size of the original document D. Although the number of distinct queries is exponential in the number of nodes of the XML tree model, our experiments suggest that determinisation will result in a smaller number of states.

Both TSPSA and TPA support only linear XPath queries. However, the techniques described here may also be relevant to the general XPath processing problem. First, processing linear expressions is a subproblem in processing more complex queries, as we can decompose them into linear fragments. Second, this can be seen as a building block for more powerful processors, such as pushdown automata, able to process branching queries. Moreover, it is

Figure 1 XML Index Problem.

easy to combine the index presented in this paper with other automata-based indexes using standard methods of automata theory.

2 Basic Notions

An *alphabet* A is a finite non-empty set whose elements are called symbols. A *nondeterministic finite automaton* (NFA) is a 5-tuple $M = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is an alphabet, δ is a state transition function from $Q \times A$ to the power set of Q, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states. A finite automaton is *deterministic* (DFA) if $\forall a \in A, q \in Q : |\delta(q, a)| \leq 1$.

A *rooted and directed tree* T is an acyclic connected directed graph $T = (N, E)$, where N is a set of nodes and E is a set of ordered pairs of nodes called directed edges. A *root* is a special node $r \in N$ with in-degree 0. All other nodes of a tree T have in-degree 1. There is just one path from the root r to every node $n \in N$, where $n \neq r$. A node n_1 is a *direct descendant* of a node n_2 if a pair $(n_2, n_1) \in E$.

A *labelling* of a tree $T = (N, E)$ is a mapping N into a set of labels. T is called a *labelled tree* if it is equipped with a labelling. T is called an *ordered* tree if a left-to-right order among siblings in T is given. Any node of a tree with out-degree 0 is called a *leaf*. A *depth* of a node n, noted as $depth(n)$, is the number of directed edges from the root to the node n.

3 Problem Statement

Basically, the XML index problem is to construct an effective data structure able to efficiently process XML query languages, such as XPath. There are two crucial issues connected with all indexing methods. First, the requirement for a small size of the index which, in the best case, should be linear in the size of the preprocessed subject. The second essential feature of the index is very fast query processing. Ideally, queries should be answered in time linear to the size of the query and should not depend on the size of the subject where the queries are located. If these requirements are fulfilled, the index structure allows us to answer number of queries with low requirements for both space and time complexity.

At first, indexing methods usually preprocess the data subject and then construct a structure (an index) that allows to efficiently answer queries related to the content of the subject. In other words, occurrences of input patterns in the subject can be located repeatedly and quickly. See the diagram of the XML index problem in Figure 1.

Among others, the theory of indexing a data structure and finding efficient solutions for particular indexing problems allow us to understand the problem better. Moreover, sometimes various indexes for particular problems can be combined to index, for instance, unions or concatenations. In this last aspect, especially the use of the theory of formal languages and automata could be helpful.

However, the XML index problem is a challenging area. Using only the two most commonly used XPath axes (child axis `/` and descendant-or-self `//` axis), the number of potential queries is exponential (e.g., $\mathcal{O}(2.62^n)$ for a simple linear XML tree with n nodes [10]).

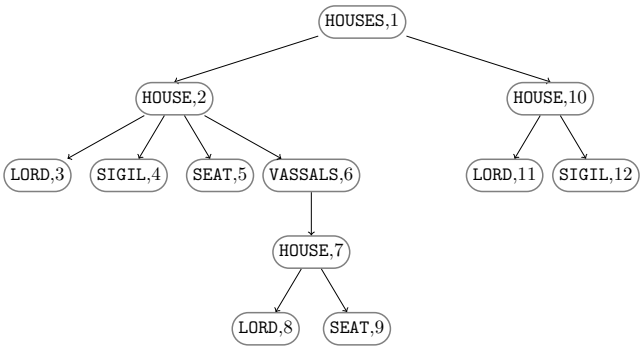

Figure 2 XML tree model $T(D)$ of the XML document D from Example 2.

In this paper, we attempt to support exactly this significant fragment of XPath queries, noted as $XP^{\{/,//,name-test\}}$.

4 XML Tree Model

We model an XML document as an ordered labelled tree where nodes correspond to elements, and edges represent element inclusion relationships. Hence, we only consider the structure of XML documents and, therefore, ignore attributes and the text in leaves.

A node in an XML tree model is represented by a pair $(label, id)$ where $label$ and id represent a tag name and an identifier, respectively. We use a preorder numbering scheme to uniquely assign an identifier to each of the tree nodes. Unique tag names of an XML document form its XML alphabet, formally defined as follows.

▶ **Definition 1** (XML alphabet)**.** Let D be an XML document. An *XML alphabet* A of D, represented by $A(D)$, is an alphabet where each symbol represents a tag name (label) of an XML element in D.

▶ **Example 2.** Let D be the following XML document. The corresponding XML alphabet A is $A(D) = \{$`HOUSES, HOUSE, LORD, SIGIL, SEAT, VASSALS`$\}$. Figure 2 shows its corresponding XML tree model $T(D)$.

```
<HOUSES>
    <HOUSE name="Stark">
        <LORD>Eddard Stark</LORD>
        <SIGIL>Direwolf</SIGIL>
        <SEAT>Winterfell</SEAT>
        <VASSALS>
           <HOUSE name="Karstark">
              <LORD>Rickard Karstark</LORD>
              <SEAT>Karhold</SEAT>
           </HOUSE>
        </VASSALS>
    </HOUSE>
    <HOUSE name="Targaryen">
        <LORD>Daenerys Targaryen</LORD>
        <SIGIL>Dragon</SIGIL>
    </HOUSE>
</HOUSES>
```

5 Tree Paths Automaton

Tree Paths Automaton (TPA) is a finite automaton designed to process a significant fragment of XPath queries which may use any combination of child (i.e., /) and descendant-or-self (i.e., //) axis, noted as $XP^{\{/,//,name-test\}}$. Formally, we can represent such fragment of XPath queries over an XML document D by the following context–free grammar:

$$G = (\{S\}, A(D), \{S \rightarrow SS \mid /a \mid //a\} \wedge a \in A(D), S)\,.$$

This section describes a systematic approach to the construction of TPA and demonstrates it by several examples. Hence, the index is simple and well understandable for anyone who is familiar with the automata theory.

Given an XML tree model T, the tree is first of all preprocessed by means of its linear fragments called string paths. Since only path queries are considered, the branching structure of the XML tree model can be omitted. For individual string paths, smaller Tree Paths Automata are built, and they are afterwards combined using product construction (union) to form the index.

▶ **Definition 3** (String path)**.** Let T be an XML tree model of height h. *A string path* $P = n_1 n_2 \dots n_t$ $(t \leq h)$ of T is a linear path leading from the root $r = n_1$ to the leaf n_t.

▶ **Definition 4** (String path alphabet)**.** Let P be a string path of some XML tree model. *A string path alphabet* A of P, represented by $A(P)$, is an alphabet where each symbol represents a node label in P.

▶ **Definition 5** (String paths set)**.** Let T be an XML tree model with k leaves. A set of all string paths over T is called a *string paths set*, denoted by $P(T) = \{P_1, P_2 \dots P_k\}$.

▶ **Example 6.** Consider the XML tree model T illustrated in Figure 2. We show the content of the corresponding string paths set $P(T)$ below. Each node n of T is represented by its label and identifier, which is shown in parenthesis.

- P_1 = HOUSES(1) HOUSE(2) LORD(3),
- P_2 = HOUSES(1) HOUSE(2) SIGIL(4),
- P_3 = HOUSES(1) HOUSE(2) SEAT(5),
- P_4 = HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) LORD(8),
- P_5 = HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) SEAT(9),
- P_6 = HOUSES(1) HOUSE(10) LORD(11),
- P_7 = HOUSES(1) HOUSE(10) SIGIL(12).

The corresponding string path alphabets are as follows:

- $A(P_1) = A(P_6) = \{$HOUSES, HOUSE, LORD$\}$,
- $A(P_2) = A(P_7) = \{$HOUSES, HOUSE, SIGIL$\}$,
- $A(P_3) = \{$HOUSES, HOUSE, SEAT$\}$,
- $A(P_4) = \{$HOUSES, HOUSE, VASSALS, LORD$\}$,
- $A(P_5) = \{$HOUSES, HOUSE, VASSALS, SEAT$\}$.

XPath queries containing only child axis (i.e., /) are basically prefixes of individual string paths. Therefore, to support only $XP^{\{/,name-test\}}$ fragment of XPath queries, we can use a prefix automaton constructed for a set of strings (a string paths set). At first, for each string path P, a deterministic prefix automaton accepting all $XP^{\{/,name-test\}}$ queries of P can be constructed. Afterwards, individual automata can be combined using product construction (union).

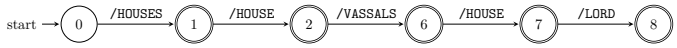

Figure 3 Deterministic prefix automaton for the string path $P =$ `HOUSES`(1) `HOUSE`(2) `VASSALS`(6) `HOUSE`(7) `LORD`(8) from Example 7.

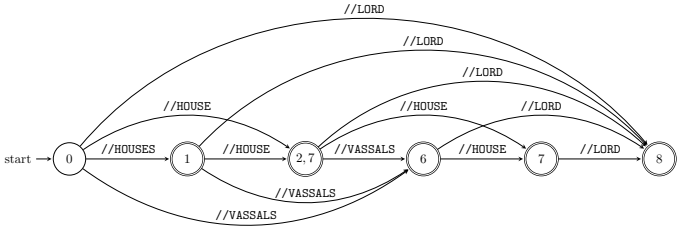

Figure 4 Deterministic subsequence automaton for the string path $P =$ `HOUSES`(1) `HOUSE`(2) `VASSALS`(6) `HOUSE`(7) `LORD`(8) from Example 7.

Data: A string path $P = n_1 n_2 \dots n_{|P|}$.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,name-test\}}$ queries of P.

1. $Q \leftarrow \{0, id(n_1), id(n_2), \dots, id(n_{|P|})\}$,
2. $F \leftarrow Q \setminus \{0\}$,
3. $A = \{/a : a \in A(P)\}$,
4. $\delta(0, /label(n_1)) \leftarrow id(n_1)$, $\forall i \in \{1, 2, \dots, |P| - 1\} : \delta(id(n_i), /label(n_{i+1})) \leftarrow id(n_{i+1})$.

Algorithm 1: Construction of a deterministic prefix automaton for a single string path.

Data: A string path $P = n_1 n_2 \dots n_{|P|}$.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{//,name-test\}}$ queries of P.

1. Construct DFA $M_1 = (Q_1, A, \delta_1, 0, F_1)$ accepting all non-empty "prefixes" of P as follows:
   a. $Q_1 \leftarrow \{0, id(n_1), id(n_2), \dots, id(n_{|P|})\}$,
   b. $F_1 \leftarrow Q_1 \setminus \{0\}$,
   c. $A = \{//a : a \in A(P)\}$,
   d. $\delta_1(0, //label(n_1)) \leftarrow id(n_1)$,
      $\forall i \in \{1, 2, \dots, |P| - 1\} : \delta_1(id(n_i), //label(n_{i+1})) \leftarrow id(n_{i+1})$.
2. Insert ε-transitions into the automaton M_1 leading from each state to its next state. Resulting automaton $M_2 = (Q_2, A, \delta_2, 0, F_2)$ where
   a. $Q_2 \leftarrow Q_1$, $F_2 \leftarrow F_1$,
   b. $\delta_2 \leftarrow \delta_1 \cup \delta'$ and $\delta'(0, \varepsilon) \leftarrow id(n_1)$,
      $\forall i \in \{1, 2, \dots, |P| - 1\} : \delta'(id(n_i), \varepsilon) \leftarrow id(n_{i+1})$.
3. Eliminate all ε-transitions. The resulting automaton is M_3.
4. Construct a deterministic finite automaton M equivalent to M_3 using standard determinisation algorithm based on the subset construction (see [11], Algorithm 1.40).

Algorithm 2: Construction of a deterministic subsequence automaton for a single string path.

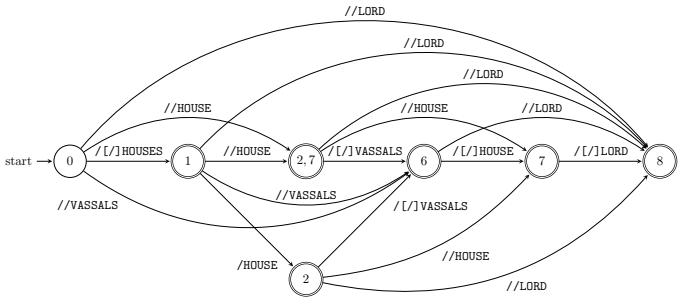

Figure 5 Deterministic Tree Paths Automaton for the string path P = `HOUSES`(1) `HOUSE`(2) `VASSALS`(6) `HOUSE`(7) `LORD`(8) from Example 7.

To satisfy XPath queries $XP^{\{//,name-test\}}$ containing descendant-or-self-axis (i.e., `//`) only, we are interested in subsequences of a string path rather than its prefixes. For each string path P, we can construct a deterministic subsequence automaton, in this case, accepting all $XP^{\{//,name-test\}}$ queries of P. Afterwards, by product construction, we get Tree String Path Subsequences Automaton, which we presented earlier in [16].

To provide a solution for XPath queries $XP^{\{/,//,name-test\}}$ containing any combination of child and descendant-or-self axis, we first propose a building algorithm that combines prefix and subsequence automata for a single string path P to answer all $XP^{\{/,//,name-test\}}$ queries of P. See Algorithm 3 and Example 7.

▶ **Example 7.** Let D and $T(D)$ be the XML document and its corresponding XML tree model from Example 2 and Figure 2, respectively. Given P = `HOUSES`(1) `HOUSE`(2) `VASSALS`(6) `HOUSE`(7) `LORD`(8) as the input string path, Algorithm 3 conducts these steps:

1. constructing a deterministic prefix automaton for P as shown in Figure 3,
2. building a deterministic subsequence automaton for P as shown in Figure 4,
3. combining these two automata as described in step 3 of the algorithm. See resulting Tree Paths Automaton for P in Figure 5. Note, that transition rules $\delta(p,$`/[/]LABEL`$) = q$ represent two transitions leading from the state p to the state q: $\delta(p,$`/LABEL`$) = q$ and $\delta(p,$`//LABEL`$) = q$.

To obtain the final index for an XML document, we again use the product construction (union) of automata that were constructed for individual string paths by Algorithm 3. Algorithm 4 describes the whole process in detail and Example 8 demonstrates the result.

▶ **Example 8.** Let D be the XML document from Example 2. The corresponding Tree Paths Automaton accepting all $XP^{\{/,//,name-test\}}$ queries, constructed by Algorithm 4, is shown in Figure 6. Again, we note that transition rules $\delta(p,$`/[/]LABEL`$) = q$ represent two transitions leading from the state p to the state q: $\delta(p,$`/LABEL`$) = q$ and $\delta(p,$`//LABEL`$) = q$.

5.1 Evaluation of Input Queries

This section describes the searching phase using the index. To compute positions of all occurrences of an input query Q in an XML tree model $T(D)$ of given XML document D, we simply run Tree Paths Automaton on the input query. Eventually, the answer for the input query is given by the d-subset contained in the terminal state of the automaton. If there is no transition that matches the input symbol, the automaton stops and rejects the input. Therefore, there are no elements in the XML document satisfying the query.

Data: A string path $P = n_1 n_2 \dots n_{|P|}$ of an XML tree model using preorder numbering scheme.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,//,name-test\}}$ queries of P.

1. Construct a deterministic finite automaton $M_1 = (Q_1, A_1, \delta_1, 0, F_1)$ accepting all $XP^{\{/,name-test\}}$ queries of P using Algorithm 1.
2. Construct a deterministic finite automaton $M_2 = (Q_2, A_2, \delta_2, 0, F_2)$ accepting all $XP^{\{//,name-test\}}$ queries of P using Algorithm 2.
3. Construct a deterministic finite automaton $M = (Q, A_1 \cup A_2, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{/,//,name-test\}}$ queries of P as follows:

initialize $Q = Q_1 \cup Q_2$;
create a new queue S and initialize $S = Q$;
while *S is not empty* **do**
  State $q \leftarrow S.pop$;
  forall $a \in A_1$ **do**
    create a new d-subset d;
    forall *numbers n in the d-subset of q* **do**
      if $\delta_1(n, a) \neq \emptyset$ **then**
        add n into d;
      end
    end
    if $d \notin Q$ **then**
      $Q = Q \cup \{d\}$;
      $S.push(d)$
    end
    $\delta(q, a) \leftarrow d$; ▷ `add / transitions`
  end
  find the smallest number m in the d-subset of q;
  find a matching state $q_2 \in Q_2$ containing m as the smallest number in its d-subset;
  $\forall a \in A_2 : \delta(q, a) \leftarrow \delta_2(q_2, a)$; ▷ `add // transitions`
end

Algorithm 3: Construction of Tree Paths Automaton for a single string path.

Data: A string paths set $P(T) = \{P_1, P_2, \dots, P_k\}$ of an XML tree model $T(D)$ with k leaves.
Result: DFA M accepting all $XP^{\{/,//,name-test\}}$ queries of the XML document D.

1. For all $P_i \in P(T)$ construct a finite automaton $M_i = (Q_i, A_i, \delta_i, 0, F_i)$ accepting all $XP^{\{/,//,name-test\}}$ queries of P_i using Algorithm 3.
2. Construct a deterministic Tree Paths Automaton $M = (Q, \{/a, //a : a \in A(D)\}, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{/,//,name-test\}}$ queries of the XML document D using product construction (union).

Algorithm 4: Construction of Tree Paths Automaton for an XML document D.

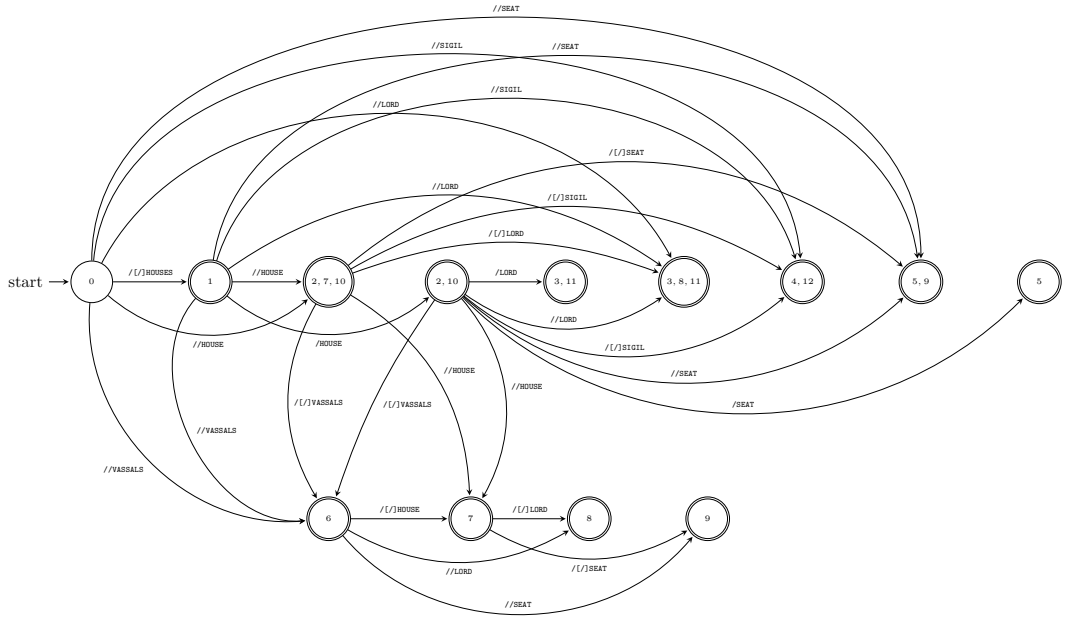

Figure 6 Deterministic Tree Paths Automaton from Example 8.

6 Time and Space Complexities

Tree Paths Automaton is designed to efficiently evaluate all $XP^{\{/,//,name-test\}}$ queries of an XML document D. The number of such queries is exponential in the number of nodes of the XML tree model $T(D)$. However, the running time for answering a query is clearly linear in the size of the query and does not depend on the size of the XML document, i.e., $\mathcal{O}(m)$ for a query of size m.

The size of the index, i.e., the number of states of the deterministic Tree Paths Automaton, is closely related to the equivalence problem of XPath queries. Two XPath queries are equivalent if and only if their result sets of elements are equal. Each state of TPA corresponds to an answer of a single query or a collection of queries. Although the number of different queries accepted by TPA is exponential, usually a lot of them is equivalent.

That is, if we know the number of unique query answers, we can construct a deterministic automaton answering all queries using exactly this number of states. On the other hand, we can obviously use TPA to decide the equivalence of two queries and even determine equivalence classes.

From another point of view, we can examine the number of states of TPA by using our knowledge about smaller automata M_i constructed for individual string paths. Assume $|M_i|_{max}$ is the maximum possible size of TPA for a single string path. Therefore, the number of states of TPA for the XML document can by trivially bounded by $\mathcal{O}(|M_i|^k_{max})$ (size of a product of k automata with maximum of $\mathcal{O}(|M_i|_{max})$ states). However, this is the asymptotic upper bound. The size of the index seems to be according to our experimental results much smaller.

For a common XML document (XML with l-property), in which nodes with the same label can only appear at the same level of the XML tree model, the asymptotic upper bound is $\mathcal{O}(h \cdot 2^k)$.

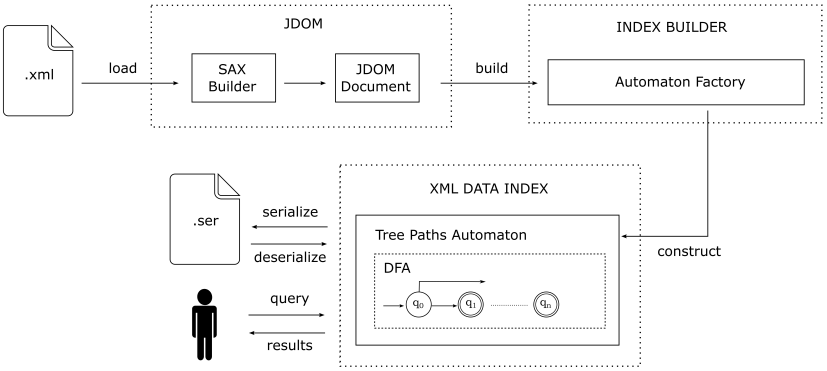

Figure 7 System Architecture of `tpalib`.

▶ **Definition 9** (Level property). Let $T = (N, E)$ be a labelled directed rooted tree. Level property (l-property):

$$\forall n_1, n_2 \in N \wedge n_1 \neq n_2 : label(n_1) = label(n_2) \implies depth(n_1) = depth(n_2)\,.$$

▶ **Definition 10** (State level). Let $M = (Q, A, \delta, q_0, F)$ be an acyclic deterministic finite automaton. A state level s of a state q is a maximal number of transitions leading from the initial state q_0 to q.

▶ **Theorem 11.** *Let D be an XML document and $T(D)$ be its XML tree model satisfying l-property with height h and k leaves. The number of states of the deterministic TPA constructed for the XML document D by Algorithm 4 is $\mathcal{O}(h \cdot 2^k)$.*

Proof. There is k string paths in $T(D)$, for which we construct a set S of k deterministic TPA of no more than h states each (due to l-property). We can run all automata "in parallel", by remembering the states of all automata by constructing k-tuples q while reading the input. This is achieved by the product construction. This way we construct the Tree Paths Automaton M for $T(D)$.

Due to l-property of $T(D)$ it holds that: The target state of a transition labelled with l is either a sink state or its state level is the same in each automaton in S. Hence, the k-tuples $(q_1, q_2, \ldots, q_k)$ are restricted as follows: If state level of q_1 is s, then each of $q_2, \ldots, q_k$ is either a sink state or of state level s. If q_1 is a sink state, then q_2 is arbitrary, but each of $q_3, \ldots, q_k$ is either a sink state or the same state level as q_2. In addition, the k-tuples of levels 0 and 1 are always $(0_1, 0_2, \ldots, 0_k)$ and $(1_1, 1_2, \ldots, 1_k)$, respectively. Therefore, the maximum number of states of M is $2 + 2^{k-1} \cdot (h-1) + 2^{k-2} \cdot (h-2)$. ◀

7 Experimental Evaluation

This section explores the performance of Tree Paths Automaton. We first present TPA System Architecture. Then, we introduce the testing environment for our experiments and characteristics of selected XML data sets. Afterwards, we study space requirements of TPA and finally present a performance study over XPath queries that are supported by TPA.

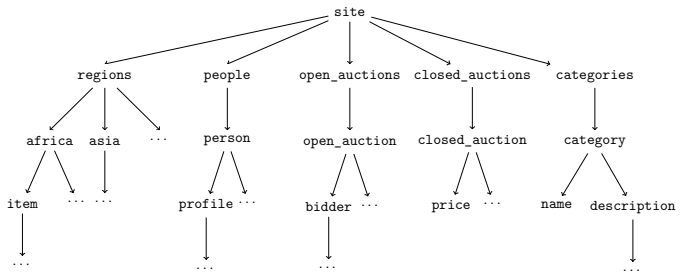

Figure 8 Partial scheme of XMark data sets.

Table 1 Characteristics of XMark benchmark files.

Key	XML File	Xmark `xmlgen` Scaling Factor	# Elements	File size [MB]
D_1	XMark-f0	0	382	0.03
D_2	XMark-f0.001	0.001	1,729	0.10
D_3	XMark-f0.005	0.005	8,518	0.60
D_4	XMark-f0.01	0.01	17,132	1.20
D_5	XMark-f0.5	0.5	832,911	58.00

7.0.1 System Architecture and Testing Environment

The XML index software was developed using Java SE, JDK 8u45 in the NetBeans IDE 8.0.2 and was designed as *Java Class Library* called `tpalib`. The system architecture of the `tpalib` is illustrated in Figure 7. The library consists of three virtual parts called JDOM, Index Builder and XML Data Index.

The experiments were conducted under the environment of Intel Core i7 CPU @ 2.00 GHz, 8.0 GB RAM and 240 GB SSD disk with Windows 8.1 operation system running.

7.1 XML Data Sets

For our experimental evaluation, we selected XML benchmark XMark data sets generated by `xmlgen` [15]. The XMark data set is a single record with a very large and fairly complicated tree structure with a maximal depth of 11 and average depth of 4.5. The XML data models an on-line auction site. Some of element relationships are illustrated in Figure 8.

Table 1 describes relevant characteristics of generated data sets. the first column defines data set keys. The second column shows names of generated XML files. The next column contains XMark `xmlgen` document scaling factors,[1] float values where 0 produces the "minimal document." The fourth column shows numbers of element nodes in generated files and, finally, the last column contains the size of files in megabytes.

7.2 Index Size and Performance on Query Processing

Table 2 shows the experimental results on the index size for generated XMark data sets. The space requirements of the index structure was measured using the size of the file with serialized `TreePathsAutomaton` Java object. The results suggest that the ratio of the index

[1] `http://www.xml-benchmark.org/faq.txt`

Table 2 Experimental results on index size.

Key	Index Size [MB]	Index Size / XML File Size
D_1	0.08	2.60
D_2	0.30	3.00
D_3	1.35	2.25
D_4	2.68	2.23
D_5	129.00	2.22

Table 3 Set of queries used in performance analysis.

Key	XPath Query
Q_1	`/site/open_auctions`
Q_2	`/site/people/person/name`
Q_3	`/site/regions/europe/item/description/parlist/listitem/text/emph`
Q_4	`//person//watch`
Q_5	`//regions//mail//date`
Q_6	`//site//regions//europe//description//listitem//text//emph`
Q_7	`/site//open_auction`
Q_8	`//people/person//watch`
Q_9	`//regions/europe//item//parlist/listitem//text/emph`

size to original XML data size stays linear since the second column shows that the size of TPA data is only about 2.5 times larger than the size of the original document size.

The analysis on performance of query processing was conducted in comparison with a well-known reference implementation called Saxon.[2] The Saxon package is a collection of Java tools for processing XML documents. One of the main components is an XPath processor accessible to applications via its supplied API. Our measurements reflect query processing time only. Hence, document loading cost and query parsing cost have been excluded from the measurements.

Table 2 lists 9 sample queries we used for the experiments. The queries are split into categories depending on the type of axis used: Q_1–Q_3 queries contain child axis only, Q_4–Q_6 include descendant-or-self axis only, and Q_7–Q_9 queries use combination of both axes. Numbers of elements satisfying individual queries in each of generated data sets are shown in Table 4.

Figure 9 summarizes the experimental results of Tree Paths Automaton and Saxon. The graph is plotted using logarithmic scale. The x-axis represents the data sets, while the y-axis shows the response time in milliseconds. We used light blue dashed lines to display Saxon results, whereas TPA score is depicted as dark blue solid lines.

As for Saxon, there appears to be a clear upward pattern in the query processing time with growing size of data sets. We can also see that queries Q_1–Q_3 that use only child axis are easier to evaluate than more complex queries including also descendant-or-self axis. However, TPA results remain stable with processing time around 1 to 3 milliseconds. That is since the searching phase of all elements satisfying the query depends only on the size of a query and does not depend on the size of a data set. Overall, the sample queries achieve better response time using our proposed indexing method.

[2] Available from `http://saxon.sourceforge.net/`

Table 4 Number of elements satisfying queries in the generated data sets.

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
D_1	1	1	2	1	5	2	1	1	2
D_2	1	25	2	50	20	4	12	50	2
D_3	1	127	5	247	124	6	60	247	5
D_4	1	255	17	488	205	50	120	488	43
D_5	1	12,750	1,235	25,414	10,455	2,357	6,000	2,5414	2,099

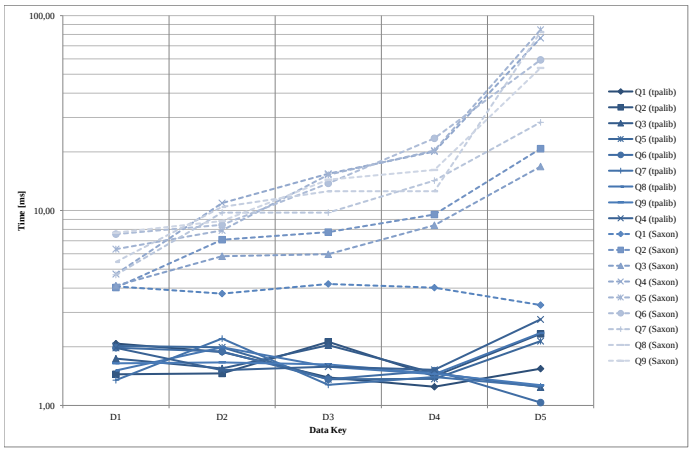

Figure 9 Performance comparison of TPA and Saxon (logarithmic scale).

8 Conclusion and Future Work

A simple method for indexing XML documents using the theory of formal languages and automata was presented. Tree Paths Automaton is able to answer all queries which may use any combination of child (i.e., `/`) and descendant-or-self (i.e., `//`) axis, noted as $XP^{\{/,//,name-test\}}$.

Given an XML document D with its corresponding XML tree model $T(D)$, the tree is preprocessed and an index, which is a finite automaton, is constructed. The searching phase uses the index, reads an input query Q of size m, and computes the list of positions of all occurrences of Q in the tree $T(D)$. The searching is performed in time $\mathcal{O}(m)$ and does not depend on the size of the original XML document.

Although the number of distinct queries is exponential in the number of nodes of the XML tree model, the size of the index seems to be according to our experimental results only about 2.5 times larger than the size of the original document. There is also a number of interesting open problems that we hope to explore in the future:

- develop an incremental building algorithm for our automata-based indexes to efficiently adapt their structure to ever changing XML data sources,
- adapt our indexing methods to be able to support multiple XML documents,
- extend our methods to support more complex queries (e.g., including attributes, wildcards, branching etc.).

References

1. Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: an adaptive path index for XML data. In *SIGMOD International Conference on Management of Data*, pages 121–132, 2002.
2. James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*, 1999. URL: `http://www.w3.org/TR/xpath`.
3. Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
4. Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.
5. Steve DeRose, Ron Daniel Jr., Paul Gross, Eve Maler, Jonathan Marsh, and Norman Walsh. *XML Pointer Language (XPointer)*, 2002. URL: `http://www.w3.org/TR/xptr`.
6. Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink) version 1.0. Technical report, World Wide Web Consortium, 2001. URL: `http://www.w3.org/TR/xlink`.
7. Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *23rd International Conference on Very Large Data Bases*, pages 436–445, 1997.
8. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD International Conference on Management of Data*, pages 133–144, 2002.
9. Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *27th International Conference on Very Large Data Bases*, pages 361–370, 2001.
10. Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *31st International Conference on Very Large Data Bases*, pages 469–480, 2005.
11. Bořivoj Melichar, Jan Holub, and Tomáš Polcar. *Text Searching Algorithms*. Czech Technical University in Prague, 2005. Available at `http://www.stringology.org/athens/TextSearchingAlgorithms`.
12. Tova Milo and Dan Suciu. Index structures for path expressions. In Catriel Beeri and Peter Buneman, editors, *7th International Conference on Database Theory*, pages 277–295, 1999.
13. P. Mark Pettovello and Farshad Fotouhi. MTree: An XML XPath graph index. In *ACM Symposium on Applied Computing*, pages 474–481, 2006.
14. Praveen Rao and Bongki Moon. PRIX: indexing and querying XML using prufer sequences. In *20th International Conference on Data Engineering*, pages 288–299, March 2004.
15. Albrecht Schmidt. XMark: an XML benchmark project. `http://www.xml-benchmark.org/`.
16. Eliška Šestáková and Jan Janoušek. Tree string path subsequences automaton and its use for indexing xml documents. In *International Symposium on Languages, Applications and Technologies (SLATE)*, pages 171–181, 2015.
17. Nan Tang, Jeffrey Xu Yu, M. Tamer Ozsu, and Kam-Fai Wong. Hierarchical indexing approach to support XPath queries. In *IEEE 24th International Conference on Data Engineering*, pages 1510–1512, April 2008.
18. Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD International Conference on Management of Data*, pages 110–121, 2003.