information

MDPI

Article

Automata Approach to XML Data Indexing

Eliška Šestáková [†],[*] and Jan Janoušek [†]

Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Praha 6, Czech Republic; jan.janousek@fit.cvut.cz

[*] Correspondence: eliska.sestakova@fit.cvut.cz; Tel.: +420-728-145-282

[†] These authors contributed equally to this work.

Received: 1 December 2017; Accepted: 3 January 2018; Published: 6 January 2018

Abstract: The internal structure of XML documents can be viewed as a tree. Trees are among the fundamental and well-studied data structures in computer science. They express a hierarchical structure and are widely used in many applications. This paper focuses on the problem of processing tree data structures; particularly, it studies the XML index problem. Although there exist many state-of-the-art methods, the XML index problem still belongs to the active research areas. However, existing methods usually lack clear references to a systematic approach to the standard theory of formal languages and automata. Therefore, we present some new methods solving the XML index problem using the automata theory. These methods are simple and allow one to efficiently process a small subset of XPath. Thus, having an XML data structure, our methods can be used efficiently as auxiliary data structures that enable answering a particular set of queries, e.g., XPath queries using any combination of the child and descendant-or-self axes. Given an XML tree model with n nodes, the searching phase uses the index, reads an input query of size m, finds the answer in time $\mathcal{O}(m)$ and does not depend on the size of the original XML document.

Keywords: XML; XPath; index; indexing; tree; automaton; finite state automaton; finite state machine

1. Introduction

Extensible Markup Language (XML), which became a World Wide Web Consortium (W3C) Recommendation in 1998, still belongs to the main methods of exchanging data over the Internet. It also plays an important role in many aspects of software development, often to simplify data storage and sharing. Thus, efficient storing and querying of XML data are key tasks that have been extensively studied during the past few years [1–15].

To be able to retrieve the data from XML documents, various query languages such as XPath [16], XPointer [17] and XLink [18] 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, we can preprocess the data subject and construct an index.

Basically, the problem of XML data indexing is constructing a 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; second, very fast query processing, which ideally means that the answers to the queries are found in time linear to the size of the query and do not depend on the size of the subject where the queries are located. If these requirements are fulfilled, the index structure allows one to answer a number of queries with low requirements for both time and space complexity.

However, the flexibility of the specifications of XML queries adds to the challenge of indexing methods, and the creation of a universal index that is able to process all of the possible XML queries efficiently is a very 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 [19]). Therefore, there is always a tradeoff between the size and the power of an XML index. It either needs to be large to perform well or performs poorly as a consequence of saving space.

In this paper, we propose three indexing methods that are all based on finite state automata. These methods are simple and allow one to efficiently process a small subset of XPath. Therefore, having an XML data structure, our methods can be used efficiently as auxiliary data structures that enable answering a particular set of queries.

All automata presented in this paper support some fragments of linear XPath queries. In particular, we focused on the two common axes (i.e., child and descendant-or-self) with name tests. However, the techniques described here may be also relevant to the general XPath processing problem. First, we believe that a similar approach can be used to build automata that support other XPath axes (e.g, an automaton supporting the parent and ancestor axis). Second, processing linear expressions is a subproblem of processing more complex queries, as we can decompose them into linear fragments. Third, this can be seen as a building block for more powerful processors able to process branching queries. Moreover, it is easy to combine our indexes presented in this paper with other automata-based indexes using standard methods of automata theory.

First, we present Tree String Paths Automaton (TSPA) and Tree String Path Subsequences Automaton (TSPSA; introduced in [20]), aimed at assisting in evaluating XPath queries with either child or descendant-or-self axes only. Then, we present Tree Paths Automaton (TPA; introduced in [21]), which is designed to process XPath queries using any combination of child and descendant-or-self axes.

The rest of this paper is organized as follows. Section 2 discusses state-of-the-art methods for XML data indexing. Section 3 gives the necessary theoretical background including a brief description of both XML and XPath. Next, in Section 4, we introduce our approach to XML data indexing using automata theory. The theoretical time and space complexities of the proposed methods and experimental evaluation are discussed in Section 5 and Section 6, respectively. Finally, we summarize the contributions of our research, discuss our future work directions and conclude the paper in Section 7.

2. Related Work

An XML document can be simply treated as a stream of plain text. Thus, stringology algorithms [22,23] are applicable in this field. The theory of text indexing is well researched and is based on 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. Trees are among the fundamental and well-studied data structures in computer science. They naturally express a hierarchical structure and are widely used in many applications.

The algorithmic discipline interested in processing tree data structures is called arbology [24], which was officially introduced at the London Stringology Days 2009 conference. Arbology solves problems such as tree pattern matching, tree indexing and finding repeats in trees. For its algorithms, arbology uses a standard pushdown automaton as the basic model of computation, unlike stringology, where a finite state automaton is used.

Nowadays, many methods solving the problem of XML data indexing exist. According to their approaches, we can classify them as follows:

- Graph-based methods construct a structural path summary that can be used to improve query efficiency, especially for single path queries. In this category, we can classify, for instance, the following methods: DataGuides [1], 1-Index [6], Hierarchical Indexing Approach to Support XPath Queries (PP-Index) [4], Forward and Backward (F&B)-Index [5], An XML XPath Graph Index (MTree) [2] or Compact Tree (CTree) [3].
- Sequence-based methods transform both the source data and query into sequences. Therefore, querying XML data is equivalent to finding subsequence matches. Into this category we can

classify, for instance, the following methods: Virtual Suffix Tree (ViST) [11], Prüfer sequences for Indexing XML (PRIX) [7] or that of [12].

- Node-coding methods (see [25]) apply certain coding strategies to design codes for each node, in order for the relationship among nodes to be evaluated by computation. Into this category, we can classify, for example, the XML Indexing and Storage System (XISS) [10], XR-tree [13], Dewey numbering schema [14] or relative region coordinate [15].
- Adaptive methods can adapt their structure to suit the query workload. Therefore, adaptive methods index only the frequently-used queries. Into this category, we can classify, for instance, the following methods: Adaptive Path index for XML data (APEX) [9] and Adaptive Index for Branching XML Queries (AB-Index) [8].

Generally speaking, every method has its own advantages; however, shortcomings do exist. Graph-based methods usually do not support complex queries; sequence-based methods tend to generate approximate solutions, thus requiring a great deal of validation; node-coding methods are very difficult to apply to an ever-changing data source; and adaptive methods have low efficiency on non-frequent query.

In this paper, we present some new methods for XML data indexing using finite state automata. To our knowledge, although a number of automaton formalisms were proposed for XML, they usually deal with different problems than XML data indexing (see [26]). For instance, finite state automata or tree automata [27–29] are used for the pattern matching (e.g., filtering of XML documents), which basically means that instead of preprocessing the XML data, they preprocess a set of queries (patterns) [28,30–33]. Automata are also often connected with the XML validation problem [28,34,35]. In [36], finite state automata are used to represent rewritten regular XPath queries, which enables answering queries posed on virtual views of XML documents.

3. Theoretical Background

3.1. Basic Notions

An alphabet A is a finite non-empty set whose elements are called symbols. A Nondeterministic Finite State Automaton (NFSA) is a five-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 and $F \subseteq Q$ is a set of final states. A finite state automaton is Deterministic (DFSA) if $\forall a \in A, q \in Q : |\delta(q, a)| \leq 1$.

For a NFSA $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$, we can construct an equivalent DFSA $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ using the standard determinisation algorithm based on subset construction [37]. Every state $q \in Q_2$ corresponds to some subset of Q_1. We call this subset a d-subset (deterministic subset). The d-subset is a totally ordered set; the ordering is equal to ordering of states of M_1 considered as natural numbers.

Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ and $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ be two finite state automata. We define a finite state automaton $M_\cup$ such that $L(M_\cup) = L(M_1) \cup L(M_2)$. We can build the automaton $M_\cup$ by running the automata M_1 and M_2 in "parallel" by remembering the states of both automata while reading the input. This is achieved by the product construction [37]: $M_\cup = (Q_1 \times Q_2, A, \delta, (q_{01}, q_{02}), (F_1 \times Q_2) \cup (Q_1 \times F_2))$, where $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

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 zero. All other nodes of a tree T have in-degree one. 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 labeling of a tree $T = (N, E)$ is a mapping N into a set of labels. T is called a labeled tree if it is equipped with a labeling. 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 zero is called a leaf. The depth of a node n, denoted as $depth(n)$, is the number of directed edges from the root to the node n.

3.2. XML

XML is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The set of marks of an XML document is not fixed and can be defined in various ways for each document. The key constructs of an XML document are tags, elements and attributes. We will illustrate these constructs by means of an example.

Example 1. *Consider the sample of an XML document below, which displays some basic information about noble houses of the Seven Kingdoms in Westeros (Westeros is a fictional continent of an American fantasy drama named Game of Thrones).*

```
<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>
```

We can see that `HOUSES` is the most outer element. A start-tag of this element is of the form `<HOUSES>`, whereas the corresponding end-tag, indicating the end of the element, is `</HOUSES>`. Therefore, the content between and including the tags `<HOUSES>` and `</HOUSES>` constitutes the `HOUSES` element.

Elements can be arbitrarily nested inside other elements. For instance, the `HOUSES` element has two `HOUSE` elements as its sub-elements. Every `HOUSE` element includes `LORD` as its first sub-element and optionally `SIGIL`, `SEAT` and `VASSALS` as its second, third and fourth sub-element, respectively.

Another key construct of an XML document is comprised by attributes. For instance, `<HOUSE name="Stark">` indicates that the value of the `name` attribute of that particular `HOUSE` element is `"Stark"`.

3.3. XPath

XPath [16] (XML Path Language) is one of the XML query languages. It gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document and operating on its tree structure. It is a query language for selecting nodes from an XML document, but can also be used to compute values (e.g., strings or numbers) from the content of an XML document.

The following examples of XPath expressions refer to the sample XML document described in Example 1 and illustrate a few key constructs of XPath, which we will refer to later.

Example 2. *Consider an XPath query Q_1 = `/HOUSES/HOUSE/LORD`. The query selects all `LORD` elements having `HOUSE` as the parent element and `HOUSES` as the grandparent element, whereas `HOUSES` must also* *match the most outer element of the document. The resulting set of elements satisfying Q_1 is* {`<LORD>Eddard Stark</LORD>`, `<LORD>Daenerys Targaryen</LORD>`}.

Example 3. *Consider an XPath query $Q_2 =$ `//VASSALS//LORD`. The query selects all `LORD` elements having `VASSALS` as an ancestor element, whereas the `VASSALS` element can be located anywhere in the document. The resulting set of elements satisfying Q_2 is* {`<LORD>Rickard Karstark</LORD>`}.

Example 4. *Consider an XPath query $Q_3 =$ `//HOUSE/LORD`. The query selects all `LORD` elements having `HOUSE` as the parent element, whereas the `HOUSE` element can be located anywhere in the document. The resulting set of elements satisfying Q_3 is* {`<LORD>Eddard Stark</LORD>`, `<LORD>Rickard Karstark</LORD>`, `<LORD>Daenerys Targaryen</LORD>`}.

4. Automata Approach to XML Data Indexing

In this section, we introduce three new methods for the problem of XML data indexing using the automata theory and show that automata can be used efficiently for the purpose of indexing XML documents. These methods are simple and allow one to efficiently process a small subset of XPath. Therefore, having an XML data structure, our methods can be used efficiently as auxiliary data structures that enable answering a particular set of queries. Given an XML document and an input XPath query, the searching phase finds the answer of the query in time linear in the size of the query and does not depend on the size of the original XML document.

This section is organized as follows. First, we provide some common preliminaries. Next, we introduce the Tree String Paths Automaton (TSPA) representing an index for all linear XPath queries using the child axis (i.e., /) only, denoted as $XP^{\{/,name-test\}}$. After that, we present the Tree String Path Subsequences Automaton (TSPSA), an index for all $XP^{\{//,name-test\}}$ queries using the descendant-or-self axis (i.e., //) only. Finally, we introduce the Tree Paths Automaton (TPA), which is designed to process all XPath queries with any combination of child (i.e., /) and descendant-or-self (i.e., //) axes, denoted as $XP^{\{/,//,name-test\}}$.

4.1. Preliminaries

We model an XML document as an ordered labeled 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 preordered 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 5. *Let D be the XML document from Example 1. The corresponding XML alphabet A is $A(D) =$ {`HOUSES`, `HOUSE`, `LORD`, `SIGIL`, `SEAT`, `VASSALS`}. Figure 1 shows its corresponding XML tree model $T(D)$.*

Given an XML tree model, we can preprocess it by means of its linear fragments called string paths. The branching structure of the XML tree model can be omitted, since only path queries will be considered.

Definition 2 (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.*

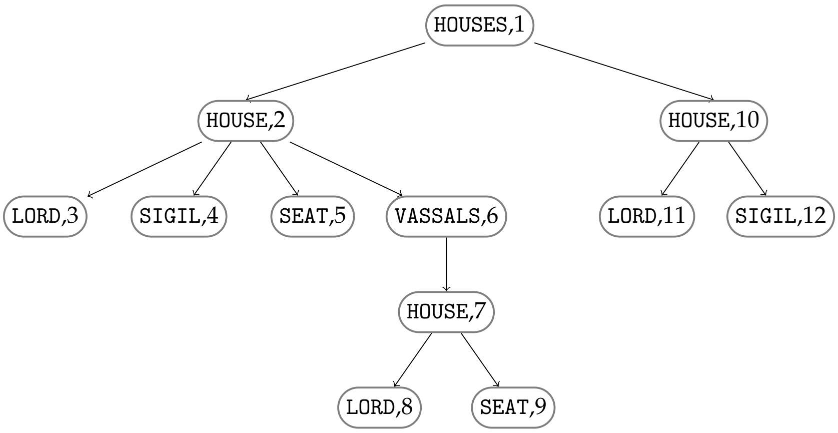


Figure 1. XML tree model $T(D)$ from Example 5.

Definition 3 (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 4 (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 path set, denoted by $P(T) = \{P_1, P_2, \ldots, P_k\}$.*

Example 6. *Consider the XML tree model T illustrated in Figure 1. We show the content of the corresponding string path set $P(T)$ below. Each node of T is represented by its label and identifier, which is shown in parenthesis.*

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

The corresponding string path alphabets are as follows:

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

4.2. Tree String Paths Automaton

The Tree String Paths Automaton (TSPA) is a finite state automaton that speeds up the evaluation of linear XPath queries $XP^{\{/,name-test\}}$ using the child axis (i.e., /-axis) only. Formally, we can represent such a fragment of XPath queries over an XML document D by the following context-free grammar:

$$G = (\{S\}, A(D), \{S \rightarrow SS \mid /a, \text{such as } a \in A(D)\}, S)$$

Definition 5 (Tree string paths automaton)**.** *Let D be an XML document. The tree string paths automaton accepts all $XP^{\{/,name-test\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.*

Since a systematic approach is used in the construction of TSPA, the index is simple and well understandable for anyone who is familiar with the automata theory. We need to point out that TSPA is very similar to strong DataGuides [1] and CTree [3]. Nevertheless, since a similar systematic approach to the construction is used in further, more complex methods, we will demonstrate the basics of our approach using TSPA.

For the XML tree model T, we first of all obtain its string path set $P(T)$. Since XPath queries containing only the child axis are basically prefixes of individual string paths, a prefix automaton for a set of strings (i.e., string paths) can be used. To build a prefix automaton for a string path set $P(T)$, we need to build a prefix automaton for all $P_i \in P(T)$; see Algorithm 1 and Figure 2. Figure 2 illustrates transition diagrams of the prefix automata constructed by Algorithm 1 for all string paths of the XML tree model T from Figure 1.

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

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

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

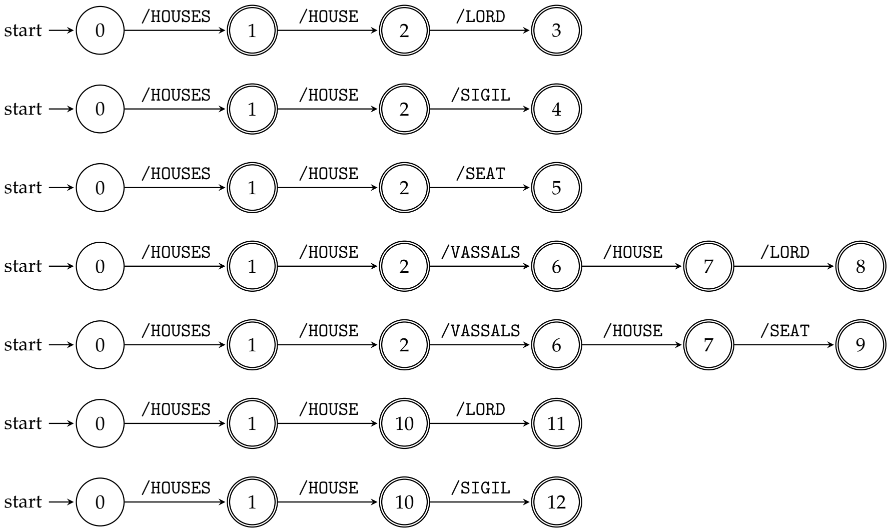


Figure 2. Prefix automata for individual string paths of the XML tree model T from Figure 1.

To build TSPA, we can run all prefix automata, constructed by Algorithm 1, for all string paths "in parallel" by remembering the states of all automata while reading the input. This is achieved by the product construction (see Section 3.1). This way, we construct the TSPA for an XML tree model; see Algorithm 2 and Figure 3. Figure 3 illustrates TSPA constructed by Algorithm 2 for the XML document D and its XML tree model from Example 5 and Figure 1, respectively.

To compute the positions of all occurrences of an input query in the XML tree model, we simply run TSPA 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 (i.e., child axis with a name-test), the automaton stops and rejects the input. Therefore, there are no elements in the XML document satisfying the query. See Example 7.

Example 7. *Consider the XML document D and its XML tree model $T(D)$ from Example 5 and Figure 1, respectively. Suppose we want to evaluate the following XPath query $Q =$ `/HOUSES/HOUSE/LORD` using TSPA in Figure 3. Starting in the initial state, the automaton follows the transition for the first symbol of the input (i.e., `/HOUSES`) and goes to the state (1). Next, the automaton continues reading the second symbol (i.e., `/HOUSE`) and goes from the state (1) to the state $(2, 10)$. Then, it reads the last part of the input, i.e., `/LORD`, and ends in the state $(3, 11)$. Since the whole input is read and the automaton is in a final state, it returns the d-subset $(3, 11)$ as the answer for the input query Q. Finally, the appropriate set of XML elements is returned to a user.*

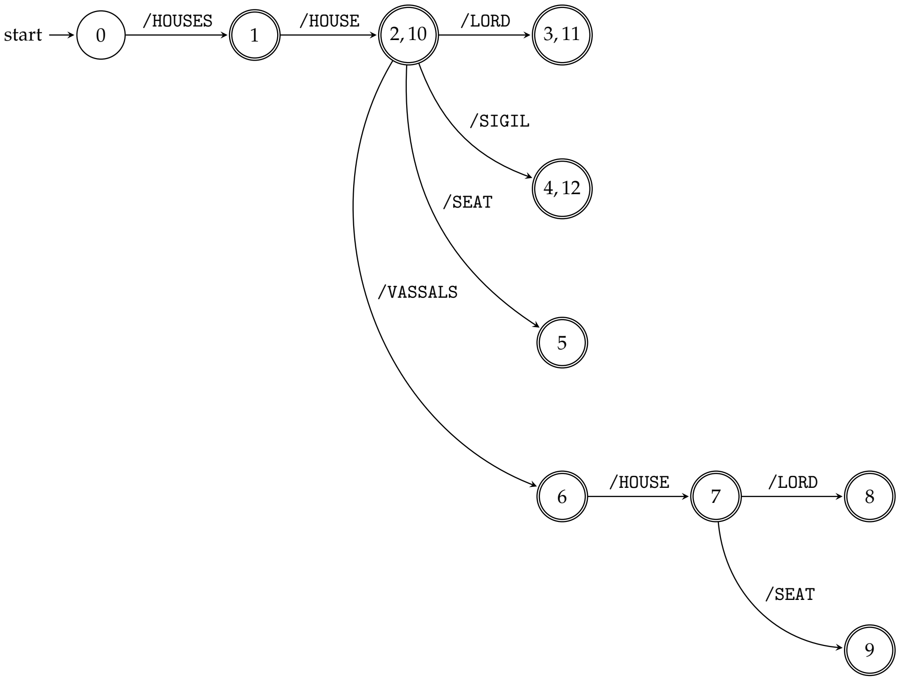


Figure 3. Deterministic tree string paths automaton for the XML tree model T from Figure 1.

Data: String paths set $P(T) = \{P_1, P_2, \ldots, P_k\}$ of an XML tree model $T(D)$ with k leaves.
Result: DFSA $M = (Q, A(D), \delta, 0, F)$ accepting all $XP^{\{/,name-test\}}$ queries of an XML document D.

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

Algorithm 2: Construction of the tree string paths automaton for an XML document.

4.3. Tree String Path Subsequences Automaton

The Tree String Path Subsequences Automaton (TSPSA) is a finite state automaton that efficiently evaluates all linear XPath queries $XP^{\{//,name-test\}}$ where only the descendant-or-self axis (i.e., //-axis) is used. Again, we can represent such a fragment of XPath queries over an XML document D by the context-free grammar as follows:

$$G = (\{S\}, A(D), \{S \rightarrow SS \mid //a, \text{such as } a \in A(D)\}, S)$$

Definition 6 (Tree string path subsequences automaton)**.** *Let D be an XML document. TSPSA accepts all $XP^{\{//,name-test\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.*

As for the TSPA, the construction of TSPSA is very systematic. The given XML tree model T is preprocessed and the string path set $P(T)$ obtained. However, to satisfy XPath queries with the $//$-axis, we are interested in subsequences of a string path rather than its prefixes. Which is why we construct a subsequence automaton for each string path $P_i \in P(T)$ instead of a prefix automaton. The automaton solving the problem of subsequences for both single and multiple strings is also referred to as the Directed Acyclic Subsequence Graph (DASG) and is further studied in [38,39]. Therefore, we propose the XML index problem to be another application area of DASG.

There are three building algorithms for DASG for a set of strings available: right-to-left [40], left-to-right [41] and on-line [39]. However, none of them is based on a subset construction, which gives the sets of positions serving as answers for input queries. Therefore, we propose a direct subset construction of a deterministic subsequence automaton; see Algorithm 3 and Figure 4. Figure 4 shows transition diagrams of the deterministic subsequence automata constructed by Algorithm 3 for all string paths contained in $P(T)$, where T is the XML tree model T illustrated in Figure 1.

```
Data: A string path $P = n_1 n_2 \dots n_{|P|}$.
Result: DFSA $M = (Q, A, \delta, q_0, F)$ accepting all (non-empty) $XP^{\{//,name-test\}}$ queries of $P$
1. $\forall e \in A(P)$ compute $O_P(e)$.
2. Build the "backbone" of the finite state automaton $M = (Q, A, \delta, q_0, F)$:
    (a) $Q \leftarrow \{q_0, q_1, \dots, q_{|P|}\}$, $A \leftarrow \{//a : a \in A(P)\}$, $F \leftarrow Q \setminus \{q_0\}$, $q_0 \leftarrow 0$
    (b) $\forall i$, where $i \leftarrow 1, 2, \dots, |P|$:
        i. set state $q_i \leftarrow O_P(label(n_i))$,
        ii. add $\delta(q_{i-1}, //label(n_i)) \leftarrow q_i$,
        iii. $O_P(label(n_i)) \leftarrow ButFirst(O_P(label(n_i)))$.
3. Insert "additional transitions" into the automaton $M$:
        $\forall i \in \{0, 1, \dots, |P| - 1\}$, $\forall a \in A(P)$:
        i. add $\delta(q_i, //a) \leftarrow q_s$, if there exists such $s > i$ where
           $\delta(q_{s-1}, //a) = q_s \wedge \neg\exists r < s : \delta(q_{r-1}, //a) = q_r$
        ii. $\delta(q_i, //a) \leftarrow \emptyset$ otherwise.
```

Algorithm 3: Construction of a deterministic subsequence automaton for a single string path.

Definition 7 (Set of occurrences of an element label in a string path)**.** *Let $P = n_1 n_2 \dots n_{|P|}$ be a string path and e be an element label occurring at several positions in P (i.e., $label(n_i) = e$ for some i). A set of occurrences of the element label e in P is a totally ordered set $O_P(e) = \{o \mid o = id(n_i) \ \wedge \ label(n_i) = e, \ i = 1, 2, \dots, |P|\}$. The ordering is equal to ordering of element prefix identifiers as natural numbers.*

Definition 8 (ButFirst)**.** *Let P and $O_P(e) = \{o_1, o_2, \dots, o_{|O_P(e)|}\}$ be a string path and a set of occurrences of an element label e in the string path P, respectively. Then, we define a function $ButFirst(O_P(e)) = \{o_2, \dots, o_{|O_P(e)|}\}$.*

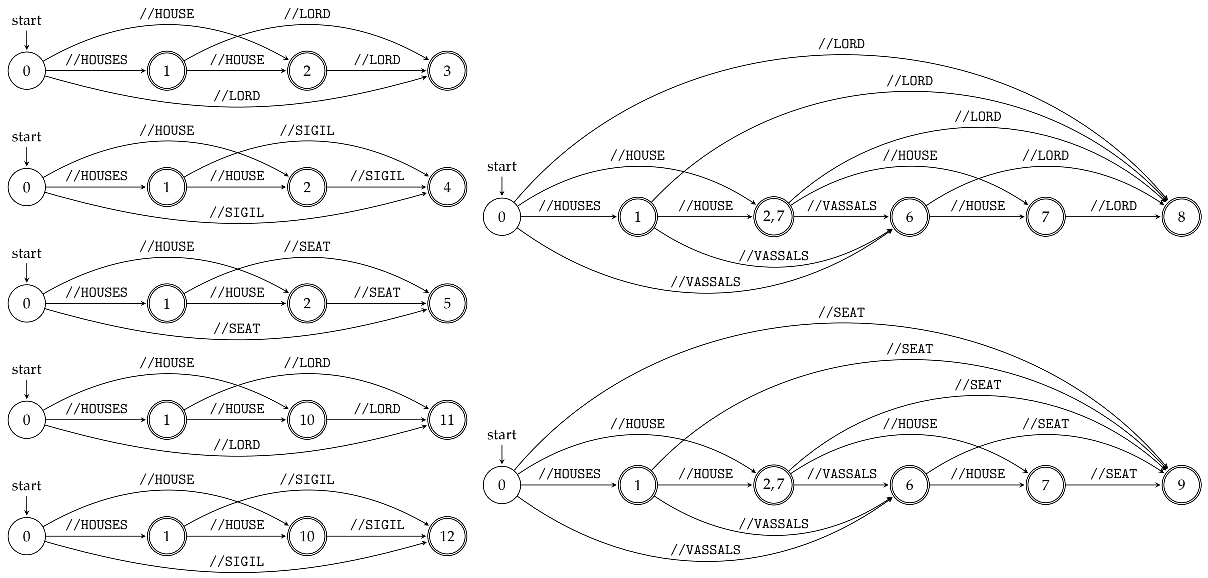


Figure 4. Subsequence automata for individual string paths of the XML tree model T from Figure 1.

Theorem 1. *Given a string path $P = n_1 n_2 \dots n_{|P|}$, Algorithm 3 correctly constructs a deterministic finite state automaton M accepting all $XP^{\{//,name-test\}}$ queries of P.*

Proof. We will prove the following equivalence: M accepts a string X if and only if X is the $XP^{\{//,name-test\}}$ query of the string path P.

1. If M accepts a string X, then X is the $XP^{\{//,name-test\}}$ query of P.
2. If X is the $XP^{\{//,name-test\}}$ query of P, then M accepts X. Assume to the contrary that $X = //x_1//x_2 \dots //x_{|X|}$ over the alphabet $A = \{//a : a \in A(P)\}$ is the $XP^{\{//,name-test\}}$ query of P and M does not accept X. If this is the case, either M reads the whole input and terminates in a non-final state or M does not read the whole input. In the first case, terminating in a non-final state means to stop in the initial state, contradicting our assumption that X is non-empty. The second case, reading just part of the input, means there exists such a symbol $//x_i$ that M has no transition leading from the current state labeled by $//x_i$.

   However, if the automaton reads some symbol, it always goes from the current state to the closest higher state representing an occurrence of that symbol. During Step ii, in Phase 2, all transitions added lead to the neighbor state, and during Step i, in Phase 3, we choose suitable higher state q_s using two conditions. First, the state has to correspond to the correct symbol, which is satisfied by the first condition: there exists such $s > i$ where $\delta(q_{s-1}, //a) = q_s$). Second, we need to ensure that the state is the closest possible, which is satisfied by the second condition: $\neg\exists r < s : \delta(q_{r-1}, //a) = q_r$. Therefore, no subsequence is missed.

   Thus, M reads $x_1 \dots x_{i-1}$, and the current state of M is q_j. Due to Steps 2 and 3, there exists no transition from state q_a to a state q_b where $a \geq b$ (i.e., to the "left"); therefore, $j \geq i - 1$. Because of (2b)i., each state q_k of M corresponds to a node n_k of P. Because of (2b)ii. and 3i., there exists a transition from q_j for x_i such that x_i occurs right of x_{i-1}, as every transition leads from q_j to the state with the incoming transition labeled with x_i (the nonexistent part of 3i.). Therefore, $x_1 \dots x_i$ is not the $XP^{\{//,name-test\}}$ query of P, which is a contradiction. □

We can run all subsequence automata "in parallel" using the product construction (see Section 3.1) and obtain the index for all $XP^{\{//,name-test\}}$ queries of the particular XML document; see Algorithm 4 and Figure 5. Figure 5 illustrates TSPSA constructed by Algorithm 4 for the XML document D and its XML tree model $T(D)$ from Example 5 and Figure 1, respectively.

The searching phase of TSPSA evaluates input queries in the same way as TSPA. Again, the answer for the input query is given by the d-subset contained in the terminal state.

Data: String paths set $P(T) = \{P_1, P_2, \ldots, P_k\}$ of XML tree model $T(D)$ with k leaves.
Result: DFSA $M = (Q, A(D), \delta, 0, F)$ accepting all $XP^{\{//,name-test\}}$ queries of the XML document D.

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

Algorithm 4: Construction of TSPSA for an XML document D.

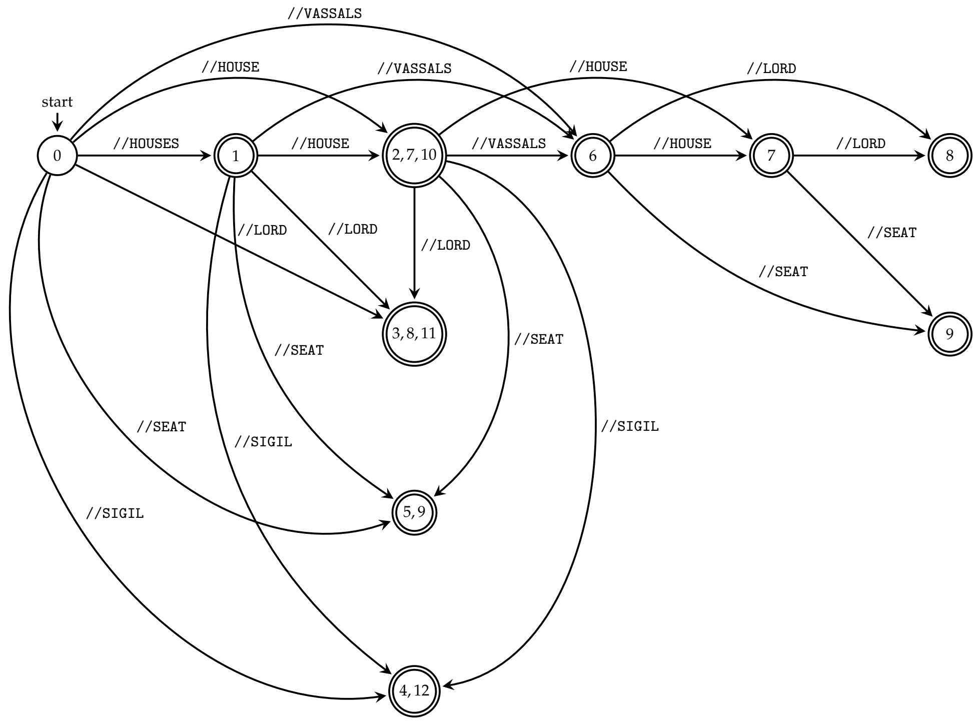


Figure 5. Deterministic tree string path subsequences automaton for the XML tree model T from Figure 1.

4.4. Tree Paths Automaton

Tree Paths Automaton (TPA) is a finite state 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., //) axes, denoted as $XP^{\{/,//,name-test\}}$. The context-free grammar representing such a fragment of XPath queries over an XML document D is as follows:

$$G = (\{S\}, A(D), \{S \rightarrow SS \mid /a \mid //a, \text{such as } a \in A(D)\}, S)$$

TPA combines the principles of both of the formerly introduced automata, i.e., TSPA and TSPSA. Since both $XP^{\{/,name-test\}}$ and $XP^{\{//,name-test\}}$ queries are subsets of $XP^{\{/,//,name-test\}}$ queries, they are naturally supported by TPA.

To provide a solution for XPath queries $XP^{\{/,//,name-test\}}$, 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 5 and Example 8.

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

1. Construct a deterministic finite state 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 state automaton $M_2 = (Q_2, A_2, \delta_2, 0, F_2)$ accepting all
   $XP^{\{//,name-test\}}$ queries of $P$ using Algorithm 3.
3. Construct a deterministic finite state 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 the $a \in A_1$ do
        create a new d-subset $d$;
        forall the numbers $n$ in the d-subset of $q$ do
            if $\delta_1(n, a) \neq \emptyset$ then
                add $n$ into $d$;
            end
        end
        if $d \neq \emptyset$ then
            if $d \notin Q$ then
                $Q = Q \cup \{d\}$;
                $S.push(d)$;
            end
            $\delta(q, a) \leftarrow d$ ;                                  ▷ add / transitions
        end
    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 5: Construction of the tree paths automaton for a single string path.

Example 8. *Let D and $T(D)$ be an XML document and its corresponding XML tree model from Example 5 and Figure 1, respectively. Given $P =$ `HOUSES`(1) `HOUSE`(2) `VASSALS`(6) `HOUSE`(7) `LORD`(8) as the input string path, Algorithm 5 conducts these steps:*

1. *constructs a deterministic prefix automaton for P as shown in Figure 6,*
2. *builds a deterministic subsequence automaton for P as shown in Figure 7,*
3. *combines these two automata as described in Step 3 of the algorithm. See the resulting Tree Paths Automaton (TPA) for P in Figure 8. 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`$) = q$.*

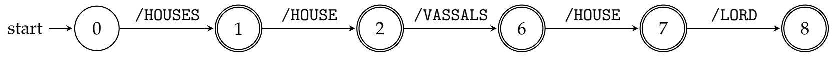


Figure 6. Deterministic prefix automaton for the string path P = HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) LORD(8) from Example 8.

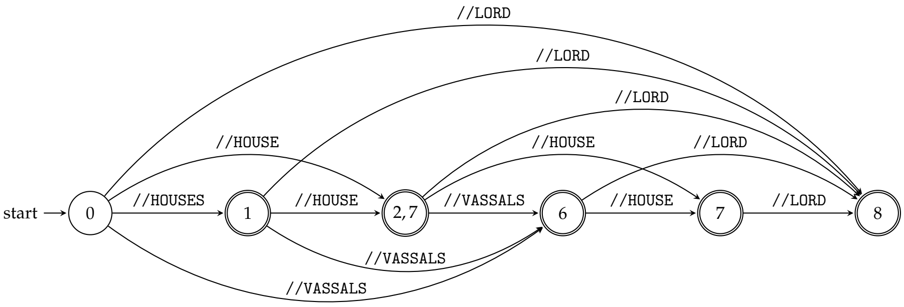


Figure 7. Deterministic subsequence automaton for the string path P = HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) LORD(8) from Example 8.

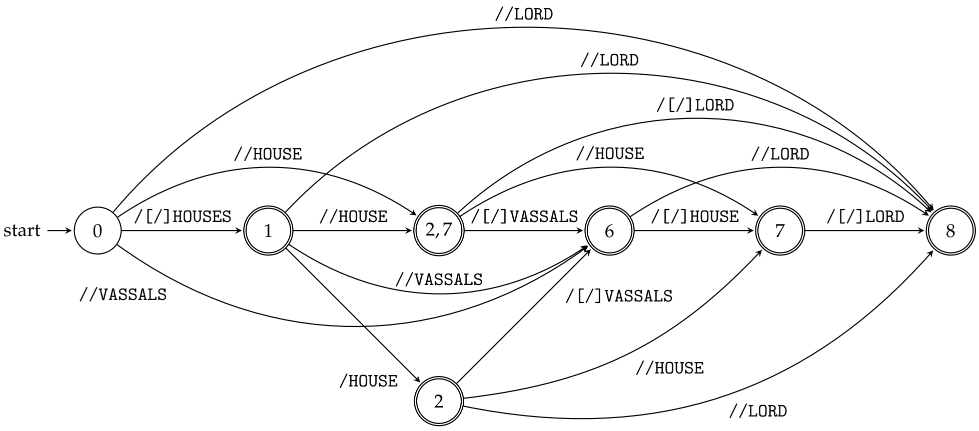


Figure 8. Deterministic tree paths automaton for the string path P = HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) LORD(8) from Example 8.

To obtain TPA for a given XML document, we again use the product construction (see Section 3.1) of the automata that were constructed for individual string paths. Algorithm 6 describes the whole process in detail, and Example 9 demonstrates the result. Again, the evaluation of input queries using TPA is very straightforward, and the final answer is given by the d-subset in the terminal state.

Example 9. *Let D be an XML document from Example 5. The corresponding TPA accepting all $XP^{\{/,//,name-test\}}$ queries, constructed by Algorithm 6, is shown in Figure 9. Again, we note that transition rules $\delta(p,\texttt{/[/]LABEL}) = q$ represent two transitions leading from the state p to the state q: $\delta(p,\texttt{/LABEL}) = q$ and $\delta(p,\texttt{//LABEL}) = q$.*

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

1. For all $P_i \in P(T)$, construct a finite state automaton $M_i = (Q_i, A_i, \delta_i, 0, F_i)$ accepting all $XP^{\{/,//,name-test\}}$ queries of P_i using Algorithm 5.
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 the product construction (union).

Algorithm 6: Construction of the tree paths automaton for an XML document D.

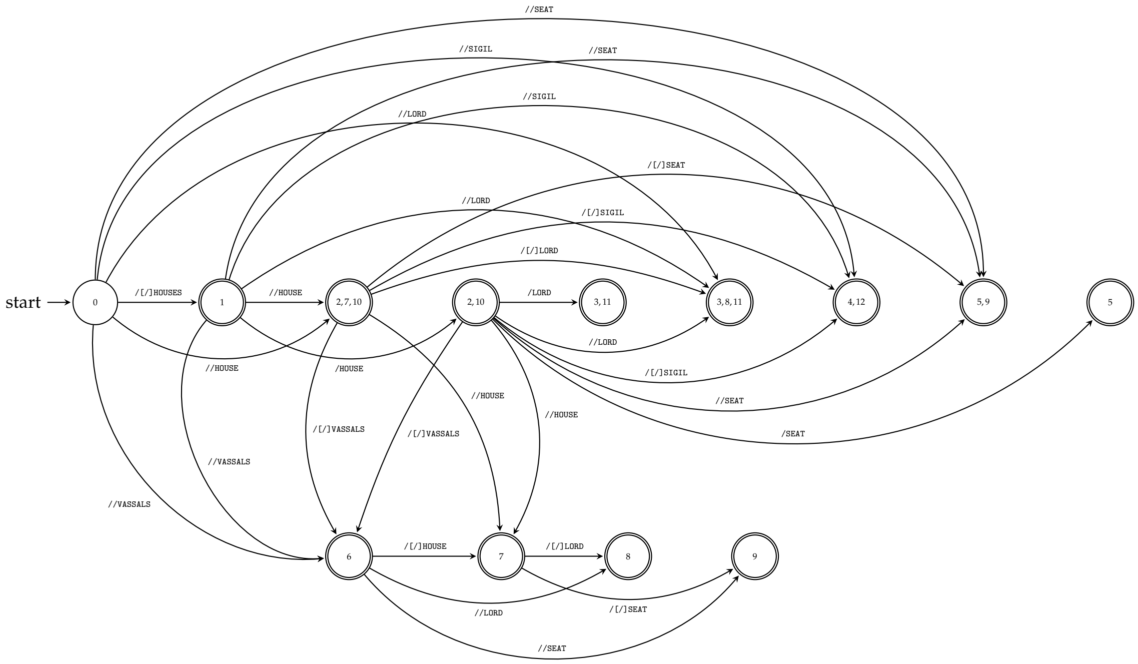


Figure 9. Deterministic tree paths automaton from Example 9.

5. Discussion of Time and Space Complexities

5.1. Tree String Paths Automaton

TSPA efficiently supports the evaluation of all $XP^{\{/,name-test\}}$ queries of an XML document D. The number of such queries is linear in the number of nodes of the XML tree model $T(D)$. For an input query Q of size m, TSPA obviously performed the searching in time $\mathcal{O}(m)$ and does not depend on the size of the original document.

More precisely, the evaluation process naturally consists of two phases: searching phase (i.e., finding the state of the TSPSA that contains the answer in its d-subset) and answering phase (i.e., returning the answer to the user). Therefore, the whole input query Q is evaluated in time $\mathcal{O}(m + k)$, where k is the number of nodes in the XML document D satisfying the query Q. In practice the number of such nodes is expected to be much smaller than the size of the XML document.

Theorem 2. *Let D and $T(D)$ be an XML document and its corresponding XML tree model, respectively. The number of states of the deterministic TSPA $M = (Q, A(D), \delta, 0, F)$, constructed by Algorithm 2, is less than or equal to n, i.e., $|Q| \leq n$, where n is the number of nodes of $T(D)$.*

Proof. Each state of a deterministic TSPA corresponds to an answer of a single query. For an XML tree model $T(D) = (N, E)$ with n nodes, the maximal number of such queries being n, the case is as follows:

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

In all other cases, the number of different queries is strictly less than n. □

Theorem 3. *Let D and $T(D)$ be an XML document and its corresponding XML tree model, respectively. The number of transitions of the deterministic TSPA $M = (Q, A(D), \delta, 0, F)$, constructed by Algorithm 2, is equal to the number of states minus one.*

Proof. TSPA is an acyclic (tree-like) finite state automaton. Each state (except the initial) has exactly one incoming transition. □

5.2. Tree String Path Subsequences Automaton

TSPSA efficiently supports the evaluation of all $XP^{\{//, name-test\}}$ queries of an XML document D. The runtime for a query of length m clearly becomes $\mathcal{O}(m)$ and does not depend on the size of the document D. Again, considering also the answering phase, the whole input query Q is evaluated in time $\mathcal{O}(m + k)$, where k is the number of nodes in the XML document D satisfying the query Q. In practice the number of such nodes is expected to be much smaller than the size of the XML document.

The number of linear XPath queries using the //-axis only is exponential in the number of nodes of the XML tree model. For example, consider just a linear XML tree model T with n nodes. The number of $XP^{\{//, name-test\}}$ queries is $\mathcal{O}(2^n)$, which is determined by the following deduction: There are $\binom{n}{i}$ combinations of i elements $(1 \leq i \leq n)$. Therefore, the exact number of all possible $XP^{\{//, name-test\}}$ queries is given by the following formula:

$$\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = \sum_{j=1}^{n} \binom{n}{j} = 2^n - 1$$

Each state of TSPSA corresponds to an answer of a single query or a collection of queries. Although the number of different queries accepted by TSPSA is exponential, usually many queries are equivalent (i.e., their result sets of elements are equal). Therefore, the equivalence problem of queries is closely related to the problem of the determination of the number of states of TSPSA. 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 the TSPSA 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 a TSPSA as a size of DASG for a set of strings (see [40,41]). For k strings of length h, the number of states can be trivially bounded by $\mathcal{O}(h^k)$, i.e., the size of a product of k automata with $\mathcal{O}(h)$ states. Therefore, the number of transitions of TSPSA is bounded by $\mathcal{O}(|A(D)|h^k)$. The lower bound for $k > 2$ strings is not known, while Crochemore and Troníček in [38] showed that $\Omega(h^2)$ states are required for $k = 2$ in the worst case.

However, considering the XML index problem, the set of strings is rather specific. Thanks to the branching tree structure, we can expect common prefixes in the set of strings, i.e., a lesser number of states (and transitions) in the resulting automaton. In the context of the XML index problem, k is a number of leaves in an XML tree model, and h is its height.

When space is more crucial, we do not need to combine the subsequence automata and just traverse them simultaneously. Finally, we return the union of resulting d-subsets of the automata that accept the input query as the answer. Given a query of length m, this approach obviously works in $\mathcal{O}(k \cdot m)$ time complexity and $\mathcal{O}(h \cdot k)$ space complexity. For parallel systems, each subsequence automaton can be handled by a different computing node.

For a common XML document (XML with the level (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 of the space complexity is $\mathcal{O}(h \cdot 2^k)$. The necessary definitions and formal proof follow.

Definition 9 (Level property). *Let $T = (N, E)$ be a labeled 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 state 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 4. *Let D be an XML document and $T(D)$ be its XML tree model satisfying the l-property with height h and k leaves. The number of states of deterministic TSPSA constructed for the XML document D by Algorithm 4 is $\mathcal{O}(h \cdot 2^k)$.*

Proof. There are k string paths in $T(D)$, for which we construct a set S of k deterministic subsequence automata of no more than h states each (due to the 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 TSPSA M for T.

Due to the l-property of T, it holds that: The target state of a transition labeled with $a \in A(D)$ 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 the 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 $2 + 2^{k-1} \cdot (h-1) + 2^{k-2} \cdot (h-2)$. □

Theorem 5. *Let D be an XML document and $T(D)$ be its XML tree model satisfying the l-property with height h and k leaves. The number of transitions of deterministic TSPSA constructed for the XML document D by Algorithm 4 is $\mathcal{O}(|A(D)| h \cdot 2^k)$.*

Proof. The maximum possible number of transitions leading from each state is $|A(D)|$. □

5.3. Tree Paths Automaton

TPA is designed to efficiently evaluate all $XP^{\{/,//,name-test\}}$ queries of an XML document. The number of such queries is exponential in the number of nodes of the XML tree model. However, the running time for searching a query answer 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.

Again, considering also the answering phase, the whole input query Q is evaluated in time $\mathcal{O}(m + k)$, where k is the number of nodes in the XML document D satisfying the query Q. In practice the number of such nodes is expected to be much smaller than the size of the XML document.

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 a maximum of $\mathcal{O}(|M_i|_{max})$ states). However, this is the asymptotic upper bound, and we note that the size of the index is according to our experimental results usually much smaller.

Again when space is more crucial, we do not need to combine individual automata and just traverse them simultaneously. In parallel systems, each automaton can be handled by a different computing node.

For a common XML document (XML with the 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, as in the case for TSPSA, $\mathcal{O}(h \cdot 2^k)$.

Theorem 6. *Let D be an XML document and $T(D)$ be its XML tree model satisfying the l-property with height h and k leaves. The number of states of deterministic TPA constructed for the XML document D by Algorithm 6 is $\mathcal{O}(h \cdot 2^k)$.*

Proof. The similar arguments of the proof of Theorem 4 hold here, as well. □

Theorem 7. *Let D be an XML document and $T(D)$ be its XML tree model satisfying the l-property with height h and k leaves. The number of transitions of deterministic TPA constructed for the XML document D by Algorithm 6 is $\mathcal{O}(|A(D)|h \cdot 2^k)$.*

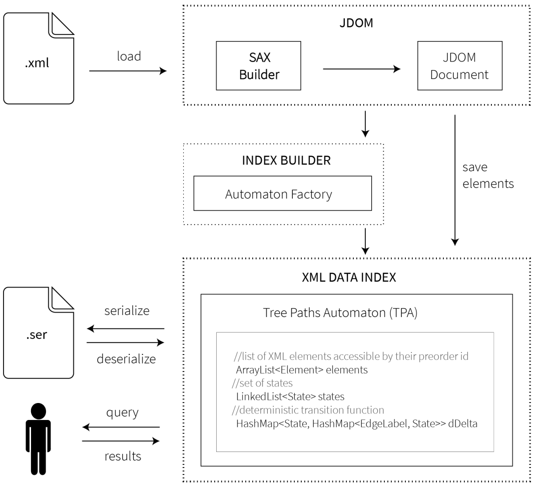
Proof. The maximal number of transitions leading from each state is $2|A(D)|$: $|A(D)|$ transitions for descendant-or-self axis and $|A(D)|$ transitions for child axis. □

6. Experimental Evaluation

This section explores the performance of one of the proposed methods for XML data indexing. Since TPA is designed to process the largest fragment of XPath queries and covers the power of both TSPA and TSPSA, we focused on its experimental evaluation only.

6.1. System Architecture

The XML index software was developed using Java SE, JDK 8u73 in the NetBeans IDE 8.1 and is designed as the Java Class Library called `tpalib` (see the Supplementary Materials for source codes and examples of usage). The system architecture of the `tpalib` is illustrated in Figure 10. Basically, the library consists of three virtual parts called JDOM (Java-based Document Object Model for XML), Index Builder and XML Data Index.



```
//list of XML elements accessible by their preorder id
 ArrayList<Element> elements
//set of states
 LinkedList<State> states
//deterministic transition function
 HashMap<State, HashMap<EdgeLabel, State>> dDelta
```

Figure 10. System architecture of `tpalib`. SAX, Simple API for XML; JDOM, Java-based Document Object Model for XML.

JDOM is used to load and parse an input XML document. During this process, the XML elements are stored in TPA as `org.jdom2.Element` objects and are used later during query evaluation so that the appropriate subset is returned to a user. Index Builder constructs the automaton (some software optimization were used in the implementation; see Chapter 7 in [42] for details). The resulting index is stored as `TreePathsAutomaton` object containing a list of TPA states, transitions and XML elements.

6.2. Experimental Setup

Our experiments were conducted under the environment of an Intel Core i7 CPU @ 2.00 GHz, 8.0 GB RAM and 240-GB SSD disk with the Windows 10 operating system running.

6.2.1. Datasets

For our experimental evaluation, we selected the datasets shown in Table 1. The XML Benchmark (XMark) datasets D_1, D_2, D_3, D_4, D_5 and D_6 were generated by `xmlgen` [43] using scaling factors 0, 0.005, 0.01, 0.1, 0.5 and 1, respectively. XMark is a synthetic on-line auction dataset. It is a single record with a very large and fairly complicated tree structure. It is also relatively deep with a maximal depth of 11 and an average depth of 4.5.

Table 1. Characteristics of the datasets.

Key	Dataset Name	File Name	Size (MB)	# of Elements	# of Leaves	Max-Depth	Avg-Depth
D_1	XMark	XMark-f0.xml	0.03	382	247	10	4.64
D_2	XMark	XMark-f0.005.xml	0.50	8518	6211	11	4.50
D_3	XMark	XMark-f0.01.xml	1.16	17,132	12,504	11	4.51
D_4	XMark	XMark-f0.1.xml	11.60	167,865	122,026	11	4.55
D_5	XMark	XMark-f0.5.xml	57.64	832,911	605,546	11	4.55
D_6	XMark	XMark-f1.xml	115.75	1,666,315	1,211,774	11	4.55
D_7	DBLP	dblp.xml	130.73	3,332,130	3,000,839	6	2.90

The DBLP dataset D_7 is a real-world dataset obtained from the University of Washington XML repository [44]. It is a database of bibliographic information of computer science journals and conference proceedings containing over 3.3 million of elements. It is a shallow and wide document with high similarity, maximal depth of six and an average depth of 2.9.

6.2.2. Queries

The queries used for our experiments are shown in Tables 2 and 3. For each dataset we used nine sample queries that differ in length and axes used. The queries were split into categories depending on the type of axis used. First, the Q_1–Q_3 queries contain the child axis only; Q_4–Q_6 include the descendant-or-self axis only; and the last Q_7–Q_9 queries use a combination of both axes. The numbers of elements satisfying individual queries in each of the datasets are shown in Table 4.

Table 2. Set of queries for XMark datasets.

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`

Table 3. Set of queries for DBLP dataset.

Key	XPath Query
Q_1	`/dblp/mastersthesis`
Q_2	`/dblp/inproceedings/title`
Q_3	`/dblp/inproceedings/title/sup/sup`
Q_4	`//author`
Q_5	`//inproceedings//booktitle`
Q_6	`//dblp//article//author`
Q_7	`//inproceedings/author`
Q_8	`//dblp/article//title`
Q_9	`//dblp/inproceedings//title/sup/sup`

Table 4. Numbers of elements satisfying the queries in the datasets.

	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	127	5	247	124	6	60	247	5
D_3	1	255	17	488	205	50	120	488	43
D_4	1	2550	210	4815	2139	388	1200	4815	350
D_5	1	12,750	1235	25,414	10,455	2357	6000	25,414	2099
D_6	1	25,500	2335	50,269	20,946	4570	12,000	50,269	4041
D_7	5	212,273	1	716,488	212,273	221,465	491,783	111,609	1

6.3. Performance Analysis

Table 5 shows the experimental results on the index size and construction time for the datasets. The space requirements of the index structure were measured using the size of the file with the serialized `TreePathsAutomaton` Java object. Since every `TreePathsAutomaton` object contains besides automaton data also all XML elements (`org.jdom2.Element` objects) to be used later during query evaluation, the size of the index is bigger than the original XML document. However, the results suggest that the ratio of the index size to the 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.

Table 5. Experimental results on the index size and construction time.

Key	Index Size (MB)	Index/XML Size (Ratio)	# of States	# of Transitions	Construction Time
D_1	0.08	2.6	306	1189	0.15 s
D_2	1.35	2.7	859	2801	1.6 s
D_3	2.67	2.3	997	3164	4.41 s
D_4	25.97	2.2	1194	3656	5 min
D_5	128.67	2.2	1220	3714	2 h
D_6	256.43	2.2	1220	3714	10 h (approx.)
D_7	370.25	2.8	169	391	20 h (approx.)

In the table, we also show the number of states and transitions of resulting automaton. These numbers are obviously mainly influenced by the structure of the XML document and by its size. For example, because of high similarity in data, the largest file (the DBLP dataset) can be indexed by the automaton with the smallest number of states (i.e., 169) and transitions (i.e., 391).

At the moment, the major drawback of the approach seems to be the index construction time. For the datasets with the size >100 MB, it took around 15 h to construct the index. This is definitely the crucial part for our future software optimization. We also plan to develop parallel building algorithm to speed up the construction. On the other hand, once the index is constructed, it is stored (serialized) as `TreePathsAutomaton` Java object. Therefore, next time the index is ready to use within a minute (depends on the size of the serialized object).

6.3.1. Performance on Query Processing

The analysis of the performance of the query processing was conducted in comparison with a well-known reference implementation called Saxon [45] and Xalan [46]. Our measurements reflect query processing time only. Hence, document loading cost has been excluded from the measurements.

Figure 11 summarizes the experimental results of TPA, Saxon and Xalan on the XMark datasets D_1–D_6. The graph is plotted using the logarithmic scale. The x-axis represents the datasets, while the y-axis shows the response time in milliseconds. We used light dashed lines to display Saxon results, whereas the TPA score is depicted as dark solid lines and Xalan results are displayed using light solid lines.

There appears to be a clear upward pattern in the query processing time with the growing size of datasets. This is most likely caused by the higher number of elements that need to be returned as the query answer. Overall, the sample queries achieve a better response time using our proposed indexing method.

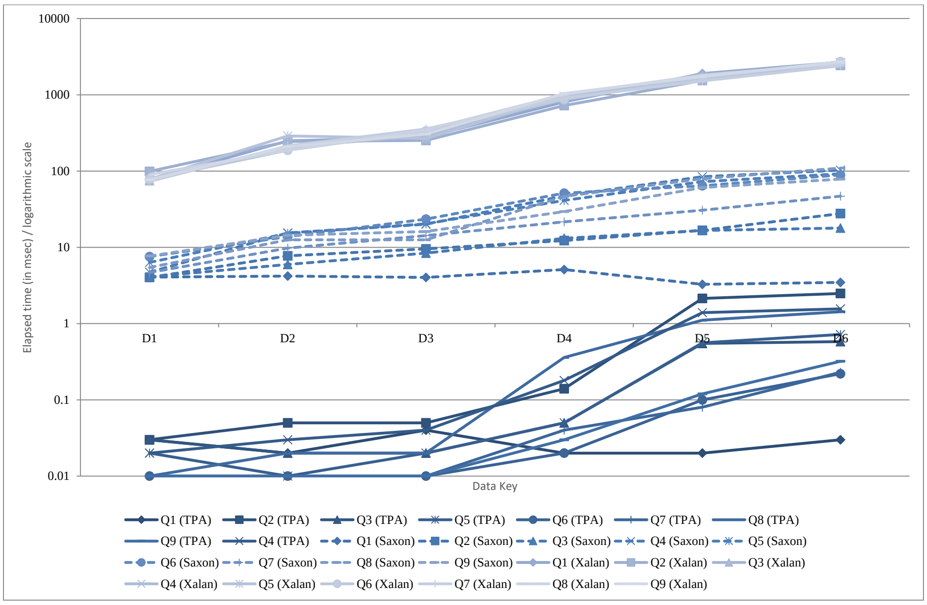
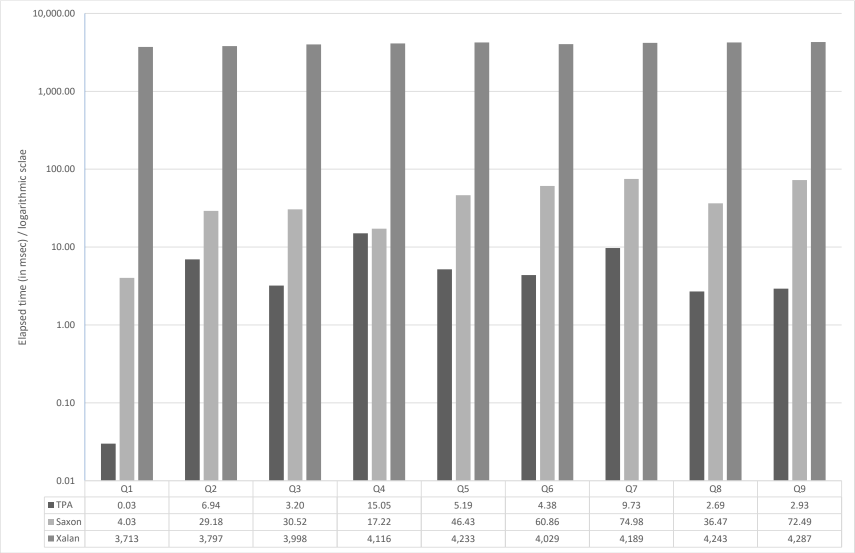


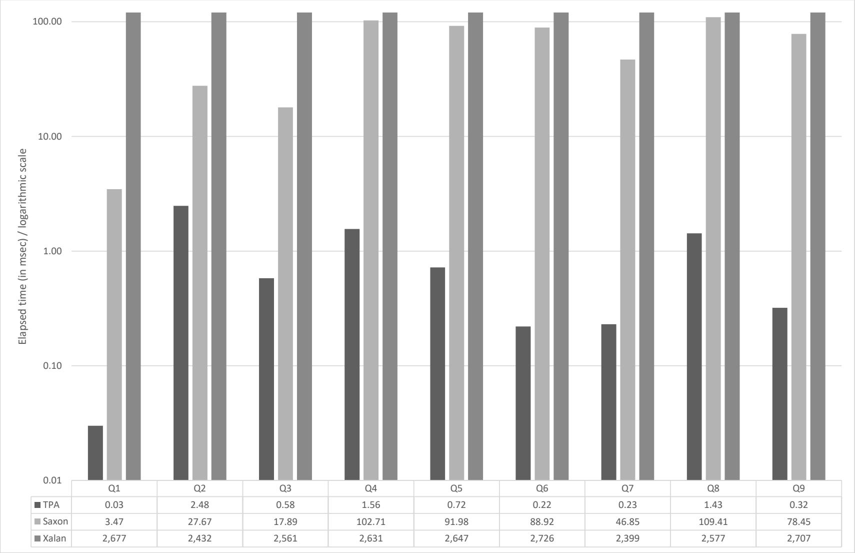
Figure 11. Performance comparison of TPA, Saxon and Xalan (logarithmic scale).

Figures 12 and 13 show the experimental results of TPA, Saxon and Xalan on DBLP and XMark (D_6) datasets, respectively. Both graphs are plotted using the logarithmic scale. The tables show elapsed time in milliseconds for individual queries. Again, TPA achieves a better response time on all sample queries. The DBLP dataset (and query Q_2 =`/dblp/inproceedings/title`) was also used for experimental evaluation of CTree (see [3]). CTree evaluated the query Q_2 in 50 ms, whereas TPA evaluated the query in 7 ms.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
■ TPA	0.03	6.94	3.20	15.05	5.19	4.38	9.73	2.69	2.93
■ Saxon	4.03	29.18	30.52	17.22	46.43	60.86	74.98	36.47	72.49
■ Xalan	3,713	3,797	3,998	4,116	4,233	4,029	4,189	4,243	4,287

Figure 12. Performance comparison of TPA, Saxon and Xalan on DBLP dataset (logarithmic scale).



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
■ TPA	0.03	2.48	0.58	1.56	0.72	0.22	0.23	1.43	0.32
■ Saxon	3.47	27.67	17.89	102.71	91.98	88.92	46.85	109.41	78.45
■ Xalan	2,677	2,432	2,561	2,631	2,647	2,726	2,399	2,577	2,707

Figure 13. Performance comparison of TPA, Saxon and Xalan on XMark (D_6) dataset (logarithmic scale).

7. Conclusions

In this paper, we proposed some new methods for indexing XML documents using the theory of formal languages and automata. These methods are simple and allow one to efficiently process a small subset of XPath. In particular, we focused on the two common axes, child (i.e., /) and descendant-or-self (i.e., //). In the future, we would like to use the similar approach to build automata that support other XPath axes (e.g, an automaton supporting parent and ancestor axis). Therefore, having an XML data structure, our methods can be used efficiently as auxiliary data structures that enable answering a particular set of queries.

First, we presented the Tree String Paths Automaton (TSPA) and the Tree String Path Subsequences Automaton (TSPSA), which are aimed at assisting in the evaluation of path queries with either the child or descendant-or-self axis only. Then, we introduced the Tree Paths Automaton (TPA), which is designed to process XPath queries using any combination of child and descendant-or-self axes.

Given an XML document D with its corresponding XML tree model $T(D)$, the tree is preprocessed, and an index, which is a finite state 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 in the case of TSPSA and TPA exponential, the size of these indexes (number of states) is for a common XML document with l-property $\mathcal{O}(h \cdot 2^k)$, where h is the height of the tree $T(D)$ and k is the number of its leaves. In practice, our experimental results suggest that the size of the index stays linear since the index files were only about 2.5-times larger than the size of the original documents.

There is also a number of interesting open problems that we hope to explore in the future:

- developing a parallel building algorithm to speed up the construction phase,
- developing an incremental building algorithm for our automata-based indexes to efficiently adapt their structure to ever-changing XML data sources,
- adapting our indexing methods to be able to support multiple XML documents,
- extending our methods to support more complex queries (e.g., including attributes, wildcards, branching, etc.).

Supplementary Materials: The supplementary material is available online at http://www.mdpi.com/2078-2489/9/1/12/s1.

Acknowledgments: This research has been partially supported by the grant from Czech Technical University in Prague, Project No. SGS17/209/OHK3/3T/18.

Author Contributions: Both authors performed the research and experiments. Eliska Sestakova wrote the paper. Both authors have read and approved final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DASG	Directed Acyclic Subsequence Graph
JDOM	Java-based Document Object Model for XML
SAX	Simple API for XML
TPA	Tree Paths Automaton
TSPA	Tree String Paths Automaton
TSPSA	Tree String Path Subsequences Automaton
XML	Extensible Markup Language
XPath	XML Path Language
W3C	World Wide Web Consortium

References

1. Goldman, R.; Widom, J. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*; Technical Report; Stanford University: Stanford, CA, USA, 1997.

2. Pettovello, P.M.; Fotouhi, F. MTree: An XML XPath Graph Index. In Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, 23–27 April 2006; ACM: New York, NY, USA, 2006; pp. 474–481.
3. Zou, Q.; Liu, S.; Chu, W.W. Ctree: A compact tree for indexing XML data. In Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management, Washington, DC, USA, 12–13 November 2004; pp. 39–46.
4. Tang, N.; Yu, J.; Ozsu, M.; Wong, K.F. Hierarchical indexing approach to support XPath queries. In Proceedings of the IEEE 24th International Conference on Data Engineering, Cancun, Mexico, 7–12 April 2008; pp. 1510–1512.
5. Kaushik, R.; Bohannon, P.; Naughton, J.F.; Korth, H.F. Covering indexes for branching path queries. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 3–6 June 2002; ACM: New York, NY, USA, 2002; pp. 133–144.
6. Milo, T.; Suciu, D. Index structures for path expressions. In *Database Theory—ICDT '99*; Beeri, C., Buneman, P., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1540, pp. 277–295.
7. Rao, P.; Moon, B. PRIX: Indexing and querying XML using prufer sequences. In Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA, 2 April 2004; pp. 288–299.
8. Zhang, B.; Wang, W.; Wang, X.; Zhou, A. AB-Index: An Efficient Adaptive Index for Branching XML Queries. In *Advances in Databases: Concepts, Systems and Applications*; Kotagiri, R., Krishna, P.R., Mohania, M., Nantajeewarawat, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4443, pp. 988–993.
9. Chung, C.W.; Min, J.K.; Shim, K. APEX: An adaptive path index for XML data. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 3–6 June 2002; ACM: New York, NY, USA, 2002; pp. 121–132.
10. Li, Q.; Moon, B. Indexing and querying XML data for regular path expressions. In Proceedings of the 27th International Conference on Very Large Data Bases, Roma, Italy, 11–14 September 2001; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001; pp. 361–370.
11. Wang, H.; Park, S.; Fan, W.; Yu, P.S. ViST: A dynamic index method for querying XML data by tree structures. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; ACM: New York, NY, USA, 2003; pp. 110–121.
12. Wang, H.; Meng, X. On the sequencing of tree structures for XML indexing. In Proceedings of the IEEE 21st International Conference on Data Engineering, Tokoyo, Japan, 5–8 April 2005; pp. 372–383.
13. Tung, H.D.T.; Luong, D.D. An improved indexing method for Xpath queries. *Indian J. Sci. Technol.* **2016**, *9*, doi:10.17485/ijst/2016/v9i31/92731.
14. Tatarinov, I.; Viglas, S.D.; Beyer, K.; Shanmugasundaram, J.; Shekita, E.; Zhang, C. Storing and querying ordered XML using a relational database system. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, WI, USA, 3–6 June 2002; ACM: New York, NY, USA, 2002; pp. 204–215.
15. Kha, D.D.; Yoshikawa, M.; Uemura, S. An XML indexing structure with relative region coordinate. In Proceedingsof the IEEE 17th International Conference on Data Engineering, Heidelberg, Germany, 2–6 April 2001; pp. 313–320.
16. Clark, J.; DeRose, S. *XML Path Language (XPath)*, version 1.0.; W3C, 16 November 1999. Available online: https://www.w3.org/TR/xpath/ (accessed on 6 January 2018).
17. DeRose, S. *XML Pointer Language (XPointer)*, version 1.0.; W3C, 2002. Available online: https://www.w3.org/TR/WD-xptr (accessed on 6 January 2018).
18. DeRose, S. *XML Linking Language (XLink)*, version 1.1.; W3C, 2010. Available online: https://www.w3.org/TR/xlink11/ (accessed on 6 January 2018).
19. Mandhani, B.; Suciu, D. Query Caching and view selection for XML databases. In Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, 30 August–2 September 2005; pp. 469–480.
20. Šestáková, E.; Janoušek, J. Tree string path subsequences automaton and its use for indexing XML documents. In *International Symposium on Languages, Applications and Technologies*; Springer: Cham, Switzerland, 2015; pp. 171–181.

21. Šestáková, E.; Janoušek, J. Indexing XML documents using tree paths automaton. In *OASIcs-OpenAccess Series in Informatics*; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Volume 56.
22. Crochemore, M.; Hancart, C.; Lecroq, T. *Algorithms on Strings*; Cambridge University Press: Cambridge, UK, 2007.
23. Crochemore, M.; Rytter, W. *Text Algorithms*; Oxford University Press: Oxford, UK, 1994.
24. Melichar, B.; Janoušek, J.; Flouri, T. *Introduction to Arbology*; Czech Technical University: Prague, Czech Republic.
25. Su-Cheng, H.; Chien-Sing, L. Node labeling schemes in XML query optimization: A survey and trends. *IETE Tech. Rev.* **2009**, *26*, 88–100.
26. Schwentick, T. Automata for XML—A survey. *J. Comput. Syst. Sci.* **2007**, *73*, 289–315.
27. Brüggemann-Klein, A.; Wood, D. *Regular Tree Languages over Non-Ranked Alphabets*, version 0.3.; 19 April 1998. Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.5397 (accessed on 6 January 2018).
28. Neven, F. Automata theory for XML researchers. *ACM Sigmod Rec.* **2002**, *31*, 39–46.
29. Neven, F. Automata, logic, and XML. In *Computer Science Logic*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 671–711.
30. Diao, Y.; Fischer, P.; Franklin, M.J.; To, R. Yfilter: Efficient and scalable filtering of XML documents. In Proceedings of the IEEE 18th International Conference on Data Engineering, San Jose, CA, USA, 26 February–1 March 2002; pp. 341–342.
31. Green, T.J.; Gupta, A.; Miklau, G.; Onizuka, M.; Suciu, D. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.* **2004**, *29*, 752–788.
32. Liu, P.; Sun, W.; Zhang, J.; Zheng, B. An automaton-based index scheme supporting twig queries for on-demand XML data broadcast. *J. Parallel Distrib. Comput.* **2015**, *86*, 82–97.
33. Chan, C.Y.; Garofalakis, M.; Rastogi, R. Re-tree: An efficient index structure for regular expressions. *VLDB J.* **2003**, *12*, 102–119.
34. Segoufin, L.; Vianu, V. Validating streaming XML documents. In Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Madison, WI, USA, 3–5 June 2002; ACM: New York, NY, USA, 2002; pp. 53–64.
35. Murata, M.; Lee, D.; Mani, M.; Kawaguchi, K. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Int. Technol.* **2005**, *5*, 660–704.
36. Fan, W.; Geerts, F.; Jia, X.; Kementsietsidis, A. Rewriting regular XPath queries on XML views. In Proceedings of the IEEE 23rd International Conference on Data Engineering, Istanbul, Turkey, 15–20 April 2007; pp. 666–675.
37. Rabin, M.O.; Scott, D. Finite automata and their decision problems. *IBM J. Res. Dev.* **1959**, *3*, 114–125.
38. Crochemore, M.; Troníček, Z. On the Size of DASG for Multiple Texts. In *String Processing and Information Retrieval*; Laender, A., Oliveira, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2476, pp. 58–64.
39. Hoshino, H.; Shinohara, A.; Takeda, M.; Arikawa, S. Online construction of subsequence automata for multiple texts. In Proceedings of the SPIRE Seventh International Symposium on String Processing and Information Retrieval, A Curuna, Spain, 29 September 2000; pp. 146–152.
40. Baeza-Yates, R.A. Searching subsequences. *Theor. Comput. Sci.* **1991**, *78*, 363–376.
41. Crochemore, M.; Melichar, B.; Troníček, Z. Directed acyclic subsequence graph—Overview. *J. Discret. Algorithms* **2003**, *1*, 255–280.
42. Šestáková, E. Indexing XML documents. Master's Thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2015.
43. Schimdt, A.; Busse, R.; Carey, M.; Florescu, D.; Kersten, M.; Manolescu, I.; Waas, F. *XMark–An XML Benchmark Project*; 2003. Available online: http://www.xml-benchmark.org/ (accessed on 6 January 2018).
44. UW XML Repository. Available online: http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata /www/repository.html (accessed on 6 January 2018).

45. Saxonica. *Saxon–The XSLT and XQuery Processor*; Available online: http://saxon.sourceforge.net/ (accessed on 6 January 2018).
46. Apache Software Foundation. *Apache Xalan Project*, version 2.7.1.; Available online: http://xalan.apache.org/ (accessed on 6 January 2018)..