Automata Approach to XML Data Indexing

Eliška Šestáková †,*, 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 \( O(m) \) 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., \( O(2.62^n) \) 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 deal with different problems than XML data indexing (see [26]). For instance, finite state automata 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, $\delta$ 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 \( \text{depth}(n) \), 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).

```xml
<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 \( \text{HOUSES} \) is the most outer element. A start-tag of this element is of the form \(<\text{HOUSES}>\), whereas the corresponding end-tag, indicating the end of the element, is \(<</\text{HOUSES}>\). Therefore, the content between and including the tags \(<\text{HOUSES}>\) and \(<</\text{HOUSES}>\) constitutes the \( \text{HOUSES} \) element.

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

Another key construct of an XML document is comprised by attributes. For instance, \(<\text{HOUSE name}="Stark">\) indicates that the value of the \text{name} attribute of that particular \( \text{HOUSE} \) 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 \( Q_1 = /\text{HOUSES}/\text{HOUSE}/\text{LORD} \). The query selects all \( \text{LORD} \) elements having \( \text{HOUSE} \) as the parent element and \( \text{HOUSES} \) as the grandparent element, whereas \( \text{HOUSES} \) 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 \( Q_2 = \text{/VASSALS/LORD} \). The query selects all \text{LORD} elements having 
\text{VASSALS} as an ancestor element, whereas the \text{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 \( Q_3 = \text{//HOUSE/LORD} \). The query selects all \text{LORD} elements having 
\text{HOUSE} as the parent element, whereas the \text{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 \((\text{label}, \text{id})\) where \text{label} and \text{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 
\( A(D) = \{ \text{HOUSES, HOUSE, LORD, SIGIL, SEAT, VASSALS} \} \). 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 \( P = n_1n_2 \ldots n_t \) \((t \leq h)\) of 
\( T \) is a linear path leading from the root \( r = n_1 \) to the leaf \( n_t \).
Definition 3 (String path alphabet). Let $P$ be a string path of some XML tree model. A string path alphabet $A(P)$, 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.

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

The corresponding string path alphabets are as follows:

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

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.

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

**Data:** A string path \( P = n_1n_2\ldots n_{|P|} \).

**Result:** DFSA \( M = (Q, A, \delta, 0, F) \) accepting all \( XP\{/\text{name-test}\} \) queries of \( P \).

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

![Figure 2](image)

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.](image)

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

**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 $XP_{/,
\text{name}\_test}$ queries of an XML document $D$.

1. For all $P_i \in P(T)$, construct a finite state automaton $M_i = (Q, \{a : a \in A(P_i)\}, \delta, 0, F_i)$ accepting all $XP_{/,
\text{name}\_test}$ 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 $XP_{/,
\text{name}\_test}$ queries of the XML document $D$ using the product construction (union).

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 $XP_{/,
\text{name}\_test}$ 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:

$$G = (\{S\}, A(D), \{S \rightarrow SS | /a, \text{such as } a \in A(D)\}, S)$$
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 \( P \) is 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 XML index problem to be another application area of DASG.

Therefore, we propose solving the problem of subsequences for both single and multiple strings is also referred to as the subsequence automaton for each string path \( P \). Let \( P \) and \( O \) be a string path and a set of occurrences of an element label in a string path. A set of occurrences of an element label \( e \) in \( P \) is a totally ordered set \( O_P(e) = \{o \mid o = id(n_i) \land label(n_i) = e, i = 1, 2, \ldots, |P|\} \). The ordering is equal to ordering of element prefix identifiers as natural numbers.

Definition 7 (Set of occurrences of an element label in a string path). Let \( P = n_1n_2 \ldots n_{|P|} \) 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 \( O_P(e) = \{o \mid o = id(n_i) \land label(n_i) = e, i = 1, 2, \ldots, |P|\} \). The ordering is equal to ordering of element prefix identifiers as natural numbers.

Definition 8 (ButFirst). Let \( P \) and \( O_P(e) = \{o_1, o_2, \ldots, o_{|O_P(e)|}\} \) 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(\( O_P(e) \)) = \( \{o_2, \ldots, o_{|O_P(e)|}\} \).

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

**Data:** A string path \( P = n_1n_2 \ldots n_{|P|} \).

**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. \( Q \leftarrow \{q_0, q_1, \ldots, q_{|P|}\} \), \( A \leftarrow \{/a : a \in A(P)\} \), \( F \leftarrow Q \setminus \{q_0\}, q_0 \leftarrow 0 \)
   
   b. \( \forall i, \) where \( i \leftarrow 1, 2, \ldots, |P| \):
      
      i. set state \( q_i \leftarrow O_P(label(n_i)) \),
      ii. add \( \delta(q_{i-1}, /label(n_i)) \) \( \leftarrow q_i \),
      iii. \( O_P(label(n_i)) \leftarrow \text{ButFirst}(O_P(label(n_i))) \).

3. Insert “additional transitions” into the automaton \( M \):

   \( \forall i \in \{0, 1, \ldots, |P| - 1\}, \forall a \in A(P) \):
   
   i. add \( \delta(q_i, /a) \leftarrow q_s, \) if there exists such \( s > i \) where
      
      \( \delta(q_{s-1}, /a) = q_s \land \exists r < s : \delta(q_{r-1}, /a) = q_r \)
   ii. \( \delta(q_i, /a) \leftarrow \emptyset \) otherwise.
Theorem 1. Given a string path $P = n_1n_2\ldots n_{|P|}$, Algorithm 3 correctly constructs a deterministic finite state automaton $M$ accepting all $XP\{/\text{name--test}\}$ queries of $P$.

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

1. If $M$ accepts a string $X$, then $X$ is the $XP\{/\text{name--test}\}$ query of $P$.
2. If $X$ is the $XP\{/\text{name--test}\}$ query of $P$, then $M$ accepts $X$. Assume to the contrary that $X = /\text{X}_1/\text{X}_2\ldots /\text{X}_k$ over the alphabet $A = \{/\text{a} : a \in A(P)\}$ is the $XP\{/\text{name--test}\}$ 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 $/\text{X}_i$ that $M$ has no transition leading from the current state labeled by $/\text{X}_i$.

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 $q_i$ 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 $\delta(q_{s-1}, /\text{a}) = q_s$.

Second, we need to ensure that the state is the closest possible, which is satisfied by the second condition: $\exists r < s : \delta(q_{r-1}, /\text{a}) = q_r$. Therefore, no subsequence is missed.

Thus, $M$ reads $x_1\ldots x_{i-1}$, and the current state of $M$ is $q_i$. 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 (2bi), each state $q_i$ of $M$ corresponds to a node $n_j$ of $P$. Because of (2bi) 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 $x_j$ (the nonexistent part of 3i). Therefore, $x_1\ldots x_i$ is not the $XP\{/\text{name--test}\}$ 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 $XP\{/\text{name--test}\}$ 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.

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

**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 $M_i = (Q_i, \{a \in A(P_i)\}, \delta_i, 0, F_i)$ accepting all $XP[//,name-test]$ queries of $P_i$ using Algorithm 3.
2. Construct the deterministic tree string path subsequences automaton $M = (Q, \{a \in A(D)\}, \delta, 0, Q\{0\})$ accepting all $XP[//,name-test]$ queries of the XML document $D$ using the product construction (union).

**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.

**Algorithm 5:** Construction of the tree paths automaton for a single string path.

**Data:** A string path $P = n_1n_2 \ldots n_k$.

**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 $M = (Q, A_1 \cup A_2, \delta, 0, Q \{0\})$ 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 $d \neq \emptyset$ then
            if $d \notin Q$ then
                $Q = Q \cup \{d\}$;
                $S$.push($d$);
                $\delta(q, a) \leftarrow d$;
            end
        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;
    $\forall a \in A_2 : \delta(q_2, a) \leftarrow \delta_2(q_2, a)$;
end

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 $q$: $\delta(p, /) = q$ and $\delta(p, //) = q$. 


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 $\delta(p, //LABEL) = q$ 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 $P = \text{HOUSES}(1) \text{HOUSE}(2) \text{VASSALS}(6) \text{HOUSE}(7) \text{LORD}(8)$ from Example 8.

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

**Figure 8.** Deterministic tree paths automaton for the string path $P = \text{HOUSES}(1) \text{HOUSE}(2) \text{VASSALS}(6) \text{HOUSE}(7) \text{LORD}(8)$ from Example 8.
Algorithm 6: Construction of the tree paths automaton for an XML document \( D \).

**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 \( XP(/, //, \text{name test}) \) 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 \( XP(/, //, \text{name test}) \) queries of \( P_i \) using Algorithm 5.

2. Construct a deterministic tree paths automaton \( M = (Q, A, \delta, 0, F) \) accepting all \( XP(/, //, \text{name test}) \) queries of the XML document \( D \) using the product construction (union).

---

**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 \( XP(/, \text{name test}) \) 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 \( O(m) \) 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 \( O(m + k) \), 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 \( T(D) \).
The number of transitions of the deterministic TSPA (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 TSPA. That is, if we know the number of unique query answers, we can construct a deterministic automaton answering all queries related to the problem of the determination of the number of states of TSPA. That is, if we know the number of different queries accepted by TSPA is exponential, usually many queries are equivalent.

\[ \forall n_1, n_2 \in N \land n_1 \neq n_2 : \text{label}(n_1) = \text{label}(n_2) \implies \text{parent}(n_1) \neq \text{parent}(n_2) \]

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

TSPA efficiently supports the evaluation of all \( \{ //, \text{name-test} \} \) queries of an XML document \( D \). The runtime for a query of length \( m \) clearly becomes \( O(m) \) 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 \( O(m + k) \), 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 \( \{ //, \text{name-test} \} \) queries is \( O(2^n) \), 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 \( \{ //, \text{name-test} \} \) 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 TSPA corresponds to an answer of a single query or a collection of queries. Although the number of different queries accepted by TSPA 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 number of states of TSPA. 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 TSPA 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 TSPA 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 \( O(h^k) \), i.e., the size of a product of \( k \) automata with \( O(h) \) states. Therefore, the number of transitions of TSPA is bounded by \( O(|A(D)|h^k) \). The lower bound for \( k > 2 \) strings is not known, while Crochemore and Tronček 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 \( O(k \cdot m) \) time complexity and \( O(h \cdot k) \) 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 \( O(h \cdot 2^k) \). 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):
\[
\forall n_1, n_2 \in N \land n_1 \neq n_2 : \text{label}(n_1) = \text{label}(n_2) \implies \text{depth}(n_1) = \text{depth}(n_2)
\]

### 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 \( O(h \cdot 2^k) \).

**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 \( O(|A(D)|h \cdot 2^k) \).

**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\(\{./,./,\text{name=test}\}\) 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., \( O(m) \) for a query of size \( m \).

Again, considering also the answering phase, the whole input query \( Q \) is evaluated in time \( O(m + k) \), 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 \( |M_i|_{\text{max}} \) is the maximum possible size of TPA for a single string path. Therefore, the number of states of TPA for the XML document can be trivially bounded by \( O(|M_i|_{\text{max}}^k) \) (size of a product of \( k \) automata with a maximum of \( O(|M_i|_{\text{max}}) \) 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, $O(h \cdot 2^k)$.

**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 $O(h \cdot 2^k)$.

**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 $O(|A(D)|h \cdot 2^k)$.

**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.

![Figure 10. System architecture of tpalib. SAX, Simple API for XML; JDOM, Java-based Document Object Model for XML.](image)

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.

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

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.

<table>
<thead>
<tr>
<th>Key</th>
<th>XPath Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>/site/open_auctions</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>/site/people/person/name</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>/site/regions/europe/item/description/parlist/listitem/text/emph</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>//person/watch</td>
</tr>
<tr>
<td>$Q_5$</td>
<td>//regions/mail/date</td>
</tr>
<tr>
<td>$Q_6$</td>
<td>//site/regions/europe//description//listitem//text/emph</td>
</tr>
<tr>
<td>$Q_7$</td>
<td>/site/open_auction</td>
</tr>
<tr>
<td>$Q_8$</td>
<td>//people/person/watch</td>
</tr>
<tr>
<td>$Q_9$</td>
<td>//regions/europe/item//parlist/listitem//text/emph</td>
</tr>
</tbody>
</table>
Table 3. Set of queries for DBLP dataset.

<table>
<thead>
<tr>
<th>Key</th>
<th>XPath Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q₁</td>
<td>/dblp/mastersthesis</td>
</tr>
<tr>
<td>Q₂</td>
<td>/dblp/inproceedings/title</td>
</tr>
<tr>
<td>Q₃</td>
<td>/dblp/inproceedings/title/sup/sup</td>
</tr>
<tr>
<td>Q₄</td>
<td>//author</td>
</tr>
<tr>
<td>Q₅</td>
<td>//inproceedings//booktitle</td>
</tr>
<tr>
<td>Q₆</td>
<td>//dblp//article//author</td>
</tr>
<tr>
<td>Q₇</td>
<td>//inproceedings/author</td>
</tr>
<tr>
<td>Q₈</td>
<td>//dblp//article//title</td>
</tr>
<tr>
<td>Q₉</td>
<td>//dblp/inproceedings//title//sup/sup</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Q₁</th>
<th>Q₂</th>
<th>Q₃</th>
<th>Q₄</th>
<th>Q₅</th>
<th>Q₆</th>
<th>Q₇</th>
<th>Q₈</th>
<th>Q₉</th>
</tr>
</thead>
<tbody>
<tr>
<td>D₁</td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D₂</td>
<td>1</td>
<td>127</td>
<td>5</td>
<td>247</td>
<td>124</td>
<td>6</td>
<td>60</td>
<td>247</td>
<td>5</td>
</tr>
<tr>
<td>D₃</td>
<td>1</td>
<td>255</td>
<td>17</td>
<td>488</td>
<td>205</td>
<td>50</td>
<td>120</td>
<td>488</td>
<td>43</td>
</tr>
<tr>
<td>D₄</td>
<td>1</td>
<td>2550</td>
<td>210</td>
<td>4815</td>
<td>2139</td>
<td>388</td>
<td>1200</td>
<td>4815</td>
<td>350</td>
</tr>
<tr>
<td>D₅</td>
<td>1</td>
<td>12,750</td>
<td>1235</td>
<td>25,414</td>
<td>10,455</td>
<td>2357</td>
<td>6000</td>
<td>25,414</td>
<td>2099</td>
</tr>
<tr>
<td>D₆</td>
<td>1</td>
<td>25,500</td>
<td>2335</td>
<td>50,269</td>
<td>20,946</td>
<td>4570</td>
<td>12,000</td>
<td>50,269</td>
<td>4041</td>
</tr>
<tr>
<td>D₇</td>
<td>5</td>
<td>212,273</td>
<td>1</td>
<td>716,488</td>
<td>212,273</td>
<td>221,465</td>
<td>491,783</td>
<td>111,609</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Key</th>
<th>Index Size (MB)</th>
<th>Index/XML Size (Ratio)</th>
<th># of States</th>
<th># of Transitions</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>D₁</td>
<td>0.08</td>
<td>2.6</td>
<td>306</td>
<td>1189</td>
<td>0.15 s</td>
</tr>
<tr>
<td>D₂</td>
<td>1.35</td>
<td>2.7</td>
<td>859</td>
<td>2801</td>
<td>1.6 s</td>
</tr>
<tr>
<td>D₃</td>
<td>2.67</td>
<td>2.3</td>
<td>997</td>
<td>3164</td>
<td>4.41 s</td>
</tr>
<tr>
<td>D₄</td>
<td>25.97</td>
<td>2.2</td>
<td>1194</td>
<td>3656</td>
<td>5 min</td>
</tr>
<tr>
<td>D₅</td>
<td>128.67</td>
<td>2.2</td>
<td>1220</td>
<td>3714</td>
<td>2 h</td>
</tr>
<tr>
<td>D₆</td>
<td>256.43</td>
<td>2.2</td>
<td>1220</td>
<td>3714</td>
<td>10 h (approx.)</td>
</tr>
<tr>
<td>D₇</td>
<td>370.25</td>
<td>2.8</td>
<td>169</td>
<td>391</td>
<td>20 h (approx.)</td>
</tr>
</tbody>
</table>

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).](image)

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.
<table>
<thead>
<tr>
<th></th>
<th>TPA</th>
<th>Saxon</th>
<th>Xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.03</td>
<td>3.03</td>
<td>3,713</td>
</tr>
<tr>
<td>Q2</td>
<td>6.94</td>
<td>29.18</td>
<td>3,797</td>
</tr>
<tr>
<td>Q3</td>
<td>3.20</td>
<td>30.52</td>
<td>3,998</td>
</tr>
<tr>
<td>Q4</td>
<td>15.05</td>
<td>17.22</td>
<td>4,116</td>
</tr>
<tr>
<td>Q5</td>
<td>5.19</td>
<td>66.43</td>
<td>4,233</td>
</tr>
<tr>
<td>Q6</td>
<td>4.38</td>
<td>60.86</td>
<td>4,029</td>
</tr>
<tr>
<td>Q7</td>
<td>9.73</td>
<td>74.98</td>
<td>4,189</td>
</tr>
<tr>
<td>Q8</td>
<td>2.69</td>
<td>36.47</td>
<td>4,243</td>
</tr>
<tr>
<td>Q9</td>
<td>2.93</td>
<td>72.49</td>
<td>4,287</td>
</tr>
</tbody>
</table>

Elapse time (in msec) / logarithmic scale

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

<table>
<thead>
<tr>
<th></th>
<th>TPA</th>
<th>Saxon</th>
<th>Xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.03</td>
<td>2.48</td>
<td>2,677</td>
</tr>
<tr>
<td>Q2</td>
<td>0.58</td>
<td>27.67</td>
<td>2,432</td>
</tr>
<tr>
<td>Q3</td>
<td>1.56</td>
<td>17.89</td>
<td>2,561</td>
</tr>
<tr>
<td>Q4</td>
<td>0.72</td>
<td>102.71</td>
<td>2,631</td>
</tr>
<tr>
<td>Q5</td>
<td>0.22</td>
<td>91.98</td>
<td>2,647</td>
</tr>
<tr>
<td>Q6</td>
<td>0.23</td>
<td>88.92</td>
<td>2,726</td>
</tr>
<tr>
<td>Q7</td>
<td>1.43</td>
<td>46.85</td>
<td>2,399</td>
</tr>
<tr>
<td>Q8</td>
<td>0.32</td>
<td>109.41</td>
<td>2,577</td>
</tr>
<tr>
<td>Q9</td>
<td>1.43</td>
<td>78.45</td>
<td>2,707</td>
</tr>
</tbody>
</table>

Elapse time (in msec) / logarithmic scale

**Figure 13.** Performance comparison of TPA, Saxon and Xalan on XMark \( D_9 \) 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 $O(m)$ 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 $O(h \cdot 2^k)$, 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


42. Šestáková, E. Indexing XML documents. Master’s Thesis, Czech Technical University in Prague, Faculty of Information Technology, Czech Republic, 2015.


© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).