ASSIGNMENT OF BACHELOR'S THESIS

Title:	Automata Approach to XML Data Indexing: Selecting Unknown Nodes
Student:	Maria Karzhenkova
Supervisor:	Ing. Eliška Šestáková
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

Study the following automata-based XML data indexing methods [1]: Tree String Path Automaton (TSPA) and Tree String Path Subsequences Automaton (TSPSA).
1) Modify these methods so they would also support XPath wildcards (asterisk) to select unknown XML nodes.
2) Discuss theoretical time and space complexities of proposed methods and implement them in the Java programming language.
3) Perform appropriate testing of your implementation.

References

[1] Šestáková, E. *Indexing XML documents*. Diss. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2015.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 30, 2017

Acknowledgements

I would like to express my gratitude to my supervisor, Eliška Šestáková, for her great support and useful remarks during my studies. Her guidance helped me throughout the whole process of writing this thesis.

Also, I would like to thank my family, friends and my partner who encouraged me and gave me their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 15th May 2018

....

Abstract

Being a part of the "Automata Approach to XML Data Indexing" project, this thesis is concerned with studying the existing methods of indexes creation algorithms based on the automata theory and extending them to deal with more significant fragment of XPath queries.

The presented methods allow us to construct XML data indexes that support evaluation of all XPath queries using any combinations of child (/), descendant-or-self (//) axes, asterisk (*) and *nodename* node tests.

Given an XML document D and its corresponding XML tree model T with n nodes, the tree is preprocessed and the index for the document D is constructed.

The searching phase time of each of the constructed indexes for a query Q is bounded by $\mathcal{O}(m)$, where m is size of the query Q, and does not depend on the indexed XML document size n.

Moreover, the space and time complexities for each of the proposed indexes are discussed, all the introduced algorithms are implemented and tested over the real-life datasets.

Keywords: XML, XPath, tree, finite automaton, index, unknown nodes

Supervisor: Ing. Eliška Šestáková

Abstrakt

Tato práce je součástí projektu "Indexování XML dokumentů pomocí automatů". Popisuje existující metody pro indexování XML dokumentů, které jsou založeny na teorii automatů, a jejich rozšíření, za účelem umožnění efektivního zpracování XPath dotazů skládajících se z libovolné kombinace child (/), descendant-or-self (//) os a asterisk (*) a *nodename* node testů, sloužících k navigaci v XML dokumentu.

Ke konstrukci indexu pro daný XML dokument D s n elementy je využít odpovídající XML stromový model T. Zpracování dotazu Q o velikosti m proběhne v čase $\mathcal{O}(m)$ nezávislém na n.

Tato práce obsahuje též diskuzi ohledně časové a paměťové složitosti pro každou z navržených metod. Všechny nově popsané algoritmy jsou implementovány a otestovány na reálních datech.

Klíčová slova: XML, XPath, strom, konečný automat, index, neznámé uzly

Překlad názvu: Indexování XML dokumentů pomocí automatů: výběr neznámých uzlů

Contents

1 Introduction **1**
1.1 Motivation and Objectives 1
1.2 Goals of the Thesis 1
1.3 Thesis Structure 2

2 Theoretical Background **3**
2.1 Notations 3
2.2 Basic Definitions 4
2.2.1 Alphabet, String 4
2.2.2 Graph 4
2.2.3 Tree 5
2.2.4 Language, Grammar 5
2.2.5 Finite Automaton 6
2.3 XML 7
2.4 XML Data Model 8
2.5 XPath 9
2.5.1 XPath Syntax 9
2.5.2 Examples 10

3 Automata-based XML data indexing methods **13**
3.1 String Paths 13
3.2 Tree String Paths Automaton . . 14
3.2.1 Construction of Tree String Paths Automaton 14
3.2.2 Time and Space Complexities 15
3.3 Tree String Path Subsequences Automaton 15
3.3.1 Construction of Tree String Path Subsequences Automaton . . 16
3.3.2 Time and Space Complexities 17
3.4 Tree Paths Automaton 17
3.4.1 Construction of Tree String Paths Automaton 18
3.4.2 Time and Space Complexities 19

4 Automata for Selecting Unknown Nodes **21**
4.1 Tree String Paths Automaton for Selecting Unknown Nodes 21
4.1.1 Discussion of Time and Space Complexities 23
4.2 Tree String Path Subsequences Automaton for Selecting Unknown Nodes 28
4.2.1 Discussion of Time and Space Complexities 31
4.3 Tree Paths Automaton for Selecting Unknown Nodes 34
4.3.1 Discussion of Time and Space Complexities 38

5 Implementation **41**
5.1 System Architecture 41
5.2 Document Parser 41
5.3 Index Builder 41
5.4 XML Data Index 43

6 Testing and Experimental Evaluation **45**
6.1 Experimental Setup 45
6.2 Performance of TPA* Construction 46
6.3 Performance of Query Processing 47

7 Conclusion **49**
7.1 Goals Fulfillment 49
7.2 Contribution of the Thesis 49
7.3 Future Work 50

Bibliography **51**

A Acronyms **53**

B Tree String Path Subsequences Automaton for Selecting Unknown Nodes **55**

C Tree Paths Automaton for Selecting Unknown Nodes **57**

Figures

2.1 XML tree model $T(D)$ from Example 2.1 9
2.2 Occurrences of XPath queries Q_1 and Q_2 from Example 2.3 11
2.3 Occurrences of XPath queries Q_3 and Q_4 from Example 2.4 11
2.4 Occurrences of the XPath query Q_5 = `//US/*` from Example 2.5 . . . 11

3.1 TSPA for the XML document D 15
3.2 TSPSA for the XML document D 16
3.3 TPA for the XML document D . 18

4.1 TSPA* automata for individual string paths of the XML tree model T from Figure 2.1 22
4.2 TSPA* for the XML tree model T from Figure 2.1 24
4.3 Occurrences of XPath queries from Example 4.3 25
4.4 Occurrences of XPath queries from Example 4.3 26
4.5 Occurrences of XPath query Q_4 = `/SERIES/*/*` from Example 4.3 . . . 26
4.6 "Backbone" of the TSPSA* automaton for the string path P from Example 4.4 30
4.7 Not-completed TSPSA* automaton for the string path P from Example 4.4 after inserting "additional" transitions 30
4.8 Completed TSPSA* automaton for the string path P from Example 4.4 after inserting "wildcard" states and transitions 30
4.9 Occurrences of XPath queries Q_1 and Q_4 from Example 4.5 33
4.10 Occurrences of XPath queries Q_5 and Q_8 from Example 4.5 33
4.11 Occurrences of the XPath query Q_5 = `//*//*//*` from Example 4.5 33
4.12 TSPA* for the string path P from Example 4.6 35
4.13 TSPSA* for the string path P from Example 4.6 35
4.14 "Not-finished" TPA* for the string path P from Example 4.6 . . 37
4.15 TPA* for the string path P from Example 4.6 38

5.1 System Architecture of the `tpaUNlib` 42
5.2 `DocumentParser` class 42
5.3 `AutomatonFactory` and `ParallelRunner` classes 43
5.4 `Automaton` class 43
5.5 `State`, `Transition` and `XMLTag` classes 44

6.1 Performance comparison of TPA* and Saxon 47

Tables

6.1 Characteristics of the datasets . . 45
6.2 Set of queries for XMark datasets 46
6.3 Numbers of elements satisfying the queries in the datasets 46
6.4 Experimental results on the index size and construction time 46

B.1 Transition table of TSPSA* for the XML document D from Example 2.1 55

C.1 Transition table of TPA* for the XML document D from Example 2.1 58

Chapter 1

Introduction

1.1 Motivation and Objectives

XML, which stands for eXtensible Markup Language, was designed to store and transport data and it still remains one of the main methods of information exchange over the Internet [1]. That is why, efficient querying of XML data belongs to key tasks that are extensively studied. XPath (XML Path Language) and other similar instruments allow us to retrieve data from XML documents [15]. However, we can achieve faster searching by preprocessing the data and building an index.

This thesis is a part of the "Automata Approach to XML Data Indexing" project [6, 10, 11, 12] that is focused on studying and creating new algorithms for building indexes of XML data by means of (as it can be seen from the project name) automata.

The whole project is about creating a new concept of solving the XML index problem using the automata theory. Nowadays, the XML index problem is still actively researched, but nevertheless the existing solutions are not usually based on a systematic approach of the standard theory of formal languages and automata. Therefore, this project is supposed to create, describe and implement new algorithms for creating XML indexes based on the automata theory, which are able to answer some subsets of the XPath queries, e.g., XPath queries using any combinations of the child and descendant-or-self axes, in time that does not depend on the size of the original XML document.

1.2 Goals of the Thesis

The main goals of this thesis are to:

- study automata-based XML data indexing methods such as Tree String Paths Automaton (TSPA) and Tree String Path Subsequences Automaton (TSPSA),
- modify these methods so they would also support XPath wildcards (asterisk) to select unknown XML nodes,

- discuss theoretical time and space complexities of the proposed methods,
- implement the proposed algorithms in the Java programming language,
- perform appropriate testing and experimental evaluation of the implementation.

1.3 Thesis Structure

The rest of the thesis is organized as follows:

Chapter 2 covers theoretical backgrounds such as notation, definitions and basic information about XML and XPath.

Chapter 3 is focused on existing automata-based XML data indexing methods.

Chapter 4 is the main part of this thesis, as it describes modification of the existing methods so they support selecting of unknown nodes. This chapter also includes discussions of time and space complexities of the proposed methods.

Chapter 5 is about implementation details and code structure.

Chapter 6 is focused on testing and experimental evaluation.

Conclusion is the closing chapter of this work.

Chapter 2

Theoretical Background

In this chapter all the concepts that are used in the next chapters of the thesis are presented and defined. Moreover, here one can find a short introduction to the XML format and to the XPath queries.

2.1 Notations

This section contains all the notations that are indispensable for understanding the following thesis text.

- A for an alphabet,
- a for an alphabet symbol,
- L for a language,
- N for a set of nonterminal symbols,
- S for a start symbol of a grammar,
- q, p for states of an automaton,
- q_0 for an initial state of an automaton,
- Q for a set of states of an automaton,
- F for a set of final states of an automaton,
- δ for automaton transition function,
- M for a finite automaton,
- G for a directed graph and for a grammar,
- V for a set of vertices (nodes) in a directed graph,
- R for a set of lists of edges in a directed graph,
- v, u for a graph vertex (node),
- D for an XML document,
- T for a tree and for an XML tree model,
- n for a node of a tree and for a node of an XML tree model,

- r for a root node of an XML tree model,
- e for an XML element,
- Q for an XML query,
- h for a height of an XML tree model,
- k for a number of leaves of an XML tree model,
- l for a label of a tree node or an XML element,
- P_i for a string path,
- P for a string paths set and for a set of production rules,
- $O_P(e)$ for a set of occurrences of an element e in a string path P.

2.2 Basic Definitions

In this section definitions useful for the rest of the thesis are presented. The definitions except the common ones are mostly taken from the "Introduction to Automata Theory, Languages and Computability" book [3] and the "Handbook of Graph Theory" [2].

2.2.1 Alphabet, String

Definition 2.1 (Alphabet)**.** An Alphabet is a finite, nonempty set of symbols.

Definition 2.2 (String (or Word))**.** A String (or Word) over a given alphabet A is a finite sequence of symbols of A.

Definition 2.3 (Length of a string)**.** A Length of a string x is the number of its symbols and is denoted by $|x|$.

Definition 2.4 (Prefix)**.** A Prefix of a string $x = x_1x_2 \dots x_n$ is a string $y = x_1x_2 \dots x_m$, where $m \leq n$.

Definition 2.5 (Suffix)**.** A Suffix of a string $x = x_1x_2 \dots x_n$ is a string $y = x_ix_{i+1} \dots x_n$, where $i \geq 1$.

Definition 2.6 (Subsequence)**.** A Subsequence of a string $x = x_1x_2 \dots x_n$ is a string y obtained by deleting zero or more symbols from x.

2.2.2 Graph

Definition 2.7 (Directed graph)**.** A Directed graph G is a pair (V, R), where V is a set of nodes and R is a set of lists of edges such that each element of R is of the form $((v, u_1), (v, u_2), \dots, (v, u_n))$, where $v, u_1, u_2, \dots, u_n \in V, n \geq 0$. This element indicates that, for node v, there are n edges leaving v, entering node u_1, node u_2, and so forth.

Definition 2.8 (Path)**.** A sequence of nodes $(v_0, v_1, \dots v_n), n \geq 1$ is a Path of length n from node v_0 to node v_n if there exists an edge which leaves node v_{i-1} and enters node v_i for all i, where $1 \leq i \leq n$.

Definition 2.9 (Cycle). A Cycle is a path $v_0, v_1, \ldots v_n$, where $v_0 = v_n$.

Definition 2.10 (Directed acyclic graph). A Directed acyclic graph is a Directed graph that has no Cycle.

Definition 2.11 (Labeling). A Labeling of a Graph $G = (V, R)$ is a mapping V into a set of labels.

Definition 2.12 (Out-degree, In-degree). Given a node v, its Out-degree is the number of distinct pairs $(v, u) \in R$, where $u, v \in V$. By analogy, the In-degree of node v is the number of distinct pairs $(u, v) \in R$ where $u, v \in V$.

2.2.3 Tree

Definition 2.13 (Tree). A Tree is an acyclic connected graph. Any node of a tree can be selected as a Root of the tree. A tree with a root is called Rooted tree.

Definition 2.14 (Rooted directed tree). A Rooted directed tree T is a Directed acyclic graph $T = (V, R)$ with a special node $r \in V$, called the root, such that

- in-degree of r is 0,
- in-degree of all other nodes of T is 1,
- there is just one path from the root r to every node $n \in V$, where $n \neq r$.

Definition 2.15 (Leaf). A node $n \in V$ is a leaf of a Rooted directed tree $T = (V, R)$ if it has out-degree 0.

Definition 2.16 (Labeled tree). A Labeled tree is a tree $T = (V, R)$ that has the following property: each node $n \in V$ is labeled by a symbol $a \in A$, where A is an alphabet.

Definition 2.17 (Subtree). Let $T = (V, R)$ be a Rooted directed tree. A Subtree $T' = (V', R')$ of T is a rooted directed tree, where $V' \subseteq V \wedge R' \subseteq R$. Also, if n is a leaf in T', then n is a leaf in T.

2.2.4 Language, Grammar

Definition 2.18 (Language). A Language L over an alphabet A is set of words over that alphabet.

Definition 2.19 (Grammar). A Grammar is a quadruple $G = (N, A, P, S)$, where

- N is a finite set of nonterminal symbols,
- A is an input alphabet,
- P is a set of production rules, i.e., finite subset of $(N \cup A)^* N (N \cup A)^* \times (N \cup A)^*$,
- $S \in N$ is the start symbol of the grammar.

Definition 2.20 (Regular grammar). A grammar $G = (N, A, P, S)$ is called Regular, if every rule is of the form $A \to aB$ or $A \to a$, where $A, B \in N$, $a \in A$. The single exception is the rule $S \to \varepsilon$ in case that S is not present in the right-hand side of any rule.

Definition 2.21 (Context-free grammar). A grammar $G = (N, A, P, S)$ is called Context-free, if every rule is of the form $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup A)^*$.

2.2.5 Finite Automaton

Definition 2.22 (Deterministic finite automaton). A Deterministic finite automaton (DFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where

- Q is a finite set of states,
- A is a finite input alphabet,
- δ is a mapping from $Q \times A$ to Q,
- q_0 is the initial state,
- $F \subseteq Q$ is a set of final states.

Definition 2.23 (Nondeterministic finite automaton). A Nondeterministic finite automaton (NFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where

- Q is a finite set of states,
- A is a finite input alphabet,
- δ is a mapping from $Q \times A$ into a set of subsets Q (denoted by 2^Q),
- q_0 is the initial state,
- $F \subseteq Q$ is a set of final states.

Definition 2.24 (d-subset). Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ be a nondeterministic finite automaton. Let $M_2 = (Q_2, A, \delta_2, q_{02}, F_1)$ be the deterministic finite automaton equivalent to automaton M_1. Automaton M_2 is constructed using the standard determinization algorithm based on subset construction [5]. Every state $q \in Q_2$ corresponds to some subset d of Q_1. This subset will be called 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.

Definition 2.25 (Product construction (union) of finite automata). Let M_1 and M_2 be two finite automata with the same input alphabet A that accept languages L_1 and L_2, respectively. Product construction (union) of finite automata is an algorithm for creation a new finite automaton M that accepts language $L = L_1 \cup L_2$. This is achieved by running both automata "in parallel", by remembering the states of both automata while reading the input.

Considering $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ and $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$, we define a finite automaton M as follows: $M = (Q, A, \delta, q_0, F)$, where

- $Q = Q_1 \times Q_2$,
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$,
- $q_0 = (q_{01}, q_{02})$,
- $(q_1, q_2) \in F$ iff $q_1 \in F_1$ or $q_2 \in F_2$.

Each state in Q is a pair consisting of a state from Q_1 and a state from Q_2. In state (q_1, q_2) on input a, the automaton M proceed by executing M_1 from q_1, and in parallel, executing M_2 from q_2.

2.3 XML

XML [1], which stands for Extensible Markup Language, was originally created in order to store and transfer data in a usable, both human- and machine-readable, format. Nowadays, it is still playing a huge role in the exchange of information all over the Internet.

The most important components of an XML document are:

- **Tags** – the markup constructs that start with `<` and end with `>`. They are divided into three different types:
  - *start-tag* – `<tag>`,
  - *end-tag* – `</tag>`,
  - *empty-element tag* – `<empty-tag/>`,
- **Elements** – everything from the start-tag to the end-tag (including both these tags),
- **Attributes** – parts of elements, that contain additional information about a related element.

XML (in contrast with HTML) is an extensible language, which means that there are no predefined tags. Tags are created by the author of a document to fit his own needs.

Definition 2.26 (Well-formed XML document)**.** An XML document with correct syntax is called "Well-formed" [14]. The syntax rules are:

- all XML documents must have a root element,
- all XML elements must have a closing tag,
- all XML tags are case sensitive,
- all XML elements must be properly nested,
- all XML attribute values must be quoted.

Example 2.1. A simple well-formed XML document that represents some information about famous TV series.

```
<SERIES>
  <US name="House M.D.">
    <ACTORS>
      <FEMALE>Lisa Edelstein</FEMALE>
      <MALE>Hugh Laurie</MALE>
    </ACTORS>
    <GENRES>Drama</GENRES>
  </US>
  <UK name="The IT Crowd">
    <ACTORS>
      <MALE>Chris O'Dowd</MALE>
      <MALE>Richard Ayoade</MALE>
    </ACTORS>
    <GENRES>Comedy</GENRES>
  </UK>
</SERIES>
```

The very first line of the Example 2.1 contains the start-tag `<SERIES>` and the last one represents the paired end-tag `</SERIES>`. So we can say that all the data between these two tags (including the tags themselves) is forming the `SERIES` element.

Since elements can be nested inside other elements, we can build almost any structure of the XML document. These nested elements give us information about its parent element. For example, such element as `MALE` and optionally `FEMALE` are sub-elements of the element `ACTORS`.

Attributes are another way of representing data related to the element. For instance, `<US name="House M.D.">` shows us that the name of the element `US` is `"House M.D."`.

2.4 XML Data Model

In order to make our algorithms describing process more clear, we define an XML Data Model that includes:

- an XML alphabet, which helps us to represent elements of an XML document in more intuitive way,
- a tree model of an XML document.

We can represent a well-formed XML document as an ordered labeled tree where nodes correspond to XML elements, and edges represent element inclusion relationships. For simplicity, we do not search elements by attributes or texts in the leaf nodes, so for now we can consider only the XML document structure and ignore this additional information. We assume that only well-formed documents are presented as inputs for our indexing methods.

A node in an XML tree model is represented by a pair (*label*, *id*), where *id* and *label* represents its identifier and tag name, respectively. Preorder

numbering scheme will be used to uniquely assign an identifier to each of the tree nodes.

Definition 2.27 (XML alphabet)**.** Let D be an XML document. An XML alphabet A of D, represented by $A(D)$, is an alphabet where each symbol represents a tag name (label) of an XML element in D.

Example 2.2. Let D be the XML document from Example 2.1. The corresponding XML alphabet A is

$$A(D) = \{\texttt{SERIES},\texttt{US},\texttt{UK},\texttt{ACTORS},\texttt{GENRE},\texttt{FEMALE},\texttt{MALE}\}.$$

Figure 2.1 shows XML tree model $T(D)$ for the XML document D.

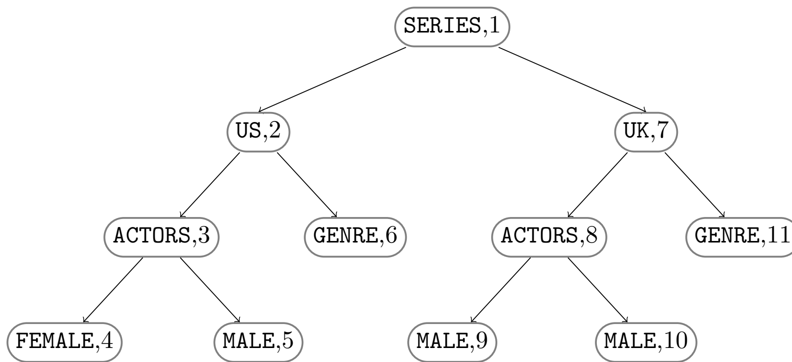


Figure 2.1: XML tree model $T(D)$ from Example 2.1

We should note that in this thesis we use a modified version of the product construction algorithm. The biggest difference from mentioned in Definition 2.25 is that the numbers in d-subsets of automata contain ids of the XML document nodes, so we do not want the states of the product to contain each id more than once. Instead of this, in the implementation, we create a set of states $consistsOf$ for each of the newly created state q of the product automaton and we use this set of states to run all their transitions "in parallel" and to construct the next states that can be reached from the state q. Moreover, the unreachable states are not included in the resulting product of automata.

2.5 XPath

XPath [15], which stands for XML Path Language, is a powerful instrument that can be used to navigate through elements and attributes in an XML document. In this section we describe only a small subset of the XPath language expressions, which will be useful for the rest of the thesis.

2.5.1 XPath Syntax

XPath uses path expressions (also called queries) to select nodes or node-sets in an XML document. XML documents are treated as trees (as it is described

in the previous Subsection 2.4). Nodes of a tree are selected in a certain context – current node in the XML tree a processor is looking at. In this thesis we work only with selecting XML document elements (we are not selecting such kinds of nodes as attributes, texts etc. of the XML document). That is why we need only a small subset of the XPath expressions to work with.

To select a subset of XML document elements we use the following types of XPath expressions:

- **Axis** – defines a node-set relative to the current node
  - *child* (`/`) – selects all children of the current node,
  - *descendant-or-self* (`//`) – selects all descendants (children, grandchildren, etc.) of the current node,
- **Node test**
  - *nodename* – matches all nodes with the name *“nodename”*,
  - *** – matches any element node.

For simplicity, we denote a subset of XPath queries that uses both child and descendant-or-self axes and both nodename and asterisk ($*$) node tests as $XP^{\{/,//,nodename,*\}}$. Analogically we can describe any subset of XPath expressions, e.g., $XP^{\{/,nodename\}}$ – for XPath queries using only the child axis and nodename node test or $XP^{\{//,*\}}$ – for XPath queries using only the descendant-or-self axis and asterisk ($*$) node test.

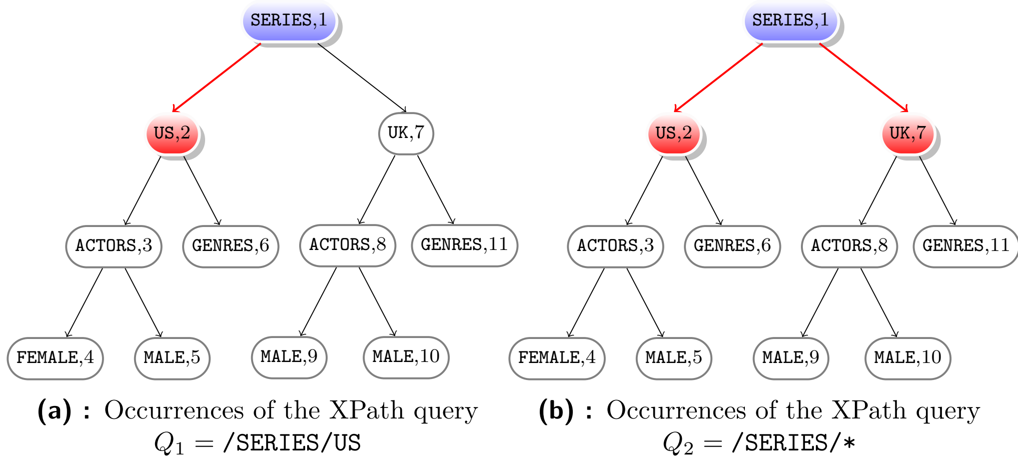
2.5.2 Examples

Let us show how the XPath queries (expressions) select nodes by means of examples.

Example 2.3. Consider XML tree model $T(D)$ from Figure 2.1. In Figure 2.2 nodes selected by queries Q_1 = `/SERIES/US` and Q_2 = `/SERIES/*` are represented in red color. Nodes that are on the path to the selected nodes, which is shown by red edges, are blue.

Example 2.4. Consider XML tree model $T(D)$ from Figure 2.1. In Figure 2.3 nodes selected by queries Q_3 = `//ACTORS` and Q_4 = `//US//*` are represented in red color. Nodes that are on the path to the selected nodes, which is shown by red edges, are blue. Note that, for example, in Figure 2.3a there are some “skipped nodes” on the path. This happens because the query Q_4 is looking for any nodes that have label `ACTORS`, that is why nodes (1), (2), (7) are not selected as parts of the path.

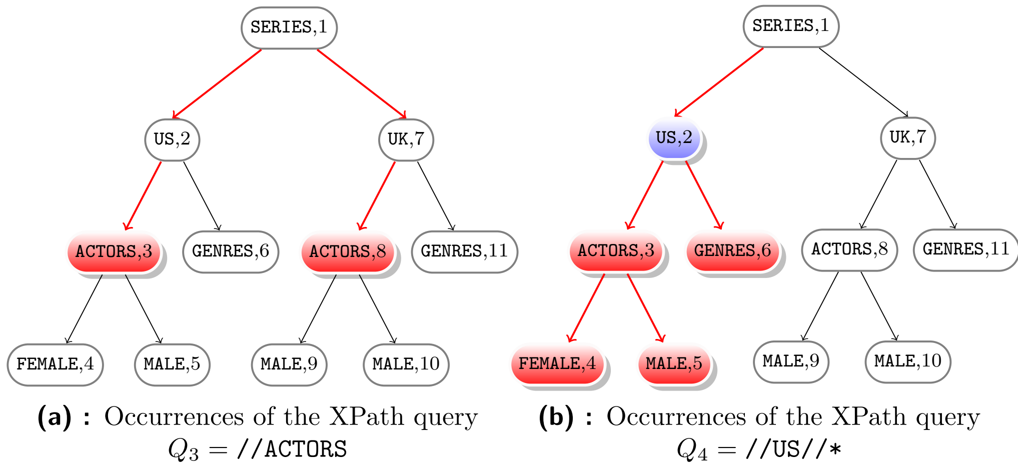
Example 2.5. Consider XML tree model $T(D)$ from Figure 2.1. In Figure 2.4 nodes selected by query Q_5 = `//US/*` are represented in red color. Nodes that are on the path to the selected nodes, which is shown by red edges, are blue.



(a) : Occurrences of the XPath query $Q_1 =$ `/SERIES/US`

(b) : Occurrences of the XPath query $Q_2 =$ `/SERIES/*`

Figure 2.2: Occurrences of XPath queries Q_1 and Q_2 from Example 2.3



(a) : Occurrences of the XPath query $Q_3 =$ `//ACTORS`

(b) : Occurrences of the XPath query $Q_4 =$ `//US//*`

Figure 2.3: Occurrences of XPath queries Q_3 and Q_4 from Example 2.4

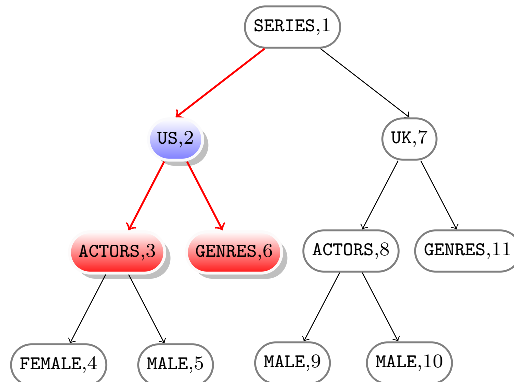


Figure 2.4: Occurrences of the XPath query $Q_5 =$ `//US/*` from Example 2.5

Chapter 3

Automata-based XML data indexing methods

In this chapter "Automata Approach to XML Data Indexing" project achievements [10, 11, 12] are briefly described.

The main outcome of the previous work are three methods of constructing automata-based indexes. The first index is Tree String Paths Automaton that allows us to efficiently evaluate all $XP^{\{/,nodename\}}$ queries, i.e., XPath expressions using only child axis (/), over an XML document. The second one is Tree String Path Subsequences Automaton, which is able to evaluate all $XP^{\{//,nodename\}}$ expressions, i.e., XPath expressions using only descendant-or-self axis (//). And the third, so far the most powerful index automaton – Tree Paths Automaton is capable of efficient evaluation of all $XP^{\{/,//,nodename\}}$ queries, i.e, queries using both child (/) and descendant-or-self (//) axes, over an XML document.

3.1 String Paths

The indexes are built as compositions of finite automata accepting parts of paths queries. That is why, we need now to describe the XML tree model from the point of view of its linear fragments – string paths.

Definition 3.1 (String path)**.** Let T be an XML tree model of height h. A string path $P = n_1 n_2 \dots n_t$ $(t \leq h)$ of T is a linear path leading from the root $r = n_1$ to the leaf n_t.

Definition 3.2 (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 3.3 (String paths set)**.** Let T be an XML tree model with k leaves. A set of all string paths over T is called a string paths set, denoted by $P(T) = \{P_1, P_2, \dots, P_k\}$.

Example 3.1. Consider the XML tree model T illustrated in Figure 2.1. The string paths set $P(T)$ for the XML tree model T is shown in this example. Each node of T is represented by its label and identifier, which is shown in parenthesis.

- $P_1 =$ SERIES(1) US(2) ACTORS(3) FEMALE(4),
- $P_2 =$ SERIES(1) US(2) ACTORS(3) MALE(5),
- $P_3 =$ SERIES(1) US(2) GENRE(6),
- $P_4 =$ SERIES(1) UK(7) ACTORS(8) MALE(9),
- $P_5 =$ SERIES(1) UK(7) ACTORS(8) MALE(10),
- $P_6 =$ SERIES(1) UK(7) GENRE(11).

The corresponding string path alphabets are as follows:

- $A(P_1) = \{$SERIES,US,ACTORS,FEMALE$\}$,
- $A(P_2) = \{$SERIES,US,ACTORS,MALE$\}$,
- $A(P_3) = \{$SERIES,US,GENRE$\}$,
- $A(P_4) = A(P_5) = \{$SERIES,UK,ACTORS,MALE$\}$,
- $A(P_6) = \{$SERIES,UK,GENRE$\}$.

3.2 Tree String Paths Automaton

The Tree String Paths Automaton (TSPA) is a finite state automaton that efficiently evaluates linear XPath queries $XP^{\{/,nodename\}}$ using the child-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 3.4 (Tree String Paths Automaton)**.** Let D be an XML document. The Tree String Paths Automaton accepts all $XP^{\{/,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

Let us describe the structure of TSPA by means of an example.

Example 3.2. Consider the XML document D from Example 2.1, the corresponding TSPA is shown in Figure 3.1. As we can see in the picture of the created TSPA, for the input XPath query $Q_1 =$ /SERIES/UK/ACTORS/MALE it returns an answer $(9, 10)$ and for the input XPath query $Q_2 =$ /SERIES/US/ACTORS it returns an answer (3).

3.2.1 Construction of Tree String Paths Automaton

To build TSPA for an XML document D we first of all need to obtain its tree model $T(D)$ and a corresponding string paths set $P(T)$.

Then we construct prefix automata for each of the string paths from the string paths set $P(T)$. We use the prefix automata, because XPath queries containing only child axes are basically prefixes of the individual string paths.

To build TSPA we run all the prefix automata "in parallel" by remembering the states of all automata while reading the input. This is achieved by the product construction (union), see Definition 2.25.

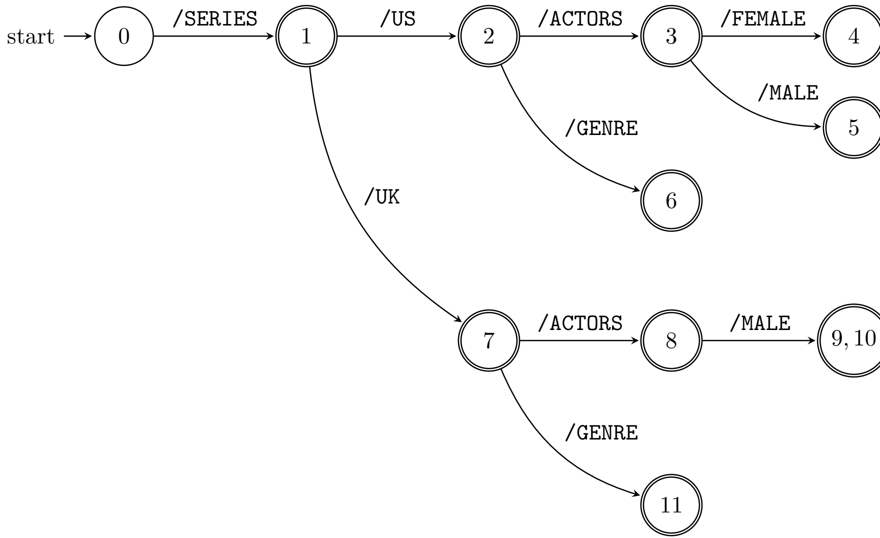


Figure 3.1: TSPA for the XML document D

3.2.2 Time and Space Complexities

The time in which TSPA for an XML document D evaluates an XPath query Q of length m depends only on the query length m.

Moreover, the number of states of TSPA is linear in size n of an XML document D and is less than or equal to n.

The number of transitions of TSPA clearly becomes $\mathcal{O}(n-1)$, since TSPA is an acyclic tree-like finite automaton. For details and proofs, see [9].

3.3 Tree String Path Subsequences Automaton

The Tree String Path Subsequences Automaton (TSPSA) is a finite state automaton that supports evaluation of linear XPath queries $XP^{\{//,nodename\}}$ using the descendant-or-self axis (//) only.

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 3.5 (Tree String Path Subsequences Automaton)**.** Let D be an XML document. The Tree String Path Subsequences Automaton accepts all $XP^{\{//,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

We can use an example for better understanding of TSPSA structure.

Example 3.3. Consider the XML document D from Example 2.1, the corresponding TSPSA is shown in Figure 3.2. Please notice that in Figure 3.2 we are using first letters of the label names (e.g., `//SERIES` is represented as

//S) to make the automaton representation more clear. The only exceptions are labels whose names consist of two letters.

As we can see in the picture of the created TSPSA, for the input XPath query $Q_1 =$ //UK//MALE it returns an answer $(9, 10)$ – this answer contains all the elements with label MALE that are descendants of element with label UK. For the input XPath query $Q_2 =$ //MALE it returns an answer $(5, 9, 10)$ – this answer contains all the elements with label MALE in the XML document D.

Figure 3.2: TSPSA for the XML document D

3.3.1 Construction of Tree String Path Subsequences Automaton

Construction of the TSPSA is very similar to construction of the TSPA. The main difference is that here we are interested in subsequences of a string path rather than in its prefixes, as we are working with XPath queries using descendant-or-self axes only.

To build TSPSA for an XML document D we again should preprocess the XML document D to obtain its tree model $T(D)$ and a corresponding string paths set $P(T)$.

Then we construct subsequence automata for each of the string paths from the string paths set $P(T)$.

To build TSPSA we run all the subsequence automata "in parallel" using the product construction (union), see Definition 2.25.

3.3.2 Time and Space Complexities

Time in which TSPSA for an XML document D evaluates an XPath query Q of length m depends only on the query length m.

Consider a tree model $T(D)$ of an XML document D of height h and having k leaves. Since length of a string path is less than or equal to h and we have exactly k string paths, we can be sure that the number of states in each of the subsequence automata is less than or equal to $h+1$ (it clearly flows from the Algorithm for building the subsequence automata for a single string path [10]) and the number of such automata is clearly k. Therefore, the number of TSPSA states can be bounded by $\mathcal{O}(h^k)$, i.e., the size of a product of k automata with $\mathcal{O}(h)$ states. And the number of transitions of TSPSA can be easily bounded by $\mathcal{O}(|A(D)| \cdot h^k)$, as the maximum out-degree of each state is equal to the size of the input alphabet.

However, due to the branching tree structure we can expect the number of TSPSA nodes and transitions to be lower. The common prefixes can appear in the set of strings. For example, the number of states of TSPSA for the XML document D and its tree model $T(D)$ that satisfies the l-property (see Definition 3.6) is $\mathcal{O}(h \cdot 2^k)$. The number of transitions of TSPSA can be estimated as $\mathcal{O}(|A(D)| \cdot h \cdot 2^k)$. For more details and proofs see [10].

Definition 3.6. [Level property] Let $T = (V, E)$ be a labeled directed rooted tree. Level property (l-property):

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

3.4 Tree Paths Automaton

The Tree Paths Automaton (TPA) is a finite state automaton that efficiently supports evaluation of linear XPath queries $XP^{\{/,//,nodename\}}$ using both child-axis (/) and descendant-or-self axis (//).

We can represent such a fragment of XPath queries over an XML document D by the following context-free grammar:

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

Definition 3.7 (Tree Paths Automaton)**.** Let D be an XML document. The Tree Paths Automaton accepts all $XP^{\{/,//,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

Let us describe the structure of TPA by means of an example.

Example 3.4. Consider the XML document D from Example 2.1, the corresponding TPA is shown in Figure 3.3. Please notice that in Figure 3.3 we are again using first letters of the label names (e.g., `//SERIES` is represented as

`//S`) to make the automaton representation more clear. The only exceptions are labels whose names consist of two letters.

Moreover, 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`$) = q$.

As we can see in the picture of the created TPA, for the input XPath query $Q_1 =$ `/SERIES/UK//MALE` it returns an answer $(9, 10)$ – this answer contains all the elements with label `MALE` that are descendants of element with label `UK` which is a descendant of the root element `SERIES`. For the input XPath query $Q_2 =$ `/SERIES//MALE` it returns an answer $(5, 9, 10)$ – this answer contains all the elements under the root element `SERIES` which have label `MALE`.

Figure 3.3: TPA for the XML document D

3.4.1 Construction of Tree String Paths Automaton

TPA combines principles of both TSPA and TSPSA described above, as both $XP^{\{/,nodename\}}$ and $XP^{\{//,nodename\}}$ queries are subsets of $XP^{\{/,//,nodename\}}$ queries, which are supported by TPA. That is why, to build TPA we first build prefix and subsequence automata for all the string paths of the tree model $T(D)$. And then, we combine each pair of these automata to create TPA for a single string path (for the algorithm see [10]).

The last step of building TPA is again product construction of the constructed automata for individual string paths (see Definition 2.25).

3.4.2 Time and Space Complexities

Time in which TPA for an XML document D evaluates an XPath query Q of length m depends only on the query length m.

The number of states of TPA for the XML document D and its tree model $T(D)$ that satisfies the l-property is $\mathcal{O}(h \cdot 2^k)$, where h is height of the tree model $T(D)$ of the XML document D and k is number of leaf nodes in $T(D)$.

The number of transitions of TPA can be estimated as $\mathcal{O}(|A(D)| \cdot h \cdot 2^k)$, since the number of outgoing transitions for each state of TPA is less than or equal to the size of the alphabet $A(D)$ of the XML document D multiplied by two (we can use both `/LABEL` and `//LABEL` transitions). For more details and proofs see [10].

Chapter 4

Automata for Selecting Unknown Nodes

In this chapter the automata described in the previous part are extended to accept all possible paths queries using combinations of child-axis (/), descendant-or-self axis (//) and wildcard (*) and *nodename* node tests.

The first automaton that is introduced in this chapter is Tree String Paths Automaton for Selecting Unknown Nodes representing an index for $XP^{\{/,*,nodename\}}$ queries, i.e., paths queries using child-axis (/), wildcard (*) and *nodename* node tests. The second one is Tree String Path Subsequences Automaton for Selecting Unknown Nodes indexing $XP^{\{//,*,nodename\}}$ queries, i.e., paths queries using descendant-or-self axis (//), wildcard (*) and *nodename* node tests. The last automaton that is described here is called Tree Paths Automaton for Selecting Unknown Nodes and accepts all the $XP^{\{/,//,*,nodename\}}$ queries.

The search phase of all elements satisfying the query of size m is performed in time linear in m, not depending on the XML document size n. The main issue is the size of the deterministic automata, which, in theory, can be exponential in n. However, the actual sizes of automata are less than exponential in the XML document size n (it is proved in this chapter for each of the automata).

4.1 Tree String Paths Automaton for Selecting Unknown Nodes

Definition 4.1 (Tree String Paths Automaton for Selecting Unknown Nodes). Let D be an XML document. The Tree String Paths Automaton for Selecting Unknown Nodes (TSPA*) accepts all $XP^{\{/,*,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

TSPA* speeds up the evaluation of linear XPath queries using child-axis (/) and wildcard (*) and *nodename* node tests (i.e., $XP^{\{/,*,nodename\}}$). We can represent such a fragment of XPath queries over an XML document D by the context-free grammar as follows:

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

To construct TSPA*, we start with building deterministic finite automata that accept all non-empty prefixes of individual string paths. Construction of a TSPA* automata for a single string path is described by Algorithm 4.1.

Algorithm 4.1. Construction of a deterministic TSPA* automaton for a single string path.
Data: A string path $P = n_1 n_2 \dots n_{|P|}$.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,*,nodename\}}$ queries of P.

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

Example 4.1. Consider the tree model $T(D)$ of an XML document D described in Figure 2.1 and the corresponding string tree paths $P(T)$ from Example 3.1. Transition diagrams of the TSPA* automata for $P(T)$, which are constructed by Algorithm 4.1, are shown in Figure 4.1.

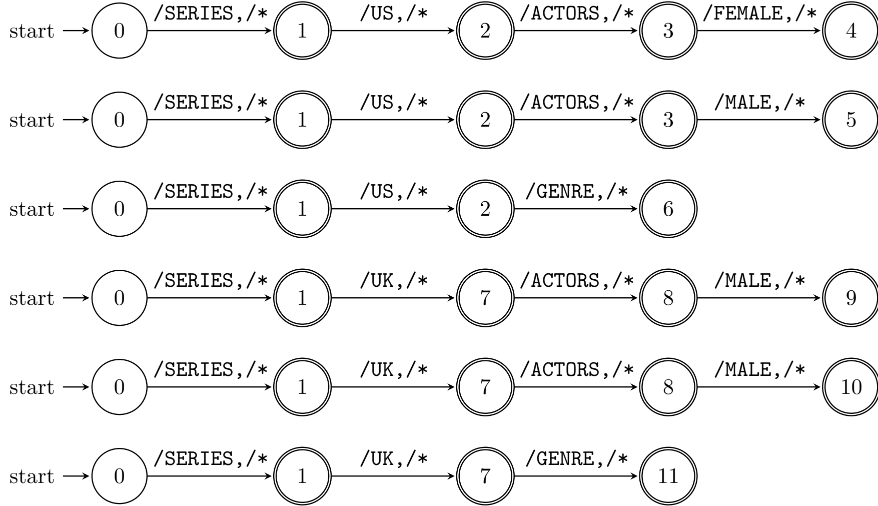


Figure 4.1: TSPA* automata for individual string paths of the XML tree model T from Figure 2.1

To build TSPA*, we can run all the TSPA* automata (constructed by Algorithm 4.1 for all string paths P_i in $P(T)$) “in parallel”, by remembering the states of all automata while reading the input. This is achieved by the product construction (see Definition 2.25).

Algorithm 4.2. Construction of the TSPA* for an XML document D.
Data: String paths set $P(T) = \{P_1, P_2, \dots, P_k\}$ of XML tree model $T(D)$ with k leaves.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,*,nodename\}}$ queries of the XML document D.

1. For all $P_i \in P(T)$, construct a deterministic TSPA* automaton $M_i = (Q_i, \{/a : a \in (A(P_i) \cup \{*\})\}, \delta_i, 0, F_i)$ accepting all $XP^{\{/,*,nodename\}}$ queries of P_i using Algorithm 4.1.

2. Construct the TSPA* $M = (Q, \{/a : a \in (A(D) \cup \{*\})\}, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{/,*,nodename\}}$ queries of the XML document D using the product construction (union) – see Definition 2.25.

Example 4.2. Consider an XML document D from Example 2.1. The corresponding TSPA* accepting all $XP^{\{/,*,nodename\}}$ queries, constructed by Algorithm 4.1, are shown in Figure 4.2. Note that, in transition rules $\delta(p, /L) = q$, L stands for a label from the XML document (from Example 2.1) that begins with this letter(-s) L, except $/*$.

As we can see in the Figure 4.2, TSPA* contains all states that belong to TSPA shown in Figure 3.1 and some extra states. These extra states are formed by different node combinations on each level due to using asterisk node test (*), it can be a combination of all the nodes on a level (e.g., state $(4, 5, 9, 10)$ on the 3rd level) or some combinations of nodes on one level that have the same labels but their ancestors on one of the previous level have different labels (e.g., state $(5, 9, 10)$ – node (5) has an ancestor `US` (2) and nodes (9), (10) have an ancestor `UK` (7) on the 2nd level, that is why this state $(5, 9, 10)$ cannot be formed in TSPA – it is not possible to select both (2) and (7) nodes in TSPA at a time).

4.1.1 Discussion of Time and Space Complexities

TSPA* efficiently supports the evaluation of all $XP^{\{/,*,nodename\}}$ queries of an XML document D. The evaluation of a query of length m is obviously $\mathcal{O}(m)$ and does not depend on the XML document D size.

To be more precise, the evaluation process always consists of two phases:

1. searching phase – finding the state of TSPA* containing the answer in its d-subset,

2. answering phase – returning the relevant nodes of the XML document to the user.

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

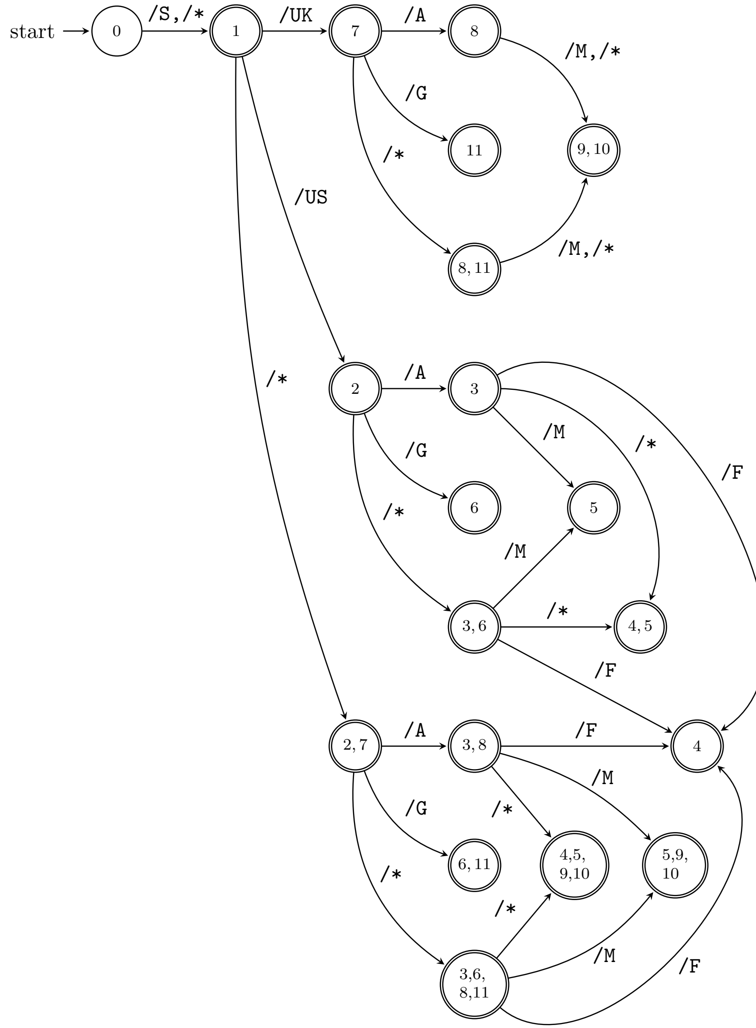


Figure 4.2: TSPA* for the XML tree model T from Figure 2.1

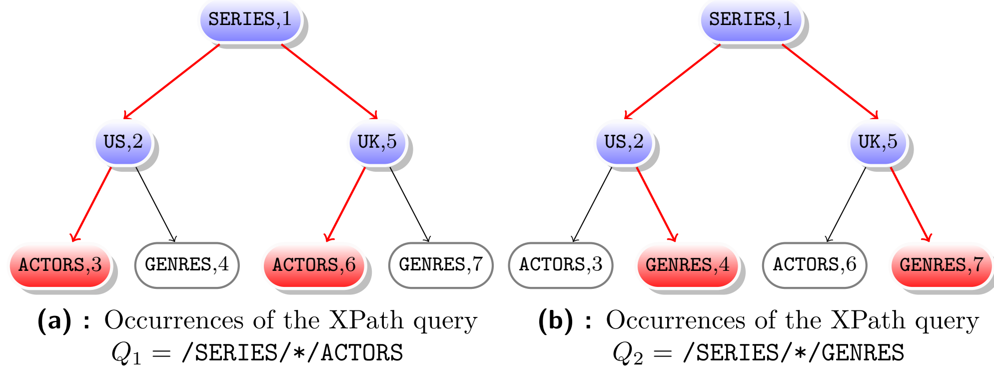
The main question here is the size of TSPA*. To answer this question we need to look on how the states of the index are created. All the states of TSPA* are combinations of nodes of one of the XML document tree $T(D)$ levels (it is illustrated in the Example 4.3). Moreover, in this example the principle of states creation is described.

Definition 4.2 (Children types in TSPA*)**.** Let $T(D)$ be an XML tree model of an XML document D with XML alphabet $A(D)$, q_i be a state (a node combination containing one or more nodes) formed on the ith level of the $T(D)$. All the nodes or node combinations which are selected from q_i by a query $Q_1 =$ `/*` are called "wildcard children" of the state q_i. Analogically, all the nodes or node combinations which are selected from q_i by a query $Q_2 =$ `/LABEL`, where `LABEL` $\in A(D)$, are called "`LABEL` children" of the q_i.

Example 4.3. As it is shown in Figures 4.3, 4.4 and 4.5 different node combinations can be chosen on one level. Note, that except nodes whose selection is described in details below we can also choose every single node of the 3rd level of the shown tree, as all of them have unique paths leading to each node, for example, node (3) can be chosen using queries `/SERIES/US/ACTORS` or `/*/US/ACTORS`.

Let us now describe different node combinations that can be chosen on the 3rd level of the tree (we should keep in mind that states formed on the 2nd level of the shown tree are: (2), (5), (2, 5)):

- nodes (3, 6) can be chosen as "`ACTORS` children" of the state (2, 5) using queries `/SERIES/*/ACTORS` or `/*/*/ACTORS` – see Figure 4.3a,
- similarly we can choose nodes (4, 7) as they are "`GENRES` children" of the state (2, 5), to choose them we use following queries: `/SERIES/*/GENRES` or `/*/*/GENRES` (Figure 4.3b),
- we can select such combination of nodes as (3, 4) ("wildcard children" of the state (2)) with the help of queries `/SERIES/US/*` or `/*/US/*`, it is shown in Figure 4.4a,
- in the same way nodes (6, 7) can be selected using queries `/SERIES/UK/*` or `/*/UK/*`, as they are "wildcard children" of the state (5), see Figure 4.4b,
- finally, as it is shown in Figure 4.5 we can choose a node combination (3, 4, 6, 7) – so called "wildcard children" of the state (2, 5).



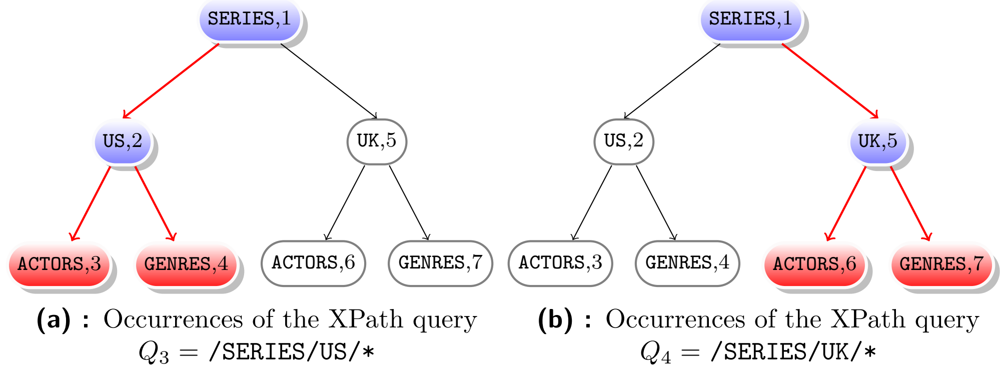
(a) : Occurrences of the XPath query Q_1 = `/SERIES/*/ACTORS`

(b) : Occurrences of the XPath query Q_2 = `/SERIES/*/GENRES`

Figure 4.3: Occurrences of XPath queries from Example 4.3

Definition 4.3 (Levels set). Let $T(D)$ be an XML tree model with n nodes and height h. We define $l_i, i \leq h$ to be a number of nodes on the i-th level of the XML tree model $T(D)$. A set of all $l_i, i \leq h$ $(l_i \geq 1)$ is called a levels set, denoted by $L(T) = \{l_1, l_2, ..., l_h\}$. Obviously $\sum_{j=1}^{h} l_j = n$.

So we can assume that all possible combinations of nodes on each level of the XML tree model can form a state in TSPA*.



(a) : Occurrences of the XPath query $Q_3 =$ `/SERIES/US/*`

(b) : Occurrences of the XPath query $Q_4 =$ `/SERIES/UK/*`

Figure 4.4: Occurrences of XPath queries from Example 4.3

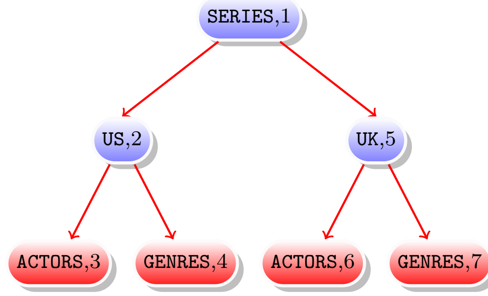


Figure 4.5: Occurrences of XPath query $Q_4 =$ `/SERIES/*/*` from Example 4.3

$$\binom{l_1}{1} + \binom{l_1}{2} + \ldots + \binom{l_1}{l_1} + \ldots + \binom{l_h}{1} + \binom{l_h}{2} + \ldots + \binom{l_h}{l_h} = \sum_{j=1}^{h} \sum_{i=1}^{l_j} \binom{l_j}{i} =$$

$$= \sum_{j=1}^{h} (2^{l_j} - 1) = 2^{l_1} + 2^{l_2} + \ldots + 2^{l_h} - h \leq 2^{l_1 + l_2 + \ldots + l_h} - h = 2^n - h$$

This assumption, however, leads to the fact that the number of states of TSPA* is exponential in the size of the XML document n.

From another point of view, we can bound the number of states of TSPA* by size of the product of all the prefix automata for individual string paths (which is created in Algorithm 4.2). For k prefix automata each containing less than or equal to $h+1$ states, the number of TSPA* states can be trivially bounded by $\mathcal{O}(h^k)$, where k is the number of leaves in a tree model $T(D)$ of an XML document D and h is its height. Hence, we can easily estimate the number of TSPA* transitions as $\mathcal{O}(|A(D)| \cdot h^k)$, since the number of outgoing transitions for each state of TSPA* is less than or equal to $|A(D)| + 1$, where $|A(D)|$ is the size of the alphabet $A(D)$ of the XML document D and plus 1 is for the wildcard transition (`/*`).

Let us, however, present one more estimation of TSPA* states and transition number.

Theorem 4.1. Let D be an XML document and $T(D)$ be its XML tree model with height h and n nodes. The number of states of deterministic TSPA* constructed for the XML document D by Algorithm 4.2 is $\mathcal{O}(n^h)$.

Proof. On the i-th level, $i \geq 2$, of the tree model $T(D)$ we can choose all the node combinations that are "`LABEL` children" or "wildcard children" of each of the states formed on the $(i-1)$-th level, where `LABEL` is any of the l_i labels of the i-th level.

Let s be the number of states of TSPA* formed by the XML document D and $s_i, 1 \leq i \leq h$ be the number of states formed by node combinations of the i-th level. Clearly, $s_1 = 1$ and $\forall s_i \geq 2 : s_i \leq s_{i-1} \cdot (l_i + 1)$, since from each of the states formed on the previous level we can choose its "`LABEL` children" and its "wildcard children", so the number of transitions leading from each state is less than or equal to $l_i + 1$.

To get the estimated states number that can be formed on the i-th level we need to deal with the recursion. Obviously, $s_1 = 1$, $s_2 \leq l_2 + 1$, $s_3 \leq (l_2 + 1) \cdot (l_3 + 1)$, etc. As we can see now, $\forall s_i \geq 2 : s_i \leq \prod_{k=2}^{i-1} (l_k + 1)$.

So the number of all states of TSPA* formed for the XML document D is

$$s = 1 + \sum_{k=2}^{h} \prod_{i=2}^{k} (l_i + 1) =$$

$$= 1 + \prod_{i=2}^{2} (l_i + 1) + \prod_{i=2}^{3} (l_i + 1) + \ldots + \prod_{i=2}^{h} (l_i + 1)$$

It is obvious that $\forall\ j$, where $2 \leq j < h : \prod_{i=2}^{j} (l_i + 1) \leq \prod_{i=2}^{h} (l_i + 1)$, so the number of TSPA* states is

$$s \leq 1 + h \cdot \prod_{i=2}^{h} (l_i + 1) \leq 1 + h \cdot \prod_{i=2}^{h} n = 1 + h \cdot n^{h-1}$$

So the number of states of TSPA* can be bounded by $\mathcal{O}(n^h)$. □

Theorem 4.2. Let D be an XML document and $T(D)$ be its XML tree model with height h and n nodes. The number of transitions of deterministic TSPA* constructed for the XML document D with alphabet $A(D)$ by Algorithm 4.2 is $\mathcal{O}(n^h \cdot |A(D)|)$.

Proof. Maximum possible out-degree of each of the TSPA* state is $|A(D)|+1$, i.e., each state can have outgoing transitions on all possible labels from $A(D)$ and on the wildcard symbol (*). That is why the number of transitions of TSPA* can be estimated as $\mathcal{O}(n^h \cdot |A(D)|)$. □

Summarizing, we can bound the number of TSPA* states by $\mathcal{O}(n^h)$ and by $\mathcal{O}(h^k)$. These estimations can differ accordingly to the structure of the indexed XML document D, i.e., for a tree model $T(D)$ of maximum height $h = 10$ having $n = 100$ nodes and $k = 60$ leaves, better estimation for the states number of TSPA* is $\mathcal{O}(n^h)$ – $(n^h = 100^{10} = 10^{20}) \leq (h^k = 10^{60})$. On the contrary, for the $T(D)$ with only $k = 10$ leaves the $\mathcal{O}(h^k)$ estimation is better – $(h^k = 10^{10}) \leq (n^h = 100^{10} = 10^{20})$.

4.2 Tree String Path Subsequences Automaton for Selecting Unknown Nodes

Definition 4.4 (Tree String Path Subsequences Automaton for Selecting Unknown Nodes)**.** Let D be an XML document. The Tree String Path Subsequences Automaton for Selecting Unknown Nodes (TSPSA*) accepts all $XP^{\{//,*,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

TSPSA* speeds up the evaluation of linear XPath queries using descendant-or-self axis (//), wildcard (*) and *nodename* node tests (i.e., $XP^{\{//,*,nodename\}}$). We can represent such a fragment of XPath queries over an XML document D by the context-free grammar as follows:

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

To construct TSPSA*, we start with building deterministic finite automata that accept all non-empty subsequences and wildcard nodes combinations of individual string paths. Construction of a TSPSA* automaton for a single string path is described by Algorithm 4.3. First we need to present some useful definitions for the following algorithms notations.

Definition 4.5 (Set of occurrences of an element label in a string path)**.** Let $P = n_1 n_2 \ldots n_{|P|}$ 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 $O_P(e) = \{o \mid o = id(n_i) \;\wedge\; label(n_i) = e, \; i = 1, 2, \ldots, |P|\}$. The ordering is equal to ordering of element prefix identifiers as natural numbers.

Definition 4.6 (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 4.3. Construction of a deterministic TSPSA* automaton for a single string path.
Data: A string path $P = n_1 n_2 \ldots n_{|P|}$.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{//,*,nodename\}}$ 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. $Q \leftarrow \{q_0, q_1, \ldots, q_{|P|}\}$,
   $A \leftarrow \{//a : a \in (A(P) \cup \{*\})\}$,
   $F \leftarrow Q \setminus \{q_0\}$,
   $q_0 \leftarrow 0$.

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) $O_P(label(n_i)) \leftarrow ButFirst(O_P(label(n_i)))$.

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 $\delta(q_{s-1}, //a) = q_s \wedge \neg\exists r < s : \delta(q_{r-1}, //a) = q_r$,
   (ii) $\delta(q_i, //a) \leftarrow \emptyset$ otherwise.

4. Insert "wildcard" states and transitions into the automaton M:

   a. create a set of the element ids of the path P: $e(P) \leftarrow \{id(n_1), id(n_2), \ldots, id(n_{|P|})\}$ and create a state variable $prevState$.

   b. $\forall i$, where $i \leftarrow 1, 2, \ldots, |P|$:
   (i) if ($\exists\ q \in Q$: d-subset of $q = e(P)$): $q_{|P|+i} \leftarrow q$, else: create state $q_{|P|+i} \leftarrow e(P)$,
   (ii) add $\delta(q_{i-1}, //*) \leftarrow q_{|P|+i}$,
   (iii) if $i > 1$: copy all the transitions from state q_{i-1} to state $prevState$,
   (iv) $e(P) \leftarrow ButFirst(e(P))$,
   (v) $prevState \leftarrow q_{|P|+i}$.

Example 4.4. Consider an XML document D from Example 2.1 and its tree model $T(D)$ shown in Figure 2.1. Given P = `SERIES`(1) `US`(2) `ACTORS`(3) `FEMALE`(4) as the input string path, Algorithm 4.3 goes through following steps:

1. construction of a "backbone" of the TSPSA* automaton M for P, which is shown in Figure 4.6,

2. insertion of the "additional" transitions construction into the automaton M, resulting state of the automaton M after this step is shown in Figure 4.7,

3. insertion of the "wildcard" states and transitions into the automaton M, the complete TSPSA* automaton for the string path P is shown in Figure 4.8.

To build TSPSA*, we can run all the TSPSA* automata (constructed by Algorithm 4.3 for all string paths P_i in $P(T)$) "in parallel", by remembering the states of all automata while reading the input. This is achieved by the product construction.

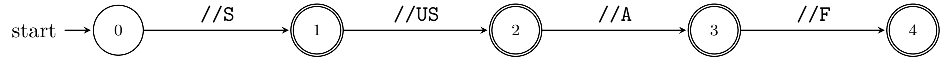


Figure 4.6: “Backbone” of the TSPSA* automaton for the string path P from Example 4.4

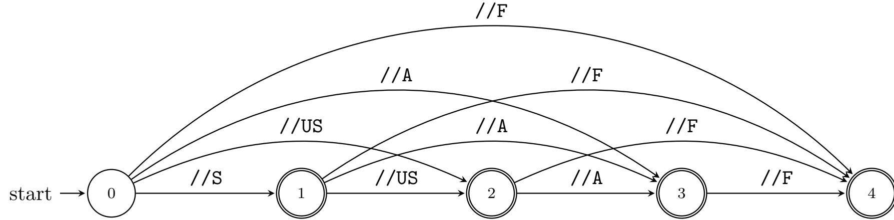


Figure 4.7: Not-completed TSPSA* automaton for the string path P from Example 4.4 after inserting “additional” transitions

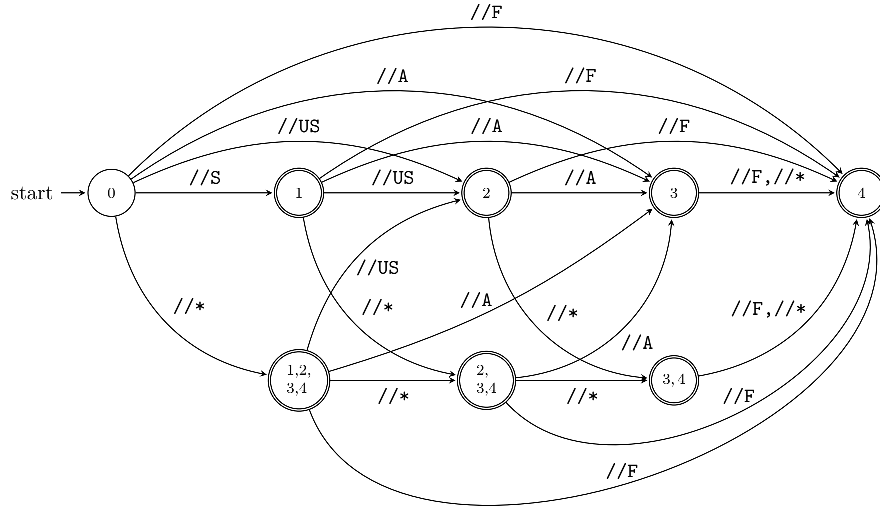


Figure 4.8: Completed TSPSA* automaton for the string path P from Example 4.4 after inserting “wildcard” states and transitions

Algorithm 4.4. Construction of the 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: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{//,*,nodename\}}$ queries of the XML document D.

1. For all $P_i \in P(T)$, construct a deterministic TSPSA* automaton $M_i =$

$(Q_i, \{//a : a \in (A(P) \cup \{*\})\}, \delta_i, 0, F_i)$ accepting all $XP^{\{//,*,nodename\}}$ queries of P_i using Algorithm 4.3.

2. Construct the TSPSA* $M = (Q, \{//a : a \in (A(D) \cup \{*\})\}, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{//,*,nodename\}}$ queries of the XML document D using the product construction (union) – see Definition 2.25.

TSPSA* for the XML document D from Example 2.1 is quite complex, so the state transition table of TSPSA* can be found in Appendix B.

4.2.1 Discussion of Time and Space Complexities

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

The main question is again the size of deterministic TSPSA*. We can bound the number of states of TSPSA* by size of the product of all the TSPSA* automata for individual string paths (which is created in Algorithm 4.4). Each of the TSPSA* automata for individual string paths contains maximum $(2h+1)$ states. This becomes clear from the Algorithm 4.3, where in the 2nd step we are adding $|P|+1$ states to the automaton and in the 4th step the maximum possible number of state additions is equal to $|P|$, and $|P|$ – the length of the string path – is obviously less than or equal to the height h of a tree model $T(D)$ of an XML document D. For k TSPSA* automata for individual string paths each containing less than or equal to $(2h+1) = \mathcal{O}(h)$ states, the number of TSPSA* states can be trivially bounded by $\mathcal{O}(h^k)$, where k is the number of leaves in the tree model $T(D)$. We can estimate the number of TSPSA* transitions as $\mathcal{O}(|A(D)| \cdot h^k)$, since the number of outgoing transitions for each state of TSPSA* is less than or equal to $|A(D)|+1$, where $|A(D)|$ is the size of the alphabet $A(D)$ of the XML document D and plus 1 is for the wildcard transition (`//*`).

Moreover, we can also estimate the size of TSPSA* created for an XML document D that satisfies l-property, i.e., an XML document in which all the nodes with the same labels are placed on one level of the tree model $T(D)$ of this XML document. Let us now show how the states of TSPSA* for the XML document D satisfying l-property can be created.

All the node combinations that form states in TSPSA* are, in fact, subtrees of the tree model $T(D)$ of the XML document D. These subtrees can even consist of one node, i.e., such subtrees have only the root node. Due to the fact that we are now talking about XML documents that satisfy l-property, all these subtrees, which can be selected, have root nodes on one level of the $T(D)$. This is illustrated in Example 4.5.

Definition 4.7 (Descendant types in TSPSA*)**.** Let $T(D)$ be an XML tree model of an XML document D with XML alphabet $A(D)$, q_i be a state (a set of subtrees) formed on the ith level (i.e., all the subtrees of the state q_i have root nodes on the ith level) of the $T(D)$. A state containing all the subtrees that can be selected from q_i using query $Q_1 =$ `//*` is called "wildcard descendant" of the state q_i. Analogically, a state that contains all the subtrees that can be selected from q_i using query $Q_2 =$ `//LABEL`, where `LABEL` $\in A(D)$, is called "`LABEL` descendant" of the state q_i.

In the following example we use a special notation for the TSPSA* states, e.g., $((\mathbf{3}-5),(\mathbf{6}))$ is a state containing two subtrees – one having the root node (3) and nodes (4) and (5) and the second containing only the root node (6).

Example 4.5. As it is shown in Figures 4.9, 4.10 and 4.11 different subtrees having roots on one level of the tree can be chosen. Note, that except subtrees whose selection is described in details below we can also choose every single node (a subtree having only root node) of the 3rd level of the shown tree, as all of them have unique paths leading to each one, for example, a state $((\mathbf{3}))$ can be chosen using queries `//SERIES//US//ACTORS` or `//*//US//ACTORS`.
Let us now describe how different subtrees having roots on the 3rd level of the tree can be selected. These subtrees sets form states in TSPSA*. Note, that states formed on the 2nd level of the tree are: $((\mathbf{2}))$, $((\mathbf{7}))$ and $((\mathbf{2}-6),(\mathbf{7}-11))$.

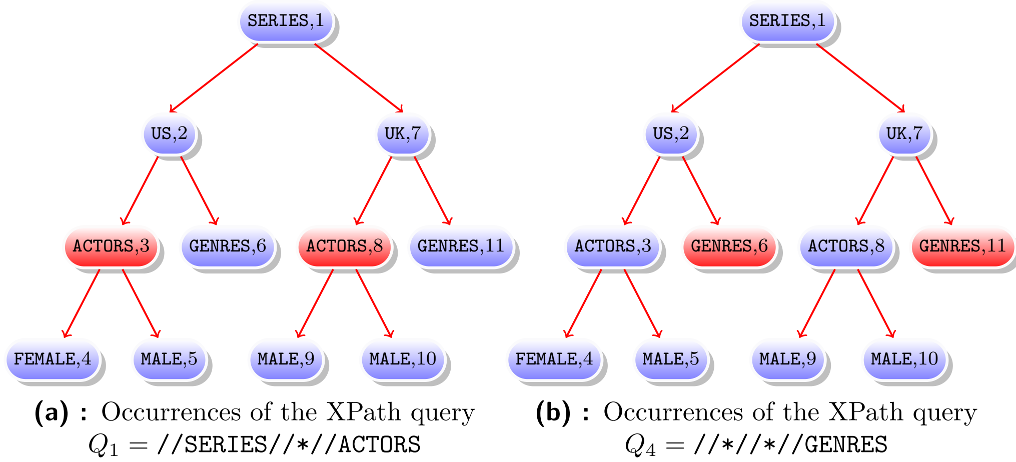
- a state $((\mathbf{3}),(\mathbf{8}))$ can be chosen as "`ACTORS` descendant" of the state $((\mathbf{2}-6),(\mathbf{7}-11))$ using queries $Q_1 =$`//SERIES//*//ACTORS` or $Q_2 =$`//* //*//ACTORS` – see Figure 4.9a, where occurrences of the query Q_1 are represented,

- similarly we can choose a state $((\mathbf{6}),(\mathbf{11}))$ as it is "`GENRES` descendant" of the state $((\mathbf{2}-6),(\mathbf{7}-11))$, to choose them we use following queries: $Q_3 =$`//SERIES//* //GENRES` or $Q_4 =$`//*//*//GENRES` (Figure 4.3b shows occurrences of the query Q_4),

- we can select such state as $((\mathbf{3}-5),(\mathbf{6}))$ ("wildcard descendant" of the state $((\mathbf{2}))$) with the help of queries $Q_5 =$`//SERIES//US //*` or $Q_6 =$`//*//US//*`, it is shown in Figure 4.10a – occurrences of the query Q_5,

- in the same way state $((\mathbf{8}-10),(\mathbf{11}))$ can be selected using queries $Q_7 =$`//SERIES //UK//*` or $Q_8 =$`//*//UK//*`, as it is "wildcard descendant" of the state $((\mathbf{7}))$, see Figure 4.10b – occurrences of the query Q_8,

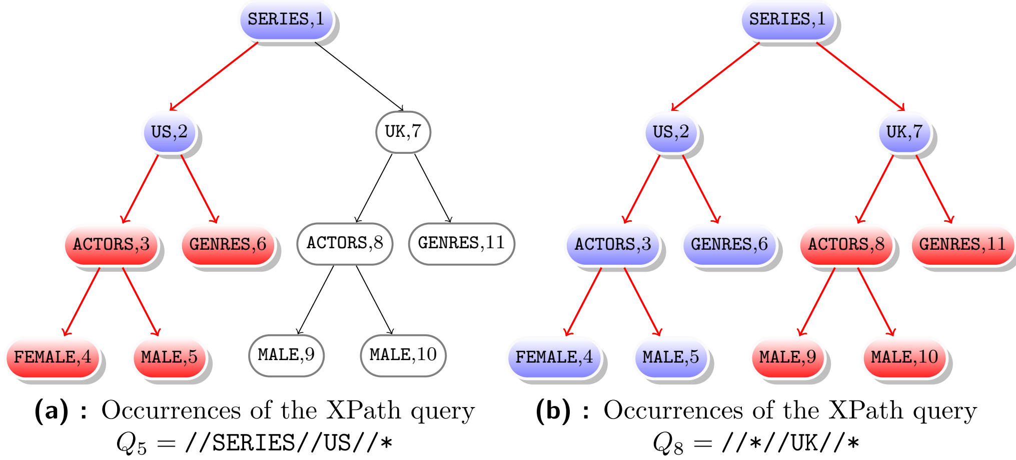
- finally, as it is shown in Figure 4.11 we can choose a state $((\mathbf{3}-5),(\mathbf{6}),(\mathbf{8}-10),(\mathbf{11}))$ using query $Q_9 =$ `//*//*//*` – this state is so called "wildcard descendant" of the state $((\mathbf{2}-6),(\mathbf{7}-11))$.



(a) : Occurrences of the XPath query Q_1 = `//SERIES//*//ACTORS`

(b) : Occurrences of the XPath query Q_4 = `//*//*//GENRES`

Figure 4.9: Occurrences of XPath queries Q_1 and Q_4 from Example 4.5



(a) : Occurrences of the XPath query Q_5 = `//SERIES//US//*`

(b) : Occurrences of the XPath query Q_8 = `//*//UK//*`

Figure 4.10: Occurrences of XPath queries Q_5 and Q_8 from Example 4.5

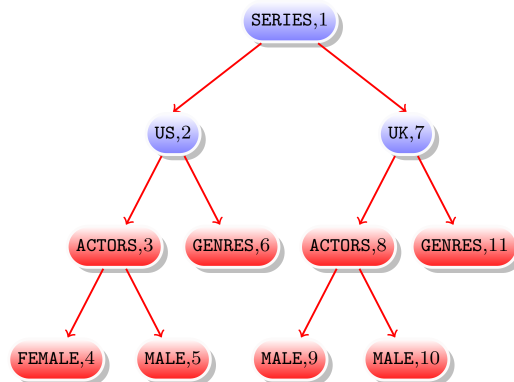


Figure 4.11: Occurrences of the XPath query Q_5 = `//*//*//*` from Example 4.5

We definitely need to note one more thing before we get to the TSPSA* size estimation. Each state formed on the ith level of a tree model can be a

"LABEL descendant" not only of some state from the $(i-1)$-th level of the tree, but also of some states on the previous tree levels (e.g., state $((\mathbf{3}),(\mathbf{8}))$ from Example 4.5 is also an "ACTORS descendant" of states $((\mathbf{1}))$ and $((\mathbf{1}-11))$ formed on the 1st level of the shown tree). However, no new states can be added to TSPSA* being descendants of some of the previous level states and not being descendants of the closest previous level state(-s).

Consider an XML document D of size n and its tree model $T(D)$ of height h. As we can now see, analogically to TSPA* case, the number of states of TSPSA* (satisfying l-property) that are formed on the ith level of a tree model $T(D)$ of an XML document D is $\forall i \in 2,3,\cdots,h : s_i = s_{i-1} \cdot (l_i+1)$, where $l_i + 1$ is the number of unique labels on the ith level of the $T(D)$ plus the wildcard symbol. So we can estimate the TSPSA* size as $\mathcal{O}(n^h)$, it can be proved analogically to the Theorem 4.2 proof. And the number of the TSPSA* transitions is bounded by $\mathcal{O}(|A(D)| \cdot n^h)$, where $|A(D)|$ is the size of the XML document D alphabet.

Analogically to the TSPA* case, we can bound the number of TSPSA* states by $\mathcal{O}(n^h)$ and by $\mathcal{O}(h^k)$. These estimations can differ accordingly to the structure of the indexed XML document D, i.e., for a tree model $T(D)$ of maximum height $h = 10$ having $n = 100$ nodes and $k = 60$ leaves, better estimation for the states number of TSPA* is $\mathcal{O}(n^h)$ – $(n^h = 100^{10} = 10^{20}) \leq (h^k = 10^{60})$. On the contrary, for the $T(D)$ with only $k = 10$ leaves the $\mathcal{O}(h^k)$ estimation is better – $(h^k = 10^{10}) \leq (n^h = 100^{10} = 10^{20})$.

4.3 Tree Paths Automaton for Selecting Unknown Nodes

The following part of the thesis is not mentioned in the assignment, but nevertheless, we describe the construction of the automaton that accepts all $XP^{\{/,//,*,nodename\}}$ queries as it is a good way to show how the previous automata (TSPA* and TSPSA*) can be used for construction of a much more complex index automaton.

Definition 4.8 (Tree Paths Automaton for Selecting Unknown Nodes)**.** Let D be an XML document. The Tree Paths Automaton for Selecting Unknown Nodes (TPA*) accepts all $XP^{\{/,//,*,nodename\}}$ queries of D, and for each query Q, it gives a list of elements satisfying Q.

TPA* speeds up the evaluation of linear XPath queries using child-axis (/), descendant-or-self axis (//) and wildcard symbol (*) (i.e., $XP^{\{/,//,*,nodename\}}$). We can represent such a fragment of XPath queries over an XML document D by the context-free grammar as follows:

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

To build an automaton supporting all the $XP^{\{/,//,*,nodename\}}$ we combine principles of both introduced earlier automata, i.e., TSPA* and TSPSA*. Since both $XP^{\{/,*,nodename\}}$ and $XP^{\{//,*,nodename\}}$ queries are subsets of $XP^{\{/,//,*,nodename\}}$ queries, they are supported by TPA*.

To provide an algorithm for building TPA*, we first propose a building algorithm that combines TSPA* and TSPSA* automata for a single string path P to support all $XP^{\{/,//,*,nodename\}}$ queries of P. See Algorithm 4.5 and Example 4.6.

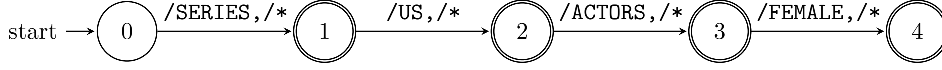


Figure 4.12: TSPA* for the string path P from Example 4.6

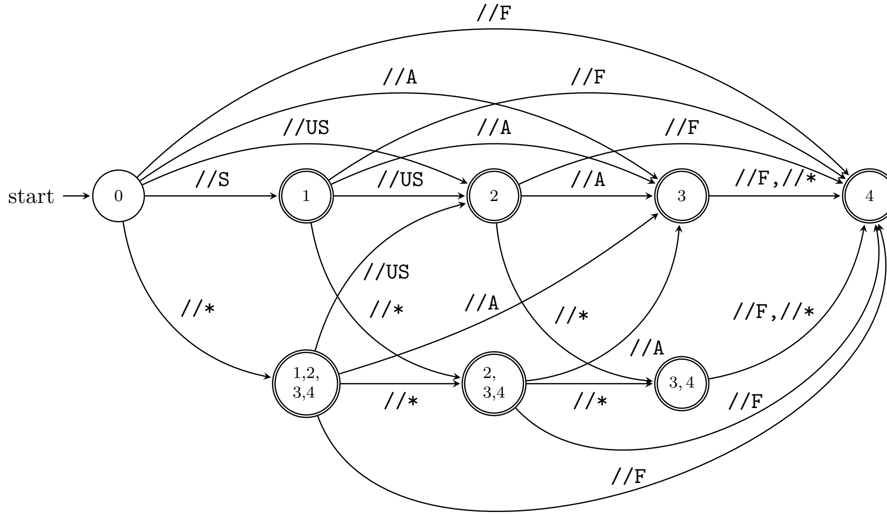


Figure 4.13: TSPSA* for the string path P from Example 4.6

Algorithm 4.5. Construction of the TPA* for a single string path.
Data: A string path $P = n_1 n_2 \ldots n_{|P|}$.
Result: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,//,*,nodename\}}$ queries of P.

1. Construct TSPA* $M_1 = (Q_1, A_1, \delta_1, 0, F_1)$ accepting all $XP^{\{/,*,nodename\}}$ queries of P using Algorithm 4.2.

2. Construct TSPSA* $M_2 = (Q_2, A_2, \delta_2, 0, F_2)$ accepting all $XP^{\{//,*,nodename\}}$ queries of P using Algorithm 4.4.

3. Construct a deterministic finite automaton $M = (Q, A_1 \cup A_2, \delta, 0, F)$ accepting all $XP^{\{/,//,*,nodename\}}$ queries of P using the product construction (union).

4. Add missing transitions to the automaton $M = (Q, A, \delta, 0, F)$ as follows:

```
create a new queue S and initialize S = Q;
while S is not empty do
    state q ← S.pop;
    d ← q.d-subset;
    if q is the initial state of M then
        continue;
    end
                                                    ▷ add // transitions
    if d.Counts = 1 then
        find first state p ∈ Q : p.d-subset.Counts > 1
        ∧ p.d-subset.First = d.First;
        copy all the // transitions from state p to state q;
    end
                                                    ▷ add / transitions
    else
        create new set of states consistsOf;
        consistsOf ← ∀p ∈ Q : p.d-subset.Counts = 1 ∧ p.d-subset⊂ d;
        create new d-subset d_new;
        foreach a ∈ A do
            foreach c ∈ consistsOf do
                if d-subset of s ⊂ d then
                    if δ(q, a) = ∅ and δ(s, a) ≠ ∅ then
                        d_new.Add(δ(s, a));
                    end
                end
            end
            if d_new is not empty then
                δ(q, a) ← d_new;
                find s ∈ Q : s.d-subset= d_new;
                S.push(s);
            end
        end
    end
end
```

Example 4.6. Consider an XML document D from Example 2.1 and its tree model $T(D)$ shown in Figure 2.1. Given P = `SERIES`(1) `US`(2) `ACTORS`(3) `FEMALE`(4) as the input string path, Algorithm 4.5 goes through following steps:

1. construction of a deterministic TSPA* for P, which is shown in Figure 4.12,
2. construction of a deterministic TSPSA* for P, which is shown in Figure 4.13,
3. construction of a "not-finished" TPA* for the string path P using the product construction (union), see Figure 4.14. It is the "not-finished"

TPA*, since some states of the TSPSA* do not still have transitions of type `/LABEL` and `/*` and vice versa.

4. addition of the missing transitions, the resulting TPA* for P is shown in Figure 4.15.

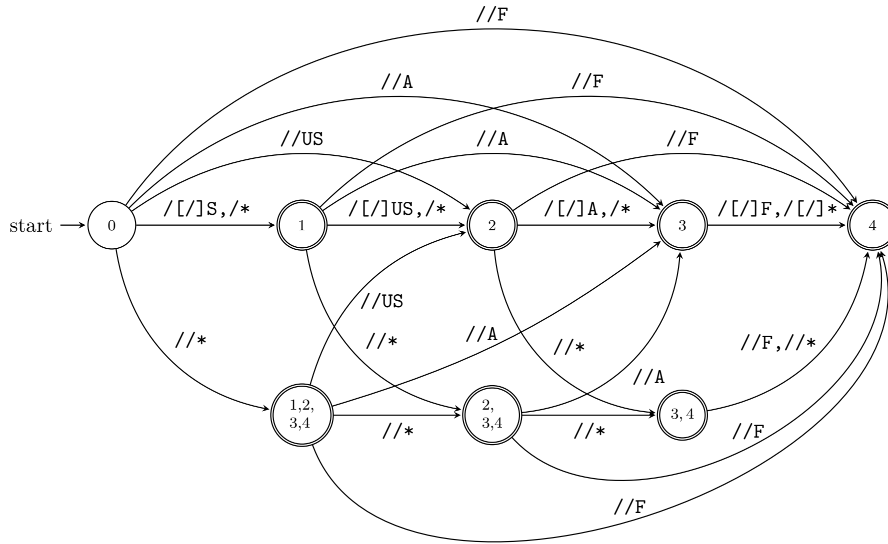


Figure 4.14: "Not-finished" TPA* for the string path P from Example 4.6

To build TPA* for the XML document D, we can again use the product construction of the automata constructed by Algorithm 4.5 for individual string paths.

Algorithm 4.6. Construction of the TPA* 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: DFA $M = (Q, A, \delta, 0, F)$ accepting all $XP^{\{/,//,*,nodename\}}$ queries of the XML document D.

1. For all $P_i \in P(T)$, construct a finite automaton $M_i = (Q_i, \{/a, //a : a \in (A(P) \cup \{*\})\}, \delta_i, 0, F_i)$ accepting all $XP^{\{/,//,*,nodename\}}$ queries of P_i using Algorithm 4.6.

2. Construct the deterministic TPA* $M = (Q, \{/a, //a : a \in (A(P) \cup \{*\})\}, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{/,//,*,nodename\}}$ queries of the XML document D using the product construction (union).

Figure 4.15: TPA* for the string path P from Example 4.6

Since the diagram of TPA* for the XML document D from Example 2.1 is too complex to place in this thesis, the transition table can be found in Appendix C.

4.3.1 Discussion of Time and Space Complexities

Consider an XML document D containing n nodes and its tree model $T(D)$ of height h. According to Algorithm 4.5, each of the formed by this algorithm TPA* for individual string paths contains $\mathcal{O}(h^2)$ states.

Theorem 4.3. The resulting automaton of the Algorithm 4.5 contains $\mathcal{O}(h^2)$ states.

Proof. The only step of the considered algorithm, where new states are added to the resulting TPA* for a single string path, is the 3rd step. In the 3rd step of the algorithm two automata constructed in the previous two steps are run in parallel.

The prefix automaton constructed in the first step by Algorithm 4.1 consists of $i \leq h$ states. The subsequences automaton constructed in the second step by Algorithm 4.3 consists of $j \leq 2 \cdot h$ states. Therefore, the product of these two automata contains maximum $2 \cdot h^2$ states, so the number of states of the TPA* for a single string path can be bounded by $\mathcal{O}(h^2)$. □

The complete TPA* for the XML document D number of states can be bounded by size of the product of all the TPA* for single string paths. The number of such automata for single string paths is obviously equal to the

number of leaves k of the tree model $T(D)$. So we can bound the number of states of TPA* by $\mathcal{O}(h^{2 \cdot k})$.

In the proof of Theorem 4.3 we assume that the mentioned algorithms are correct and though we do not prove their correctness in this thesis, these algorithms are implemented and tested.

Theorem 4.4. TPA* for the XML document D contains $\mathcal{O}(h^{2 \cdot k} \cdot |A(D)|)$ transitions, where $|A(D)|$ is power of the XML alphabet of the XML document D.

Proof. Each state of TPA* can have maximum $|A(D)|$ out-going transitions of `/LABEL` type, maximum $|A(D)|$ out-going transitions of `//LABEL` type and both `/*` and `//*` transitions. That is why, we can estimate the number of transitions of TPA* as $\mathcal{O}(h^{2 \cdot k} \cdot 2 \cdot (|A(D)| + 1) = \mathcal{O}(h^{2 \cdot k} \cdot |A(D)|)$. □

Chapter 5

Implementation

In this chapter the main aspects of the implementation introduced in previous chapter methods for indexing XML documents are described. All three methods for building indexes of XML documents are implemented and will hopefully be a part of some big library including different algorithms for indexing XML data later. Since Tree Paths Automaton for Selecting Unknown Nodes (TPA*) is designed to be able to resolve the biggest subset of XPath queries and its building process involves the main steps of the TSPA* and TSPSA* construction (such as building TSPA* and TSPSA* automata for individual string paths), in this and the following chapter we describe, test and conduct experiments on TPA* only.

5.1 System Architecture

The XML data index software is developed in Java SE, JDK 8u171 in the IntelliJ IDEA IDE [4], it is designed as Java Class Library called `tpaUNlib`. The system architecture is shown in Figure 5.1. The library consists of three virtual parts: Document Parser, Index Builder and XML Data Index.

5.2 Document Parser

This virtual part of the library is used for loading an XML file and parsing it using the Simple API for XML (SAX) [13]. The `DocumentParser` class (Figure 5.2) contains method `getStringPathsSet` that takes a path to an XML file D as an input and returns a set of string paths of a tree model of D. These string paths are then used in the following steps of the program. This part of code was written by Lukáš Renc as a part of the future library of the whole "Automata Approach to XML Data Indexing" project in his bachelor's thesis [6].

5.3 Index Builder

The second part of the implementation is Index Builder, it is clearly used for building index automata. It disposes of `AutomatonFactory` class (see

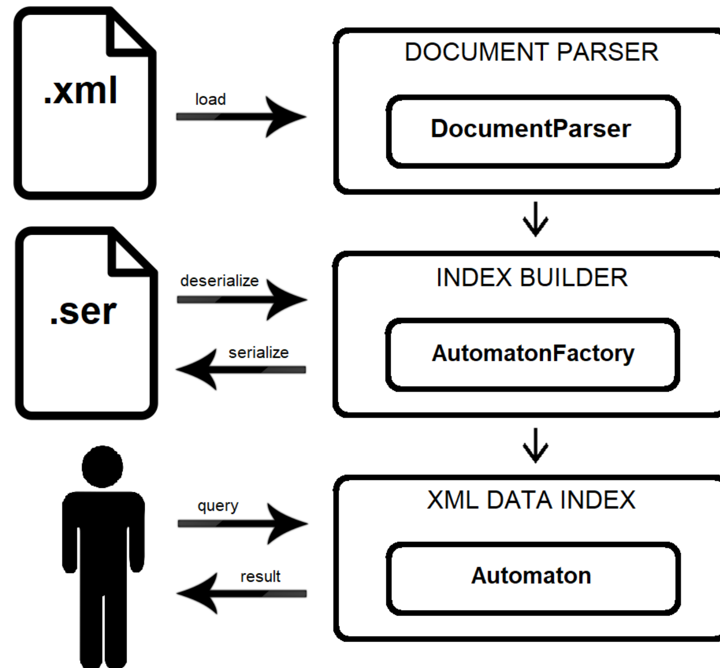


Figure 5.1: System Architecture of the `tpaUNlib`

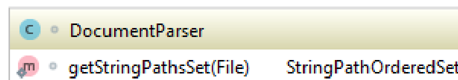


Figure 5.2: `DocumentParser` class

Figure 5.3) that presents construction methods for all the introduced in this thesis index automata using the string paths set, built in previous step. These methods are: `buildTSPAUnknownNodes`, `buildTSPSAUnknownNodes` and `buildTPAUnknownNodes`. All these methods construct the XML Data Index of a certain type and returns it as an instance of `Automaton` class to a user.

The construction process is totally the same as it is described in the Chapter 4 for each automaton index type. The product construction of automata is implemented using multithreading to increase speed of the construction process. Threads are implemented as objects of a `ParallelRunner` class whose structure is shown in Figure 5.3.

Moreover, the `AutomatonFactory` allows us to serialize the built index automaton for further usages and deserialize it later instead of building the index again. This helps us to save time when we are working with huge XML documents.

AutomatonFactory	
getTSPAUnknownNodes(File)	Automaton
getTSPSAUnknownNodes(File)	Automaton
getTPAUnknownNodes(File)	Automaton
deserialize(String)	Automaton
buildTSPAUnknownNodes(File)	Automaton
buildTSPSAUnknownNodes(File)	Automaton
buildTPAUnknownNodes(File)	Automaton
parallelParallelRun(List<Automaton>, AutomatonType)	Automaton
parallelParallelRun(List<Automaton>, AutomatonType, int)	Automaton
parallelRun(List<Automaton>)	Automaton
buildTSPAUNStringPath(StringPath)	Automaton
buildTSPSAUNStringPath(StringPath)	Automaton
buildTPAUNStringPath(StringPath)	Automaton

ParallelRunner	
automata	ArrayList<Automaton>
automaton	Automaton
automatonType	AutomatonType
imDone	Boolean
ParallelRunner(ArrayList<Automaton>, AutomatonType)	
run()	void
parallelRun(List<Automaton>, AutomatonType)	Automaton
mainCycleParallelRun(LinkedList<State>, Automaton)	void

Figure 5.3: `AutomatonFactory` and `ParallelRunner` classes

5.4 XML Data Index

The third part of the system architecture is the XML Data Index. Here we already have an index automaton built (or desrialized) by the Index Builder. This index automaton is an instance of `Automaton` class (Figure 5.4).

Automaton	
firstState	State
states	ArrayList<State>
automatonType	AutomatonType
Automaton(State, AutomatonType)	
getNextState(State)	State
getState(ArrayList<XMLTag>)	State
getStateMinID(int)	State
getStatesUniqueID(ArrayList<XMLTag>)	HashSet<State>
resolveQuery(String)	ArrayList<XMLTag>
serialize(String)	void
toString()	String

Figure 5.4: `Automaton` class

The `Automaton` class contains all information about the built index and the main part of this information are the automaton states. Each state is an instance of the `State` class, which keeps information about its `XMLTags` and the outgoing transitions. Instances of the `XMLTag` class are basically nodes of the document D, each having its unique ID and some label. Finally, `Transition` class keeps information about "from" state, "to" state and the transition phrase. Class diagrams of the `State`, `Transition` and `XMLTag` classes are shown in Figure 5.5.

Moreover, having TPA* built, a user can process a query. For this aim he uses the `Automaton` class method `resolveQuery` that gets a query in a string format as an input and returns a list of instances of nodes (`XMLTag` classes instances). The input query is split onto single transition phrases. The evaluation is, in fact, running the deterministic TPA* and getting the required result.

State	
label	String
transitions	HashSet<Transition>
xmlTags	TreeMap<Integer, XMLTag>
consistsOfStates	HashSet<State>
State(XMLTag)	
State()	
State(ArrayList<XMLTag>)	
getLabelFromXMLTags(ArrayList<XMLTag>)	String
getXmlTags()	ArrayList<XMLTag>
getTransitionByPhrase(String, TransitionType)	Transition
addTransition(State, String)	void
addTransition(State, String, TransitionType)	void
addWildcardTransition(State)	void
addWildcardTransition(State, TransitionType)	void
hasTransitionOn(String, TransitionType)	Boolean
copyTransitions(State)	void
copyTransitionsOfType(State, TransitionType)	void
addModifyTransition(Transition)	void
getMinXMLTag()	int
inCollection(Collection<State>)	Boolean
getFromCollection(Collection<State>)	State
getXMLTagsIDs()	Set<Integer>
getXMLTagsIDs(TreeMap<Integer, XMLTag>)	Set<Integer>
copyStateNoTransitions(State)	State
addXMLTags(ArrayList<XMLTag>)	void
addXMLTag(XMLTag)	void
toString()	String

Transition	
from	State
to	State
phrase	String
transitionType	TransitionType
Transition(State, State, String)	
Transition(State, State, String, TransitionType)	
containsPhrase(String)	boolean
toString()	String

XMLTag	
name	String
prefixID	int
XMLTag(String, int)	
getPrefixID()	int
getName()	String
toString()	String

Figure 5.5: `State`, `Transition` and `XMLTag` classes

Chapter 6

Testing and Experimental Evaluation

In this chapter the testing process is described. Moreover, the experimental evaluation of the implementation discussed in the previous chapter is introduced.

6.1 Experimental Setup

The experiments over the implementation are conducted under the environment of Intel Core i7 CPU @ 4.20 GHz, 16 GB RAM and 240 GB SSD disk with the Window 10 operating system running.

For our experimental evaluation, we selected the XMark datasets [8] that were generated by `xmlgen` [16] using scaling factors $0, 0.001, 0.005, 0.01, 0.1, 0.5$. Information about these datasets is shown in Table 6.1. XMark is a synthetic on-line auction dataset, it has large and complicated tree structure [10].

Key	File name	Size (MB)	# of Elements	# of Leaves	Max-depth	Avg-depth
D_1	XMark-f0.xml	0.03	382	247	10	4.64
D_2	XMark-f0.001.xml	0.10	1,729	1,204	11	4.69
D_3	XMark-f0.005.xml	0.50	8,518	6,211	11	4.50
D_4	XMark-f0.01.xml	1.16	17,132	12,504	11	4.51
D_5	XMark-f0.1.xml	11.60	167,865	122,026	11	4.55

Table 6.1: Characteristics of the datasets

We use a set of queries for our testing and experimental aims. All the queries are shown in Table 6.2. Note, that the queries Q_1, Q_2, Q_3 use only child axes (/), queries Q_4, Q_5, Q_6 – only descendant-or-self axes (//) and the three last queries Q_7, Q_8, Q_9 use both child and descendant-or-self axis. All the queries contain wildcard node-tests, as we are testing primarily the wildcard queries evaluation.

The reference numbers of elements satisfying are generated using SAXON API [13] and are represented in Table 6.3. The results generated by our implementation are exactly the same as these reference numbers are.

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

Table 6.2: Set of queries for XMark datasets

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
D_1	6	1	5	7	5	7	1	6	4
D_2	6	25	10	296	20	14	12	128	6
D_3	6	127	29	1,591	124	57	60	642	28
D_4	6	255	85	3,088	205	185	120	1,270	107
D_5	6	2,550	922	30,116	2,139	1,691	1,200	12,686	1,069

Table 6.3: Numbers of elements satisfying the queries in the datasets

6.2 Performance of TPA* Construction

The Table 6.4 shows the experimental results on the TPA* index size and its construction time for the datasets $D_1 - D_6$. The size of the index is measured by the size of the serialized `Automata` object.

The Index/XML Size (ratio) column in the Table 6.4 shows that the index size is about 80 times bigger than the original XML file, this fact is leading to a suggestion that the index size stays linear. However, this suggestion should be explored more on other datasets.

Key	Index Size (MB)	Index/XML Size (ratio)	# of States	# of Transitions	Construction Time
D_1	2.40	80.00	599	4,715	0.4 sec
D_2	9.53	95.30	1,205	8,218	1.2 sec
D_3	43.79	87.58	1,957	11,213	4.1 sec
D_4	88.26	76.10	2,259	12,344	10.5 sec
D_5	862.72	74.37	2,778	13,815	3 min

Table 6.4: Experimental results on the index size and construction time

Moreover, the Table 6.4 shows the number of states and transitions of the constructed indexes. These values are obviously mostly influenced by the structure of the XML document and by its size.

6.3 Performance of Query Processing

We analyze the performance of query evaluation in comparison with a reference implementation called Saxon [7]. The measurements reflect query processing time only.

Figure 6.1 shows the experimental results of TPA* and Saxon on the datasets $D_1 - D_5$ from the Table 6.1.

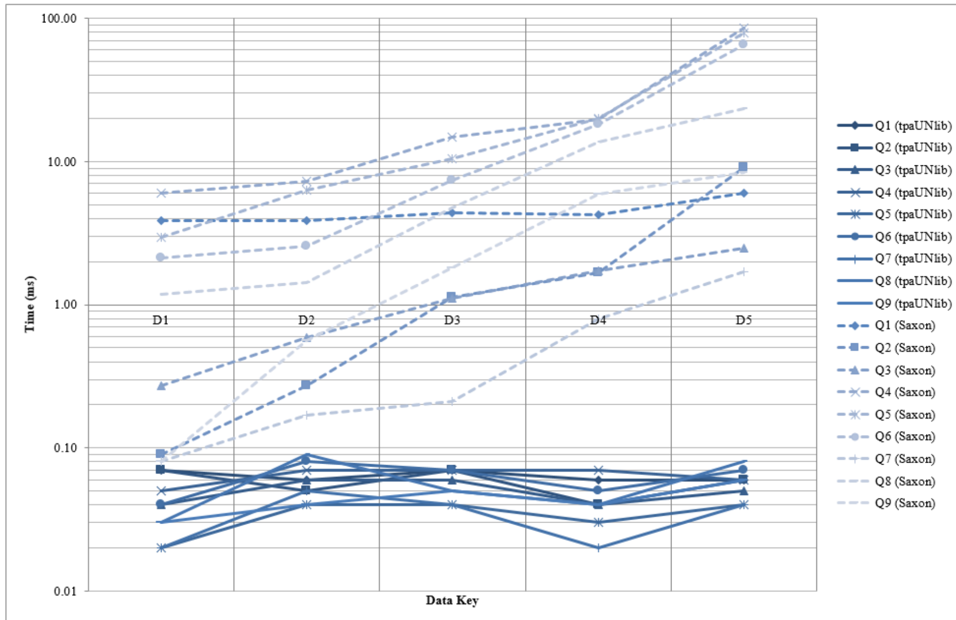
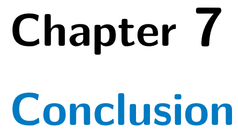


Figure 6.1: Performance comparison of TPA* and Saxon

We should note, that the implementation of the query resolving of the TPA* index returns only ids of the XML document elements, while Saxon returns the actual XML elements satisfying the input query. That is why in Figure 6.1 we compare searching phase of TPA* query resolving with searching and answering phase of Saxon query resolving. The answering phase will be implemented and tested during future work on the project.

However, Figure 6.1 shows that the searching phase of TPA* query resolving does not depend on the growing datasets size, this fact confirms our assumption that the searching phase of TPA* depends only on the processed query size.



Chapter 7
Conclusion

In this chapter we evaluate the fulfillment of the thesis goals and present the main directions of the future work.

7.1 Goals Fulfillment

One of the goals of the thesis was to study the existing indexing methods based on automata theory, such as Tree String Paths Automaton (TSPA) ans Tree String Path Subsequences Automaton (TSPSA). This is covered by Chapter 3. Moreover, in this chapter the main achievement of the previous work – Tree Paths Automaton (TPA) – is also studied and described.

The main aim of this thesis was to extend the existing methods so they would also support XPath wildcards (*) to select unknown XML nodes. This aim is fulfilled in Chapter 4. We have also discussed time and space complexities of all of the newly presented methods.

Finally, the created methods are implemented and tested, the appropriate experimental evaluation is performed. Details can be found in Chapter 5 and Chapter 6.

7.2 Contribution of the Thesis

The main contribution of this thesis is creation of new XML data indexing methods that are based on automata theory. These methods extend the subset of XPath queries that can be accepted by automata created within the "Automata Approach to XML Data Indexing" project. We have also shown that these index automata are able to efficiently process XPath queries in time that does not depend on the indexed document size, but only on the processed query size.

7.3 Future Work

There is still a lot of work to do, as this thesis and previous works on these theme are not covering all the existing problems related to the automata-based XML indexing methods. The main tasks for future are to:

- proof the correctness of the described algorithms for creating index automata. This can also help to determine the tight upper bound on number of states and transitions for Tree Paths Automaton [9].
- create a global library that will support all the existing methods for XML data indexing based on automata theory and will be easily extensible with the new ones.
- improve the implementation to make it more memory efficient, as the indexes are growing with every new extension of the algorithms.
- extend the existing methods to support more complex queries, e.g., including attributes, branching etc.

Bibliography

[1] Bray T. Paoli J. Sperberg-McQueen C. et al. *Extensible Markup Language (XML) 1.0*. online. URL: http://www.w3.org/XML (visited on 05/04/2018).

[2] Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory, Second Edition*. 2nd. Chapman & Hall/CRC, 2013. ISBN: 1439880182, 9781439880180.

[3] John E. Hopcroft et al. *Introduction to Automata Theory, Languages and Computability*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201441241.

[4] JetBrains. *IntelliJ IDEA*. online. URL: https://www.jetbrains.com/idea/ (visited on 05/04/2018).

[5] Michael O Rabin and Dana Scott. "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3 (Apr. 1959), pp. 114–125.

[6] Lukáš Renc. "Automata Approach to XML Data Indexing: Implementation and Experimental Evaluation". Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

[7] SAXONICA. *SAXON – The XSLT and XQuery Processor*. online. URL: http://saxon.sourceforge.net/ (visited on 05/06/2018).

[8] Schimdt, et al. *XMark – An XML Benchmark Project*. online. URL: http://www.xml-benchmark.org/ (visited on 05/04/2018).

[9] Eliška Šestáková. "Indexing XML Documents". Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

[10] Eliška Šestáková and Jan Janoušek. "Automata Approach to XML Data Indexing". In: *Information* 9.1 (2018). ISSN: 2078-2489. DOI: 10.3390/info9010012.

[11] Eliška Šestáková and Jan Janoušek. “Indexing XML Documents Using Tree Paths Automaton”. In: *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Ed. by Ricardo Queirós et al. Vol. 56. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:14. ISBN: 978-3-95977-056-9. DOI: 10.4230/OASIcs.SLATE.2017.10. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7945.

[12] Eliška Šestáková and Jan Janoušek. “Tree String Path Subsequences Automaton and Its Use for Indexing XML Documents”. In: *Languages, Applications and Technologies*. Ed. by José-Luis Sierra-Rodríguez, José-Paulo Leal, and Alberto Simões. Cham: Springer International Publishing, 2015, pp. 171–181. ISBN: 978-3-319-27653-3.

[13] *Simple API for XML*. online. URL: http://www.saxproject.org/ (visited on 05/04/2018).

[14] W3Schools. *XML Validator*. online. URL: https://www.w3schools.com/xml/xml_validator.asp (visited on 04/30/2018).

[15] W3Schools. *XPath Tutorial*. online. URL: https://www.w3schools.com/xml/xpath_intro.asp (visited on 04/27/2018).

[16] Florian Waas. *xmlgen – faq*. online. URL: http://www.xml-benchmark.org/faq.txt (visited on 05/04/2018).

Appendix A

Acronyms

- **DFA** Deterministic finite automaton
- **NFA** Nondeterministic finite automaton
- **TPA** Tree Paths Automaton
- **TSPA** Tree String Paths Automaton
- **TSPSA** Tree String Path Subsequences Automaton
- **TPA*** Tree Paths Automaton for Selecting Unknown Nodes
- **TSPA*** Tree String Paths Automaton for Selecting Unknown Nodes
- **TSPSA*** Tree String Path Subsequences Automaton for Selecting Unknown Nodes
- **XML** eXtensible markup language
- **XPath** XML Path Language
- **SAX** Simple API for XML

Appendix B

Tree String Path Subsequences Automaton for Selecting Unknown Nodes

	`//S`	`//US`	`//UK`	`//A`	`//G`	`//F`	`//M`	`//*`
$\rightarrow (0)$	(1)	(2)	(7)	$(3,8)$	$(6,11)$	(4)	$(5,9,10)$	$(1-11)$
$\leftarrow (1)$		(2)	(7)	$(3,8)$	$(6,11)$	(4)	$(5,9,10)$	$(2-11)$
$\leftarrow (2)$				(3)	(6)	(4)	(5)	$(3-6)$
$\leftarrow (7)$				(8)	(11)		$(9,10)$	$(8-11)$
$\leftarrow (3,8)$						(4)	$(5,9,10)$	$(4,5,9,10)$
$\leftarrow (6,11)$								
$\leftarrow (4)$								
$\leftarrow (5,9,10)$								
$\leftarrow (1-11)$		(2)	(7)	$(3,8)$	$(6,11)$	(4)	$(5,9,10)$	$(2-11)$
$\leftarrow (2-11)$				(3)	(6)	(4)	(5)	$(3-6,8-11)$
$\leftarrow (3)$						(4)	(5)	$(4,5)$
$\leftarrow (6)$								
$\leftarrow (5)$								
$\leftarrow (3-6)$						(4)	(5)	$(4,5)$
$\leftarrow (8)$							$(9,10)$	$(9,10)$
$\leftarrow (11)$								
$\leftarrow (9,10)$								
$\leftarrow (8-11)$							$(9,10)$	$(9,10)$
$\leftarrow (4,5,9,10)$								
$\leftarrow (3-6,8-11)$						(4)	$(5,9,10)$	$(4,5,9,10)$
$\leftarrow (4,5)$								

Table B.1: Transition table of TSPSA* for the XML document D from Example 2.1

Appendix C

Tree Paths Automaton for Selecting Unknown Nodes

	/S	//S	/US	//US	/UK	//UK	/A	//A	/G	//G	/F	//F	/M	//M	/*	//*
→ (0)	(1)	(1)		(2)		(7)		(3, 8)		(6, 11)		(4)		(5, 9, 10)	(1)	(1 − 11)
← (1)			(2)	(2)	(7)	(7)		(3, 8)		(6, 11)		(4)		(5, 9, 10)	(2, 7)	(2 − 11)
← (2)							(3)	(3)	(6)	(6)		(4)		(5)	(3, 6)	(3 − 6)
← (7)							(8)	(8)	(11)	(11)				(9, 10)	(8, 11)	(8 − 11)
← (3, 8)											(4)	(4)	(5, 9, 10)	(5, 9, 10)	(4, 5, 9, 10)	(4, 5, 9, 10)
← (6, 11)																
← (4)																
← (5, 9, 10)																
← (1 − 11)			(2)	(2)	(7)	(7)	(3, 8)	(3, 8)	(6, 11)	(6, 11)	(4)	(4)	(5, 9, 10)	(5, 9, 10)	(2 − 11)	(2 − 11)
← (2, 7)							(3, 8)	(3, 8)	(6, 11)	(6, 11)		(4)		(5, 9, 10)	(3, 6, 8, 11)	(3 − 6, 8 − 11)
← (2 − 11)							(3, 8)	(3, 8)	(6, 11)	(6, 11)	(4)	(4)	(5, 9, 10)	(5, 9, 10)	(3 − 6, 8 − 11)	(3 − 6, 8 − 11)
← (3)											(4)	(4)	(5)	(5)	(4, 5)	(4, 5)
← (6)																
← (5)																
← (3, 6)											(4)	(4)	(5)	(5)	(4, 5)	(4, 5)
← (3 − 6)											(4)	(4)	(5)	(5)	(4, 5)	(4, 5)
← (8)													(9, 10)	(9, 10)	(9, 10)	(9, 10)
← (11)																
← (9, 10)																
← (8, 11)													(9, 10)	(9, 10)	(9, 10)	(9, 10)
← (8 − 11)													(9, 10)	(9, 10)	(9, 10)	(9, 10)
← (4, 5, 9, 10)																
← (3, 6, 8, 11)											(4)	(4)	(5, 9, 10)	(5, 9, 10)	(4, 5, 9, 10)	(4, 5, 9, 10)
← (3 − 6, 8 − 11)											(4)	(4)	(5, 9, 10)	(5, 9, 10)	(4, 5, 9, 10)	(4, 5, 9, 10)
← (4, 5)																

Table C.1: Transition table of TPA* for the XML document D from Example 2.1