FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Automata Approach to XML Data Indexing: Implementation and Experimental Evaluation
Student:	Lukáš Renc
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: Tree String Path Automaton (TSPA), Tree String Path Subsequences Automaton (TSPSA), and Tree Paths Automaton (TPA).
For these methods, suggest an appropriate implementation.
1) Focus on optimal time and space complexity, easy experimental evaluation and system architecture.
2) Implement the algorithms as a Java library and test its functionality using a suitable data set.
3) Provide experimental evaluation (time and space complexity) of these methods using different input data sets.
For example, use different size of XML files, different average depth of XML trees or different number of leaves.
In the experiments, focus on the index construction phase, size of the index structure and time complexity of query evaluation.

References

Will be provided by the supervisor.

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

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

Prague December 29, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Automata Approach to XML Data Indexing: Implementation and Experimental Evaluation

Lukáš Renc

Department of Theoretical Computer Science
Supervisor: Ing. Eliška Šestáková

May 15, 2018

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. Eliška Šestáková, for sharing her knowledge, words of encouragement, and her guidance through the program.

I would also like to thank my family: my parents and my brother. Last but not least I am extremely thankful to my partner Terezka for unconditionally believing in me.

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 accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 15, 2018 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

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

Abstrakt

Tato práce se zabývá implementací a experimentálním vyhodnocením metod pro indexování XML dokumentů. Konkrétně se jedná o metody Tree String Path Automaton (TSPA), Tree String Path Subsequence Automaton (TSPSA) a Tree Path Automaton (TPA). Tyto metody jsou založeny na teorii konečných automatů a umožňují nalezení odpovědi pro omezenou podmnožinu XPath dotazů (obsahující pouze /, // přechody a jejich kombinaci) v lineárním čase délky dotazu. Jednotlivé metody jsou v této práci implementovány jako Java knihovna. K předzpracování XML dokumentu je použita knihovna SAX. Hlavní část práce se věnuje popisu, implementaci a podmínkám běhu experimentů. V práci jsou prezentovány provedené experimenty. Tyto experimenty zkoumají, jak závisí vlastnosti indexu na velikosti (hloubce, šířce) vstupního XML souboru. Při tvorbě indexu měříme spotřebu RAM a čas. Proto XML dokumenty použity pro experimenty tvoří set s navzájem různými klíčovými parametry (např. průměrná hloubka, maximální hloubka, velikost, počet listů). V závěru práce jsou graficky prezentovány výsledky experimentů. Ve výsledné knihovně je zabudována podpora pro spuštění výše zmíněného experimentálního prostředí.

Klíčová slova XML, indexování dat, automat, konečný automat, XPath, index

Abstract

This thesis deals with implementation and an experimental evaluation of some XML data indexing methods. The methods are as follows:Tree String Path Automaton (TSPA), Tree String Path Subsequence Automaton (TSPSA) and Tree Path Automaton (TPA). All of these methods are based on the theory of finite automata and answer a limited subset of XPath query (limited to /, // transitions and their combination) in linear time to the length of the query. They are implemented as a Java library. SAX library is used to preprocess an XML document. The main part of the thesis is dedicated to a description, an implementation and conditions under which experiments are conducted. In the thesis experiments are run to clarify relations between Size/Depth/Width of an XML document and RAM consumption/Time to build an index. The chosen XML documents, which are presented in this thesis, form a set of mutually different documents in crucial aspects (average depth, maximal depth, size, number of leaves). Results of the conducted experiments are described in the end of the thesis. There is built-in support for experimental environment in the resulting Java library.

Keywords XML, data indexing, automaton, finite state machine, XPath, index

Contents

Introduction **1**
Motivation 2
Goals 2

1 Theoretical Background **3**
1.1 Basic Notions 3
1.2 XML 4
1.3 XPath 8

2 Automata Approach to XML Data Indexing **9**
2.1 Tree String Path Automaton 10
2.2 Tree String Path Subsequences Automaton 13
2.3 Tree Path Automaton 16

3 Research **21**
3.1 Effective XML Preprocessing 21
3.2 Finite Automaton Incremental Construction 22
3.3 Trie Data Strucuture 23
3.4 Finite Automaton Transition Table 23
3.5 Summary 23

4 Implementation **25**
4.1 New Algorithm description 25
4.2 Classes Description 29
4.3 Used Libraries & Data Structures 32
4.4 Advantages & Disadvantages of the Algorithm 33

5 Experimental Evaluation **35**
5.1 Methods Description 35
5.2 DataSets Description 35

5.3 Experiments 37
5.4 Experimental Results Summary 48

6 Library Usage **49**
6.1 Method `getdTSPA()` 49
6.2 Method `getdTSPSA()` 50
6.3 Method `getdTPA()` 50
6.4 Method `getnTPA()` 50
6.5 Method `getResults()` 51
6.6 Method `resolveQuery()` 51

7 Goals Fulfillment **53**

Conclusion **55**

Bibliography **57**

A Acronyms **61**

B Contents of enclosed SD Card **63**

List of Figures

1.1 Sample XML file 5
1.2 XML tree model $T(D)$ from Example 1.1 6

2.1 String path set for the sample XML from Figure 1.1 10
2.2 String path alphabet for the sample XML from Figure 1.1 10
2.3 Individual TSPA for the string path set from Example 2.1 11
2.4 TSPA for the string path set from Example 2.1 12
2.5 Evaluation of the sample query from Figure 1.2.2 13
2.6 Deterministic TSPSA for the XML tree model T from Figure 2.1. . 15
2.7 TSPA for the String path $P =$ `TEAMS`(1) `TEAM`(5) `TOPPPLAYER`(9) `TEAM`(10) `TOPPLAYER`(11) from Example 2.3.1. 16
2.8 TSPSA for the String path $P =$ `TEAMS`(1) `TEAM`(5) `TOPPPLAYER`(9) `TEAM`(10) `TOPPLAYER`(11) from Example 2.3.1. 18
2.9 TPA for the String path $P =$ `TEAMS`(1) `TEAM`(5) `TOPPPLAYER`(9) `TEAM`(10) `TOPPLAYER`(11) from Example 2.3.1 19
2.10 TPA for the string path set from Example 2.1 20

4.1 Sample string path set to demonstrate differences between two backtracking approaches. 27
4.2 Visualization of the Semideterministic backtracking approach. . . . 28
4.3 Visualization of the Nondeterministic backtracking approach. . . 29

5.1 Time during TPA build for the Sample XML dataset 39
5.2 RAM usage during TPA build for the Sample XML dataset 40
5.3 Time of TPA Build for the Generated XML dataset 42
5.4 RAM Usage during TPA build for the Generated XML dataset . . 43
5.5 Time during TPA build for the Real World XML dataset 44
5.6 RAM usage during TPA build for the Real World XML dataset . 45
5.7 Time of evaluation for queries Q_1, Q_2, Q_3 for the D_2 46
5.8 Time of evaluation for queries Q_4, Q_5, Q_6 for the D_5 47

5.9 Time of evaluation for queries Q_7, Q_8, Q_9 for the D_{10} 48

6.1 High level of TSPA structure 50

List of Tables

5.1 Features of the chosen XML documents for experiments 37
5.2 Time of TPA Build for the Sample XML dataset 39
5.3 RAM usage of TPA Build for the Sample XML dataset 40
5.4 Time of TPA Build for the Generated XML dataset 41
5.5 RAM Usage of TPA Build for the Generated XML dataset 43
5.6 Time of TPA Build for the Real World XML dataset 44
5.7 RAM Usage of TPA Build for the Real World XML dataset 45
5.8 Query evalution performance test for NbaSample2.xml 46
5.9 Query evalution performance test for XMark-f0.01.xml 47
5.10 Query evalution performance test for Orders.xml 47

Introduction

Extensible Mark Up Language (XML) [1] is a widespread way to share and store data. It has been a W3C recommendation for 20 years. Because of popularity of XML language [2] efficient quering is needed. This scientific field has been studied for many years. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Unfortunately, XML has a few significant drawbacks for the speed performance (e.g., variable length of an element due to parallel processing is not possible) which slows down evaluation of queries. Therefore time and memory efficient evaluator is needed.

XPath [17], XLink [2] (query languages providing standards for queries on XML documents for more efficient evaluation), etc., do not make use of prior structuring of a given XML file, therefore queries are not resolved fast due to excessive XML tree traversal. This issue can be fixed via indexing. Finite automaton is one of possible solutions for the design of the index [18]. It meets time efficiency, unlike XPath or XLink.

This thesis implements and provides experimental evaluation to three automata-based methods which were presented in [18] namely Tree String Path Automaton (TSPA), Tree String Path Subsequence Automaton (TSPSA), Tree Path Automaton (TPA) [18].

Being able to quickly answer queries on any XML document is fundamental for a smooth run of an application. Suggested solution is very effective for different queries on the same XML file. (Because of the time spent on creating of the evaluator which is later used again without modifications.)

The main goal of this thesis is to implement the design and conduct experiments on a finite automaton based index which is used to quickly resolve queries on any given XML file. The thesis contains analysis of already known solutions. Author focuses not only on suggesting implementation of the index but also on implementing environment for experiments. The thesis presents conducted experiments which show efficiency and speed of the evaluator in tables and graphs. Last but not least the author aims to create extensible interface for further improvements and/or additional functionality.

The thesis continuous in the following structure. Starts with theoretical background (principles, definitions) then move on to chapter about existing approach. After theoretical chapters comes a chapter presenting author's approach. In the second part of the thesis there is an introduction to experimental environment and in the end there are interpreted results of the experiments.

Motivation

Author's main motivation for choosing this topic is making use of the theory of automata in practice. Challenging current solutions and possible everyday usage of the result is another reason author made the decision.

Goals

The main goals of this thesis is to

- study existing automata-based XML data indexing methods [18] namely Tree String Path Automaton, Tree String Path Subsequence Automaton, Tree Path Automaton,
- implement these automata-based XML data indexing methods as a Java library,
- build into the library support for running experiments,
- provide detailed experimental evaluation of the implemented methods.

Chapter 1

Theoretical Background

1.1 Basic Notions

1.1.1 Alphabet

Definition 1 (Alphabet)**.** *An alphabet A is a finite non-empty set whose elements are called symbols.*

1.1.2 String

Definition 2 (String)**.** *A string over a given alphabet A is a finite sequence of symbols of A.*

1.1.3 Subsequence

Definition 3 (Subsequence)**.** *A subsequence of a string $x = x_1x_2...x_n$ is a string y obtained by deleting zero or more symbols from x.*

1.1.4 Prefix

Definition 4 (Prefix)**.** *A prefix of a string $x = x_1x_2...x_n$ is a string $y = x_1x_2...x_m$, where $m \leq n$.*

1.1.5 Nondeterministic Finite State Automaton

Definition 5 (NFSA)**.** *A Nondeterministic Finite State Automaton (NFSA) is a five-tuple $M = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is an alphabet, δ is a state transition function from $Q \times A$ to the power set of Q, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states.*

1.1.6 Deterministic Finite State Automaton

Definition 6 (DFSA)**.** *A finite state automaton is Deterministic (DFSA) if* $\forall a \in A, q \in Q : |\delta(q, a)| \leq 1$.

For a nondeterministic finite state automaton $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$, we can construct an equivalent deterministic finite state automaton $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ using the standard determinization algorithm based on subset construction [19]. Every state $q \in Q_2$ corresponds to some subset of Q_1. We call this subset a d-subset (deterministic subset). The d-subset is a totally ordered set; the ordering is equal to ordering of states of M_1 considered as natural numbers.

1.1.7 Automata Product Construction

Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ and $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ be two finite state automata. We define a finite state automaton $M_\cup$ such that $L(M_\cup) = L(M_1) \cup L(M_2)$. We can build the automaton $M_\cup$ by running the automata M_1 and M_2 in "paralllel" by remembering the states of both automata while reading the input. This is achived by the product construction [19]: $M_\cup = (Q_1 \times Q_2, A, \delta, (q_{01}, q_{02}), (F_1 \times Q_2) \cup (Q_1 \times F_2))$, where $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

1.1.8 Trees

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

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

1.2 XML

XML [1] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The set of marks of an XML document is not fixed and can be defined in various ways for each document. The key constructs of an XML document are tags, elements and attributes. XML document can be seen as a tree of nodes where each of the nodes is created according to an element in the XML document

and edges represent element inclusion [18]. XML indexes which are implemented in this thesis deal with only the structure of the elements. Attributes and texts are ignored and therefore not inserted into leaves. Every node consists of a pair - label + id. The label corresponds to a tag name and the id is an identifier made of an associated preorder number of its element in an XML document. XML alphabet is formed by a set of unique tag names as it is defined below.

Definition 7 (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.*

Consider having this sample XML document which shows a basic info about NBA teams and their own supportive team in G League. • [1]

Example 1.2.1.

```
<TEAMS>
   <TEAM name = "L.A. CLippers">
        <TOPPLAYER>Blake Griffin</TOPPLAYER>
        <COACH>Doc Rivers</COACH>
    </TEAM>
    <TEAM name = "Washington Wizards">
        <TOPPLAYER>Tomas Satoransky</TOPPLAYER>
        <COACH>Scott Brooks</COACH>
        <ARENA>Capital One ARENA</ARENA>
        <GLEAGUE>
            <TEAM name = "Capital City Go-Go">
                <TOPPLAYER>Tobias Love</TOPPLAYER>
                <ARENA>
                 St. Elizabeths East Entertainment and Sports Arena
                </ARENA>
            </TEAM>
        </GLEAGUE>
    </TEAM>
</TEAMS>
```

Figure 1.1: Sample XML file

The sample XML document, from Figure 1.1, can be represented as a tree. See Figure 1.2.

[1] Tomas Satoransky is a Czech player! Go Sato Go !

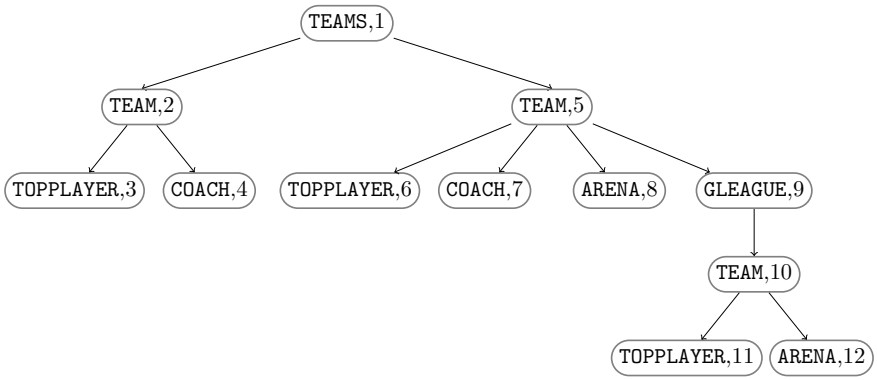


Figure 1.2: XML tree model $T(D)$ from Example 1.1

1.2.1 Supported Set of Queries

All automata presented in this paper support some fragments of linear XPath queries. In particular, the author focused on the two common axes (i.e., child and descendant-or-self) with name tests. Following XPath queries are supported.

$XP^{/,nametest}$ Queries using the child axis (i.e., /) only. By using only */transitions* each element on a path to the result has to be specified

$XP^{//,nametest}$ Queries using the descendant-or-self axis (i.e., //) only. By using only *//transitions* multiple previous elements on a path to the result cannot be specified

$XP^{/,//,nametest}$ Queries using any combination of child (i.e., /) and descendant-or-self (i.e., //) axis.

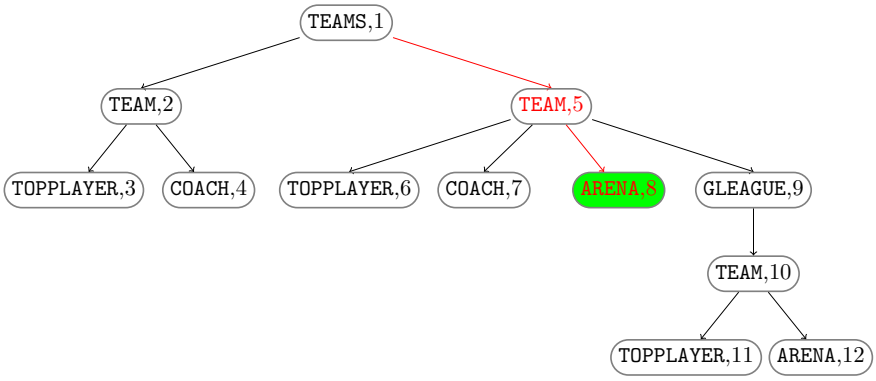
1.2.1.1 Visual Demonstration for $XP^{[/]/,nametest}$

Visual demonstration of above mentioned notation of transition. Sample queries are evaluated for the sample XML document, see 1.1.

Example 1.2.2 (Sample $XP^{/,nametest}$ query)**.**

Input Query: `/TEAMS/TEAM/ARENA`

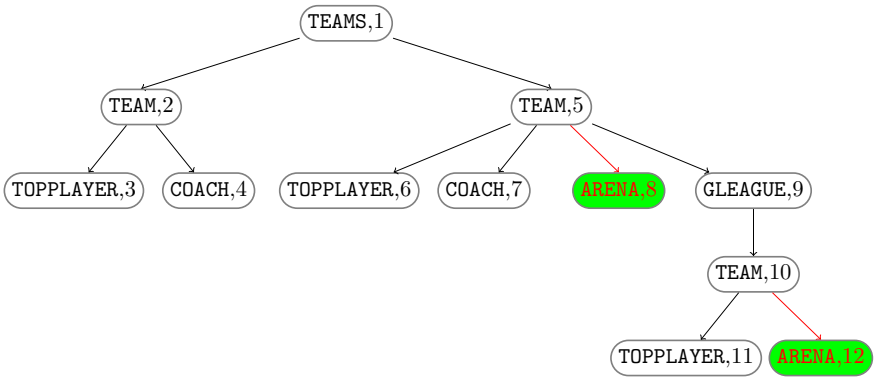
Output Node: `ARENA 8`



Example 1.2.3 (Sample $XP^{//,nametest}$ query)**.**

Input Query: `//ARENA`

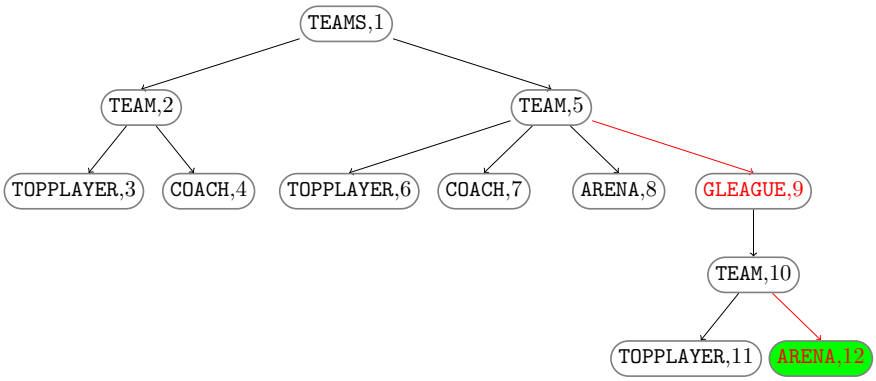
Output Node: `ARENA 8, ARENA 12`



Example 1.2.4 (Sample $XP^{[/]/,nametest}$ query)**.**

Input Query: `//TEAM/GLEAGUE//ARENA`

Output Node: `ARENA 12`



1.3 XPath

XPath [17] (XML Path Language) is one of the XML query languages. Its name comes from the use of the path as a navigation notation on an XML document. It is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.

Chapter 2

Automata Approach to XML Data Indexing

This chapter shows possible automata-based solution to XML data indexing. First we specify 3 definitions which are later used in description of each automaton. Starting with TSPA, the thesis moves onto the TSPSA and the TPA.

The finite automata and methods described in this chapter were presented in [18][20][21] .

Definition 8 (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 9 (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 10 (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 $P(T) = \{P_1, P_2, \dots, P_k\}$.*

Following examples demonstrate principles of above mentioned definitions. String path set and Alphabet for the sample XML document, see 1.1, are presented in the following example.

The String path set for 1.1 XML document is as follows:
Example 2.0.1 (Sample String path set)**.** •

- P_1 = `TEAMS`(1) `TEAM`(2) `TOPPLAYER`(3),
- P_2 = `TEAMS`(1) `TEAM`(2) `COACH`(4),
- P_3 = `TEAMS`(1) `TEAM`(5) `TOPPLAYER`(6),
- P_4 = `TEAMS`(1) `TEAM`(5) `COACH`(7),
- P_5 = `TEAMS`(1) `TEAM`(5) `ARENA`(8),
- P_6 = `TEAMS`(1) `TEAM`(5) `GLEAGUE`(9)`TEAM`(10)`TOPPLAYER`(11),
- P_7 = `TEAMS`(1) `TEAM`(5) `GLEAGUE`(9)`TEAM`(10)`ARENA`(12),

Figure 2.1: String path set for the sample XML from Figure 1.1

The corresponding string path alphabet for 1.1 XML document is as follows:
Example 2.0.2 (Sample String path Alphabet)**.** •

- $A(P_1) = A(P_3) = \{$`TEAMS,TEAM,TOPPLAYER`$\}$,
- $A(P_2) = A(P_4) = \{$`TEAMS,TEAM,COACH`$\}$,
- $A(P_3) = \{$`TEAMS,TEAM,ARENA`$\}$,
- $A(P_4) = \{$`TEAMS,TEAM,GLEAGUE,TOPPLAYER`$\}$,
- $A(P_5) = \{$`TEAMS,TEAM,GLEAGUE,COACH`$\}$.

Figure 2.2: String path alphabet for the sample XML from Figure 1.1

2.1 Tree String Path Automaton

The Tree String Path Automaton (TSPA), for details see [18], is a finite automaton. Its purpose is to speed up the evaluation for all of the XPath queries which use only the child axis (i.e., `/`-axis). Formal definition of such a fragment is represented by the following context-free grammar:

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

For the given XML tree we first obtain all of the string paths which form together a string path set. XPath queries that contain only child axis are basically prefixes of individual string paths. Therefore using a prefix automaton for a set of string paths is possible. To achieve this we construct prefix automata for each of the string path in the string path set and afterwards run all of the automata "in parallel" by remembering the states of all automata while reading input. This is achieved by the product construction and as a result we get the desired tree string path automaton.

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

ht! $Q \leftarrow \{0, id(n_1), id(n_2), \dots, id(n_{|P|})\}$,

ht! $A = \{/a : a \in A(P)\}$,

ht! $\delta(0, /label(n_1)) \leftarrow id(n_1)$,
$\forall i \in \{1, 2, \dots, |P| - 1\} : \delta(id(n_i), /label(n_{i+1})) \leftarrow id(n_{i+1})$,

ht! $F \leftarrow Q \setminus \{0\}$.

[18]
Algorithm 1: Construction of a deterministic prefix automaton for a single string path.

By running Algorithm 1 desired automata are as follow:

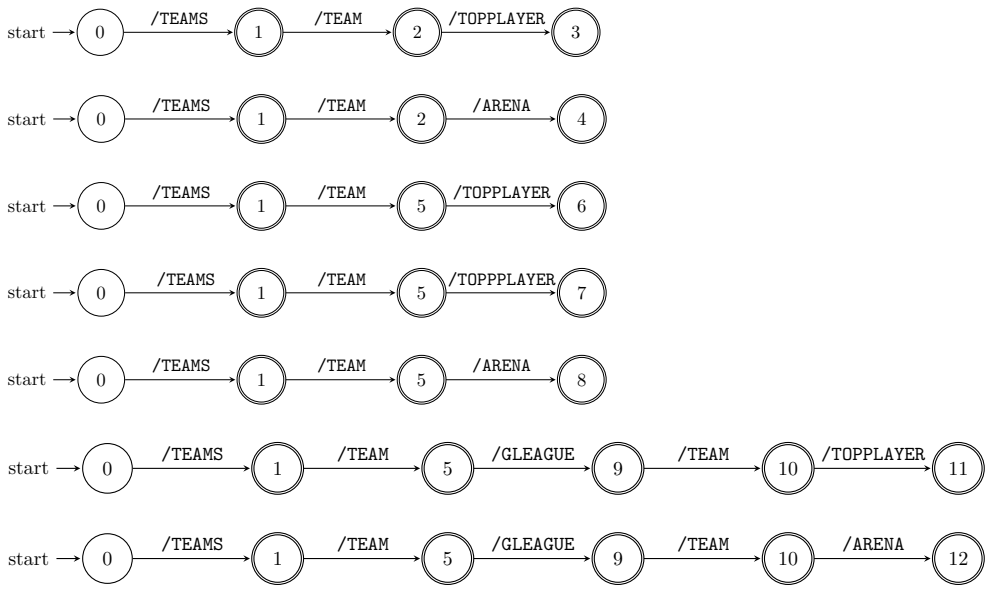


Figure 2.3: Individual TSPA for the string path set from Example 2.1

Example 2.1.1.

After combining each of the individual TSPA Figure 2.3 by running them in parallel, final TSPA Figure 2.1.2 is created.

Example 2.1.2 (Resulting sample TSPA)**.**

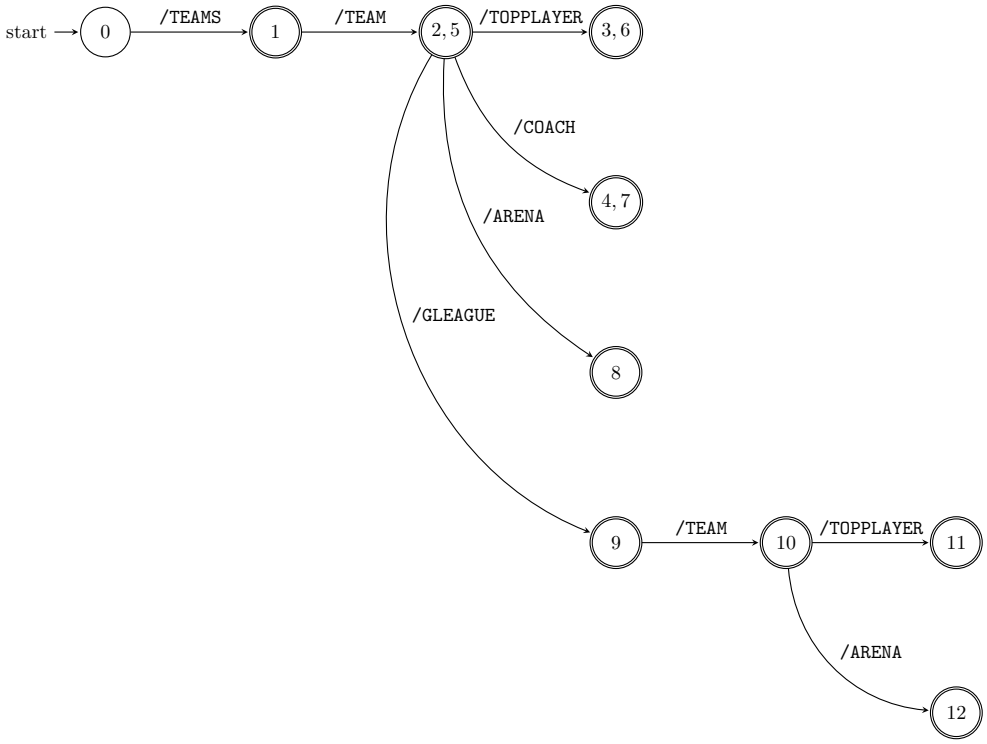


Figure 2.4: TSPA for the string path set from Example 2.1

Example 2.1.3 (TSPA sample query evaluation)**.** Sample evaluation of the sample $XP^{/,nametest}$ query `/TEAMS/TEAM/ARENA` 1.2.2 follows.

2.1.1 Discussion of Time and Space Complexities

TSPA 2.1.2 efficiently supports the evaluation of all $XP^{/,nametest}$ queries of an XML document D. The number of such queries is linear in the number of nodes of the XML tree model T(D). For an input query Q of size m and output length of size n, TSPA obviously performed the searching in time $O(m + n)$ and does not depend on the size of the original document. A result cannot be returned any faster beacause the query of m elements has to be read first $\rightarrow$ $complexityTime + O(m)$. Result of length n returns in time $O(n) \rightarrow complexityTime + O(n)$. Therefore the minimum amount of complexity time for query evaluation is $O(m + n)$. For more details (proof) see [18].

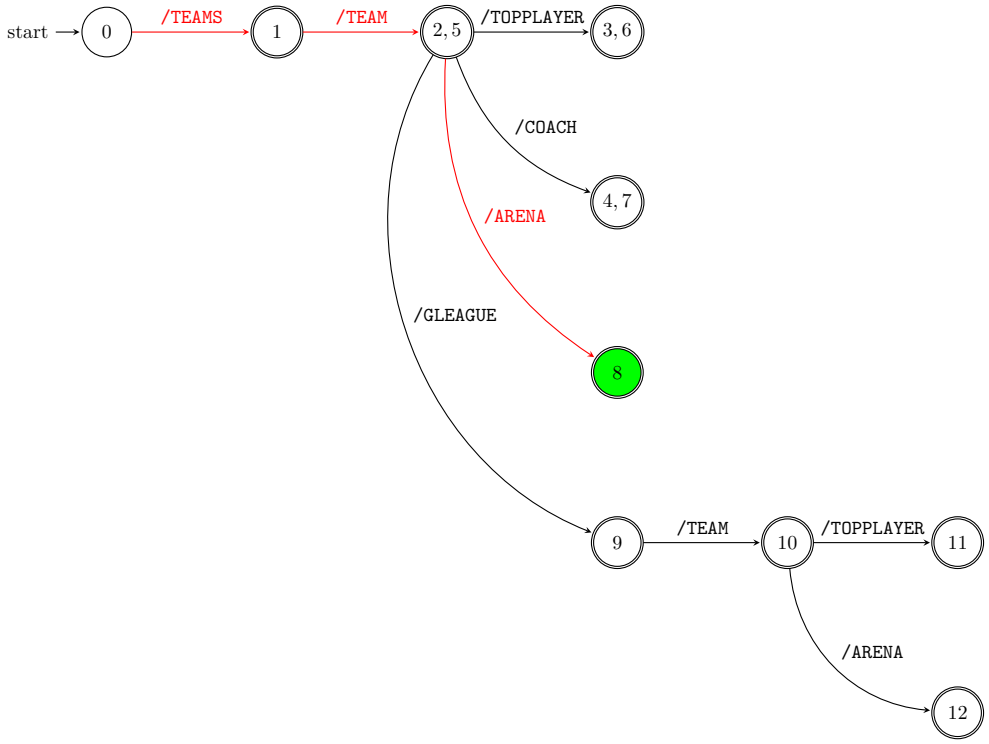


Figure 2.5: Evaluation of the sample query from Figure 1.2.2

2.2 Tree String Path Subsequences Automaton

The Tree String Path Subsequences Automaton (TSPSA) [18] is a finite state automaton that efficiently evaluates all linear XPath queries $XP^{\{//,name-test\}}$ where only the descendant-or-self axis (i.e., `//`-axis) is used. This kind of an XPath fragment, can be represented by the context-free grammar as follows:

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

In the very same fashion as for the TSPA the construction of TSPSA is very systematic. At first, an XML document is processed in order to obtain a string path set. There is a crucial difference between TSPA and TSPSA. For satisfying $XP^{//,nametest}$ queries algorithm focuses on subsequences of a string path rather than prefixes. Because of that we create a subsequence automaton for each of the string paths and then by production merge them all together to get the desired TSPSA.

Paper [18] defines following definitions which are used in the Section 2.1.2. TSPSA construction algorithm follows as well.

Definition 11 (Set of occurrences of an element label in a String path)**.** *Let $P = n_1 n_2 \dots n_{|P|}$ be a String path and e be an element label occurring at several positions in P (i.e., $label(n_i) = e$ for some i). A set of occurrences of the element label e in P is a totally ordered set $O_P(e) = \{o \mid o = id(n_i) \wedge label(n_i) = e,\ i = 1, 2, \dots, |P|\}$. The ordering is equal to ordering of element prefix identifiers as natural numbers.*

Definition 12 (ButFirst)**.** *Let P and $O_P(e) = \{o_1, o_2, \dots, o_{|O_P(e)|}\}$ be a String path and a set of occurrences of an element label e in the String path P, respectively. Then, we define a function $ButFirst(O_P(e)) = \{o_2, \dots, o_{|O_P(e)|}\}$.*

Data: A String path $P = n_1 n_2 \dots n_{|P|}$.
Result: DFSA $M = (Q, A, \delta, q_0, F)$ accepting all (non-empty) $XP^{\{//, name-test\}}$ queries of P

1. $\forall e \in A(P)$ compute $O_P(e)$.

2. Build the "backbone" of the finite state automaton $M = (Q, A, \delta, q_0, F)$:

   a) $Q \leftarrow \{q_0, q_1, \dots, q_{|P|}\}$, $A \leftarrow \{//a : a \in A(P)\}$, $F \leftarrow Q \setminus \{q_0\}$, $q_0 \leftarrow 0$

   b) $\forall i$, where $i \leftarrow 1, 2, \dots, |P|$:
      i. set state $q_i \leftarrow O_P(label(n_i))$,
      ii. add $\delta(q_{i-1}, //label(n_i)) \leftarrow q_i$,
      iii. $O_P(label(n_i)) \leftarrow ButFirst(O_P(label(n_i)))$.

3. Insert "additional transitions" into the automaton M:

   $\forall i \in \{0, 1, \dots, |P| - 1\}$, $\forall a \in A(P)$:
   i. add $\delta(q_i, //a) \leftarrow q_s$, if there exists such $s > i$ where $\delta(q_{s-1}, //a) = q_s \wedge \neg\exists r < s : \delta(q_{r-1}, //a) = q_r$
   ii. $\delta(q_i, //a) \leftarrow \emptyset$ otherwise.

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

We can run all subsequence automata "in parallel" using the product construction (similarly to TSPA) and obtain the index for all $XP^{\{//, name-test\}}$

queries of the particular XML document. Figure 2.6 illustrates TSPSA constructed by Algorithm 2 for the XML document D and its XML tree model $T(D)$ from Figure 1.2.

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

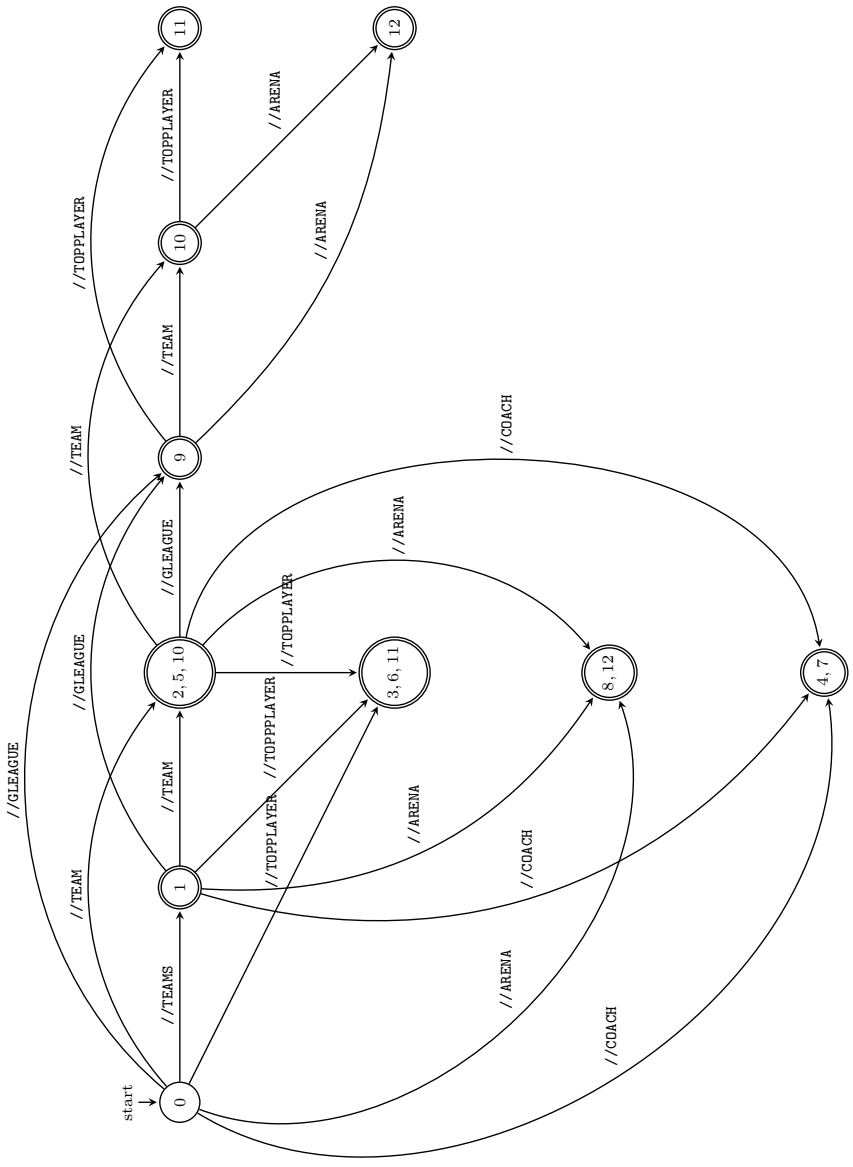


Figure 2.6: Deterministic TSPSA for the XML tree model T from Figure 2.1.

2.3 Tree Path Automaton

Tree Paths Automaton (TPA) [18] is an automaton designed to answer significant amount of possible XPath queries.Those which only contains any combination of child (i.e., `/`) and descendant-or-self (i.e., `//`) axes, denoted as $XP^{\{/,//,name-test\}}$. All the supported XPath queries can be represented by the context-free grammar as follows.

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

One can see TPA as a combination of previously introduced prefix and subsequence automaton. Both $XP^{\{/,name-test\}}$ and $XP^{\{//,name-test\}}$ queries are subsets of $XP^{\{/,//,name-test\}}$ queries, therefore they are supported by TPA as well. The paper [18] suggests algorithm that combines prefix and subsequence automata together for each String path separately and later on creates final TPA by combining TPAs of individual String paths. The algorithm follows.

Example 2.3.1. Let D and $T(D)$ be an XML document and its corresponding XML tree model from Example 1.1 and Figure 1.2, respectively. Given $P =$ `TEAMS`(1) `TEAM`(2) `TOPPPLAYER`(6) `TEAM`(7) `TOPPLAYER`(8) as the input String path, Algorithm 3 follows these steps:

1. creates TSPA for P as shown in Figure 2.7,
2. creates TSPSA for P as shown in Figure 2.8,
3. combines TSPA and TSPSA. See the resulting TPA for P in Figure 2.9.

Figure 2.7: TSPA for the String path $P =$ `TEAMS`(1) `TEAM`(5) `TOPPPLAYER`(9) `TEAM`(10) `TOPPLAYER`(11) from Example 2.3.1.

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

1. Construct a deterministic finite state automaton $M_1 = (Q_1, A_1, \delta_1, 0, F_1)$ accepting all $XP^{\{/,name-test\}}$ queries of P using Algorithm 1.

2. Construct a deterministic finite state automaton $M_2 = (Q_2, A_2, \delta_2, 0, F_2)$ accepting all $XP^{\{//,name-test\}}$ queries of P using Algorithm 2.

3. Construct a deterministic finite state automaton $M = (Q, A_1 \cup A_2, \delta, 0, Q \setminus \{0\})$ accepting all $XP^{\{/,//,name-test\}}$ queries of P as follows:

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

Algorithm 3: Construction of TPA for a single string path

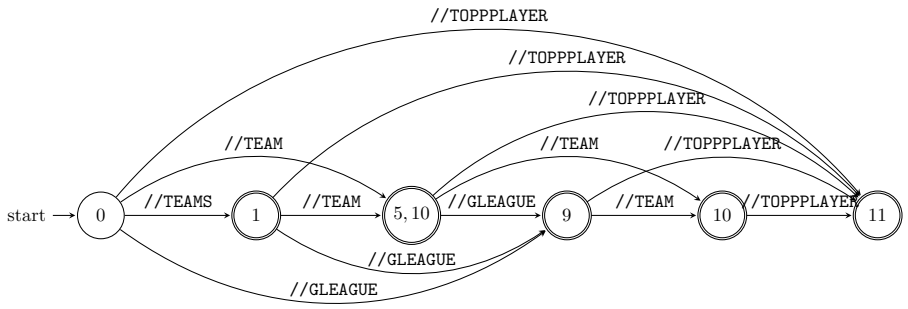


Figure 2.8: TSPSA for the String path $P =$ `TEAMS`(1) `TEAM`(5) `TOPPPLAYER`(9) `TEAM`(10) `TOPPLAYER`(11) from Example 2.3.1.

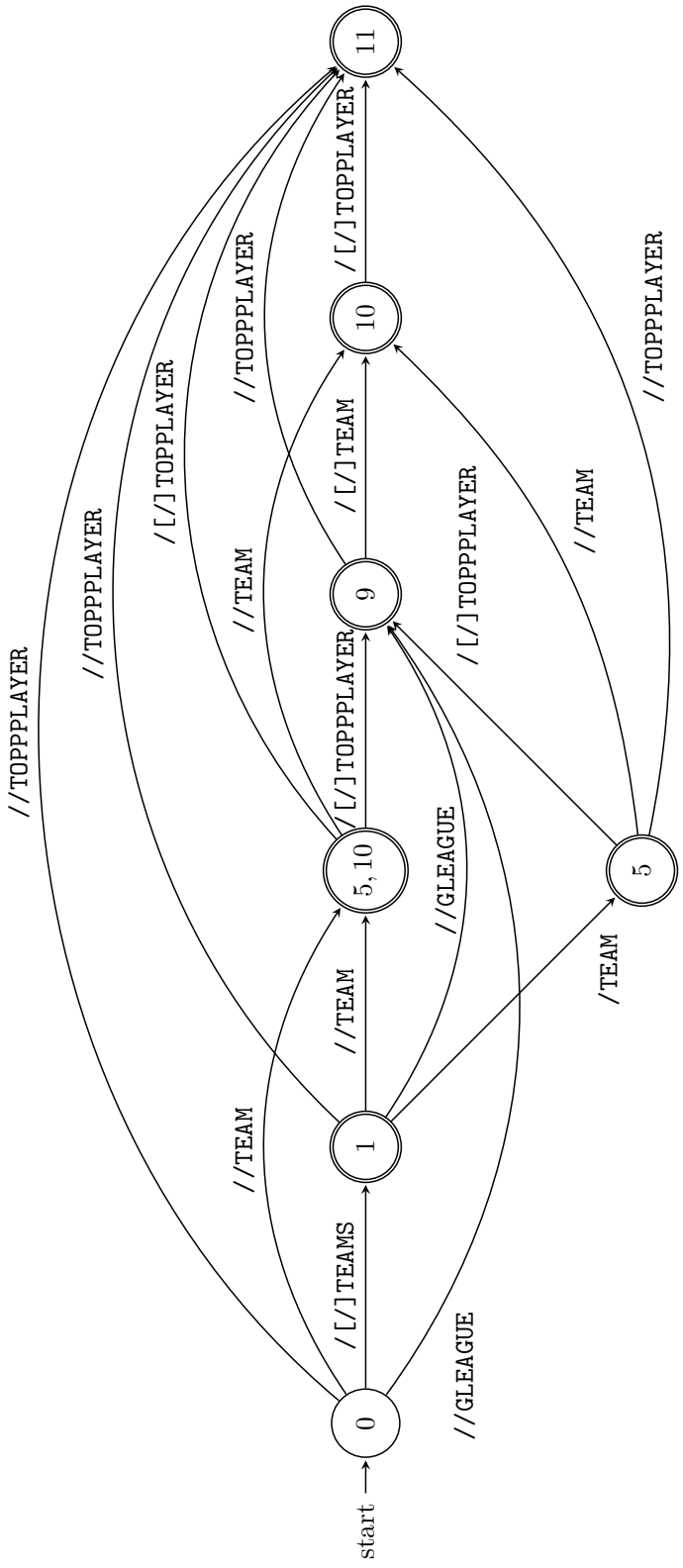


Figure 2.9: TPA for the String path $P =$ TEAMS(1) TEAM(5) TOPPPLAYER(9) TEAM(10) TOPPLAYER(11) from Example 2.3.1

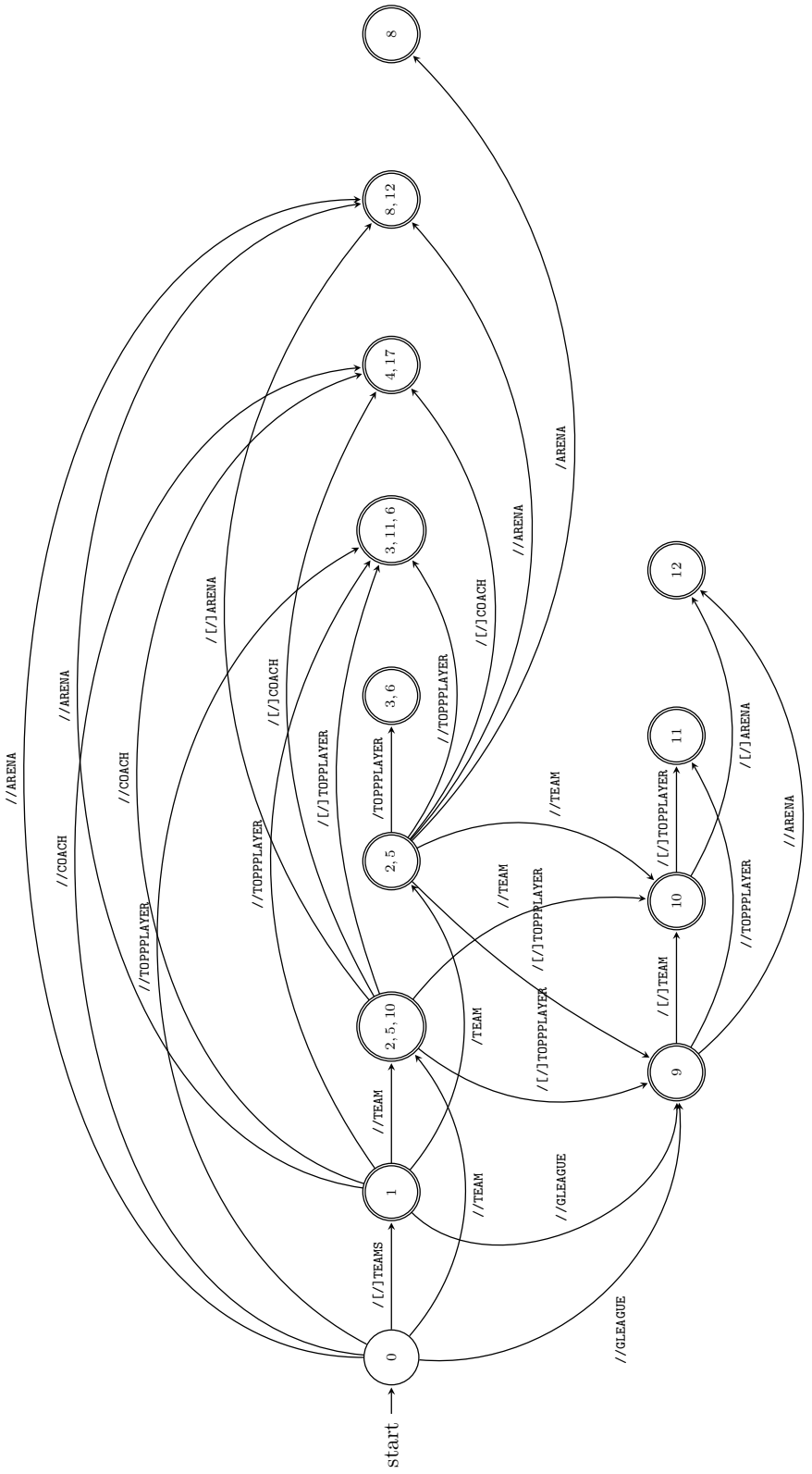


Figure 2.10: TPA for the string path set from Example 2.1

Chapter 3

Research

The author did a research for good practices and effective implementation of a finite automaton before suggesting any improvements. This chapter consists of gathered ideas from multiple papers. The gained knowledge is later used in author's implementation of TPA.

3.1 Effective XML Preprocessing

In Java there are two main methods how to parse an XML document. These are SAX [22] library and JDOM [23] library. The main goal during preprocessing phase is to lower RAM consumption as much as possible and speed up parsing.

Build phase of automaton doesn't need any special order of elements. Therefore only a method to access the next element is needed. There is no need to use methods for getting previous element and/or any jumps in tree traversal.

In addition, it is desired to avoid memorizing the whole structure of the parsed XML. RAM consumption is lowered by avoiding this, because there is no copy of the XML document stored in RAM.

3.1.1 SAX lib vs. JDOM lib Performance Comparison

The followed statements are presented according to [23][24]. Both advantages and disadvantages are aimed fulfill the goal of building an automata.

SAX lib

Advantages

Faster runtime Runtime is faster than JDOM parsing.

Only the current element is stored in RAM The method `nextElement()` returns only the next element which saves memory.

Suitable for parsing large XML documents SAX was designed to parse large XML documents.

Disadvantages

API is not the super friendliest Other parsers feature friendlier API.

JDOM lib

Advantages

Friendly API JDOM parser features friendly API.

Disadvantages

Slow runtime Runtime is slower as JDOM creates a new full copy, modeled as a tree, of the given XML in memory.

High RAM consumption JDOM creates a new full copy, modeled as a tree, of the given XML in memory.

Based on these facts the author made a decision to favor the SAX for building an automaton.

3.2 Finite Automaton Incremental Construction

According to [25] and [26], the incremental construction influence final automaton in positive ways. Main advantages are as follow.

Extends advantages of SAX parsing Because of a step by step building concept, only the current and the next element are important. An output of the SAX parser perfectly fits into it.

Lower RAM usage Resulting automaton instance is known from the very beginning and is only expanded by new states. There are no other redundant instances of work automata left.

No redundant states If implemented properly, there are no redundant states. Because of it, the build phase does not rely on behavior of the Java Garbage Collector.

Transparency It is easier to debug.

3.3 Trie Data Strucuture

Trie is a data structure usually used to store a set where the keys are mostly strings. See [27][28][29]. Typical usage is for dictionaries and word completion apps.

Trie data structure features following advantages.

No hash function → There are no collisions. Therefore no collision management is needed.

Look up in $O(|query|)$ Fast look up even in for the worst case scenario.

Insert in $O(|word|)$ Fast insert even in for the worst case scenario.

Transparent Data structure Easy to maintain.

Problem of indexing an XML document is similar to dictionaries. Author's goal is to quickly find an element and return its info back. A dictionary works in the same way - finds a given word and returns its translation (info) back. Therefore the author can benefit from making use of well studied area of Trie data structure.

3.4 Finite Automaton Transition Table

Papers [30] [31] [32] deal with transition table as a data structure to store transitions for automata purposes.

Transition tables aim to store data. Each of its element can be retrieved by specifying its column and row. Transition tables for automata purposes are designed in a way that each column represents a set of transitions for individual phrases. Each row represents a set of transitions for individual states.

First one of the main advantage is that no transitions are duplicated. Therefore no RAM is wasted for redundant transitions. Secondly, the method `getTransition()` works in $O(log(|TransitionSet|))$. If the exact coordinates are specified (there is no need for look up of coordinates) then `getTransition()` operates in $O(1)$.

The disadvantage is the need to provide a TableManager which takes care of maintaining the structure and retrieval of elements in a transition table.

3.5 Summary

The author makes use of the newly gained knowledge in the following implementation of TPA. Because of advantages of Incremental construction and Trie data structure, the decision was made to redesign the original algorithm from the paper [21]. The new algorithm benefits from both the incremental

production and the similarity to Trie data structure. In addition, switch to the SAX library was made (to lower RAM requirements), despite the fact of the less user-friendly API. Furthermore, advantages of Transition table concept are used as well. In the implementation automaton instance contains a Map of all states. Each state contains a Map of its transitions. As already stated above, there are no duplicates states. Therefore, there are no duplicates transitions as well.

Chapter 4

Implementation

The whole implementation is written in Java [33]. IntelliJ IDEA used as an IDE. Author's Java library is split into 2 packages - Automaton and Experiments. Created automata are serialized (Automaton class implements Serializable interface) to save time for repeated usage. The library is controlled via the command line. The author made a decision to optimize the previous attempts by thinking through the main algorithm. A new algorithm has been made up.

4.1 New Algorithm description

A new algorithm has been made up to speed up the process of a creation and lower the amount of used RAM during it. The creation of TPA 2.10 splits into the three phases.

1. Preprocessing of XML document
2. Build phase
3. Determinization

4.1.1 Preprocessing

Given XML document is firstly preprocessed with SAX library. The main advantage of the SAX library is smaller RAM consumption as it only returns element by element. (Unlike JDOM library which returns the whole XML document as a tree). At the end of this phase String paths 2.9 in the form of String path set are passed to the next phase.

4.1.2 Build phase

This is the crucial part of the algorithm. An input for this phase is a String path set. An output is a nondeterministic finite automaton reflecting the structure of the given XML. The algorithm extends concept similar to TRIE [27]. On top of the edited TRIE concept, there is a backtracking for each of the added XMLTags. The backtracking is responsible for assigning correctly // transitions. Pseudo code follows:

Data: A String path set S .
Result: NFSA $M = (Q, A, \delta, q_0, F)$ accepting all $XP^{\{//name-test,/name-test\}}$ queries.
foreach *Path P of String Path Set S* **do**
  foreach *XMLTag tag of Path P* **do**
    Transition t = get */transiton* according to *tagName* from *currentState*;
    `/* Retrieve a` */transition* `according to state and phrase */`
    if *t != null* **then**
      `/* CurrentState already contains a transition according to` *tagName*`. */`
      if *(!(State pointed by t contains tagPid))* **then**
        Add *tagPid* to pointed state by t;
        Backtracking(tag,currentState,automaton);
        `/* If pointed state does not contain tag pid add tag pid and backtrack[??]. */`
      else
        `/* CurrentState does not contain transition with phrase. */`
        Create a new state *newState* containing *tag*;
        Set a path to *newState*;
        Set a parent of *newState* to *currentState*;
        Create both [/]/transitions from currentState to *newState* on phrase == *tagName*;
        Update transition t to newly created;
        Backtracking(tag,currentState,automaton);
        `/* Create a new state, transition to it and backtrack[??]. */`
      end
      Update *currentState* according to transition t;
      `/* Jump forward with` *currentState*`. */`
  end
end

Algorithm 4: Author's TPA build-phase design

4.1.2.1 Bactracking

The backtracking, already mentioned above, assigns correctly all // transitions.

There are two approaches.

Semideterministic Some states are merged and/or cloned.

Nondeterministic No states are merged, nor cloned.

Each approach has its advantage. The build phase with the Semideterministic backtracking is slower, however a determinization is faster. Vica-versa for the Nondeterministic approach. The thesis shows experimental comparison for both of them.

Differences between both approaches are visualized in the Figures 4.2 4.3. (Please bear in mind that both Figures 4.2 and 4.3 are **not** the final results of the build phase for the string path set 4.1 .)

- P_1 = `LOCATION`(1) `EAST`(2)`CITY`(3)
- P_2 = `LOCATION`(1) `WEST`(4) `CITY`(5)

Figure 4.1: Sample string path set to demonstrate differences between two backtracking approaches.

4.1.2.2 Semideterministic backtracking

The Semideterministic approach implements a feature which merges two states in a new one. This is done by observing whether the parent state `(P)` of the current state `(C)` has a // transition `(T)` to any other state `(S)`. if so, the current state `(C)` and the state `S` are merged together into a state `(C,S)`. The transition `(T)` is modified to connect the parent `(P)` with the new merged state `(C,S)`.

See Figure 4.2 for visualization. Pseudocode 5 follows:

Data: XML tag, currentState, automaton
Result: Changes in the structure of the given automaton
State *propagated* = currentState;
State *parent* = currentState;
//transition t2;
while *parent != null* **do**
  t2 = *currentState* get *//transiiton* on *tagName*;
  if *t2 != null* **then**
    if *! (Pointed state by t2 contain tagPid)* **then**
      if *!(propagated state contains all Xml tags from t2.getPointedState)* **then**
        new State *clone*;
        clone = merge(*propagated* and *t2*getPointedState);
        `/* merge all // and` */transitions `and XmlTags */`
        add *clone* to automata states;
      set *t2* to point to *clone*;
      foreach *Clone = c created by merge of parent* **do**
        add */transition* from *c* to *propagated* on *tagName*;
        add *//transition* from *c* to *propagated* on *tagName*;
      end
  else
    add new *//transition* from *parent* to *propagated* on *tagName*;
  end
  parent = *parent*.getParent();
end

Algorithm 5: Semideterministic backtracking approach

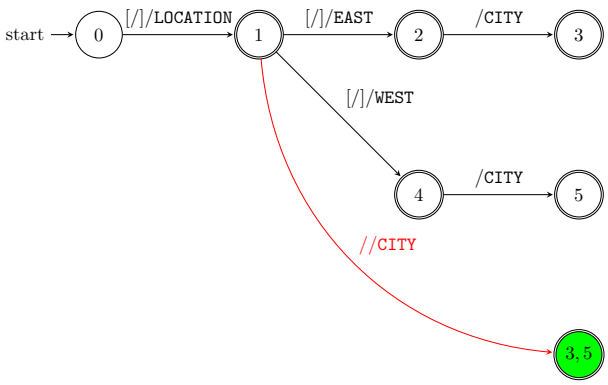


Figure 4.2: Visualization of the Semideterministic backtracking approach.

4.1.2.3 Nondeterministic backtracking

The Nondeterministic backtracking approach is more straightforward and easier to implement. There are neither merges, nor cloning involved. Every new state is connected with the previous ones with // transition. This approach saves time by avoiding computing possible merged states and/or clones.

See Figure 4.3 for visualization. Pseudocode 6 follows:

Data: XML tag, currentState, automaton
Result: Changes in the structure of the given automaton
State *parent* = parent of currentState;
while *parent != null* **do**
    Add new //transition from *parent* to *currentState* on *tagName*;
    parent = parent.getParent();
end

Algorithm 6: Nondeterministic backtracking approach

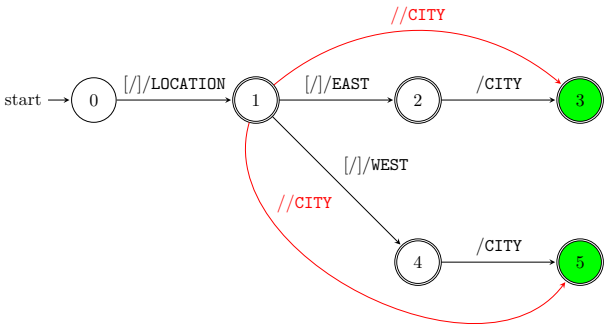


Figure 4.3: Visualization of the Nondeterministic backtracking approach.

4.1.3 Determinization

Well studied algorithm that converts NFSA (Nondeterministic finite automaton) to DFSA (Deterministic finite automaton) which accepts the same set of queries. To study the algorithm see [34] and [35] .

4.2 Classes Description

Author's Java library is split into 2 packages - Automaton and Experiments. Classes in both packages aim to provide an easy to use, yet effective interface. Created automata are serialized (via Serializable interface) to save time for repeated usage.

4.2.1 Automaton package

Aims to provide simple to use and effective methods to create, manipulate and control finite automata which serves as an index for a given XML document. The code itself fulfills object oriented principles. Additional automata are possible to be added to extended the family of currently implemented ones. This package consists of the following classes:

4.2.1.1 `XMLTag` class

The class that stores data that represent original XML tag. In the current design class stores name of the XML tag and assigned unique ID. Class overrides Clone method for String Path 4.2.1.4 creation purpose. The author suggests storing XML elements data here as a further possible extension to design. This class is used under the hood in all of the automata.

4.2.1.2 `State` class

The class that represents states according to the theoretical model of finite automata. Each instance contains Hashset of all its Transitions4.2.1.3 and all XML tags 4.2.1.1. In addition, there is a back reference to a parent (for TSPSA the parent is set as a State which would be the parent in TSPA). This class is used under the hood in all of the automata.

4.2.1.3 `Transition class`

The class that represents transitions according to the theoretical model of finite automata. Each instance contains a phrase to jump onto the next state and exactly two States 4.2.1.2 - state From and state To (Excluding starting and ending states of an automaton. Those contain only To, respectively From state). This class is used under the hood in all of the automata.

4.2.1.4 `StringPath class`

The class that implements String Path 8 idea. Consists of a LinkedList of XMLTags. It is a base part for StringPathOrderedSet class.

4.2.1.5 `StringPathOrderedSet class`

Class that holds all of the StringPath instances. Its instance is created while preprocessing an XML document. It is implemented as an ArrayList of StringPaths.

4.2.1.6 `Automaton class`

An abstract class which is extended by TSPA class 4.2.1.8, TSPSA class 4.2.1.9 and TPA class 4.2.1.10. The fundamental class for this thesis which puts

Transitions 4.2.1.3 and States 4.2.1.2 together. It is an implementation of the theoretical model of an automaton. The only class property is an instance of State 4.2.1.2 firstState, which defines which state is the starting point for automaton. Besides handy methods for manipulation with the automaton, there is an abstract method resolveQuery which is overridden by TSPA 4.2.1.8, TSPSA 4.2.1.9 and TPA 4.2.1.10.

4.2.1.7 `AutomatonFactory class`

The class fulfilling factory design pattern. Static methods buildTSPA(), buildTSPSA() return ready-to-use tree string path automaton 2.1, respectively tree string path subsequence automaton 2.2. Method getStringpathSet() is responsible for pre processing an XML document to StringPathSet 4.2.1.5 instance which is passed to build automaton methods.

4.2.1.8 `dTSPA class`

The class that represents TSPA. It stores only the unnecessary values to provide full functionality of an automaton. Use the method resolveQuery() to correctly parse and answer a specific query in $O(|query| + |returnedelements)$ time complexity.

4.2.1.9 `dTSPSA class`

The class that represents TSPSA. It stores only the unnecessary values to provide full functionality of an automaton. Use the method resolveQuery() to correctly parse and answer a specific query in $O(|query| + |returnedelements)$ time complexity.

4.2.1.10 `dTPA class`

The class that represents TPA. It stores only the unnecessary values to provide full functionality of an automaton. Use the method resolveQuery() to correctly parse and answer a specific query in $O(|query| + |returnedelements)$ time complexity.

4.2.2 Experiment package

The purpose of this package is to easily conduct experiments on the given XML documents. An user benefits from measured values during runtime which are then presented in a table in command line. Package Experiments contains following classes.

4.2.2.1 `KnowYourXml class`

The class to get info about a given XML document. Concept of Lazy initialization is used. All values available via getters. XML document is checked for: Maximal depth, Average depth, Element count, Leaves count, Size of the document, Memory consumption for TPA, Time to build TPA and Number of states.

4.2.2.2 `ExperimentRunner class`

The class that runs experiments on a directory consisting of XML documents. KnowYourXml class 4.2.2.1 methods are used for calculating results for each of the documents. Results are returned in the form of an instance of the ExperimentResult class 4.2.2.3.

4.2.2.3 `ExperimentResult class`

The class to store and share results calculated out of methods from ExperimentRunner class 4.2.2.2. Overriden method toString() returns data in the format of an Ascii table.

4.3 Used Libraries & Data Structures

Multiple advanced data structures and libraries are used among the author's library.

Data Structures

HashMap The data structure is used in class State4.2.1.2 to store XMLTags 4.2.1.1.

MultiValuedHashMap The data structure is used in class State 4.2.1.2 to store Transitions 4.2.1.3.

Libraries

SAX The library is used to access individual elements of a raw XML document. The decision to use this library was made to lower RAM consumption.

javax.ws.rs.core The library is used to provide MultiValuedHashMaps.

4.4 Advantages & Disadvantages of the Algorithm

4.4.1 Advantages

No redundant states during the build phase Every created state is used in the resulting automaton. This feature enables algorithm to be independent of Java Garbage Collector behavior.

Possible immediate processing of a StringPath StringPaths, which represent inner structure of elements, don't have to be stored.

Processing speed Algorithm runs faster because of usage of advanced data structures

Ram consumption Algorithm uses less RAM because of shallow copies of data between two different State 4.2.1.2 instances.

SAX preprocessing By using the SAX lib for preprocessing algorithm runs faster and consumes less RAM

Advantages of TRIE The algorithm makes use of TRIE - well studied data structure concept

Incremental Construction As the result is incrementally constructed there are no working copies which increases RAM consumption

Semideterministic approach speeds up determination up to 60x Merging states during backtracking in Build phase 4.1.2 is an effective way how to save significant amount of time.

4.4.2 Disadvantages

Sequential processing of StringPath. Algorithm cannot be easily run in parallel. Therefore effective threads implementation is not possible.

Chapter 5

Experimental Evaluation

For the conducted experiments the author uses Intel i5 CPU at 1.30GHz with TurboBoost 2.0 up to 2.6GHz, 4GB of 1600MHz LPDDR3 RAM, running Mac OS X Sierra. All the following data were gathered and computed by running every experiment at least 20 times to minimize measurement error.

5.1 Methods Description

Performance of the new algorithm 4 (for the both backtracking approaches 6 5) is tested. Results for the previous implementation of TPA 2.10 and Saxon lib are evaluated as well for comparison purposes.

List of the methods follows:

M_1 = **tpalib** The previous implementation. For more information see: [18].

M_2 = **New Algorithm - Semideterministic backtracking** The new Algorithm proposed by this thesis with the semideterministic backtracking approach. For more information see: 4.2.

M_3 = **New Algorithm - Nondeterministic backtracking** The new Algorithm proposed by this thesis with the nondeterministic backtracking approach. For more information see: 4.3.

M_4 = **Saxon** The java library for evaluating XPath [36] queries. For more information see: [22]

5.2 DataSets Description

Following datasets are used as input data for each of the experiments. They are split into 3 categories. Each category has its own typical features among its XML documents.

5.2.1 Sample XML Documents

Very small XML documents are used through this thesis for example purposes. They feature one or two tricky sections to test a correct output of the methods. GoT.xml (For more information see: [18]) was put into the dataset to reevauluate experiments under the same performance conditions as other XML documents are evaluated.

D_1 = **NBASample.xml** The document shown in the beginning 1.1 of the thesis. It describes my favorite NBA teams. Easy to process.

D_2 = **NBASample2.xml** Slightly upgraded original NBASample.xml document. It features a section which causes that output of the Build phase 4.1.2 is always nondeterministic.

D_3 = **GoT.xml** Sample XML document from the paper [18].

5.2.2 Generated XML documents

These datasets were generated by Xmlgen [37] using different scaling factors. XMark xmlgen scaling factors of a document are a float value where 0 produces the "minimal document". They have in common big average depth and big diversity of XML elements resulting in big automata.

D_4 = **XMark-f0.005.xml** Generated XML document, scaling factor = 0.005

D_5 = **XMark-f0.01.xml** Generated XML document, scaling factor = 0.01

D_6 = **XMark-f0.1.xml** Generated XML document, scaling factor = 0.1

D_7 =**XMark-f1.xml** Generated XML document, scaling factor = 1

5.2.3 Real World XML Document Examples

Taken from Washington.edu XML Data Repositary, these datasets feautures real data from different situations. Usually they are more shallow and contain repetitive constructions which result in a smaller resulting automaton

D_8 = **Auction.xml** Auction data converted to XML from EBay web sources.

D_9 = **Uwm.xml** Course data derived from an university website.

D_{10} = **Orders.xml** Order data from a supplier.

5.3 Experiments

The conducted experiments are split into two categories by the focused subject. In these categories both RAM usage and time consumption are measured and visualized.

1. TPA build
    a) Time consumption
    b) RAM usage
2. Query Evalution
    a) Time consumption
    b) RAM usage

5.3.1 Differences in Datasets

As shown in the tab 5.1 datasets differ not only in size but also in depth and number of leaves. These are the main three aspects which influence performance on building the automaton and the resulting size of automaton. The most significant difference in duration of the build is for the D_6 and the D_9 XML documents. Although the D_6 is almost two times smaller than the D_9, build for the D_6 takes 8.49 seconds whereas for the D_9 only 0.36 second.

	Size (MB)	Max Depth	Average Depth	Number of Elements	Number of Leaves	Build TPA RAM (MB)	Build TPA Time (s)	Index/ XML Size ratio	Number of States
D_1	0.005	4	2.08	12	7	6	12	5.74	13
D_2	0.007	4	2.18	16	8	6	12	4.09	13
D_3	0.005	4	2.08	12	7	6	12	5.74	13
D_4	0.11	11	4.71	1729	1204	14	0.92	1.11	576
D_5	0.57	11	4.50	8518	6211	16	1.63	0.97	897
D_6	1.16	11	4.51	17132	12504	103	8.49	0.87	1020
D_7	115.7	11	4.54	1666315	1211774	741	573	0.85	1296
D_8	0.23	4	2.76	311	250	10	5.62	0.01	33
D_9	2.28	4	2.9	66729	50885	27	0.36	0.73	22
D_{10}	5.25	3	1.9	149989	134990	57	19.36	1.2	50

Table 5.1: Features of the chosen XML documents for experiments

5.3.2 TPA build

Experiments start at the moment of preprocessing data and end when a finite automaton is determinized. Following tables, for each of datasets separate, show measured data during building TPA with a different method. In the following subsection there is a new abbreviation DNF. It stands for Did Not Finished. The experiments were conducted with parameter `--Xmx3500m` which extends maximum available RAM to 3500MB. This is the most author's computer can handle. Therefore extensive usage of RAM may result in Java Out of memory extension. If this situation repetitively happens during an experiment then the abbreviation DNF is used.

5.3.2.1 Sample XML documents Time

The table presents values from the conducted experiments. Values in rows M_1 and M_2 represent duration in miliseconds for each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

		D_1			D_2			D_3	
	pre	build	det	pre	build	det	pre	build	det
M_1	0.24	0.11	0.01	0.24	0.12	0.02	0.24	0.11	0.01
M_2	0.24	0.02	0.11	0.24	0.01	0.10	0.24	0.01	0.10
M_3	0.94	•	0.25	0.89	•	0.22	0.96	•	0.24
M_4	•	•	0.29	•	•	0.33	•	•	0.31

Table 5.2: Time of TPA Build for the Sample XML dataset

Following graphs visualize measured values.

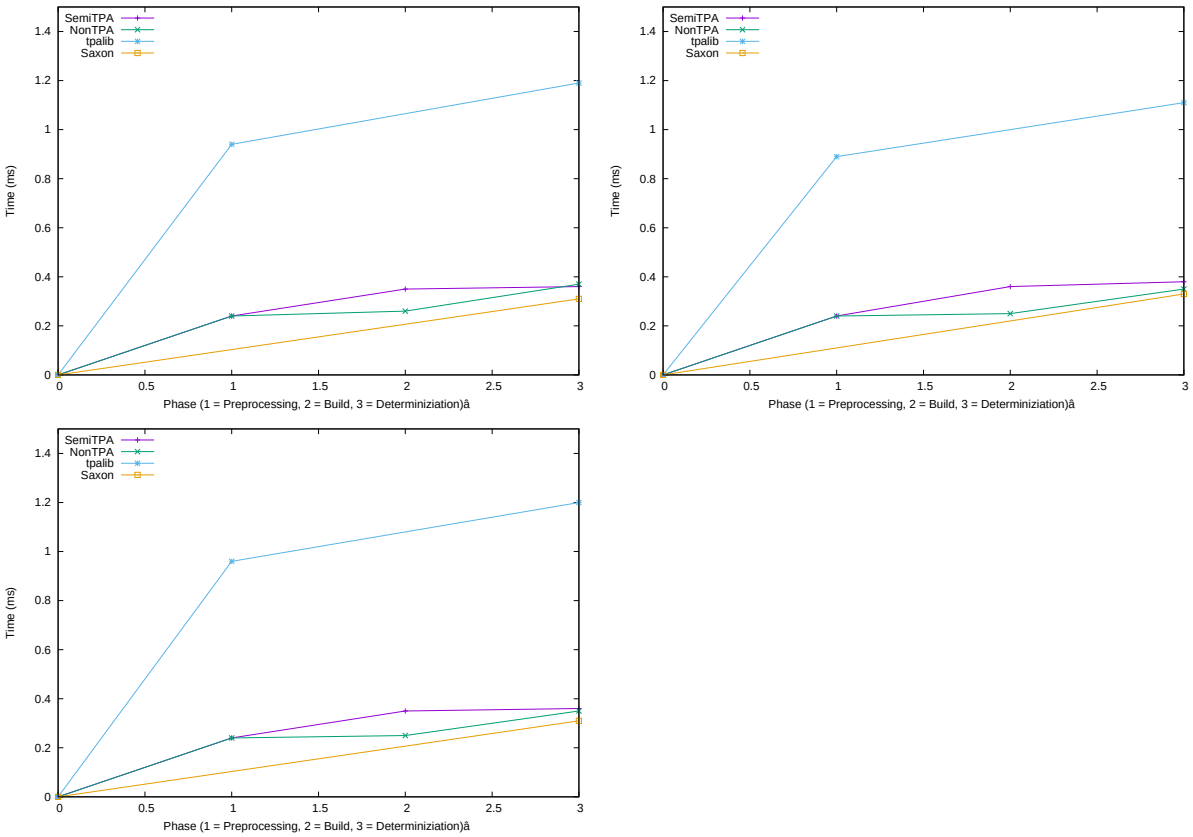


Figure 5.1: Time during TPA build for the Sample XML dataset

5.3.2.2 Sample XML documents RAM Usage

The table presents values from the conducted experiments. Values in rows M_1 and M_2 represent RAM usage (MB) after each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

		D_1			D_2			D_3	
	pre	build	det	pre	build	det	pre	build	det
M_1	10.1	12.1	12.4	8.4	12.1	12.4	10.1	12.1	12.3
M_2	10.1	11.9	12.5	8.4	12.0	12.5	10.1	12.1	12.4
M_3	2.1	•	3.0	2.2	•	3.1	2.1	•	3.1
M_4	•	•	16	•	•	16	•	•	17

Table 5.3: RAM usage of TPA Build for the Sample XML dataset

Following graphs visualize measured values.

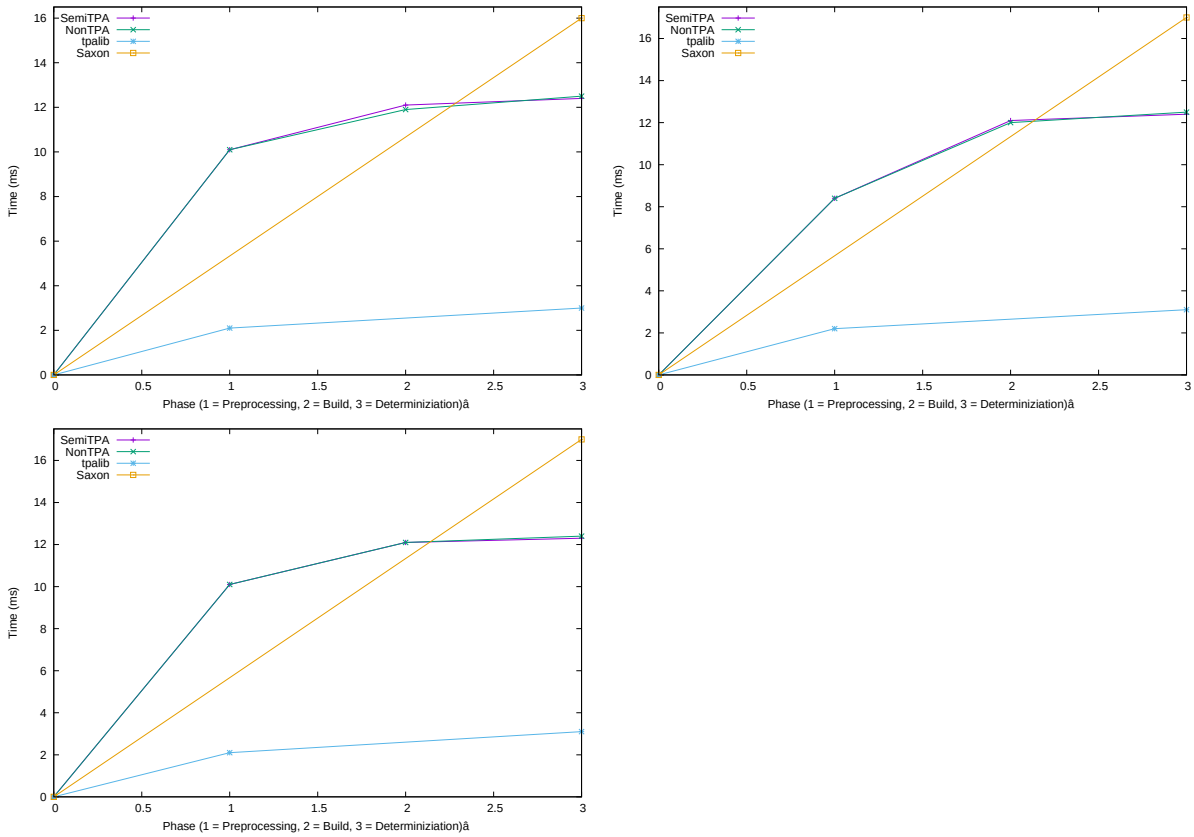


Figure 5.2: RAM usage during TPA build for the Sample XML dataset

5.3.2.3 Generated XML documents Time

The table presents values from the conducted experiments. Values in rows M_1 and M_2 represent duration in miliseconds for each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

		D_4			D_5			D_6			D_7	
	pre	build	det	pre	build	det	pre	build	det	pre	build	det
M_1	0.36	0.44	0.12	0.41	0.91	0.31	0.81	7.1	0.58	8.2	547	18.1
M_2	0.36	0.09	0.54	0.41	0.28	1.62	0.81	5.5	516	8.2	446	68400
M_3	0.33	•	10.4	0.59	•	33.75	22.73	•	DNF	DNF	•	•
M_4	•	•	0.35	•	•	0.40	•	•	0.93	•	•	4.3

Table 5.4: Time of TPA Build for the Generated XML dataset

Following graphs visualize measured values.

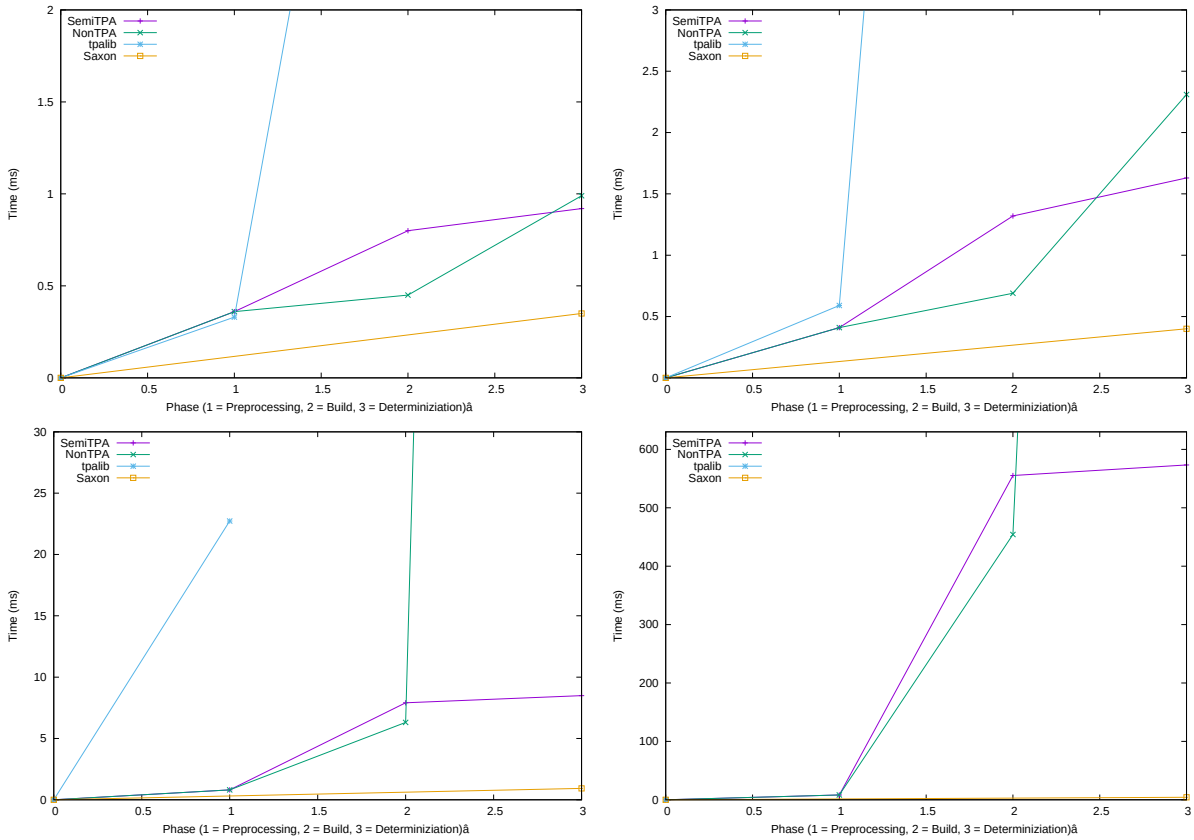


Figure 5.3: Time of TPA Build for the Generated XML dataset

5.3.2.4 Generated XML documents RAM Usage

The table presents values from the the conducted experiments. Values in rows M_1 and M_2 represent RAM usage (MB) after each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

Following graphs visualize measured values.

		D_4			D_5			D_6			D_7	
	pre	build	det	pre	build	det	pre	build	det	pre	build	det
M_1	7.5	14.0	14.1	7.5	15.7	15.9	37.7	64.4	103	221	464	741
M_2	7.5	13.5	17.8	7.5	13.5	17.5	37.7	67.7	112	221	541	762
M_3	10	•	159	17	•	306	126	•	DNF	DNF	•	•
M_4	•	•	21.2	•	•	11.3	•	•	46.2	•	•	448

Table 5.5: RAM Usage of TPA Build for the Generated XML dataset

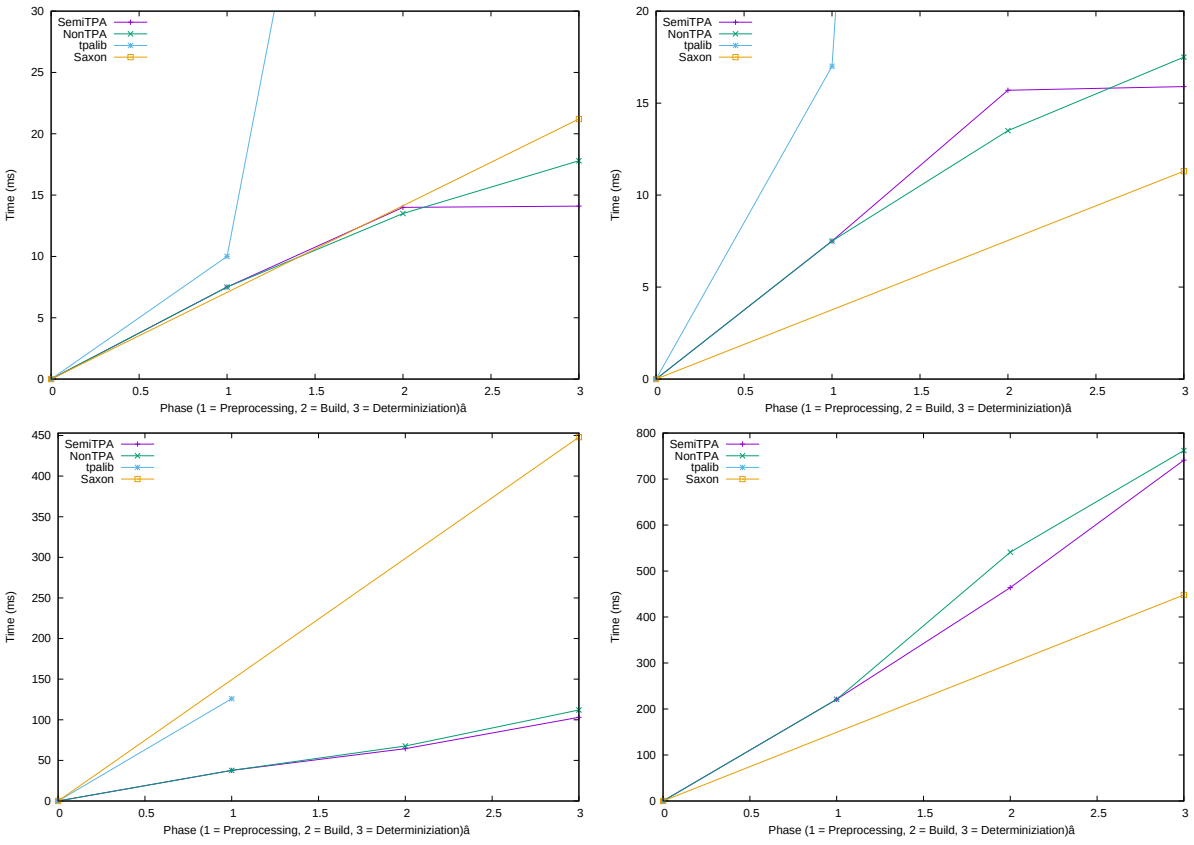


Figure 5.4: RAM Usage during TPA build for the Generated XML dataset

5.3.2.5 Real World documents Time

The table presents values from the conducted experiments. Values in rows M_1 and M_2 represent duration in miliseconds for each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

		D_8			D_9			D_{10}	
	pre	build	det	pre	build	det	pre	build	det
M_1	1.05	0.08	0.11	0.37	5.11	0.14	0.39	18.8	0.17
M_2	1.05	0.06	0.15	0.37	4.81	0.19	0.39	18.1	0.19
M_3	0.41	•	0.15	4.46	•	12.1	13.1	•	20.3
M_4	•	•	1.42	•	•	0.71	•	•	0.82

Table 5.6: Time of TPA Build for the Real World XML dataset

Following graphs visualize measured values.

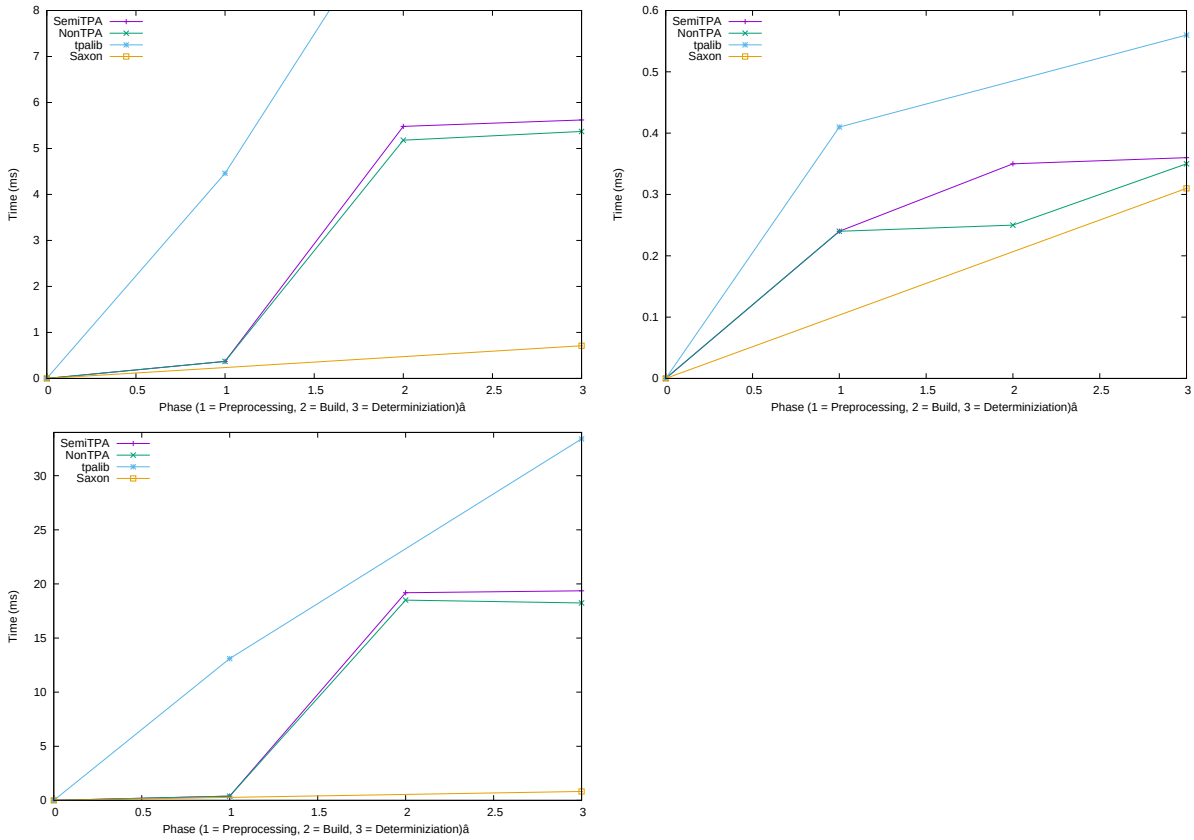


Figure 5.5: Time during TPA build for the Real World XML dataset

5.3.2.6 Real World documents Ram Usage

The table presents values from the the conducted experiments. Values in rows M_1 and M_2 represent RAM usage (MB) after each phase, namely Preprocessing, Build phase, Determinization. Abbreviations are used. As both the M_3 and the M_4 either does not contain or cannot be divided into the above mentioned phases, not every field is filled with measured data in the rows M_3 and M_4. The table of measured values follows:

		D_8			D_9			D_{10}	
	pre	build	det	pre	build	det	pre	build	det
M_1	10.1	10.2	10.2	15.4	26.6	27.4	27	52	57.4
M_2	10.1	10.2	10.2	15.4	26.2	27.1	27	39	57.7
M_3	2.2	•	9.1	33.2	•	265	67.2	•	571.4
M_4	•	•	19.4	•	•	23.3	•	•	24

Table 5.7: RAM Usage of TPA Build for the Real World XML dataset

Following graphs visualize measured values.

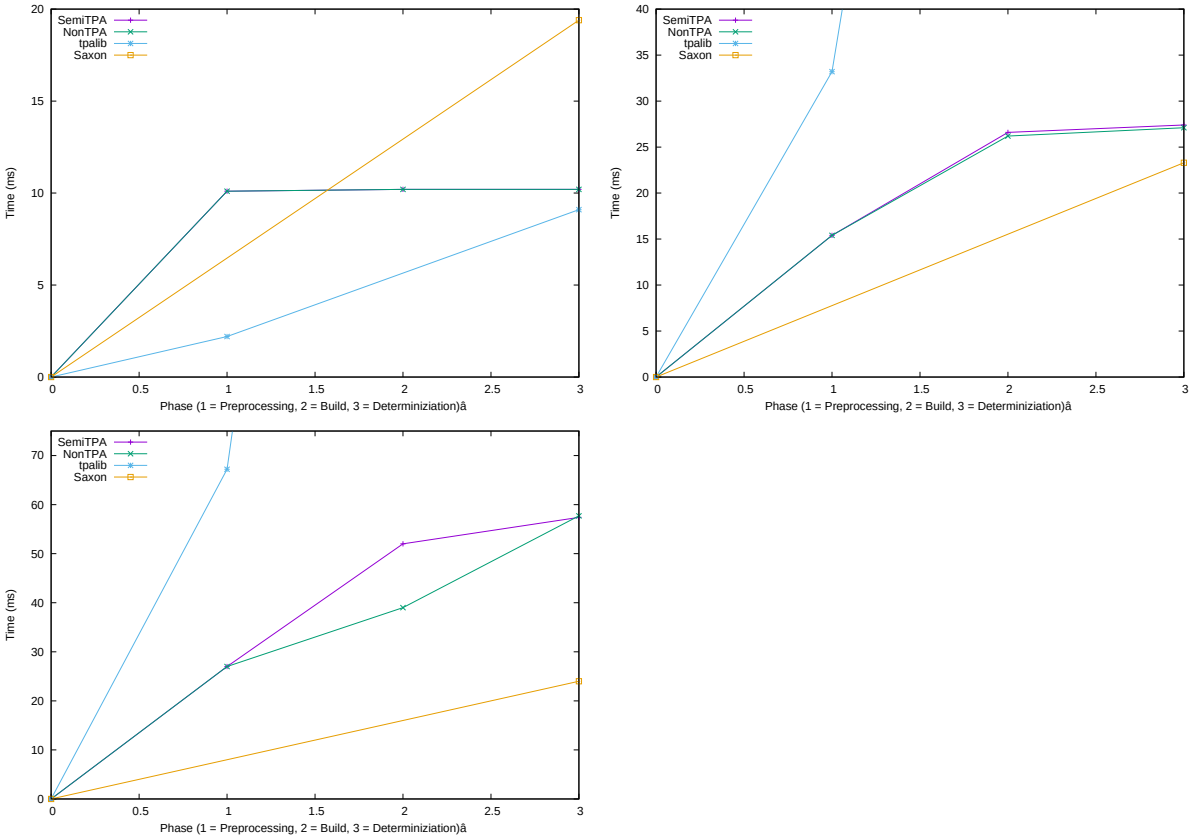


Figure 5.6: RAM usage during TPA build for the Real World XML dataset

5.3.3 Query evaluation

This subsection contains data from conducted experiments on query evaluation performance. Both backtracking approaches of the new algorithm 4 produce the same result. The following experiments use only the result of build methods. Therefore, it is sufficient to evaluate only the result of TPA indifferent to its backtracking approach during its build phase.

Following experiments share a common structure as follow. All XML documents in each dataset has very similar structure. The author chose one sample out of each dataset to represent it. For each of the samples 3 queries are run. These 3 queries has mutually different length, combination of axis and result in different size of the set of returned elements. Therefore performance of each method is evaluated well. Experiments starts at the moment of the beginning of resolving a query and end as soon as a method returns a result.

The RAM usage appeared constant because the amount of the used RAM for evaluation is too minor to the amount of a loaded index. Therefore a visualization of graphs is unnecessary .

5.3.3.1 Sample XML documents

For this dataset the author chose D_2 =NBASample2.xml to repsent the dataset. This XML document features the biggest structure out of others. The following table reveals measured values.

	Time (ms) & RAM Usage (MB)						
	TPA	M_3	M_4	TPA	M_3	M_4	$\#elements$
$Q_1 = //gleague//arena$	1	1	7	5	4	16	1
$Q_2 = //team/topplayer$	1	1	13	5	4	16	4
$Q_3 = /teams//team$	1	1	5	5	4	16	5

Table 5.8: Query evalution performance test for NbaSample2.xml

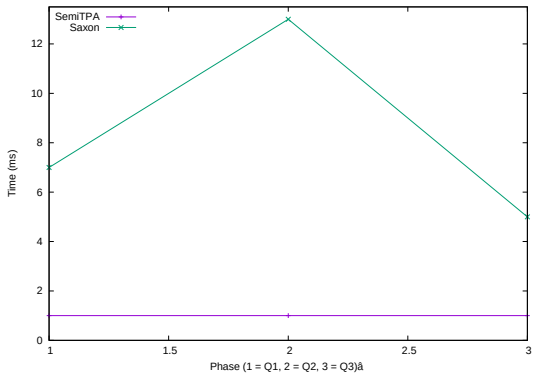


Figure 5.7: Time of evaluation for queries Q_1, Q_2, Q_3 for the D_2

5.3.3.2 Generated XML documents

For this dataset the author chose D_5 =XMark-f0.01.xml to repsent the dataset. This XML document is the biggest whose M_3 build method successfully finished. The following table reveals measured values.

	Time (ms) & RAM Usage (MB)						
	M_1	M_3	M_4	M_1	M_3	M_4	$\#elements$
$Q_4 = /site/open\_auctions$	1	1	8	15	31	9	1
$Q_5 = //site//open\_auction$	1	1	30	15	31	9	120
$Q_6 = //people/person$	1	1	21	15	31	9	255

Table 5.9: Query evalution performance test for XMark-f0.01.xml

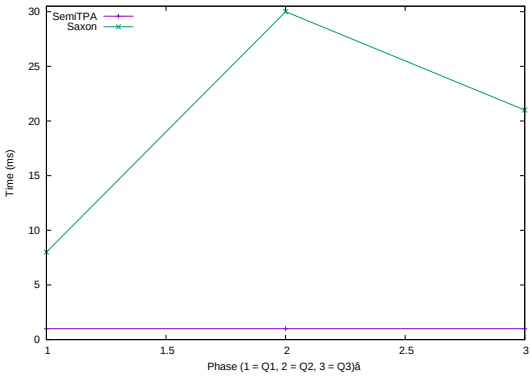


Figure 5.8: Time of evaluation for queries Q_4, Q_5, Q_6 for the D_5

5.3.3.3 Real World documents

For this dataset the author chose D_{10} = Orders.xml to represent the dataset. This XML document represents a typical XML document from everyday life. The following table reveals measured values.

	Time (ms) & RAM Usage (MB)						•
	M_1	M_3	M_4	M_1	M_3	M_4	$\#elements$
$Q_7 = /table/t/$	1	1	15	47	131	24	14995
$Q_8 = //t/o\_clerk$	1	1	123	47	131	26	14998
$Q_{=}//t/o\_orderkey$	1	1	131	47	131	24	14997

Table 5.10: Query evalution performance test for Orders.xml

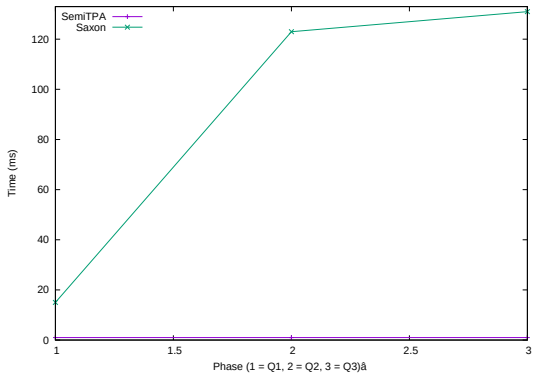


Figure 5.9: Time of evaluation for queries Q_7, Q_8, Q_9 for the D_{10}

5.4 Experimental Results Summary

Out of the results of the conducted experiments the method M_1 (SemiTPA) clearly dominates from M_2 (NonTPA) and M_3 (tpalib) in the means of performance during build. This is due to the new algorithm and the usage of advanced data structures. All of the mentioned methods M_1, M_2, M_3 evaluate queries in $O(|query| + |returnedelements|)$. Unlike the method M_4 (Saxon). However M_4 (Saxon) build time and RAM usage are clearly the lowest among all of the mentioned methods for files bigger than a few megabytes.

Chapter 6

Library Usage

The library is designed to create deterministic automata for indexing purposes and check features of XML documents. Author named the library BPIndex. Library contains following public methods: After a standard import process of the `BPIndex.jar` it is needed to import also `javax.ws.rs:javax.ws.rs-api:2.1` library.

- `getdTSPA()`
- `getdTSPSA()`
- `getdTPA()`
- `getnTPA()`
- `getResults()`
- `resolveQuery()`

A closer look to each of the above mentioned methods follows.

6.1 Method `getdTSPA()`

Method which returns a TSPA. The basic structure of the method is as follows.

On a returned object method `resolveQuery()` shall be called to evalaute a query. As the TSPA automaton answers only */transition* queries. The method returns the automaton significantly faster than `getdTPA()` or `getdTSPSA()` methods. This method has the most in common with TRIE concept out of the above mentioned 3 methods, namely `getdTPA()`, `getdTSPSA()` and `getdTSPA()`. The methods `getdTSPSA()` and `getdTPA()` extends the designe of the `getdTSPA()` method.

1. Check if serialize TSPSA exists for the given XML document
2. If the seriliaze automaton doesn't exist then create one and serialize it
3. If the seriliazed automaton exists then deserialize it
4. Return automaton

Figure 6.1: High level of TSPA structure

6.2 Method `getdTSPSA()`

Method which returns an instance of the DeterministicTSPSA class. The methods behaviour is structured in the same fashion as `getdTSPA()`. 6.1.

On a returned object method `resolveQuery()` shall be called to evalaute a query. The method `getdTSPSA()` is very close to the `getdTPA()` method, because the resulting automata are very similiar. In fact, Implementanting the `getdTSPSA()` is easier via the `getdTPA()` method.

6.3 Method `getdTPA()`

The method returns an instance of the DeterministicTPA class. This method is an implememtation of the new algorithm, with the semidetermenistic backtracking, suggested by the author. It further extends the concept of the `getdTSPA()` method. See 4 for detailed description of internal callings. On a returned object method `resolveQuery()` shall be called to evalaute a query. It is recommended to use this method as the resulting automaton is the most complex out of other possible ones. The methods behaviour is structured in the same fashion as `getdTSPA()` 6.1.

6.4 Method `getnTPA()`

The method has the very same output as the `getdTPA()` method. This method is an implememtation of the new algorithm, with the nondetermenistic backtracking, suggested by the author. This method is not as efficient as the `getdTPA()` one. It was created as a part of the conducted research. There are no benefits in using this method instead of using the `getdTPA()` method. The methods behaviour is structured in the same fashion as `getdTSPA()`. 6.1.

6.5 Method `getResults()`

For information about an XML document or a folder containing XML documents call the method `getResults()`. The result of this method (according to a given XML document) consists of: Name of the XML Document, Size of the XML Document, Maximal Depth, Average Depth, Number of Leaves, Number of Elements, Time to build TPA, Used RAM to build TPA and Number of States of TPA.

6.6 Method `resolveQuery()`

The method handles answering a query. Each automaton class has its own parser for the specific type of the supported queries. The method returns a collection of XMLTag instances. The begining of the thesis specifies a requirement which states to answer a query in $O(query.length() + result.length())$. This method meets the requirement.

Chapter 7

Goals Fulfillment

The goals were stated as follow.

1. Study TSPA,TSPSA and TPA
2. Implement TSPA,TSPSA and TPA
3. Support for experiments
4. Evaluation of the implemented methods

The author not only implemented but also came with the new algorithm for TPA. The idea of the new algorithm came from studying existing approaches and their implementation. The efficient implementation is due to studying previous attempts and following appropriate recommendations. The differences in both the time and the RAM consumption are significant. Up to 60 times faster and less than a half of the RAM consumption in the TPA build phase. What's more, the final index saved on a hard drive takes less than 50% of memory in comparison to the original one. The author conducted experiments for evaluation purposes. Three different types of datasets were made up to test performance of author's suggested methods. Experiments compared performance (Time and RAM usage) of four different methods, namely tpalib, Saxon, tpalibIM TPA, tpalibIM TPAn. The tpalib and Saxon were included for comparison reasons. Both author's methods performed better in the experiments than the original one from the tpalib. However, there is a significant difference in the performance of tpalibIM TPA and tpalibIM TPAn methods. The author does not recommend using tpalibIM TPAn as it was created as a part of the research. The experiments assured expectations that the tpalibIM TPA performs better in comparison to tpalib. The author also implemented a method into the library which returns values for experimental usage.

The author states that all of the goals were fulfilled.

Conclusion

The main goal of this thesis was the implementation and the experimental evaluation of the XML data indexing methods. The implemented methods were TSPA, TSPSA and TPA. Experiments conducted on XML data sets were focused on comparison of perfomance of the implemented method, clarifying relations between size and time to build an index. Both implementation goals were fulfilled. Implementation is written in Java. A further extension is possible due to an appropriate design. Library features user-friendly API. Figure 5.7 5.8 5.9 shows performance of the suggested solution versus wildly used Saxon library on the sample XML documents. The speed of answering a query is reached due to memorized structure of the given XML document. It is up to the user to decide whether it is worth it or not. The implementation significantly optimize time and memory usage. Furthermore, the implementation of the index is designed to correspond with the theoretical design of finite automata.

Bibliography

[1] Bray, T.; Paoli, J.; et al. Extensible markup language (XML). *World Wide Web Journal*, volume 2, no. 4, 1997: pp. 27–66.

[2] DeRose, S. XML Linking Language (XLink) Version 1.0. online, 2001. Available from: `http://www.w3.org/TR/xlink`

[3] Goldman, R.; Widom, J. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical report, Stanford, 1997.

[4] Pettovello, P. M.; Fotouhi, F. MTree: An XML XPath Graph Index. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-108-2, pp. 474–481, doi:10.1145/1141277.1141389. Available from: `http://doi.acm.org/10.1145/1141277.1141389`

[5] Zou, Q.; Liu, S.; et al. Ctree: a compact tree for indexing XML data. In *Web Information and Data Management*, 2004, pp. 39–46, doi:10.1145/1031453.1031462.

[6] Tang, N.; Yu, J.; et al. Hierarchical Indexing Approach to Support XPath Queries. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, April 2008, pp. 1510–1512, doi:10.1109/ICDE.2008.4497606.

[7] Kaushik, R.; Bohannon, P.; et al. Covering Indexes for Branching Path Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-497-5, pp. 133–144, doi:10.1145/564691.564707. Available from: `http://doi.acm.org/10.1145/564691.564707`

[8] Milo, T.; Suciu, D. Index Structures for Path Expressions. In *Database Theory — ICDT'99*, *Lecture Notes in Computer Science*, volume 1540,

edited by C. Beeri; P. Buneman, Springer Berlin Heidelberg, 1999, ISBN 978-3-540-65452-0, pp. 277–295, doi:10.1007/3-540-49257-7_18. Available from: http://dx.doi.org/10.1007/3-540-49257-7_18

[9] Rao, P.; Moon, B. PRIX: indexing and querying XML using prufer sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, March 2004, ISSN 1063-6382, pp. 288–299, doi:10.1109/ICDE.2004.1320005.

[10] Zhang, B.; Wang, W.; et al. AB-Index: An Efficient Adaptive Index for Branching XML Queries. In *Advances in Databases: Concepts, Systems and Applications*, *Lecture Notes in Computer Science*, volume 4443, edited by R. Kotagiri; P. R. Krishna; M. Mohania; E. Nantajeewarawat, Springer Berlin Heidelberg, 2007, ISBN 978-3-540-71702-7, pp. 988–993. Available from: http://dx.doi.org/10.1007/978-3-540-71703-4_90

[11] Chung, C.-W.; Min, J.-K.; et al. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-497-5, pp. 121–132, doi:10.1145/564691.564706. Available from: http://doi.acm.org/10.1145/564691.564706

[12] Li, Q.; Moon, B. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, ISBN 1-55860-804-4, pp. 361–370. Available from: http://dl.acm.org/citation.cfm?id=645927.672035

[13] Wang, H.; Park, S.; et al. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-634-X, pp. 110–121, doi:10.1145/872757.872774. Available from: http://doi.acm.org/10.1145/872757.872774

[14] Tung, H. D. T.; Luong, D. D. An Improved Indexing Method for Xpath Queries. *Indian Journal of Science and Technology*, volume 9, no. 31, 2016.

[15] Tatarinov, I.; Viglas, S. D.; et al. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 2002, pp. 204–215.

[16] Kha, D. D.; Yoshikawa, M.; et al. An XML indexing structure with relative region coordinate. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, IEEE, 2001, pp. 313–320.

[17] Clark, J.; DeRose, S. XML Path Language (XPath) Version 1.0. online, Nov 1999. Available from: `http://www.w3.org/TR/xpath`

[18] Šestáková, E.; Janoušek, J. Automata Approach to XML Data Indexing. *Information*, volume 9, no. 1, Jan 2018: p. 12, ISSN 2078-2489, doi:10.3390/info9010012. Available from: `http://dx.doi.org/10.3390/info9010012`

[19] Rabin, M. O.; Scott, D. Finite automata and their decision problems. *IBM journal of research and development*, volume 3, no. 2, 1959: pp. 114–125.

[20] Šestáková, E.; Janoušek, J. Tree string path subsequences automaton and its use for indexing xml documents. In *International Symposium on Languages, Applications and Technologies*, Springer, 2015, pp. 171–181.

[21] Šestáková, E. *Indexing XML documents*. Dissertation thesis, Master?s thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2015.

[22] Kay, M. Saxon XSLT and XQuery processor. *http://sourceforge.net/projects/saxon*, 2001.

[23] Hunter, J.; McLaughlin, B. The jdom project. *Available in http://www.jdom. org*, volume 114, 2000.

[24] Harold, E. R. *Processing Xml with Java*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN 0201771861.

[25] Daciuk, J.; Mihov, S.; et al. Incremental construction of minimal acyclic finite-state automata. *Computational linguistics*, volume 26, no. 1, 2000: pp. 3–16.

[26] Sgarbas, K. N.; Fakotakis, N. D.; et al. Incremental construction of compact acyclic NFAs. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, 2001, pp. 482–489.

[27] Aref, W. G. Trie based method for indexing handwritten databases. June 18 1996, uS Patent 5,528,701.

[28] Heafield, K. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Association for Computational Linguistics, 2011, pp. 187–197.

[29] Willard, D. E. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, volume 28, no. 3, 1984: pp. 379–394.

[30] Steele, K. M.; Agarwal, A. Pattern matching in a multiprocessor environment with finite state automaton transitions based on an order of vectors in a state transition table. Sept. 28 2010, uS Patent 7,805,392.

[31] Ficara, D.; Giordano, S.; et al. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, volume 38, no. 5, 2008: pp. 29–40.

[32] Wyschogrod, D.; Arnaud, A.; et al. Method of generating of DFA state machine that groups transitions into classes in order to conserve memory. July 3 2007, uS Patent 7,240,040.

[33] Arnold, K.; Gosling, J.; et al. *The Java programming language*. Addison Wesley Professional, 2005.

[34] Salomaa, K.; Yu, S. NFA to DFA transformation for finite languages over arbitrary alphabets. *Journal of Automata, Languages and Combinatorics*, volume 2, no. 3, 1998: pp. 177–186.

[35] Shepherdson, J. C. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, volume 3, no. 2, 1959: pp. 198–200.

[36] Clark, J.; DeRose, S.; et al. XML path language (XPath) version 1.0. 1999.

[37] Schmidt, A.; Waas, F.; et al. XMark: A benchmark for XML data management. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, Elsevier, 2002, pp. 974–985.

Appendix A

Acronyms

XML Extensible markup language

SAX Simple API for XML

TPA Tee Paths Automaton

TSPA Tree String Paths Automaton

TSPSA Tree String Path Subsequences Automaton

XPath XML Path Language

DNF Did Not Finished

API Application Programming Interface

RAM Random Access Memory

NFSA Non-deterministic Finite State Automata

DFSA Deterministic Finite State Automata

SemiTPA Semi-deterministic Backtracking TPA

NonTPA Non-deterministic Backtracking TPA

Appendix B

Contents of enclosed SD Card

/
- `readme.txt` ..... description of content of SD Card
- `src`
  - `impl`
    - `sourcecode` ..... raw java code
    - `BPIndex.jar` ..... BPIndex.jar
    - `Example.java` ..... runnable example
    - `Datasets` ..... xml datasets
  - `thesis`
    - `thesis.pdf` ..... thesis in pdf
    - `thesis.tex` ..... thesis source in LaTeX