Master's Thesis

Czech
Technical
University
in Prague

F8

Faculty of Information Technology
Department of Theoretical Computer Science

Indexing XML Documents

Bc. Eliška Šestáková
System Programming (Master)

May 2015
Supervisor: doc. Ing. Jan Janoušek, Ph.D.

Acknowledgement / Declaration

I take this opportunity to express gratitude to my supervisor doc. Ing. Jan Janoušek, Ph.D. for his motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. My sincere thanks also goes to all of the Department faculty members for their help and support.

Last but not the least, I would like to thank my parents for the unceasing encouragement, support and attention. I am also grateful to my partner who supported me through this venture.

In Prague on 5th May 2015

....

Abstrakt / Abstract

Výzkum v oblasti indexování řetězců má již mnoho prezentovaných výsledků, což však neplatí pro ostatní datové struktury, jakými jsou například stromy. Tato práce obsahuje v prvé řadě shrnutí metod pro indexování řetězců a stromů. Dále se podrobně zabývá rešerší existujících řešení indexování XML dokumentů.

Představena je zde nová jednoduchá metoda využívající deterministický konečný automat, jež umožňuje efektivně zpracovat XPath dotazy skládající se z libovolné kombinace child (/) a descendant-or-self (//) os, sloužících k navigaci v XML dokumentu. Spolu s touto metodou byly dále navrženy dva další konečné automaty na podporu jednodušších dotazů obsahujících vždy pouze jednu z uvedených os.

Ke konstrukci indexu pro daný XML dokument $\mathcal{D}$ s n elementy je využit odpovídající XML stromový model $\mathcal{T}$. Zpracování dotazu Q o m elementech proběhne v čase $\mathcal{O}(m)$ nezávislém na n. Výsledkem dotazu je poté množina elementů splňujících dané požadavky.

Ačkoli automat podporující všechny dotazy s // osou indexuje až $\mathcal{O}(2^n)$ různých dotazů, počet stavů vlastního deterministického automatu je $\mathcal{O}(h^k)$, kde h je výška XML stromového modelu $\mathcal{T}$ a k je počet listů $\mathcal{T}$. Pro běžné XML dokumenty lze navíc tuto mez triviálně snížit až na $\mathcal{O}(h.2^k)$.

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

Překlad titulu: Indexování XML dokumentů

The theory of text indexing is very well-researched, which does not hold for theories of indexing other data structures, such as trees for example. In this thesis we review existing techniques for indexing texts and trees and study state-of-the-art methods for indexing XML documents. We show that automata can be used effectively for the purpose of indexing XML documents.

A new and simple method for indexing XML documents using deterministic finite automaton is introduced. The presented method supports a significant fragment of XPath queries which may use any combination of child (/) and descendant-or-self (//) axes. We also propose another two indexing techniques based on finite automata, aimed to assist in evaluating paths queries with either / or // axis only.

Given a subject XML document $\mathcal{D}$ and its corresponding XML tree model $\mathcal{T}$ with n nodes, the tree is preprocessed and the index is constructed. The searching phase uses the index, reads an input query Q of size m and computes the list of positions of all occurrences of target nodes of Q in $\mathcal{T}$.

All the proposed automata performed the searching in time $\mathcal{O}(m)$ and do not depend on n. Although the automaton that supports all linear XPath queries where just // axis is used evaluates $\mathcal{O}(2^n)$ distinct queries, number of states of the deterministic automaton is $\mathcal{O}(h^k)$, where h is the height of $\mathcal{T}$ and k is the number of its leaf nodes. Moreover, we discuss that in case of indexing a common XML document the number of state in the deterministic finite automaton is at most $\mathcal{O}(h.2^k)$.

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

Contents /

1 Introduction 1
1.1 Motivation and Objectives 1
1.2 Problem Statement 1
1.3 Goals of the Thesis 2
1.4 Structure of the Thesis 2
2 Theoretical Background 3
2.1 Notations 3
2.2 Basic Definitions 4
2.2.1 Alphabet 4
2.2.2 String 4
2.2.3 Graph, Tree 5
2.2.4 Language, Grammar 6
2.2.5 Finite and Pushdown Automaton 6
2.3 XML 7
2.3.1 XML Data Model 9
2.4 XPath 11
2.4.1 Syntax and Semantics 11
2.4.2 Examples 13
3 Arbology and XPath Relation 15
3.1 Classification of XPath Queries 17
3.1.1 Query Constructs 17
3.1.2 Query Orientation 17
3.1.3 Query Structure 19
3.1.4 Query Arity 19
4 Indexing Texts and Trees 21
4.1 Text Indexing 21
4.1.1 Data Structures for Storing the Suffixes 21
4.1.2 Factor Automaton 23
4.1.3 Subsequence Automaton 24
4.2 Tree Indexing 24
4.2.1 Subtree Pushdown Automaton 25
4.2.2 Tree Pattern Pushdown Automaton 27
4.2.3 A Full and Linear Index of a Tree 27
5 Indexing XML Data 31
5.1 MTree 31
5.1.1 MTree Index Structure 32
5.1.2 Query Processing 32
5.2 CTree 34
5.2.1 CTree Index Structure 34
5.2.2 CTree Properties 37
5.2.3 Query Processing 37
5.3 PP-Index 38
5.3.1 PP-Index Structure 38
5.3.2 Query Processing 40
5.4 PRIX 44
5.4.1 PRIX Index Structure 44
5.4.2 Prüfer Sequence Properties 46
5.4.3 Query Processing 46
6 Automata Approach to Indexing XML Data 49
6.1 XML Data Model 49
6.2 Tree String Paths Automaton 51
6.2.1 Discussion on Time and Space Complexities 55
6.3 Tree String Path Subsequences Automaton 55
6.3.1 Building the Tree String Path Subsequences Automaton 55
6.3.2 Discussion on Time and Space Complexities 57
6.4 Tree Paths Automaton 63
6.4.1 Building the Tree Paths Automaton 63
6.4.2 Discussion on Time and Space Complexities 65
7 Implementation 69
7.1 System Architecture 69
7.1.1 JDOM 69
7.1.2 Index Builder 70
7.1.3 XML Data Index 71
7.2 Query Processing 72
8 Experimental Evaluation 73
8.1 Experimental Setup 73
8.1.1 XML Data Sets 73
8.2 Index Size and Construction Time 74
8.3 Performance on Query Processing 75
9 Conclusions 77
9.1 Summary of the Thesis 77
9.2 Contributions of the Thesis 77
9.3 Future Work 78
References 81
A Acronyms 85

B User’s Manual for tpalib 87
C Syntax Structure of Complete XPath 1.0 89
D Contents of Enclosed CD 91

Tables / Figures

2.1. XPath expressions in abbreviated and full syntax . . . 13
4.1. Subtree jump table . . . 29
6.1. Working data structures for Tree Paths Automaton construction . . . 64
6.2. Set of occurrences of element labels in an XML tree model . . 64
8.1. Characteristics of XMark benchmark files . . . 74
8.2. Experimental results on construction time and size . . . 74
8.3. Ratio of index size to XML file size . . . 74
8.4. Set of queries used in performance analysis . . . 75
8.5. Numbers of elements satisfying queries in test data sets . . . 75
8.6. `tpalib` experimental evaluation . . . 76
8.7. Saxon experimental evaluation . 76

2.1. XML tree model . . . 10
2.2. XPath axes . . . 12
3.1. Tree model of an XPath query . 15
3.2. Tree model of an XPath query . 15
3.3. Tree model of an XPath query . 15
3.4. Occurrences of an XPath query in an XML tree model . . 16
3.5. Occurrences of an XPath query in an XML tree model . . 16
3.6. Occurrences of an XPath query in an XML tree model . . 16
3.7. An XPath query . . . 17
3.8. An XPath query . . . 17
3.9. XPath queries classification . . . 18
4.1. Data structures for suffixes . . . 22
4.2. Position heap . . . 23
4.3. Nondeterministic factor automaton . . . 24
4.4. Nondeterministic subsequence automaton with ε-transitions . . . 24
4.5. Nondeterministic subsequence automaton after ε-transitions removal . . . 24
4.6. Ranked tree . . . 25
4.7. Nondeterministic subtree pushdown automaton . . . 26
4.8. Nondeterministic tree pattern pushdown automaton . . . 28
4.9. Compact suffix automaton . . . 29
5.1. f-graph . . . 33
5.2. p-graph . . . 33
5.3. a-graph . . . 33
5.4. d-graph and f-graph . . . 33
5.5. q-graph . . . 33
5.6. $attr$-graph . . . 33
5.7. MTree graph index . . . 35
5.8. Ordered path summary . . . 37
5.9. CTree . . . 37
5.10. An XPath query . . . 38
5.11. 1-Index . . . 40
5.12. Containment index . . . 41
5.13. PP-Index . . . 42
5.14. Compressed PP-Index . . . 43
5.15. XML tree model . . . 45
5.16. An XPath query . . . 46

5.17. An XPath query 46
6.1. XML tree model 50
6.2. Prefix automata for a string path set 53
6.3. Tree String Paths Automaton . 54
6.4. Subsequence automata 58
6.5. Deterministic subsequence automata 59
6.6. Deterministic subsequence automata 60
6.7. Tree String Path Subsequences Automaton 61
6.8. Nondeterministic Tree Paths Automaton 66
6.9. Deterministic Tree Paths Automaton 67
7.1. System architecture of `tpalib` . 69
7.2. `DocumentLoader` class 70
7.3. `AutomatonFactory` and `Node` class 70
7.4. `EdgeLabel` class and `Axis` enum 71
7.5. `TreePathsAutomaton` class 71
7.6. `State` class 72
8.1. Partial schema of XMark data sets 73
8.2. Performance comparison of `tpalib` and Saxon 76

Chapter 1
Introduction

1.1 Motivation and Objectives

XML documents [1] are used in many aspects of software development, often to simplify data storage and sharing. XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data. Therefore, it reduces the complexity of data transport between incompatible systems over the Internet, since XML documents can be read by different incompatible applications.

To be able to retrieve the data from XML documents efficiently, various query languages, such as XPath [2], XPointer [3] and XLink [4], have been designed. Purpose of these languages is not only making queries, but also possibility of creating views over the XML document, data transformation and specification of data subset to be updated.

The query is an expression containing keywords as basic elements. The frequently used operation in data retrieval systems is selection of documents containing some keywords of a given query. The subfield of algorithmic research interested in question whether given keyword is contained in the document is called *stringology*[1]).

To achieve fast text searching and efficient processing of queries, we can preprocess the data subject and construct an index. The size of the index is typically linear to the size of the preprocessed subject. Therefore it allows us to answer number of queries with low requirements for both space and time complexity.

An XML document can be simply treated as a stream of plain text, thus stringology algorithms are applicable in this field. However, we can think of an XML document as a kind of a linear notion of a tree structure. The algorithmic discipline interested in processing tree data structures is called *arbology*[2]). Arbology solves problems such as tree pattern matching, tree indexing, finding repeats in trees, etc. For its algorithms, the arbology uses a standard *pushdown automaton* as the basic model of computation, unlike stringology where a *finite automaton* is used.

1.2 Problem Statement

The XML index problem means to construct an effective XML index structure that is preferably small in size and is able to efficiently process an XML query language or just its fragment. For instance, XPath which stands for XML Path Language, is one of the commonly used XML query languages. It operates on tree structured XML documents and its primary purpose is to address the nodes (i.e., elements, attributes, etc.) of trees or specify data subsets to be selected.

[1]) Stringology (term was for the first time used by Zvi Galil in 1984 [5]) denotes a science on algorithms on strings and sequences. It solves such problems like exact and approximate pattern matching, searching for repetitions in various texts, etc.

[2]) Arbology [6] (from the spanish word *arbol*, meaning tree) is a new algorithmic discipline. It was officially introduced at London Stringology Days 2009 conference.

However, without a structural summary, a query processing can be quite inefficient due to an exhaustive traversal on XML data. Indexing the structure of XML data is an effective way to accelerate the query processing, since it can greatly reduce the search space. Therefore, several XML documents indexes have been proposed.

We can classify the indexes into several categories such as: *graph-based* methods constructing a structural path summary that can be used to improve query efficiency, especially for single path queries. Next category covers *sequence-based* methods transforming both the source data and query into sequences. Therefore, querying XML data is equivalent to finding subsequence matches. *Node coding* methods form the next category. These methods apply certain coding strategy to design codes for each node, in order that the relationship among nodes can be evaluated by computation. The last category includes *adaptive* methods that can adapt their structure to suit the query workload by indexing the most frequent queries only.

1.3 Goals of the Thesis

The primary aim of this thesis is summarized as follows:

- Study algorithmic methods for indexing texts and tree data structures.
- Review current state-of-the-art methods for XML index problem.
- Propose a new method for the purpose of indexing XML documents that would be based on suitable formal models from the theory of formal languages and automata.
- Discuss theoretical and time and space complexities of proposed method, implement it and perform appropriate testing of the implementation.

1.4 Structure of the Thesis

The rest of the thesis is organised as follows. In Chapter 2, we provide the necessary theoretical background, i.e., notations used through this thesis, essential definitions concerning trees and theory of formal languages and automata along with a brief introduction to XML and XPath. We discuss the relationship between arbology and XPath in Chapter 3. The Chapter 4 is an overview of the texts and trees indexing techniques. The state-of-the-art methods for XML index problem are discussed in Chapter 5 in detail. In Chapter 6 we show that automata can be used effectively for the purpose of indexing XML documents and introduce three types of such automata. Chapters 7 and 8 deal with the implementation and the experimental evaluation of the proposed methods, respectively. The last part of the thesis is a summary of the thesis results and contributions along with a discussion about future work.

Chapter 2 Theoretical Background

In this chapter, all the notations through this thesis are given along with basic definitions. Next, we present a fairly short introduction to XML and rather illustrate it by means of an example. In the end, the XPath query language is introduced.

2.1 Notations

Notations used throughout the thesis are given:

- $\mathcal{A}$ for an alphabet,
- a for an alphabet symbol,
- w, x, y, z for strings,
- $\mathcal{L}$ for a language,
- $\mathcal{N}$ for a set of nonterminal symbols,
- S for a start symbol of a grammar,
- p, q for states,
- q_0 for an initial state,
- $\mathcal{Q}$ for a set of states,
- F for a set of final states,
- G as a pushdown store alphabet,
- Z_0 as an initial pushdown store symbol,
- δ for automaton transition function,
- $\mathcal{M}$ for both a finite and a pushdown automaton,
- $\mathcal{G}$ 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),
- T for an arbitrary tree,
- $\mathcal{D}$ for an XML document,
- $\mathcal{T}$ for an XML tree model,
- $\mathfrak{n}$ for a node of both an XML tree model and an arbitrary tree,
- r for a root node of both an XML tree model and an arbitrary tree,
- $\mathcal{E}$ for an XML alphabet,
- e for an XML element,
- Q for an XML query,
- h for a height of both an XML tree model and an arbitrary tree,

- k for a number of leaves of both an XML tree model and an arbitrary tree,
- l for a label of a tree node or an XML element,
- P for a string path and for a string or tree pattern,
- $\mathcal{P}$ 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,
- $O_\mathcal{T}(l)$ for a set of occurrences of an element label l in an XML tree model $\mathcal{T}$.

2.2 Basic Definitions

2.2.1 Alphabet

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

Definition 2.2. (Ranked alphabet). A *ranked alphabet* is an alphabet where each symbol of a set has a unique nonnegative arity (or rank). ♣

Definition 2.3. (Arity of a symbol). Given a ranked alphabet $\mathcal{A}$, the *arity of a symbol* $a \in \mathcal{A}$ is denoted $arity(a)$. As a short declaration of symbols with arity we use superscripts (for instance, a_2 is a short declaration of a symbol a of arity 2). ♣

2.2.2 String

Definition 2.4. (String). A *string* over a given alphabet $\mathcal{A}$ is a finite sequence of symbols of $\mathcal{A}$. ♣

Definition 2.5. (Length of a string). A *length* of a string x is the number of its symbols and is denoted by $|x|$. ♣

Definition 2.6. (Empty string). An *empty string* is an empty sequence of symbols denoted by ε. ♣

Definition 2.7. (Set of all strings). A *set of all strings* over a given alphabet $\mathcal{A}$ is denoted by $\mathcal{A}^*$. ♣

Definition 2.8. (Set of all nonempty strings). A *set of all nonempty strings* over a given alphabet $\mathcal{A}$ is denoted $\mathcal{A}^+$. ♣

Definition 2.9. (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 2.10. (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 2.11. (Factor). A *factor (substring)* of a string $x = x_1x_2 \dots x_n$ is a string $y = x_ix_{i+1} \dots x_j$, where $1 \leq i \leq j \leq n$. ♣

Definition 2.12. (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. ♣

2.2.3 Graph, Tree

Definition 2.13. (Directed graph). A *directed graph* $\mathcal{G}$ is a pair (N, R), where N is a set of nodes and R is a set of lists of edges such that each element of R is of the form $((v, u_1), (v, u_2), \ldots, (v, u_3))$, where $v, u_1, u_2, \ldots, u_n \in N, n \geq 0$. This element will indicate that, for node v, there are n edges leaving v, entering node u_1, node u_2, and so forth. ♣

Definition 2.14. (Path). A sequence of nodes $(v_0, v_1, \ldots v_n), n \geq 1$ is a *path* of length n from node v_0 to node v_n if there is an edge which leaves node v_{i-1} and enters node v_i for $1 \leq i \leq n$. ♣

Definition 2.15. (Cycle). A *cycle* is a path $v_0, v_1, \ldots v_n$, where $v_0 = v_n$. ♣

Definition 2.16. (Directed acyclic graph). A *directed acyclic graph* is an directed graph that has no cycle. ♣

Definition 2.17. (Labelling). A *labelling* of a graph $\mathcal{G} = (N, R)$ is a mapping N into a set of labels. ♣

Definition 2.18. (Out-degree, in-degree). Given a node v, its *out-degree* is the number of distinct pairs $(v, u) \in R$, where $u \in N$. By analogy, the *in-degree* of node v is the number of distinct pairs $(u, v) \in R$ where $u \in N$. ♣

Definition 2.19. (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 2.20. (Rooted directed tree). A *rooted and directed tree* T is a directed acyclic graph $T = (N, R)$ with a special node $r \in N$, called the *root*, such that

- r has in-degree 0,
- all other nodes of T have in-degree 1,
- there is just one path from the root r to every node $\mathfrak{n} \in N$, where $\mathfrak{n} \neq r$. ♣

Definition 2.21. (Leaf). Let $T = (N, R)$ be a rooted directed tree. Node $\mathfrak{n} \in N$ is called a *leaf* if it has out-degree 0. ♣

Definition 2.22. (Labelled tree). A *labelled* (rooted, directed) tree is a tree having the following property:

- every node $\mathfrak{n} \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet. ♣

Definition 2.23. (Ordered tree). An *ordered* (labelled, rooted, directed) tree is a tree where direct descendants $\mathfrak{n}_1, \mathfrak{n}_2, \ldots, \mathfrak{n}_m$ of a tree node $\mathfrak{n}$ with an out-degree m are ordered. ♣

Definition 2.24. (Ranked tree). A *ranked* (labelled, rooted, directed) tree is a tree labelled by symbols from a ranked alphabet and out-degree of a node $\mathfrak{n}$ labelled by symbol $a \in \mathcal{A}$ is $arity(a)$. ♣

Definition 2.25. (Unranked tree). A (labelled, rooted, directed) tree is called *unranked* if it is not ranked. ♣

Definition 2.26. (Subtree). Let $T = (N, R)$ be a rooted directed tree. A *subtree* $T' = (N', R')$ of T is a rooted directed tree, where $N' \subseteq N \wedge R' \subseteq R$. Also, if $\mathfrak{n}$ is a leaf in T', then $\mathfrak{n}$ is a leaf in T. ♣

Definition 2.27. (Tree pattern). To define a tree pattern, we use special symbol S with arity 0, not in $\mathcal{A}$, which serves as a placeholder for any subtree. A *tree pattern* is defined as a labelled ordered ranked tree over ranked alphabet $\mathcal{A} \cup S$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. ♣

Definition 2.28. (Tree template). A *tree template* is a tree pattern containing at least one symbol S. ♣

Definition 2.29. (Prefix notation of a tree). A *prefix notation* $pref(T)$ of an ordered, labelled, rooted, directed tree T is defined in this way:

1. $pref(T) = \mathfrak{n}$ if T has only node $\mathfrak{n}$,
2. $pref(T) = \mathfrak{n}\, pref(\mathfrak{n}_1) \dots pref(\mathfrak{n}_m)$, where $\mathfrak{n}$ is the root of the tree T and $\mathfrak{n}_1, \mathfrak{n}_2 \dots \mathfrak{n}_m$ are direct descendants of $\mathfrak{n}$. ♣

Definition 2.30. (Arity checksum). Let $x = a_1 a_2 \dots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then the *arity checksum* of string x denoted $ac(x)$ can be computed as follows:

$$ac(x) = \sum_{i=1}^{m} arity(a_i) - m + 1$$ ♣

2.2.4 Language, Grammar

Definition 2.31. (Language). A *language* $\mathcal{L}$ over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. ♣

Definition 2.32. (Grammar). A *grammar* is a quadruple $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{P}, S)$, where

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

Definition 2.33. (Regular grammar). A grammar $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{P}, S)$ is called *regular*, if every rule is of the form $A \to aB$ or $A \to a$, where $A, B \in \mathcal{N}, a \in \mathcal{A}$. 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 2.34. (Context-free grammar). A grammar $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{P}, S)$ is called *context-free*, if every rule is of the form $A \to \alpha$, where $A \in \mathcal{N}, \alpha \in (\mathcal{N} \cup \mathcal{A})^*$. ♣

2.2.5 Finite and Pushdown Automaton

Definition 2.35. (Deterministic finite automaton). A *deterministic finite automaton* (DFA) is a quintuple $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$, where

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

Definition 2.36. (Total DFA). A DFA $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$ is called *total* if the mapping δ is defined for all pairs of state $q \in \mathcal{Q}$ and symbol $a \in \mathcal{A}$. ♣

Definition 2.37. (Nondeterministic finite automaton). A *nondeterministic finite automaton* (NFA) is a quintuple $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$, where

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

Definition 2.38. (d-subset). Let $\mathcal{M}_1 = (\mathcal{Q}_1, \mathcal{A}, \delta_1, q_{01}, F_1)$ be a nondeterministic finite automaton. Let $\mathcal{M}_2 = (\mathcal{Q}_2, \mathcal{A}, \delta_2, q_{02}, F_1)$ be the deterministic finite automaton equivalent to automaton $\mathcal{M}_1$. Automaton $\mathcal{M}_2$ is constructed using the standard determinisation algorithm based on subset construction [7]. Every state $q \in \mathcal{Q}_2$ corresponds to some subset d of $\mathcal{Q}_1$. 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 $\mathcal{M}_1$ considered as natural numbers. ♣

Definition 2.39. (Nondeterminsitic pushdown automaton). A *nondeterminsitic pushdown automaton* is a seven-tuple $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where

- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{A}$ is an input alphabet,
- G is a pushdown store alphabet,
- δ is a mapping from $\mathcal{Q} \times (\mathcal{A} \cup \{\varepsilon\}) \times G$ into a set finite subsets of $\mathcal{Q} \times G^*$,
- $q_0 \in \mathcal{Q}$ is an initial state,
- $Z_0 \in G$ is the initial pushdown store symbol,
- $F \subset \mathcal{Q}$ is the set of final (accepting) states. ♣

Definition 2.40. (Deterministic pushdown automaton). A pushdown automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, G, \delta, q_0, Z_0, F)$ is *deterministic*, if the following holds:

1. $|\delta(q, a, \gamma)| \leq 1, \forall q, a, \gamma$ where $q \in \mathcal{Q}, a \in (\mathcal{A} \cup \{\varepsilon\}), \gamma \in G^*$,
2. If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, then α is not a suffix of β and β is not a suffix of α (i.e., $\gamma\alpha \neq \beta$, $\alpha \neq \gamma\beta$),
3. If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α (i.e., $\gamma\alpha \neq \beta$, $\alpha \neq \gamma\beta$). ♣

2.3 XML

Extensible Markup Language [1], abbreviated XML, was defined by the consortium W3C's XML 1.0 Specification as a format used for transfer of general documents and data. XML is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable. The XML specification defines an XML document as a well-formed text, meaning that it satisfies a list of syntax rules provided in the specification.

Definition 2.41. (Well-formed XML document). A *well-formed* XML document is a document that adheres to the syntax rules specified by the XML 1.0 specification in that it must satisfy both physical and logical structures. ♣

Set of marks of an XML document is not fixed and can be defined in various ways for each document (unlike HTML which uses fixed set of marks to express form of the final document). Both XML and HTML are restricted forms of SGML[1]). The key constructs of an XML document are as follows:

- **tag** a markup construct that begins with `<` and ends with `>`. Tags can be classified into following categories:
    - start-tags,
    - end-tags,
    - empty-element tags.
- **element** a logical document component which either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The symbols between the start- and end-tags, if any, are the element's content.
- **attribute** a markup construct consisting of a name-value pair separated by equality sign that exists within a start-tag or an empty-element tag. The value of an attribute is always atomic, hence cannot be nested.

We will illustrate the key constructs of an XML document by means of an example. Consider the XML document in Example 2.42 which displays basic information about noble houses of the Seven Kingdoms in Westeros[2]). We can see that `HOUSES` is the most outer element. A *start-tag* of this element is of the form `<HOUSES>`, whereas the corresponding *end-tag*, indicating the end of the element, is `</HOUSES>`. So, the content between and including the tags `<HOUSES>` and `</HOUSES>`, constitutes a `HOUSES` *element*. Elements can be arbitrarily nested inside other elements. For instance, the `HOUSES` element has two `HOUSE` elements as its subelements. Every `HOUSE` element includes `LORD` and `SIGIL` as its first and second subelement, respectively and optionally `SEAT` as its third subelement. Another key construct of an XML document are *attributes*. For instance, `<HOUSE name="Stark">` indicates that the value of the `name` attribute of that particular `HOUSE` element is `"Stark"`.

Example 2.42. A sample of a well-formed XML document.

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

♠

Usually, we are interested in documents that satisfy some specific constraints, rather then documents containing arbitrary elements. Such constraints can be defined by means of *Document Type Definitions*, abbreviated DTD, describing a schema of corresponding XML documents.

[1]) Standard Generalized Markup Language (SGML) is a standard for how to specify a document markup language or a tag set.
[2]) Westeros is a fictional continent of an American fantasy drama named Game of Thrones.

A DTD describing schema of the XML document from Example 2.42 is shown in Example 2.43. The DTD specifies that `HOUSES` is the most outer element and it allows an unbounded number of `HOUSE` elements to be nested. Also, we can learn that every `HOUSE` element includes at first `LORD` and then `SIGIL` as obligatory elements followed by optional `SEAT` element. So, `*`, `?` and `,` denote any number of occurrences, optionality and concatenation, respectively. `#PCDATA` indicates that the element has no subelements but consists of text only. By `ATTLIST` we determine which attribute belongs to which element. In this DTD an attribute `NAME` belongs to `HOUSE` element and can only have a single string value.

Example 2.43. A DTD describing a structure of the XML document from Example 2.42.

```
<!DOCTYPE HOUSES [
    <!ELEMENT HOUSES  (HOUSE*)>
    <!ELEMENT HOUSE   (LORD, SIGIL, SEAT?)>
    <!ELEMENT LORD    (#PCDATA)>
    <!ELEMENT SIGIL   (#PCDATA)>
    <!ELEMENT SEAT    (#PCDATA)>
    <!ATTLIST HOUSE NAME CDATA>
]>
```

♠

In case of simple XML documents we get along with the DTD language as a tool of schema specification. However, complex documents often need complex structure specification, so DTDs are not sufficient. For this reason, there exists XML Schema language containing additional constructs with significant power of expression.

2.3.1 XML Data Model

Although an XML document may have complex internal structures, it can be viewed as a tree in a natural way. Most logical formalism for trees deal with *ranked trees* (see formal Definition 2.24). In such trees, all nodes have the same fixed number of children or a bit more generally, the number of children of a node is determined by the label of that node [8]. However, XML documents have frequently no restrictions on the number of children a node can have (e.g., the DTD from Example 2.43 allows an unbounded number of `HOUSE` elements and optionality of `SEAT` element).

Therefore, XML data is typically modelled as labelled *unranked trees* [9] (see formal Definition 2.25) over a finite alphabet determined, for instance, by a DTD. There is no unique best way for how to encode XML documents as trees. One possibility, on the basis of variety of XML data, is to introduce following different types of tree nodes [10]:

- root,
- element,
- text,
- attribute,
- comment,
- processing instruction,
- namespace.

As a root of the XML tree model we can consider the most outer element. Except for attribute and namespace nodes, the edges represent the parent–child (or element–subelement) relationship. The attribute and namespace nodes are not actually contained inside the element as they give only additional information. Hence, they are not considered to be in a parent–child relationship.

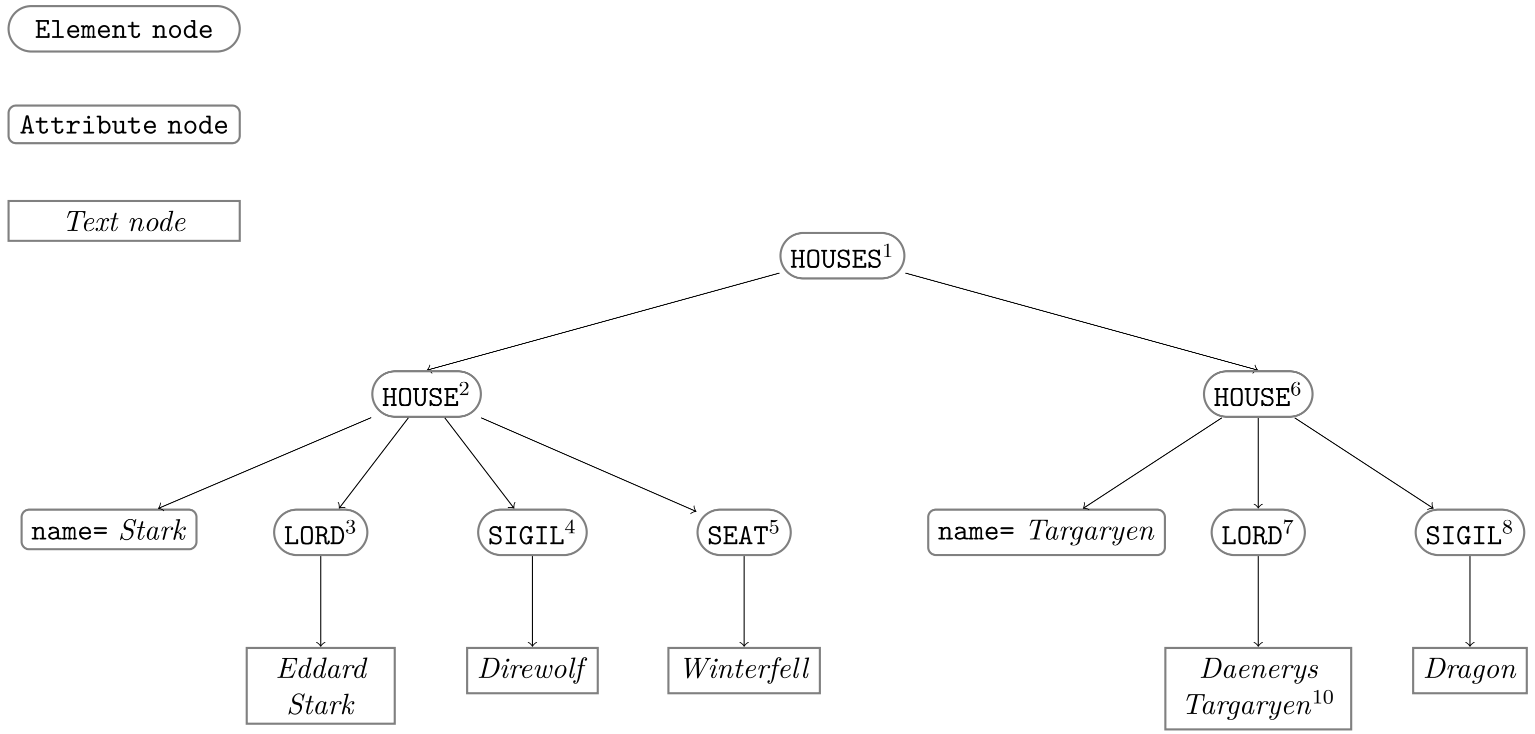


Figure 2.1. XML tree model (preorder numbering scheme – element nodes)

Example 2.44. Let $\mathcal{D}$ be the XML document from Example 2.42. The corresponding XML tree model $\mathcal{T}$ is illustrated in Figure 2.1. ♠

2.4 XPath

XML Path Language (XPath) 1.0 [2] is an XML query language that became a W3C Recommendation in 1999. The language gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document and operating on its tree structure. It is a query language for selecting nodes from an XML document but can also be used to compute values (e.g., strings, numbers, or Boolean values) from the content of an XML document. XPath is a major element in the XSLT standard and both XQuery and XPointer are built on XPath expressions.

2.4.1 Syntax and Semantics

The primary purpose of XPath is to address nodes of an XML tree model using a compact, non-XML syntax. XPath expressions can be quite simple or very complex. The syntax structure of simplified version of XPath can be expressed by EBNF[1]) as follows (for complete XPath see Appendix C):

```
AbsoluteLocationPath = '/' , [ LocationPath ] ;
LocationPath = Step
              | LocationPath '/' Step
Step = AxisSpecifier , NodeTest , [ Predicate ] ;
AxisSpecifier = AxisName , '::' ;
AxisName = 'ancestor'
            | 'ancestor-or-self'
            | 'attribute'
            | 'child'
            | 'descendant'
            | 'descendant-or-self'
            | 'following'
            | 'following-sibling'
            | 'parent'
            | 'preceding'
            | 'preceding-sibling'
            | 'self' ;
NodeTest = '*'
           | 'ElementName'
           | NodeType , '(' , ')' ;
NodeType = 'text'
        | 'node' ;
Predicate = '[' , Number , ']' ;
Number = Digit , { Digit } ;
Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;
```

XPath uses path expressions to select nodes in an XML document. The path expressions are used to select nodes or node-sets in an XML document. Such nodes are usually referred to as target nodes. A path consists of a sequence of steps, where each step

[1]) ISO/IEC 14977 Extended BNF [11] defines a standard syntactic metalanguage based on BNF, which can be used to express a context-free grammar.

has following structure: a location path (i.e,. “where to look for”), node tests (i.e., identifying a node) and predicates (i.e., additional tests).

`axis::node-test[predicate]*`

Expression evaluation occurs with respect to a context (the current node in the XML tree a processor is looking at). An axis specifier such as child or descendant specifies the direction to navigate from the context node and selects an initial node-set. The node test and the predicate are used to filter the node-set by specifying desired element names, type of nodes or by defining some properties a node must have.

- **Axis** – indicates navigation direction within the tree representation of an XML document and defines a node-set relative to the current node. There are 13 axes available (see Figure 2.2):
  - `ancestor` selects all ancestors (parent, grandparent, etc.) of the current node,
  - `ancestor-or-self` selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself,
  - `attribute` selects all attributes of the current node,
  - `child` selects all direct descendants (children) of the current node,
  - `descendant` selects all descendants (children, grandchildren, etc.) of the current node,
  - `descendant-or-self` selects all descendants (children, grandchildren, etc.) of the current node and the current node itself,
  - `following` selects everything in the document after the closing tag of the current node,
  - `following-sibling` selects all siblings after the current node,
  - `namespace` selects all namespace nodes of the current node,
  - `parent` selects the parent of the current node,
  - `preceding` selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes,
  - `preceding-sibling` – selects all siblings before the current node,
  - `self` selects the current node.

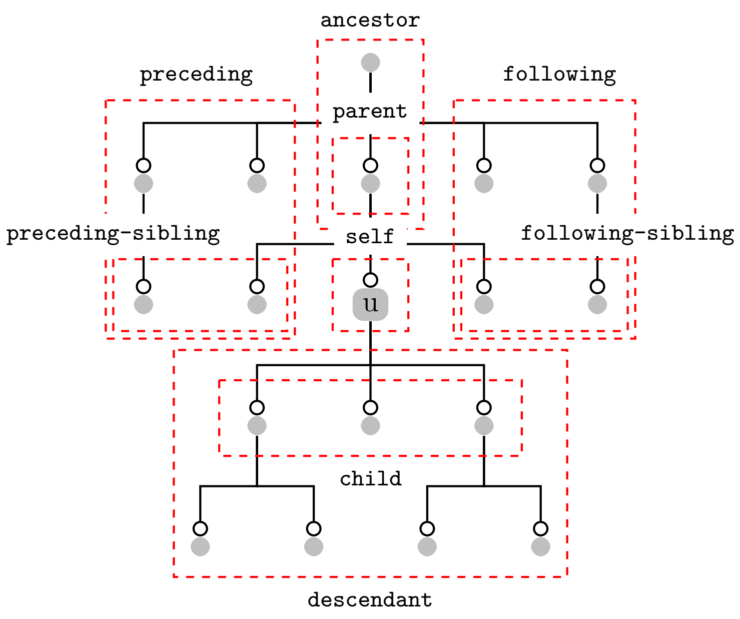


Figure 2.2. XPath axes with respect to a context node u

- **Node test** may consists of specific node names or more general expressions such as:
  - `*` matches any element node,
  - `@*` matches any attribute node,
  - `node()` matches any node of any kind,
  - `text()` matches any text node,
  - `comment()` matches any comment node.
- **Predicate** is an expression written in square brackets, that can be used to find specific nodes or nodes that contain a specific value. There is no limit to the number of predicates in a step, and they need not be confined to the last step in an XPath. They can also be nested to any depth.

The XPath syntax comes in two flavours: the abbreviated syntax, is more compact and allows XPaths to be written and read easily using intuitive and, in many cases, familiar symbols and constructs. The full syntax is more verbose, but allows for more options to be specified, and is more descriptive if read carefully. We list some abbreviated expressions in Table 2.1.

Abbreviated syntax	Full syntax
	`child::`
`@`	`attribute::`
`.`	`self::node()`
`..`	`parent::node()`
`//`	`/descendant-or-self::node()`
`//X`	`/descendant-or-self::node()/child::X`
`[number]`	`[position()=number]`

Table 2.1. XPath expressions in abbreviated and full syntax

2.4.2 Examples

The following examples of XPath expressions refer to the sample XML document $\mathcal{D}$ described in Figure 2.42 with matching XML tree model $\mathcal{T}$ illustrated in Figure 2.1.

Example 2.45. The following XPath expression selects `name` attributes for all `HOUSE` elements.

`//HOUSE/@name` ♠

Example 2.46. The following XPath expression selects all `LORD` elements having `HOUSES` element as an ancestor.

`/HOUSES//LORD` ♠

Example 2.47. The following XPath expression selects all *text* nodes of `SIGIL` elements having `HOUSE` element as parent and `SEAT` element as sibling.

`//HOUSE[SEAT]/SIGIL/text()` ♠

Example 2.48. The following XPath expression selects any node having an attribute.

`//*[@*]` ♠

Example 2.49. The following XPath expression selects all `HOUSE` elements with at least one `SEAT` child element.

`//HOUSE[count(SEAT)>=1]` ♠

Chapter 3
Arbology and XPath Relation

Arbology [6] (from the spanish word *arbol*, meaning tree) is a new algorithmic discipline interested in processing tree data structures. It solves problems such as tree pattern matching, tree indexing or finding repeats in a tree. The internal structure of an XML document can be represented as a tree in natural way and an XPath query can be viewed as a kind of a tree pattern. Thus, we can declare querying or searching XML document using XPath to be analogous problems of a tree pattern matching.

A tree pattern is a tree whose leaves can be labelled by a special symbol S, which servers as a placeholder for any subtree. It holds that a tree pattern in a linear notation corresponds to a substring of the linear notation of the tree, where the symbols S are replaced with linear notations of subtrees. We propose XPath queries to be a new kind of a tree pattern that differs, for example, in following:

- Although an XPath query can have a fairly complex structure, it is often oriented to a few target nodes only.

  Example 3.1. Consider the XML tree model $\mathcal{T}$ in Figure 2.1 and a sample XPath query Q_1 = `/HOUSES/HOUSE/SIGIL` illustrated in Figure 3.1. The query expects occurrences of nodes `SIGIL` that satisfy the nesting property as the answer. Therefore, the occurrences of elements `HOUSES` and `HOUSE` are irrelevant. In a query tree model we show a target node in red. We illustrate the occurrences of Q_1 in Figure 3.4. ♠

- An occurrence of a tree pattern in a tree always forms a connected graph. However, an XPath query might allow nodes skipping and thus, the final mapping of nodes can result in a disconnected graph.

  Example 3.2. Consider the XML tree model $\mathcal{T}$ in Figure 2.1 and a sample XPath query Q_2 = `/HOUSES//LORD` illustrated in Figure 3.2. There are two occurrences of Q_2 in $\mathcal{T}$ and both of them form a disconnected graph, since the descendant edge maps to a path of length two. The occurrences are illustrated in Figure 3.5. In a query tree model we show the `//` axis using a twisted line. ♠

- Considering types of nodes, XML documents are usually heterogeneous structures. Hence, XPath query is able to distinguish such nodes using special constructs.

  Example 3.3. Consider the XML tree model $\mathcal{T}$ in Figure 2.1 and a sample XPath query Q_3 = `//SIGIL//text()` illustrated in Figure 3.3. The query selects only text nodes that have a `SIGIL` element as an ancestor. There are two occurrences of Q_3 in $\mathcal{T}$ shown in Figure 3.6. ♠

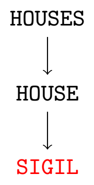


Figure 3.1. Query Q_1

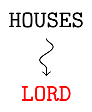


Figure 3.2. Query Q_2

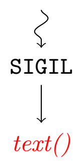


Figure 3.3. Query Q_3

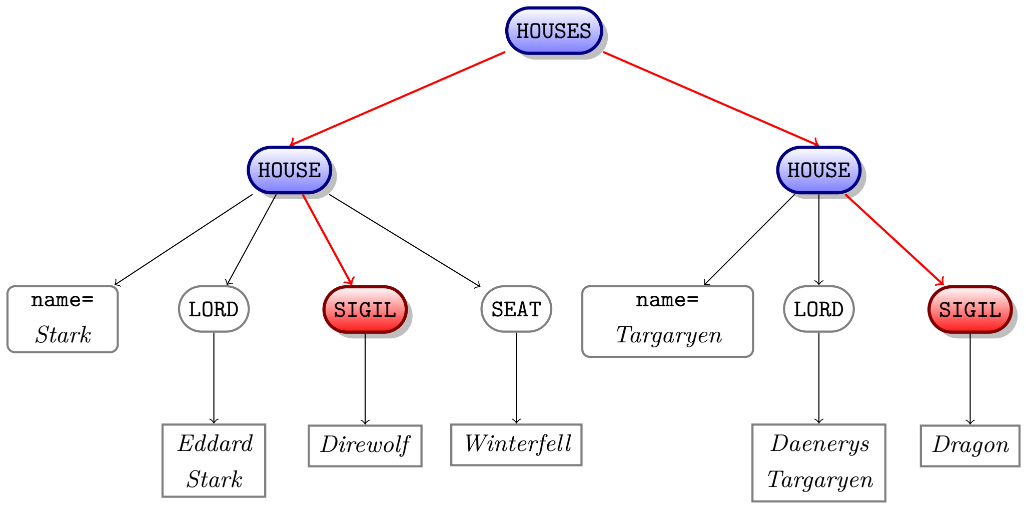


Figure 3.4. Occurrences of the XPath query $Q_1 =$ `/HOUSES/HOUSE/SIGIL`

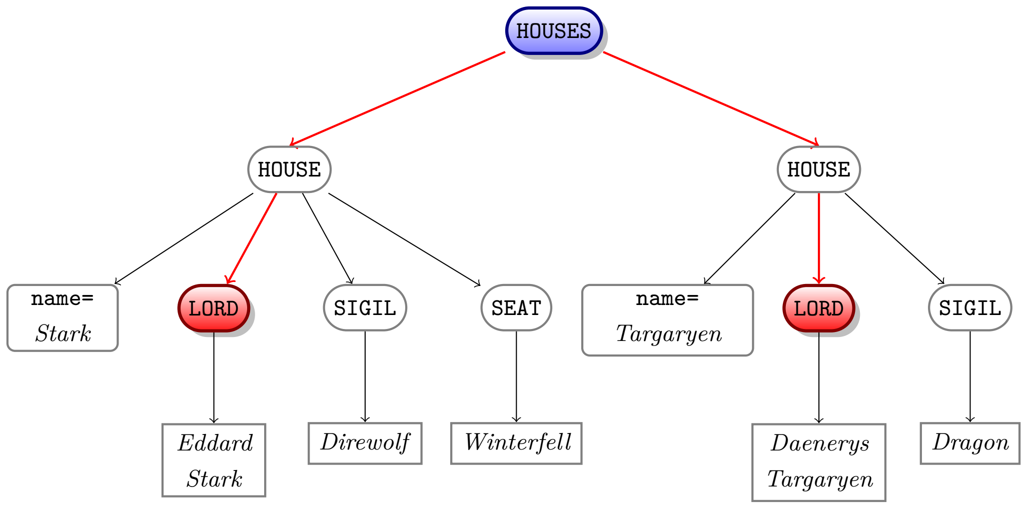


Figure 3.5. Occurrences of the XPath query $Q_2 =$ `/HOUSES//LORD`

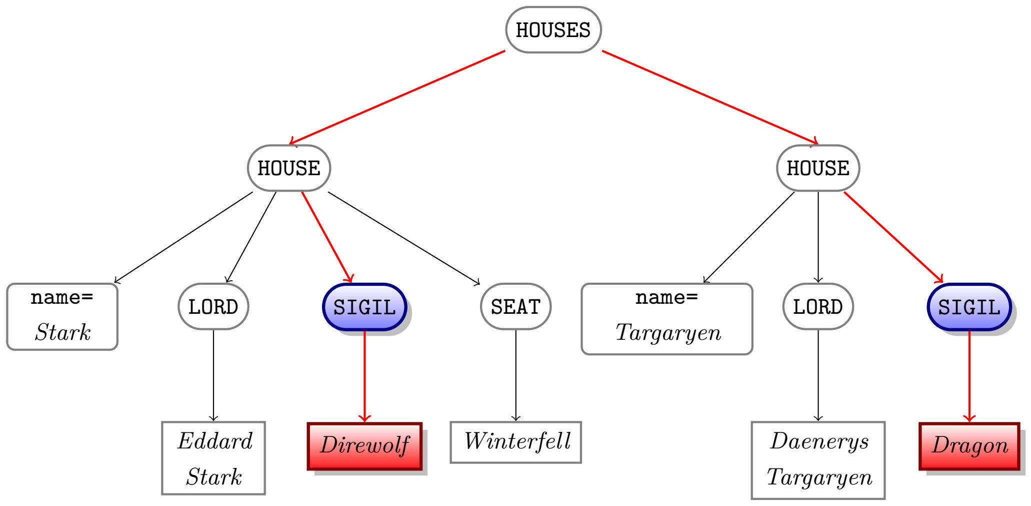


Figure 3.6. Occurrences of the XPath query $Q_3 =$ `//SIGIL//text()`

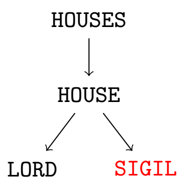


Figure 3.7. Query Q_4

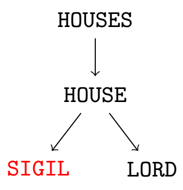


Figure 3.8. Query Q_4

- In general, an XPath query that contains a branching structure (e.g., predicate) is viewed as an unordered tree.

  Example 3.4. Consider the XML tree model $\mathcal{T}$ in Figure 2.1 and a sample XPath query Q_4 = `/HOUSES/HOUSE[LORD]/SIGIL`. Since, this XPath expression has no restriction on the order of the sibling nodes `LORD` and `SIGIL`, we need to consider both arrangements illustrated in Figures 3.7 and 3.8. We have also no guarantee that there are no other child elements of `HOUSE` element "between" a pair of `LORD` and `SIGIL` or `SIGIL` and `LORD` elements. ♠

Speaking in general, it does not hold that an XPath query tree model in a linear notation corresponds to a substring of the linear notation of the XML tree model. We can rather think about it as its subsequence. However, more complex XPath queries containing axes such as: `parent`, `ancestor` or `preceding` are even difficult to represent by tree models.

3.1 Classification of XPath Queries

Building a universal data structure that is able to efficiently evaluate all the constructs of XPath query language is really not a simple task. Therefore, structures supporting just fragments of XPath are the most common subjects of study. To simplify description of such fragments we propose a classification of XPath queries as illustrated in Figure 3.9.

3.1.1 Query Constructs

We can simply classify XPath queries, by restricting the constructs c (i.e., $*$, $//$) available in the query syntax. We denote such class of queries by $XP^{\{c_1,c_2,\ldots,c_n\}}$.

Example 3.5. Consider a fragment of XPath which consists of: node name tests, child axes, descendant axes and wildcards. Isolating the key features of XPath we get $XP^{\{/,//,*,node-name\}}$. ♠

3.1.2 Query Orientation

We may also group queries according to the orientation used for navigating through the XML tree structure. Therefore, we get following main categories: down, up, left and right. Other queries can be denoted by their combination, such as down-right oriented queries.

- **Down** orientated queries navigate from the context node only downwards using following axes:
  - `child::`,
  - `descendant::`,
  - `descendant-or-self::`.

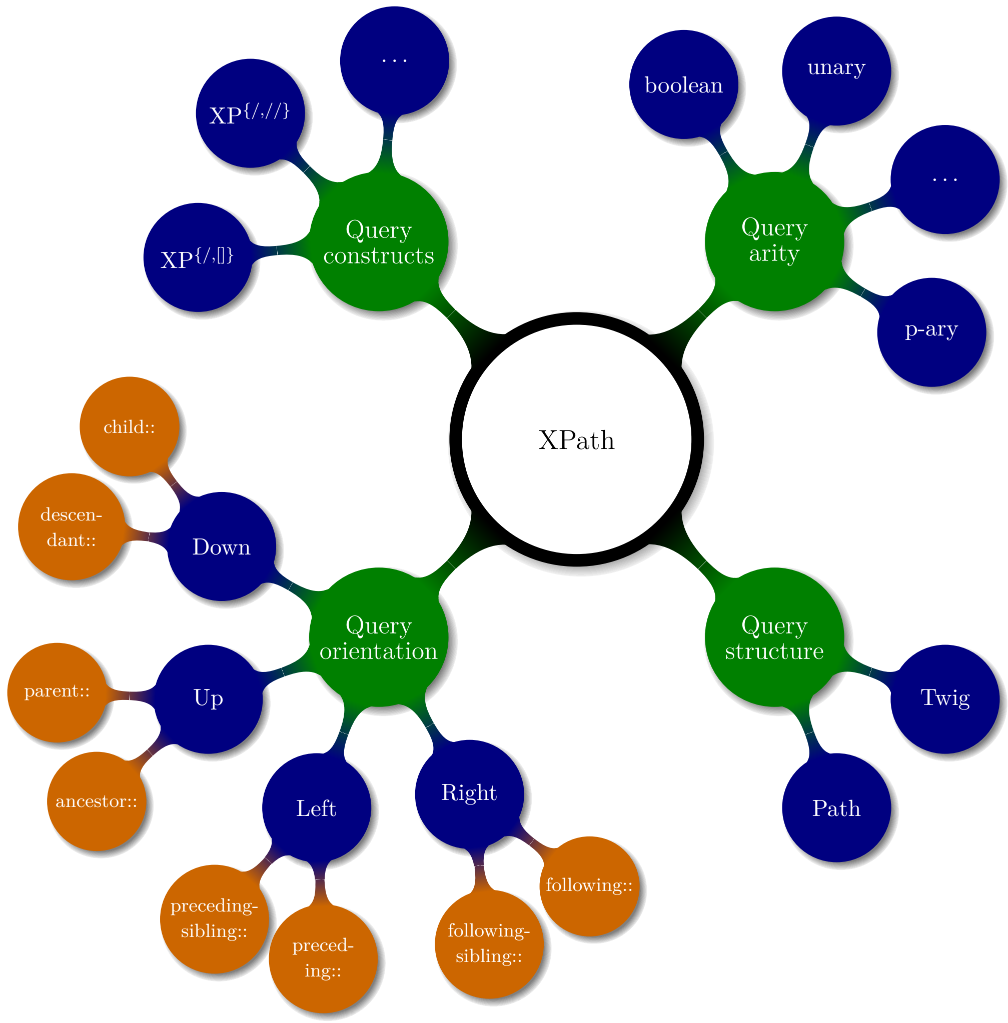


Figure 3.9. XPath queries classification

- **Up** orientated queries navigate from the context node only upwards using following axes:
  - `parent::`,
  - `ancestor::`,
  - `ancestor-or-self::`.
- **Left** orientated queries navigate from the context node only leftwards using following axes:
  - `preceding::`,
  - `preceding-sibling::`.
- **Right** orientated queries navigate from the context node only rightwards using following axes:
  - `following::`,
  - `following-sibling::`.

Example 3.6. Consider a fragment of XPath which consists of child (/) and descendant-or-self (//) axes only. We classify such queries as down oriented. ♠

3.1.3 Query Structure

Considering the query tree structure, there are two elementary classes:

- **Path** queries are XPath expressions that might be represented by a simple linear tree model (i.e., no branching).
- **Twig** queries are XPath expressions that might by represented by a tree model with at least two leaves (i.e., single branching).

Example 3.7. Consider queries Q_1, Q_2 and Q_3 in Figures 3.1, 3.2 and 3.3, respectively and. We classify all of them as path queries. The query Q_4 illustrated in Figure 3.7 is an example of a twig query. ♠

3.1.4 Query Arity

Every XPath expression is oriented to k target nodes only, for some $k \geq 0$. We propose k to represent a query arity. Queries of arity $0, 1, 2, \dots, p$ are respectively called boolean, unary, binary, $\dots$, p-ary queries.

Example 3.8. Let $Q_5 =$ `//HOUSE/*[self::LORD or self::SEAT]` The query selects all `LORD` and `SEAT` elements that are children of `HOUSE` element. Therefore, Q_5 represents a binary query. ♠

Chapter 4
Indexing Texts and Trees

4.1 Text Indexing

Text or string data naturally arises in many contexts including document processing, information retrieval, natural and computer language processing, and describing molecular sequences. In broad terms, the goal of text indexing is to design methodologies to store text data, so that performance of answering queries [12] is improved. The most classic task in text processing is to find all occurrences of a pattern in a given text.

The theory of text indexing, which is a result of stringology research [13–15], is very well-researched and uses many sophisticated data structures, such as: suffix trie, suffix tree, suffix or factor automaton and subsequence automaton. The compacted and minimized version of both suffix trees and suffix automata is represented by compact suffix automaton. Recently, another text indexing structure, called position heap, has been proposed in [16]. This chapter briefly introduces these existing data structures of text indexing and summarizes their ability.

4.1.1 Data Structures for Storing the Suffixes

Definition 4.1. (Set of all suffixes). Given a string x, its corresponding set $Suff(x)$, called the *set of all suffixes* of the string x, is defined as follows:

$$Suff(x) = \{y : x = wy,\ w, x, y \in \mathcal{A}^*\} \qquad \clubsuit$$

Data structures for storing the suffixes of a text are conceived for providing a direct and fast access to the factors of the text. As the basic models of computation, standard finite automata, trees and their compact versions, are used. The representation of the suffixes of a string by a trie has the advantage to be simple but lead to a quadratic memory space according to the length of the considered string. The (compact) suffix tree avoids this drawback and admits a linear space implementation. The minimization (in the sense of automata) of the suffix trie gives the minimal suffix automaton. Compaction and minimization together give the compact suffix automaton. [13] Let us summarize these data structures in the list below:

- **Suffix trie** [13, 17] of a string x is a deterministic automaton that recognizes $Suff(x)$. The underlying graph structure of the automaton is a tree whose edges are labelled by symbols of given alphabet.
  - number of states: $\mathcal{O}(|x|^2)$.
- **Suffix tree** [13, 17] of a string x is a compact trie accepting $Suff(x)$ obtained from the suffix trie of a string x by deleting all nodes having outdegree 1 that are not terminal. Edges are labelled by subwords of x instead of symbols.
  - number of states: between $(|x| + 1)$ and $2|x|$.

- **Suffix automaton** [13, 17–18] (DAWG – Directed Acyclic Word Graph) is a minimal deterministic automaton accepting $Suff(x)$ obtained from the suffix trie of x by minimization.
  - number of states: between $(|x| + 1)$ and $(2|x| - 1)$,
  - number of transitions: between $|x|$ and $(3|x| - 4)$.
- **Compact suffix automaton** [13, 17] is a compact minimal automaton accepting $Suff(x)$ obtained either from the suffix tree of x by minimization or from the suffix automaton of x by compaction.
  - number of states: between 2 and $(|x| + 1)$,
  - number of transitions: less or equal to $2(|x| - 1)$.

Example 4.2. Consider string $x = baaba$ over alphabet $\mathcal{A} = \{a, b\}$ and its corresponding set $Suff(x) = \{baaba, aaba, aba, ba, a, \varepsilon\}$. The data structures accepting $Suff(x) \setminus \{\varepsilon\}$ and the relationships between them, as described above, are illustrated in Figure 4.1. ♠

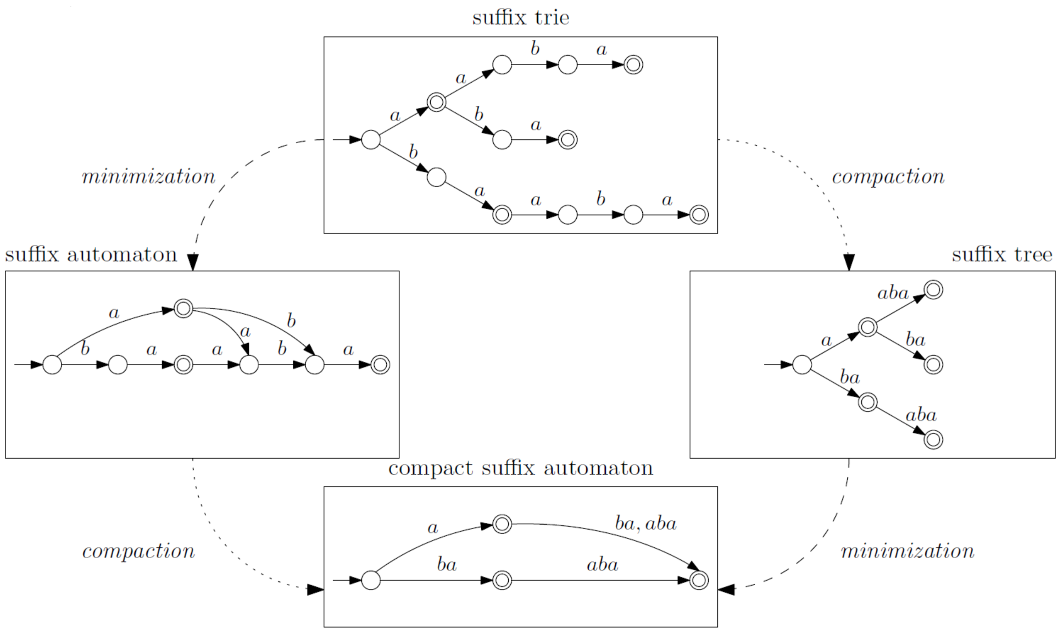


Figure 4.1. Efficient data structures accepting the set $Suff(baaba) \setminus \{\varepsilon\}$ and the relationships between them

- **Position heap** [19] is patterned on a data structure, called a *sequence hash tree*[1]) that adapts to the problem of finding occurrences of a pattern string P (length m) in a string x (length n). The position heap of a string x is obtained by iteratively inserting the suffixes $(x_1, x_2, \ldots, x_n)$ of x, in ascending order of length into the sequence hash tree using following insertion operation: x_i is inserted by creating a new node that is the shortest prefix of x_i that is not already a node of the tree, and labelling it with position i.
  - trivial construction: $\mathcal{O}(nh(T))$, where $h(T)$ is the length of the longest substring y of x that is repeated at least $|y|$ times in x,

[1]) Sequence hash tree is a data structure of Coffman and Eve [20], originally designed for the problem of implementing hash tables whose keys are strings.

- optimized construction: $\mathcal{O}(n)$.

The trivial query algorithm for finding all occurrences of a pattern string P in x consists of following steps:

1. Index into the position heap to find the longest prefix y of P that is a node of the position heap. For each ancestor y' of y (including y), look up the position i stored in y'. Determine whether this occurrence is followed by $P \setminus y'$. If it is, report i as an occurrence of P.
2. If $y = P$, also report all positions stored at descendants of y.

- trivial query algorithms: $\mathcal{O}(min(m^2, mh(T)) + occ)$, where occ is the number of occurrences of pattern string P in the string x,
- optimized query algorithm: $\mathcal{O}(m + occ)$.

Example 4.3. Consider string $x = abaababbabbab$ over an alphabet $\mathcal{A} = \{a, b\}$ and the set $Suff(x) = \{b, ab, bab, bbab, abbab, babbab, bbabbab, abbabbab, babbabbab, ababbabbab, aababbabbab, baababbabbab, abaababbabbab, \varepsilon\}$. The corresponding position heap is illustrated in Figure 4.2. ♠

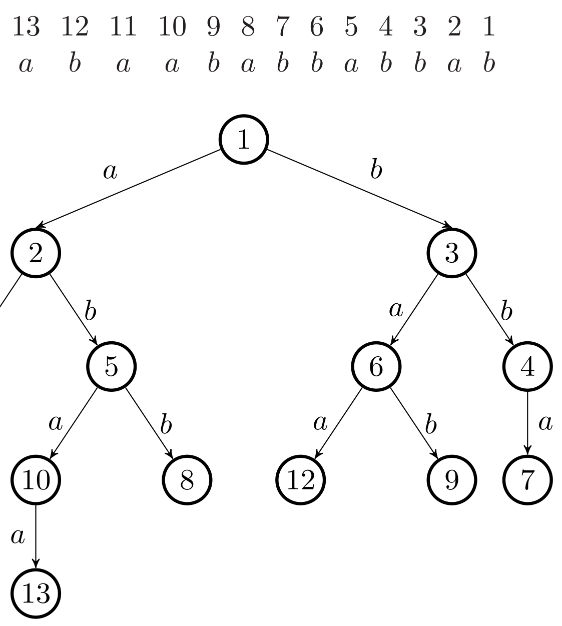


Figure 4.2. Position heap of the string $x = abaababbabbab$ from Example 4.3

4.1.2 Factor Automaton

Definition 4.4. (Set of all factors). Given a string x, its corresponding set $Fact(x)$, called the *set of all factors* (substrings) of the string x, is defined as follows:

$$Fact(x) = \{y : x = wyz, \; w, z, x, y \in \mathcal{A}^*\}$$ ♣

A *factor automaton* [18], in some sources called Directed Acyclic Word Graph (DAWG), is the minimal deterministic automaton that recognizes the set $Fact(x)$.

Example 4.5. Consider string $x = baaba$ over alphabet $\mathcal{A} = \{a, b\}$ and its corresponding set $Fact(x) = \{\varepsilon, a, b, ab, ba, aa, aab, aba, baa, aaba, baab, baaba\}$. The nondeterministic automaton accepting the set of all factors (substrings) is illustrated in Figure 4.3. ♠

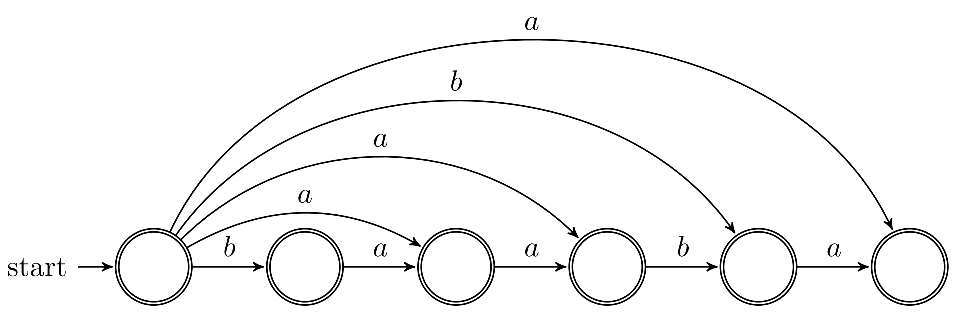


Figure 4.3. Nondeterministic factor automaton accepting $Fact(baaba)$ from Example 4.5

4.1.3 Subsequence Automaton

Definition 4.6. (Set of all subsequences). Given a string x, its corresponding set $Sub(x)$, called the set of all subsequences of the string x, is defined as follows:

$Sub(x) = \{a_1a_2 \dots a_m : x = y_0a_1y_1a_2 \dots a_my_m,\ y_i \in \mathcal{A}^*,\ i = 0, 1, 2, \dots m,\ a_j \in \mathcal{A},\ j = 1, 2, 3, \dots m,\ m \geq 0\}$ ♣

A *subsequence automaton* of a string x [18] is the minimal deterministic automaton that recognizes the set $Sub(x)$.

Example 4.7. Consider string $x = abba$ over alphabet $\mathcal{A} = \{a, b\}$ and its corresponding set $Sub(x) = \{\varepsilon, a, b, ab, ba, aa, bb, aba, bba, abb, abba\}$. The nondeterministic automaton (with ε-transitions) accepting the set of all subsequences is illustrated in Figure 4.4. After elimination of ε-transitions the nondeterministic subsequence automaton has transition diagram as described in Figure 4.5. ♠

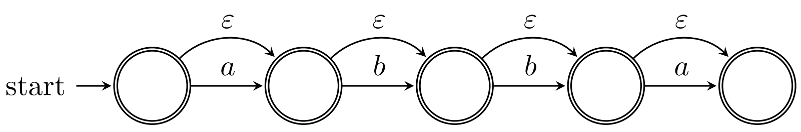


Figure 4.4. Nondeterministic subsequence automaton with ε-transitions accepting the set $Sub(abba)$ from Example 4.7

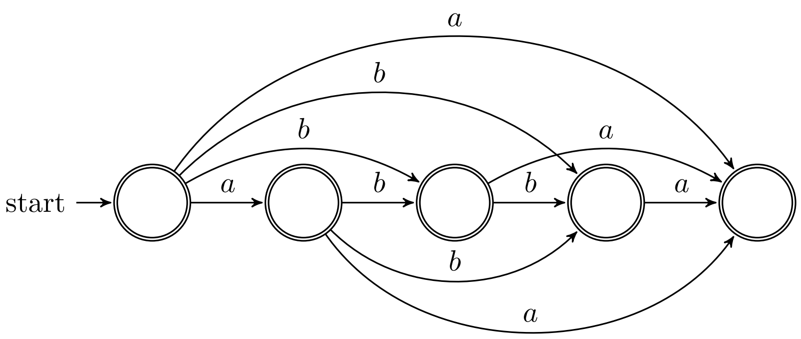


Figure 4.5. Nondeterministic subsequence automaton after ε-transitions removal accepting the set $Sub(abba)$ from Example 4.7

4.2 Tree Indexing

Subtree matching and tree pattern matching are often declared to be analogous problems of string pattern matching. There is a key property of linear notations of trees:

"*The linear notation of a subtree is a substring of the linear notatation of the tree.*" [6]

As a result of arbology research [21, 6], the algorithmic discipline interested in processing tree data structures, some sophisticated solutions of tree pattern matching and tree indexing have been proposed. For the algorithms a standard pushdown automaton is used as the basic model of computation.

We briefly present two kinds of acyclic pushdown automata for trees in prefix notation in this section. First, *subtree pushdown automaton* accepts all subtrees of the tree. Second, *tree pattern pushdown automaton* accepts all tree patterns which match the tree. The presented pushdown automata can be determinised. Finally a *full and linear index of a tree for tree patterns* consisting of a standard string compact suffix automaton and a subtree jump table is introduced.

4.2.1 Subtree Pushdown Automaton

A subtree pushdown automaton [21] for a tree recognizes all of its subtrees. The formal definition follows:

Definition 4.8. (Subtree pushdown automaton). Let T and $pref(T)$ be a tree and its prefix notation, respectively. A subtree pushdown automaton for $pref(T)$ accepts all subtrees of T in corresponding prefix notation. ♣

The deterministic subtree pushdown automaton for a tree T can have just states and transitions which correspond to states and transitions of the deterministic suffix automaton constructed for $pref(T)$, where the transitions of the subtree pushdown automaton are extended with pushdown operations.

The pushdown operations compute the arity checksum (see Definition 2.30). For every substring w of $pref(T)$, it holds that w is the prefix notation of a subtree of T, if and only if $ac(w) = 0$, and $ac(w_i) \geq 1$ for each proper prefix w_i of w. The construction of the deterministic subtree pushdown automaton for trees in prefix notation consists of following steps:

1. Construction of a nondeterministic subtree pushdown automaton as described by Algorithm 8.5 in [21],
2. Determinisation of the nondeterministic subtree pushdown automaton constructed in previous step.

For a tree T with n nodes, the constructed deterministic pushdown automaton has just one pushdown symbol, N states and E transitions, where $N \leq 2n + 1$ and $E \leq N + n - 1 \leq 3n$.

Example 4.9. Consider a tree T illustrated in Figure 4.6 and its prefix notation $pref(T) = a_2a_1a_0a_1a_0$. The corresponding nondeterministic pushdown automaton $\mathcal{M}(T)$ accepts $pref(T)$ by empty pushdown store with S as an initial pushdown symbol. The transition diagram of $\mathcal{M}(T)$ is shown in Figure 4.7. In this figure, for each transition rule $\delta(p, a, \alpha) = (q, \beta)$ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \to \beta$. ♠

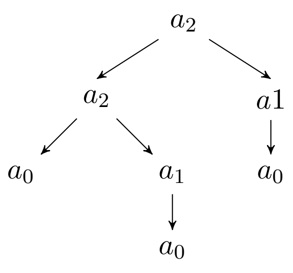


Figure 4.6. Ranked tree T for Examples 4.9, 4.11 and 4.13

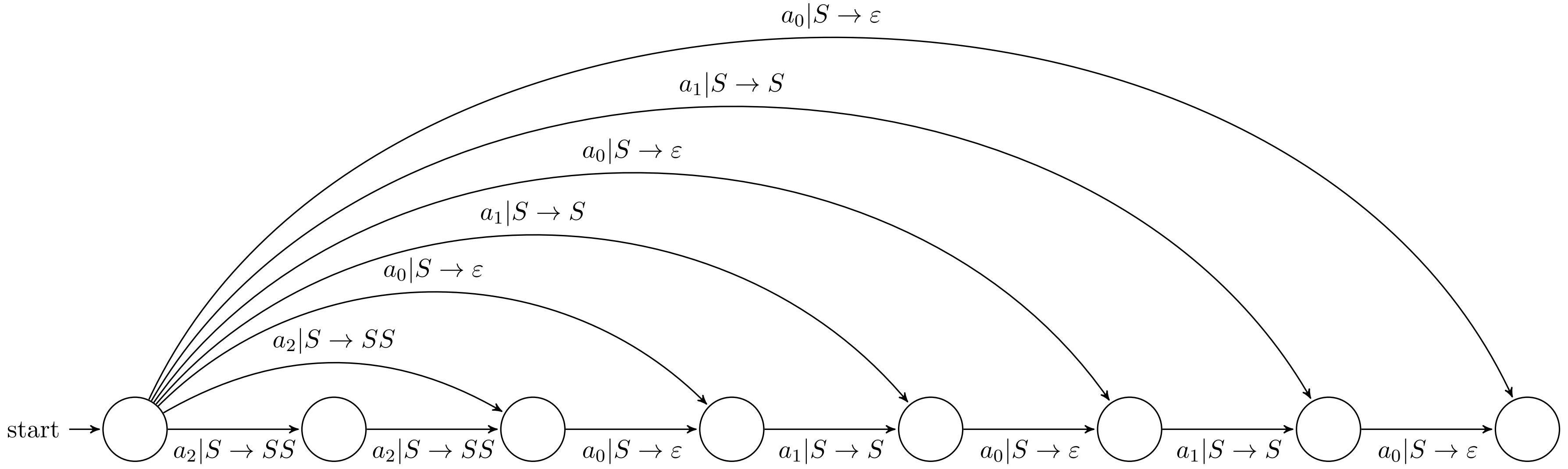


Figure 4.7. Transition diagram of nondeterministic subtree pushdown automaton $\mathcal{M}$ for tree T from Example 4.9

4.2.2 Tree Pattern Pushdown Automaton

Tree pattern pushdown automaton [21] is an extension of subtree pushdown automaton, so that also tree templates (see Definition 2.28) would be accepted. The formal definition follows:

Definition 4.10. Let T and $pref(T)$ be a tree and its prefix notation, respectively. A tree pattern pushdown automaton for $pref(T)$ accepts all tree patterns (see Definition 2.27) in prefix notation which match the tree T. ♣

New states and transitions, which are used for processing the special nullary symbols S in tree templates, are additionaly present in the tree patterns. The pushdown operations are the same and compute arity checksum. The construction of the deterministic tree pattern pushdown automaton for trees in prefix notation consists of following steps:

1. Construction of a deterministic pusdown automaton, that accepts all tree patterns that match the subject tree and contain the root of the subject tree, as described by Algorithm 8.13 in [21].
2. Extension of the deterministic pushdown automaton from previous step, so that it accepts also all subtrees of tree T. The resulting automaton is nondeterministic in general.
3. Determinisation of the nondeterministic tree pattern pushdown automaton constructed in previous step.

Given a tree T with n nodes, the number of distinct tree patterns which match the tree T can be at most $2^{n-1} + n$. For a special kind of trees (called trees with periodical subtrees[1])), the deterministic tree pattern pushdown automaton has just one pushdown symbol, N states and E transitions, where $N \leq 2n + 1$ and $E \leq 2N + n - 3 \leq 5n - 1$. However, in general the deterministic tree pattern pushdown automaton can have more than linear number of states. [21]

Example 4.11. Consider a tree T illustrated in Figure 4.6 and its prefix notation $pref(T) = a_2a_1a_0a_1a_0$. The corresponding nondeterministic tree pattern pushdown automaton $\mathcal{M}(T)$ accepts all tree patterns in prefix notation which match the tree T by empty pushdown store with S as an initial pushdown symbol. The transition diagram of $\mathcal{M}(T)$ is described in Figure 4.8. Again, in this figure for each transition rule $\delta(p, a, \alpha) = (q, \beta)$ the transition leading from state p to state q is labelled by the triple of the form $a|\alpha \to \beta$. ♠

4.2.3 A Full and Linear Index of a Tree

Janoušek, Melichar, Polách, Poliak and Trávníček in [16] presented a full and linear index of a tree for tree patterns. Given a subject tree T wit n nodes, the tree is preprocessed and an index, which consists of a standard string compact suffix automaton and a subtree jump table, is constructed.

The number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, and the size of the index is $\mathcal{O}(n)$. The searching phase uses the index, reads an input tree pattern of size m and computes the list of positions of all occurrences of the pattern in the tree T. The index consists of two parts:

1. A compact suffix automaton for $pref(T)$, by which occurrences of all substrings of $pref(T)$ can be located. We note that not all substrings of $pref(T)$ are subtrees in the prefix notation.

[1]) for description of a tree with periodical subtrees see Definition 8.22 in [21]

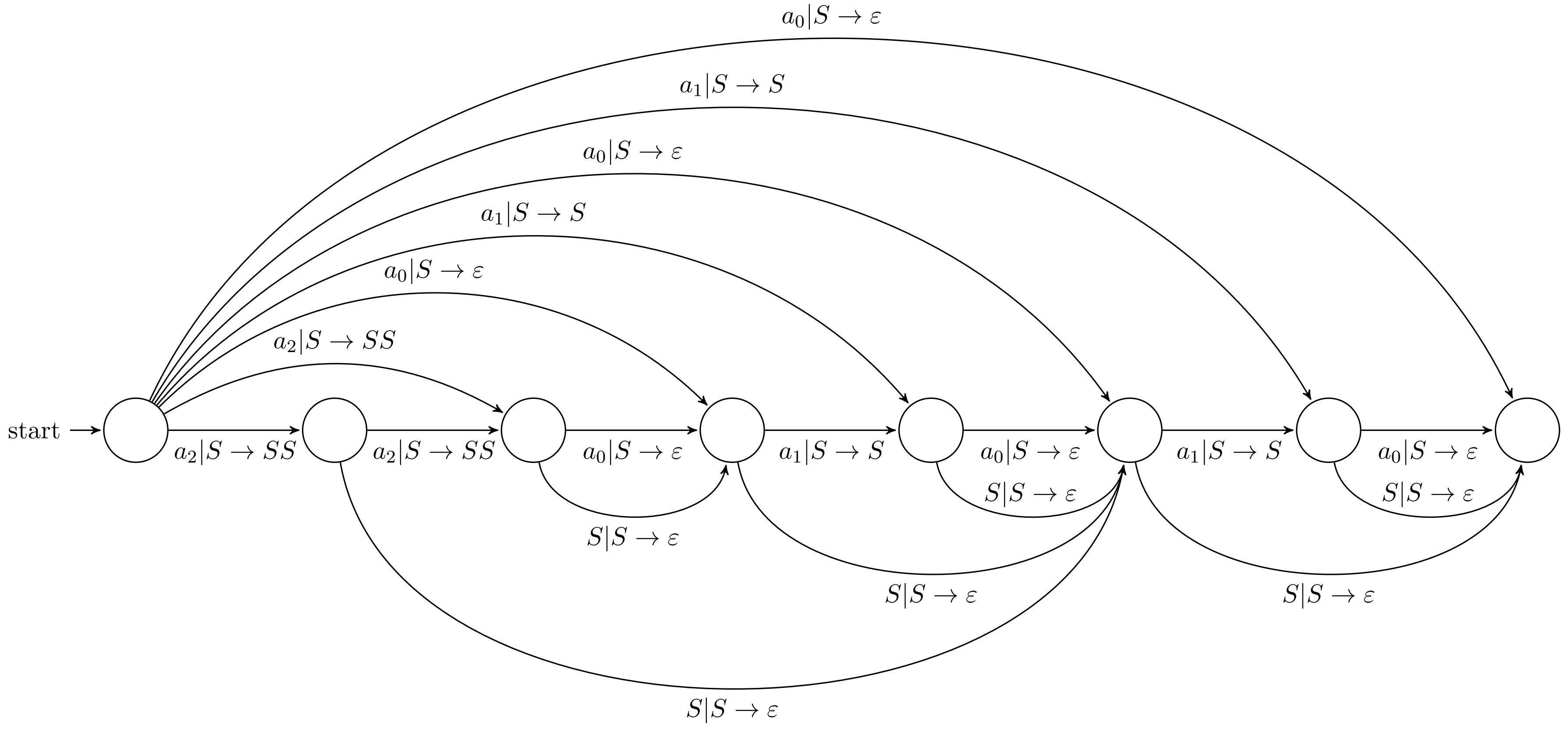


Figure 4.8. Transition diagram of nondeterministc tree pattern pushdown automaton $\mathcal{M}$ for T from Example 4.11

2. A subtree jump table, a linear-size structure needed for finding positions of ends of subtrees represented by special symbols S.

Definition 4.12. (Subtree jump table) Let T and $pref(T) = a_1 a_2 \dots a_n, n \geq 1$, be a tree and its prefix notation, respectively. A subtree jump table $SJT(T)$ is defined as a mapping from set $\{1, \dots, n\}$ into set $\{2, \dots, n+1\}$. If $a_i a_{i+1} \dots a_{j-1}$ is the prefix notation of a subtree of tree T, then $SJT(T)[i] = j$, $1 \leq i < j \leq n+1$. ♣

Example 4.13. Consider a tree T illustrated in Figure 4.6 and its prefix notation $pref(T) = a_2 a_1 a_0 a_1 a_0$. The corresponding subtree jump table and compact suffix automaton $\mathcal{M}(pref(T))$ are illustrated in Table 4.1 and Figure 4.9, respectively. ♠

The searching phase uses the index, reads an input pattern of size m and computes the list of positions of all occurrences of the pattern in the tree T. For an input tree pattern P in linear prefix notation $pref(P) = P_1 S P_2 S \dots S P_t$, $t \geq 1$, the searching is performed in time $\mathcal{O}(m + \sum_{i+1}^{t} |occ(P_i)|)$, where no substring P_i, $1 \leq i \leq t$, contains any symbol S and $occ(P_i)$ is the set of all occurrences of P_i in $pref(T)$.

i	1	2	3	4	5	6	7
$SJT[i]$	8	6	4	6	6	8	8

Table 4.1. Subtree jump table for tree T from Example 4.13

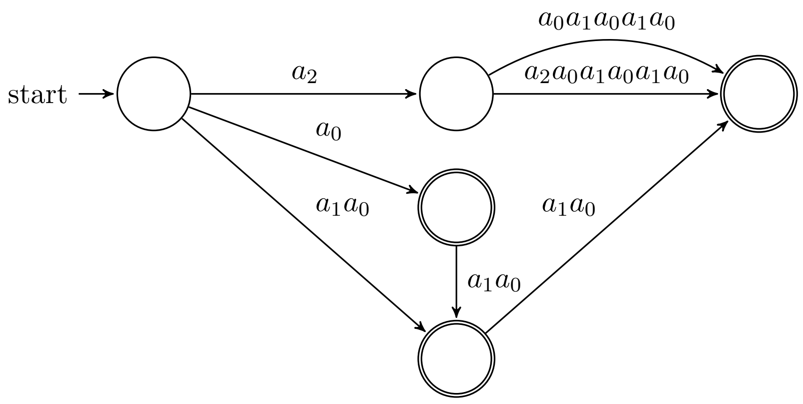


Figure 4.9. Compact suffix automaton for $pref(T)$ from Example 4.13

Chapter 5
Indexing XML Data

XML is used in many aspects of software development, often to simplify data storage and sharing. The simplest type of XML data storage is flat file storage, that is, the main entity is a complete document and the internal structure does not play a role. This provides a software- and hardware-independent way of storing data. Therefore, it reduces the complexity of data transport between incompatible systems, since XML documents can be read by different incompatible applications. [22]

To be able to retrieve the data from XML documents efficiently, various query languages, such as XPath [2], XPointer [3] and XLink [4], have been designed. These query languages use label paths to traverse the irregularly structured data. Without a structural summary, query processing can be quite inefficient due to an exhaustive traversal on XML data. Indexing the structure of XML data is an effective way to accelerate XML query processing, since it can greatly reduce the search space. Therefore, several XML documents indexes have been proposed in the research community. At present, the methods of XML indexing may be classified into following categories:

- **Graph-based** methods construct a structural path summary that can be used to improve query efficiency, especially for single path queries. To this category we can classify following methods: *DataGuides* [23], *1-Index* [24], *PP-Index* [25], *F&B-Index* [26], *MTree* [27] or *CTree* [28].
- **Sequence-based** methods transform both the source data and query into sequences. Therefore, querying XML data is equivalent to finding subsequence matches. To this category we can classify following methods: *ViST* [29], *PRIX* [30].
- **Node coding** methods apply certain coding strategy to design codes for each node, in order that the relationship among nodes can be evaluated by computation. To this category we can classify, for example, *XISS* [31] method.
- **Adaptive** methods can adapt their structure to suit the query workload. Therefore, adaptive methods index only the frequently used queries. To this category we can classify following methods: *APEX Index* [32], *AB-Index* [33].

Generally speaking, every method has its own advantages, however, shortcomings do exist: path-based methods often lack of support complex queries, sequence-based methods are likely to generate approximate solutions, thus requiring a great deal of validation, node coding method is very difficult to be applied to ever changing data source and adaptive methods perform low efficiency on non-frequent query. In this chapter, we will introduce some of the above index structures in detail.

5.1 MTree

Pettrovello and Fotouhi [27] proposed an XML structure index, called MTree, which is designed to be optimal for traversing all XPath axes. The primary feature of MTree lies in its ability to provide the next subtree root node in document order, for all axes, to each context node in $\mathcal{O}(1)$. The MTree works on the fact, that XPath partitions an XML document into four primary axes (i.e., ancestor, descendant, following and preceding). All of the remaining secondary axes can be algebraically derived from the four primary axes.

5.1.1 MTree Index Structure

MTree is a composite overlay of several graphs. It augments the XML tree data structure by introducing edges in acyclic XML graph that induces cycles corresponding to the ancestor, descendant, following and preceding primary axis semantics mandated by the XPath language specification. The axes graphs contain nodes that are roots of subtrees of vertices that belong to the respective axis set.

Axis paths are threaded reference chains of subtree root nodes connected in document order. When an axis path is traversed from a context node to the end of the path, all of the nodes under the subtree root nodes along the path belong to the requested axis set for that context node. XPath queries are solved by doing tree traversals on MTree. MTree is composed by the following graphs:

- ***f*-graph** is the set of *following* axis pointers for some context node.
- ***p*-graph** is the set of *preceding* axis pointers for some context node.
- ***a*-graph** is the set of *ancestor* axis pointers for some context node.
- ***d*-graph** is the set of *descendant* axis pointers for some context node. Since XML trees support multiple children, the descendant axis retains edges to only the first child for each node (to ensure the node structure is deterministic in size). Non first-children are derived by the first obtaining the first child reference, using the descendant axis, and then subsequently traversing the XPath following-sibling axis until the parent axis changes.
- ***q*-graph** is the set of directed paths of a sequence of vertices $v_1, v_2, \ldots, v_n$ linked in document order, having the same *element name* (qname). *q*-graph improves query processing by reducing the need to scan the entire tree when seeking specific element name. Each directed path is rooted in an $\mathcal{O}(1)$ lookup structure.
- ***attr*-graph** is the set of directed paths of a sequence of vertices $v_1, v_2, \ldots, v_n$ linked in document order, having the same *attribute name*. *attr*-graph improves query processing by reducing the need to scan the entire tree when seeking specific attribute name. Each directed path is rooted in an $\mathcal{O}(1)$ lookup structure.

Definition 5.1. (MTree). An XPath XML graph index *MTree* is a composite graph: MTree = *f*-graph ∪ *p*-graph ∪ *a*-graph ∪ *d*-graph ∪ *q*-graph ∪ *attr*-graph. ♣

Example 5.2. Consider the XML tree in Figure 2.1. The individual graphs *f*-graph, *p*-graph, *a*-graph, *d*-graph, *q*-graph and *attr*-graph are illustrated in Figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, respectively. The corresponding MTree (showing an overlay of previous graphs) is illustrated in Figure 5.7. ♠

5.1.2 Query Processing

XPath queries are managed by twig cursors that execute the location step query against the index for each node in the input sequence using tree traversal algorithms. Each location step query receives as input, a location step and a set of twig root nodes enqueued in tree order, and returns as output, a results set of unique twig root nodes in document order. The final set of twig root nodes obtained from the last location step query is used to constructs the results projection, by traversing the subtree vertices

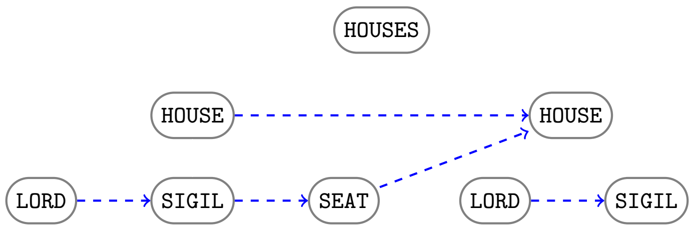


Figure 5.1. *f*-graph

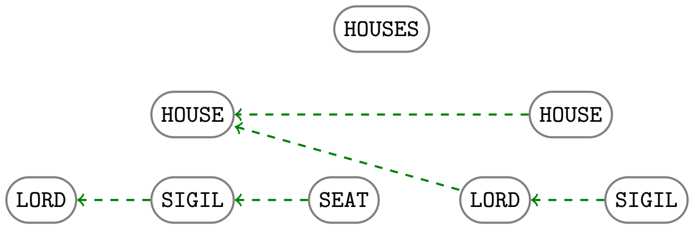


Figure 5.2. *p*-graph

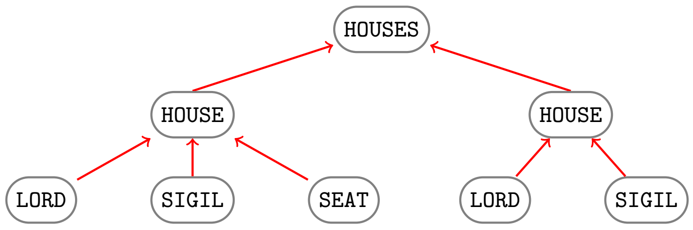


Figure 5.3. *a*-graph

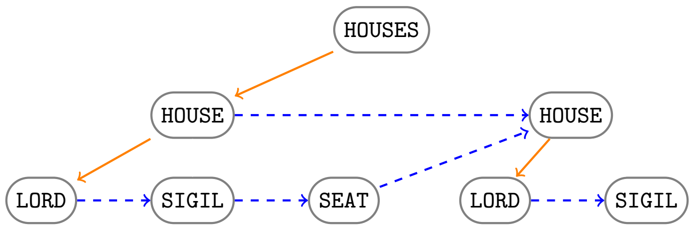


Figure 5.4. *d*-graph (orange) and *f*-graph (blue)

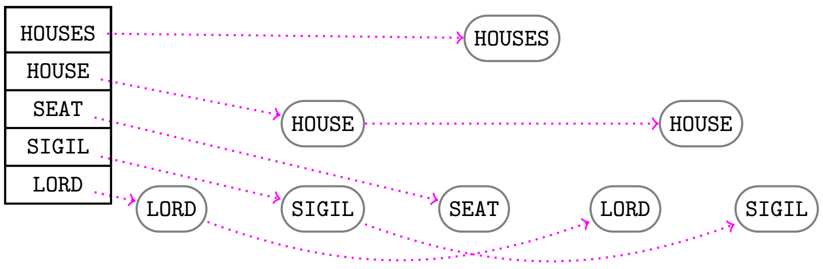


Figure 5.5. *q*-graph

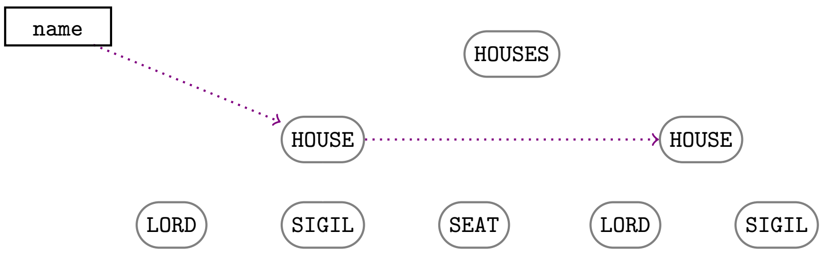


Figure 5.6. *attr*-graph

attached to the subtree root vertex using DFS[1]) tree traversal. By using only subtree root nodes in intermediate steps, where feasible to do so, significant performance improvements can be achieved.

Example 5.3. Let $\mathcal{D}$ and $\mathcal{T}$ be the sample XML document described in Example 2.42 and its XML tree model illustrated in Figure 2.1, respectively. Suppose we want to execute a query $Q =$ `//LORD/ancestor::*`.

The query begins by using q-graph to get `LORD` element nodes. The next step query follows the ancestor axis for both occurrences of `LORD` element. This will result in a set containing `HOUSES` element and both of `HOUSE` elements. ♠

Example 5.4. Let $\mathcal{D}$ and $\mathcal{T}$ be the sample XML document from Example 2.42 and its XML tree model illustrated in Figure 2.1, respectively. Consider a query $Q =$ `/HOUSES/HOUSE[1]/LORD/following::*`. The query begins by evaluating the first node in the path, node `HOUSES`. Since node `HOUSES` exists, its child axis is queried for the `HOUSE` element. This will result in the set containing both of `HOUSE` elements, however the predicate selects only the first one (in document order).

The next step, the child axis of `HOUSE` element is queried for `LORD` element. Finally, we are interested in following axis of `LORD` element. By traversing the f-path, the following axis query for context node `LORD` returns the set of f-subtree root vertexes: (first) `SIGIL` element, `SEAT` and (second) `HOUSE` element in three steps. Eventually we searched the subtrees under `SIGIL`, `SEAT` and `HOUSE` get the final answer containing both `SIGIL` elements, `SEAT` and (second) `LORD` and `HOUSE` elements. ♠

5.2 CTree

Zou, Liu and Chu in [28] proposed a compact tree (CTree) for XML indexing, which provides not only concise path summaries at the group level but also detailed child–parent links at the element level. Group level mapping allows efficient pruning of a large search space while element level mapping provides fast access to the parent of an element. Therefore, CTree is efficient for processing both single-path and branching queries with various value predicates.

5.2.1 CTree Index Structure

CTree is a two-level tree which provides a concise structure summary of the XML document at its group level and detailed child–parent links at its element level which can provide fast access to elements' parents. CTree also uses group-based element reference instead of standard global element IDs. First, we introduce the definitions of a label path and equivalent nodes, which are useful for describing a path summary and a CTree.

Definition 5.5. (Label path). Let $\mathcal{T}$ be an XML tree model. A *label path* for a node $\mathfrak{n}$ in $\mathcal{T}$, denoted by $L(\mathfrak{n})$, is a sequence of dot-separated labels of the nodes on the path from root to node $\mathfrak{n}$. ♣

Example 5.6. Consider the XML tree in Figure 2.1. Node `SEAT` can be reached from the root node `HOUSES` through path: `HOUSES` $\rightarrow$ `HOUSE` $\rightarrow$ `SEAT`. Therefore the label path for the `SEAT` node is `HOUSES.HOUSE.SEAT`. ♠

Definition 5.7. (Equivalent nodes). Nodes $\mathfrak{n}_1$ and $\mathfrak{n}_2$ in an XML tree model $\mathcal{T}$ are *equivalent* if they have the same label path. ♣

[1]) Depth-first search (DFS) is an algorithm for traversing tree data structures. One starts at the root and explores as far as possible along each branch before backtracking.

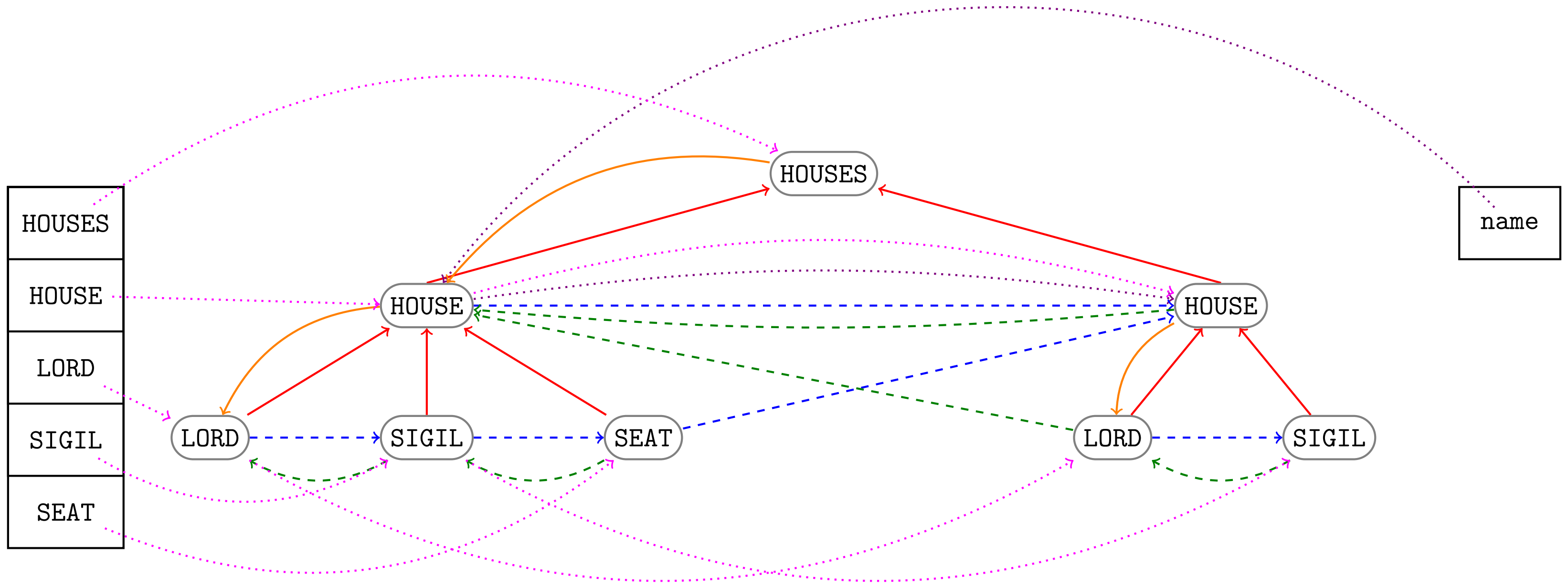


Figure 5.7. MTree for the XML tree model $\mathcal{T}$ illustrated in Figure 2.1 showing an overlay of f-graph, p-graph, a-graph, d-graph, q-graph and $attr$-graph

Example 5.8. The two `SIGIL` nodes in the XML tree model illustrated in Figure 2.1 are equivalent since their label paths are the same `HOUSES.HOUSE.SIGIL`. The parent nodes of `SIGIL` elements are `HOUSE` elements, which are equivalent (their label paths are `HOUSES.HOUSE`). Note, that parents of equivalents nodes are always equivalent as they also share the same label path. ♠

For an XML tree $\mathcal{T}$, a *path summary* is a tree on which each node is called *group* and corresponds to exactly one label in $\mathcal{T}$. The group contains all the equivalent nodes in $\mathcal{T}$ sharing the label path. We call a path summary an *ordered path summary* if the equivalent nodes in every group are sorted by their pre-order identifiers.

Example 5.9. An ordered path summary for the XML tree model in Figure 2.1 is shown in Figure 5.8. Each dotted box represents a group and the numbers in the box are the pre-order identifiers of equivalent data nodes. Each group has a label and an identifier listed below the group. For instance, nodes $3, 7$ are in group 2 since their label paths are the same: `HOUSES.HOUSE.LORD`. ♠

A path summary greatly facilities the evaluation of path queries. For example, consider a query $Q =$ `HOUSES/HOUSE/LORD`, the answers are nodes $3, 7$ because their label paths satisfy the query. However, the path summary does not preserve the hierarchical relationships among individual nodes. Therefore, the path summary is unable to answer twig queries.

Note, that in Figure 5.8, there is, for instance, no information to indicate which node in group 1 is the parent of node 3 in group 2. Such node-level relationships are important for answering twig queries. For instance, for a query $Q =$ `/HOUSES/HOUSE[SEAT]`, the path summary indicates that elements in group 1 are candidate answers but does not provide information about which elements in the group actually have `SEAT` sub-element.

Informally, CTree is a bi-level tree containing a *group level* and an *element level*. At the group level, CTree provides a summarized view of hierarchical structure of the XML document. At the element level, CTree preservers detailed child–parent links. Each group in CTree has an array mapping elements to their parents.

Definition 5.10. (CTree). *CTree* is a rooted tree where each node g, called *group*, contains an array of elements denoted as $g.pid[]$ such that:

- Each group g is associated with an *identifier* and a *name*, denoted by $g.id$ and $g.name$ respectively.
- Edge directions are from the root to the leaves. If there is an edge from g_1 to g_2, then g_1 is called *parent* of g_2 and g_2 is called *child* of g_1. If there is path from g_1 to g_3, then g_1 is called an *ancestor* of g_3 and g_3 is called a *descendant* of g_1.
- An array index e of $g.pid[]$ represents an *element* in g, denoted by $g : e$. The value of $g.pid[e]$ points to an element in g's parent g_p, and $g_p : g.pid[e]$ is called the *parent element* of $g : e$.
- For any two elements $g : e_1$ and $g : e_2$, if $e_1 < e_2$, then $g.pid[e_1] \leq g.pid[k_2]$. ♣

Definition 5.11. (Ordered CTree). An *ordered CTree* is CTree where siblings groups are ordered. ♣

Definition 5.12. (Absolute and relative element reference). For referring an element e in group g, $g : e$ is called an *absolute reference* and e is called a *relative reference.* ♣

Example 5.13. Consider the XML tree in Figure 2.1. The corresponding CTree is illustrated in Figure 5.9. Again as for the path summary, each dotted box represents a group and the numbers in the box are the pre-order identifiers of equivalent nodes.

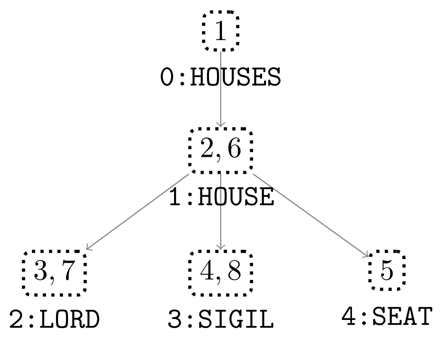


Figure 5.8. Ordered path summary

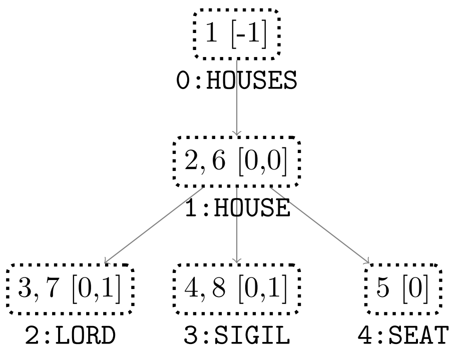


Figure 5.9. CTree

Moreover, there is an array in each group. The array contains the values which are relative reference for parent elements. The two elements in group 2 are referred to by absolute reference $2 : 0$ and $2 : 1$, whose values are 0 and 1 which are relative references for elements $1 : 0$ and $1 : 1$. ♠

5.2.2 CTree Properties

Definition 5.14. (Regular and irregular group). Let g be a group and g_c be one of its child groups. If every element in g has the same number of children in g_c, then g_c is called a *regular group*. Otherwise, g_c, is called an *irregular group*. ♣

The information about group regularity is useful for query optimization. For instance, for a query $Q =$ `//HOUSES[LORD and SIGIL]`, the CTree directly returns all elements in group 1 as answers without further checking the element-level links since groups 2 and 3 are regular groups. Note, that the array in a regular group can be even removed since the content of the array can be inferred from number of elements in groups.

Definition 5.15. (Monotonic property). The values of a group's array are arranged in increasing order. That is, if $i < j$, then $g.pid[i] \leq g.pid[j]$. ♣

This property enables us to use a binary search to locate the child elements of a given element.

5.2.3 Query Processing

To take full advantages of CTree characteristics a CTree-based query processing method has been proposed. We use a tree model to represent an XPath query Q. It is assumed that each query has only one target node, which is emphasized in red. After a query is transformed into a tree T_Q, it can be evaluated using CTree index $\mathcal{T}$ in following steps:

1. Evaluate group level structure constraints.
2. For each frame, do:

   (i) Evaluate value constraints on the frame,
   (ii) Evaluate element level structure constraints,
   (iii) Output the list of elements.

Example 5.16. Let $\mathcal{D}$ and $\mathcal{T}$ be the sample XML document from Example 2.42 and its XML tree model illustrated in Figure 2.1, respectively. Consider a query $Q_1 =$ `//HOUSE[LORD=`*Eddard Stark*`]/SIGIL`. Figure 5.10 illustrates a tree model representation of Q_1.

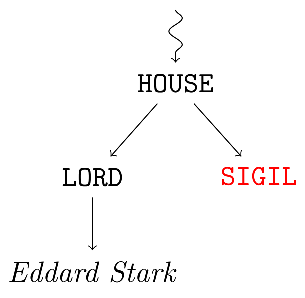


Figure 5.10. XPath query Q_1

First, it locates a set of frames matching Q's tree structure, where each frame is an assignment of CTree groups to the query nodes in Q that satisfy the structure of Q at the group-level. There is one frame consisting of groups (1, 2, 3) in the CTree (Figure 5.9) for Q_1, which match query nodes (`HOUSE`, `LORD`, `SIGIL`), respectively.

By assigning gid 1 to the query node `HOUSE`, we exclude the other elements which also have the tag name `HOUSE` (i.e., we would exclude potential group labelled `HOUSE` under a `SEAT` group) and thus reduce search space. The frames are found in a top-down fashion starting from candidate groups for the root of the query tree down to the leaves.

Second, for each frame, it evaluates value predicates. There is one value predicate in Q_1: `[LORD=`*Eddard Stark*`]`. We search group 2, since the query node `LORD` is mapped to group 2 in step 1, and element $2 : 0$ is returned (i.e., the first `LORD` element).

Finally, it evaluates element level structure constraints and returns the query results to the user. For the frame $(1, 2, 3)$ for Q_1, the second step of the query processor determines that element $\{2 : 0\}$ satisfies value constraint. Now the last step is to determine which elements in the target group 3 can answer Q_1. The answers can be determined by projecting relevant elements from other nodes to the target node.

The projecting direction for a query node can be either downward or upward depending on its position in the query tree. If a query node is an ancestor of the target node, its projecting direction is downward. Otherwise, it is upward (i.e., the projecting directions for `HOUSE` is downward, while for `LORD` is upward). Since CTree stores detailed child-to-parent relationships, upward projecting is straight forward. Downward projecting can be done similar to upward projecting by keeping track of children.

In this example we, at first, use upward projection of $\{2 : 0\}$ element getting $\{1 : 0\}$, followed by downward projection of $\{1 : 0\}$ to get the final answer $\{2 : 0\}$, (i.e., first `SIGIL` element). ♠

5.3 PP-Index

Nan Tang, Jeffrey Xu Yu, M. Tamer Ozsu, Kam-Fai Wong in [25] proposed a hierarchical index of an XML document, called PP-index and its compressed version, to process a fragment of XPath queries denoted by $XP^{\{/,//,*,name-test\}}$. The hierarchical index consists of index entries that are pairs of queries and their (full/partial) answers (called extends). With such an index, XPath queries can be processed to extract the results if they match the queries maintained in those index entries.

5.3.1 PP-Index Structure

The PP-Index has a set of entries of the form $(Q, X(Q))$, where Q is an XPath query, and $X(Q)$ is the extent (full/partial result) of Q. Let C be a class $XP^{\{/,//,*,name-test\}}$. We can split C into three following subsets:

- $C_c = XP^{\{/,*,name-test\}}$ represents queries with child (/) axis only,
- $C_d = XP^{\{//,*,name-test\}}$ represents queries with descendant-or-self (//) axis only,
- C_x includes the remaining queries.

There are two interrelated but different relationships among C queries to construct a hierarchical index for C queries. They are *prefix* and *path-containment*. The former provides a mechanism to find the requested entry for a C query. The latter provides a way to identify the requested extends to answer such a query.

Let $Q = \alpha_1 l_1 \alpha_2 l_2 \ldots \alpha_m l_m$ and $Q' = \alpha'_1 l'_1 \alpha'_2 l'_2 \ldots \alpha'_n l'_n$ be two C queries where $m < n$, α_i is an axis and l_i is a label.

Definition 5.17. (Prefix relationship). Q is a prefix of Q' iff $\alpha_i = \alpha'_i$ and $l_i = l'_i$ $(1 \leq i \leq m)$. Q is the maximal prefix of Q' iff Q is a prefix of Q' and $n = m + 1$ ♣

Example 5.18. Let $Q =$ `/HOUSES/HOUSE/LORD`, $Q' =$`/HOUSES` and $Q'' =$`/HOUSES/HOUSE`. Therefore, Q' is a prefix of Q and Q'' is a maximal prefix of Q. ♠

Definition 5.19. (Path-containment relationship). Q' is contained in Q, denoted as $Q' \sqsubseteq Q$, if the labels in Q match the labels in Q' in order, and the last labels match (i.e., $l_m = l'_n$). Furthermore, for two matched labels (e.g., l_i and l'_j), the corresponding axes have that: child axis / maps to child axis / (i.e., $\alpha'_j =$ / if $\alpha_i =$ /) and descendant-or-self axis // maps to rightward path (i.e., $\alpha_i =$ //). ♣

Example 5.20. `//LORD` $\sqsubseteq$ `//HOUSES//LORD` $\sqsubseteq$ `//HOUSES//HOUSE//LORD`. ♠

Definition 5.21. (1-Index). *1-Index* is a hierarchical index for C_c-queries using prefix relationship. In 1-Index, there is an edge from an entry $(Q_1, X(Q_1))$ to another entry $(Q_2, X(Q_2))$ if Q_1 is the maximal prefix of Q_2. ♣

Example 5.22. Consider the XML tree model in Figure 2.1. Corresponding 1-Index is shown in Figure 5.11 with five entries: `/HOUSES`, `/HOUSES/HOUSE`, `/HOUSES/HOUSE/LORD`, `/HOUSES/HOUSE/SIGIL`, `/HOUSES/HOUSE/SEAT`. Each entry Q maintains an extent $X(Q)$, indicated by $\langle\rangle$ containing the pre-order identifiers of nodes forming the extent. For instance, the extent of `/HOUSES/HOUSE` for the XML tree model has two tree nodes $2, 6$ (i.e., two `HOUSE` elements). ♠

Note, that *1-Index* defines very similar structure as the *Path summary* in Figure 5.8. However, we will continue with 1-Index terminology to preserve the definitions of following Containment index and PP-Index.

1-Index is able to support C_c queries by finding a corresponding entry and extracting its extend. Note that for two index entries, $(Q_1, X(Q_1))$ and $(Q_2, X(Q_2))$, in 1-Index, there is $X(Q_1) \cap X(Q_2) = \emptyset$ if $Q_1 \neq Q_2$. However, 1-Index cannot handle all C-queries. If we build an index for C-queries using prefix relationship, the number of entries is exponential (because of the combination of child and descendant-or-self axis) and the overlapping between two extends is high. Therefore, a large storage is required and the duplications require to be removed during query processing.

Definition 5.23. (Containment index). *Containment index* is a hierarchical index for C_d-queries. There is an edge from an entry $(Q_1, X(Q_1))$ to another entry $(Q_2, X(Q_2))$ if $Q_2 \sqsubseteq Q_1$ and there does not exist an entry $(Q_3, X(Q_3))$ where $Q_2 \sqsubseteq Q_3 \sqsubseteq Q_1$. ♣

Example 5.24. Consider the XML tree in Figure 2.1. Corresponding Containment index is shown in Figure 5.12. The entry $(Q, X(Q))$ does not need to maintain its extent $X(Q)$ if $X(Q)$ can be identified by searching its descendants in the index. For instance, consider C_d-queries: $Q_1 =$`//LORD`, $Q_2 =$`//HOUSES//LORD`, $Q_3 =$`//HOUSE//LORD` and

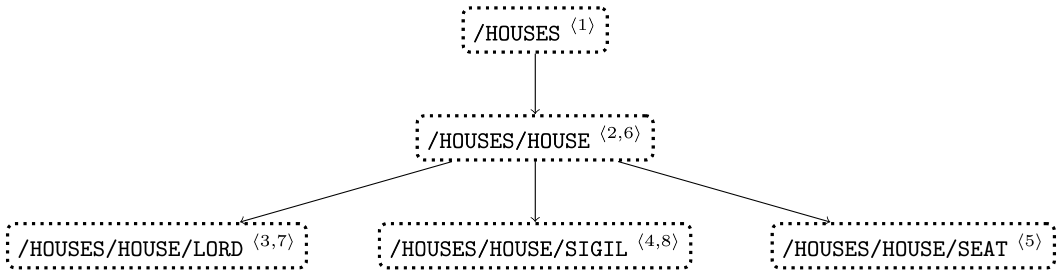


Figure 5.11. 1-Index

$Q_4 =$ `HOUSES//HOUSE//LORD`. The results for them are the same. There is no need to maintain the same extend four times. The same extent for the four queries can be maintained at $X($`//HOUSES//HOUSE//LORD`$)$ only. ♠

The Containment index can efficiently support path-containment relationship, but it cannot support prefix relationship as 1-Index, e.g., it cannot easily identify $Q =$ `//HOUSES//HOUSE` from $Q' =$ `//HOUSES`. To be able to support all C-queries the index needs to maintain both prefix and path-containment relationship. To define such an index, called PP-Index, we need to first introduce the notion of weak extend and functions $single(Q)$ and $double(Q)$.

Definition 5.25. (Weak extend). *Weak extent*, denoted as $\hat{X}(Q)$, is such an extent that $\hat{X}(Q) \subseteq X(Q)$. Hence, a query Q may need to be answered by several entries. ♣

Definition 5.26. (Functions $single(Q)$ and $double(Q)$). $double(Q)$ is a function replacing all child axes `/` in Q by descendant-or-self axis `//`. Similarly, $single(Q)$ is a function replacing all descendant-or-self axes `//` in Q by child axes `/`. ♣

Definition 5.27. (PP-Index). *PP-Index* is an index for C-queries. Index entries are C_d-queries, since any C-query Q has a unique corresponding C_d-query, $Q_d = DOUBLE(Q)$. The weak extend maintained for $(Q_d, \hat{X}(Q_d))$ is $\hat{X}(Q) = X(Q_c)$ where $Q_c = SINGLE(Q_d)$. ♣

Definition 5.28. (Real and virtual nodes). If the weak extent of the node v is non-empty then v is called *real node*, otherwise it is called *virtual node*. ♣

Definition 5.29. (Removable virutal node). A virtual node v in a PP-Index is *removable* if all nodes v_i, which have p-edges from v_i to v, are virtual nodes. ♣

All removable nodes and the incoming/outgoing edges around them can be removed, which results in a compressed graph. Since number of virtual nodes is large, the compression greatly reduces the entry size from $\mathcal{O}(2^h)$ to $\mathcal{O}(h^2)$, where h is the height of an XML tree model. The compressed PP-Index is illustrated in Figure 5.14.

5.3.2 Query Processing

First, a C_c-query (Q_c) can be processed to find its corresponding entry $(Q_d, \hat{X}(Q_d))$ where $Q_d = DOUBLE(Q_c)$ and return $\hat{X}(Q_d)$ since $\hat{X}(Q_d) = X(Q_c)$. Second, a C_d-query (Q_d) can be processed to find the entry $(Q_d, \hat{X}(Q_d))$, and combine $\hat{X}(Q_d)$ and all the weak extends $\hat{X}(Q'_d)$ if $(Q'_d, \hat{X}(Q'_d))$ is a descendant of $(Q_d, \hat{X}(Q_d))$ in the index and both Q_d and Q'_d have the same last label (path-containment). Finally, any other C-queries (i.e., C_x-queries) can be processed by combining the techniques mentioned above.

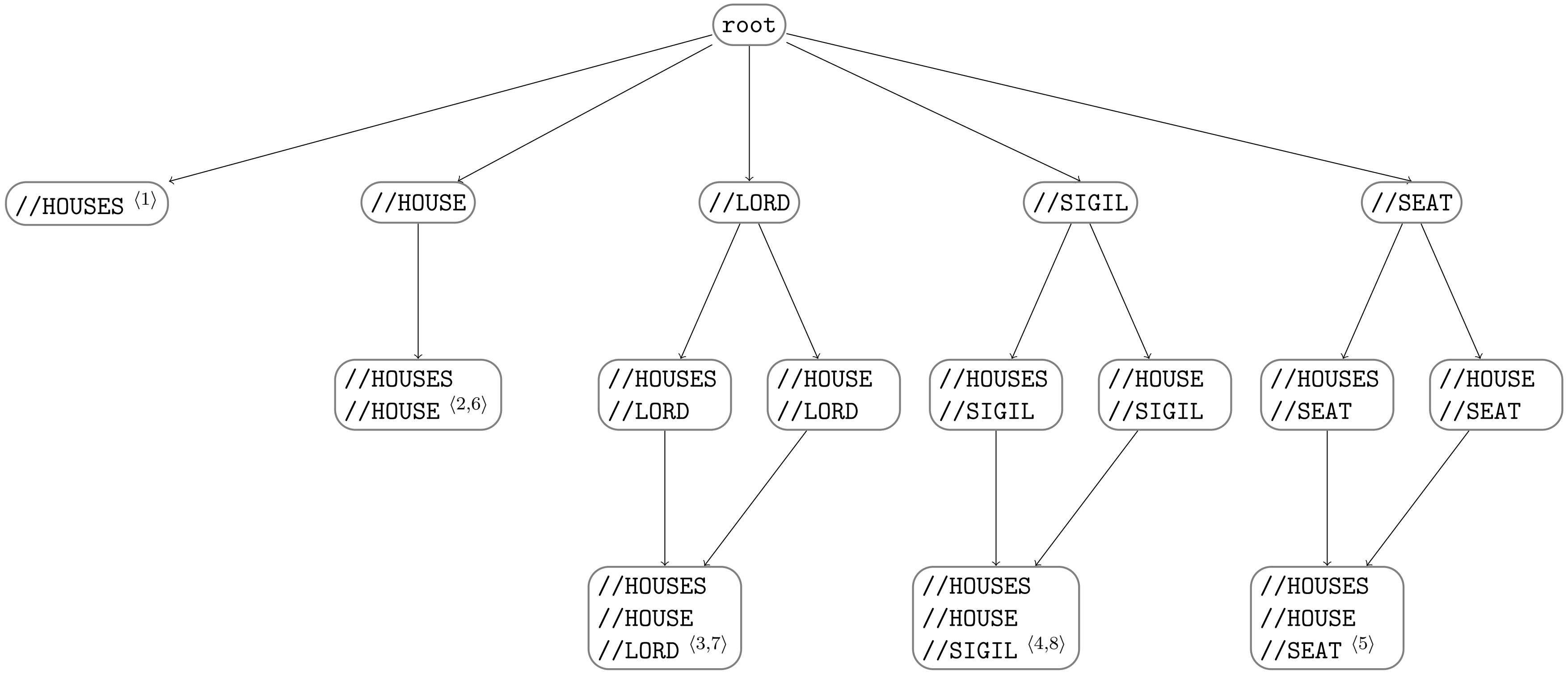


Figure 5.12. Containment index

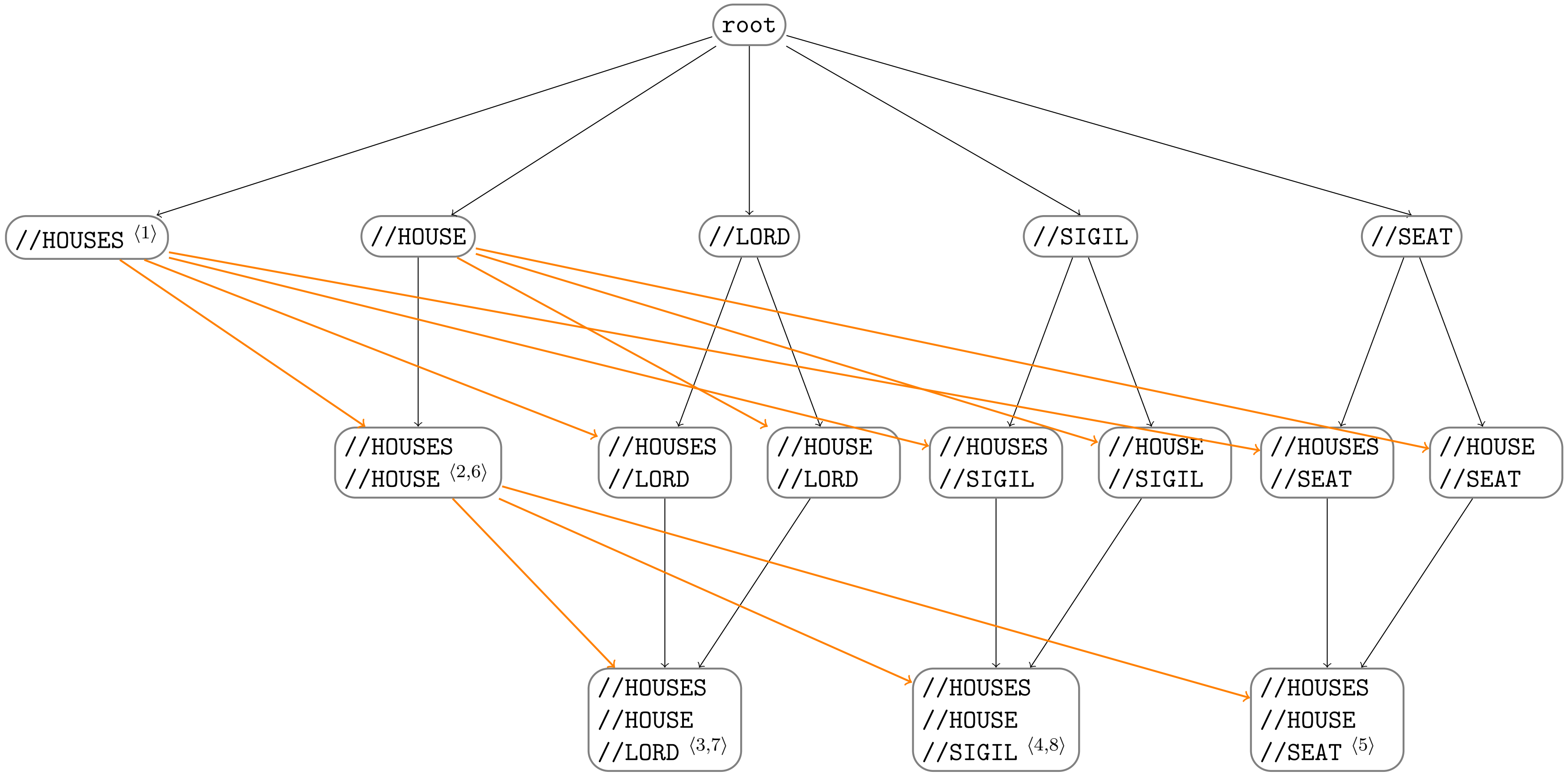
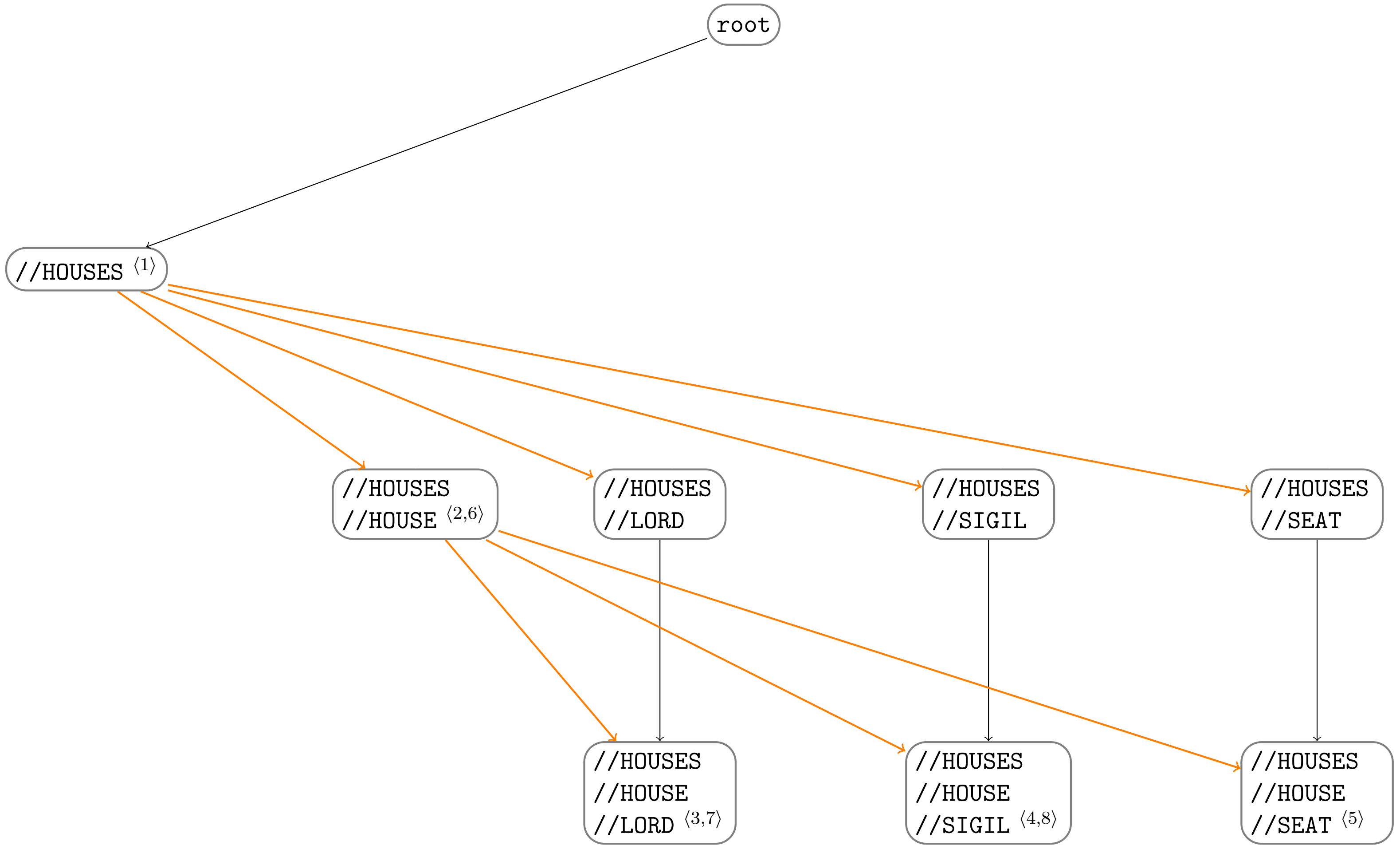


Figure 5.13. PP-Index



root
//HOUSES ⟨1⟩
//HOUSES //HOUSE ⟨2,6⟩
//HOUSES //LORD
//HOUSES //SIGIL
//HOUSES //SEAT
//HOUSES //HOUSE //LORD ⟨3,7⟩
//HOUSES //HOUSE //SIGIL ⟨4,8⟩
//HOUSES //HOUSE //SEAT ⟨5⟩

Figure 5.14. Compressed PP-Index

Example 5.30. Consider query $Q =$ `/HOUSES/HOUSE/LORD`. We need to find its corresponding entry $(Q_d, \hat{X}(Q_d))$ where $Q_d = double(Q) =$ `//HOUSES//HOUSE//LORD`. There we just return the extend $3, 7$. ♠

Example 5.31. Consider query $Q =$ `//LORD`. We need to find its corresponding entry $(Q_d, \hat{X}(Q_d))$ where $Q_d = Q =$ `//LORD`. Then we combine the result with descendants as follows: $X($`//LORD`$) = \hat{X}($`//LORD`$) \cup \hat{X}($`//HOUSES//LORD`$) \cup \hat{X}($`//HOUSE//LORD`$) \cup \hat{X}($`//HOUSES//HOUSE//LORD`$)$. The weak extends for the first three entries are empty. Therefore, we have $X($`//LORD`$) = \hat{X}($`//HOUSES//HOUSE//LORD`$) = 3, 7$. ♠

5.4 PRIX

Rao and Moon in [33] proposed sequence-based indexing method called *PRIX (PRüfer sequences for Indexing XML)* which is, as indicated by the name, based on Prüfer sequences. In the PRIX system, an XML document is transformed into a sequence of labels by Prüfer's method that constructs one-to-one correspondence between trees and sequences. During query processing, a query tree model is also transformed into its Prüfer sequence.

By performing subsequence matching and a series of refinement phases, all the occurrences of a query tree are found. This approach returns correct answers without false alarms and false dismissals. The PRIX system is able to support both path and twig queries efficiently. Additionally, this tree-to-sequence transformation guarantees a worst-case bound on the index that is linear in the total number of nodes in the XML document tree.

5.4.1 PRIX Index Structure

Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labelled tree and a sequence by removing nodes from the tree one at a time.

Definition 5.32. (Prüfer sequence). Let T be a tree with n nodes labelled from 1 to n. From T, delete a leaf with the smallest label to form a smaller tree T with $n-1$ nodes. Let l_1 denote the label of the node that was the parent of the deleted node. Repeat this process on T with $n-1$ nodes to determine l_2 (the parent of the next node to be deleted), and continue until only two nodes joined by an edge are left. The sequence $(l_1, l_2, l_3, \ldots, l_{n-2})$ is called the *Prüfer sequence* of tree T. The length of Prüfer sequence of tree T is $n-2$. From the sequence $(l_1, l_2, l_3, \ldots, l_{n-2})$, the original tree T can be reconstructed. ♣

In PRIX approach a Prüfer sequence of length $n-1$ is constructed by continuing the deletion of nodes till only one node is left. To uniquely label an XML document tree, the postorder numbering scheme has been chosen. Figure 5.15 illustrates an XML tree model $\mathcal{T}$ of the XML document $\mathcal{D}$ from example 2.42 using postorder numbering scheme.

Definition 5.33. (Numbered Prüfer sequence). *Numbered Prüfer sequence* (NPS) is a sequence consisting entirely of postorder numbers. ♣

Definition 5.34. (Labelled Prüfer sequence). *Labelled Prüfer sequence* (LPS) is the NPS where each number is replaced by its corresponding tag. ♣

Example 5.35. Consider the XML tree $\mathcal{T}$ in Figure 5.15. $LPS(\mathcal{T}) =$ `HOUSE LORD HOUSE SIGIL HOUSE SEAT HOUSE HOUSES HOUSE LORD HOUSE SIGIL HOUSE HOUSES`, and $NPS(\mathcal{T}) = 8\,3\,8\,5\,8\,7\,8\,15\,14\,11\,14\,13\,14\,15$. ♠

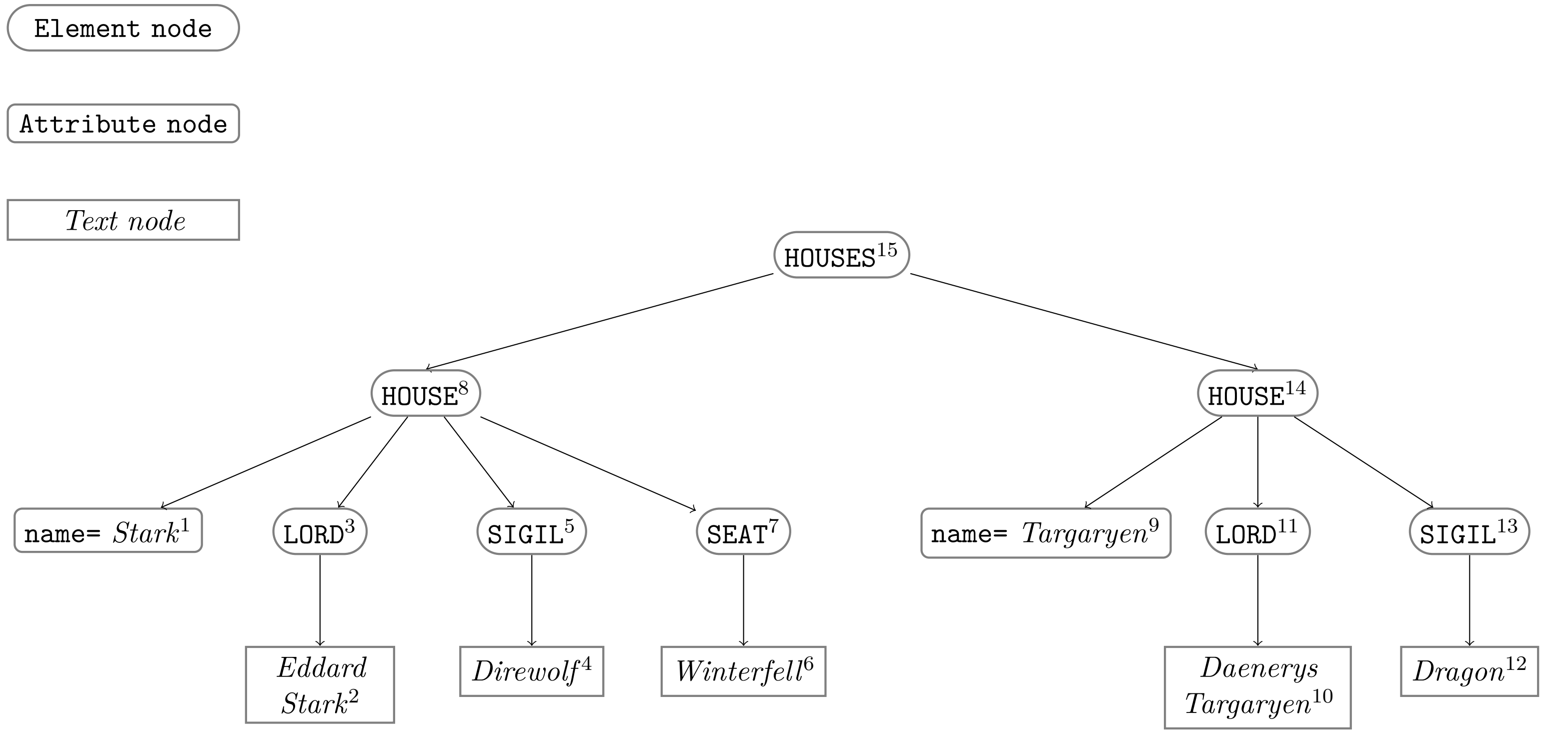


Figure 5.15. XML tree model (postorder numbering scheme)

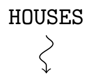


Figure 5.16. Query Q_2

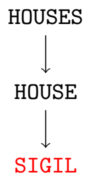


Figure 5.17. Query Q_1

In the PRIX system, Prüfer sequence is constructed for an XML tree model (with nodes numbered in postorder) using the method described above. In order to support fast subsequence matching during query processing, the Labeled Prüfer sequence is indexed using B^+-trees.

5.4.2 Prüfer Sequence Properties

- Given a tree T with n nodes, numbered from 1 to n in postorder, the node deleted the i^{th} time during Prüfer sequence construction is the node numbered i.
- The i^{th} element in the NPS denotes the postorder number of the parent of node i.
- The number of times a number n occurs in an NPS indicates the number of child nodes of n in the tree, and the positions that n occurs in the NPS depend on the subtrees rooted at node n.

5.4.3 Query Processing

A query tree is transformed into its Prüfer sequence like XML documents. Finding the matches involves a series of filtering and refinement phases, namely

- filtering by *subsequence matching*,
- refinement by *connectedness*,
- refinement by *structure*,
- refinement by *leaf nodes*.

The *filtering phase* involves subsequence matching. In this phase, given a query Q, all the subsequences in $LPS(\mathcal{T})$ that match $LPS(Q)$ are found. If tree Q is a subgraph of tree $\mathcal{T}$, then $LPS(Q)$ is a subsequence of $LPS(\mathcal{T})$, so there is guaranteed to have no false dismissals.

Example 5.36. Consider the XML tree model $\mathcal{T}$ in Figure 5.15 and the tree model of $Q_1 =$ `/HOUSES/HOUSE/LORD` in Figure 5.17.

- $NPS(\mathcal{T}) = 8\,3\,8\,5\,8\,7\,8\,15\,14\,11\,14\,13\,14\,15$,
- $LPS(T) =$ `HOUSE LORD HOUSE SIGIL HOUSE SEAT HOUSE HOUSES HOUSE LORD HOUSE SIGIL HOUSE HOUSES`,
- $NPS(Q) = 2\,3$,
- $LPS(Q) =$ `HOUSE HOUSES`.

Q is a subgraph of T, and $LPS(Q)$ matches a subsequence S of $LPS(\mathcal{T})$ at positions $(1, 8)$ (for example). The postorder number sequence of subsequence S is $8\,15$. ♠

The subsequences matched during the filtering phase are further examined for the property of *connectedness*. This is because, only for some of the subsequences, all the labels in the subsequence correspond to nodes that are connected in the tree. The sequences that satisfy the connectedness property are called *tree sequences*.

Definition 5.37. (Connectedness). Given a tree $\mathcal{T}$, let $N_{\mathcal{T}} = NPS(\mathcal{T})$. Let S be a subsequence of $LPS(\mathcal{T})$ and let N be the postorder number sequence of S and N_i its i^{th} element. Then the tree nodes in $\mathcal{T}$ corresponding to the labels of S are connected only if:

$$\forall N_i : (N_i \neq max(N_1, N_2, \ldots, N_{|S|})) \wedge (\neg\exists(j > i) : N_j = N_i) \Rightarrow N_{i+1} = N_{\mathcal{T}}[N_i] \quad ♣$$

The intuition for the definition above is as follows. Let i be the index of the last occurrence of a postorder number n in an NPS. This last occurrence is a result of deletion of the last child of n during Prüfer sequence construction. Hence the next child to be deleted is the node n itself. Hence the number at the $(i+1)^{th}$ index in the NPS, say m, is the postorder number of the parent of node n. Thus n followed by m indicates that there is and edge between node m and node n.

Example 5.38. Consider the XML tree model $\mathcal{T}$ in Figure 5.15 and two subsequences S_A and S_B of $LPS(\mathcal{T})$.

- $NPS(\mathcal{T}) = 8\,3\,8\,5\,8\,7\,8\,15\,14\,11\,14\,13\,14\,15$,
- $LPS(\mathcal{T}) =$ `HOUSE LORD HOUSE SIGIL HOUSE SEAT HOUSE HOUSES HOUSE LORD HOUSE SIGIL HOUSE HOUSES`,
- $S_A =$ `SIGIL HOUSE HOUSES` whose postorder number sequence $N_A = 5\,14\,15$,
- $S_B =$ `HOUSE HOUSES` whose postorder number sequence $N_B = 8\,15$.

The nodes represented by labels of S_A form a disconnected graph. In this case $max(N_{A1}, N_{A2}, N_{A3}) = max(5, 14, 15) = 15$. The last occurence of postorder number 5 in N_A is at the 1^{st} position since there is no index $j > 1$ such that $N_{Aj} = 5$. However N_{A1} is not followed by $N_T[5]$, i.e., $N_{A2} \neq 8$. Hence, the necessary connectedness condition is not satisfied. The nodes represented by of S_B represents a tree because the necessary connectedness condition is satisfied. ♠

The tree sequences are further refined based on the query structure. In this phase it is determined if the structure of the tree represented by a tree sequence matches the query structure.

Definition 5.39. (Gap). The *gap* between two nodes $\mathfrak{n}_1$ and $\mathfrak{n}_2$ in a tree is defined as the difference between the postorder numbers of the nodes $\mathfrak{n}_1$ and $\mathfrak{n}_2$. ♣

Definition 5.40. (Gap consistency). Tree sequence A is said to be gap consistent with respect to tree sequence B if

- A and B have the same length.
- For every pair of adjacent elements in A and the corresponding adjacent elements in B, their gaps, g_A and g_B have the same sign, and if $|g_A| > 0$ then $|g_A| \leq |g_B|$, else $g_A = g_B = 0$. ♣

Intuitively, the gap between two nodes in a tree gives an idea of how many nodes are encountered during postorder traversal between two nodes. If more nodes are traversed in the query tree as compared to the XML tree, then this indicates that there is a structural difference between the XML tree and query tree.

Definition 5.41. (Frequency consistency). Tree sequences A and B are frequency consistent if

- A and B have the same length n.
- Let N_A and N_B be the postorder number sequences of A and B respectively. Let n_{Ai} and n_{Bi} be the i^{th} element in N_A and N_B respectively. For every i from $1, \ldots, n$,

n_{Ai} occurs k times in N_A at positions $p_1, p_2, \ldots, p_k$, iff n_{Bi} occurs k times in N_B at positions $p_1, p_2, \ldots, p_k$. ♣

It should be noted that the LPS of a tree contains only the non-leaf node labels. Therefore, filtering by subsequence matching followed by refinement by connectedness and structure can only find matches in the data tree whose tree structure is the same as query tree and whose non-leaf node labels match the non-leaf node labels of the query twig. Such matches are called partial matches.

Definition 5.42. (Partial match). Tree model of Q has a *partial match* in XML tree model $\mathcal{T}$ iff

- $LPS(Q)$ matches a subsequence S of $LPS(\mathcal{T})$ such that S is a tree sequence, and
- $LPS(Q)$ is gap consistent and frequency consistent with subsequence S. ♣

In the final refinement phase, the leaf node labels of the query are tested with the leaf node labels of partially matched queries in the data to find complete query matches.

Example 5.43. Consider the XML tree $\mathcal{T}$ in Figure 5.15 and the tree model of $Q =$ `/HOUSES/HOUSE[SIGIL=`*Direwolf*`]` in Figure 5.17.

- $NPS(\mathcal{T}) = 8\,3\,8\,5\,8\,7\,8\,15\,14\,11\,14\,13\,14\,15$, $LPS(\mathcal{T}) =$ `HOUSE LORD HOUSE SIGIL HOUSE SEAT HOUSE HOUSES HOUSE LORD HOUSE SIGIL HOUSE HOUSES`,
- $Leaves(\mathcal{T}) = 1\,2\,4\,6\,9\,10\,12$,
- $NPS(Q) = 2\,3\,4$, $LPS(Q) =$ `HOUSE HOUSES SIGIL`.

$LPS(Q)$ matches a subsequence S of $LPS(\mathcal{T})$ at positions $(4, 7, 8)$. The postorder number sequence of subsequence S is $N = 5\,8\,15$. $LPS(Q)$ is gap consistent and frequency consistent with S. We can match leaf $Direwolf^1$ in Q as follows. Since the leaf has postorder number 1, its parent node matches the node numbered 5 (i.e., the 1st element of N) in the data tree. Also since the node numbered 5 occurs at the 4th position in $LPS(\mathcal{T})$, it may have a leaf $Direwolf^4$. And indeed, we have $Direwolf^4$ in the leaf node list of $\mathcal{T}$. ♠

However, this refinement phase can be eliminated by special treatment of leaf nodes in the query tree and data trees. The key idea is to make the leaf nodes of the query tree and the data trees appear in their LPS, so that all the nodes are examined during subsequence matching and refinement by connectedness and structure phases. To handle descendant-or-self axis and wildcards a simple modification to the refinement-by-connectedness phase is needed.

Chapter 6
Automata Approach to Indexing XML Data

This chapter shows that automata can be used effectively for the purpose of indexing XML documents. Although we attempt to support paths queries only, the techniques described here are relevant to the general XPath processing problem, for two reasons.

First, processing linear expressions is a subproblem in processing more complex queries as we can decompose them into linear fragments. Second, this can be seen as a building block for more powerful processor, such as pushdown automaton, able to process twig queries.

We start with introduction of *Tree String Paths Automaton* representing an index for $XP^{\{/,name-test\}}$ queries, i.e., paths queries using child-axis (/) only. Second, *Tree String Path Subsequences Automaton* indexing $XP^{\{//,name-test\}}$ queries, i.e., paths queries using descendant-or-self axis (//) only, is presented. Finally, we introduce a full XML index for $XP^{\{/,//,name-test\}}$ queries, i.e., paths queries which may use any combination of / and // called *Tree Paths Automaton*.

For the XML document of size n, the search phase of all elements satisfying the query of size m is performed in time linear in m and not depending on n. The major issue is the size of the deterministic automaton, which, in theory, can be exponential in the size of the XML document being indexed.

However, we provide a series of experiments in following chapter to show that the determinisation will result in a small number of states although the number of queries accepted by the *Tree Paths Automaton* is exponential in n (e.g., $\mathcal{O}(2.62^n)$ [34] for a linear XML tree model with n nodes).

6.1 XML Data Model

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

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. Without loss of generality, we have chosen to use a preorder numbering scheme to uniquely assign an identifier to each of the tree nodes.

It is obvious that only for well-formed XML documents there exists an XML tree model. Therefore, we will assume that only well-formed documents are presented as inputs for our indexing methods. We now introduce the term XML alphabet, which we need to refer to in algorithms presented later.

Definition 6.1. (XML alphabet). Let $\mathcal{D}$ be an XML document. An *XML alphabet* $\mathcal{E}$ of $\mathcal{D}$ denoted $\mathcal{E}(\mathcal{D})$ is a an alphabet where each symbol represents a label of an XML element. ♣

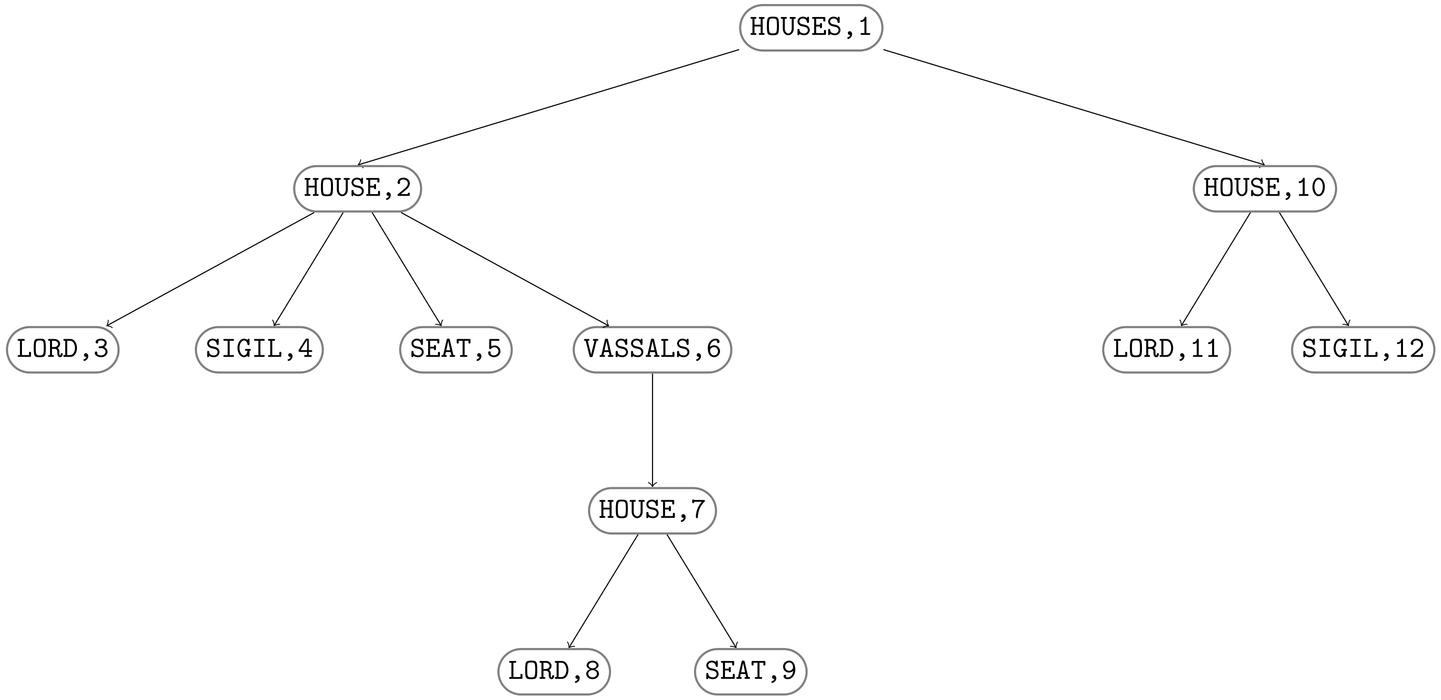


Figure 6.1. XML tree model of the XML document from Example 6.2

Example 6.2. Consider following sample of an XML document $\mathcal{D}$ that is an extended version of the XML document presented in Example 2.42. Figure 6.1 shows its corresponding XML tree model $\mathcal{T}$. The XML alphabet $\mathcal{E}(\mathcal{D}) = \{$`HOUSES`, `HOUSE`, `LORD`, `SIGIL`, `SEAT`, `VASSALS`$\}$.

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

♠

6.2 Tree String Paths Automaton

Definition 6.3. (Tree String Paths Automaton). Let $\mathcal{D}$ be an XML document. A Tree String Paths Automaton accepts all $XP^{\{/,name-test\}}$ queries of $\mathcal{D}$. ♣

A *Tree String Paths Automaton* (TSPA) speeds up the evaluation of linear XPath queries using only child-axis (/) (i.e., $XP^{\{/,name-test\}}$). The most similar approaches from XML indexing techniques are graph-based methods constructing a structural path summary [28, 23, 25].

However, we build our index as a composition of finite automata accepting parts of paths queries. Hence, the index is simple and well understandable for anyone who is familiar with the automata theory. As we attempt to index paths queries only, we can omit the branching structure and describe the XML tree model by means of its linear fragments, called string paths.

Definition 6.4. (String path). Let $\mathcal{T}$ be an XML tree model with height h. *A string path* $P = \mathfrak{n}_1\mathfrak{n}_2 \dots \mathfrak{n}_t$ of length t, where $t \leq h$, of $\mathcal{T}$ is a linear path leading from a root $r = \mathfrak{n}_1$ to a leaf $\mathfrak{n}_t$. Each element $\mathfrak{n}_i$ of the path is associated with an identifier and label, denoted $id(\mathfrak{n}_i)$ and $label(\mathfrak{n}_i)$, respectively. The identifier corresponds to a preorder number of the element. ♣

Definition 6.5. (String paths set). Let $\mathcal{D}$ and $\mathcal{T}$ be an XML document and its XML tree model, respectively. A set of all string paths over $\mathcal{T}$ is called *string paths set*, denoted by $\mathcal{P}_{\mathcal{T}} = \{P_1, P_2 \dots P_k\}$, where k is the number of leaves in $\mathcal{T}$. ♣

Example 6.6. Consider the XML tree model $\mathcal{T}$ illustrated in Figure 6.1. We show its corresponding string paths set $\mathcal{P}_{\mathcal{T}}$ below. We represent each node $\mathfrak{n}$ in $\mathcal{T}$ by its label (i.e., $label(\mathfrak{n})$) and show its identifier (i.e., $id(\mathfrak{n})$) in parenthesis.

```
  𝒫𝒯 = {
         HOUSES(1) HOUSE(2) LORD(3),
         HOUSES(1) HOUSE(2) SIGIL(4),
         HOUSES(1) HOUSE(2) SEAT(5),
         HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) LORD(8),
         HOUSES(1) HOUSE(2) VASSALS(6) HOUSE(7) SEAT(9),
         HOUSES(1) HOUSE(10) LORD(11),
         HOUSES(1) HOUSE(10) SIGIL(12)
       }
```

♠

TSPA is basically a prefix automaton for a set of strings [18], where the strings are the string paths of the XML tree model that corresponds to the XML document being indexed. To construct TSPA, we at first start with a finite automaton that accepts all non-empty prefixes of a single string path whose construction is described by Algorithm 6.7. The constructed finite automaton is deterministic.

Algorithm 6.7. Construction of a prefix automaton for a single string path.
Input: A string path $P = \mathfrak{n}_1\mathfrak{n}_2 \dots \mathfrak{n}_t$.
Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, 0, F)$ accepting all non-empty prefixes of P.
Method:

- $\mathcal{Q} \leftarrow \{0, id(\mathfrak{n}_1), id(\mathfrak{n}_2), \dots, id(\mathfrak{n}_t)\}$,
- $\mathcal{A}$ is the set of all different node labels in P,
- $\delta(0, /label(\mathfrak{n}_1)) \leftarrow id(\mathfrak{n}_1)$ and
  $\delta(id(\mathfrak{n}_i), /label(\mathfrak{n}_{i+1})) \leftarrow id(\mathfrak{n}_{i+1})$ for all $i = 1, 2, \dots, t-1$,
- $F \leftarrow \{id(\mathfrak{n}_1), id(\mathfrak{n}_2), \dots, id(\mathfrak{n}_t)\}$. ♢

Example 6.8. Consider the XML tree model $\mathcal{T}$ illustrated in Figure 6.1 and its corresponding string paths set $\mathcal{P}_\mathcal{T}$ described in Example 6.6. Transition diagrams of prefix automata constructed by Algorithm 6.7 for each string path contained in $\mathcal{P}_\mathcal{T}$ are shown in Figure 6.2. ♠

To build TSPA, we can run all the prefix automata (constructed by Algorithm 6.7 for all string paths P_i in $\mathcal{P}_\mathcal{T}$) "in parallel", by remembering the states of all automata while reading the input. This is achieved by the product construction. This way we construct the Tree String Paths Automaton $\mathcal{M}$ for $\mathcal{T}$.

Algorithm 6.9. Construction of Tree String Paths Automaton for an XML document $\mathcal{D}$ and its corresponding XML tree model $\mathcal{T}$.
Input: A string paths set $\mathcal{P}_\mathcal{T} = \{P_1, P_2, \dots P_k\}$.
Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{E}(\mathcal{D}), \delta, 0, F)$ accepting all $XP^{\{/,name-test\}}$ queries of $\mathcal{D}$.
Method:

1. Construct finite automata $\mathcal{M}_i = (\mathcal{Q}_i, \mathcal{A}_i, \delta_i, 0, F_i)$ accepting a set of non-empty prefixes of P_i using Algorithm 6.7.
2. Construct deterministic Tree String Paths Automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{E}(\mathcal{D}), \delta, 0, F)$ accepting a set of all non-empty prefixes of each P_i using product construction. ♢

Example 6.10. Let $\mathcal{D}$ and $\mathcal{T}$ be the XML document and tree model from Example 6.2 and Figure 6.1, respectively. The corresponding Tree String Paths Automaton, constructed by Algorithm 6.9, is shown in Figure 6.3. ♠

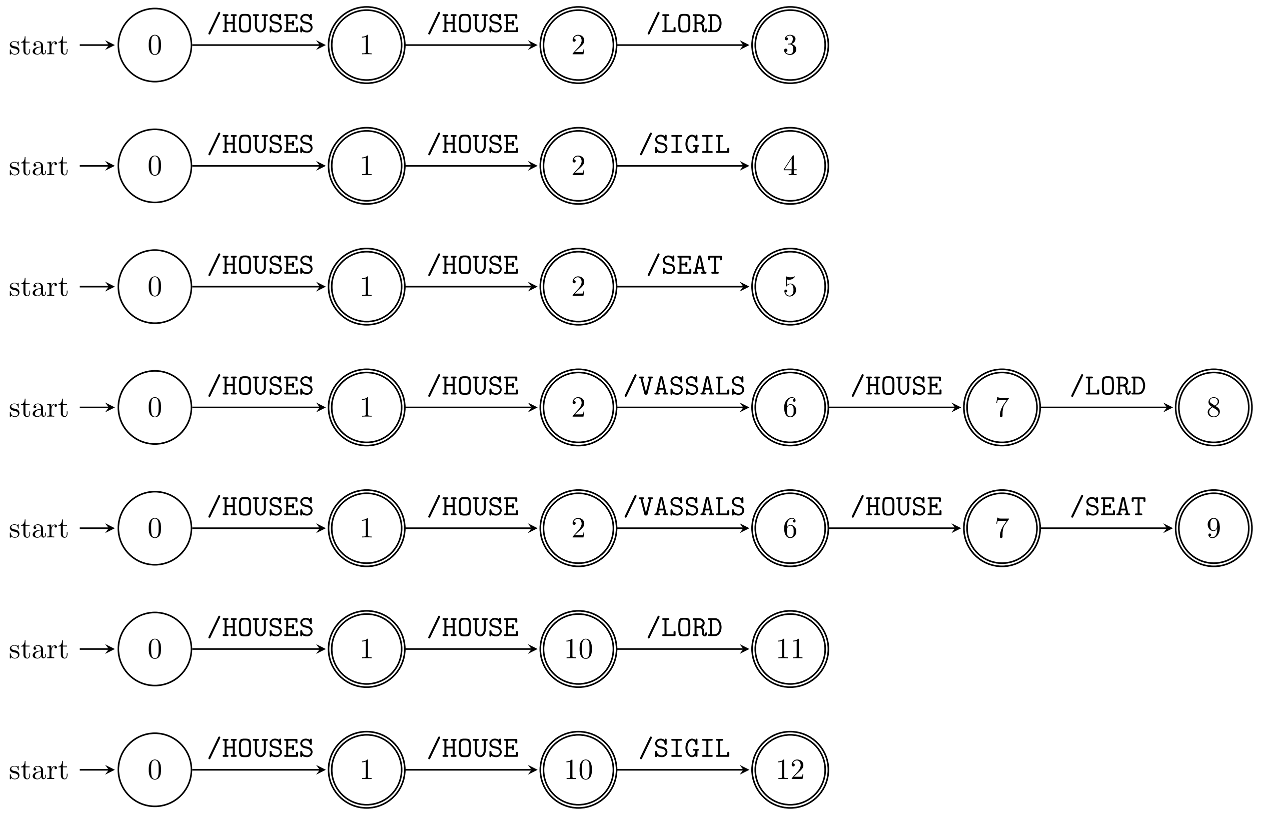

start 0 /HOUSES 1 /HOUSE 2 /LORD 3
start 0 /HOUSES 1 /HOUSE 2 /SIGIL 4
start 0 /HOUSES 1 /HOUSE 2 /SEAT 5
start 0 /HOUSES 1 /HOUSE 2 /VASSALS 6 /HOUSE 7 /LORD 8
start 0 /HOUSES 1 /HOUSE 2 /VASSALS 6 /HOUSE 7 /SEAT 9
start 0 /HOUSES 1 /HOUSE 10 /LORD 11
start 0 /HOUSES 1 /HOUSE 10 /SIGIL 12

Figure 6.2. Prefix automata for each of string paths contained in $\mathcal{P}_{\mathcal{T}}$ from Example 6.6

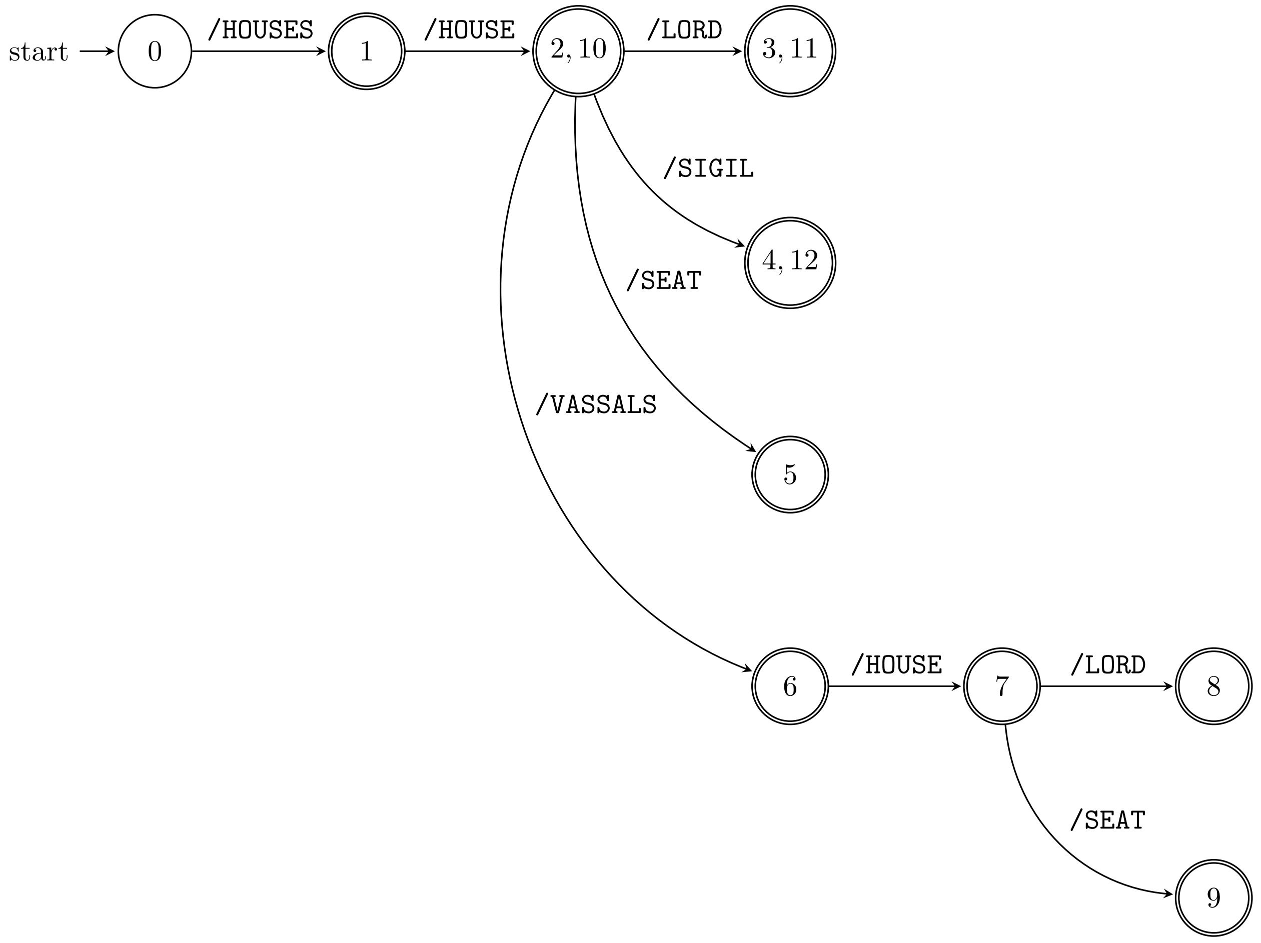


Figure 6.3. Tree String Paths Automaton

6.2.1 Discussion on Time and Space Complexities

A Tree String Paths Automaton effectively supports the evaluation of all $XP^{\{/,name-test\}}$ queries of an XML document. For example, for a query $Q =$ `/HOUSES/HOUSE/LORD`, the result set of element nodes is contained in the d-subset $\{3, 11\}$.

Given a subject XML document $\mathcal{D}$ and its corresponding XML tree model $\mathcal{T}$ with n nodes, the tree is preprocessed and the index is constructed. The searching phase uses the index, reads an input query Q of size m and computes the list of positions of all occurrences of target nodes of Q in $\mathcal{T}$.

TSPA performed the searching in time $\mathcal{O}(m)$ and does not depend on n. Number of states of deterministic TSPA is linear in number of nodes in $\mathcal{T}$, i.e., $\mathcal{O}(n)$ (proof is trivial).

6.3 Tree String Path Subsequences Automaton

Definition 6.11. Let $\mathcal{D}$ be an XML document. A Tree String Path Subsequences Automaton accepts all $XP^{\{//,name-test\}}$ queries of $\mathcal{D}$. ♣

A *Tree String Path Subsequences Automaton* (TSPSA) efficiently evaluates all linear XPath queries where just descendant-or-self axis (`//`) is used (i.e., $XP^{\{//,name-test\}}$). For an XML document of size n, the automaton processes a query of size m in time linear in m and not depending on n. The structural path summaries [28, 23, 25] usually need further tree traversal to support queries containing `//` axis. Furthermore, the proposed index is again based on the idea of automata composing, that makes it well understandable for anyone familiar with automata theory.

Once more, we use the idea to describe an XML tree model by means of its linear fragments called string paths. However, to satisfy queries with `//` axes we are in this case interested in (non-empty) subsequences of string paths, rather than its prefixes.

Definition 6.12. (Subsequence of a string path). Let $\mathcal{D}$ be an XML document and $P = \mathfrak{n}_1\mathfrak{n}_2 \ldots \mathfrak{n}_t$ be one of its string paths. A *subsequence* of P is any sequence of elements $\mathfrak{n}_i$ obtainable by deleting zero or more elements from P. ♣

Example 6.13. Let $\mathcal{T}$ be an XML tree model in Figure 6.1 and $P =$ `HOUSES HOUSE LORD` be one of its string paths (we represent individual XML element nodes by their labels). There are 7 non-empty subsequences of P: `HOUSES HOUSE LORD`, `HOUSES HOUSE`, `HOUSES LORD`, `HOUSE LORD`, `HOUSES`, `HOUSE`, `LORD`. ♠

6.3.1 Building the Tree String Path Subsequences Automaton

TSPSA is in fact a subsequence automaton [18] for a set of strings where the strings are string paths of the XML tree model that represents the XML document being indexed. The automaton solving the problem of subsequences for both single and multiple strings is also referred as Directed Acyclic Subsequence Graph (DASG), first introduced by Baeza-Yates in [35] and further studied in [36–39]. Therefore, we propose an XML index problem to be another application area of DASG.

There are three building algorithms for DASG for a set of strings available: right-to-left [35], left-to-right [37] and on-line [38]. However, none of them is based on a standard subset construction, that is crucial for answering queries. Therefore, we propose a construction of TSPSA consisting of two steps. First, deterministic subsequence automata accepting a set of non-empty subsequences for each of string paths contained in $\mathcal{P}_\mathcal{T}$ are constructed using subset construction. Second, a TSPSA is built using product construction.

To build the deterministic subsequence automaton, we propose two following algorithms: Algorithm 6.14 (slightly modified method described in [18]) and Algorithm 6.18 (direct subset construction of deterministic subsequence automaton). Resulting automata are used to build TSPSA by Algorithm 6.20.

Algorithm 6.14. Construction of a subsequence automaton for a single string path.
Input: A String Path $P = \mathfrak{n}_1\mathfrak{n}_2 \dots \mathfrak{n}_t$.
Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, 0, F)$ accepting all non-empty subsequences of P.
Method:

1. Construct a deterministic finite automaton $\mathcal{M}_1 = (\mathcal{Q}_1, \mathcal{A}, \delta_1, 0, F_1)$ accepting all prefixes of P:
   a) $\mathcal{Q}_1 \leftarrow \{0, id(\mathfrak{n}_1), id(\mathfrak{n}_2), \dots, id(\mathfrak{n}_t)\}$,
   b) $\mathcal{A}$ is the set of all different node labels in P,
   c) $\delta_1(0, //label(\mathfrak{n}_1)) \leftarrow id(\mathfrak{n}_1)$ and
   $\delta_1(id(\mathfrak{n}_i), //label(\mathfrak{n}_{i+1})) \leftarrow id(\mathfrak{n}_{i+1})$, $\forall i = 1, 2, \dots, t-1$,
   d) $F_1 \leftarrow \{id(\mathfrak{n}_1), id(\mathfrak{n}_2), \dots, id(\mathfrak{n}_t)\}$.
2. Insert ε-transitions into the automaton $\mathcal{M}_1$ leading from each state to its next state. Resulting automaton $\mathcal{M}_2 = (\mathcal{Q}_2, \mathcal{A}, \delta_2, 0, F_2)$, where
   a) $\mathcal{Q}_2 \leftarrow \mathcal{Q}_1$, $F_2 \leftarrow F_1$,
   b) $\delta_2 \leftarrow \delta_1 \cup \delta'$ and $\delta'(0, \varepsilon) \leftarrow id(\mathfrak{n}_1)$, $\delta'(id(\mathfrak{n}_i), \varepsilon) \leftarrow id(\mathfrak{n}_{i+1})$, $\forall i \leftarrow 1, 2, \dots, t-1$.
3. Eliminate all ε-transitions. The resulting automaton is $\mathcal{M}_3$.
4. Construct a deterministic finite automaton $\mathcal{M}$ equivalent to $\mathcal{M}_3$ using standard determinisation algorithm based on a subset construction. ♢

Example 6.15. Consider the XML tree model $\mathcal{T}$ illustrated in Figure 6.1 and its corresponding string paths set $\mathcal{P}_\mathcal{T}$ described in Example 6.6. Transition diagrams of subsequence automata with ε-transitions constructed for each string path contained in $\mathcal{P}_\mathcal{T}$ are shown in Figure 6.4. Resulting deterministic automata are illustrated in Figures 6.5 and 6.6. ♠

In order to speed up the subsequence automaton construction, we simplify the previous algorithm and propose a direct subset construction of a deterministic subsequence automaton for a single string path described by Algorithm 6.18.

Definition 6.16. (Set of occurences of an element in a string path). Let $P = \mathfrak{n}_1\mathfrak{n}_2 \dots \mathfrak{n}_t$ be a string path and e be an element node with label occurring at several positions in P (i.e., $label(\mathfrak{n}_i) = label(e)$ for some i). A *set of occurrences of the element e in P* is a totally ordered set $O_P(e) = \{o \mid o = id(\mathfrak{n}_i) \wedge label(\mathfrak{n}_i) = label(e), i = 1, 2, \dots, t\}$. The ordering is equal to ordering of element prefix identifiers as natural numbers. ♣

Definition 6.17. (ButFirst). Let P and $O_P(e) = \{o_1, o_2, \dots, o_t\}$ be a string path and a set of occurrences of an element e in a string path P, respectively. Then we define a function $ButFirst(O_P(e)) = \{o_2, \dots, o_t\}$. ♣

Algorithm 6.18. A direct subset construction of a subsequence automaton for a single string path.
Input: A string path $P = \mathfrak{n}_1\mathfrak{n}_2 \dots \mathfrak{n}_t$.

Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$ accepting all non-empty subsequences of P.
Method:

```
1. Let $\mathcal{A}'$ be the set of all different node labels in $P$.
2. $\forall \mathfrak{n}_i \in P$ compute $O_P(\mathfrak{n}_i)$.
3. Construct finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$ accepting all prefixes of a string $P$:
   a) $\mathcal{Q} \leftarrow \{q_0, q_1, \ldots, q_t\}$,
   b) $\mathcal{A} \leftarrow \{//\}.\mathcal{A}'$,
   c) $q_0 \leftarrow \{0\}$ and
      $\forall \mathfrak{n}_i$, where $i \leftarrow 1, 2, \ldots, t$:
        (i) set state $q_i \leftarrow O_P(\mathfrak{n}_i)$
       (ii) add $\delta(q_{i-1}, //label(\mathfrak{n}_i)) \leftarrow q_i$
      (iii) $O_P(\mathfrak{n}_i) \leftarrow ButFirst(O_P(\mathfrak{n}_i))$
   d) $F \leftarrow \{q_1, q_2, \ldots, q_t\}$.
4. Insert additional transitions into the automaton $\mathcal{M}$:
   a) $\forall i \in \{0, 1, \ldots, t-1\} \, \forall l \in \mathcal{A}'$:
        (i) add $\delta(q_i, //l) \leftarrow q_s$, if there exists such $s > i$ where
            $\delta(q_{s-1}, //l) = q_s \wedge \neg\exists r < s : \delta(q_{r-1}, //l) = q_r$
       (ii) $\delta(q_i, //l) \leftarrow \emptyset$ otherwise.   ♢
```

Example 6.19. Consider the XML tree model $\mathcal{T}$ illustrated in Figure 6.1 and its corresponding string paths set $\mathcal{P}_\mathcal{T}$ described in Example 6.6. Transition diagrams of deterministic subsequence automata for each string path contained in $\mathcal{P}_\mathcal{T}$ constructed by Algorithm 6.18 are evidently the same as resulting deterministic automata built by Algorithm 6.14. We illustrate the automata in Figures 6.5 and 6.6. ♠

Algorithm 6.20. Construction of a Tree String Path Subsequences Automaton for an XML document $\mathcal{D}$ and its corresponding XML tree model $\mathcal{T}$.
Input: A string paths set $\mathcal{P}_\mathcal{T} = \{P_1, P_2, \ldots P_k\}$.
Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{E}(\mathcal{D}), \delta, 0, F)$ accepting all $XP^{\{//, name-test\}}$ queries of $\mathcal{D}$.
Method:

```
1. Construct finite automata $\mathcal{M}_i = (\mathcal{Q}_i, \mathcal{A}_i, \delta_i, 0, F_i)$ accepting a set of non-empty
   subsequences of $P_i$ using Algorithm 6.18.
2. Construct deterministic Tree String Path Subsequences Automaton $\mathcal{M} =
   (\mathcal{Q}, \mathcal{E}(\mathcal{D}), \delta, 0, F)$ accepting a set of all non-empty subsequences of each $P_i$ us-
   ing product construction.   ♢
```

Example 6.21. Let $\mathcal{D}$ and $\mathcal{T}$ be an XML document and XML tree model from Example 6.2 and Figure 6.1, respectively. The corresponding Tree String Path Subsequences Automaton accepting all $XP^{\{//, name-test\}}$ queries of $\mathcal{D}$, constructed by Algorithm 6.20, is illustrated in Figure 6.7. ♠

6.3.2 Discussion on Time and Space Complexities

TSPSA effectively supports the evaluation of all $XP^{\{//, name-test\}}$ queries of an XML document $\mathcal{D}$. The number of such queries is exponential in the number of nodes of the XML tree model $\mathcal{T}$ of $\mathcal{D}$.

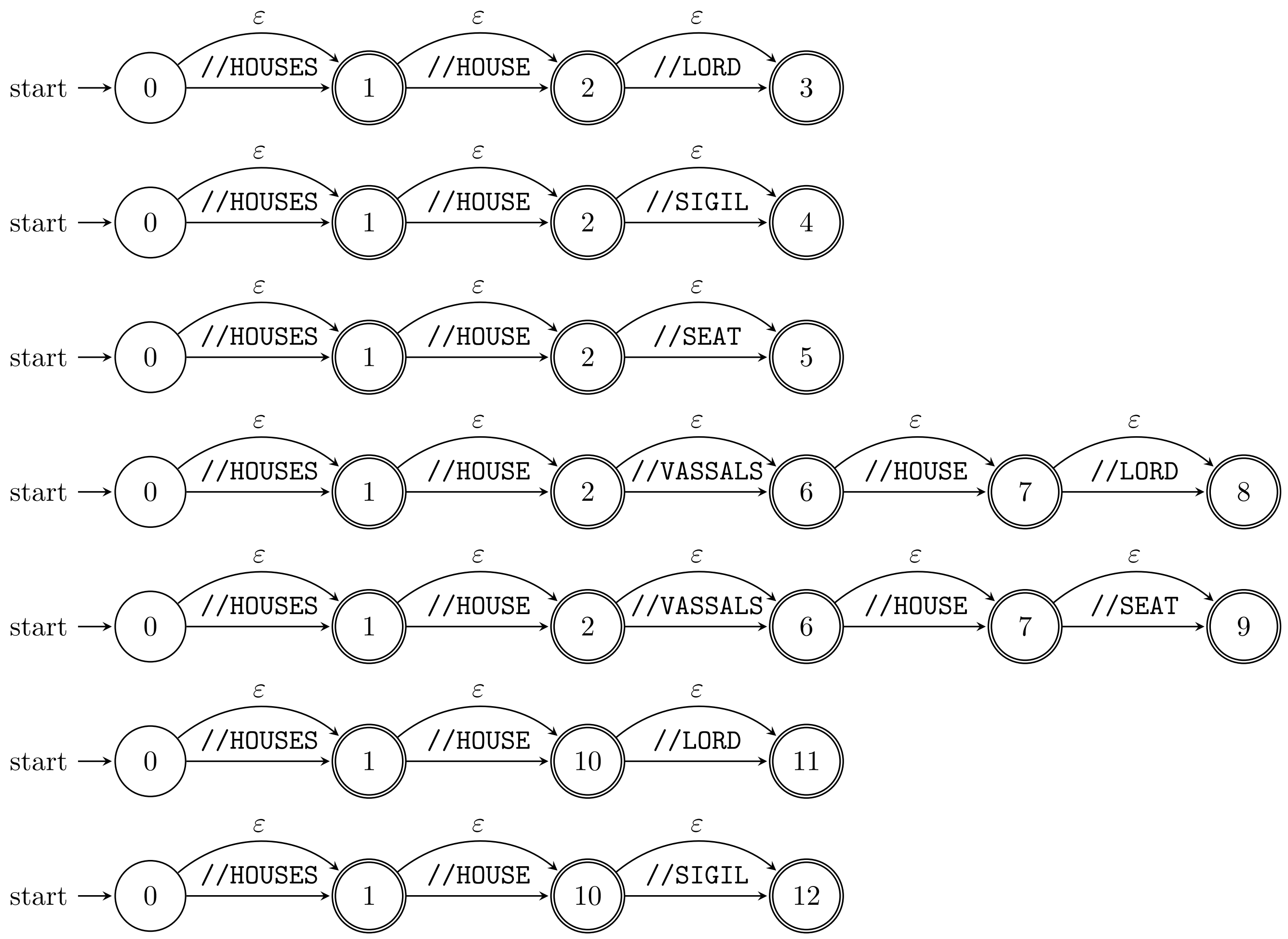


Figure 6.4. Subsequence automata

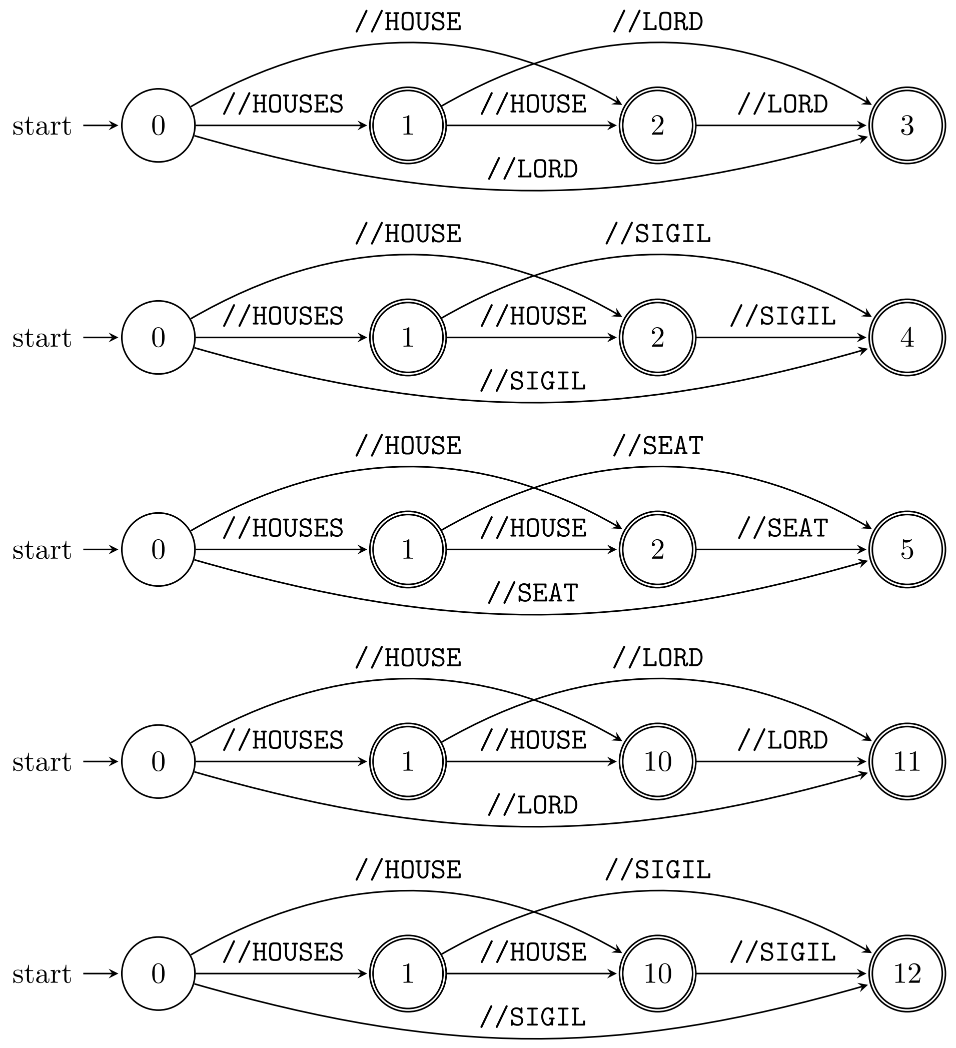


Figure 6.5. Deterministic subsequence automata

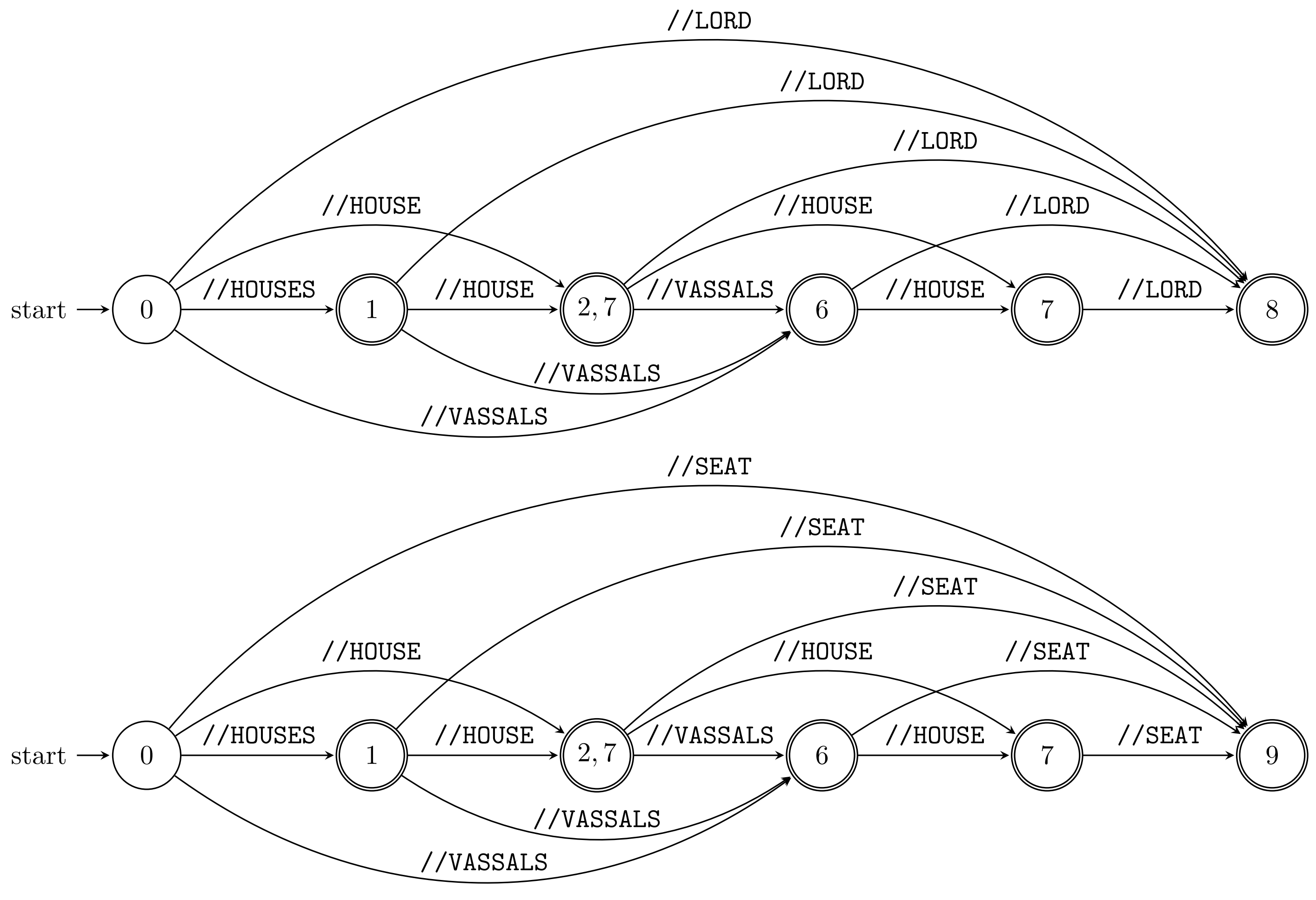


Figure 6.6. Deterministic subsequence automata

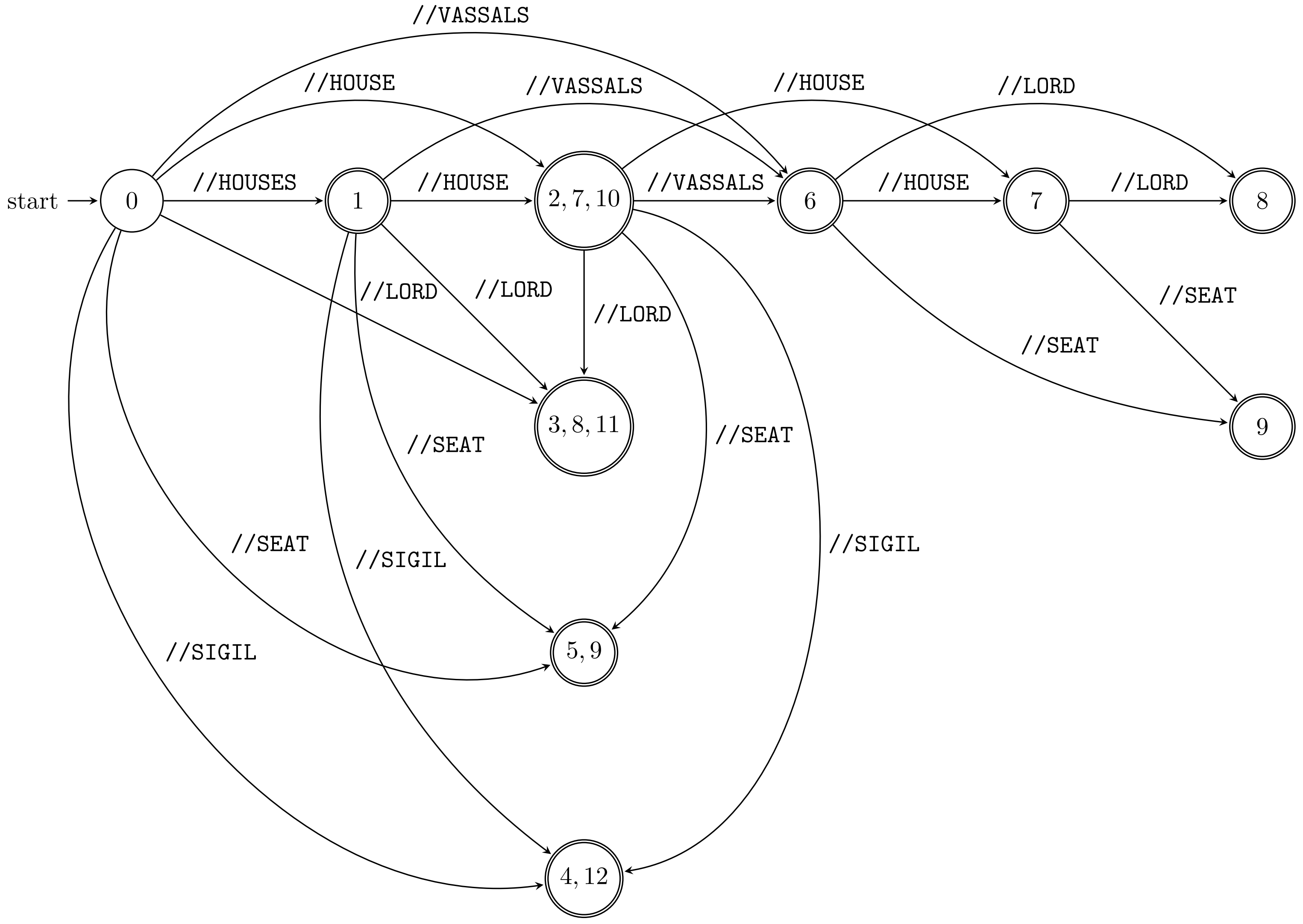


Figure 6.7. Tree String Path Subsequences Automaton

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

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

.

Basically, each state of TSPSA corresponds to an answer of a single query or a collection of queries. Although the number of different queries accepted by TSPSA is exponential, for most of XML documents a lot of the queries is equivalent (i.e., their result sets of elements are equal).

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

The containment and equivalence problems for a fragment of XPath query language was studied in [40–41]. For $XP^{\{//,name-test\}}$ a PTIME containment algorithm was provided by Buneman et al. in [42].

From another point of view, we can examine the number of states of a TSPSA as a size of DASG for a set of strings. For k strings of length h, the number of states can be trivially bounded by $\mathcal{O}(h^k)$ (size of a product of k automata with $\mathcal{O}(h)$ states). The running time for a query of length m becomes $\mathcal{O}(m)$. The lower bound for $k > 2$ strings in not known, while Crochemore and Troníček in [39] showed that $\Omega(h^2)$ states are required for $k = 2$ at the worst case. Considering an XML index problem, k is a number of leaves in an XML tree model $\mathcal{T}$ and h is its height. The construction of the TSPSA may be characterised as follows:

- Standard subset construction is needed, in case we want to return the answers of queries. Otherwise, we are just able to decide, whether given query has a non-empty result set of elements.
- The set of strings is rather specific. Thanks to the branching tree structure, we can expect common prefixes in the set of strings, i.e., less number of states in resulting automaton.

When space is more crucial, we can construct subsequence automata for each string path in $\mathcal{P}_{\mathcal{T}}$. Given a query of length m, we have to only traverse all subsequence automata simultaneously, an return the union of resulting subsets of automata that accept the query as the answer. It obviously runs in $O(km)$.

Definition 6.22. (Tree level). Let T be a rooted tree. *Tree level* L is a set of nodes such that $\forall \mathfrak{n} \in L$ the number of edges from $\mathfrak{n}$ to the root node r is equal. ♣

Definition 6.23. (State level). Let $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, q_0, F)$ be a total deterministic subsequence automaton. State level s of state q is a maximal number of transitions from the initial state q_0 to q. ♣

Definition 6.24. (Level property). Let T be a rooted tree and $\mathfrak{n}_1, \mathfrak{n}_2$ be two nodes of T with labels l_1, l_2, respectively. Level property (l-property): If $l_1 = l_2$ then $\mathfrak{n}_1, \mathfrak{n}_2$ must be in the same tree level. ♣

Theorem 6.25. Maximal number of states of the deterministic Tree String Path Subsequences Automaton constructed for an XML tree model $\mathcal{T}$ that satisfies the l-property is $\mathcal{O}(h \cdot 2^k)$. ♡

Proof. Let $\mathcal{T}$ be an XML tree model of height h and k leaves. Therefore, there is k string paths in $\mathcal{T}$, for which we construct a set S of k total deterministic subsequence automata of no more than h states each. We can run all automata "in parallel", by remembering the states of all automata by constructing k-tuples q while reading the input. This is achieved by the product construction. This way we construct the Tree String Path Subsequences Automaton $\mathcal{M}$ for $\mathcal{T}$.

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

6.4 Tree Paths Automaton

Definition 6.26. Let $\mathcal{D}$ be an XML document. A Tree Paths Automaton accepts all $XP^{\{/,//,name-test\}}$ queries of $\mathcal{D}$. ♣

A *Tree Paths Automaton* (TPA) is designed to process a significant fragment of XPath queries, which may use any combination of child (/) and descendant-or-self (//) axis (i.e., $XP^{\{/,//,name-test\}}$). The proposed index combines principles of the formerly introduced automata (i.e., TSPA and TSPSA). Note, that both $XP^{\{/,name-test\}}$ and $XP^{\{//,name-test\}}$ queries are subsets of $XP^{\{/,//,name-test\}}$ queries. Hence, both of them are naturally supported by TPA.

6.4.1 Building the Tree Paths Automaton

Unfortunately, no standard automata composition operations allow us to easily combine TSPA and TSPSA to construct TPA. Therefore, one approach is to reuse the idea of string paths. First, define appropriate automaton and construct it for each string path. Second, compose the constructed automata and finally determinise the result.

Another applicable approach is to build a nondeterministic Tree Paths Automaton without need of describing an XML tree model by means of its linear fragments (i.e., string paths). Then a standard determinisation algorithm based on subset construction is available to produce desired deterministic Tree Paths Automaton.

Before we delve into details of the construction algorithm, we shall first introduce some working data structures called *Arity array*, *Label array* and *Set of occurrences of an element label in an XML tree model*. We also need a structure for finding positions of ends of subtrees called *Subtree jump table* proposed in [16] (for ranked trees).

Definition 6.27. (Label array). Let $\mathcal{T}$ be an XML tree model with n nodes. Then $Label_{\mathcal{T}}$ is a Label array of $\mathcal{T}$, where $\forall i \in \{1, 2, \ldots, n\}$, $Label_{\mathcal{T}}[i]$ contains the label of node i (i.e., $id(\mathfrak{n}) = i$). ♣

Definition 6.28. (Arity array). Let $\mathcal{T}$ be an XML tree model with n nodes. Then $Arity_{\mathcal{T}}$ is an Arity array for $\mathcal{T}$, where $\forall i \in \{1, 2, \ldots, n\}$, $Arity_{\mathcal{T}}[i]$ contains the number of children of node i (i.e., $id(\mathfrak{n}) = i$). ♣

Definition 6.29. (Subtree jump table for an XML tree model). Let $\mathcal{T}$ be an XML tree model with n nodes, where each node is assigned an identifier i according to preorder numbering scheme. A Subtree jump table for $\mathcal{T}$, denoted $SJT_{\mathcal{T}}$, is defined as a mapping from set $\{1, \dots, n\}$ into set $\{2, \dots, n+1\}$. $SJT_{\mathcal{T}}[i] = j$, $1 \leq i < j \leq n+1$ if there exists a subtree of $\mathcal{T}$ rooted at node i with $j-1$ as the rightmost leaf. ♣

Definition 6.30. (Set of occurences of an element label in an XML tree model). Let $\mathcal{T}$ and $Label_{\mathcal{T}}$ be and XML tree model with n nodes and relevant *Label* array, respectively. A set of occurrences of an element label l in $\mathcal{T}$ is a totally ordered set $O_{\mathcal{T}}(l) = \{i \,|\, Label[i] = l,\ i = 1, 2, \dots, n\}$. The ordering is equal to ordering of the array indexes (i.e., prefix identifiers) as natural numbers. ♣

Example 6.31. Let $\mathcal{D}$ and $\mathcal{T}$ be an XML document and XML tree model from Example 6.2 and Figure 6.1, respectively. Table 6.1 shows corresponding structures $Label_{\mathcal{T}}$, $Arity_{\mathcal{T}}$ and $SJT_{\mathcal{T}}$. Table 6.2 describes sets of occurrences of all element labels in $\mathcal{E}(\mathcal{D})$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$Label_{\mathcal{T}}[i]$	HOUSES	HOUSE	LORD	SIGIL	SEAT	VASSALS	HOUSE	LORD	SEAT	HOUSE	LORD	SIGIL
$Arity_{\mathcal{T}}[i]$	2	4	0	0	0	1	2	0	0	2	0	0
$SJT_{\mathcal{T}}[i]$	13	10	4	5	6	10	10	9	10	13	12	13

Table 6.1. Working data structures for Tree Paths Automaton construction

l	HOUSES	HOUSE	LORD	SIGIL	SEAT	VASSALS
$O_{\mathcal{T}}(l)$	$\{1\}$	$\{2, 7, 10\}$	$\{3, 8, 11\}$	$\{4, 12\}$	$\{5, 9\}$	$\{6\}$

Table 6.2. Set of occurrences of all element labels $l \in \mathcal{E}(\mathcal{D})$ of XML document $\mathcal{D}$ from Example 2.42

♠

The working structures need to be set only once during the preprocessing phase. When the Tree Paths Automaton is built, other data structures are no longer needed. Since construction of *Arity* and *Label array* together with computing the *Set of occurrences of all element labels in an XML alphabet* is trivial, we include a building method for *Subtree jump table* only described by Algorithm 6.32.

Algorithm 6.32. Construction of a Subtree jump table for an XML tree model $\mathcal{T}$.
Input: $Arity_{\mathcal{T}}$ - array of nodes arity, $rootIndex$ - index of current node, $SJT_{\mathcal{T}}$ - reference to an empty Subtree jump table
Output: $exitIndex$ - exit index, $SJT_{\mathcal{T}}$ - Subtree jump table
Method:

```
1. $index \leftarrow rootIndex + 1$
2. $\forall i,\ i \leftarrow 1, \dots Arity_{\mathcal{T}}[rootIndex]$
  (i) $index \leftarrow exitIndex$ of Algorithm 6.32 where $rootIndex \leftarrow index$
3. $SJT[rootIndex] \leftarrow index$
4. $exitIndex \leftarrow index$
```
◊

Algorithm 6.33. Construction of a Tree Paths Automaton for an XML document $\mathcal{D}$.
Input: XML data tree $\mathcal{T}$ for $\mathcal{D}$ with n nodes, $Arity_{\mathcal{T}}$, $Label_{\mathcal{T}}$, $SJT_{\mathcal{T}}$ and $O_{\mathcal{T}}(l)$, $\forall l \in \mathcal{E}(\mathcal{D})$.

Output: A deterministic finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, 0, F)$ accepting all $XP^{\{/,//,name-test\}}$ queries of $\mathcal{D}$.
Method:

```
1. Construct a finite automaton $\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \delta, 0, F)$ as follows:
   a) $\mathcal{Q} \leftarrow \{0, 1, 2, \ldots, n\}$,
   b) $\mathcal{A} \leftarrow \{/, //\}.\mathcal{E}(\mathcal{D})$,
   c) $\delta(0, /Label_{\mathcal{T}}[1]) \leftarrow \{1\}$,
   d) $\forall q$, where $q \leftarrow 1, 2, \ldots, n-1$,
      (i) $i \leftarrow q + 1$
      (ii) repeat $Arity_{\mathcal{T}}[q]$ times:
         (i) $\delta(q, /Label_{\mathcal{T}}[i]) \leftarrow \delta(q, /Label_{\mathcal{T}}[i]) \cup \{i\}$,
         (ii) $i \leftarrow SJT_{\mathcal{T}}[i]$.
   e) $F \leftarrow \{1, 2, \ldots, n\}$.
2. Insert additional transitions into the automaton $\mathcal{M}$:
   a) $\forall q \in \{0, 1, \ldots, n-1\} \, \forall l \in \mathcal{E}(\mathcal{D})$:
      (i) $\delta(q, //Label_{\mathcal{T}}[j]) \leftarrow \{j \mid j \in O_{\mathcal{T}}(l) \land q < j < SJT_{\mathcal{T}}[q]\}$.
3. Use standard determinisation algorithm based on subset construction to get determin-
   istic automaton $\mathcal{M}$.                                                   ♢
```

Example 6.34. Let $\mathcal{D}$ and $\mathcal{T}$ be an XML document and XML tree model from Example 6.2 and Figure 6.1, respectively. The transition diagram of nondeterministic Tree Paths Automaton is illustrated in Figure 6.8. Figure 6.9 shows transition diagram of corresponding deterministic Tree Paths Automaton. In these figures, the transition rule $\delta(p,$`/[/]LABEL`$) = q$ represents two transitions leading from state p to state q: $\delta(p,$`/LABEL`$) = q$ and $\delta(p,$`//LABEL`$) = q$. ♠

6.4.2 Discussion on Time and Space Complexities

A Tree Paths Automaton is designed to effectively process all $XP^{\{/,//,name-test\}}$ queries of $\mathcal{D}$. The major issue is the size of the deterministic automaton, which, in theory, can be exponential in the size of the XML document being indexed. Despite our effort, the problem of a tight upper bound on the number of states of TPA remains open.

However, we provide a series of experiments in following chapter to show that the determinisation will result in a small number of states although the number of queries accepted by the *Tree Paths Automaton* is exponential in n (e.g., $\mathcal{O}(2.62^n)$ [34] for a linear XML tree with n nodes).

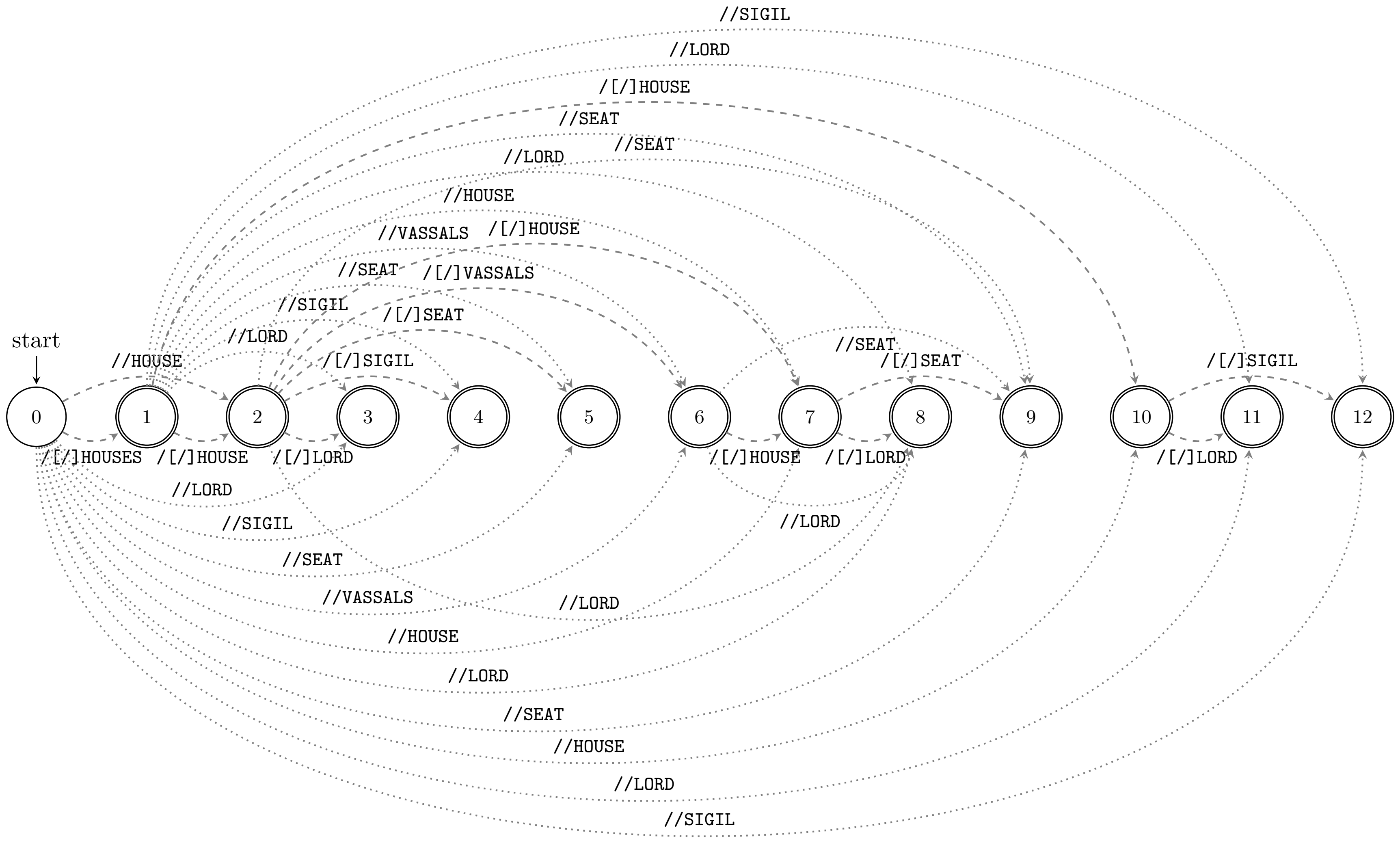


Figure 6.8. Nondeterministic Tree Paths Automaton

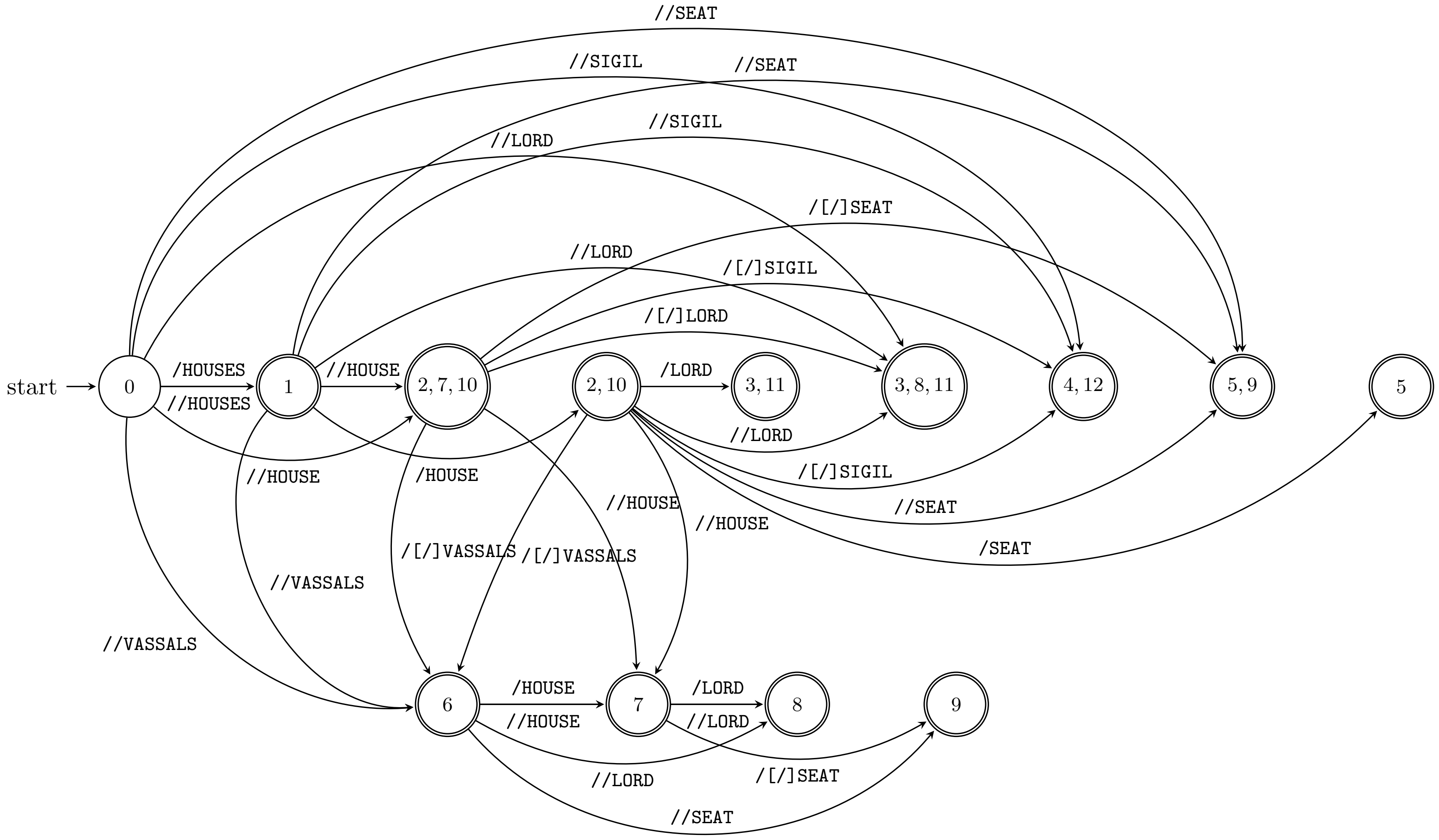


Figure 6.9. Deterministic Tree Paths Automaton

Chapter 7
Implementation

In this chapter we deal with the implementation of one of the proposed methods for indexing XML documents introduced in previous chapter. Since a Tree Paths Automaton is designed to process the largest fragment of XPath queries and covers the power of both Tree String Paths and Tree String Path Subsequences Automaton, we focus on its realization only.

7.1 System Architecture

The XML index software is developed using Java SE, JDK 8u45 in the NetBeans IDE 8.0.2 [43] and is designed as *Java Class Library* called `tpalib`. The system architecture of the `tpalib` is illustrated in Figure 7.1. The library consists of three virtual parts called JDOM, Index Builder and XML Data Index.

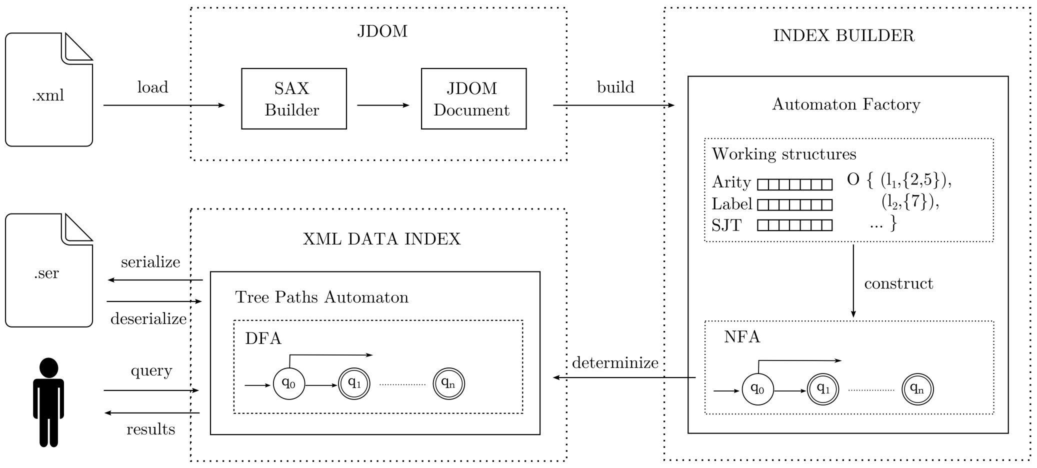


Figure 7.1. System architecture of `tpalib`

7.1.1 JDOM

An XML document is processed using the first main part of the system architecture called JDOM [44], an open source Java-based document object model for XML that was designed specifically for the Java platform so that it can take advantage of its language features. The `tpalib` proposes a `DocumentLoader` class (see Figure 7.2) with `load` method accepting as an argument a path to the XML document represented as `String`. The method reads and parses the XML document using `SAXBuilder` class (SAX stands for Simple API for XML) and builds a JDOM `Document` that returns to a user. The `Document` object is further processed in the next part of the system, where the XML Data Index is being constructed.

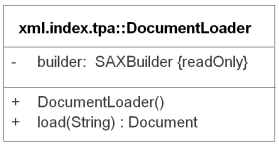

xml.index.tpa::DocumentLoader
- builder: SAXBuilder {readOnly}
+ DocumentLoader()
+ load(String) : Document

Figure 7.2. `DocumentLoader` class

7.1.2 Index Builder

The Index Builder comes as the second main part of the system architecture. As the name indicates, the Index Builder is responsible for a construction of the XML Data Index. It disposes of `AutomatonFactory` class (see Figure 7.3) that presents a `build` method accepting the JDOM `Document` object, built in previous step, as its argument. The method constructs the XML Data Index and returns it as an instance of `TreePathsAutomaton` class to a user. The construction process can be divided into three following phases:

1. working structures initialization,
2. NFA construction,
3. NFA determinisation.

At first, a preorder traversal of the JDOM `Document` object is used to initialize the working structures (see section 6.4.1) needed for nondeterministic finite automaton construction. We realize *Arity* and *Label* arrays and *SJT* as a single array of objects. The objects are accessible by their array index that corresponds to a preorder identifier of the XML element they represent.

We implement the objects as instances of `Node` class (see Figure 7.3) defined within `AutomatonFactory` class. The inner class stores information about label, arity and subtree jump table value of an element in its attributes. The sets of occurrences of all element labels in XML alphabet are implemented as a mapping from each of element labels to a set containing preorder identifiers of XML elements sharing the label.

Based on the information included in the working structures, second phase of the construction process deals with building a nondeterministic Tree Paths Automaton as described in steps 2 and 3 in Algorithm 6.33. We represent the transition function of the finite automaton as a mapping from each of states to a collection of its transitions implemented as a mapping from edge labels to their corresponding set of target states (i.e., `HashMap<Integer, HashMap<EdgeLabel, TreeSet<Integer>>>`).

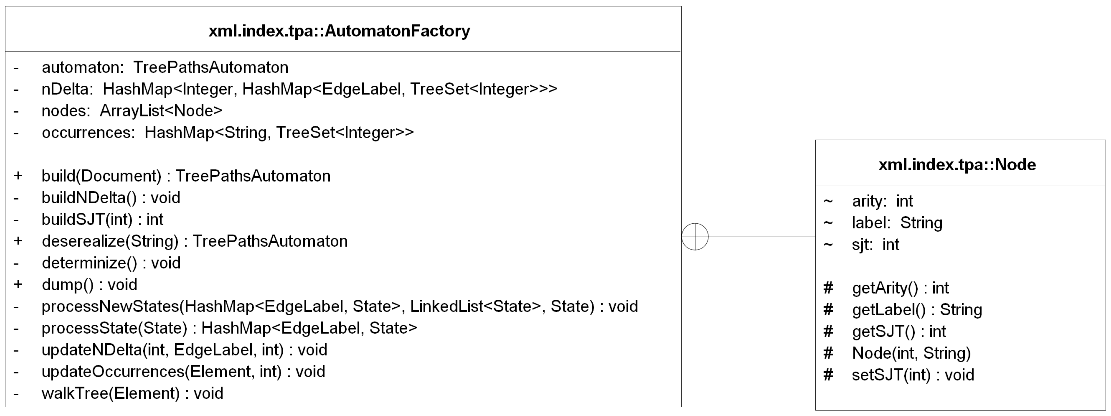

xml.index.tpa::AutomatonFactory
- automaton: TreePathsAutomaton
- nDelta: HashMap<Integer, HashMap<EdgeLabel, TreeSet<Integer>>>
- nodes: ArrayList<Node>
- occurrences: HashMap<String, TreeSet<Integer>>
+ build(Document) : TreePathsAutomaton
- buildNDelta() : void
- buildSJT(int) : int
+ deserealize(String) : TreePathsAutomaton
- determinize() : void
+ dump() : void
- processNewStates(HashMap<EdgeLabel, State>, LinkedList<State>, State) : void
- processState(State) : HashMap<EdgeLabel, State>
- updateNDelta(int, EdgeLabel, int) : void
- updateOccurrences(Element, int) : void
- walkTree(Element) : void
xml.index.tpa::Node
~ arity: int
~ label: String
~ sjt: int
# getArity() : int
# getLabel() : String
# getSJT() : int
# Node(int, String)
# setSJT(int) : void

Figure 7.3. `AutomatonFactory` and `Node` class

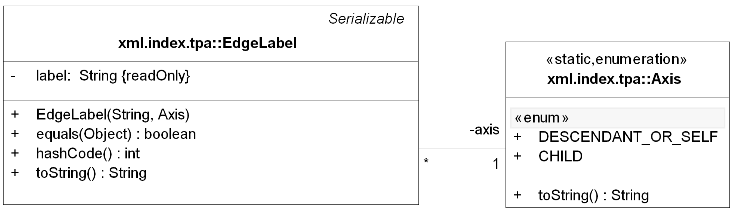


Figure 7.4. `EdgeLabel` class and `Axis` enum

Transition labels are modelled as instances of `EdgeLabel` class (see Figure 7.4) containing information about axis and label used. Other information about automaton such as set of states, alphabet, initial state or a set of final states are either directly given by the definition of the Tree Paths Automaton or insignificant for our purpose.

Eventually, a standard determinisation algorithm based on subset construction is applied to the constructed NFA to get a final XML Data Index that is returned as an instance of `TreePathsAutomaton` class to a user.

7.1.3 XML Data Index

The XML Data Index stands for the third main part of the system architecture. We realize the deterministic Tree Paths Automaton as an instance of `TreePathsAutomaton` class (see Figure 7.5) constructed by `AutomatonFactory` in previous step.

Since the index construction is fairly a time-consuming process, shorthand methods for object serialization and deserialization are available to a user once an instance of `TreePathsAutomaton` class is built. Therefore, next time the index of the XML document is needed, the `TreePathAutomaton` object can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Information about deterministic Tree Paths Automaton are stored similarly as in the case of its nondeterministic version. Again, the transition function is implemented as a mapping, however, in this case from `State` (see Figure 7.6) objects to collections of transitions. Individual transitions are realized as a mapping from transition labels to `State` objects. Transition labels are once more modelled as instances of `EdgeLabel` class. The `State` class represents a state of deterministic finite automaton and encapsulates a content of its d-subset. Other information about finite automaton such as alphabet, initial state or a set of final states are again either directly given by the definition of the Tree Paths Automaton or insignificant for our purpose.

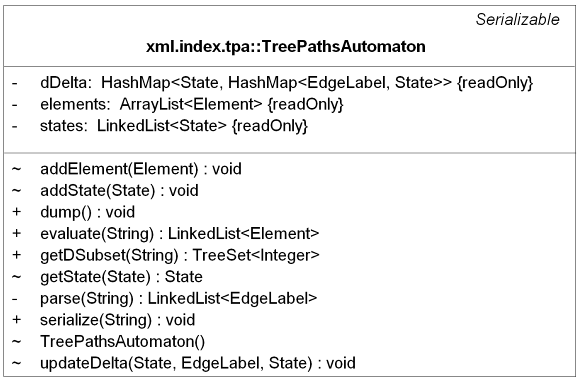

Serializable
xml.index.tpa::TreePathsAutomaton
- dDelta: HashMap<State, HashMap<EdgeLabel, State>> {readOnly}
- elements: ArrayList<Element> {readOnly}
- states: LinkedList<State> {readOnly}
~ addElement(Element) : void
~ addState(State) : void
+ dump() : void
+ evaluate(String) : LinkedList<Element>
+ getDSubset(String) : TreeSet<Integer>
~ getState(State) : State
- parse(String) : LinkedList<EdgeLabel>
+ serialize(String) : void
~ TreePathsAutomaton()
~ updateDelta(State, EdgeLabel, State) : void

Figure 7.5. `TreePathsAutomaton` class

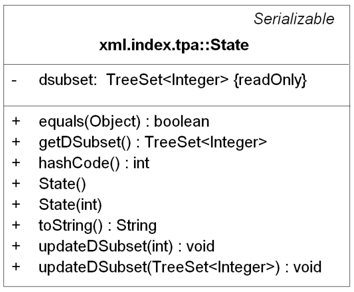

Serializable
xml.index.tpa::State
- dsubset: TreeSet<Integer> {readOnly}
+ equals(Object) : boolean
+ getDSubset() : TreeSet<Integer>
+ hashCode() : int
+ State()
+ State(int)
+ toString() : String
+ updateDSubset(int) : void
+ updateDSubset(TreeSet<Integer>) : void

Figure 7.6. `State` class

7.2 Query Processing

As a query processor, a user uses directly the `TreePathsAutomaton` object that accepts and evaluates queries represented as `String` (e.g., `"/site//regions"`). There are three methods provided by the `TreePathsAutomaton` regarding the query processing:

- `TreeSet<Integer> getDSubset(String query)` – parses the query and returns an ordered set of preorder identifiers of elements in an XML document satisfying the query,
- `LinkedList<Element> evaluate(String query)` – parses the query and returns list of elements, or more precisely `Element` objects, in an XML document satisfying the query,
- `LinkedList<Element> evaluate(TreeSet<Integer> dsubset)` – gets d-subset and maps it to appropriate elements (`Element` objects) in the XML document and returns them in a list.

The parsing process split a query into a sequence of automaton inputs (i.e., tokens starting with either `/` or `//` where an element label follows). Eventually, the query is evaluated by running the deterministic automaton and the required result is returned.

Chapter 8
Experimental Evaluation

This chapter explores the performance of one of the proposed methods for indexing XML documents based on a finite automaton construction. Since a Tree Paths Automaton is designed to process the largest fragment of XPath queries and covers the power of both Tree String Paths and Tree String Path Subsequences Automaton, we focus on its experimental evaluation only. The Tree Paths Automaton is introduced in Section 6.4 and its implementation details are described in previous chapter.

We first present a testing environment for our experiments and characteristics of selected XML data sets. Then we study the space requirements by measuring a size of file with serialized `TreePathsAutomaton` object and the time it takes to build such deterministic automaton. Finally, we go on to a performance study over XPath queries that are supported by our index.

8.1 Experimental Setup

The XML index software is developed using Java SE, JDK 8u45 and its implementation is described in previous chapter in details. Our experiments are conducted under the environment of Intel Core i7 CPU @ 2.00 GHz, 8.0 GB RAM and 240 GB SSD disk with Windows 8.1 Pro operation system running.

8.1.1 XML Data Sets

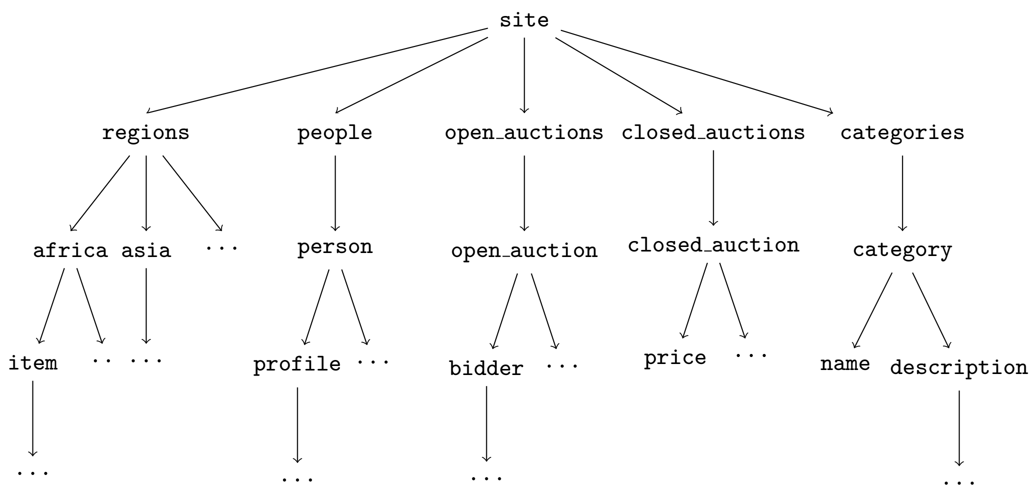


Figure 8.1. Partial schema of XMark data sets

For our experimental evaluation, we selected XML benchmark XMark data sets generated by `xmlgen` [45]. The XMark data set is a single record with a very large and fairly complicated tree structure with a maximal depth of 11 and average depth of 4.5. The XML data models an on-line auction site. The main element relationships are illustrated for convenience in Figure 8.1. The tag names are largely self-explanatory.

Table 8.1 describes relevant characteristics of generated data sets. First column define keys of data sets. Second column shows names of generated XML files. The next column are XMark `xmlgen` scaling factors [46] of the document, float value, where 0 produces the "minimal document". The fourth column contains numbers of element nodes in the XML file and finally, the last column shows the size of files in megabytes.

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

Table 8.1. Characteristics of XMark benchmark files

8.2 Index Size and Construction Time

In this section we study the space requirements of the index structure by measuring the size of the file with serialized `TreePathsAutomaton` object and the time it takes to build such deterministic automaton. Table 8.2 shows the experimental results on both index construction time and size for generated XMark data sets.

The time of index creating is insignificant when we consider just files with less than 100,000 elements. However, the time cost on 60 MB data set with over 800,000 elements is more than an hour. We detected that the most time-consuming phase is the construction of nondeterministic Tree Paths Automaton. Therefore, it could be the crucial part for our future software optimization.

As for the index size, it should be noted that even though the index occupies a great deal of space, the ratio of index size to original XML data size stays linear. Table 8.3 shows that the index data of Tree Paths Automaton is about 2.5 times of the original document size.

Key	Index Construction Time [sec]	Index Size [MB]
D_1	0.3	0.08
D_2	0.5	0.30
D_3	2.8	1.35
D_4	6.1	2.68
D_5	6,780.0	129.00

Table 8.2. Experimental results on construction time and size

Key	Index Size / XML File Size
D_1	2.60
D_2	3.00
D_3	2.25
D_4	2.23
D_5	2.22

Table 8.3. Ratio of index size to XML file size

8.3 Performance on Query Processing

In this section we report experimental results conducted on Tree Paths Automaton in comparison with well-known reference implementation called Saxon [47]. The Saxon package is a collection of Java tools for processing XML documents. One of the main components is an XPath processor accessible to applications via a supplied API. This supports both XPath 2.0 and XPath 3.0, and it can be used in backwards-compatibility mode to evaluate XPath 1.0 expressions.

Measurements reflect query processing time only, equivalent to the `evaluate()` method in both Saxon and `tpalib`. Hence, document loading cost and query parsing cost (`compile()` method in Saxon) have been excluded from the measurements.

Table 8.4 lists 9 sample queries we use for the experimental evaluation. We split them into categories made up of three queries each, depend on type of axes used. First Q_1–Q_3 queries contain child axis only, Q_4–Q_6 are created of descendant-or-self axis only and last Q_7–Q_9 queries use combination of both axes. Each of categories lists its queries with ascending length. Numbers of elements satisfying individual queries in each of data sets are shown in Table 8.5.

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

Table 8.4. Set of queries used in performance analysis

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

Table 8.5. Numbers of elements satisfying queries in test data sets

Tables 8.6 and 8.7 summarize the experimental results of `tpalib` and Saxon, respectively. Each of cells in row r and column c contains a time score in milliseconds for running a query Q_c on data set D_r.

In Figure 8.2, the elapsed time for processing the queries in Table 8.4 on individual data sets in Table 8.2, using both `tpalib` and Saxon are plotted using logarithmic scale. The x-axis represents the data sets, while the y-axis shows the response time in milliseconds. We use light blue dashed lines to display Saxon results, whereas `tpalib` score is depicted as dark blue solid lines.

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

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
D_1	2.08	1.45	1.74	1.98	2.02	1.98	1.35	1.51	1.64
D_2	1.88	1.46	1.55	1.52	1.99	1.89	2.20	1.99	1.66
D_3	1.39	2.12	2.04	1.58	1.36	1.36	1.28	1.58	1.62
D_4	1.25	1.41	1.47	1.52	1.37	1.51	1.40	1.45	1.45
D_5	1.54	2.33	1.24	2.76	2.14	1.03	1.25	2.35	1.27

Table 8.6. `tpalib` experimental evaluation in milliseconds

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
D_1	4.08	4.03	4.12	4.72	6.33	7.57	4.68	5.45	7.70
D_2	3.75	7.09	5.83	10.92	7.91	8.44	9.76	10.43	8.83
D_3	4.20	7.75	5.97	15.44	15.26	13.78	9.76	12.56	14.39
D_4	4.02	9.54	8.40	20.07	20.31	23.46	14.24	12.56	16.15
D_5	3.27	20.76	16.82	76.72	84.62	59.26	28.34	82.36	53.88

Table 8.7. Saxon experimental evaluation in milliseconds

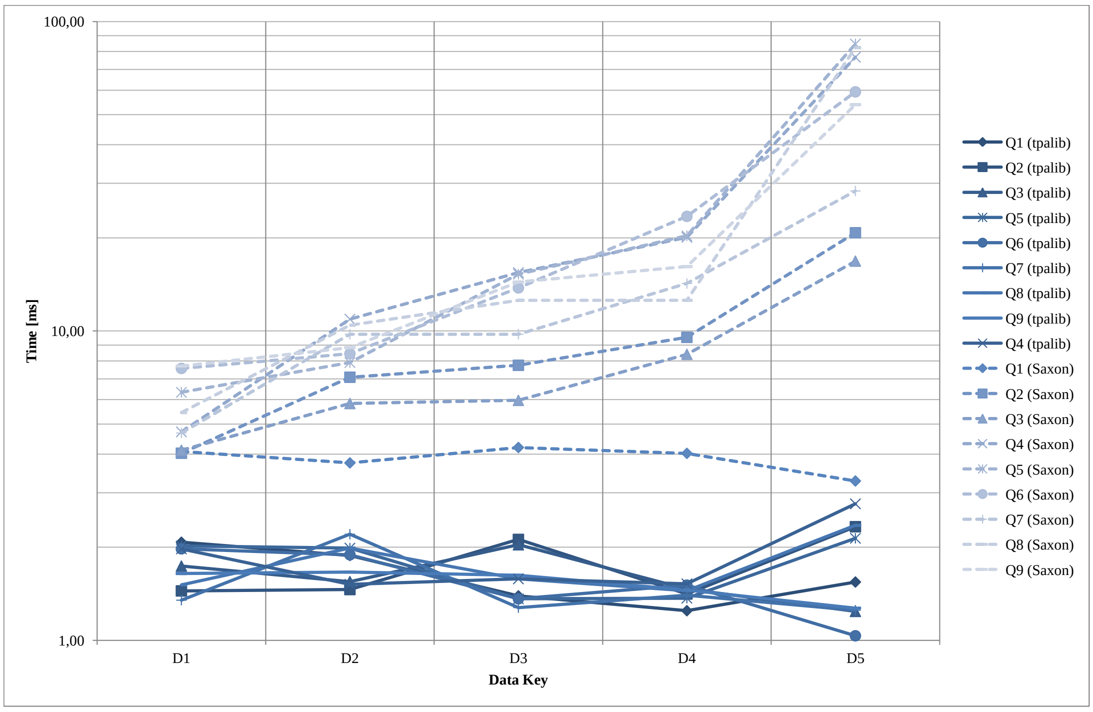


Figure 8.2. Performance comparison of `tpalib` and Saxon (logarithmic scale)

Chapter 9
Conclusions

In this chapter we summarize this thesis and conclude its results and contributions. We also present a list of open problems and propose future research directions and recommendations for future work to extend the research presented in this thesis.

9.1 Summary of the Thesis

In the introductory Chapter 1, the XML index problem is stated along with motivation and objectives. In Chapter 2, we provide the necessary theoretical background, i.e., notations used through this thesis along with essential definitions concerning trees and theory of formal languages and automata. At the end of this chapter, a brief introduction to XML and XPath is also given. We illustrate relevant XML and XPath features by means of examples rather than using an exhaustive detailed description. We discuss the relationship between arbology and XPath in Chapter 3.

In the next two Chapters 4 and 5, we review existing techniques for indexing texts and trees and most importantly we study the state-of-the-art methods for indexing XML documents. We can classify the XML indexes into several categories such as: graph-based methods, sequence-based methods, node coding or adaptive methods. We have introduced some of the index structures in detail.

The following chapters present a new and simple method of indexing XML documents for processing linear XPath expressions (i.e., $XP^{\{/,//,name-test\}}$) using deterministic finite automaton called Tree Paths Automaton. We also proposed another two indexing techniques based on finite automata (i.e., Tree String Paths Automaton and Tree String Path Subsequences Automaton), aimed to assist in evaluating paths queries with either child (/) or descendant-or-self (//) axis only. The implementation details of Tree Paths Automaton are given in Chapter 7. We also provide experimental results to demonstrate the efficient processing of path queries by Tree Paths Automaton in Chapter 8.

9.2 Contributions of the Thesis

Contributions of this thesis are as follows:

- **Chapter 3**. This chapter discusses the relationship between arbology and XPath. Since the internal structure of an XML document can be represented as a labelled unranked tree, searching XML documents becomes a relevant field of study for arbology that is interested in processing tree data structures. We can also view an XPath query as a kind of a tree pattern. Thus, we may declare querying or searching XML document using XPath to be analogous problem of a tree pattern matching.

  We have also proposed a classification of XPath to be able to describe fragments of XPath queries with elegance. First, we can simply classify XPath queries by restricting the constructs available in query syntax (e.g., $XP^{\{/,//,*\}}$). Next, grouping queries according to orientation used for navigating through the XML tree structure is possible (e.g., down oriented queries). Considering the query tree structure, path and twig queries are other potential classes. Finally, we propose queries of arity $0, 1, 2, \dots, p$ to be respectively called boolean, unary, binary, ..., p-ary, where arity is the number of target nodes of an XPath query.

- **Chapter 6**. This chapter shows that automata can be used effectively for the purpose of indexing XML documents. A new and simple method of indexing XML documents for processing linear XPath expressions (i.e., $XP^{\{/,//,name-test\}}$) using deterministic finite automaton called Tree Paths Automaton is presented. We also proposed another two indexing techniques based on finite automata (i.e., Tree String Paths Automaton and Tree String Path Subsequences Automaton), aimed to assist in evaluating paths queries with either child (/) or descendant-or-self (//) axis only.

  Given a subject XML document $\mathcal{D}$ and its corresponding XML tree model $\mathcal{T}$ with n nodes, the tree is preprocessed and the index is constructed. The searching phase uses the index, reads an input query Q of size m and computes the list of positions of all occurrences of target nodes of Q in $\mathcal{T}$.

  All the proposed automata performed the searching in time $\mathcal{O}(m)$ and do not depend on n. Although the Tree String Path Subsequences Automaton supporting $XP^{\{//,*\}}$ fragment of XPath queries supports $\mathcal{O}(2^n)$ distinct queries, the number of states of deterministic automaton is $\mathcal{O}(h^k)$, where h is the height of the $\mathcal{T}$ and k is the number of its leaves. Moreover, we discuss that in case of indexing a common XML document the number of states of the deterministic finite automaton is at most $\mathcal{O}(h.2^l)$

- **Chapter 7**. This chapter deals with the implementation one of the proposed indexing method using Tree Paths Automaton. The XML index software is developed using Java SE and is designed as Java Class Library called `tpalib`.

- **Chapter 8**. This chapter explores the performance of the Tree Paths Automaton and provides experimental results to demonstrate the efficient processing of path queries. We report the results conducted on Tree Paths Automaton in comparison with Saxon as reference implementation. Measurements show that our method achieve better response time for all sample queries.

9.3 Future Work

There is a number of interesting open problems that we hope to explore in future work. For example:

- determine the tight upper bound on number of states and transitions for Tree Paths Automaton,
- develop incremental building algorithm for Tree Paths Automaton to efficiently adapt its structure to ever changing XML data source,
- propose an indexing method able to support multiple XML documents,
- study simulation techniques of nondeterministic finite automata and develop efficient simulation of Tree Paths Automaton or its implementation using dynamic programming,
- extend our method to support more complex queries (e.g., including attributes, wildcards, branching etc.),

- adjust our method to solve general tree index problem,
- study possibilities of combination the pushdown automata, which is a result of arbology, with the proposed methods,
- develop an indexing techniques for abstract syntax trees (AST) used in practical applications.

References

[1] al, Bray T. Paoli J. Sperberg-McQueen C. et. *Extensible Markup Language (XML) 1.0*.
http://www.w3.org/XML.

[2] Clark, J.; and S. DeRose. *XML Path Language (XPath) Version 1.0* [online]. [cit. 2015-02-03].
http://www.w3.org/TR/xpath.

[3] DeRose, S.. *XML Pointer Language (XPointer)* [online]. [cit. 2015-03-04].
http://www.w3.org/TR/xptr.

[4] DeRose, S.. *XML Linking Language (XLink) Version 1.0* [online]. [cit. 2015-03-04].
http://www.w3.org/TR/xlink.

[5] *The Prague Stringology Club* [online]. [cit. 2015-02-07].
http://www.stringology.org.

[6] Melichar, Bořivoj; Jan Janoušek; and Tomáš Flouri. *Introduction to Arbology* [online]. [cit. 2015-05-01].
https://edux.fit.cvut.cz/oppa/PI-ARB/prednasky/arbology.pdf.

[7] Rabin, "Michael O; and Dana Scott". "Finite automata and their decision problems". *IBM journal of research and development*. IBM, 1959, Vol. 3, No. 2, pp. 114–125.

[8] Neven, Frank. Automata, Logic, and XML. In: Julian Bradfield, ed. *Computer Science Logic*. Springer Berlin Heidelberg, 2002. pp. 2–26. Lecture Notes in Computer Science. ISBN 978-3-540-44240-0.

[9] Libkin, Leonid. Logics for Unranked Trees: An Overview. In: Luís Caires; Giuseppe Italiano; Luís Monteiro; Catuscia Palamidessi; and Moti Yung, eds. *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2005. pp. 35–50. Lecture Notes in Computer Science. ISBN 978-3-540-27580-0.

[10] Pokorný, Jaroslav. *XML Technologie – Principy a aplikace v praxi*. Praha: Grada Publishing, a.s., 2008. ISBN 978-80-247-2725-7.

[11] "BSI (British Standards Institution)". *BS 6154:1981 Method of defining – syntactic metalanguage*.

[12] Aluru, Srinivas. Text Indexing. In: Ming-Yang Kao, ed. *Encyclopedia of Algorithms*. Springer US, 2008. pp. 1-99. ISBN 978-0-387-30770-1.
http://dx.doi.org/10.1007/978-0-387-30162-4_422.

[13] Crochemore, M.; Ch. Hancart; and T. Locroq. *Algorithms on Strings*. Oxford University Press, 2002.

[14] Crochemore, Maxime; and W. Rytter. *Jewels of stringology*. World Scientific, 2002.

[15] Crochemore, M.; and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[16] Janoušek, Jan; Bořivoj Melichar; Radomír Polách; Martin Poliak; and Jan Trávníček. A Full and Linear Index of a Tree for Tree Patterns. In: Helmut Jürgensen; Juhani Karhumäki; and Alexander Okhotin, eds. *Descriptional Complexity*

of Formal Systems. Springer International Publishing, 2014. pp. 198–209. Lecture Notes in Computer Science. ISBN 978-3-319-09703-9.
`http://dx.doi.org/10.1007/978-3-319-09704-6_18`.

[17] Lecroq, T.. Structure for indexing texts. In: International PhD School in Formal Languages and Applications, Tarragona, 2006.

[18] Melichar, B.; J. Holub; and T. Polcar. *Text Searching Algorithms* [online]. Prague, 2005 [cit. 2015-05-01].
`http://www.stringology.org/athens/TextSearchingAlgorithms`.

[19] Ehrenfeucht, Andrzej; Ross M. McConnell; Nissa Osheim; and Sung-Whan Woo. Position Heaps: A Simple and Dynamic Text Indexing Data Structure. *J. of Discrete Algorithms*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., mar, 2011, Vol. 9, No. 1, pp. 100–121. ISSN 1570-8667. Available from DOI 10.1016/j.jda.2010.12.001.

[20] Coffman, E. G., Jr.; and J. Eve. File Structures Using Hashing Functions. *Commun. ACM*. New York, NY, USA: ACM, jul, 1970, Vol. 13, No. 7, pp. 427–432. ISSN 0001-0782. Available from DOI 10.1145/362686.362693.

[21] Janoušek, J.. *Arbology: Algorithms on Trees and Pushdown Automata.*

[22] Lu, Jiaheng. Introduction. In: *An Introduction to XML Query Processing and Keyword Search*. Springer Berlin Heidelberg, 2013. pp. 1–8. ISBN 978-3-642-34554-8.
`http://dx.doi.org/10.1007/978-3-642-34555-5_1`.

[23] Goldman, Roy; and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. pp. 436–445. VLDB '97. ISBN 1-55860-470-7.
`http://dl.acm.org/citation.cfm?id=645923.671008`.

[24] Milo, Tova; and Dan Suciu. Index Structures for Path Expressions. In: Catriel Beeri; and Peter Buneman, eds. *Database Theory – ICDT'99*. Springer Berlin Heidelberg, 1999. pp. 277–295. Lecture Notes in Computer Science. ISBN 978-3-540-65452-0.
`http://dx.doi.org/10.1007/3-540-49257-7_18`.

[25] Tang, Nan; J.X. Yu; M.T. Ozsu; and Kam-Fai Wong. Hierarchical Indexing Approach to Support XPath Queries. In: *Data Engineering, 2008. On ICDE 2008, IEEE 24th International Conference.* 2008. pp. 1510–1512. Available from DOI 10.1109/ICDE.2008.4497606.

[26] Kaushik, Raghav; Philip Bohannon; Jeffrey F Naughton; and Henry F Korth. Covering Indexes for Branching Path Queries. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2002. pp. 133–144. SIGMOD '02. ISBN 1-58113-497-5. Available from DOI 10.1145/564691.564707.

[27] Pettovello, P. Mark; and Farshad Fotouhi. MTree: An XML XPath Graph Index. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2006. pp. 474–481. SAC '06. ISBN 1-59593-108-2. Available from DOI 10.1145/1141277.1141389.

[28] Zou, Qinghua; Shaorong Liu; and Wesley W. Chu. Ctree: a compact tree for indexing XML data. In: *Web Information and Data Management*. 2004. pp. 39–46. Available from DOI 10.1145/1031453.1031462.

[29] Wang, Haixun; Sanghyun Park; Wei Fan; and Philip S. Yu. 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*. New York, NY, USA: ACM, 2003. pp. 110–121. SIGMOD '03. ISBN 1-58113-634-X. Available from DOI 10.1145/872757.872774.

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

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

[32] Chung, Chin-Wan; Jun-Ki Min; and Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2002. pp. 121–132. SIGMOD '02. ISBN 1-58113-497-5. Available from DOI 10.1145/564691.564706.

[33] Zhang, Bo; Wei Wang; Xiaoling Wang; and Aoying Zhou. AB-Index: An Efficient Adaptive Index for Branching XML Queries. In: Ramamohanarao Kotagiri; P. Radha Krishna; Mukesh Mohania; and Ekawit Nantajeewarawat, eds. *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin Heidelberg, 2007. pp. 988–993. Lecture Notes in Computer Science. ISBN 978-3-540-71702-7.
`http://dx.doi.org/10.1007/978-3-540-71703-4_90`.

[34] Mandhani, Bhushan; and Dan Suciu. Query Caching and View Selection for XML Databases. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 2005. pp. 469–480. VLDB '05. ISBN 1-59593-154-6.
`http://dl.acm.org/citation.cfm?id=1083592.1083648`.

[35] Baeza-Yates, Ricardo A.. Searching subsequences. *Theoretical Computer Science*. 1991, Vol. 78, No. 2, pp. 363–376. ISSN 0304-3975. Available from DOI http://dx.doi.org/10.1016/0304-3975(91)90358-9.
`http://www.sciencedirect.com/science/article/pii/0304397591903589`.

[36] Troníček, Zdeněk; and Ayumi Shinohara. The size of subsequence automaton. *Theoretical Computer Science*. 2005, Vol. 341, No. 1-3, pp. 379–384. ISSN 0304-3975. Available from DOI http://dx.doi.org/10.1016/j.tcs.2005.03.027.
`http://www.sciencedirect.com/science/article/pii/S030439750500157X`.

[37] Crochemore, Maxime; Bořivoj Melichar; and Zdeněk Troníček. Directed acyclic subsequence graph – Overview. *Journal of Discrete Algorithms*. 2003, Vol. 1, No. 3-4, pp. 255–280. ISSN 1570-8667.
`http://www.sciencedirect.com/science/article/pii/S1570866703000297`.

[38] Hoshino, H.; A. Shinohara; M. Takeda; and S. Arikawa. Online construction of subsequence automata for multiple texts. In: *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. 2000. pp. 146–152. Available from DOI 10.1109/SPIRE.2000.878190.

[39] Crochemore, Maxime; and Zdeněk Troníček. *On the Size of DASG for Multiple Texts*.
`http://dx.doi.org/10.1007/3-540-45735-6_6`.

[40] Miklau, Gerome; and Dan Suciu. Containment and Equivalence for an XPath Fragment. In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA: ACM, 2002. pp. 65–76. PODS '02. ISBN 1-58113-507-6. Available from DOI 10.1145/543613.543623.

[41] Miklau, Gerome; and Dan Suciu. Containment and Equivalence for a Fragment of XPath. *J. ACM*. New York, NY, USA: ACM, 2004, Vol. 51, No. 1, pp. 2–45. ISSN 0004-5411. Available from DOI 10.1145/962446.962448.

[42] Buneman, Peter; Susan Davidson; Wenfei Fan; Carmem Hara; and Wang-Chiew Tan. Reasoning about Keys for XML. In: Giorgio Ghelli; and Gösta Grahne, eds. *Database Programming Languages*. Springer Berlin Heidelberg, 2002. pp. 133–148. Lecture Notes in Computer Science. ISBN 978-3-540-44080-2.
`http://dx.doi.org/10.1007/3-540-46093-4_8`.

[43] *NetBeans IDE* [online]. [cit. 2015-04-23].
`http://www.netbeans.org/`.

[44] *JDOM* [online]. [cit. 2015-04-22].
`http://www.jdom.org/`.

[45] Schimdt, et al. *XMark – An XML Benchmark Project* [online]. [cit. 2015-04-27].
`http://www.xml-benchmark.org/`.

[46] Waas, Florian. *xmlgen – faq* [online]. [cit. 2015-04-27].
`http://www.xml-benchmark.org/faq.txt`.

[47] SAXONICA. *SAXON – The XSLT and XQuery Processor* [online]. [cit. 2015-04-28].
`http://saxon.sourceforge.net/`.

Appendix A
Acronyms

- **DFA** Deterministic finite automaton
- **FA** Finite automaton
- **LPS** Labelled Prüfer sequence
- **NFA** Nondeterministic finite automaton
- **NPS** Numberef Prüfer sequence
- **TPA** Tree Paths Automaton
- **TSPA** Tree String Paths Automaton
- **TSPSA** Tree String Paths Subsequences Automaton
- **XML** Extensible markup language
- **XPath** XML Path Language
- **XSLT** eXtensible Stylesheet Language Transformations
- **W3C** World Wide Web Consortium

Appendix B
User's Manual for tpalib

The XML index software is developed using Java SE 8 and is designed as Java Class Library called `tpalib`. Example programmes using `tpalib` can be found on enclosed CD in `src/examples` directory. Possible use of `tpalib` is also shown in the following code snippet:

```java
import org.jdom2.Document;
import org.jdom2.Element;
import xml.index.tpa.*;

...

TreePathsAutomaton automaton;
String query = "//HOUSE/LORD";
String path = "GoT.xml";
Document document = (new DocumentLoader()).load(path);

//if the automaton was built before
if (new File(path + ".ser").isFile()) {
   //use it
   automaton = (new AutomatonFactory()).deserealize(path + ".ser");
} else {
   //build it and save it
   automaton = (new AutomatonFactory()).build(document);
   automaton.serialize(path + ".ser");
}

//get the list of XML elements satisfying the query
LinkedList<Element> result1 = automaton.evaluate(query);

//or

//get the occurrences
TreeSet<Integer> dsubset = automaton.getDSubset(query);
//maps the content of dsubset to individual XML elements
LinkedList<Element> result2 = automaton.evaluate(dsubset);
```

Appendix C
Syntax Structure of Complete XPath 1.0

```
LocationPath = RelativeLocationPath
               | AbsoluteLocationPath ;
AbsoluteLocationPath = '/' , [ RelativeLocationPath ]
                       | AbbreviatedAbsoluteLocationPath ;
AbbreviatedAbsoluteLocationPath = '//' , RelativeLocationPath ;
RelativeLocationPath = Step
                       | RelativeLocationPath '/' Step
                       | AbbreviatedRelativeLocationPath ;
AbbreviatedRelativeLocationPath = RelativeLocationPath , '//' , Step;
Step = AxisSpecifier , NodeTest , { Predicate }
       | AbbreviatedStep ;
AbbreviatedStep = '.' | '..' ;
AxisSpecifier = AxisName , '::'
                | AbbreviatedAxisSpecifier ;
AbbreviatedAxisSpecifier = [ '@' ]
AxisName = 'ancestor'
           | 'ancestor-or-self'
           | 'attribute'
           | 'child'
           | 'descendant'
           | 'descendant-or-self'
           | 'following'
           | 'following-sibling'
           | 'namespace'
           | 'parent'
           | 'preceding'
           | 'preceding-sibling'
           | 'self' ;
NodeTest = NameTest
          | NodeType , '(' , ')'
          | 'processing-instruction' , '(' Literal ')' ;
NodeType = 'text'
       | 'node' ;
Predicate = '[' , PredicateExpr , ']' ;
PredicateExpr = Expr;
Expr = OrExpr ;
PrimaryExpr = VariableReference
              | '(' , Expr ')'
              | Literal
              | Number
              | FunctionCall
FunctionCall = FunctionName , '(' , [ Argument {, Argument } ] , ')';
Argument = Expr;
UnionExpr = PathExpr
```

```
             | UnionExpr '|' PathExpr ;
PathExpr = LocationPath
           | FilterExpr
           | FilterExpr , '/' , RelativeLocationPath
           | FilterExpr , '//' , RelativeLocationPath ;
FilterExpr = PrimaryExpr
             | FilterExpr , Predicate ;
OrExpr = AndExpr
         | OrExpr 'or' AndExpr ;
AndExpr = EqualityExpr
          | AndExpr 'and' EqualityExpr ;
EqualityExpr = RelationalExpr
               | EqualityExpr '=' RelationalExpr
               | EqualityExpr '!=' RelationalExpr ;
RelationalExpr = AdditiveExpr
                 | RelationalExpr '<' AdditiveExpr
                 | RelationalExpr '>' AdditiveExpr
                 | RelationalExpr '<=' AdditiveExpr
                 | RelationalExpr '>=' AdditiveExpr ;
AdditiveExpr = MultiplicativeExpr
               | AdditiveExpr , '+' , MultiplicativeExpr
               | AdditiveExpr , '-' , MultiplicativeExpr
MultiplicativeExpr = UnaryExpr
                     | MultiplicativeExpr , MultiplyOperator ,
                       UnaryExpr
                     | MultiplicativeExpr , 'div' , UnaryExpr
                     | MultiplicativeExpr , 'mod' , UnaryExpr
UnaryExpr = UnionExpr
            | '-' UnaryExpr
ExprToken = '(' | ')' | '[' | ']' | '.' | '..' | '@' | ',' | '::'
            | NameTest
            | NodeType
            | Operator
            | FunctionName
            | AxisName
            | Literal
            | Number
            | VariableReference
Literal =
Number = Digits , [ '.' , [ Digits ] ]
        | '.' , Digits
Digits = Digit , { Digit }
Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;
Operator = OperatorName
          | MultiplyOperator
          | '/' | '//' | '|' | '+' | '-' | '=' | '!=' | '<' | '<='
          | '>' | '>='
OperatorName =
```

Appendix D
Contents of Enclosed CD

```
readme.txt ........................ the file with CD contents description
├── exe ........................... the directory with compiled tpalib library
├── lib ........................... dependencies
├── javadoc ....................... tpalib API documentation
├── datasets ...................... experimental XML documents
├── src ........................... the directory of source codes
│   ├── tpalib .................... implementation sources of tpalib
│   ├── examples .................. implementation sources of example programmes using tpalib
│   └── thesis .................... the directory of CSplainTEX source codes of the thesis
│       ├── images ................ the directory of images of the thesis
│       └── tikz .................. the directory of TikZ source codes of the thesis
└── text .......................... the thesis text directory
    └── DP_Sestakova_Eliska_2015.pdf .. the thesis text in PDF format
```