Raymond and Beverly Sackler
Faculty of Exact Sciences
The Blavatnik School of Computer Science

Optimizing XML Processing

Thesis submitted for the degree of
“Doctor of Philosophy”

by

Shachar Harussi

This thesis was carried out under the supervision of
Prof. Amir Averbuch

Submitted to the Senate of Tel-Aviv University
December, 2009
Dedicated to my beloved family
Abstract

Broadly used in database and networking applications, the Extensible Markup Language (XML) is the de facto standard for the interoperable document format. In this thesis, we optimize various XML tasks for processing of database (DB) and network applications. The thesis provides an integrated theory, algorithms, software and implementation for the following XML processing tasks: holistic join and holistic index for XML processing in a DB, XML compression for network servers and for smart cards clients, XML streaming parser API for smart cards clients, XML compressed validation for network servers and XML Schema extraction. Each XML processing task has its own obstacles and difficulties which we overcome to achieve optimized performance by providing constructive solutions.

Querying is a major task in XML processing of DB. Twig pattern is a common model that defines a structural part of an XML query. Answering a twig pattern query, which is a small labeled tree, is one of the most fundamental tasks in XML query processing. An efficient twig pattern query answering in XML, which is stored in DB, contains two tasks: join and indexing. In recent years, several holistic join methods were proposed. The ‘holistic’ term means that a twig pattern query is processed as a whole and it is not decomposed into local relations.

A twig tree pattern is unordered. Its branches define the logical operators AND and OR. In this thesis, we define a new tree automaton, which processes unordered trees, that is called an Unordered Unranked Tree Automaton (UUTA). We show how to construct a twig-UUTA from a twig pattern and how to query an unordered tree, which represents an XML document, by this twig-UUTA. In addition, the thesis shows how to query by a twig-UUTA another tree-UUTA that recognizes a collection of potential XML documents. These constructions enable to implement holistic join and holistic index. Our join technique queries the XML document by an exact twig pattern. Therefore, it
is more accurate than other holistic join methods. These constructions also enable to implement an holistic structural index.

The thesis also contributes to processing of XML in networks applications. Accessing streams of XML is a major task in processing XML in networks. XML is a highly verbose language. Therefore, XML textual data encoding is oversized. As a result, XML accessing is computationally expensive. In this thesis, we developed several algorithms that access efficiently XML streams that optimize CPU and memory usages.

Accessing XML involves parsing and validation. Parsing checks whether an XML message is well-formed. The thesis modifies and ports the StAX XML streaming parser for smart cards. Smart cards have limited memory. The ported API is called MinStAX. MinStAX, which is aimed to reduce the memory footprint, is a seamless support of the compressed XML API for closed environments. The support is ‘seamless’ since MinStAX supplies the same standard API for parsing of both standard XML and compressed XML.

We show how MinStAX can compress XML by the application of the Tagged Sub-optimal Encoding (TSC). In addition, the thesis also improves the TSC compression scheme in several ways. We show how to process the TSC as a universal code. We introduce the $TSC^k$ as a family of universal codes where $TSC^0$ is the original TSC. We introduce a fast decoding technique, which uses compact transition tables, in order to decode the compressed data as bytes. We adopt a pattern matching algorithm to use the same compact tables. The encoding, decoding and search times of the $TSC^k$ compression scheme are similar.

In addition to being well-formed, an XML message may be invalid. The XML message is validated against an XML schema. The thesis presents a compression algorithm, which combines validation with access to compressed XML streams. It optimizes the XML access time by reducing the size of the XML textual encoding via lossless compression and by accessing the compressed XML instead of accessing the original message.

Validation uses an XML schema. But sometimes the XML schema has to be extracted from the XML data itself. This raises the problem how to extract a schematic information from an XML stream. A schema should tightly represents the data while being compact. The thesis suggests a new XML Schema extraction algorithm if the Schema is missing.
Acknowledgments

"Good morning," he said. He was standing before a garden, all a-bloom with roses. "Good morning," said the roses. The little prince gazed at them. They all looked like his flower. "Who are you?" he demanded, thunderstruck. "We are roses," the roses said. And he was overcome with sadness. His flower had told him that she was the only one of her kind in all the universe. And here were five thousand of them, all alike, in one single garden! Then he went on with his reflections: "I thought that I was rich, with a flower that was unique in all the world; and all I had was a common rose ... That doesn’t make me a very great prince ..." And he lay down in the grass and cried.

.....

And then the fox added: "Go and look again at the roses. You will understand now that yours is unique in all the world. Then come back to say goodbye to me." The little prince went away, to look again at the roses. "You are not at all like my rose," he said. "As yet you are nothing. No one has tamed you, and you have tamed no one. You are like my fox when I first knew him. He was only a fox like a hundred thousand other foxes. But I have made him my friend, and now he is unique in all the world." The Little Prince, written by Antoine de Saint-Exupery

First and foremost, I wish to extend my deepest gratitude to my fox, friend and adviser Prof. Amir Averbuch for helping me tame both my personal and academic rosebush. More specifically, I want to thank Amir for his invaluable help, support and guidance throughout the preparation of this thesis. For his assistance with my research, for his care, for introducing me to new collaborators and for making everything he can so my studies become as fruitful as possible. Working with Prof. Averbuch was a fascinating journey and an enriching experience for me and I am grateful for him. Prof. Averbuch is a rare combination of an advisor, a researcher and a human being. And most importantly, I thank Prof. Averbuch for his friendship during my academic voyage. For everything you have done for me, I thank you.

I thank Prof. Amiram Yehudai for his great help. I also thank Neta Rabin, Sagi Amiel, Dianea Lematov and Christina Akinshin for helping gardening my academic rosebush.

I want to express my deepest gratitude to my dearest wife Gat, the most beautiful untamed foxy rose in my garden. Thank you for your constant support, encouragement, patience and belief in me throughout my Ph.D. studies. Thank you for knowing how to be both my fox and my rosebush. Thank you for keeping me rooted to the soil of reality. I also thank my dear roses sprouts, my children Noa, Hila and Ofir for being my secret garden, my haven.
I am grateful to my parents, Hana and Yaniv. The loving seeds of free thinking they planted years ago has shaped me to be the person I am today.

"Goodbye," said the fox. "And now here is my secret, a very simple secret: It is only with the heart that one can see rightly; what is essential is invisible to the eye."

*The Little Prince, written by Antoine de Saint-Exupery*


## Contents

1. **Introduction and Background**  
   1. **Introduction**  
      1.1 Background and motivation  
      1.2 Contributions of the thesis to XML DB query  
      1.3 Contributions of the thesis to XML processing in networks  
      1.4 Structure of the thesis  
   2. **Background**  
      2.1 XML  
      2.1.1 XML Schema (XSD)  
      2.1.2 Document Type Definition (DTD)  
      2.2 XML as a semi-structured data  
      2.3 XML query languages  
      2.4 Automata and languages  

II. **Querying of XML in a DB**  
   3. **Preliminaries for querying of XML in a DB**  
      3.1 DB and XML  
      3.1.1 XML storage model  
      3.1.2 XML querying model  
      3.2 Tree Automata and XML  
   4. **TA Methodology**  
      4.1 Bottom-up Unordered Unranked Trees Automata (UUTA)  
      4.2 Construction of UUTA from a twig pattern  
      4.2.1 Extension of twig-UUTA to derive subtrees
4.3 Application of a Selecting Tree Automaton (STA) on a tree $T$.  
4.3.1 FSA $A_{pc}^{Selecting}$ construction.  
4.3.2 Top-down traversal by $STA_{pc}^{Selecting}(T)$.  
4.4 Application of STA to a tree automaton $A_{Tree}$.  
4.4.1 Intersection between UUTAs.  
4.4.2 FSA construction for $STA_{pc}^{Selecting}(A_{Tree})$.  
4.4.3 Top-down traversal of $STA_{pc}^{Selecting}(A_{Tree})$.  
4.4.4 The relation between $STA(A_{Tree})$ and $STA(T)$.  
4.5 Efficient FSA construction of $(A_{Tree} \cap A_{Selecting})_{pc}$.  
4.5.1 The children-search techniques.  
4.5.2 $(A_{Tree} \cap A_{Selecting})_{pc}$ construction time analysis.  
4.6 Conclusion.  

5 Holistic structural-join.  
5.1 Introduction.  
5.1.1 TwigTA basic operation.  
5.2 Related work: twig join algorithms.  
5.3 $STA_{pc}^{Twig}(T)$ based join.  
5.4 The TwigTA Algorithm.  
5.4.1 Extending the STA operations for the TwigTA algorithm.  
5.4.1.1 $A_{Tree}(T, P)$ operation.  
5.4.1.2 $STA_{pc}^{Twig}(A_{Tree}, history)$ operation.  
5.4.2 Description of the TwigTA Algorithm.  
5.4.3 Partition construction.  
5.4.4 Prediction-tree construction.  
5.4.5 $A_{Tree}$ construction.  
5.4.6 Selecting nodes.  
5.4.7 Removal of unselected nodes.  
5.4.8 Nodes extraction from streams.  
5.4.8.1 Using a DB index.  
5.4.9 Intermediate paths construction.  
5.4.10 Removal of present-nodes.  
5.4.11 Output of twig pattern solutions.  
5.4.12 Example of a TwigTA run.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>192</td>
</tr>
<tr>
<td>7.2</td>
<td>XML background and related work</td>
<td>194</td>
</tr>
<tr>
<td>7.2.1</td>
<td>XML parsers classifications</td>
<td>194</td>
</tr>
<tr>
<td>7.2.2</td>
<td>An introduction to StAX</td>
<td>196</td>
</tr>
<tr>
<td>7.3</td>
<td>Data structures redesign</td>
<td>200</td>
</tr>
<tr>
<td>7.4</td>
<td>Data compression</td>
<td>207</td>
</tr>
<tr>
<td>7.5</td>
<td>Experimental results</td>
<td>209</td>
</tr>
<tr>
<td>7.6</td>
<td>Conclusions and future work</td>
<td>210</td>
</tr>
<tr>
<td>8</td>
<td>Tagged Suboptimal Code (TSC)</td>
<td>211</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>211</td>
</tr>
<tr>
<td>8.2</td>
<td>Preliminaries and related work</td>
<td>212</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Encoding</td>
<td>212</td>
</tr>
<tr>
<td>8.2.1.1</td>
<td>Tagged Suboptimal Code (TSC)</td>
<td>214</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Compressed string matching</td>
<td>215</td>
</tr>
<tr>
<td>8.3</td>
<td>The TSC compression algorithm</td>
<td>216</td>
</tr>
<tr>
<td>8.3.1</td>
<td>TSC as a universal code</td>
<td>217</td>
</tr>
<tr>
<td>8.3.2</td>
<td>$TSC^k$ universal code</td>
<td>218</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Fast and compact $TSC^0$ decoding</td>
<td>220</td>
</tr>
<tr>
<td>8.3.3.1</td>
<td>Fast decoding by $TSC^1$</td>
<td>224</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Text compression by TSC universal codes</td>
<td>226</td>
</tr>
<tr>
<td>8.4</td>
<td>Compressed string matching in TSC</td>
<td>229</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Encoded data translation</td>
<td>230</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Pattern string translation</td>
<td>231</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Pattern matching</td>
<td>233</td>
</tr>
<tr>
<td>8.5</td>
<td>Theoretical analysis of $TSC^k$ compression for k=0,1</td>
<td>235</td>
</tr>
<tr>
<td>8.6</td>
<td>Experimental results</td>
<td>238</td>
</tr>
<tr>
<td>8.7</td>
<td>Conclusion</td>
<td>242</td>
</tr>
<tr>
<td>8.8</td>
<td>APPENDIX I: Extended theory</td>
<td>243</td>
</tr>
<tr>
<td>8.8.1</td>
<td>Case I, Input grows like $2^n$</td>
<td>243</td>
</tr>
<tr>
<td>8.8.2</td>
<td>Case II, Input grows like the Fibonacci series</td>
<td>244</td>
</tr>
</tbody>
</table>
IV Compressed validation of XML documents

9 DPDT algorithm

9.1 Introduction
   9.1.1 Motivation
   9.1.2 The basic idea
   9.1.3 Outline of the lossless XML compression algorithm
   9.1.4 Main results

9.2 Related work
   9.2.1 Context-Free-Grammar (CFG) encoding models
      9.2.1.1 LL guided parsing encoding models
      9.2.1.2 LR vs. LL guided-parsing encoding models
      9.2.1.3 Encoding methods for CFG models
   9.2.2 XML parsing
      9.2.2.1 Parsing of XML streams
      9.2.2.2 XML validation
   9.2.3 XML Compression
      9.2.3.1 XML encoding models
      9.2.3.2 DTD awareness
      9.2.4 Prediction by Partial Matching (PPM) encoding

9.3 XML compression: the DPDT-L algorithm
   9.3.1 Dictionary conversion
   9.3.2 The RE lexer
   9.3.3 Parser Generator
   9.3.4 XML lexer
   9.3.5 The DPDT parser
   9.3.6 DPDT guided encoding

9.4 XML compression results
   9.4.1 The source data
   9.4.2 Compression ratios

9.5 Future work
## V XML Schema extraction

### 10 Two Levels State Machine algorithm

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Introduction</td>
<td>287</td>
</tr>
<tr>
<td>10.2 Preliminaries</td>
<td>288</td>
</tr>
<tr>
<td>10.2.1 Prefix Tree Acceptor (PTA)</td>
<td>289</td>
</tr>
<tr>
<td>10.2.2 XSD II Notation</td>
<td>290</td>
</tr>
<tr>
<td>10.3 Related work: XML schema extraction</td>
<td>292</td>
</tr>
<tr>
<td>10.4 The XTLSM algorithm</td>
<td>293</td>
</tr>
<tr>
<td>10.4.1 Two Layers State Machine (TLSM)</td>
<td>293</td>
</tr>
<tr>
<td>10.4.2 Ordered Labeled Tree Validation</td>
<td>295</td>
</tr>
<tr>
<td>10.4.3 Main Steps of the XTLSM Algorithm</td>
<td>297</td>
</tr>
<tr>
<td>10.4.4 Tree to PTA</td>
<td>299</td>
</tr>
<tr>
<td>10.4.5 Generalization of the PTA</td>
<td>303</td>
</tr>
<tr>
<td>10.4.5.1 Sub-Tree Similarity</td>
<td>305</td>
</tr>
<tr>
<td>10.4.5.2 Merging Two States</td>
<td>307</td>
</tr>
<tr>
<td>10.4.6 Adding of Order Layer</td>
<td>310</td>
</tr>
<tr>
<td>10.4.7 Extract II Notation XSD from FSM</td>
<td>313</td>
</tr>
<tr>
<td>10.4.7.1 Remove Global Detour method</td>
<td>314</td>
</tr>
<tr>
<td>10.4.7.2 Remove Global Loop</td>
<td>316</td>
</tr>
<tr>
<td>10.4.7.3 Vertical Chopping</td>
<td>317</td>
</tr>
<tr>
<td>10.4.7.4 Horizontal Chopping</td>
<td>323</td>
</tr>
<tr>
<td>10.4.7.5 State Elimination</td>
<td>327</td>
</tr>
<tr>
<td>10.5 Experimental results</td>
<td>328</td>
</tr>
<tr>
<td>10.5.1 Synthetic examples</td>
<td>328</td>
</tr>
<tr>
<td>10.5.1.1 FSMs with Loops</td>
<td>328</td>
</tr>
<tr>
<td>10.5.1.2 Definitions for elements with the same name</td>
<td>330</td>
</tr>
<tr>
<td>10.5.2 Experimental settings for XML datasets</td>
<td>333</td>
</tr>
<tr>
<td>10.5.3 Computational complexity of the XTLSM Algorithm</td>
<td>333</td>
</tr>
<tr>
<td>10.5.4 Generalization measurements of the XTLSM algorithm</td>
<td>334</td>
</tr>
<tr>
<td>10.6 Conclusions</td>
<td>335</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example of XHTML document</td>
<td>29</td>
</tr>
<tr>
<td>1.2</td>
<td>The tree structure of the XML document in Fig. 1.1</td>
<td>29</td>
</tr>
<tr>
<td>1.3</td>
<td>XML processing tasks and their relations to the presented algorithms</td>
<td>30</td>
</tr>
<tr>
<td>1.4</td>
<td>The relations among XML DB querying and automata applications</td>
<td>32</td>
</tr>
<tr>
<td>1.5</td>
<td>Example of $A(T)$ operation</td>
<td>33</td>
</tr>
<tr>
<td>1.6</td>
<td>Example of $STA(T)$ operation</td>
<td>35</td>
</tr>
<tr>
<td>1.7</td>
<td>Example of $ATree$</td>
<td>36</td>
</tr>
<tr>
<td>1.8</td>
<td>Illustration of FSA $(ATree \cap ATwig)_{pc}$</td>
<td>37</td>
</tr>
<tr>
<td>1.9</td>
<td>General flow of the index and join operations</td>
<td>39</td>
</tr>
<tr>
<td>1.10</td>
<td>Relations among XML networking applications and XML processing</td>
<td>40</td>
</tr>
<tr>
<td>1.11</td>
<td>Flow of dependencies among the thesis parts and chapters</td>
<td>43</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of an XML document</td>
<td>44</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of XML schema</td>
<td>46</td>
</tr>
<tr>
<td>2.3</td>
<td>DTD of the XHTML example in Fig. 1.1</td>
<td>48</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustration of a node-labeled tree $T$</td>
<td>50</td>
</tr>
<tr>
<td>2.5</td>
<td>Graphical representation of the twig pattern <code>//al//c//bl//e//d/</code></td>
<td>53</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of a FSM</td>
<td>54</td>
</tr>
<tr>
<td>2.7</td>
<td>Illustration of several OLTs</td>
<td>55</td>
</tr>
<tr>
<td>3.1</td>
<td>Sample from an XML document $T$</td>
<td>61</td>
</tr>
<tr>
<td>3.2</td>
<td>Sample from an XML document $T$</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of the $A(T)$ run</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustration of a run $ATwig(T)$</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration of a run $ATwig(T)$</td>
<td>75</td>
</tr>
</tbody>
</table>
4.4 The flow of the \( ST A^S_{Selecting}(T) \) operation .......................................................... 78
4.5 Illustration of \( ATwig \) ........................................................................................................ 81
4.6 Illustration of a \( \rho_{ST A^S_{Twig}}(T) \) .................................................................................. 85
4.7 General Flow of the \( ST A^S_{Selecting}(ATree) \) operation ............................................. 86
4.8 Illustration of \( L(ATree) \) .................................................................................................. 88
4.9 Illustration of the FSA \( (ATree \cap ATwig)_{pc} \) .................................................................... 90
4.10 Illustration of the FSA \( (ATree \cap ATwig)_{pc} \) without inaccessible states ....................... 92
4.11 Illustration of \( ATree_{pc} \) and \( ATwig_{pc} \) .......................................................................... 95
4.12 Demonstration of Algorithm 6 operation .......................................................................... 100

5.1 General flow of the \( TwigTA \) algorithm ........................................................................... 109
5.2 Description of \( \rho_{ATree(T)} \) and \( \rho_{ATree(T,P)} \) ................................................................. 120
5.3 Demonstration of the \( ST A^S_{ATwig}(ATree, history) \) operation ..................................... 124
5.4 Illustration of the status of nodes in \( DT \) during the \( TwigTA \) operation ......................... 125
5.5 Illustration of the \textit{present-future-subtree}(\( DT, extracted \)) ........................................... 126
5.6 Illustration of 1-prediction \( T^{Prediction} \) ........................................................................ 132
5.7 Illustration of 2-prediction \( T^{Prediction} \) ........................................................................ 133
5.8 Illustration of the run of \( A_{Tree(T, P)} \) ........................................................................... 135
5.9 Illustration of the FSA \( (ATree \cap ATwig)_{pc} \) ..................................................................... 137
5.10 Illustration of \( T^{Prediction}_{pc} \) and \( \rho_{ST A^S_{Twig}(ATree, history)} \) ............................... 142
5.11 \( T^{Prediction}_{pc} \) in the \( TwigTA \) run over XML document in Fig. 3.2 ................................. 147
5.12 \( TwigStack \) operation on an XML document ................................................................ 150
5.13 Comparison between the performance of \( TwigTA \) and other algorithms ....................... 153
5.14 Performance of \( TwigTA \) with different prediction sizes .................................................. 155

6.1 General flow of the \( SelectIndex(T, Q) \) algorithm ........................................................ 158
6.2 General flow of the three families from the \( SelectIndex(T, Q) \) algorithm ....................... 160
6.3 Illustration of the 1-index .................................................................................................... 164
6.4 Illustration of the \( A(K) \)-indexes ..................................................................................... 165
6.5 Description of \( A_{summary} \), \( A^Q \), \( A_{summary} \cap A^Q \) .................................................... 169
6.6 Description of \( A^Q_{paths} \) and \( A_{summary} \cap A^Q_{paths} \) .................................................. 177
6.7 Description of \( A^Q \) and \( (A_{summary} \cap A^Q)_{pc} \) ................................................................. 178
6.8 Description of \( (A_{summary})_{pc} \) and \( (A^Q_{summary} \cap A^Q)_{pc} \) ....................................... 181
6.9 Description of \( A^F_{A} \) and \( A^F_{A} \cap A^Q_{paths} \) ................................................................. 184
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.10</td>
<td>Description of ((A_{T,4}^{\text{summary}})^{\text{pc}} \text{ and } (A_{T,4}^{\text{summary}} \cap A_Q)^{\text{pc}})</td>
</tr>
<tr>
<td>7.1</td>
<td>Example of a seamless support of both standard and compressed XML</td>
</tr>
<tr>
<td>7.2</td>
<td>Example of StAX API</td>
</tr>
<tr>
<td>7.3</td>
<td>Description of the modules and processing of StAX API</td>
</tr>
<tr>
<td>7.4</td>
<td>Illustration of StAX internal data structures</td>
</tr>
<tr>
<td>7.5</td>
<td>Description of MinStAX internal data structures</td>
</tr>
<tr>
<td>7.6</td>
<td>Description of the compressed MinStAX internal data structures</td>
</tr>
<tr>
<td>8.1</td>
<td>TSC quadtree</td>
</tr>
<tr>
<td>8.2</td>
<td>(TSC^0) decoding tables for nibble input-words</td>
</tr>
<tr>
<td>8.3</td>
<td>Decoding the integers 6, 2, 0 from table 8.1</td>
</tr>
<tr>
<td>8.4</td>
<td>Decoding integers 0 and 6 in table 8.2</td>
</tr>
<tr>
<td>8.5</td>
<td>Illustration of (TSC^0) codec context</td>
</tr>
<tr>
<td>8.6</td>
<td>Changes of context by (TSC^0) when the string “bccc” is decoded</td>
</tr>
<tr>
<td>8.7</td>
<td>Illustration of data translation into code-subwords symbols domain</td>
</tr>
<tr>
<td>8.8</td>
<td>Illustration of a refined data translation into a code-subwords domain</td>
</tr>
<tr>
<td>8.9</td>
<td>Translation of the string ‘were’ into code-subword symbols patterns</td>
</tr>
<tr>
<td>9.1</td>
<td>Flow of the XML lossless compression algorithm: the main components</td>
</tr>
<tr>
<td>9.2</td>
<td>A (CFG) definition of XHTML subset in Fig. 2.3</td>
</tr>
<tr>
<td>9.3</td>
<td>A decision table of the (CFG) that is defined in Fig. 9.2</td>
</tr>
<tr>
<td>9.4</td>
<td>Parsing process of the XHTML document that was defined in Fig. 1.1</td>
</tr>
<tr>
<td>9.5</td>
<td>Flow of the XML compression algorithm (DPDT-L)</td>
</tr>
<tr>
<td>9.6</td>
<td>DTD conversion of XHTML subset</td>
</tr>
<tr>
<td>9.7</td>
<td>FSM for RE lexer</td>
</tr>
<tr>
<td>9.8</td>
<td>DPDT that accepts the XHTML elements in Fig. 9.6</td>
</tr>
<tr>
<td>9.9</td>
<td>DPDT parsing of the XHTML document which appears in Figure 1.1</td>
</tr>
<tr>
<td>9.10</td>
<td>The XML lexer FSM</td>
</tr>
<tr>
<td>9.11</td>
<td>XHTML relevant symbols</td>
</tr>
<tr>
<td>10.1</td>
<td>Illustration of a (PTA)</td>
</tr>
<tr>
<td>10.2</td>
<td>Example of XSD written in (\Pi) notation</td>
</tr>
<tr>
<td>10.3</td>
<td>Illustration of a TLSM</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>10.4</td>
<td>Creation of a $PTA^T$ from an OLT $T$</td>
</tr>
<tr>
<td>10.5</td>
<td>Creation of a $PTA^T$ from an OLT $T$. A continue</td>
</tr>
<tr>
<td>10.6</td>
<td>Illustration of parent merging</td>
</tr>
<tr>
<td>10.7</td>
<td>Example of low similarity value between two states</td>
</tr>
<tr>
<td>10.8</td>
<td>Example of high similarity value between two states</td>
</tr>
<tr>
<td>10.9</td>
<td>Example of merging two states using the $\text{MergeStates}$ function</td>
</tr>
<tr>
<td>10.10</td>
<td>Example of merging two states using the $\text{MergeStates}$ function</td>
</tr>
<tr>
<td>10.11</td>
<td>Illustration of several states in a $FSM$</td>
</tr>
<tr>
<td>10.12</td>
<td>Tree nodes that are the origin of state $q_1$ in Fig. 10.11</td>
</tr>
<tr>
<td>10.13</td>
<td>An example of an $\text{Inner FSM}$</td>
</tr>
<tr>
<td>10.14</td>
<td>Illustration of the removal by the $\text{RemoveGlobalDetour}$ method</td>
</tr>
<tr>
<td>10.15</td>
<td>Illustration of removal by the $\text{RemoveGlobalLoop}$ method</td>
</tr>
<tr>
<td>10.16</td>
<td>Illustration of the vertical chopping operation</td>
</tr>
<tr>
<td>10.17</td>
<td>Illustration of a selected path, min, ane and max values</td>
</tr>
<tr>
<td>10.18</td>
<td>Illustration of horizontal chopping</td>
</tr>
<tr>
<td>10.19</td>
<td>Illustration of reducing a state by the state elimination approach</td>
</tr>
<tr>
<td>10.20</td>
<td>Example of an XML document that contains a loop</td>
</tr>
<tr>
<td>10.21</td>
<td>Example of the TLSM that was created</td>
</tr>
<tr>
<td>10.22</td>
<td>Generated XML Schema from the XML in Fig. 10.20</td>
</tr>
<tr>
<td>10.23</td>
<td>Example of an XML document</td>
</tr>
<tr>
<td>10.24</td>
<td>Example of the TLSM that was created</td>
</tr>
<tr>
<td>10.25</td>
<td>Example of an XSD with elements definition of the same element name</td>
</tr>
<tr>
<td>10.26</td>
<td>A graph of the $XTLSM$ generalization parameter</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Example of the application of Algorithm 6</td>
<td>99</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of the application of Algorithm 7</td>
<td>102</td>
</tr>
<tr>
<td>4.3</td>
<td>Example for an optimized application of Algorithm 7</td>
<td>103</td>
</tr>
<tr>
<td>4.4</td>
<td>Example for an optimized application of Algorithm 7</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>A flow of Algorithm 8 $A^{Twig}(D^T)$ operation</td>
<td>115</td>
</tr>
<tr>
<td>5.2</td>
<td>Description of Algorithm 9 application of the $A^{Twig}_p$</td>
<td>118</td>
</tr>
<tr>
<td>5.3</td>
<td>Description of the run of Algorithm 11</td>
<td>143</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of the intermediate solutions of Algorithm 11</td>
<td>145</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of the sorted intermediate solutions of Algorithm 11</td>
<td>145</td>
</tr>
<tr>
<td>5.6</td>
<td>Example of the final solutions of Algorithm 11</td>
<td>146</td>
</tr>
<tr>
<td>5.7</td>
<td>$TwigTA$ internal data in the run over the XML document in Fig. 3.2</td>
<td>146</td>
</tr>
<tr>
<td>5.8</td>
<td>Characteristics of the XML Datasets for the $TwigTA$ Experiments</td>
<td>151</td>
</tr>
<tr>
<td>5.9</td>
<td>Queries used in the $TwigTA$ experiments</td>
<td>152</td>
</tr>
<tr>
<td>6.1</td>
<td>Datasets used in the index experiments</td>
<td>187</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance metrics of the index experiments</td>
<td>189</td>
</tr>
<tr>
<td>7.1</td>
<td>Example of the stack structure of the $SymbolsString$</td>
<td>204</td>
</tr>
<tr>
<td>8.1</td>
<td>Example of $TSC$ universal codes for the numbers 0, . . . , 8</td>
<td>218</td>
</tr>
<tr>
<td>8.2</td>
<td>Example of $TSC^1$ universal codes for the numbers 0, . . . , 8</td>
<td>220</td>
</tr>
<tr>
<td>8.3</td>
<td>Partial input-subwords table for $TSC^1$ decoding of nibble input-words</td>
<td>226</td>
</tr>
<tr>
<td>8.4</td>
<td>Comparison between $TSC$ and Elias gamma codes implied frequencies</td>
<td>238</td>
</tr>
<tr>
<td>8.5</td>
<td>Summary of the compression ratios in percentage</td>
<td>240</td>
</tr>
<tr>
<td>8.6</td>
<td>Summary of the encoding, decoding and string matching times</td>
<td>241</td>
</tr>
<tr>
<td>8.7</td>
<td>Summary of the encoding and decoding times</td>
<td>242</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>9.1 Characteristics of files in the DPDT-L algorithm benchmark</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>9.2 bpc from different compression methods operated on small datasets</td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>9.3 bpc from different compression methods operated on medium datasets</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>9.4 bpc from different compression methods operated on large datasets</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>9.5 Improvements in % of the DPDT-L against different compression methods</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>10.1 Sizes of the XML documents</td>
<td>334</td>
<td></td>
</tr>
</tbody>
</table>
List of Notations

2.1 Directed graph $G \overset{\Delta}{=} (V^G, E^G)$ ................................. 49
   Definition.2.1

2.2 Tree $T \overset{\Delta}{=} (V^T, E^T)$ ................................................. 49
   Definition.2.2

2.3 Node-labeled graph $(V^G, E^G, label^G)$, Node-labeled tree $(V^T, E^T, label^T)$ ............................... 49
   Definition.2.3

2.4 Graph $G$ and tree $T$ languages: $L(G, v), L(G), w_v, L(T), L_{leaf}(T)$ ........ 50
   Definition.2.4

2.5 FSA $A$ is the tuple $(Q^A, \Sigma, q^A_0, F^A, \delta^A)$ .............................................. 53
   Definition.2.5

2.7 TA $A$ is the tuple $(Q^A, \Sigma, F^A, \delta^A)$ ................................................ 56
   Definition.2.7

2.7 $L(A)$ of TA $A$ ................................................................. 56
   Definition.2.7

4.1 UUTA $A \overset{\Delta}{=} (Q^A, \Sigma, F^A, \delta^A)$ .............................................. 69
   Definition.4.1

4.2 $A(T)$ applies UUTA $A$ on tree $T$ and outputs $\rho^{A(T)}$ ........................................ 69
   Definition.4.2

4.3 STA is a pair $(A_{Selecting}, S)$ ................................................ 77
   Definition.4.3

4.3 $STA^S_{Selecting}(T)$ applies STA $(A_{Selecting}, S)$ on tree $T$ and outputs $\rho^{STA^S_{Selecting}(T)}$ .............................................. 77
   Definition.4.3

4.3 FSA $A_{Selecting}^{Selecting}$ is constructed from UUTA $A_{Selecting}^{Selecting}$ ........................ 79
   subsection.4.3.1

4.4 UUTA $A$, state $q$ and tree $T$ languages: $L(A, T, q), L_k(A, q), L^*(A, q)$ ........ 81
   Definition.4.4
4.5 \( STA^S_{Selecting}(A\text{Tree}) \) applies \( STA(A^S_{Selecting}, S) \) on \( UUTA A^{Tree} \) and outputs \( \rho^{STA^S_{Selecting}(A\text{Tree})} \) .......................... 85

Definition.4.5

4.6 \( A^1 \cap A^2 \) intersects UUTAs \( A^1 \) and \( A^2 \) .......................... 87

Definition.4.6

4.7 Maximal Sets UUTA \( A^{Max} \) and Maximal set \( S^{Max}_{A,q} \) .......................... 102

Definition.4.7

5.1 Partition \( P : V^T \mapsto Q^{A\text{Tree}} \) .......................... 119

Definition.5.1

5.1 \( A^{Tree}(T, P) \) Applies UUTA \( A^{Tree} \) on tree \( T \) and partition \( P \) and outputs \( \rho^{A^{Tree}(T,P)} \) .......................... 119

Definition.5.1

5.2 \( STA^S_{A^{Tree}}(T, P) \) applies \( STA(A^{Tree}, S) \) on tree \( T \) and partition \( P \) and outputs \( \rho^{STA^S_{A^{Tree}}(T,P)} \) .......................... 120

Definition.5.2

5.3 \( STA^S_{A^{Selecting}}(A\text{Tree},\text{history}) \) applies \( STA(A^S_{Selecting}, S) \) on UUTA \( A^{Tree} \) and \( \text{history} \) function and outputs \( \rho^{STA^S_{A^{Selecting}}(A\text{Tree,history})} \) .......................... 122

Definition.5.3

8.1 code \( c_1, \ldots, c_n \) .......................... 212

Definition.8.1

9.1 D-grammar \( G = (N, \Sigma, P, A_1) \) .......................... 267

Definition.9.1

9.2 \( \Rightarrow \) derivation relation .......................... 268

Definition.9.2

9.3 DPDT \( M = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F) \) .......................... 268

Definition.9.3

10.1 \( PTAT = (\Sigma, S, s_0, \delta, F) \) is an FSM .......................... 289

Definition.10.1

10.2 XSD \( \Pi \) notion is \( X = (\Pi, \pi_0 \in \Pi) \) where \( \Pi = \{\pi_0, \pi_1, \pi_2, \ldots\} \) .......................... 290

Definition.10.2

10.3 TLSM \( B = (Q, q_0 \in Q, \delta, F, \Sigma) \) is an outer FSM where each outer state \( q = (\Sigma^q, S^q, s^q_0, \delta^q, F^q) \) is an inner FSM .......................... 293

Definition.10.3
List of Algorithms

1. The twig-UUTA construction algorithm ........................................ 71
2. The Modify-UUTA algorithm ..................................................... 74
3. The $\Delta$Selecting construction algorithm ................................... 79
4. The STA(T) top-down traversal algorithm .................................... 84
5. The $STA^S_{Selecting}(ATree)$ top-down traversal algorithm .............. 91
6. An efficient construction function for intersected-FSA ..................... 97
7. The children-search function ..................................................... 98
8. $ATwig(D^T)$ - the join bottom-up traverse ................................. 114
9. The top-down traverse of the STA(T)-based join ............................. 117
10. The children-search function that receives history ......................... 123
11. The TwigTA algorithm ............................................................. 128
12. The MakePartition algorithm ................................................... 129
13. The ConstructTree algorithm .................................................... 131
14. The ConstructAutomaton algorithm .......................................... 134
15. The RemoveUnselected algorithm .............................................. 138
16. The ExtractNodes algorithm ..................................................... 139
17. The IntermediatePaths algorithm .............................................. 141
18. The RemovePresent algorithm .................................................. 144
19. The SelectIndex$^G_{\text{Graph}}$ algorithm ....................................... 161
20. The SelectIndex$^F_{\text{A}}$ algorithm ............................................ 166
21. The SelectIndex$^T_{\text{A}}$ algorithm ............................................ 170
22. The ConstructUUTA algorithm from TA ..................................... 173
23. The ConstructUUTA algorithm from FA ..................................... 174
24. The improved ConstructUUTA algorithm from FA ......................... 179
25. $TSC^0$ byte-oriented transform .............................................. 223
26. The adaptive algorithm .......................................................... 228
27 The Aho-Corasick search algorithm ........................................... 235
28 Validation of OLT using TLSM ................................................... 296
29 XML to XSD algorithm ............................................................. 298
30 Generation of the PTA from a given XML document ...................... 299
31 Generalization of the PTA into FSM ........................................... 303
32 Similarity calculation between two sub-trees in the PTA .................... 305
33 Merging two states of a FSM ....................................................... 308
34 Adding order layer ................................................................. 310
35 TLSM to $\Pi$ notation XSD ......................................................... 313
36 Convert inner FSM to $\Pi$ notation XSD element .......................... 314
37 Remove global detour from FSM ............................................... 315
38 Remove global loop from FSM .................................................... 316
39 FSM vertical chopping - main method ....................................... 318
40 FSM vertical chopping - calculate min anc and max ...................... 321
41 FSM vertical chopping - removal of non bridges ......................... 323
42 FSM horizontal chopping ......................................................... 324
43 Mark Cluster ........................................................................... 324
Part I
Introduction and Background
Chapter 1

Introduction

1.1 Background and motivation

Broadly used in DB and networking applications, the Extensible Markup Language (XML) \[32\] is the de facto standard for the interoperable document format. In this thesis, we optimize various tasks of XML processing for DB and networking applications. Each XML processing task has its own obstacles and difficulties which we overcome to achieve optimized performance by providing constructive solutions.

XML is a set of rules for encoding documents electronically. The characters, which make an XML document, are divided into structure and content. Strings, which compose of structures, are called tags. Tags begin with the character ‘<’ and end with the character ‘>’. Strings, which are not tags, constitute the content of an XML document. Element is a logical component of a document which begins with a start-tag and ends with a matching end-tag. Figure 1.1 is an example for XML. Figure 1.1(a) shows the textual syntax of XHTML \[8\] which is a reformulation of HTML as XML. Figure 1.1(b) represents Fig. 1.1(a) in a WEB browser.
Figure 1.1: Example of XHTML document. (a) The XML syntax of the XHTML document. It contains an html element (‘<html>’) with two nested elements: an empty header element (‘<head>’) and a body element (‘<body>’). The body contains two paragraphs (‘<p>’). Each paragraph contains text followed by an image element (‘<img>’). (b) The WEB browser representation of this XHTML document from (a).

The XML elements, which are nested in each other, form a logical node-labeled tree structure where the label of a node $v$ is the name of the element that $v$ represents. In this way, XML becomes a semi-structured data which presents data through a node-labeled graph. Figure [1.2] illustrates a tree structure of an XML document.

![Tree Structure](image)

Figure 1.2: The tree structure of the XML document in Fig. [1.1]

Figure [1.2] describes the XML processing tasks and their relations to the algorithms which are presented in this thesis.

Part II Introduction and Background
Query is a major task in processing XML that is stored in a DB. Due to XML popularity, query of XML documents attracts great interest. In semi-structured data model, there is no separation between data and schema. The nodes in a semi-structured model store the data. Instead of a single schema as in a relational model, the DB schema of a semi-structured model describes a collection of node-labeled graphs which the DB can accept. This provides a flexible data model. In order to support this flexibility, semi-structured data query languages are richer than the relational model of query languages. They express two pattern types: a graph structure pattern, which is called a structural-pattern, and a pattern in the data that is stored in graph nodes. The structural-pattern expresses the relations between nodes in the graph structure. Tree is a natural representation of structural-patterns. Each branch in the tree defines either the logical AND or the logical OR operators. Most of the existing algorithms for XML query of a DB decompose the tree structural-pattern into paths, process separately each path structural-patten and ‘stitch’ together the results. This type of processing is inefficient since it produces many redundant intermediate solutions which are filtered out when the intermediate solutions are ‘stitched’ together. This can be classified as a ‘local’ processing. An holistic structural-pattern processing, which processes the whole structural-

Part I: Introduction and Background
pattern tree in contrast to the ‘local’ approach, is proposed in the thesis. A well known structural-pattern tree is the twig (a small tree) pattern [106]. In this thesis, we develop the holistic twig pattern processing.

Structural-index and structural-join are two fundamental DB operations that were suggested to speed up data retrieval according to structural-patterns. Several algorithms were suggested for processing structural-join holistically. These algorithms did not provide a comprehensive and fundamental theory for understanding of holistic structural-pattern processing. No algorithm for holistic structural-index processing has been suggested. In this thesis, we present comprehensive and coherent methodology for holistic structural-pattern processing [129]. Based on this methodology, we implemented an holistic structural-join algorithm that is called TwigTA [128]. We also introduce a new family of holistic structural-index algorithms [127, 24]. Experimental results prove that the holistic structural-pattern processing methodology reduces substantially the number of intermediate solutions.

Accessing streams of XML is a major task in XML network applications. XML is a highly verbose language in relation especially to the structure duplication in the elements form. Therefore, XML textual data encoding is oversized. Due to the XML oversized encoding, XML accessing computationally expensive. In this thesis, we implemented several algorithms, which access XML streams, that optimize CPU and memory usages.

Accessing XML involves parsing and validation. Parsing checks whether an XML message is well-formed, i.e., whether it satisfies a list of syntax rules provided in its specification. This thesis implemented an XML streaming parser for smart cards [126]. Smart cards have limited memory. This XML parser focused on optimizing memory utilization.

In addition to being well-formed, an XML message may be invalid. Validation verifies that the XML message contains a reference to a schema and validates that its elements follow the grammatical rules which the schema specifies. This thesis presents a compression algorithm, which combines validation with access to compressed XML streams [74, 23]. It optimizes the XML access time by reducing the size of the XML textual encoding via compression and by accessing the compressed XML instead of accessing the original message.

Validation uses an XML schema. But XML streams are often distributed without any
XML schema attached to them. In such a case, the XML schema should be extracted from the XML message itself. This raises the problem how to extract a schematic information from an XML stream. A schema should tightly represent the data while being compact. The thesis suggests a new XML Schema extraction algorithm [122] if the Schema is missing.

1.2 Contributions of the thesis to XML DB query

This section outlines the theory and the DB applications which are presented in the thesis. They are based on tree automata theory. Figure 1.4 shows the relations among XML DB querying tasks, automata theory and the presented algorithms.

Figure 1.4: The relations among XML DB querying and automata applications according to the presented algorithms. A white box denotes both XML processing tasks and automata applications. A gray box denotes a related algorithm from the thesis.

In the presented methodology, the inputs for structural-pattern processing are formalized as automata. Throughout this thesis, we use two automata concepts: Finite State Automaton (FSA) and Tree Automaton (TA). The input to a FSA, which is also called by Finite Automaton (FA), is a string of input symbols. The FSA operates on each input symbol according to a transition function that updates the current state of the FSA. When the last input symbol is processed, then the FSA either accepts or rejects the
string depending on whether the current state is an accepting or a non-accepting state of the FSA. This way, the FSA recognizes a specific collection of strings.

TA describes sets of trees. In this thesis, we apply a bottom-up TA that processes trees from the “bottom” of the tree (leafs) to the “top” of the tree (root). A TA application to a node-labeled tree $T$ is denoted by $A(T)$. The $A(T)$ annotates a state to each node $v$ according to a transition function. The transitions of a TA are from states, which were annotated to children nodes of $v$, and from $v$ label to a state given by a transition function that annotates $v$. When $v$ is the root state, it either accepts or rejects the tree depending on whether the state, which annotates the root $v$, is in an accepting or a non-accepting state. This way, TA describes a specific set of trees. Figure 1.5 illustrates by example the $A(T)$ operation. $A(T)$ starts by applying transition 1 to leaf nodes 3 and 6 and by applying transition 2 to leaf 7. Then, $A(T)$ applies transition 3 to node 5. Finally transition 4 is applied to node 1.

\[\begin{array}{ccc}
1 & \text{Children States} & \text{Parent Color} \\
2 & \text{Parent State} \\
3 & \text{Parent State} \\
4 & \text{Parent State} \\
5 & \text{Parent State} \\
\end{array}\]

(a) $A$ transition table

(b) $A(T)$ operation

Figure 1.5: An example of $A(T)$ operation. (a) Description of the TA $A^{Twig}$ transition table. Each row denotes a transition from a tuple of children states and a color (label) of a parent node to a parent state. Each state is denoted by a regular triangle. (b) Description of the application of $A^{Twig}$ to $T$. The tree nodes are denoted by circles. Each node $v$ contains a state which annotates $v$ in the $A^{Twig}(T)$ operation.

‘ranked trees’ and ‘unranked trees’ are two types of bottom-up TA. The difference is in the transition function. Ranked trees have a finite set of children for each parent. Therefore, ranked tree transitions have a finite set of children states. A node in an unranked tree can have any number of children. Therefore, unranked TA transitions have an unknown number of children states. In order to process the input, the ordered TA transitions are extended by regular expressions (RE) [79]. Children states of an

Part I: Introduction and Background
unranked ordered TA transitions are described by RE over the TA states. The transition takes place if a string, which is composed of annotated children states, is accepted by a RE.

In recent years, several papers suggested to process twig patterns as tree automata. A TA methodology, which is presented in [42], was suggested to query XML data. This methodology, which is called Selecting Tree Automaton application on a tree, is denoted by $STA(T)$. $STA(T)$ is viewed as a tree $T$ query. The $STA(T)$ operation in [42] is called two ways Selecting Tree Automata. The two ways Selecting Tree Automata operation is given by a $ATw$ TA and by a $BTw$ FSA. This operation first applies the TA $ATw$ bottom-up and then applies the FSA $BTw$ top-down on the annotations which $ATw$ made. In this methodology, the unranked tree is first transformed into a binary tree $T$. Next, FSA $BTw$ and TA $ATw$ are constructed from the twig pattern and are applied to the binary tree $T$.

In the thesis, we define a new bottom tree automaton, called Unordered Unranked Tree Automaton (UUTA), which recognizes unordered trees. We also define the $STA(T)$ operation for UUTA. The UUTA operation is simple in several aspects. First, although UUTA operates on unranked trees, its transitions are similar to ranked TA transitions. The UUTA transitions are not extended with RE because they process the children states as sets and not as sequences. Further more, the UUTA $STA(T)$ operation requires no tree transformations as in [42]. The tree transformation helps in converting the ordered tree processing to an unordered trees processing. But the tree processing is already unordered in UUTA. Therefore, a tree transformation is not needed. Another source for simplicity is the fact that FSA $BTw$ is not constructed from the twig pattern but induced from $UUTA ATw$. In the thesis, we denote the induced FSA by $A_{pc}$ in order to relate it to $ATw$. Figure 1.6 illustrates the $STA(T)$ operation. The twig pattern searches for parent nodes where both have yellow and magenta colored children. The two-ways $STA(T)$ operation starts by applying $ATw$ to $T$ as described in Fig. 1.5(b). The top-down traverse, which is illustrated in Fig. 1.6(b), applies the $A_{pc}$, which is illustrated in Fig. 1.6(a), to states that are annotated by the $ATw$ application to $T$. The $A_{pc}$ application starts from the start state, which annotates the root, and continues top-down to the states, which annotates the children of the root. FSA $A_{pc}$ is inferred from the $ATw$. For example, the internal loop in Fig. 1.6(a) is inferred from transition 5 in Fig. 1.5(a),
The \(STA(T)\) operation selects nodes that were derived by the accepting states of \(AT_{pc}^{\text{twig}}\). The selected nodes compose the twig pattern. In this example, nodes 5, 6 and 7 were selected by the \(STA(T)\) operation.

![Diagram of \(A_T\) operation and \(STA(T)\) operation](image)

**Figure 1.6:** An example of \(STA(T)\) operation. (a) Description of the FSA \(A_T^{\text{pc}^{\text{twig}}}\) as a state machine, which was constructed from \(AT_{pc}^{\text{twig}}\) in Fig. 1.5(a). A state is denoted by a circle. The label of a circle denotes the state Id. A start state is denoted by incoming arrow and accepting states are denoted by double circles. (b) Description of the \(STA(T)\) top-down operation of \(A_T^{\text{pc}^{\text{twig}}}\). The tree nodes are denoted by circles. States are denoted by regular triangulares. Circles, which contain states, are selected by the \(STA(T)\) operation.

The \(STA(T)\) operation is insufficient to query an XML document, which is stored in DB, because the DB “physical” data model of XML documents is not in a tree format. In many cases, the DB “physical” data model decomposes the tree logical structure into a sequence of records. Loading the DB “physical” data model into RAM and converting it into a tree type data model is unapplicable if the size of the tree is too big.

We conclude that query on a tree is insufficient to support XML in DB. Therefore, enhanced representation of the stored XML is needed. Thus, instead of a tree query, we query a collection of trees that is represented by a tree automaton. In order to query a tree automaton \(ATree\), which accepts the stored XML document \(T\), we define a new STA methodology. The \(STA(ATree)\) operation applies \(AT_{pc}^{\text{twig}}\) query to UUTA \(ATree\).

As an example, we define the \(ATree\) in Fig. 1.7(a) that accepts the tree \(T\) in Fig. 1.6(b). The \(ATree\) accepts binary trees with three levels where its leafs are colored in cyan, magenta and yellow (transitions 1-3 in Fig. 1.7(a)). The middle level nodes are colored in red, blue and green (transitions 4-6 in Fig. 1.7(a)) and the root is colored in cyan.
black (transitions 7-9 in Fig. 1.7(a)). Figure 1.7(b) illustrates the STA applications of $A^{Tree}$, which was described in (a), and $A^{Twig}$, which was described in Fig. 1.5(a), in $T$.

Figure 1.7(b) is an example, together with Fig. 1.8, of the $STA(A^{Tree})$ operation that relates between states, which annotates common nodes, in STA applications of $A^{Tree}$ and $A^{Twig}$.

![Table and Diagram]

**Figure 1.7**: An example of $A^{Tree}$. (a) Description of the UUTA $A^{Tree}$ transitions table. Each row denotes a transition from a tuple of children states and a color (label) of a parent node to a parent state. Each $A^{Tree}$ state is denoted by an up-side-down triangle. (b) Description of the $STA(T)$ operation of both $A^{Tree}$ and $A^{Twig}$ on the tree $T$ in Fig. 1.6(b). The tree nodes are denoted by circles. Each selected node contains $A^{Tree}$ and $A^{Twig}$ states, which are denoted by an up-side-down triangle and a regular triangle, respectively.

The $STA(A^{Tree})$ operation constructs the FSA $(A^{Tree} \cap A^{Twig})_{pc}$ which defines the relations between the applications of both UUTA $A^{Twig}$ and UUTA $A^{Tree}$ to common trees. If there is a node $v$ in a tree $T$, which is selected by the $STA(T)$ operation of both UUTAs, then $v$ is annotated by both $q^{Tree}$ and $q^{Twig}$ where $(q^{Tree}, q^{Twig})$ is an accepting state of $(A^{Tree} \cap A^{Twig})_{pc}$. $STA(A^{Tree})$ selects the $A^{Tree}$ states which annotate nodes in $T$ that were selected by $STA(T)$ application of $A^{Twig}$ in $T$. The $STA(T)$ operation of $A^{Twig}$ selects nodes that compose a solution for the twig pattern. Therefore, $STA(A^{Tree})$ selects the states of $A^{Tree}$, which annotates nodes that compose a solution for the twig pattern. Figure 1.8 illustrates the $(A^{Tree} \cap A^{Twig})_{pc}$. We can see

Part II: Introduction and Background
that the accepting states of \((A^{Tree} \cap A^{Twig})_{pc}\) relate the \(A^{Twig}\) and \(A^{Tree}\) states that annotate common nodes in Fig. 1.7(b).

Figure 1.8: Illustration of FSA \((A^{Tree} \cap A^{Twig})_{pc}\) as a state machine, which is constructed from \(A^{Tree}\) in Fig. 1.7(b) and \(A^{Twig}\) in Fig. 1.5(a). A state is denoted by a circle. The label of this circle denotes the state Id. A state Id is composed of an up-side-down triangle, which is a \(A^{Tree}\) state, and a regular triangle that is a \(A^{Twig}\) state. A start state is denoted by incoming arrow and an accepting state is denoted by a double circle.

The \(STA(A^{Tree})\) operation enables us to represent both an XML structural summary and a partial knowledge of an XML document as UUTA \(A^{Tree}\). Representing both XML summary and partial knowledge of the XML document as UUTA enables us to implement an holistic twig pattern processing of XML that is stored in a DB. The flow of the algorithms, which process holistic structural join and holistic structural index, is given in Fig. 1.9. The preprocessing phase constructs the twig-UUTA \(A^{Twig}\) from the twig pattern. The index operation has two phases: offline and online. The offline phase constructs from the XML document structure \(T\) a summary-UUTA \(A^{Tree}\) which accepts \(T\). Then, the offline phase constructs the index by applying the \(STA(T)\) of \(A^{Tree}\) and mapping each node to the \(A^{Tree}\) state that selects it. The offline phase is performed once. The online phase prunes the indexed XML data according to \(A^{Twig}\) by applying the \(STA(A^{Tree})\) of \(A^{Twig}\) and selecting the states in \(A^{Tree}\) which annotate data nodes that match the twig pattern. Then, the holistic index operation prunes the indexed data nodes according to the selected states in \(A^{Tree}\). The index operation outputs the nodes of the XML document \(T\) that are mapped to the selected states in \(A^{Tree}\). In the running example, the offline operation maps each node to a state which was annotated in Fig. 1.7(b). Then, the online \(STA(A^{Tree})\) operation in Fig. 1.8 selects the \(A^{Tree}\) states with

Part I: Introduction and Background
yellow, magenta and red colors which annotate the nodes that compose the twig pattern. The index operation returns nodes 3, 5, 6 and 7.

The join operation iteratively extracts a fixed number of data nodes that were outputted by the index operation. Then, the join operation constructs a prediction-UUTA $A^{Tree}$ from these extracted nodes. This prediction-UUTA predicts the entire tree structure of the XML data. Next, the join applies the $STA(A^{Tree})$ to $A^{Twig}$ and selects the prediction-UUTA $A^{Tree}$ states which annotate the XML data nodes that match the twig pattern nodes. Then, the join operation outputs paths of data nodes which were annotated by the selected $A^{Tree}$ states. These output paths are the partial solutions for the twig pattern. The join operation sorts these output paths and merges them into twig pattern solutions.
Figure 1.9: General flow of the index and join operations.

1.3 Contributions of the thesis to XML processing in networks

This section shows the relations between XML processing in networks applications, automata, compression, learning and the algorithms which are presented in this thesis. Figure 1.10 illustrates these relations.
Parsing, which determines the well-formedness of XML streams, is a major task. The thesis presents the MinStAX streaming API, which is an XML API, that supplies XML parsing capabilities for devices with extremely limited memory. Reduction of memory utilization of XML parsing is not a simple task. XML is a textual format, therefore, its parsing is memory intensive. MinStAX combines tiny memory footprint with a fast enough runtime. One of the features of MinStAX, which is aimed to reduce the memory footprint, is a seamless support of the compressed XML API for closed environments. The support is ‘seamless’ since MinStAX supplies the same standard API for parsing of both standard XML and compressed XML. Therefore, XML format is ‘seamless’ to the user application.

MinStAX applies the Tagged Suboptimal Encoding (TSC) \cite{27} to compress XML streams. The thesis improves the TSC compression scheme in several ways \cite{130}. We show how to process the $TSC_k$ as a universal code. We introduce the $TSC_k^k$ as a family of universal codes where $TSC_0^k$ is the original $TSC$. Instead of constructing an optimal-code such as Huffman, we choose the best near-optimal-code from the $TSC_k^k$ family of universal prefix-codes. We introduce a fast decoding technique that uses compact transition tables in order to decode the compressed data as bytes. We adopt a pattern matching algorithm to use the same compact tables, which were used in the decoding process, in order to perform a fast pattern matching in the $TSC_k^k$ compressed domain.
These improvements make the $TSC^k$ compression scheme fast and compact. The encoding, decoding and search times of the $TSC^k$ compression scheme are similar. These make the $TSC^k$ an ideal compression scheme for parsing compressed XML data, which takes place in a streaming mode, in a machine/device that has a limited memory (several kilobytes).

Validation of XML streams is another major task in accessing XML streams. The thesis presents Dictionary Deterministic Pushdown Transducer (DPDT), which is an efficient and compact XML streams validator. DPDT is based on a new grammar, called D-grammar, which defines an XML structure for a specific XML schema that is given in a DTD format. DTD is chosen as an explanatory example. The grammar can be extended to define other deterministic XML scheme languages such as XML Scheme. DPDT is a validator for the DTD which the D-grammar reflects. The thesis also presents a parser generator which generates a DPDT from a DTD. The thesis presents a compression technique that encodes the DPDT validation choices instead of the textual tags that compose the XML structure. This encoding technique enhances the XML text compression twofold: first, there are less symbols to encode and second, the encoded structure symbols can predict the preceding text better than what the textual structure tags do.

The thesis also presents a new algorithm that extracts an XML schema from XML document. The proposed algorithm processes the structure of the XML document. It compares between sub-trees, merges similar nodes and describes the structure of the XML document as a new state machine type that is called two layer state machine (TLSM). TLSM accepts a grammar of ordered trees. TLSM holds in each state an internal state machine that describes the internal order of the children of each node in the ordered tree. A new algorithm, called XTLSM, which extracts TLSM from a given XML sample, is presented. The output from TLSM is tight and compact. Finally, a method to create a compact XML Schema from TLSM is also introduced.

1.4 Structure of the thesis

The thesis contains several parts. Here we provide a brief description of the structure of each part. Figure 1.11 shows the dependencies flow among the thesis parts and chapters.

Part I outlines the main methodologies, theories and algorithms. Chapter 1 provides the main motivation and background of the thesis. XML and automata theory are
described in Chapter 2.

**Part II** describes algorithms that perform query optimization for XML in a DB. Chapter 3 provides the needed preliminaries for the query optimization algorithms. It describes: 1. How XML is stored and queried in a DB; 2. The current TA applications for XML. Chapter 4 describes the TA methodology for UUTA. Chapter 5 describes the *TwigTA* holistic structural-join algorithm. Chapter 6 describes the holistic structural-index algorithm. Chapters 5 and 6 are based on the TA methodology that was described in Chapter 4.

**Part III** describes how to access XML in a limited memory device. Chapter 7 describes the implementation of a streaming XML API for smart cards. The implementation in Chapter 7 applies the *TSC* compression schema that is described in Chapter 8.

**Part IV** describes the *DPDT* algorithm for compressed validation of XML documents in Chapter 9.

**Part V** describes in Chapter 10 the *XLTSM* algorithm for XML Schema extraction from XML documents.
Figure 1.11: Flow of dependencies among the thesis parts (I-V) and chapters (1-10). Parts and chapters are denoted by gray and white rectangles, respectively.
Chapter 2
Background

2.1 XML

An XML document contains a single root element, elements, attributes and content. For simplicity, we do not use attributes and content in the XML examples. The algorithm treats the attributes and content as XML elements.

```
 1.  <a>
 2.    <b>
 3.      <c>
 4.        <e/>
 5.          <f><a><a/></a></a><a/></f>
 6.        </c>
 7.      </b>
 8.  </b>
 9.  </d/>
10.  </b>
11.  <b>
12.  <d>
13.    <e/>
14.      <f><a><a><a><a/></a></a><a/></a><a/></f>
15.    </d>
16.  </b>
17.  </a>
```

Figure 2.1: Example of an XML document

Example 2.1 Figure 2.1 is an example of a simple XML document. Each element begins with the start tag `<LABEL>` and ends with the end tag `</LABEL>`. Leaf elements
can also be represented by using the single start-end tag `<LABEL/>`. The root element `<a>` (lines 1-17) contains three instances of `<b>` elements:

**First instance** of `<b>` (lines 2-7) contains a single `<c>` element (lines 3-6). This element contains two elements: an empty `<e>` (line 4) followed by `<f>` (line 5).

**Second instance** of `<b>` (lines 8-10) contains a single empty `<d>` element.

**Third instance** of `<b>` (lines 11-16) contains a single `<d>` element, which contains elements like `<c>` (lines 3-6) `<e>` and `<f>`.

Both `<f>` elements (lines 5 and 14) contain recursive instances of `<a>` elements. We can assume that these `<a>` elements take a different role than the role of the root `<a>` element (line 1-17).

### 2.1.1 XML Schema (XSD)

*XML Schema* is a type of XML schema language. XSD expresses a schema which is a set of rules that validate an XML document. The XML Schema data model contains the vocabulary (element and attribute names), the content model (relationships and structure) and the data type. We focus on the structure and on the relationships among the XML elements only. All the work can be extended to include attributes and data.
Figure 2.2: An example of XML schema which recognizes the XML document in Fig 2.1.

Example 2.2 Figure 2.2 is an example for XML schema. This XML schema validates the XML document in Fig. 2.1. XSD is a complex language with many keywords. For our purpose, we focus on:

**element** is a simple element definition. It cannot contain sub-elements. It is used also for a declaration for the root element of the XML;

**complexType** is a complex element definition. It can contain sub-elements like sequence, choice, etc;

**sequence** is a sequence of elements. It must appear in exact order in the XML document;

**choice** is a choice between sub-elements. Only one sub-element must appear in the XML document.

Each of the XSD components may have quantity restrictions:
**minOccurs:** A non-negative integer that indicates the minimal occurrences of such element in the XML document;

**maxOccurs:** Either a non-negative integer or unbounded that indicates the maximal occurrences of such element in the XML document.

The following refers to Fig. 2.2. The first element in this sample (line 1) is the root element of the XSD. This means that all XML documents, which can be validated using this XSD, must contain a single root element that is labeled as a. The root element is of `complexType` called s1 (lines 2-6). Any element in the XML document, which is of this type, must contain either a sequence of zeroes or one occurrence of s2 type element that is labeled as b. s2 type elements in the XML document will contain a single s3 element. Its label will be either c or d (lines 7-12). s3 type elements in the XML document may be either empty elements or contain s4 element that is labeled as e followed by s5 element that is labeled as f (lines 13-18). s4 type elements must be empty elements (lines 19-20). s5 type elements may contain a single s5 labeled as a (lines 21-23). From this XML schema we get that the root element of the XML is labeled a. There is another element called a. The two elements do not share the same type. □

### 2.1.2 Document Type Definition (DTD)

DTD is another type of XML schema language. DTD declares the elements names, the allowed element sequences and the elements attributes. Fig 2.3 shows the DTD of the XHTML example introduced in figure 1.1. This DTD defines a subset of the XHTML standard DTD.
Figure 2.3: DTD of the XHTML example introduced in figure [11]. The DTD defines the XHTML subset of XML. A html element ‘html’ with an header and a body elements. The header element (‘head’) has an optional ‘title’ element. The ‘body’ element contains multiple paragraph elements (‘p’). Each paragraph contains a mixture of image elements (‘img’) and text. The image elements are empty elements.

There are two relevant types of declarations in DTD:

1. **Element type declaration** identifies the name of declared elements (element_name) and the nature of its content (content_model) as follows: ‘<!ELEMENT element_name content_model ‘>’. The content model defines what an element may contain between start-tag and end-tag. The content model is defined with RE. There are three types of content-models: *Element content* solely contains elements, *Mixture of content* mixes text (the ‘#PCDATA’ special symbol) with other elements and the *EMPTY* content model indicates that the element has no content.

2. **Attribute list declaration** identifies the element that has the attributes (element_name), its attributes (att_name), the value types of the attributes (value_type) and the default values (default_value). Its format is: ‘<!ATTLIST element_name att_name value_type default_value >’.

Part [II] Introduction and Background
value_type default_value)+ '>'.

2.2 XML as a semi-structured data

A semi-structured data is a directed graph. Definition 2.1 defines a directed graph.

**Definition 2.1 Directed graph** $G \triangleq (V^G, E^G)$ is a graph where $V^G$ is a set of nodes and $E^G$ is a set of edges that connect them. The edges are directed. The edge is incoming to a node and outgoing from another. A path in a graph is a sequence of vertices $v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E^G$, $1 \leq i < n$. The distance between two vertices in a graph is the number of edges in the shortest path that connects them.

XML documents are semi-structured. In this thesis we model XML documents as trees. Definition 2.2 defines a tree. where the nodes represent elements, attributes and texts. The edges represent element-subelement, element-attribute and element-text pairs.

**Definition 2.2 Tree** $T = (V^T, E^T)$ is a directed graph with the following constraint: all its nodes except one have a single incoming edge. The exceptional node is called root and has no incoming edge. The function $\text{root}(T)$ returns the root of $T$. A node without outgoing edge is called leaf. A node $u_p$ is called parent of a node $v_c$ if there is an outgoing edge from $u_p$ into $v_c$. The node $v_c$ is called a child of node $u_p$. The function $\text{children}(T, v)$ returns all the children for the node $v \in V^T$. There is no a priori bound on the number of children nodes in $\text{children}(T, v)$. Therefore, these trees are unranked. A node $v_d$ is called descendant of a node $u_p$ if there is a path from $u_p$ to $v_d$. The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path). A subtree is a portion of a tree data structure that can be viewed as a complete tree by itself. Any node in a tree $T$, together with its decedents, comprise a subtree of $T$. The subtree that corresponds to the root node is the entire tree. The subtree that corresponds to any other node is called a proper subtree.

A semi-structured data is a node-labeled graph. Definition 2.3 defines node-labeled graphs and trees.

**Definition 2.3 Node-labeled graph** $G$ is a tuple $(V^G, E^G, \text{label}^G)$ where $\text{label}^G : V^G \rightarrow \Sigma$ assigns labels from an alphabet $\Sigma$ to nodes $V^G$. These assignments do not have to be unique, i.e. different nodes can have the same label. A node-labeled tree is the tuple $(V^T, E^T, \text{label}^T)$ where the graph has a tree form.
Figure 2.4: An illustration of node-labeled graph \( T \). Each node is denoted by a circle. The label of each node is a textual label inside its circle. The id of each node is placed externally to its circle.

Example 2.3 Figure 2.4 illustrates an node-labeled tree \( T \) that is constructed from the XML document in Fig. 2.1: \( T = (V^T, E^T, label^T) \) where: \( \Sigma = \{a, b, c, d, e, f\} \), \( V^T = \{v_1, v_2, v_3...v_{17}\} \), \( root(T) = v_1 \). The leafs are \( v_8, v_{10}, v_{14}, v_{17} \). Parent of \( v_6 \) is \( v_3 \) and \( v_2, \ldots, v_{17} \) are descendents of the ancestor \( v_1 \). \( children(T, v_1) = \{v_2, v_3, v_4\} \).

This thesis examines different languages that contain strings of node labels. Definition 2.4 detail these languages.

Definition 2.4 languages of node-labeled graphs

Label-Path given a node-labeled graph \( G = (V^G, E^G, label^G) \). A label-path: \( label(v_1), \ldots, label(v_n) \) is a sequence of symbols where \( v_1, \ldots, v_n \) is a path in \( G \).

Language of a node Given a node-labeled rooted graph \( G = (V^G, E^G, label^G, root^G) \). The language of a node \( v \in V^G \), which is denoted by \( L(G, v) \) is the collection of label-paths of paths from \( root^G \) to \( v \). When \( T \) is a tree, \( L(T, v) = \{w_v\} \) contains a single word. We denote, hereinafter, this word by \( w_v \).

Language of a graph Given a node-labeled rooted graph \( G \). \( L(G) = \bigcup_{v \in V^G} L(G, v) \).
Language of a tree  Given a node-labeled tree $T$ we examine two languages $L(T) = \bigcup_{v \in V_T} w_v$ and $L_{leaf}(T) = \{ w_v | v \text{ is a leaf} \}$. 

Example 2.4 Figure 2.4 illustrates an node-labeled tree $T$ that is constructed from the XML document in Fig. 2.1. Label of node $v_8 (\text{label}_T(v_8))$ is ‘e’. Label-Path of node $v_8 (w_{v_8})$ is ‘abce’. The languages of node-labeled tree $T$ in Fig. 2.4 are: $L_{leaf}(T) = \{ abce, abcfaa, abd, abde, abdfaaa \}$, $L(T) = L_{leaf}(T) \cup \{ a, ab, abc, abcf, abcfa, abdf, abdfaa, abdfaaa \}$. $w_{v_6} = abd.$

XML schema handles XML document as an Ordered node-Labeled Tree, which is denoted, hereinafter, by $OLT$. Let $T = (V, E, label_T)$ be an $OLT$. Then, we use the following notations:

- $\lambda$ is the error node;
- $\text{MostLeft}(v) : V_T \mapsto V_T \cup \{ \lambda \}$ is the left most child of $v$;
- $\text{MostRight}(v) : V_T \mapsto V_T \cup \{ \lambda \}$ is the right most child of $v$;
- $\text{Left}(v) : V_T \mapsto V_T \cup \{ \lambda \}$ is the left sibling of $v$.

$\text{Left}(v) = \lambda \iff v$ has no left sibling;

$\text{Right}(v) : V_T \mapsto V_T \cup \{ \lambda \}$ is the right sibling of $v$.

$\text{Right}(v) = \lambda \iff v$ has no right sibling;

$\text{Children}(v) : V_T \mapsto \text{Sequence}(\Sigma)$ is a sequence of children symbols according to the children order.

Example 2.5 Figure 2.4 illustrates an node-labeled tree $T$ that is constructed from the XML document in Fig. 2.1. $\text{Parent}(v_2) = v_1$, $\text{Children}(v_1) = (v_2, v_3, v_4)$, $\text{MostLeft}(v_1) = v_2$, $\text{Right}(v_2) = v_3$, $\text{Left}(v_2) = \lambda.$

A twig pattern ignores the order of the nodes. Therefore, in a twig pattern processing context, we define an XML document as unordered node-labeled tree. Definition 2.2 defines a unordered tree.

Part I: Introduction and Background
2.3 XML query languages

Modern applications face the challenge of dealing with complex structured and semi-structured data. Though XML documents are the most known, chemical compounds, CAD drawings, web-sites and many other applications have to deal with similar problems. In such environments, ordered and unordered tree pattern matching are the fundamental search operations. The goal of this thesis is to evaluate a query tree pattern.

Tree patterns are expressed in *XPath* [1] which is the querying language standard for XML. *XPath* is a query language for selecting nodes from an XML document. In addition, XPath may be used to compute values (e.g., strings, numbers, or Boolean values) from the content of an XML document. The XPath language is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria.

*XPath* expresses tree patterns but tree patterns query language is not standardized. For example tree patterns can be expressed in *Datalog* [68] which is a query language for deductive databases that store semi-structured data.

Recently, a twig pattern was suggested as a formal representation for tree patterns of semi-structured languages [106]. The labels of a twig tree are a subset of queried semi-structured data labels. A twig pattern also maps each edge to either A-D or P-C. A twig pattern expresses the structural portion of queries which are written in a query language.

This thesis details twig patterns in *XPath* syntax. A twig pattern represents the structural criteria of *XPath* which is the dominant query language for XML. *XPath* has a string format. Each node is denoted by its label. A P-C node relation is denoted by ‘/’. An A-D node relation is denoted by ‘//’ and branch is denoted by square brackets. For example, the string ‘/a[/b]//c’ is equivalent to a twig pattern with the label ‘a’ in the root and children with the labels ‘b’ and ‘c’ where the last has an A-D node relation with the root.

*XPath* is more expressive the twig patterns. *XPath* express other nodes relations then P-C and A-D. For example, a node relation of two siblings of the same parent. These relations are out of the scope of this thesis. However, the automata model that is presented in thesis is general and can be adopted to support additional *XPath* nodes relations.
A twig pattern is defined as a node-labeled tree \( T^\text{Twig} \triangleq (V^\text{Twig}, E^\text{Twig}, \text{label}^\text{Twig}, \text{type}^\text{Twig}) \). Twig pattern nodes can be elements, attributes and texts. The \( \text{label}^\text{Twig} : V^\text{Twig} \mapsto \Sigma^\text{Twig} \) maps each node to a label \( \Sigma^\text{Twig} \). The \( \text{type}^\text{Twig} \) function maps each edge to its nodes-relation type. Twig pattern edges types either have P-C node relationships (denoted by ‘/’) or have A-D nodes relationships (denoted by ‘//’). If the number of children of a node \( v \) is greater than one, then \( v \) is called a branching node. Otherwise, if a node \( v \) has only one child then it is called a non-branching node. Figure 2.5 describes a twig pattern in a graphical notation. It is a 2–branchnode twig pattern because two nodes ‘a’ and ‘b’ have more then one child (branch) each.

![Figure 2.5: A graphical representation of the twig pattern ‘//a[/c]/b[/e]/d’. A twig pattern node \( v \) is denoted by a circle. The label of a node \( v \) is ‘label^\text{Twig}(v)’. Edges \( e_1, e_2 \in E^\text{Twig} \) with \( \text{type}^\text{Twig}(e_1) = \text{P-C} \) and \( \text{type}^\text{Twig}(e_2) = \text{A-D} \) are denoted by a single line and a double line, respectively.](image)

### 2.4 Automata and languages

In the presented methodology, the inputs for XML processing are formalized as automata. In this section, we provide the background that is needed for this formalism. We describe two concepts: Finite State Automaton (FSA), which accepts strings, and Tree Automaton (TA) that accepts trees. Definition 2.5 defines a FSA.

**Definition 2.5** FSA is a tuple \( (Q^A, \Sigma, q_0^A, F^A, \delta^A) \) where \( \Sigma \) is the input alphabet (a finite non-empty set of symbols), \( Q^A \) is a finite non-empty set of states, \( q_0^A \in Q \) is an initial state, \( \delta^A \) is the state-transition function \( \delta^A(\sigma, a) \in Q^A \) and \( F^A \subseteq Q^A \). In this thesis we refer to a FSA also as a final state machine (FSM) and a final automaton (FA). The FSM states \( q \in Q^A \) are alternatively denoted by \( s \in S^A \).
Figure 2.6: Example of a FSM. Each state is denoted by a circle. Double circles denote accepting states. The state-transition function is presented by arrows. An arrow without source circle points to the start state of the FSM. For simplicity, the error state and all its transitions are not illustrated.

Example 2.6 Figure 2.6 illustrates a finite state machine that is defined by $A = (\Sigma^A, S^A, s_0^A, \delta^A, F^A)$ where:

- $\Sigma^A = \{a, b, c, d, e, f\}$,
- $S^A = \{s_0, s_1, s_2, s_3, s_4, s_5\}$,
- $F^A = \{s_3, s_4, s_5\}$,
- $\delta^A$:

<table>
<thead>
<tr>
<th></th>
<th>$s_0$</th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$s_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>$s_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>$s_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$s_4$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$s_5$</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$s_5$</td>
</tr>
</tbody>
</table>

Definition 2.6 details some properties and operations of the $FSA$ that are needed in this thesis.

**Definition 2.6** Properties and operations of $FSA$:

**Derivation of word** $\delta^A(q, w) \rightarrow q'$ is the derivation of word $w \in \Sigma^*$ from state $q \in Q^A$. We denote a derivation $\delta^A(q_0^A, w) \rightarrow q'$ from the start state by $\delta^A(w) \rightarrow q'$.

**$L(A)$** is the language of words $w$ that $FA$ accepts. More formally, $L(A) = \{w | \delta^A(w) \rightarrow q \text{ and } q \in F^A\}$. 

Part I: Introduction and Background
Accessible state $q \in Q^A$ is a state which participate in a derivation of a word $w \in L(A)$. State $q \in Q^A$ is accessible if $w = \alpha \beta$ where $\delta^A(\alpha) \rightarrow q, \delta(q, \beta) \rightarrow q'$ and $q' \in F^A$.

Intersection: Given finite automata $A^1 = (Q^{A^1}, \Sigma, q_0^{A^1}, F^{A^1}, \delta^{A^1})$ and $A^2 = (Q^{A^2}, \Sigma, q_0^{A^2}, F^{A^2}, \delta^{A^2})$. We define $A^1 \cap A^2 \overset{\Delta}{=} (Q^{A^1} \times Q^{A^2}, \Sigma, (q_0^{A^1}, q_0^{A^2}), F^{A^1} \times F^{A^2}, \delta^{A^1 \cap A^2}((q, p), a) = \langle \delta^{A^1}(q, a), \delta^{A^2}(p, a) \rangle$.

**Example 2.7** The language of the FSM $A$ from Fig. 2.6 contains: $abce, abde, abcfa, abcfaa, abcfaaa, abdf, abdfaaaa...$ We can also describe $L(A)$ as all the words that match the RE: $ab (c | d) (e | f (a *))$. The derivation word ‘abc’ by FSM $A$ is $\delta^A(abc) \rightarrow s_3$. Therefore, state $s_3$ is accessible.

When two node-labeled trees $t_i$ and $t_j$ produce the same language $L(t_i) = L(t_j)$ then an FSA can not distinguish between the trees that represent them.

**Example 2.8** Figure 2.7 illustrates several node-labeled trees. Each tree language is a subset of the language of the FSM in Fig. 2.6. By comparing between the trees $t_4$ and $t_5$, we get that the difference between these trees is in the nodes order. Since $L(t_4) = L(t_5)$ we cannot use an FSA to distinguish between $t_4$ and $t_5$.

![Figure 2.7: Illustration of several OLTs that produce words that the FSM in Fig. 2.6 accepts.](image)

Part II: Introduction and Background
In order to differentiate between node-labeled trees $t_4$ and $t_5$, we apply a ordered Tree Automaton (TA). In this thesis, we use a bottom-up TA that processes trees from the “bottom” of the tree (tree leaves) to the “top” of the tree (root). An input to a bottom-up TA is a labeled tree whose labels are the TA input symbols. The TA traverses a tree from leaves to root. The TA annotates a state to each node according to a transition function. The transitions of a TA are from states, which were annotated to children nodes, and from a node label to a state given by a transition function. When the root state is reached, it either accepts or rejects the tree depending on whether the root state is in an accepting or a non-accepting state. This way, a TA describes a specific set of trees. Definition 2.7 defines a TA.

**Definition 2.7** a bottom-up finite ordered tree automaton is defined by $(Q^A, \Sigma, F^A, \delta^A)$ where $Q^A$ is a set of states, $\Sigma$ is a finite set of input symbols, $F^A \subset Q^A$ and $\delta^A$ is a set of transitions. A transition is a rewrite rule from a string, which is composed from children states, to a parent state. Thus, state of a node is deduced from the states of its children. There is no initial state but the transition rules for constant symbols (leaves) can be considered as initial states. A tree is accepted if the state at the root is an accepting state. TA defines a collection of trees. Given a TA $A$, The *(Language of a TA)* $L(A)$ is the set of all the tree that are accepted by $A$.

There are two types of ordered bottom-up TA: ‘ranked trees’ and ‘unranked trees’. The difference is in the transition function. Ranked trees have a finite set of children for each parent. Ranked tree transitions are for a finite set of children states. A node in an unranked tree can have any number of children. Therefore, the input for an unranked TA transition has an unknown number of states. In order to process the input, the transition is extended by REs [79]. Children of a transition are described by a RE over the TA states. The transition takes place if a string, which is composed of annotated children states, is accepted by a transition RE. A tree automaton run relates tree vertices to their annotated states.

**Example 2.9** Figure 2.2 gives an example for an XML Schema. This XML Schema can be translated into an unranked ordered TA $A = (Q^A, \Sigma, F^A, \delta^A)$ where $\Sigma = \{a, b, c, d, e, f\}$, $Q^A = \{q_{s1}, q_{s2}, q_{s3}, q_{s4}, q_{s5}\}$ translates each XML schema element definition $s_i$ into a state $q_{s_i}$. $F^A = \{q_{s1}\}$. The transitions function include the following transitions: 1.$\delta^A((q_{s5}),?, a) \rightarrow q_{s5}$, 2.$\delta^A((q_{s5}), f) \rightarrow q_{s5}$, 3.$\delta^A((q_{s5}), c) \rightarrow q_{s4}$, 4.$\delta^A((q_{s4}, q_{s5}), c) \rightarrow q_{s3}$, 5.$\delta^A((q_{s4}, q_{s5}), d) \rightarrow q_{s3}$, 6.$\delta^A((q_{s3}), b) \rightarrow q_{s2}$, 7.$\delta^A((q_{s2}), *, a) \rightarrow q_{s1}$. We use RE operators ‘?’ and ‘*’. The question mark indicates there is zero or one of the preceding element and the asterisk indicates there are zero or more of the preceding element. The empty brackets ‘()’ indicates the empty sequence. A node-labeled tree $T$ is valid if and only if $T \in L(A)$.

For example, we examine the state that annotates $v_5$. The application of $A$ on $T$ in Fig. 2.4 starts in the leaves. Leaf $v_{14}$ is annotated by $q_{s5}$ and leaf $v_8$ is annotated by $q_{s4}$ according to transitions 1 and 3, respectively. Then, $v_{12}$ is annotated by $q_{s5}$ according to transition 1. Next, $v_9$ is annotated by $q_{s5}$ according to transition 2. Finally, $v_5$ is annotated by $q_{s3}$ according to transition 4.

Part I: Introduction and Background
Part II
Querying of XML in a DB
Chapter 3

Preliminaries for querying of XML in a DB

3.1 DB and XML

A DB is a collection of information organized in a structured way so that it can be easily retrieved, managed and updated. The data in a DB is organized according to a model. In this thesis, we consider two models. The dominant model, which is a relational model, and the semi-structured model, which defines XML data. In the low level, DB model contains a collection of DB files. Each DB file is a collection of DB records.

In a relational DB, a record contains a fixed set of fields. A field is a content of a certain data type: numeric, character, logic, date, etc. A DB schema is given by a formal language description. A DB schema defines the files, the records and the fields in each file and the relationships between them. A relational DB has a single known schema.

Query languages enable to retrieve data from a DB. Retrieval requests from a DB are expressed by a query language. Each type of a data model such as relational and semi-structured has its own set of query languages. The common standard relational query language is SQL [144].

Database indexes speed up data retrieval. They are the same as book indexes, thus, providing the DB with a quick jump to find the full reference. The indexes are additional data structures that store references in the actual records. For example, an index can be a hash table that stores all the DB records references in buckets sorted by specific values. When the user requests to retrieve all the records that match this value, then, the DB retrieves the requested records by first retrieving the references from the hash table and then retrieving one by one the DB records from the DB file. It is faster than performing a full traverse (scan) of the DB file records. When data records are retrieved from multiple
related files, the DB join mechanism combines these records, creates a joined records set and returns it to the user. A join mechanism must efficiently join records from different files according to a join criterion that relates multiple DB files. Efficient indexing and join operations, which extract a minimal number of records from DB files, are the basis for efficient query processing.

A semi-structured DB model presents data through a labeled graph. In this model, there is no separation between data and schema; therefore, the size of the graph structure depends on the usage goals. The nodes in a semi-structured model store the data. The nodes in the graph model are equivalent to records and fields in a relational DB model. They are equivalent to fields because they store the data itself. They are equivalent to records because they refer to a collection of fields that are children nodes. Thus, instead of having a fixed records set, the representation is done by a graph. Instead of a single schema as in a relational DB, the DB schema of a semi-structured model describes a collection of node-labeled graphs that the DB can accept. This provides a flexible semi-structured data management.

In order to support this flexibility, semi-structured data query languages are “richer” than query languages that represent relational model. A semi-structured data query obeys two patterns types: 1. Structural patterns on a graph structure; 2. Data patterns on the data which is stored in the graph nodes. The data patterns are the same as in SQL. The data patterns describe a Boolean expression of the data values. The structural-patterns represents relations between nodes in a graph structure. There are several structural relation types. The most common are: 1. Parent-child (P-C): If \( v_p \) is a parent of node \( v_c \) then \((v_p, v_c)\) has a P-C relation; 2. Ancestor - descendant (A-D): if node \( v_a \) is a parent of node \( v_d \) or if nodes \((v_c, v_d)\) have an A-D relation, where node \( v_c \) is a child of node \( v_a \), then the nodes \((v_a, v_d)\) have an A-D relation.

The main advantage of semi-structured model is in its flexible (“loose”) format. The primary trade-off being made by using a semi-structured model is that queries cannot be answered efficiently as in a relational DB due to an additional meta data. The motivation for using the tree automata methodology in this thesis is to eliminate this tradeoff by speeding up handling more efficiently the processing of the meta data. In view of the inefficiencies in retrieving semi-structured data to provide solutions to queries on such data in a relational DB environment, there is a need to make these methods more efficient.

Part II Querying of XML in a DB
3.1.1 XML storage model

In this thesis we examine two XML storage models. The “logical” model access the stored XML document as a node-labeled tree \( T = (V^T, E^T, \text{label}^T) \). The “logical” model does not consider how \( T \) is stored in the DB. The “logical” model is similar to the Object Exchange Model (OEM) \[112\]. We use the “logical” model in the index algorithm in Chapter 6. Fig. 3.1 illustrates an XML document that is stored in a “logical” model.

![XML Data Tree]

Figure 3.1: A sample from an XML document \( T \) that is composed only from elements and is stored in a DB. The XML document is presented as a node-labeled tree. A circle denotes a node \( v_i \). The label of node \( v_i \) has two lines. The top line is \( \text{label}^T(v_i) \). The bottom line is in the format ‘&\text{i}’.

But “Real” XML data tree is not stored as a whole in a DB because its size is too big to be loaded into a computer memory. The “physical” storage of XML data splits the tree into smaller subtrees. For relational DB, a tree form is not a natural representation. The common way (see \[106\]) to store semi-structured data in a relational DB is to split the tree into its smallest subtrees i.e. nodes. Each node in a semi-structured model is stored as a record. The join algorithm in Chapter 5 use such a “physical” storage model.

Part II: Querying of XML in a DB
for XML, which is called a region encoding, that is based on a labeling scheme that encodes each element in an XML DB by its positional information.

We denote the region encoding of a node \( v \) by the label \( R_v = (\text{start}_v, \text{end}_v, \text{level}_v) \) where \( \text{start}_v \) is the position in the tree from which a DFS based traverse of a node \( v \) starts, \( \text{end}_v \) is the position in the tree from which the DFS based traverse of a node \( v \) ends and \( \text{level}_v \) is the tree level of node \( v \). Region encoding supports an efficient evaluation of the structural relationships between two nodes \( v_i \) and \( v_j \) in the XML tree. Let \( R_{v_i} = (\text{start}_{v_i}, \text{end}_{v_i}, \text{level}_{v_i}) \) and \( R_{v_j} = (\text{start}_{v_j}, \text{end}_{v_j}, \text{level}_{v_j}) \) be two nodes in the tree. A node \( v_i \) is an ancestor of a node \( v_j \) if and only if \( \text{start}_{v_i} < \text{start}_{v_j} < \text{end}_{v_i} \). To have a P-C nodes relationship, \( \text{level}_{v_i} = \text{level}_{v_j} - 1 \) and node \( v_i \) has to be an ancestor of node \( v_j \).

Figure 3.2 describes the region encoding labels of the XML document in Fig 3.1. The region encoding of the root is \((0, 29, 1)\). It means that the DFS based traverse of the region encoding starts with the root element at position 0 and level 1. The DFS based traverse of the region encoding ends in the root itself after visiting 30 elements. There are 15 elements and each element in the traverse is visited twice.
Figure 3.2: Region encoding of the XML document in Fig [3.1]. The XML document is presented as a tree. Each node in the tree represents an element in the XML document. A node $v$ has a label that is composed from two lines. The top line denotes the name of the element $v$. The bottom line denotes the label $R_v$.

In the labeling scheme storage model, an XML document is clustered into streams. Each stream groups together all the encoded labels of elements with the same tag name. The encoded labels of the elements in the stream are ordered. In the region encoding case, the labels in the stream are ordered by their start position. In order to describe an extraction of a label encoded data in a clearer way, we use the cursor concept. The stream has an imaginary cursor which can either move to the next element or read the element that the cursor refers to. We denote a stream of a tree $T$ by $\text{cursor}^T$. The cursor splits the stream into two parts: $\text{head}(\text{cursor}^T)$, which is the stream’s first element, and $\text{tail}(\text{cursor}^T)$, which is the rest of the stream. Only the head of a stream can be read but not its tail’s portion. For example, the stream of elements $a$ in the XML document, which is presented in Fig. [3.2] is $\text{cursor}^T = \{(0, 29, 1), (2, 5, 3), (8, 17, 3), (22, 27, 3)\}$. The encoded label of the root is $\text{head}(\text{cursor}^T) = (0, 29, 1)$.

The following notation of the labeling scheme storage model and basic data struc-
tures, which are used by algorithms in Chapter 5 and the basic operations on these data structures, are described.

Positions\(T\): Let \(T\) be an XML labeled-tree, \(\text{Start}\(T\) \(=\) \{\(\text{start}_v \mid v \in V^T\}\) and \(\text{End}\(T\) \(=\) \{\(\text{end}_v \mid v \in V^T\}\). Positions\(T\) \(=\) \(\text{Start}\(T\) \(\cup\) \(\text{End}\(T\)\). In other words, it is numbered 1, \ldots, \(n\) where \(n = 2 \cdot |V^T|\).

Nodes\(T\), which is the set of labels \(\{R_v \mid v \in V^T\}\), denotes the set of nodes labels in the Twig\(T\) algorithm.

Cursor\(T\) (also called Stream) is a sequence \(R_{v_1}, \ldots, R_{v_n}\). The nodes are ordered by \(\text{start}_{v_i}\). Let \(R_{v_i}, R_{v_j} \in \text{Cursor}\(T\)\). \(i > j\) if and only if \(\text{start}_{v_i} > \text{start}_{v_j}\). \(\text{head}(\text{Cursor}\(T\))\) returns the first node in the \(\text{Cursor}\(T\)\). \(\text{tail}(\text{Cursor}\(T\))\) returns \(\text{Cursor}\(T\)\) without the head(\(\text{Cursor}\(T\))).

Database\(D^T\) maps \(D^T : \Sigma \to \text{Cursor}\(T\)\). Let \(a \in \Sigma\) then \(\text{Cursor}\(D^T(a)\) contains all \(R_v\) where \(\text{label}\(T(v)\) = a\).

ancestor: Let \(R_{v_i} \in \text{Nodes}\(T\)\) and \(pos \in \text{Positions}\(T\)\). ancestor:Nodes\(T\) \(\times\) Positions\(T\) \(\to\) \(\{0, 1\}\) is defined by \(\text{ancestor}(R_{v_i}, \text{pos}) = 1\) if \(\text{start}_{v_i} < \text{pos} < \text{end}_{v_i}\), otherwise 0.

parent: Assume \(R_{v_i}, R_{v_j} \in \text{Nodes}\(T\)\). parent:Nodes\(T\) \(\times\) Nodes\(T\) \(\to\) \(\{0, 1\}\) is defined as \(\text{parent}(R_{v_i}, R_{v_j}) = 1\) if \(\text{ancestor}(R_{v_i}, \text{start}_{v_j}) = 1\) and \(\text{level}_{v_i} = \text{level}_{v_j} - 1\), otherwise 0.

minimal_ancestor: Let \(S \subseteq \text{Nodes}\(T\), \(R_{v_i} \in S\) and \(pos \in \text{Positions}\(T\)\). minimal_ancestor: 2\(\text{Nodes}\(T\) \(\times\) Positions\(T\) \(\to\) Nodes\(T\) is defined by minimal_ancestor(S, pos) \(=\) \(R_{v_i}\). \(R_{v_j}\) satisfies ancestor(\(R_{v_i}, \text{pos}\)) \(=\) 1 if another node, \(R_{v_j} \in S\), also satisfies ancestor(\(R_{v_j}, \text{pos}\)) \(=\) 1 then ancestor(\(R_{v_j}, \text{start}_{v_j}\)) \(=\) 1.

Given \(D^T\). min_future_node: \(D^T \to \text{Nodes}\(T\)\), which is defined by \(R_{v_j} = \text{min_future_node}(D^T)\), satisfies \(\text{start}_{v_j} \geq \text{start}_{v_i}\) for every \(R_{v_j} = \text{head}(D^T(a))\)

where \(a \neq \text{label}(v_i)\). min_future: Database\(T \to \text{Positions}\(T\)\) is defined by \(\text{min_future}(D^T) = \text{start}_v\) where \(R_v = \text{min_future_node}(D^T)\).

Example 3.1 demonstrates these notations.
Example 3.1 The XML tree $T$ in Fig. 3.2 illustrates an XML region encoding. $D^T$ has the following structure: $D^T(a) = \{(0, 29, 1), (2, 5, 3), (8, 17, 3), (22, 27, 3)\}$, $D(b) = \{(1, 6, 2), (7, 18, 2), (11, 16, 4), (19, 28, 2)\}$, $D^T(c) = \{(3, 4, 4), (9, 10, 4)\}$, $D^T(d) = \{(12, 13, 5), (20, 21, 3)\}$ and $D^T(e) = \{(14, 15, 5), (24, 25, 5)\}$. There are 15 elements in the XML document. $\text{Positions} = 0, \ldots, 29$. $|\text{Positions}| = 30 = 2 \cdot 15$. $\text{min\_future\_node}(D^T) = (0, 29, 1)$ and $\text{min\_future}(D^T) = 0$. □

3.1.2 XML querying model

A solution (also called a match) of a twig pattern $Q$ in a document $T = (V^T, E^T, \text{label}^T)$, which is stored in $D^T$ in a labeling scheme storage model, maps distinct labels $R_{v_{D_1}}, \ldots, R_{v_{D_n}}$ into twig pattern nodes $v_{Q_1}, \ldots, v_{Q_n}$. This solution maps each $R_{v_{D_i}}$ in $D^T$ into a query node $V_{Q_i}$ in $Q$ such that nodes relationships between query nodes $v_{Q_i}$ and $v_{Q_j}$ are satisfied by the corresponding DB elements $R_{v_{D_i}}$ and $R_{v_{D_j}}$. For example, three solutions for $Q$, which is presented in Fig. 3.2, is $\{(0, 29, 1), (19, 28, 2), (3, 4, 4), (20, 21, 3), (24, 25, 5)\}$, $\{(0, 29, 1), (19, 28, 2), (9, 10, 4), (20, 21, 3), (24, 25, 5)\}$, $\{(8, 17, 3), (11, 16, 4), (9, 10, 4), (12, 13, 5), (14, 15, 5)\}$.

Many of the structural-index and structural-join techniques process path patterns and not twig patterns. Definition 3.1 details how to translate the twig pattern $Q$ into path pattern $Q^{\text{paths}}$. $Q^{\text{paths}}$ expresses all the strings $\{w\} = L(T, v)$ where $Q$ has a match in tree $T$.

Definition 3.1 Given a twig pattern $(V^{T_{\text{twig}}}, E^{T_{\text{twig}}}, \text{label}^{T_{\text{twig}}}, \text{type}^{T_{\text{twig}}})$ we construct for each node $v \in V^{T_{\text{twig}}}$ path pattern $Q_v$. For path $v_1, \ldots, v_n$ from $v_1 = \text{root}(T_{\text{twig}})$ to $v_n = v$ we concatenate $Q_v = e_{v_1} \ldots e_{v_n}$ where

$$e_{v_i} = \begin{cases} \text{label}^{T_{\text{twig}}}(v_i), \text{type}^{T_{\text{twig}}}(v_i) = P - C \\ \Sigma^*\text{label}^{T_{\text{twig}}}(v_i), \text{type}^{T_{\text{twig}}}(v_i) = A - D. \end{cases}$$

The twig paths pattern is the union of all the path patterns $Q_v$ of nodes $v \in V^{T_{\text{twig}}}$ using an or predicate $'Q_{v_1} \land \ldots \land Q_{v_n}'$, where $V^{T_{\text{twig}}} = \{v_1, \ldots, v_n\}$. Given twig pattern $Q$ we denote twig paths pattern by $Q^{\text{paths}}$. □

Example 3.2 The path pattern, which is constructed from the twig pattern ‘/a[/l/c]/b[/l/e]/d’ that is detailed in Fig. 3.5 is $Q^{\text{paths}} = \Sigma^a|\Sigma^c|\Sigma^e|\Sigma^a|\Sigma^b|\Sigma^c|\Sigma^e$. A shorter way to write the same equation is $Q^{\text{paths}} = \Sigma^a(\Sigma^c(b(d(\Sigma^e)))).$ □
3.2 Tree Automata and XML

Different types of XML languages for different types of tasks exist and several competing languages exist for most tasks. Four XML tasks, which involve tree automata (TA) processing, are defined in [141]: 1. Validation of an XML document against a schema; 2. Navigation in an XML document according to a pattern. From a DB point of view, navigation languages express unary queries; 3. Querying an XML document by a pattern. There is a big difference between navigation and querying. Navigation does not produce an output but it is used as a sub-task for other processing types; 4. Transformation of an XML document to and from another format.

Foundations for general XML querying task, compared with other XML processing tasks, are far less developed ([141]). In this thesis, we lay theoretical foundations for XML querying by queries that are modeled as twig patterns. The proposed theory adopts an existing navigation automata [99, 42] to query XML data that is stored in a relational DB. The DB stores the XML as a collection of nodes and not as a complete tree.

In this thesis we adopt the navigation methodology, which is called Selecting Tree Automaton operation on a tree. This operation is denoted by STA(T). STA is a data structure, which is a pair of UUTA A and a subset of states from the set of states of the automaton A. STA(T) is also viewed as a query on the tree T.

The STA(T) operation in [42] is called two ways Selecting Tree Automata. The two ways Selecting Tree Automata operation is given by a ATwig TA and by a BTwig FSA. This operation first applies the TA ATwig bottom-up and then applies the FSA BTwig top-down on the annotations which ATwig made. In this methodology, the unranked tree is first transformed into a binary tree T. Next, FSA BTwig and TA ATwig are constructed from the twig pattern and are applied to the binary tree T.

We define the STA(T) operation for UUTA. The UUTA operation is simple in several aspects. First, although UUTA operates on unranked trees, its transitions are similar to ranked TA transitions. The UUTA transitions are not extended with RE because they process the children states as sets and not as sequences. Further more, the UUTA STA(T) operation requires no tree transformations as in [42]. The tree transformation helps in converting the ordered tree processing to an unordered trees processing. But the tree processing is already unordered in UUTA. Therefore, a tree transformation is not needed. Another source for simplicity is the fact that FSA BTwig is not constructed.
from the twig pattern but induced from \( UUTA \ A^{Twig} \). In the thesis, we denote the induced FSA by \( A^{Twig}_{pc} \) in order to relate it to \( A^{Twig} \). In order to process XML queries in DB, the two-ways \( STA(T) \) query is extended to query with an automaton \( A \) denoted \( STA(A^{Tree}) \). In \( STA(A^{Tree}) \), the UUTA, which is part of the STA, navigates on another UUTA states of \( A \) rather than on a tree nodes \( T \).

Three main navigation tree automata categories, which were defined in [141], are parallel tree automata, sequential tree automata and a document automata. Sequential tree automaton uses a single head that moves in many directions (top-down, bottom-up, etc.). The basic sequential tree automaton model is tree-walking automaton (TWA) [110]. Parallel automaton operates in one direction only (top-down or bottom-up) while using many heads. The UUTA is a parallel automaton. It moves one way (bottom-up) but derives parallel states in each transition. Document automaton is a string automaton which reads documents as a string.

The class of regular node-selecting queries, which is similar to the class of regular tree languages, was defined in [141]. Most navigation tree automata models equivalently capture this class of queries. The first equivalent automata model was obtained by selecting tree automata [99], which is a nondeterministic bottom-up tree automata that is equipped with selecting states. A node is selected if the automaton visits it during an accepting run in a selecting state. An equivalent deterministic automaton model for unary queries was proposed in [63]. Deterministic bottom-up automata are too narrow to express all regular node-selecting queries. Therefore, following [12, 56], two-way automata with the ability to move up and down in the tree along cuts is presented in this thesis.

There are many other equivalent models (e.g., [68, 62, 18, 98]) which define the same class of unary queries. Node-selecting queries are described in [80] by a stepwise tree automata using a particular encoding by binary trees of unranked trees. Yet another way of specifying node-selecting queries, which are based on regular hedge expressions, is studied in [102]. A special type of document automata is used in [18] to evaluate node-selecting queries. A similar type of processing was proposed in [42], which stores XML data in a secondary storage. How regular node-selecting queries can be evaluated in two streaming passes of a document automata is shown in [42].

The \( STA(T) \), which is suggested in this thesis, is based on [42, 99]. It involves two automata passes on the document: a parallel bottom-up pass of a UUTA and a top-down
pass of a document automaton.
Chapter 4

TA Methodology

4.1 Bottom-up Unordered Unranked Trees Automata (UUTA)

Definition 4.1 A bottom-up Unordered Unranked Tree Automaton, denoted by UUTA, processes unordered unranked labeled trees. UUTA is a tuple \( A \triangleq (Q^A, \Sigma, F^A, \delta^A) \) where \( Q^A \) is a finite set of states, \( \Sigma \) is an alphabet which is a finite set of labeling symbols, \( F^A \subseteq Q^A \) is the set of accepting states and \( \delta^A : 2^{Q^A} \times \Sigma \rightarrow Q^A \) is the transition function.

Definition 4.2 An application (run) of a UUTA \( A \) to a tree \( T \) is defined by \( A(T) \triangleq \rho^A(T) \). The output of the run \( \rho^A(T) \) is the map \( \rho^A(T) : V^T \rightarrow 2^{Q^A} \) where each node \( v \in V^T \) is constructed during the run as follows: If \( v \) is a leaf then \( \rho^A(T)(v) = \bigcup S \delta^A(\emptyset, \text{label}^T(v)) \) where \( S \subseteq \bigcup_{v_c \in \text{children}(T,v)} \rho^A(T)(v_c) \) and every \( v_c \in \text{children}(T,v) \) satisfies \( S \cap \rho^A(T)(v_c) \neq \emptyset \). The run is called accepting if \( \rho^A(T)(\text{root}(T)) \cap F^A \neq \emptyset \). The automaton \( A \) accepts \( T \) if there is an accepting run for \( A \) on \( T \). We say that a node \( v \) is annotated by state \( q \) if \( q \in \rho^A(T)(v) \).

Example 4.1 We construct the UUTA \( A \triangleq (Q^A, \Sigma, F^A, \delta^A) \) where the alphabet is \( \Sigma = \{a, b, c, d\} \), the states of \( A \) are \( Q^A = \{q_a, q_b, q_c, q_{d_1}, q_{d_2}\} \), the accepting state is \( F^A = \{q_a\} \) and the transition function includes the following transitions: \( \delta^A(\emptyset, d) = q_{d_1}, \delta^A(\emptyset, b) = q_b, \delta^A(\emptyset, d) = q_{d_2}, \delta^A(q_{d_2}, c) = q_c, \delta^A(q_b, q_c), a = q_b \). This UUTA \( A \) defines a collection of unordered trees. A tree in this collection has three levels. All the leafs nodes are in the lower level and are labeled ‘d’. The middle level contains at least one node-labeled ‘b’ and one node-labeled ‘c’. The root is labeled ‘a’. An example of run \( A(T) \) is given in Fig. 4.1. \( A \) accepts the tree \( T \) in the figure because \( \rho^A(T)(\text{root}(T)) \cap F^A = \{q_a\} \neq \emptyset \).

Part II: Querying of XML in a DB
Figure 4.1: An example of UUTA \( A \) run, which is defined in example 4.1, on a tree \( T \) that is accepted by \( A \). A tree node is denoted by a circle. A node \( v \) has a label in the format ‘\( \text{label}^T(v); q_1, \ldots, q_n \)’ where \( \rho^A(T)(v) = \{q_1, \ldots, q_n\} \).

### 4.2 Construction of UUTA from a twig pattern

A twig pattern \( T^{T_{\text{wig}}} \) is defined by a node-labeled tree \((V^{T_{\text{wig}}}, E^{T_{\text{wig}}}, \text{label}^{T_{\text{wig}}}, \text{type}^{T_{\text{wig}}})\) where with an additional \( \text{type}^{T_{\text{wig}}} \) function that maps each edge to its nodes-relation type (P-C or A-D). A twig-UUTA \( A^{T_{\text{wig}}} \) is a tuple \((Q^{A^{T_{\text{wig}}}}, \Sigma, F^{A^{T_{\text{wig}}}}, \delta^{A^{T_{\text{wig}}}})\). Each node \( v \in V^{T_{\text{wig}}} \) is mapped into at most two states \( q^u, q^v \in Q^{A^{T_{\text{wig}}}} \). The root of the twig pattern has no label. We denote it by the special label \( \top \). The final state is the root state \( F^{A^{T_{\text{wig}}}} = \{q^\top\} \).

A \( A^{T_{\text{wig}}} \) construction begins by iterating over the edges \((v_p, v_c)\) that have A-D relations. For each \((v_p, v_c)\) and a label \( \sigma \in \Sigma \), two additional transitions are constructed: 1. \( \text{up1 transition} \delta^{A^{T_{\text{wig}}}}(\{q^u\}, \sigma) = q^u \). 2. \( \text{up2 transition} \delta^{A^{T_{\text{wig}}}}(\{q^v\}, \sigma) = q^v \). These transitions ensure that for a tree \( T \) and a node \( v \in V^T \), where \( q^v \in \rho^{A^{T_{\text{wig}}}}(T)(v) \), each ancestor \( u \) of a node \( v \) satisfies \( q^u \in \rho^{A^{T_{\text{wig}}}}(T)(u) \).

The algorithm also constructs a \textit{parent transition} for each node \( v_p \in V^{T_{\text{wig}}} \). The construction determines the subset \( S_{\text{decedents}} \) of children(\( T^{T_{\text{wig}}}, v_p \)) that are connected to \( v_p \) by an A-D relation. Each subset \( S'_{\text{decedents}} \subseteq S_{\text{decedents}} \) contributes a transition \( \delta^{A^{T_{\text{wig}}}}(S, \text{label}^{T_{\text{wig}}}(v_p)) = q_{v_p} \) where each child \( v_c \in \text{children}(T^{T_{\text{wig}}}, v_p) \) contributes a state

\[
\begin{align*}
q^u, & \quad v_c \in S'_{\text{decedents}} \\
q^v, & \quad \text{otherwise}
\end{align*}
\]

and the descendants states \( q^v \) annotate the children nodes in \( T \), then, the parent state \( q_{v_p} \) annotates the parent node in \( T \). We construct a transition for each subset because a child is also a descendant. Algorithm 4.1 describes a twig-UUTA construction.
Algorithm 1: The twig-UUTA construction algorithm.

Construct

twig-UUTA\((T_{\text{Twig}}) \triangleq (V_{\text{Twig}}, E_{\text{Twig}}, \text{label}_{\text{Twig}} : V_{\text{Twig}} \mapsto \Sigma,\)

type_{\text{Twig}} : E_{\text{Twig}} \mapsto \{P-C,A-D\})\)

Output: \(T_{\text{Twig}} \triangleq (Q_{T_{\text{Twig}}}, \Sigma \cup \{\top\}, F_{T_{\text{Twig}}}, \delta_{T_{\text{Twig}}})\)

begin
\[
\begin{align*}
Q_{T_{\text{Twig}}} & \leftarrow \{q_\top\} \cup \bigcup_{v \in V_{T_{\text{Twig}}}} \{\delta v\}; \\
F_{T_{\text{Twig}}} & \leftarrow \{q_\top\}; \\
\forall (v_p, v_c) \in E_{T_{\text{Twig}}}, & \text{ where } type_{T_{\text{Twig}}}((v_p, v_c)) = A-D \text{ do} \\
\forall label \sigma \in \Sigma \text{ do} & \text{ Add state } q_{v_c}^\sigma \text{ to } Q_{T_{\text{Twig}}}; \\
\forall \delta v \in V_{T_{\text{Twig}}} \text{ do} & \text{ Add } \text{up}1 \text{ transition } \delta v = \{q_{v_c}^\sigma\}, \sigma = q_{v_c}^\sigma; \\
\forall \text{decendants} & \text{ of } v_p \text{ do} \\
& \text{ Add } \text{up}2 \text{ transition } q_{v_c}^\sigma; \\
\forall \text{decendants} \subseteq S & \text{ of } v_p \text{ do} \\
& \text{ Add parent transition } q_{v_c}^\sigma; \\
& \text{ if } \text{child } v_c \text{ contributes state } \begin{cases} 
q_{v_c}^u \text{, } u \in S' \text{decendants} \\
q_{v_c} \text{, otherwise}
\end{cases} \text{ to } S;
\end{align*}
\]
end

Example 4.2 Figure 2.5 demonstrates the twig pattern ‘/a//c/b//e]/d’. The twig-UUTA \(T_{\text{Twig}}\), which is constructed from the twig pattern in Fig. 2.5, is \(T_{\text{Twig}} = (Q_{T_{\text{Twig}}}, \Sigma, F_{T_{\text{Twig}}}, \delta_{T_{\text{Twig}}})\) where \(\Sigma = \{a, b, c, d, e, \top\}\), \(Q_{T_{\text{Twig}}} = \{q_a, q_b, q_c, q_d, q_e, q_\top\}\), and \(F_{T_{\text{ Twig}}} = \{q_\top\}\). In order to achieve a compact description of \(\delta\), we denote, hereinafter, the set of transitions \(\delta(S, a) = q\), where \(a \in S\) by \(\delta(S) = q\). For example, \(\delta(S) = q\) denotes the set of transitions \(\delta(S, a) = q\) where \(a \in S\). The constructed transitions are \(\delta_{T_{\text{Twig}}}((\emptyset, d) = q_d\) (parent), \(\delta_{T_{\text{Twig}}}((d, e) = q_c\) (parent), \(\delta_{T_{\text{Twig}}}((q_c), \Sigma) = q_e\) (up1), \(\delta_{T_{\text{Twig}}}((q_e), \Sigma) = q_{e}^u\) (up2), \(\delta_{T_{\text{Twig}}}((q_{e}^u), b) = q_b\) (parent), \(\delta_{T_{\text{Twig}}}((q_b, q_{e}^u), b) = q_b\) (parent), \(\delta_{T_{\text{Twig}}}((\emptyset, c) = q_c\) (parent), \(\delta_{T_{\text{Twig}}}((q_e), \Sigma) = q_{e}^u\) (up1), \(\delta_{T_{\text{Twig}}}((q_b, q_c), a) = q_a\) (parent), \(\delta_{T_{\text{Twig}}}((q_b, q_{e}^u), a) = q_a\) (parent), \(\delta_{T_{\text{Twig}}}((q_a), \Sigma) = q_{a}^u\) (up1), \(\delta_{T_{\text{Twig}}}((q_{a}^u), \Sigma) = q_{a}^u\) (up2), \(\delta_{T_{\text{Twig}}}((q_{a}^u), \top) = q_\top\) (parent), \(\delta_{T_{\text{Twig}}}((q_a), \top) = q_\top\) (parent). Figure 4.2 describes the run of this \(T_{\text{Twig}}\) on tree \(T\) that is accepted by \(T_{\text{Twig}}\). The algorithm adds a virtual root node that has the label \(\top\). The virtual root is added to tree \(T\) whenever a run.
of twig-UUTA $A^{Twig}$ on $T$ takes place.

![Diagram of twig-UUTA](image)

Figure 4.2: Illustration of the run of $A^{Twig}$, which is constructed in example 4.2, on a tree. A tree node $v$ is denoted by a circle. It has a two lines label. The first line has the format ‘v; label$^T$$(v)$’. The second line has the format ‘q’ where $\rho^{A^{Twig}}(T)(v) = \{q\}$.

### 4.2.1 Extension of twig-UUTA to derive subtrees

$A^{Twig}$, which is constructed from a twig pattern, accepts a tree $T$ if the labels of the leafs in $T$ are the labels of the leafs in the twig pattern and each branching node in $T$ matches a branching node in the twig pattern. Figure 4.2 is an example of $T$ that has the twig pattern form of ‘//a//c//b//e//d’.

To match a twig pattern we have to recognize $T'$ that has one or more subtrees $T$ where $\text{root}(T) = \text{root}(T')$ and $T$ has a twig pattern form. In this section, we modify
the \( A^{T_{\text{wig}}} \) to recognize \( T' \).

\( A^{T_{\text{wig}}} \) modification begins by adding transitions to \( \delta^{A^{T_{\text{wig}}}} \) that annotate each node in a tree by the default state \( q_\bot \). This construction adds for each label \( \sigma \in \Sigma \) two default state transitions: default1 transition is \( \delta^{A^{T_{\text{wig}}}}(\emptyset, \sigma) = q_\bot \) and default2 transition is \( \delta^{A^{T_{\text{wig}}}}(\{q_\bot\}, \sigma) = q_\bot \). Lemma 4.1 formalizes the relation between the default transitions and the annotations in \( \rho^{A(T)} \).

**Lemma 4.1** Given a UUTA \( A \) with default state transitions and a tree \( T \) then every node \( v \) in \( T \) satisfies \( q_\bot \in \rho^{A(T)}(v) \).

**Proof** The proof is by induction on the height of the annotated nodes. A node with height 1 is a leaf. The default1 transition applies to this leaf and annotates it by \( q_\bot \). The induction step assumes that a node with height \( \leq k \) is annotated by \( q_\bot \) and checks a node \( v \) with height \( k + 1 \). A node \( v_c \in \text{children}(T, v) \) is annotated by \( q_\bot \) because \( v_c \) height is \( \leq k \). Therefore, the default2 transition is used to annotate \( v \) by \( q_\bot \).

Algorithm 2 shows that \( A^{T_{\text{wig}}} \) is modified by duplicating the original transitions function of \( A^{T_{\text{wig}}} \). For each original transition, \( \delta^{A^{T_{\text{wig}}}}(S, \sigma) = q \). The algorithm constructs an additional transition \( \delta^{A^{T_{\text{wig}}}}(S \cup \{q_\bot\}, \sigma) = q \) that adds \( q_\bot \) to the original transition children states. A new transition \( \delta^{A^{T_{\text{wig}}}}(S \cup \{q_\bot\}, \sigma) = q \) annotates a parent node \( v \) where the children states \( S \) of the original transition annotate only a subset of \( \text{children}(T, v) \).

**Theorem 4.1** Given a UUTA \( A \) with default state transitions and a tree \( T \). For every transition, \( \delta^A(S, \sigma) = q \), where \( S \neq \emptyset \), and node \( v \in V^T \). If \( \delta^A(S, \sigma) = q \) applies to the node \( v \) then \( \delta^A(S \cup \{q_\bot\}, \sigma) = q \) applies to the node \( v \).

**Proof** If \( \delta^A(S, \sigma) = q \) applies to a node \( v \in V^T \) then \( v \) has at least one child \( v_c \) because \( S \neq \emptyset \). From Lemma 4.1 we get that \( q_\bot \in \rho^{A(T)}(v_c) \). Therefore, \( \delta^A(S \cup \{q_\bot\}, \sigma) = q \) applies to \( v \) as well.

We can refine the modification by Theorem 4.1. We duplicate only the original transition \( \delta^{A^{T_{\text{wig}}}}(\emptyset, \sigma) = q \) with an additional transition \( \delta^{A^{T_{\text{wig}}}}(\{q_\bot\}, \sigma) = q \). The transition \( \delta^{A^{T_{\text{wig}}}}(S, \sigma) = q \), where \( S \neq \emptyset \), is replaced by the transition \( \delta^{A^{T_{\text{wig}}}}(S \cup \{q_\bot\}, \sigma) = q \). Theorem 4.1 guarantees that the transition \( \delta^{A^{T_{\text{wig}}}}(S \cup \{q_\bot\}, \sigma) = q \) is used whenever the original transition \( \delta^{A^{T_{\text{wig}}}}(S, \sigma) = q \) is used. Therefore, the original transition is not needed. Algorithm 2 describes the modification of \( A^{T_{\text{wig}}} \).

Part II: Querying of XML in a DB
Algorithm 2: The Modify-UUTA algorithm.

\[
\text{Modify-UUTA} (A) \xrightarrow{\Delta} (Q^A, \Sigma, F^A, \delta^A)
\]

\begin{algorithm}
\begin{algorithmic}
\State \textbf{begin}
\ForAll {label } \sigma \in \Sigma \do
\State Add default1 transition $\delta^A(\emptyset, \sigma) = q_\bot$;
\State Add default2 transition $\delta^A(q_\bot, \sigma) = q_\bot$;
\ForAll {\delta^A(S, \sigma) = q} \do
\State Add transition $\delta^A(S \cup \{q_\bot\}, \sigma) = q$;
\If {\!$S \neq \emptyset$ \!} \then
\State Remove original transition $\delta^A(S, \sigma) = q$;
\EndIf
\EndFor
\EndFor
\State \textbf{end}
\end{algorithmic}
\end{algorithm}

Theorem 4.2 describes the collection of trees that is accepted by $A$ after the modification in Algorithm 2 took place.

**Theorem 4.2** Given a UUTA $A$ and a tree $T'$. $A'$ is the modified UUTA $A$ by Algorithm 2. If $A$ accepts a subtree $T$ of $T'$ where root($T$) = root($T'$), then $A'$ accepts $T'$.

**Proof** The proof is by induction on the height of the annotated nodes in $T$. The basis of the induction checks a leaf node $v$ of $T$ with height 1. If the transition $\delta^A(\emptyset, \sigma) = q$ is applied to $v$ in $T$ and $q \in \rho^A(T)(v)$ then $q \in \rho^A(T')(v)$ as well. If $v$ is a leaf node in $T'$ then the same transition $\delta^A'(\emptyset, \sigma) = q$ is used. Otherwise, $v$ is an internal node in $T'$ and transition $\delta^A'\{q_\bot\}, \sigma) = q$ is used. The induction step assumes that the claim is true for an internal node of $T$ with height $\leq k$ and checks a node $v$ of $T$ with height $k + 1$. If the transition $\delta^A(S, \sigma) = q$ is used, $v$ in $T$ and $q \in \rho^A(T)(v)$ then $q_c \in \rho^A(T)(v_c)$ where $v_c \in \text{children}(T, v)$ and $q_c \in S$. From the induction assumption, we have that $q_c \in \rho^A(T')(v_c)$ where $v_c \in \text{children}(T', v)$. From Lemma 4.1 we have that $q_\bot \in \rho^A(T')(v_c)$ for every $v_c \in \text{children}(T', v)$. Therefore, the transition $\delta^A'(S \cup \{q_\bot\}, \sigma) = q$ is applied to $v$ in $T'$ and $q \in \rho^A(T')(v)$.

The run of automaton, which is defined in Definition 4.2 is simplified when $A$ is modified by Algorithm 2. If $v$ is an internal node then the run applied to the transitions $\delta^A(S \cup \{q_\bot\}, label^T(v))$ where $S \subseteq \bigcup_{v_c \in \text{children}(T, v)} \rho^A(T)(v_c)$. Every $v_c \in \text{children}(T, v)$ satisfies $(S \cup \{q_\bot\}) \cap \rho^A(T)(v_c) \neq \emptyset$ because both sets $S \cup \{q_\bot\}$ and $\rho^A(T)(v_c)$ contain the default state $q_\bot$ as we get from Algorithm 2 construction and from Lemma 4.1.

**Example 4.3** The twig-UUTA $A^{Twig}$, which is constructed from the twig pattern ‘/a[/l[c] /l[e][d]’ in example 4.2 after being modified by Algorithm 2 is $A^{Twig} = (Q^{Twig}, \Sigma, F^{Twig}, \delta^{Twig})$ where $\Sigma = \{a, b, c, d, e, \top\}$, $Q^{Twig} = \{q_\bot, q_a, q_b, q_c, q_d, q_e, q_\bot, q_{+}\}$, $F^{Twig} = \{q_{+}\}$. The transitions are: $\delta^{Twig}(\emptyset, \Sigma) = q_\bot$ (default), $\delta^{Twig}(\{q_\bot\}, \Sigma)$
Lemma 4.2 and 4.3 describe the behavior of the $\nu_v$ annotations. Theorem 4.3 uses these theorems to describe the behavior of $\nu_v$ annotations and to prove that annotating a tree node $v$ with $\nu_v$ is the same as matching a subtree, which is rooted in $v$, by the twig sub-pattern that is rooted in $v_p$. From Theorem 4.3 we get that if $\nu_T$ annotates the root($T$) in a run of $A_{Twig}$ on $T$, which is constructed from a twig pattern $Q$, then there is at least one subtree in $T$ that matches the twig pattern $Q$. 

Figure 4.3: Illustration of the run of $A_{Twig}$ automaton, which was constructed in example 4.3 on a tree $T$. A node $v$ in $T$ is denoted by a circle. It has a two lines label. The first line is in the format ‘$v$; label$^T(v)$’. The second line is in the format ‘$q_1, \ldots, q_n$’ where $\rho^{A_{Twig}}(v) = \{q_1, \ldots, q_n\}$. 

Part II: Querying of XML in a DB
Lemma 4.2 Given a twig-UUTA $A^{Twig}$ which is constructed from a twig pattern $T^{Twig}$. If a node $v \in V^T$ satisfies $q_{vd} \in \rho^{A^{Twig}(T)}(v)$, where $\text{type}^{A^{Twig}}(\text{parent}(T^{Twig}, v_d), v_d) = A-D$, then every node $u \in V^T$, which is an ancestor of $v$, satisfies $q_{ud}^u \in \rho^{A^{Twig}(T)}(u)$. □

PROOF The proof is by induction on the distance between $u$ and $v$. The node $u$, which is of distance 1 from node $v$, is a parent of $v$. From Lemma 4.1, we get that $q_{vd} \in \rho^{A^{Twig}(T)}(u_c)$ where $u_c \in \text{children}(T, u)$. We also know that $v \in \text{children}(T, u)$ and $q_{vd} \in \rho^{A^{Twig}(T)}(v)$. Therefore, we can apply the transition $u_1$ that annotates $u$ by $q_{vd}$. The induction step assumes that an ancestor node, which is of distance $\leq k$ from $v$, is annotated by $q_{vd}$ and checks an ancestor node $u$ that is of distance $k + 1$. From Lemma 4.1 we have that $q_{vd} \in \rho^{A^{Twig}(T)}(u_c)$ where $u_c \in \text{children}(T, u)$. We know that there is a node $u_c \in \text{children}(T, u)$ which is an ancestor of $v$. From the induction assumption we get that $q_{vd}^u \in \rho^{A^{Twig}(T)}(u_c)$. Therefore, we can apply the transition $u_2$ that annotates $u$ by $q_{vd}$.

Lemma 4.3 Given a twig-UUTA $A^{Twig}$ which is constructed from a twig pattern $T^{Twig}$. If a node $u \in V^T$ satisfies $q_{ud}^u \in \rho^{A^{Twig}(T)}(u)$ then there is a node $v \in V^T$, which is a descendant of $u$ that satisfies $q_{vd} \in \rho^{A^{Twig}(T)}(v)$. □

PROOF By induction on the height of $u$. A node $u$ with height 1 is a leaf. A leaf is not annotated by $q_{vd}^u$. Therefore, the basis for the induction is a node $u$ with height 2. In this case, a node $u$ is a parent of a node $v$ where $q_{vd} \in \rho^{A^{Twig}(T)}(v)$. Therefore, an $u_1$ transition operates on a node $u$. Therefore, a node $u$ is a parent of a leaf $v$ where $q_{vd}^u \in \rho^{A^{Twig}(T)}(u)$. The induction step assumes that if a node, which has height $\leq k$, is annotated by $q_{vd}^u$, then it is an ancestor of a node $v$ where $q_{vd} \in \rho^{A^{Twig}(T)}(v)$ and checks node $u$ that has height $k + 1$. The transition that annotates $u$ with $q_{vd}^u$ is either $u_1$ transition or $u_2$ transition. The $u_1$ transition is applied when $v_c \in \text{children}(T, u)$ is annotated by $q_{vd}$. Therefore, the Lemma’s claim is satisfied directly. When the $u_2$ transition is applied, $v_c \in \text{children}(T, u)$ is annotated by $q_{vd}$, therefore, the Lemma’s claim is satisfied by the induction step assumption.

Theorem 4.3 Given a twig-UUTA $A^{Twig}$ that is constructed from a twig pattern $T^{Twig}$. A node $v \in V^T$ satisfies $q_{vp} \in \rho^{A^{Twig}(T)}(v)$ if and only if a subtree, which is rooted in $v$, matches the twig sub-pattern that is rooted in $v_p \in V^{ Twig}$. □

PROOF By induction on the height of $v_p$ in $T^{Twig}$. The basis for the induction assumptions assumes that a node $v$ in the tree $T$ has a label $\sigma$, where $\sigma = \text{label}^{T^{Twig}}(v_p)$ for a leaf $v_p \in V^{ Twig}$, if and only if $v$ is annotated by $q_{vp}$. The proof splits the tree nodes $v$ into two types: leaf nodes and internal nodes. A leaf node is annotated by the application of $\delta^{A^{Twig}}(\emptyset, \sigma) = q_{vp}$. An internal node is annotated by the application of $\delta^{A^{Twig}}(\{q_{vd}\}, \sigma) = q_{vp}$. In both cases, the Lemma’s claim is satisfied.

The induction step assumes that the Lemma’s claim is true for twig pattern nodes with height $\leq k$ and proves it for twig pattern nodes with height $k + 1$. The first direction
assumes that a subtree of $T$, which is rooted in node $v$, matches a twig sub-pattern that is rooted in a twig node $v_p$ with height $k + 1$. We examine $v_c \in \text{children}(T_{\text{Twig}}, v_p)$. If $\text{type}(v_p, v_c) = P-C$ then there is a node $u \in \text{children}(T, v)$ that matches the twig sub-pattern that is rooted in $v_c$. The node $v_c$ has height $k$ in $T_{\text{Twig}}$. Therefore, from the induction step we get that $u$ is annotated by $q_{v_c}$. If $\text{type}(v_p, v_c) = A-D$ then $v$ has a descendant $u$ where a subtree, which is rooted in $u$, matches the twig sub-pattern that is rooted in $v_c$. The node $v_c$ has height $< k$. Therefore, from the induction step we get that $u$ is annotated by $q_{v_c}$. If $u \notin \text{children}(T, v)$ then there is a node $w \in \text{children}(T, v)$ that is an ancestor of $u$. From Lemma 4.2 we get that the node $w$ is annotated by $q_{v_c}$. Due to the construction of the parent transitions in Algorithm 1, we are able to apply a parent transition, which annotates the node $v$ with the state $q_{v_p}$ from the $\text{children}(T, v)$ nodes annotations.

The opposite direction assumes that a subtree, which is rooted in a node $v$, is annotated by $q_{v_p}$ where $v_p$ is a twig pattern node with height $k + 1$. State $q_{v_p}$ is annotated by a transition $\delta^{A_{\text{Twig}}}(S, \sigma) = q_{v_p}$ where every $v_c \in \text{children}(T_{\text{Twig}}, v_p)$ contributes a state $\left\{ \begin{array}{ll} q_{v_c}, & u \in S' \text{decdents} \\ q_{v_c}, & \text{otherwise} \end{array} \right.$ to $S$. We examine $u \in \text{children}(T, v)$ that is annotated by the states in $S$. If $u$ is annotated by $q_{v_c}$, then, from the induction assumption we get that $u$ matches a twig sub-pattern that is rooted in $v_c$. If $u$ is annotated by $q_{v_c}$, then, from Lemma 4.4 we get that there is a node $w$, which is a descendant of $u$, that is annotated by $q_{v_c}$. From the induction step assumption we get that a subtree, which is rooted in the node $w$, matches a twig sub-pattern that is rooted in $v_c$. The node $w$ is also a descendant of $v$. Since subtrees, which are rooted in nodes $u$ and $w$, match twig sub-patterns that are rooted in nodes $v_c$ and have the requested A-C and P-C node relations with $v$, then, there is a subtree, which is rooted in $v$, that matches the twig sub-pattern that is rooted in node $v_p$.

4.3 Application of a Selecting Tree Automaton (STA) on a tree $T$

UUTA $A$ can only decide whether to accept or not to accept the tree $T$. In order to extract nodes that match a twig pattern, we have to enhance the tree automata by an additional mechanism to select nodes in the tree $T$. The application of the STA query on a tree $T$ is defined in Definition 4.3.

**Definition 4.3** Selecting Tree Automaton (STA) is a pair $(A^{\text{Selecting}}, S)$ where $A^{\text{Selecting}}$ is a selecting-UUTA and $S \subseteq Q^{A^{\text{Selecting}}}$ is a set of selecting states. STA defines a query. The application of the STA $(A^{\text{Selecting}}, S)$ on tree $T$ is $\Sigma T \overset{A^{\text{Selecting}}}{\rightarrow} (A^{\text{Selecting}}, S) \cdot T \rightarrow \rho^{\Sigma T \overset{A^{\text{Selecting}}}{\rightarrow}}$. The output of $\Sigma T \overset{A^{\text{Selecting}}}{\rightarrow}$ operation $\rho^{\Sigma T \overset{A^{\text{Selecting}}}{\rightarrow}} : V_T \mapsto 2^S$ maps selected nodes $v \in V_T$ to states in $S$ that derive an accepting run $\rho^{A^{\text{Selecting}}}(T)$. The selected nodes are the query answer.

In this thesis, we describe the application of STA to a twig-UUTA. In all the examples in this chapter, $A^{\text{Selecting}} = A_{\text{Twig}}$. If $A^{\text{Selecting}} = A_{\text{Twig}}$ then the selecting states
\[ S = \bigcup_{v \in \mathcal{T}^{\text{Twig}}} q_v \] annotate nodes that match twig sub-patterns. The \( STA^S_{\mathcal{T}^{\text{Twig}}} (T) \) operation guarantees that the selected nodes match twig sub-patterns that are part of a whole twig pattern which \( A^{\text{Twig}_T} \) accepts. The \( STA^S_{\mathcal{T}^{\text{Twig}}} (T) \) operation is used in both the join operation in Section 5.3 and the holistic index operation in Chapter 6.

Figure 4.4 describes the flow of the \( STA^S_{\mathcal{T}^{\text{Selecting}}} (T) \) algorithm. The \( STA^S_{\mathcal{T}^{\text{Selecting}}} (T) \) algorithm operates in two phases: offline and online. The offline part receives the selecting-UUTA \( A^{\text{Selecting}} \) and the selecting states \( S \) and constructs the FSA \( A^{\text{Selecting}}_{pc} \). The online phase receives as an input the node-labeled tree and the \( A^{\text{Selecting}}_{pc} \). It has two steps. The first step performs a \( A^{\text{Selecting}}(T) \) as defined in Definition 4.2. It computes the \( \rho^{\text{Selecting}}(T) \). However, the states in \( \rho^{\text{Selecting}}(T) \) do not yet represent the selection because there may be states that do not derive the root node. The second step traverses \( T \) top-down with the FSA \( A^{\text{Selecting}}_{pc} \), which is constructed in the offline phase and prunes from \( \rho^{\text{Selecting}}(T) \) the only nodes that are mapped to the selected states and derive the root node. The output from the \( STA^S_{\mathcal{T}^{\text{Selecting}}} (T) \) operation is \( \rho^{STA^S_{\mathcal{T}^{\text{Selecting}}} (T)} \).

![Figure 4.4: The flow of the \( STA^S_{\mathcal{T}^{\text{Selecting}}} (T) \) operation](image)

Section 4.3.1 describes the offline construction of the FSA. The bottom-up traversal step is the run \( \rho^{\text{Selecting}}(T) \) that was described in section 4.1. The top-down traversal step is described in section 4.3.2.
4.3.1 FSA $A^\text{Selecting}_{pc}$ construction

FSA, which is constructed from $A^\text{Selecting}$, is denoted by $A^\text{Selecting}_{pc}$. $A^\text{Selecting}_{pc}$ represents the application of $A^\text{Selecting}$ to a collection of all node-labeled trees that $A^\text{Selecting}$ accepts. $q \in Q^\text{Selecting}$ is a state in $A^\text{Selecting}_{pc}$. A transition $\delta^\text{Selecting}_{pc}((q_p, q_c)) = q_c$ exists if there is a tree $T = (V^T, E^T, \text{label}^T)$, which is accepted by $A^\text{Selecting}$, a node $v_p \in V^T$ where $q_p \in \rho^\text{Selecting}(T)(v_p)$ and another node $v_c \in V^T$ where $q_c \in \rho^\text{Selecting}(T)(v_c)$ and $v_c \in \text{children}(T, v_p)$.

The $A^\text{Selecting}_{pc}$ construction is described in Algorithm 3. In each cycle, it searches for new parent states $q_p \in Q^\text{Selecting}_{pc}$. In each cycle, the FSA states annotate the tree $T$ nodes in $\rho^\text{Selecting}(T)$. The algorithm checks if a transition from these states to $q_p$ can be applied to $T$. If such a transition exists, then $q_p$ is added to $A^\text{Selecting}_{pc}$. If $q_p$ is not found, then, the $A^\text{Selecting}_{pc}$ construction terminates. The start state $q^\text{Selecting}_{pc}0$ is a final state in $A^\text{Selecting}_{pc}$. This construction limits $F^\text{Selecting}$ to have a single state. If $A^\text{Selecting} = A^\text{Twig}$ then $F^\text{Twig} = \{q_T\}$ and this limitation holds.

Algorithm 3: The $A^\text{Selecting}_{pc}$ construction algorithm.

```
Construct-FSA($A^\text{Selecting}$ $\Delta$ = ($Q^\text{Selecting}, \Sigma, F^\text{Selecting}, \delta^\text{Selecting}$), $S \subseteq Q^\text{Selecting}$)
Output: $A^\text{Selecting}_{pc}$ $\Delta$ = ($Q^\text{Selecting}_{pc}, \Sigma, F^\text{Selecting}_{pc}, q^\text{Selecting}_{pc0}, \delta^\text{Selecting}_{pc}$)
Data: $Q^\text{Selecting}_{pc}$ previous is a temporary set of $A^\text{Selecting}_{pc}$ states
begin
    $Q^\text{Selecting}_{pc} \leftarrow \emptyset$;
    repeat
        $Q^\text{Selecting}_{pc}$ previous $\leftarrow Q^\text{Selecting}_{pc}$;
        forall transition $\delta^\text{Selecting}_{pc}((S, a)) = q_p$ where $S \subseteq Q^\text{Selecting}_{pc}$ previous do
            $Q^\text{Selecting}_{pc} \leftarrow Q^\text{Selecting}_{pc}$ previous $\cup \{q_p\}$;
            forall $q_c \in S$ do
                add transition $\delta^\text{Selecting}_{pc}((q_p, q_c)) = q_c$;
        until $Q^\text{Selecting}_{pc}$ previous $\setminus = Q^\text{Selecting}_{pc}$ previous;
        $q^\text{Selecting}_{pc0} \in F^\text{Selecting}_{pc}$;
        $F^\text{Selecting}_{pc} \leftarrow S$;
    end
```

Part III Querying of XML in a DB
Example 4.4: It demonstrates the FSA construction. Example 4.3 defines a UUTA $A^{T Twig}$ that accepts the twig pattern ‘//a[//c]/b[//e]/d’. The FSA $A^{T Twig}_{pc}$, which is constructed by Algorithm 3, is described in Fig. 4.5.

$A^{T Twig}_{pc} = (Q^{A^{T Twig}_{pc}}, \Sigma, F^{A^{T Twig}_{pc}}, q^{A^{T Twig}_{pc}}_0, \delta^{A^{T Twig}_{pc}})$ where $\Sigma^A = \{a, b, c, d, e, \top\}$, $Q^{A^{T Twig}_{pc}} = \{q_{\top}, q_{\perp}, q_a, q^{u}_{a}, q_b, q_c, q^{u}_{c}, q_d, q_e, q^{u}_{e}\}$, $F^{A^{T Twig}_{pc}} = \{q_{a}, q_{b}, q_{c}, q_{d}, q_{e}\}$, $q^{A^{T Twig}_{pc}}_0 = q_{\top}$. The transitions are

$\delta^{A^{T Twig}_{pc}}(q_{\top}, q^{u}_{a}) = q^{u}_{a}$,
$\delta^{A^{T Twig}_{pc}}(q_{a}, q_a) = q_a$, $\delta^{A^{T Twig}_{pc}}(q^{u}_{a}, q_a) = q_a$, $\delta^{A^{T twig}_{pc}}(q_a, q_c) = q_c$,
$\delta^{A^{T Twig}_{pc}}(q_{a}, q^{u}_{c}) = q^{u}_{c}$, $\delta^{A^{T Twig}_{pc}}(q^{u}_{c}, q^{u}_{c}) = q^{u}_{c}$, $\delta^{A^{T Twig}_{pc}}(q^{u}_{c}, q^{u}_{c}) = q^{u}_{c}$, $\delta^{A^{T Twig}_{pc}}(q_a, q_{b}) = q_b$,
$\delta^{A^{T Twig}_{pc}}(q^{u}_{a}, q_{b}) = q_{b}$, $\delta^{A^{T Twig}_{pc}}(q^{u}_{b}, q_{b}) = q_{b}$, $\delta^{A^{T Twig}_{pc}}(q_{b}, q_{d}) = q_d$,
$\delta^{A^{T Twig}_{pc}}(q_{c}, q_{c}) = q_c$, $\delta^{A^{T Twig}_{pc}}(q^{u}_{c}, q_{c}) = q_{c}$, $\delta^{A^{T Twig}_{pc}}(q_{c}, q_{e}) = q_e$,
$\delta^{A^{T Twig}_{pc}}(q^{u}_{c}, q_{e}) = q_{e}$, $\delta^{A^{T Twig}_{pc}}(q_{e}, q_{e}) = q_{e}$, $\delta^{A^{T Twig}_{pc}}(q_{e}, q_{e}) = q_{e}$,
$\delta^{A^{T Twig}_{pc}}(q_{d}, q_{d}) = q_d$, $\delta^{A^{T Twig}_{pc}}(q_{e}, q_{d}) = q_{d}$, $\delta^{A^{T Twig}_{pc}}(q_{d}, q_{e}) = q_{e}$,
$\delta^{A^{T Twig}_{pc}}(q_{e}, q_{d}) = q_{d}$, $\delta^{A^{T Twig}_{pc}}(q_{e}, q_{e}) = q_{e}$, $\delta^{A^{T Twig}_{pc}}(q_{e}, q_{e}) = q_{e}$,
$\delta^{A^{T Twig}_{pc}}(q_{d}, q_{d}) = q_d$, $\delta^{A^{T Twig}_{pc}}(q_{d}, q_{d}) = q_d$, $\delta^{A^{T Twig}_{pc}}(q_{d}, q_{d}) = q_d$. 

Part II: Querying of XML in a DB
Figure 4.5: An illustration of $A_{pc}^{Twig}$, which is constructed by Algorithm 3 from the $A_{pc}^{Twig}$ in example 4.3. A state node $q \in Q_{A_{pc}^{Twig}}$ is denoted by a labeled circle. The label denotes the state Id. A transition $\delta_{A_{pc}^{Twig}}(q_p, q_c) = q_c$ is denoted by an edge from $q_p$ circle to $q_c$ circle. The start state $d_{pc}^{Twig}$ is denoted by an incoming edge. The states in $F_{pc}^{Twig}$ are denoted by double circles.

Definition 4.4 describes the language that $A_{pc}^{Selecting}$ recognizes.

**Definition 4.4** Given a UUTA $A$. The following languages are defined:

$L(A, T, q)$ is defined for a tree $T$. It defines the language of strings $q_{v_1}, \ldots, q_{v_n}$ where:

1. $v_1, \ldots, v_n$ is a path in $T$ from node $v_1$ down to a node $v_n$; 2. $q_{v_i} \in \rho^A(T)(v_i)$, $1 \leq i \leq n$; 3. The state $q_{v_{i+1}}$ derives the state $q_{v_i}$, i.e. $\delta^A(S \cup \{q_{v_{i+1}}\}, \sigma) = q_{v_i}$, $1 \leq i \leq n - 1$; 4. $q_{v_1} = q$. These strings are called **deriving-states strings** of the state $q$.

$L^k(A, q)$ is the union of the languages $L(A, T, q)$ of all trees $T$ with height $k$. 

---

**Part II** Querying of XML in a DB
\[ L^*(A, q) \] is the union of the languages \( L^k(A, q) \) of all trees, i.e., \( L^*(A, q) = \bigcup_{1 \leq k < \infty} L^k(A, q). \]

Lemmas 4.4 and 4.5 prove that for a given UUTA \( A^\text{Selecting} \) and a FSA \( A^\text{pc} \), which is constructed by Algorithm 3 from \( A^\text{Selecting} \), satisfies \( L(A^\text{pc}) = L^*(A^\text{Selecting}, q)^\text{pc} \). These Lemmas examine the run of Algorithm 3. In each repeat-until loop in Algorithm 3 both Lemmas examine the FSA \( A^\text{Selecting} \), where \( Q^\text{Selecting} \) and the appropriate states that were added to \( Q^\text{Selecting} \) until the current iteration, \( F^\text{Selecting} \) are the states that were added to \( Q^\text{Selecting} \) until the current iteration and \( q^\text{Selecting} = q \).

**Lemma 4.4** Given a UUTA A. Algorithm 3 is applied to construct the FSA \( A^\text{Selecting} \). Then, \( L(A^\text{Selecting}) \subseteq L^*(A^\text{Selecting}, q) \) for every \( q \).

**Proof** By induction on Algorithm 3 operation. The first iteration constructs \( A^\text{Selecting} \) that satisfies \( L(A^\text{Selecting}) = L^1(A^\text{Selecting}, q) \). The induction step assumes that the Lemma is true for previous iterations and proves it for the deriving-states strings in the current iteration. The current iteration finds the transition \( \delta^\text{Selecting}(S, a) = q_p \) where \( S \subseteq Q_{\text{previous}}^\text{Selecting} \) and constructs the automaton \( A^\text{Selecting}_{q_p} \) that accepts the states string \( q_p w_n q_p w_{n-1} \ldots q_p w_1 \in L(A^\text{Selecting}_{q_p}) \). We examine string \( q_p w_1 \). From the Lemma’s induction, we have that \( L(A^\text{Selecting}_{q_c}) \subseteq L^*(A^\text{Selecting}, q_c) \) where \( q_c \in S \). We construct the tree \( T \) where \( \text{label}^T(\text{root}(T)) = a \) and each appropriate subtree \( T_{q_c} \) of \( T \), which is rooted in \( v_c \in \text{children}(T, \text{root}(T)) \), satisfies \( L(A^\text{Selecting}, T_{q_c}, q_c) \neq \emptyset \). We make sure that \( w_1 \in L(A^\text{Selecting}, T_{q_c}, q_c) \). From Definition 4.2 we get that \( \delta^\text{Selecting}(S, a) = q_p \) annotates \( \text{root}(T) \). Therefore, the deriving-state string \( q_p w_1 \in L(A^\text{Selecting}, T, q) \subseteq L^*(A^\text{Selecting}, q_p) \).

**Lemma 4.5** Given a UUTA \( A^\text{Selecting} \). Algorithm 3 is applied to construct an FSA \( A^\text{pc} \). Then, in iteration \( i \), \( L(i, q) \subseteq L(A^\text{Selecting}) \) for \( q \in Q_{\text{previous}}^\text{Selecting} \).

**Proof** By induction on Algorithm 3 operation. The first iteration constructs \( A^\text{Selecting} \) that satisfies \( L(A^\text{Selecting}) = L^1(A^\text{Selecting}, q) \). The induction step assumes that \( L^i(A^\text{Selecting}, q) \subseteq L(A^\text{Selecting}) \) in iteration \( i \) and proves that \( L^{i+1}(A^\text{Selecting}, q) \subseteq L(A^\text{Selecting}) \) in iteration \( i + 1 \). The proof examines the tree \( T \) with height \( i + 1 \) where \( \text{root}(T) \) is annotated by the transition \( \delta^\text{Selecting}(S, a) = q \) on the \( \text{root}(T) \). \( T \) is composed from the \( \text{root}(T) \) and the appropriate subtrees \( T_{v_c} \) where \( v_c \in \text{children}(\text{root}(T)) \) and \( v_c = \text{root}(T_{v_c}) \). From Lemma 4.4 and since the height of \( \text{root}(T_{v_c}) \) is at most \( i \), we know that \( L(A^\text{Selecting}, T_{v_c}, q_c) \subseteq L^i(A^\text{Selecting}, q_c) \) where \( q_c \in S \). From the induction step assumption, we get that \( L^i(A^\text{Selecting}, q_c) \subseteq L(A^\text{Selecting}, q_c) \). Therefore, \( L(A^\text{Selecting}, T_{v_c}, q_c) \subseteq L(A^\text{Selecting}, q_c) \). Iteration \( i + 1 \) finds all the existing transitions from \( Q_{\text{previous}}^\text{Selecting} \).
Therefore, it finds $\delta^{A_{Selecting}}(S, a) = q$, adds $q$ to $Q^{A_{pc}}$ and adds the transitions $\delta^{A_{pc}}(q, q_c) = q_c$ where $q_c \in S$, to $\delta^{A_{pc}}$. Therefore, if $qw \in L(A_{Selecting}, T, q)$ then $w \in L(A_{q_{Selecting}})$ and $\delta^{A_{pc}}(q, q_c) = q_c$ derives the symbol $q$. Therefore, $qw \in L(A_{q_{Selecting}})$ and in general $Li+1(A_{Selecting}, q) \subseteq L(A_{Selecting})$.

From Lemma 4.5 we get that after infinite number of iterations $L^*(A_{Selecting}, q) \subseteq L(A_{q_{Selecting}})$. $A_{q_{Selecting}}$ is finite, therefore, after $k$ iterations, all the transitions of $A_{q_{Selecting}}$ are constructed and $Q_{previous} = Q^{A_{Selecting}}$. $A_{q_{Selecting}}$ in iteration $k$ is equal to $A_{q_{Selecting}}$ in iteration $k + 1$ which is equal to $A_{q_{Selecting}}$ in iteration $\infty$. Therefore, in iteration $k$, $L^i(A_{Selecting}, q) \subseteq L(A_{q_{Selecting}})$ for $i > 0$. Therefore, Algorithm 3 produces the FSA $A_{q_{Selecting}}$ where $L^*(A_{Selecting}, q) \subseteq L(A_{q_{Selecting}})$. Lemmas 4.5 and 4.4 prove that $L^*(A_{Selecting}, q) = L(A_{q_{Selecting}})$. Algorithm 3 returns the FSA $A_{pc} = A_{q_{Selecting}}$ where $q = q_0$ and, therefore, $L(A_{Selecting}) = L^*(A_{Selecting}, q_0)$. The FSA construction in Algorithm 3 constructs the start state $q_0 \in F^{A_{Selecting}}$. Therefore, $A_{pc}$ defines the language of all the deriving-states strings from the final state in $A_{Selecting}$. The final state annotates $\text{root}(T)$ and, therefore, $A_{pc}$ defines the language of all the deriving-states strings from $\text{root}(T)$ where $T$ is accepted by $A_{Selecting}$.

### 4.3.2 Top-down traversal by $STA_{S}^{A_{Selecting}}(T)$

The top-down phase is accomplished by the FSA $A_{pc}$ that was constructed in section 4.3.1. The $A_{pc}$ processes the $A_{Selecting}$ states assignments to nodes in $\rho^{A_{Selecting}}(T)$, which were derived in a bottom-up traversal. The top-down traversal prunes the run $\rho^{A_{Selecting}}(T)$ by mapping to states that derived the $F^{A_{pc}}$ state in $\rho^{A_{Selecting}}(\text{root}(T))$. The presented $STA_{S}^{A_{Selecting}}(T)$ algorithm applied $A_{pc}$ to each path in the tree from the root to its leafs. In each node $v$, $A_{pc}$ inputs are the states in $\rho^{A_{Selecting}}(T)(v)$. If $\rho^{STA_{S}^{A_{Selecting}}}(T)(v)$ is mapped to a selecting state then $v$ is a selected node. Algorithm 4 describes the top-down traversal function. The function is recursive. Initially, it is called with $\text{root}(T)$ and $q_0$ as parameters.
Algorithm 4: The STA(T) top-down traversal algorithm.

```
top-down traversal( vₚ ∈ VT, qₚ ∈ QApcSele )
Data: ρASApcSele(T), FSA ApcSele, tree T
Output: ρSTASApcSele(T)

begin
  forall qₑ ∈ ρASApcSele(T)(vₑ) where vₑ ∈ children(T, vₚ) do
    if exists δASApcSele(qₚ, qₑ) = qₑ then
      if qₑ ∈ FASApcSele then
        Add qₑ to ρSTASApcSele(T)(vₑ);
        top-down traversal(vₑ, qₑ);
    end
end
```

Example 4.5 The UUTA ATwig in example 4.4 recognizes the twig pattern ‘//a[///c]/b[//e]/d’. The bottom-up run of ATwig on the tree T is described in Fig. 4.3. The FSA ATwig is constructed from the ATwig in example 4.4 by Algorithm 3. Figure 4.6 illustrates the output ρSTASApcSele(T) from the top-down traversal of the ATwig on the tree T. The selected nodes in this STASApcSele(T) operation are \{1, 4, 6, 7, 8, 9, 10, 11, 12, 15\}. □
Figure 4.6: Illustration of the output from $\rho^{\text{STA}}_{S}^{T_{\text{wig}}}(T)$. Notation: A tree node $v$ is denoted by a circle. It has a two lines label. The first line has the format ‘$v$; label$^T(v)$’. The second line has the format ‘$q_1, \ldots, q_n$’ where $\rho^{\text{STA}}_{S}^{T_{\text{wig}}}(T)(v) = \{q_1, \ldots, q_n\}$.

4.4 Application of STA to a tree automaton $A^{\text{Tree}}$

The application of the STA query to UUTA $A^{\text{Tree}}$ is defined in Definition 4.5.

**Definition 4.5** Selecting Tree Automaton (STA) is a pair $(A^{\text{Selecting}}, S)$ where $A^{\text{Selecting}}$ is a selecting-UUTA and $S \subseteq Q^{A^{\text{Selecting}}}$ is a set of selecting states. STA defines a query. The application of the STA $(A^{\text{Selecting}}, S)$ to UUTA $A^{\text{Tree}}$ is $STA_{S}^{A^{\text{Selecting}}}(A^{\text{Tree}}) \triangleq (A^{\text{Selecting}}, S) \cdot A^{\text{Tree}} \rightarrow \rho^{STA_{S}^{A^{\text{Selecting}}}}(A^{\text{Tree}})$. The output of $STA_{S}^{A^{\text{Selecting}}}(T)$ operation $\rho^{STA_{S}^{A^{\text{Selecting}}}}(A^{\text{Tree}}) : Q^{A^{\text{Tree}}} \rightarrow 2^{S}$ maps the selected states $q \in Q^{A^{\text{Tree}}}$ to states in $S$. The selected states are the query answer. The $STA_{S}^{A^{\text{Selecting}}}(A^{\text{Tree}})$ operation extends the $STA_{S}^{A^{\text{Selecting}}}(T)$ operation and operates on a collection of trees that are accepted by $A^{\text{Tree}}$. A selected $q_{T^{\text{ree}}} \in Q^{A^{\text{Tree}}}$ satisfies $q^{\text{Selecting}} \in \rho^{STA_{S}^{A^{\text{Selecting}}}}(q_{T^{\text{ree}}})$ where $q^{\text{Selecting}} \in S$ if there is a tree $T$, which is accepted by both automata $A^{\text{Tree}}$ and $A^{\text{Selecting}}$. 

Part II Querying of XML in a DB
Figure 4.7 describes the flow of the STA($A_{Tree}$) algorithm. The input to the algorithm is a tree-UUTA $A_{Tree}$ and a STA ($A_{Selecting}$, $S$). The $STA_S(A_{Selecting}, A_{Tree})$ operation intersects $A_{Tree}$ and $A_{Selecting}$. The output from this intersection is the intersected-UUTA $A_{Tree} \cap A_{Selecting}$. The $STA_S(A_{Selecting}, A_{Tree})$ operation constructs the intersected-FSA $(A_{Tree} \cap A_{Selecting})_{pc}$ from the intersected-UUTA. The construction process is the same as the FSA construction in section 4.3.1. The top-down traversal traverses the $(A_{Tree} \cap A_{Selecting})_{pc}$ states. Each state in $(A_{Tree} \cap A_{Selecting})_{pc}$ contains two components: a selecting-UUTA state $q_{Selecting} \in Q_{A_{Selecting}}$ and a tree-UUTA state $q_{Tree} \in Q_{A_{Tree}}$. The top-down traversal selects states from the intersected-FSA that contains the selecting state $q_{Selecting} \in S$. The output maps $q_{Selecting}$ to $\rho^{STA_S(A_{Selecting}, A_{Tree})}(q_{Tree})$.

![Diagram of STA algorithm](image_url)
Section 4.4.1 defines the intersection between UUTAs, section 4.4.2 describes the FSA construction from the intersected-UUTA and section 4.4.3 describes the $ST\.A_S^{Selecting}(ATree)$ top-down traversal.

### 4.4.1 Intersection between UUTAs

**Definition 4.6** The intersection between two UUTAs $A^1 = (Q^{A^1}, \Sigma, F^{A^1}, \delta^{A^1})$ and $A^2 = (Q^{A^2}, \Sigma, F^{A^2}, \delta^{A^2})$ is the automaton $A^1 \cap A^2 = (Q^{A^1 \cap A^2}, \Sigma, F^{A^1 \cap A^2}, \delta^{A^1 \cap A^2})$ where $Q^{A^1 \cap A^2} = Q^{A^1} \times Q^{A^2}$, $\delta^{A^1 \cap A^2} : 2^Q^{A^1} \times 2^Q^{A^2} \times \Sigma \mapsto Q^{A^1} \times Q^{A^2}$, $\delta^{A^1 \cap A^2}((S_1, S_2), a) = \langle \delta^{A^1}(S_1, a), \delta^{A^2}(S_2, a) \rangle$ and $(q_1, q_2) \in F^{A^1 \cap A^2}$ implies that $q_1 \in F^{A^1}$ and $q_2 \in F^{A^2}$.

**Example 4.6** The $AT_{\text{Twig}}$, which is constructed from the twig pattern '/a[b/d]/c'/d', is $(Q^{AT_{\text{Twig}}}, \Sigma, F^{AT_{\text{Twig}}}, \delta^{AT_{\text{Twig}}})$. The alphabet is $\Sigma = \{a, b, c, d, \top\}$. The states of $AT_{\text{Twig}}$ are $Q^{AT_{\text{Twig}}} = \{q_1, q_\top, qa, qb, qc, qd_1, qd_2, qd_3\}$. The accepting states are $F^{AT_{\text{Twig}}} = \{q_1\}$. The transition function includes the following transitions: $\delta^{AT_{\text{Twig}}}((\{q_1\}, \Sigma) = q_1$, $\delta^{AT_{\text{Twig}}}((\emptyset, \Sigma) = q_\top$, $\delta^{AT_{\text{Twig}}}((\emptyset, d\}) = q_{d_1}$, $\delta^{AT_{\text{Twig}}}((\{q_1\}, d\}) = q_{d_2}$, $\delta^{AT_{\text{Twig}}}((\emptyset, c\}) = q_c$, $\delta^{AT_{\text{Twig}}}((\{q_1\}, c\}) = q_2$, $\delta^{AT_{\text{Twig}}}((\{q_1\}, b\}) = q_b$, $\delta^{AT_{\text{Twig}}}((\{q_1\}, a\}) = q_\top$. Figure 4.8 defines an automaton $AT_{\text{Tree}} = (Q^{AT_{\text{Tree}}}, \Sigma, F^{AT_{\text{Tree}}}, \delta^{AT_{\text{Tree}}})$ which accepts a collection of trees. The states of $AT_{\text{Tree}}$ are $Q^{AT_{\text{Tree}}} = \{0, 1, 2, 3\}$. The alphabet is $\Sigma = \{\top, a, b, c, d\}$. The accepting state is $F^{AT_{\text{Tree}}} = \{0\}$. The transitions are $\delta^{AT_{\text{Tree}}}((\emptyset, d) = 3$, $\delta^{AT_{\text{Tree}}}((\{3\}, b) = 2$, $\delta^{AT_{\text{Tree}}}((\{3\}, c) = 2$, $\delta^{AT_{\text{Tree}}}((\{2\}, b) = 2$, $\delta^{AT_{\text{Tree}}}((\{2\}, c) = 2$, $\delta^{AT_{\text{Tree}}}((\{2\}, a) = 1$, $\delta^{AT_{\text{Tree}}}((\{1\}, \top) = 0$. The intersection of $AT_{\text{Tree}}$ and $AT_{\text{Twig}}$ is the automaton $AT_{\text{Tree}} \cap AT_{\text{Twig}} = (Q^{AT_{\text{Tree}} \cap AT_{\text{Twig}}}, \Sigma, F^{AT_{\text{Tree}} \cap AT_{\text{Twig}}}, \delta^{AT_{\text{Tree}} \cap AT_{\text{Twig}}})$ which is denoted by $\delta((\emptyset, \{q_{d_2}\}, d) = (3, 0, q_{d_2})$, $\delta((\emptyset, \{q_{d_2}\}, \{3\}, q_{d_2}) = (3, 0, q_{d_2})$, $\delta((\emptyset, \{q_{d_2}\}, \{3\}, q_{d_2}) = (3, 0, q_{d_2})$, $\delta((\emptyset, \{q_{d_2}\}, \{3\}, q_{d_2}) = (3, 0, q_{d_2})$.\}
Lemma 4.6 Given two UUTAs $A^1$ and $A^2$ and a tree $T$. Then, if $q_1 \in \rho^{A^1(T)}(v)$ and $q_2 \in \rho^{A^2(T)}(v)$ then $(q_1, q_2) \in \rho^{(A^1 \cap A^2)(T)}(v)$.

PROOF By induction on the height of the annotated nodes in $T$. The annotated nodes with height 1 are leaves. If a leaf node $v$ with a label $\sigma$ is annotated by $q_1 \in \rho^{A^1(T)}(v)$ then there is a transition $\delta^{A^1}(\emptyset, \sigma) = q_1$. If the same leaf node $v$ is annotated by $q_2 \in \rho^{A^2(T)}(v)$ then there is a transition $\delta^{A^2}(\emptyset, \sigma) = q_2$. Therefore, according to Definition 4.7 there is a transition $\delta^{A^1 \cap A^2}(\emptyset, 0, \sigma) = (q_1, q_2)$ and $(q_1, q_2) \in \rho^{(A^1 \cap A^2)(T)}(v)$. The induction step assumes that the Lemma is true for a node of height $\leq k$ and proves the Lemma for a node $v$ of height $k + 1$. If a node $v$ with label $\sigma$ is annotated as $q_1 \in \rho^{A^1(T)}(v)$ then there is a transition $\delta^{A^1}(S_1, \sigma) = q_1$. If the same leaf node $v$ is

\[
\delta((\{2\}, \{q_\bot, q_{d2}\}), c) = (2, q_c), \quad \delta((\emptyset, \{q_\bot, q_{d2}\}), c) = (2, q_c), \quad \delta((\{3\}, \{q_\bot, q_{d1}\}, b) = (2, q_b), \quad \delta((\emptyset, \{q_\bot, q_{d1}\}), b) = (2, q_b), \quad \delta((\{2\}, \{q_\bot, q_{c}\}, a) = (1, q_a), \quad \delta((\{2\}, \{q_\bot, q_{d2}\}), a) = (1, q_{d2}), \quad \delta((\{2\}, \{q_\bot\}, a) = (1, q_{\bot}), \quad \delta((\{1\}, \{q_a\}), \top) = (0, q_1), \quad \delta((\{1\}, \{q_{d2}\}, \top) = (1, q_{d2}), \quad \delta((\{1\}, \emptyset), \top) = (0, q_\bot), \quad \delta((\{1\}, \{q_{\bot}\}, \top) = (0, q_{\bot}).
\]

Figure 4.8: Illustration of a collection of trees. A tree in this collection has a root node that is labeled ‘a’. It has leaf nodes that are labeled by ‘d’, ‘b’ or ‘c’. It has a varying number of internal nodes which are labeled by ‘b’ or ‘c’. We denote the varying number of nodes by the symbol ‘*’ next to the labels ‘b’ and ‘c’. The gray triangle illustrates a subtree of these nodes. The trees are accepted by the automaton $A^{\text{Tree}}$ in example 4.6.

Lemmas 4.6 and 4.7 define the relation between individual UUTAs runs and the run of the intersected-UUTA.

**Lemma 4.6** Given two UUTAs $A^1$ and $A^2$ and a tree $T$. Then, if $q_1 \in \rho^{A^1(T)}(v)$ and $q_2 \in \rho^{A^2(T)}(v)$ then $(q_1, q_2) \in \rho^{(A^1 \cap A^2)(T)}(v)$.

**Proof** By induction on the height of the annotated nodes in $T$. The annotated nodes with height 1 are leaves. If a leaf node $v$ with a label $\sigma$ is annotated by $q_1 \in \rho^{A^1(T)}(v)$ then there is a transition $\delta^{A^1}(\emptyset, \sigma) = q_1$. If the same leaf node $v$ is annotated by $q_2 \in \rho^{A^2(T)}(v)$ then there is a transition $\delta^{A^2}(\emptyset, \sigma) = q_2$. Therefore, according to Definition 4.7, there is a transition $\delta^{A^1 \cap A^2}(\emptyset, 0, \sigma) = (q_1, q_2)$ and $(q_1, q_2) \in \rho^{(A^1 \cap A^2)(T)}(v)$. The induction step assumes that the Lemma is true for a node of height $\leq k$ and proves the Lemma for a node $v$ of height $k + 1$. If a node $v$ with label $\sigma$ is annotated as $q_1 \in \rho^{A^1(T)}(v)$ then there is a transition $\delta^{A^1}(S_1, \sigma) = q_1$. If the same leaf node $v$ is
annotated by \(q_2 \in \rho^{A^2(T)}(v)\) then there is a transition \(\delta^{A^2}(S_2, \sigma) = q_2\). Therefore, according to Definition 4.6, there is a transition \(\delta^{A^1 \cap A^2}((S_1, S_2), \sigma) = (q_1, q_2)\). We examine \(v_c \in \text{children}(T, v)\). According to Definition 4.2, both \(\rho^{A^1(T)}(v_c) \cap S_1 \neq \emptyset\) and \(\rho^{A^2(T)}(v_c) \cap S_2 \neq \emptyset\). Therefore, there are \(q_{c_1} \in \rho^{A^1(T)}(v_c) \cap S_1\) and \(q_{c_2} \in \rho^{A^2(T)}(v_c) \cap S_2\). From the induction assumption we know that \((q_{c_1}, q_{c_2}) \in \rho^{(A^1 \cap A^2)(T)}(v_c). \quad (S_1, S_2) = \bigcup_{v_c \in \text{children}(T, v)} (q_{c_1}, q_{c_2}) \in \rho^{(A^1 \cap A^2)(T)}(v_c)\) and \(\delta^{A^1 \cap A^2}((S_1, S_2), \sigma) = (q_1, q_2)\) on node \(v\). Therefore, \((q_1, q_2) \in \rho^{(A^1 \cap A^2)(T)}(v)\).

**Lemma 4.7** Given two STAs \((A^1, S^1)\) and \((A^2, S^2)\) and a tree \(T\). If \(q_1 \in \rho^{STA_{S^1}(T)}(v)\) and \(q_2 \in \rho^{STA_{S^2}(T)}(v)\) then \((q_1, q_2) \in \rho^{STA_{S^1 \cap S^2}(T)}(v)\).

**Proof** We use Lemma 4.6 to show by induction on the level of node \(v \in V^T\). In the basis of the induction, \(v = \text{root}(T)\) and \(\text{level}(T, v) = 1\). If \(q_1 \in \rho^{STA_{S^1}(T)}(v)\) then \((q_1) = F^{A^1}\). If \(q_2 \in \rho^{STA_{S^2}(T)}(v)\) then \((q_2) = F^{A^2}\). Therefore, \((q_1, q_2) = \rho_{(A^1 \cap A^2)_{pc}}\) and \((q_1, q_2) = \rho_{STA_{S^1 \cap S^2}(T)}(v)\). The induction step assumes that the Lemma is true for a node \(v\), where \(\text{level}(T, v) \leq k\), and proves the Lemma for a node \(v\) where \(\text{level}(T, v) = k + 1\). Because \(q_0^c \in \rho^{STA_{S^1}(T)}(v)\) there is \(q_1^p \in \rho^{STA_{S^1}(T)}(v_p), v_c \in \text{children}(T, v_p)\) and there is a transition \(\delta^{A^1_{pc}}(q_1^p, q_1^c) = q_1^c\). Because \(q_2^p \in \rho^{STA_{S^2}(T)}(v)\) there is \(q_2^p \in \rho^{STA_{S^2}(T)}(v_p)\) and there is a transition \(\delta^{A^2_{pc}}(q_2^p, q_2^c) = q_2^c\). From the induction assumption, \((q_1^p, q_2^p) \in \rho^{STA_{S^1 \cap S^2}(T)}(v_p)\). If \(\delta^{A^1_{pc}}(q_1^p, q_1^c) = q_1^c\) and \(\delta^{A^2_{pc}}(q_2^p, q_2^c) = q_2^c\) then there is a transition \(\delta^{(A^1 \cap A^2)_{pc}}((q_1^p, q_2^p), (q_1^c, q_2^c)) = (q_1^c, q_2^c)\). Due to Lemma 4.6 \((q_1^c, q_2^c) \in \rho^{(A^1 \cap A^2)(T)}(v_c)\). Therefore, \((q_1^c, q_2^c) \in \rho^{STA_{S^1 \cap S^2}(T)}(v_c)\).

4.4.2 FSA construction for \(STA_S^{Selecting}(ATree)\)

The \(STA_S^{Selecting}(ATree)\) operation intersects \(ATree\) and \(ASelecting\). The \(STA_S^{Selecting}(ATree)\) operation applies Algorithm 3 to construct \((ATree \cap ASelecting)_{pc}\) FSA from \(ATree \cap ASelecting\). The only preprocessing, which is needed for the application of Algorithm 3 is the calculation of the selecting states input. The input of the selecting states of \(STA_S^{Selecting}(ATree)\) in Algorithm 3 is the set \(Q^{ATree} \times S\) where \((q_{Tree}, q_{Selecting}) \in Q^{ATree} \times S\) for \(q_{Selecting} \in S\) and \(q_{Tree} \in Q^{ATree}\). The FSA construction limits \(F^{ATree \cap ASelecting}\) to a single state because \(F^{ATree \cap ASelecting} = \{q_0\}_{(ATree \cap ASelecting)_{pc}}\). Therefore, both \(F^{ATree}\) and \(F^{ASelecting}\) are limited to a single state. If \(ASelecting = ATwig\) then \(F^{ATwig} = \{q_T\}\) has a single state. The \(STA_S^{Ttwig}(ATree)\) algorithm modifies \(ATree\) and adds a new state \(q_{new} \in Q^{ATree}\). It replaces the original final states \(q_{old} \in F^{ATree}\) with

Part II: Querying of XML in a DB
the new state \( \{q_{\text{new}}\} = F^{A_{\text{Tree}}} \) and adds the transitions \( \delta^{A_{\text{Tree}}}(\{q_{\text{old}}\}, \top) = q_{\text{new}} \). The modified \( A_{\text{Tree}} \) contains a single state \( \{q_{\text{new}}\} = F^{A_{\text{Tree}}} \) that accepts trees that \( A_{\text{Twig}} \) accepts whose roots are labeled by \( \top \). The state \( 0 \in Q^{A_{\text{Tree}}} \) in \( A_{\text{Tree}} \), which is described in example 4.6, is an example for \( q_{\text{new}} \). Figure 4.9 illustrates the \( (A_{\text{Tree}} \cap A_{\text{Selecting}})^{pc} \) that was constructed from \( A_{\text{Tree}} \cap A_{\text{Twig}} \) in example 4.6.

Figure 4.9: An illustration of the FSA \( (A_{\text{Tree}} \cap A_{\text{Twig}})^{pc} \) that is constructed from \( A_{\text{Tree}} \cap A_{\text{Twig}} \) in example 4.6. A state \( \langle q_{A_{\text{Tree}}}^{0}, q_{A_{\text{Selecting}}}^{0} \rangle \) is denoted by a labeled circle. The label denotes the state Id. The transition \( \delta^{(A_{\text{Tree}} \cap A_{\text{Twig}})^{pc}}(\langle q_{p_{A_{\text{Tree}}}^{0}}, q_{p_{A_{\text{Selecting}}}^{0}} \rangle, \langle q_{c_{A_{\text{Tree}}}^{0}}, q_{c_{A_{\text{Selecting}}}^{0}} \rangle) = \langle q_{c_{A_{\text{Tree}}}^{0}}, q_{c_{A_{\text{Selecting}}}^{0}} \rangle \) is denoted by an edge from \( \langle q_{p_{A_{\text{Tree}}}^{0}}, q_{p_{A_{\text{Selecting}}}^{0}} \rangle \) circle to \( \langle q_{c_{A_{\text{Tree}}}^{0}}, q_{c_{A_{\text{Selecting}}}^{0}} \rangle \) circle. The start state \( q_{0}^{(A_{\text{Tree}} \cap A_{\text{Twig}})^{pc}} \) is denoted by an incoming edge. The states in \( F^{(A_{\text{Tree}} \cap A_{\text{ Twig}})^{pc}} \) are denoted by double circles.

### 4.4.3 Top-down traversal of \( STA_{S}^{A_{\text{Selecting}}} (A_{\text{Tree}}) \)

A top-down traversal of \( STA(A_{\text{Tree}}) \) prunes only the accessible states of FSA \( (A_{\text{Tree}} \cap A_{\text{Selecting}})^{pc} \). The top-down traversal starts in the start state \( q_{0}^{(A_{\text{Tree}} \cap A_{\text{Selecting}})^{pc}} = \langle q_{0}^{A_{\text{Tree}}^{pc}}, q_{0}^{A_{\text{Selecting}}^{pc}} \rangle \) that annotates the top of the trees i.e. the roots. The traversal continues to accessible states that annotate the internal nodes. The traversal selects the states \( q_{\text{Tree}} \in Q^{A_{\text{Tree}}} \) that compose the final states \( \langle q_{\text{Tree}}, q_{\text{Selecting}} \rangle \in F^{(A_{\text{Tree}} \cap A_{\text{Selecting}})^{pc}} \) where \( q_{\text{Selecting}} \in \).
Algorithm 5: The $\text{STA}_{S}^{\text{Selecting}} (ATree)$ top-down traversal algorithm.

begin

accessible_states $\leftarrow \emptyset$;

forall $\delta(\text{ATree} \cap \text{ASelecting})_{pc} (\langle q_{p\text{Tree}}, q_{p\text{Selecting}} \rangle, \langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle) = \langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle$ do

if $\langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle \not\in \text{accessible_states}$ then

Add $\langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle$ to accessible_states;

if $\langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle \in F(\text{ATree} \cap \text{ASelecting})_{pc}$ then

Add $q_{c\text{Selecting}}$ to $\rho(\text{Selecting}, S)(\text{ATree})$

end

top-down traversal($\langle q_{c\text{Tree}}, q_{c\text{Selecting}} \rangle$)

end

Example 4.7 Figure 4.10 describes the $(\text{ATree} \cap \text{ASelecting})_{pc}$ from Fig 4.9 after the removal of the inaccessible states $\langle 0, q_{\bot} \rangle$, $\langle 0, q_{u2} \rangle$, and $\langle 1, q_{u2} \rangle$ that were not traversed by Algorithm 5. The top-down traversal selects the $F(\text{ATree} \cap \text{ASelecting})_{pc}$ final states: $\langle 1, q_{a} \rangle$, $\langle 2, q_{b} \rangle$, $\langle 2, q_{c} \rangle$, $\langle 3, q_{d1} \rangle$, and $\langle 3, q_{d2} \rangle$. The states 1, 2 and 3 in $Q_{\text{Tree}}$ are the selected states, which compose the final states, in this $\text{STA}_{S}^{\text{Selecting}} (\text{ATree})$ operation.
4.4.4 The relation between \( \text{STA}(A^{\text{Tree}}) \) and \( \text{STA}(T) \)

Given a STA \( (A^{\text{Selecting}}, S) \), a UUTA \( A^{\text{Tree}} \) and a tree \( T \), which is accepted by \( A^{\text{Tree}} \). Theorem 4.4 describes the relation between the \( \text{STA}_S^{\text{Selecting}}(T) \), \( \text{STA}_S^{\text{Selecting}}(A^{\text{Tree}}) \) and \( \text{STA}^{A^{\text{Tree}}}(T) \). If a node \( v \) is a selected node in \( \text{STA}_S^{\text{Selecting}}(T) \), where \( q^{\text{Selecting}} \) is the selecting state, then the following is true:

1. Node \( v \) is a selected node in \( \text{STA}^{A^{\text{Tree}}}(T) \);
2. Node \( v \) is selected by the selecting state \( q^{\text{Tree}} \in Q^{A^{\text{Tree}}} \);
3. \( q^{\text{Tree}} \) is a selected state, which is selected by \( q^{\text{Selecting}} \), in \( \text{STA}^{A^{\text{Selecting}}}(A^{\text{Tree}}) \). Lemma 4.8 is needed for the proof of Theorem 4.4.

**Lemma 4.8** Given STA \( (A, Q^A) \) and a tree \( T \). If \( T \) is accepted by \( A \) then \( \rho_{\text{STA}^A_{Q^A}(T)} \) maps every node \( v \in V^T \) to at least one state in \( Q^A \).

**Proof** By negation. We assume that there is a node \( v \) where \( \rho_{\text{STA}^A_{Q^A}(T)}(v) = \emptyset \). Therefore, there is no transition in \( \delta^{A^{\text{Tree}}} \) from \( q_p \in \rho^{A^{\text{Tree}}}(v_p) \) to \( q_c \in \rho^{A(T)}(v) \) where \( v \in \text{children}(T, v_p) \). From Lemmas 4.4 and 4.5 we know that \( L(A_{pc}) = L^*(A, q_0^{A_{pc}}) \).

Part II: Querying of XML in a DB
Therefore, there is no transition $\delta^A(S \cup \{ q_p \}, \text{label}^T(v_p)) = q_p$, $q_p \in \rho^{A(T)}(v)$, that annotates $q_p \in \rho^{A(T)}(v_p)$. From Definition 4.2, we know that $S \cap \rho^{A(T)}(v) = S \cap \{ q_e \} \neq \emptyset$. Therefore, there is no transition $\delta^A(S, \text{label}^T(v_p)) = q$ that annotates $v_p$. In the same way, all the ancestors of $v$ including root($T$) can neither be annotated nor selected.

**Theorem 4.4** Given a STA $(A^{\text{Selecting}}, S)$, a UUTA $A^{\text{Tree}}$ and a tree $T$ that is accepted by $A^{\text{Tree}}$. If a node $v$ satisfies $q_{\text{Selecting}} \in \rho_{S^{\text{UUTA}}\text{Selecting}}(T)(v)$ then exists $q_{\text{Tree}} \in \rho_{S^{\text{UUTA}}\text{Tree}}(v)$ such that $q_{\text{Selecting}} \in \rho_{S^{\text{UUTA}}\text{Selecting}}(v)$. □

**Proof** From the Lemma’s claim, we get that $q_{\text{Selecting}} \in \rho_{S^{\text{UUTA}}\text{Selecting}}(T)(v)$. We also know that the tree $T$ is accepted by $A^{\text{Tree}}$. Therefore, from Lemma 4.8, we know that there is a state $q_{\text{Tree}} \in \rho_{S^{\text{UUTA}}\text{Tree}}(v)$. From Lemma 4.7, we know that $(q_{\text{Tree}}, q_{\text{Selecting}}) \in \rho_{S^{\text{UUTA}}\text{Tree} \times S^{\text{Selecting}}}(T)$. Because of the STA($T$) operation top down traverse, which is described in Algorithm 4, the FSA $(A^{\text{Tree} \cap A^{\text{Selecting}}})_{pc}$ derives the state $(q_{\text{Tree}}, q_{\text{Selecting}})$. $(A^{\text{Tree} \cap A^{\text{Selecting}}})_{pc}$ derives the input symbol $(q_{\text{Tree}}, q_{\text{Selecting}})$ by one of the states in $F(A^{\text{Tree} \cap A^{\text{Selecting}}})_{pc}$. $\rho_{S^{\text{Selecting}}}^{A^{\text{Tree}}}$ maps $q_{\text{Selecting}}$ to $q_{\text{Tree}}$. Therefore, $q_{\text{Selecting}} \in \rho_{S^{\text{Selecting}}}^{A^{\text{UUTA}}}(q_{\text{Tree}})$. □

Both the holistic join and the holistic index operations are based on Theorem 4.4. Given a tree-UUTA $A^{\text{Tree}}$ and a tree $T$, which is accepted by $A^{\text{Tree}}$, then the holistic index offline operation maps a node $v$ into $q_{\text{Tree}} \in \rho_{S^{\text{UUTA}}\text{Tree}}(v)$. The input to the holistic index online operation is STA $(A^{\text{Twig}}, S)$ that defines a twig pattern. The index online operation calculates $\rho_{S^{\text{Twig}}}^{A^{\text{Tree}}}(v)$ and returns a node $v$ that is mapped into $q_{\text{Tree}}$ where $q_{\text{Tree}}$ satisfies $\rho_{S^{\text{Twig}}}^{A^{\text{UUTA}}}(q_{\text{Tree}}) \neq \emptyset$. Theorem 4.4 ensures that if $\rho_{S^{\text{Twig}}}^{A^{\text{UUTA}}}(v) \neq \emptyset$, i.e. the node $v$ is selected by the twig-UUTA, then $v$ is returned by the index operation. The holistic join operation uses Theorem 4.4 in negation. Given a twig pattern STA $(A^{\text{Twig}}, S)$ and a UUTA $A^{\text{Tree}}$ that accepts the stored XML data structure tree $T$. The join operation checks if $q_{\text{Tree}}$ is not a selected state in the $S^{\text{Twig}}_A(T)$ operation. By Theorem 4.4 terms, the join operation checks if $\rho_{S^{\text{Twig}}}^{A^{\text{Tree}}}(q_{\text{Tree}}) = \emptyset$. If $\rho_{S^{\text{Twig}}}^{A^{\text{Tree}}}(q_{\text{Tree}}) = \emptyset$ then a $q_{\text{Tree}} \in \rho_{S^{\text{UUTA}}\text{Tree}}(v)$, where $q_{\text{Selecting}} \in \rho_{S^{\text{UUTA}}\text{Selecting}}(q_{\text{Tree}})$, does not exist. Therefore, $q_{\text{Selecting}} \notin \rho_{S^{\text{Twig}}}^{A^{\text{UUTA}}}(v)$. Therefore, node $v$, which is annotated by the state $q_{\text{Tree}}$, can be filtered from the processing of the join operation.
4.5 Efficient FSA construction of \((A^{Tree} \cap A^{Selecting})_{pc}\)

The FSA-construction in Algorithm \(\mathbf{3}\) was designed without taking into consideration its performance issues. The core operation of a tree automata, which is presented in this thesis, is the \(STA^{Selecting}_{S}(A^{Tree})\) operation. The \(STA^{Selecting}_{S}(A^{Tree})\) core operation is the \((A^{Tree} \cap A^{Selecting})_{pc}\) FSA construction that is performed by Algorithm \(\mathbf{3}\). Therefore, the algorithm for \((A^{Tree} \cap A^{Selecting})_{pc}\) construction has to be optimized. The optimization is achieved by reducing the language of deriving-states strings that \((A^{Tree} \cap A^{Selecting})_{pc}\) recognizes. Given a tree \(T\), the deriving-states strings language in Definition \(\mathbf{4.4}\) recognizes the deriving-states string \(\langle q_{v_{1Tree}}, q_{v_{1Selecting}} \rangle, \ldots, \langle q_{v_{nTree}}, q_{v_{nSelecting}} \rangle\) that satisfies: 1. \(v_{1}, \ldots, v_{n}\) is a path in \(T\) from node \(v_{1} = \text{root}(T)\); 2. \(\langle q_{v_{iTree}}, q_{v_{iSelecting}} \rangle \in \rho^{STA^{Selecting}_{S}(T)}(v_{i}), 1 \leq i \leq n;\) 3. The state \(\langle q_{v_{i+1Tree}}, q_{v_{i+1Selecting}} \rangle\) derives the state \(\langle q_{v_{iTree}}, q_{v_{iSelecting}} \rangle\) by the transition \(\delta_{A^{Tree} \cap A^{Selecting}}(S \cup \{ \langle q_{v_{i+1Tree}}, q_{v_{i+1Selecting}} \rangle \}, \sigma) = \langle q_{v_{iTree}}, q_{v_{iSelecting}} \rangle, 1 \leq i \leq n - 1.\) The optimized intersected-FSA recognizes strings with an additional constraint such that both \(q_{v_{iTree}} \in \rho^{A^{Tree}(T)}(v_{i})\) and \(q_{v_{iSelecting}} \in \rho^{A^{Selecting}(T)}(v_{i}), 1 \leq i \leq n.\) This constraint does not affect the validity of the \(STA^{Selecting}_{S}(A^{Tree})\) operation since the proof of Theorem \(\mathbf{4.4}\) considers only the states \(\langle q_{pTree}, q_{pSelecting} \rangle \in \rho^{(A^{Tree} \cap A^{Selecting})_{pc}(T)(v)}\) where \(q_{pSelecting} \in \rho^{A^{Selecting}(T)}(v)\) and \(q_{pTree} \in \rho^{A^{Tree}(T)}(v)\).

The \((A^{Tree} \cap A^{Selecting})_{pc}\) construction in Algorithm \(\mathbf{6}\) adds the state \(\langle q_{pTree}, q_{pSelecting} \rangle\) to \(Q^{(A^{Tree} \cap A^{Selecting})_{pc}}\) only if there is a tree \(T\) and a node \(v\) that satisfy \(q_{pSelecting} \in \rho^{A^{Selecting}(T)}(v)\) and \(q_{pTree} \in \rho^{A^{Tree}(T)}(v)\). Therefore, from Lemma \(\mathbf{4.6}\) \(\langle q_{pTree}, q_{pSelecting} \rangle \in \rho^{(A^{Tree} \cap A^{Selecting})_{pc}(T)(v)}\).

Initially, Algorithm \(\mathbf{6}\) constructs \(A^{Selecting}_{pc}\) and \(A^{Tree}_{pc}\) and adds transitions to both \(A^{Selecting}_{pc}\) and \(A^{Tree}_{pc}\) from states that have no outgoing transitions to the empty state \(q_{0}.\) Figure \(\mathbf{4.11}\) illustrates \(A^{Tree}_{pc}\) and \(A^{Twig}_{pc}\) that are constructed from \(A^{Tree}_{pc}\) and \(A^{Twig}_{pc}\), respectively, in example \(\mathbf{4.6}\). The construction of \(A^{Tree}_{pc}\) and \(A^{Twig}_{pc}\) by Algorithm \(\mathbf{3}\) is modified. Instead of constructing the transition \(\delta^{A_{pc}}(q_{p}, q_{c}) = q_{c}\), the FSA constructs the transition \(\delta^{A_{pc}}(q_{p}, a) = q_{c}\) where \(\delta^{A}(S \cup \{q_{c}\}, a) = q_{p}\) is the transition that annotates \(q_{p}\) in the FSA construction.
Figure 4.11: Illustration of $A_{pc}^{Tree}$ and $A_{pc}^{Twig}$, which are constructed from $A^{Tree}$ and $A^{Twig}$, respectively, in example 4.6. A state $q$ is denoted by a labeled circle. The label denotes the state Id. A transition $\delta^{A_{pc}}(q_p, a) = q_c$ is denoted by an edge from $q_p$ circle to $q_c$ circle. An edge $(q_p, q_c)$ label $\Sigma$ denotes the transitions $\delta^{A_{pc}}(q_p, a) = q_c$. The label of the edge $(q_p, q_c)$ is $a \in \Sigma$. The start state $q_0^{A_{pc}}$ is denoted by an incoming edge. The states in $F^{A_{pc}}$ are denoted by double circles.

The FSA construction in Algorithm 6 is recursive. The recursion starts from states $q_\emptyset$. The first call finds states that annotate leaves (the bottom) in trees. The recursion propagates bottom-up from states, which annotate children, to states that annotate parents. Each recursion step receives children states $q_{cSelecting}^{A_{pc}} \in Q^{A_{Selecting}}$ and $q_{cTree}^{A_{pc}} \in Q^{A_{Tree}}$. 

Part II Querying of XML in a DB
which annotate children nodes, as an input. A precondition in each recursion step is that \( q_{\text{tree}}, q_{\text{selecting}} \) \( \in Q(A_{\text{tree}} \cap A_{\text{selecting}})_{pc} \) and there is a tree \( T \) and a node \( v \) such that \( q_{\text{selecting}} \in \rho_{A_{\text{selecting}}}(T)(v) \) and \( q_{\text{tree}} \in \rho_{A_{\text{tree}}}(T)(v) \). The recursive step searches new states \( q_{p_{\text{selecting}}} \in \rho_{A_{\text{selecting}}}(T)(v_p) \) and \( q_{p_{\text{tree}}} \in \rho_{A_{\text{tree}}}(T)(v_p) \) that annotate the parent node \( v_p = \text{parent}(T, v) \). From Lemma 4.7, we know that if \( q_{p_{\text{selecting}}} \in \rho_{A_{\text{selecting}}}(T)(v_p) \) and \( q_{p_{\text{tree}}} \in \rho_{A_{\text{tree}}}(T)(v_p) \) then \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \in \rho(A_{\text{tree}} \cap A_{\text{selecting}})(T)(v_p) \). Therefore, when state \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \) is found then it is added to \( Q(A_{\text{tree}} \cap A_{\text{selecting}})_{pc} \). When the state \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \) is added to \( Q(A_{\text{tree}} \cap A_{\text{selecting}})_{pc} \), the recursion is called with \( q_{p_{\text{tree}}} \) and \( q_{p_{\text{selecting}}} \) as parameters.

Finding out if \( q_{p_{\text{tree}}} \in \rho_{A_{\text{tree}}}(T)(v_p) \) and if \( q_{p_{\text{selecting}}} \in \rho_{A_{\text{selecting}}}(T)(v_p) \) is guided by \( A_{pc_{\text{tree}}} \) and \( A_{pc_{\text{selecting}}} \), respectively. In the mapping \( \text{maybe}_\text{children} \), the algorithm maps for every parent state candidate \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \) the children states \( q_{c_{\text{tree}}}, q_{c_{\text{selecting}}} \) \( \in Q(A_{\text{tree}} \cap A_{\text{selecting}})_{pc} \) where \( \delta_{A_{pc_{\text{tree}}}}(q_{p_{\text{tree}}}, a) = q_{c_{\text{tree}}} \) and \( \delta_{A_{pc_{\text{selecting}}}}(q_{p_{\text{selecting}}}, a) = q_{c_{\text{selecting}}} \) exist. The children states may belong to a number of transitions of the parent states. Therefore, the algorithm checks if there is a subset \( S \subseteq \text{maybe}_\text{children}(\langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle) \) which applies transitions that derive \( q_{p_{\text{tree}}} \) and \( q_{p_{\text{selecting}}} \). If the set \( S \) exists then a tree \( T \) exists where \( q_{p_{\text{tree}}} \in \rho_{A_{\text{tree}}}(T)(\text{root}(T)) \) and \( q_{p_{\text{selecting}}} \in \rho_{A_{\text{selecting}}}(T)(\text{root}(T)) \). Therefore, \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \) \( \in \rho_{A_{\text{tree}}} \cap A_{\text{selecting}}(\text{root}(T)) \) and \( \langle q_{p_{\text{tree}}}, q_{p_{\text{selecting}}} \rangle \) can be added to \( Q(A_{\text{tree}} \cap A_{\text{selecting}})_{pc} \).
**Algorithm 6:** An efficient construction function for intersected-FSA.

ConstructIntersectedFSA\( (\mathcal{T} \cap A)_{pc} \equiv (A_{selecting}, S) \)

**Output:** \( (\mathcal{T} \cap A)_{pc} \)

**Data:** \( \text{maybe} \_ \text{children} : Q(\mathcal{T} \cap A)_{pc} \mapsto 2Q(\mathcal{T} \cap A)_{pc} \)

begin

\( \mathcal{T} \cap A \leftarrow \text{Construct-FSA}(\mathcal{T}, Q) \); \hspace{1cm} \text{[Algorithm 3]}

Add \( q_0 \) to \( \mathcal{T} \);

forall transition \( \delta^{\mathcal{T}}(q, a) = q \) do

Add transition \( \delta_{pc}^{\mathcal{T}}(q, a) = q_0 \);

end

\( \mathcal{T} \cap A_{pc} \leftarrow \text{Construct-FSA}(A_{selecting}, S) \); \hspace{1cm} \text{[Algorithm 3]}

Add \( q_0 \) to \( A_{pc} \);

forall transition \( \delta^{A_{pc}}(q, a) = q \) do

Add transition \( \delta_{pc}^{A_{pc}}(q, a) = q_0 \);

end

return CreateTransitions\( (q_0, q_0, \text{maybe} \_ \text{children}) \);

end

CreateTransitions\( (q_{pc} \in Q_{A_{pc}} \cap A_{selecting}) \)

\( q_{pc} \in Q_{A_{pc}}^{A_{pc}} \cap A_{pc} \)

**Data:** \( \mathcal{T}, A_{pc}, A_{selecting}, A_{pc} \)

**Output:** \( (\mathcal{T} \cap A)_{pc} \)

begin

forall \( \delta^{\mathcal{T}_{pc}}(q_{pc}, a) = q \) do \hspace{1cm} \text{[Algorithm 7]}

\( \delta^{A_{pc}}(q_{pc}, a) = q \)

Add transition \( \delta_{pc}^{A_{pc}}(q, a) = q \);

end

CreateTransitions\( (q_{pc} \in Q_{pc} \cap A_{pc}) \)

Part II: Querying of XML in a DB
Algorithm 7: The children-search function.

children-search (⟨q_{pSelecting}, q_{pTree}⟩ ∈ Q^{(ATree ∩ ASelecting)}_pc, a ∈ Σ, maybe_children ⊆ Q^{(ATree ∩ ASelecting)}_pc)

Data: ATree, ASelecting

Output: children-sets ⊆ 2Q^{(ATree ∩ ASelecting)}_pc

begin

forall δ^{ATree} (S_{Tree}, a) = q_{pTree} do

forall δ^{ASelecting} (S_{Selecting}, a) = q_{pSelecting} do

if exists S ⊆ maybe_children where

U_{(q_{Tree}, q_{Selecting}) ∈ S} q_{Tree} = S_{Tree} and

U_{(q_{Tree}, q_{Selecting}) ∈ S} q_{Selecting} = S_{Selecting} then

Add S to children-sets;

end

Example 4.8 Table 4.1 describes the application of Algorithm 6 to ATree and ATwig, which are described in example 4.6. The rows of the table show the state ⟨q_{cTree}, q_{cSelecting}⟩ where q_{cSelecting} ∈ Q^{ASelecting} and q_{cTree} ∈ Q^{ATree} are the inputs to the CreateTransitions, which is a recursive function call in the operation of Algorithm 6. Figures 4.12 (e)-(f) describe the (ATree ∩ ASelecting)_pc in the beginning of the recursive calls that rows 2-6 and 14 in table 4.1 describe. We see that the FSA (ATree ∩ ASelecting)_pc, which was constructed by Algorithm 3 in Fig. 4.9, includes the state ⟨3, q_{u_2}⟩. The FSA (ATree ∩ ASelecting)_pc in Fig. 4.12(f) does not include the state ⟨3, q_{u_2}⟩. Algorithm 6 does not construct the state ⟨3, q_{u_2}⟩ because the state q_{u_2} annotates internal tree nodes whereas state 3 annotates leafs. Therefore, there is no node v which both q_{u_2} ∈ ρ^{ASelecting}(T)(v) and 3 ∈ ρ^{ATree}(T)(v) annotate.  

Part II Querying of XML in a DB
Table 4.1: Details of the recursive calls to the function *CreateTransitions* in the application of Algorithm 6 to $A_{Tree}$ and $A_{Twig}$ that were described in example 4.6.

<table>
<thead>
<tr>
<th></th>
<th>$\langle q_0, q_0 \rangle$</th>
<th>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</th>
<th>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</th>
<th>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</th>
<th>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>2</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>4</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>5</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>6</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>7</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>8</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>9</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>10</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>11</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>12</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>13</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
<tr>
<td>14</td>
<td>$\langle q_0, q_0 \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{\bot}), (2, q_{\bot}), (1, q_{\bot}) \rangle$</td>
<td>$\langle q_0, q_0, (3, q_{d_2}) \rangle$</td>
</tr>
</tbody>
</table>
Figure 4.12: Demonstration of how Algorithm 6 operates. Figures (a)-(f) illustrate \((A^{\text{Tree}} \cap A^{\text{Twig}})_{pc}\). A state \(\langle q_{\text{Tree}}, q_{\text{Twig}} \rangle\) is denoted by a labeled circle. The label denotes the state Id. A transition \(\delta((A^{\text{Tree}} \cap A^{\text{Twig}})_{pc})_{pc}(\langle q_{p\text{Tree}}, q_{p\text{Twig}} \rangle, \langle q_{c\text{Tree}}, q_{c\text{Twig}} \rangle) = \langle q_{c\text{Tree}}, q_{c\text{Twig}} \rangle\) is denoted by an edge from \(q_{p\text{Tree}}, q_{p\text{Twig}} \rangle\) circle to \(q_{c\text{Tree}}, q_{c\text{Twig}} \rangle\) circle. The start state \(q_{0}(A^{\text{Tree}} \cap A^{\text{Twig}})_{pc}\) is denoted by an incoming edge. The states in \(F((A^{\text{Tree}} \cap A^{\text{Twig}})_{pc})\) are denoted by double circles. A dotted circle is a parent state candidate \(\langle q_{\text{Tree}}, q_{\text{Twig}} \rangle\), which is mapped in maybe_children, but yet \(\langle q_{\text{Tree}}, q_{\text{Twig}} \rangle \notin Q((A^{\text{Tree}} \cap A^{\text{Twig}})_{pc})\).

4.5.1 The children-search techniques

The core of Algorithm 6 is the children-search function that is described in Algorithm 7. The function searches for a children-set \(S \subseteq \text{maybe}_\text{children}(\langle q_{p\text{Tree}}, q_{p\text{Selecting}} \rangle)\).
where
\[
\delta^\text{ATree} \left( \bigcup_{(q_{\text{Tree}}, q_{\text{Selecting}}) \in S_{\text{Tree}}} q_{\text{Tree}}, a \right) = q_{\text{PTree}} \quad \text{and} \quad \delta^\text{ASelecting} \left( \bigcup_{(q_{\text{Tree}}, q_{\text{Selecting}}) \in S_{\text{Selecting}}} q_{\text{Selecting}}, a \right) = q_{\text{PSelecting}}.
\]
The children-search function has to match all the children-sets combinations of \( S_{\text{Tree}} \) and \( S_{\text{Selecting}} \), where \( \delta^\text{ATree}(S_{\text{Tree}}, a) = q_{\text{PTree}} \) and \( \delta^\text{ASelecting}(S_{\text{Selecting}}, a) = q_{\text{PSelecting}} \), and to compare it to \( \text{maybe\_children}(\langle q_{\text{PTree}}, q_{\text{PSelecting}} \rangle) \) children subset.

Example 4.8 demonstrates how Algorithm 6 operates. In this section, we examine in more details one parent state candidate \( \langle q_{\text{PTree}}, q_{\text{PTwig}} \rangle = \langle 2, q_c \rangle \) that is constructed in the \( (A_{\text{Tree}} \cap A_{\text{Twig}})_{pc} \) FSA construction in example 4.8. The possible children-sets \( S_{\text{Tree}} \in \{\{3, 2\}, \{3\}, \{2\}\} \) are taken from the transitions \( \delta^\text{ATree}(\{2\}, c) = 2 \) and \( \delta^\text{ATree}(\{3, 2\}, c) = 2 \). The possible children-sets \( S_{\text{Twig}} \in \{\{q_\perp, q_d\}, \{q_\perp, q_d \uparrow\}\}\) are taken from the transitions \( \delta^\text{ATwig}(\{q_\perp, q_d \uparrow\}, 2) = 2 \) and \( \delta^\text{ATwig}(\{q_\perp, q_d\}, 2) = 2 \). There are six children-sets combinations to match against
\[
\text{maybe\_children}(\langle 2, q_c \rangle) = \{\{3, 2\}, \{q_\perp, q_d\}\}, \{\{3, 2\}, \{q_\perp, q_d \uparrow\}\}\}.
\]
When the state \( \langle 2, q_c \rangle \) is visited by \( \langle q_{c\text{Tree}}, q_{c\text{Twig}} \rangle \in \{\{3, q_\perp\}, \{3, q_d\}, \{2, q_\perp\}, \{2, q_d\}\} \) then these six children-sets combinations are checked.

A more efficient technique is to store the children-set combinations for each \( \langle q_{\text{PTree}}, q_{\text{PSelecting}} \rangle \). When \( \langle q_{\text{PTree}}, q_{\text{PSelecting}} \rangle \) is visited, the algorithm marks the recursion input variables \( q_{c\text{Tree}} \) and \( q_{c\text{Selecting}} \) in the stored children-set combinations. The algorithm marks combinations that include both \( q_{c\text{Tree}} \) and \( q_{c\text{Selecting}} \). If the entire children-set combination is marked then a children-set \( S \subseteq \text{maybe\_children}(\langle q_{\text{PTree}}, q_{\text{PSelecting}} \rangle) \) is found. In example 4.8 we store the children-sets combinations \( \{\{3, 2\}, \{q_\perp, q_d\}\}, \{\{3\}, \{q_\perp, q_d \uparrow\}\}\} \) for the parent state candidate \( \langle q_{\text{PTree}}, q_{\text{PTwig}} \rangle = \langle 2, q_c \rangle \). Table 4.2 describes the marking of \( \langle 2, q_c \rangle \) children-sets combinations. The FSA construction finds four children-sets combinations: \( \{\{3, 2\}, \{q_\perp, q_d \uparrow\}\}, \{\{3, 2\}, \{q_\perp, q_d\}\}, \{\{2\}, \{q_\perp, q_d\}\}, \{\{3\}, \{q_\perp, q_d\}\} \).
Table 4.2: The marked children-sets combinations status. The first column describes the input to the recursion steps that visits the parent state $\langle 2, q_\perp \rangle$. The second column describes the status of the children-sets combinations. A bold state denotes a marked state.

When restricting the forms of both $A_{T \text{ree}}$ and $A_{S \text{eleting}}$, then the children-search function is simplified. Definition 4.7 defines the restriction for $A_{T \text{ree}}$. Lemmas 4.9 and 4.10 explain why we can simplify the children-search operation.

**Definition 4.7** Given UUTA $A$ and a state $q \in Q^A$. A maximal children state $S_{A,q}^{\text{Max}}$ is a set of states that satisfies: 1. There is $\delta^A(S_{A,q}^{\text{Max}}, a) = q$; 2. Every other transition $\delta^A(S, a) = q$ satisfies $S \subseteq S_{A,q}^{\text{Max}}$. An automaton $A$ has maximal children-sets if there is a set $S_{A,q}^{\text{Max}}$ for every $q \in Q^A$. UUTA $A$, which has maximal children-sets, is denoted by $A_{S_{A,q}^{\text{Max}}}$.

**Lemma 4.9** Given a UUTA $A_{S \text{eleting}} = A_{T \text{ree}}$ and a UUTA $A_{T \text{ree}}$ then $\langle q_{T \text{ree}}, q_\perp \rangle \in Q(A_{T \text{ree}} \cap A_{S \text{eleting}})_p$ for every $q \in Q^A_{T \text{ree}}$.

**Proof** By induction on the FSA construction algorithm.

**Lemma 4.10** Given a UUTA $A_{S \text{eleting}} = A_{T \text{ree}}$, $A_{T \text{ree}} = A_{S_{A,q}^{\text{Max}}}$ and a set $S \subseteq Q(\bigcap_{q_{T \text{ree}} \in S} q_{T \text{ree}})$, where $\delta^{A_{T \text{ree}}}(q_{T \text{ree}}, q_{T \text{ree}}) \in S$, $q_{T \text{ree}} \in S_{A_{T \text{ree}}}^{\text{Max}}$. Then, there is a set $S_{A_{T \text{ree}}}^{\text{Max}} \subseteq Q(\bigcap_{q_{T \text{ree}} \in S} q_{T \text{ree}})$, $\delta^{A_{T \text{ree}}}(q_{T \text{ree}} \cap S_{A_{T \text{ree}}}^{\text{Max}} q_{T \text{ree}}, a) = q_{T \text{ree}}$ and $\delta^{A_{T \text{ree}}}(q_{T \text{ree}} \cap S_{A_{T \text{ree}}}^{\text{Max}} q_{T \text{ree}}, a) = q_{T \text{ree}}$.

**Proof** We examine the set $S$. From definition 4.7, we get that $\bigcup_{q_{T \text{ree}} \in S} q_{T \text{ree}} \subseteq S_{A_{T \text{ree}}}^{\text{Max}}$. We examine the set $S_{A_{T \text{ree}}}^{\text{Max}} \subseteq Q(\bigcap_{q_{T \text{ree}} \in S} q_{T \text{ree}})$ where the state $q_{T \text{ree}} \in S_{A_{T \text{ree}}}^{\text{Max}}$. If $q_{T \text{ree}} \notin \bigcup_{q_{T \text{ree}} \in S} q_{T \text{ree}}$ and $q_{T \text{ree}} \in S_{A_{T \text{ree}}}^{\text{Max}}$. According to Lemma 4.9, there are states $\langle q_{T \text{ree}}, q_\perp \rangle \in Q(\bigcap_{q_{T \text{ree}} \in S} q_{T \text{ree}})$ for every $q_{T \text{ree}} \in S_{A_{T \text{ree}}}^{\text{Max}}$. We construct $S_{A_{T \text{ree}}}^{\text{Max}} = S \cup \bigcup_{q_{T \text{ree}} \in S_{A_{T \text{ree}}}^{\text{Max}}} q_{T \text{ree}}$. The construction satisfies $S \subseteq S_{A_{T \text{ree}}}^{\text{Max}}$. Part II Querying of XML in a DB
It also satisfies
\[ \delta^{ATwig}((\bigcup_{(q_{T ree},q_{Twig}) \in S_{Max}^{T ree}} q_{Twig}, a)) = q_{PTwig} \]
where \( q_{cT ree} \in S_{Max}^{T ree} \), contribute the default state \( q_{\perp} \in Q^{ATwig} \) that exists in every transition
\[ \delta^{ATwig}(\bigcup_{(q_{T ree},q_{Twig}) \in S_{Max}^{T ree}} q_{T ree}, a) = q_{PTree} \] because \( \bigcup_{(q_{T ree},q_{Twig}) \in S_{Max}^{T ree}} q_{T ree} = S_{Max}^{T ree} \cdot q_{PTree}. \)

From Lemma 4.11, we get that if \( A_{Selecting} = ATwig \) and \( A_{Tree} = A_{Tree}^{S_{Max}} \) then Algorithm 6 which constructs \( (A_{Tree} \cap ATwig)_{pec} \), does not have to perform a children-search on \( A_{Tree} \). If the algorithm finds children-set \( S \) where \( \delta^{ATwig}(\bigcup_{(q_{T ree},q_{Twig}) \in S} q_{Twig}, a) = q_{PTwig} \) then there is set \( S_{Max}^{T ree} \) where \( \delta^{A_{Tree}}(\bigcup_{(q_{T ree},q_{Twig}) \in S_{Max}^{T ree}} q_{T ree}, a) = q_{PTree} \) and
\[ \delta^{ATwig}(\bigcup_{(q_{T ree},q_{Twig}) \in S} q_{Twig}, a) = q_{PTwig}. \] The children-search function in Algorithm 7 is updated. The function searches for a children-set \( S \subseteq maybe_children(\langle q_{PTree}, q_{PTwig} \rangle) \) where \( \delta^{ATwig}(\bigcup_{(q_{T ree},q_{Twig}) \in S} q_{Twig}, a) = q_{PTwig}. \)

In example 4.8 \( A_{Tree} = A_{S_{Max}}, S_{Max}^{ATree,1} = \{3, 2\} \). Table 4.3 describes the marking of the children-set \( \langle q_{PTree}, q_{PTwig} \rangle \) in the example. When Algorithm 7 is modified, then the FSA construction marks two children-sets \( \{q_{\perp}, q_{d_2}^{u}\} \) and \( \{q_{\perp}, q_{d_2}\} \).

<table>
<thead>
<tr>
<th>\langle q_{cT ree}, q_{Selecting} \rangle</th>
<th>children-sets status</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle 3, q_{\perp} \rangle</td>
<td>{q_{\perp}, q_{d_2}, }, {q_{\perp}, q_{d_2}^{u}}</td>
</tr>
<tr>
<td>\langle 2, q_{\perp} \rangle</td>
<td>{q_{\perp}, q_{d_2}, }, {q_{\perp}, q_{d_2}^{u}}</td>
</tr>
<tr>
<td>\langle 2, q_{d_2}^{u} \rangle</td>
<td>{q_{\perp}, q_{d_2}, }, {q_{\perp}, q_{d_2}^{u}}</td>
</tr>
<tr>
<td>\langle 3, q_{d_2} \rangle</td>
<td>{q_{\perp}, q_{d_2}, }, {q_{\perp}, q_{d_2}^{u}}</td>
</tr>
</tbody>
</table>

Table 4.3: The marked children-sets status. The first column describes the input to recursion steps that visit state \( \langle 2, q_{c}\rangle \). The second column describes the status of the children-set. A bold state denotes a marked state.

Due to the form of \( ATwig \), the children-search function can be simplified more. Lemma 4.11 explains why.

**Lemma 4.11** Given a UUTA \( ATwig \) and two states \( q_{v}^{u}, q_{v} \in Q^{ATwig} \). Then, there is a transition \( \delta^{ATwig}(S \cup \{q_{v}\}, a) = q \) if and only if there is transition \( \delta^{ATwig}(S \cup \{q_{v}^{u}\}, a) = q. \)

**Proof** Can be seen from the \( ATwig \) construction in Algorithm 1.

Lemma 4.11 shows that children-sets of a parent state \( q_{vp} \) contains combinations of \( q_{vi} \) and \( q_{vi}^{u}, 1 \leq i \leq n, \) of \( \{v_{1}, \ldots, v_{n}\} = children(T^{Twig}, v_{p}). \) We see that for a parent
state \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \), it is sufficient to match a single children-set \( S \subseteq Q^{A_{T_{\text{wig}}}} \) where \( S = \{q_{v_1}, \ldots, q_{v_n}\}, \{v_1, \ldots, v_n\} = \text{children}(T_{\text{twig}}, v_p) \) and \( q_{\text{PTwig}} = v_p \). If the input to the recursive function is \( \langle q_{\text{Tree}}, q_{\text{PTwig}}^u \rangle \), then the algorithm translates \( q_{v_i}^u \) into \( q_{v_i} \) and marks \( q_{v_i} \) in \( S \). When \( S = \{q_{v_1}, \ldots, q_{v_n}\} \) is entirely marked there is at least one children-set \( S' \subseteq \text{maybe_children}(\langle q_{\text{Tree}}, q_{\text{PTwig}}^u \rangle) \) where \( \gamma_{\text{PTwig}}(\bigcup_{(q_{\text{Tree}}, q_{\text{PTwig}}^u) \in S'} q_{\text{Tree}}, a) = q_{\text{PTwig}} \). \( S' \) is a variation of \( S \) that has one or more \( q_{v_i}^u \) instead of \( q_{v_i} \). Therefore, Algorithm 6 constructs transitions from \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \) to all the states \( \langle q_{\text{Tree}}, q_{v_i} \rangle \) and \( \langle q_{\text{Tree}}, q_{v_i}^u \rangle \) in \( \text{maybe_children}(\langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle) \), where \( q_{v_i} \in S \), and calls the recursive function with the parameters \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \). Each time \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \) is visited by \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \) after the children-set was entirely marked, then a transition from \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \) to \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle \) is added to \( \delta(A_{\text{Tree}} \cap A_{\text{Selecting}})_{pc} \). Due to Lemma 4.11 the algorithm knows that \( q_{\text{PTwig}} \) derives \( q_{\text{PTwig}} \). If, for example, \( q_{\text{PTwig}} = q_{v_i}^u \) and \( S' \subseteq \text{maybe_children}(\langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle) \) contains \( q_{v_i} \), then Lemma 4.11 ensures that the transition \( \delta_{\text{PTwig}}(\bigcup_{(q_{\text{Tree}}, q_{\text{PTwig}}^u) \in S'} q_{\text{Tree}}, a) = q_{\text{PTwig}} \) exists when we replace \( q_{v_i} \in S' \) with \( q_{v_i} \).

Table 4.4 describes the marking of \( \langle q_{\text{Tree}}, q_{\text{PTwig}} \rangle = \langle 2, q_c \rangle \) children-set \( \{q_1, q_2\} \) in the FSA construction in example 4.8. The first two visits mark \( q_\perp \). The third visit translates \( q_{d_2}^u \) into \( q_{d_2} \) and marks it in the children-set. Now the set \( \{q_\perp, q_{d_2}\} \) is entirely marked and the set \( S' = \{q_\perp, q_{d_2}^u\} \) exists in \( \text{maybe_children}(\langle 2, q_c \rangle) \). The transitions from \( \langle 2, q_c \rangle \) to \( \langle 3, q_\perp \rangle \), \( \langle 2, q_\perp \rangle \) and \( \langle 2, q_{d_2}^u \rangle \) are added to \( \delta(A_{\text{Tree}} \cap A_{\text{PTwig}})_{pc} \). In the fourth visit, the algorithm does not translate the symbol because the set \( \{q_\perp, q_{d_2}\} \) is entirely marked. Therefore, due to Lemma 4.11, the transition from \( \langle 2, q_c \rangle \) to \( \langle 2, q_{d_2} \rangle \) is added to \( \delta(A_{\text{Tree}} \cap A_{\text{PTwig}})_{pc} \).

<table>
<thead>
<tr>
<th>( \langle q_{\text{Tree}}, q_{\text{Selecting}} \rangle )</th>
<th>Translation</th>
<th>children-set status</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle 3, q_\perp \rangle )</td>
<td>( q_\perp )</td>
<td>( {q_\perp, q_{d_2}} )</td>
</tr>
<tr>
<td>( \langle 2, q_\perp \rangle )</td>
<td>( q_\perp )</td>
<td>( {q_\perp, q_{d_2}} )</td>
</tr>
<tr>
<td>( \langle 2, q_{d_2}^u \rangle )</td>
<td>( q_{d_2} )</td>
<td>( {q_\perp, q_{d_2}} )</td>
</tr>
<tr>
<td>( \langle 3, q_{d_2} \rangle )</td>
<td>( q_{d_2} )</td>
<td>( {q_\perp, q_{d_2}} )</td>
</tr>
</tbody>
</table>

Table 4.4: The marked children-set status. The column \( \langle q_{\text{Tree}}, q_{\text{Selecting}} \rangle \) describes the input to the recursion steps that visit state \( \langle 2, q_c \rangle \). The children-set-status column describes the status of the children-set. A bold state denotes a marked state.

Part II Querying of XML in a DB
4.5.2 \((A_{\text{Tree}} \cap A_{\text{Selecting}})_{pc}\) construction time analysis

The recursive function \(CreateTransitions\) in Algorithm 6 is called utmost once for every input parameters \(q_{ctree} \in Q_{\text{Tree}}^{pc}\) and \(q_{cselecting} \in Q_{\text{Selecting}}^{pc}\). In the worse case, the algorithm checks all the combinations of the transitions \(\delta_{\text{pc}}^{T_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}\) and \(\delta_{\text{pc}}^{A_{\text{selecting}}}(q_{p_{\text{selecting}}}, a) = q_{cselecting}\). Therefore, we can bound the operation of Algorithm 6 by \(\sum_{\delta_{\text{pc}}^{A_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}} \sum_{\delta_{\text{pc}}^{A_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}} \text{children-search}(\langle q_{p_{\text{tree}}}, q_{p_{\text{tree}}}, a \rangle, \text{maybe-children})\). In section 4.5.1 we simplified the \(\text{children-search}(\langle q_{p_{\text{tree}}}, q_{p_{\text{tree}}}, a \rangle, \text{maybe-children})\) operation. If \(A_{\text{Selecting}} = A_{\text{Tree}}\) and \(A_{\text{Tree}} = A_{S_{\text{Max}}}\) then the algorithm performs \(\text{children-search}(\langle q_{p_{\text{tree}}}, q_{p_{\text{tree}}}, a \rangle, \text{maybe-children})\) in \(O(1)\) time.

The holistic-join [128] and holistic-index [127] preform the STA\((A_{\text{Tree}})\) operation on \(A_{\text{Tree}} = A_{S_{\text{Max}}}\). In this case, the time is bounded by \(\sum_{\delta_{\text{pc}}^{A_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}} \sum_{\delta_{\text{pc}}^{A_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}} O(1)\).

Given the symbol \(a \in \Sigma\), the number of transitions \(\delta_{\text{pc}}^{A_{\text{tree}}}(q_{p_{\text{tree}}}, a) = q_{ctree}\) is bounded by the size of \(V_{T_{\text{tree}}}^{\text{pc}}\) that is denoted by \(|V_{T_{\text{tree}}}^{\text{pc}}|\). The time worse case is when all the twig nodes have the same label \(a \in \Sigma\) and have A-D node relations with each other. In this case, each \((v_p, v_c) \in E_{T_{\text{tree}}}^{\text{pc}}\) contributes three transitions to \(\delta_{\text{pc}}^{A_{\text{tree}}}: 1. \delta_{\text{pc}}^{A_{\text{tree}}}(q_{v_p}, q_{v_c}) = q_{v_c}; 2. \delta_{\text{pc}}^{A_{\text{tree}}}(q_{v_p}, q_{v_c}) = q_{v_c}; 3. \delta_{\text{pc}}^{A_{\text{tree}}}(q_{v_p}, q_{v_c}) = q_{v_c}.\)

Since \(T_{\text{tree}}^{\text{twig}}\) is a tree then the number of edges is equal to the number of nodes. Therefore, there are maximum \(3 \cdot |V_{T_{\text{tree}}}^{\text{pc}}|\) transitions with the label \(a\) in \(A_{\text{pc}}^{\text{tree}}\). Given a FSA \(A\), we denote by \(|A|\) the number of transitions in \(\delta_{\text{a}}\). From all the above we can bound the time of \((A_{\text{Tree}} \cap A_{\text{twig}})_{pc}\) construction in Algorithm 6 to be \(O(|A_{\text{pc}}^{\text{tree}}| \times |V_{T_{\text{tree}}}^{\text{pc}}|)\). The \((A_{\text{Tree}} \cap A_{\text{twig}})_{pc}\) construction is the dominant operation in terms of time in the \(STA_{\Sigma}^{\text{twig}}(A_{\text{Tree}})\) operation. Therefore, we can bound the time consumption of \(STA_{\Sigma}^{\text{twig}}(A_{\text{Tree}})\) operation to be \(O(|A_{\text{pc}}^{\text{tree}}| \times |V_{T_{\text{tree}}}^{\text{pc}}|)\).

4.6 Conclusion

In this chapter, we define the UUTA tree automaton that recognizes unordered trees. We show how to construct a twig-UUTA from a twig pattern and how to query an unordered tree, which represents an XML document, by this twig-UUTA. In addition, this chapter shows how to query a UUTA, which recognizes a collection of potential XML documents, by a twig-UUTA. The described tree automata theory enables to query XML documents, by a twig-UUTA.
efficiently XML documents, which are stored in a DB. The presented theory is the basis for the implementation of the holistic structural join in Chapter 5 and the holistic structural index in Chapter 6.
Chapter 5

Holistic structural-join

5.1 Introduction

TwigStack \[106\] is considered to be the state-of-the-art algorithm for twig pattern matching of XML documents that are stored in RDBMS. TwigStack technique uses intermediate storage of from-root-to-leaf paths, which are then composed to obtain a match for a twig pattern. The core of the TwigStack algorithm is the TwigJoin algorithm which selects TwigStack intermediate paths. Many recent papers have investigated the TwigJoin algorithm.

This chapter presents the TwigTA algorithm that matches a twig pattern in XML document. TwigTA bases its operation on tree automata methodology in chapter 4. TwigTA enhances the accuracy of intermediate paths selection of the TwigJoin. TwigTA extends TwigStack twofold:

1. The number of extracted nodes in each iteration: TwigTA is not limited to a fixed number of nodes from the external memory at each iteration. TwigTA can extract any number of nodes from the RDBMS at each iteration. When TwigTA extracts more symbols, its intermediate paths selection becomes more accurate.

2. Matching of twig patterns with parent-child (P-C) nodes relationships: TwigTA outperforms TwigStack accuracy in selecting intermediate paths on a twig pattern that has P-C nodes relationships. TwigTA models the exact twig pattern as a tree automata whereas TwigStack replaces all P-C nodes relationships with ancestor-descendant (A-D) nodes relationships. Due to its accuracy, TwigTA constructs less intermediate paths than TwigStack.
Unlike TwigStack, TwigTA can predict nodes that are not yet extracted from an external memory and are not part of any solution of the twig pattern. TwigTA can avoid fetching these nodes from the external memory. This makes TwigTA more efficient while saving CPU resources.

5.1.1 TwigTA basic operation

TwigTA uses the tree automata methodology, which is presented in Chapter 4, in order to select XML tree nodes that match a twig pattern. The TwigTA algorithm uses TwigStack streams data model (see section 3.1.1). Due to the large number of nodes in the streams, TwigTA cannot extract all the nodes to reconstruct the complete XML tree. Instead, the TwigTA join operation iteratively extracts a fixed number of nodes from the streams and constructs a tree automaton \( A^{Tree} \) that accepts the complete XML tree structure. Figure 5.1 details the flow of the TwigTA join operation. Initially, the Algorithm constructs a STA \((A^{Twig}, S)\) from the input twig pattern. In each iteration the TwigTA algorithm extracts a fixed number of XML tree nodes from the streams. Then, it constructs a partition that maps each node of the XML tree to a state in \( A^{Tree} \). Next, the TwigTA algorithm constructs a prediction-tree from this partition. The prediction-tree is not a single tree. It expresses a collection of trees where the stored XML document tree is one of them. The TwigTA constructs a tree automaton \( A^{Tree} \) which accepts the collection of trees that the prediction-tree expresses. Next, TwigTA performs \( STA_{S^{Twig}} (A^{Tree}) \) operation, which is described in Chapter 4, and selects extracted XML document nodes which are part of any twig pattern solution. Extracted XML document nodes, which are not part of a twig pattern solution, can safely be removed. If none of the extracted nodes is removed then the TwigTA constructs intermediate paths from the extracted XML document nodes and removes these extracted nodes. After all the intermediate paths are constructed, the TwigTA join operation sorts the intermediate paths and merges them into solutions for the twig pattern. The TwigTA sort-merge operation is identical to the TwigStack sort-merge operation.

Part II Querying of XML in a DB
Figure 5.1: General flow of the TwigTA algorithm. The white boxes denote the operations of the TwigTA algorithm that are presented in this thesis. The grey boxes denote operations that are presented in Chapter 4.

The chapter has the following structure. Section 5.2 describes related work. Section 5.3 describes STA(T) based join operation. Section 5.4 describes the TwigTA algorithm and experimental results are presented in section 5.5. Theory and tools for the constructions, which are needed to process the TwigTA algorithm, are given in Chapter 4.

Part II: Querying of XML in a DB
5.2 Related work: twig join algorithms

This section describes the existing twig join algorithms.

In the last few years, many join algorithms ([106, 138, 146, 77, 76]) were suggested to join XML data that is encoded by a labeling scheme. In order to answer a twig pattern, these algorithms access the labels only without traversing the original XML documents.

Structural join techniques [121] propose to decompose a twig pattern into a set of binary nodes relationships which are either P-C or A-D. After that, each binary node relationship is joined separately. The final twig pattern solutions are obtained by ‘stitching’ together individual binary join results. The main problem with structural join is that it may generate large and possibly unnecessary intermediate paths because the join results of individual binary nodes relationships may not appear in the twig pattern solution. In recent years, research activities have focused on finding an algorithm which does not produce unnecessary intermediate paths while performing a single traverse on the DB streams. We call this algorithm an optimal pattern matching algorithm.

TwigStack [106] was suggested to reduce the number of unnecessary intermediate paths. The TwigStack algorithm receives a twig pattern $Q$ and an XML DB $D^T$. It generalizes the twig pattern $Q$ into $Q'$ by replacing all the P–C nodes relationships with A–D nodes relationships. It associates each node in the twig pattern with a stack. The algorithm operates in two main phases: (1) TwigJoin. The algorithm reads the region encoded labels of nodes from $D^T$ and constructs paths of labels of intermediate results. Each path is a solution for one root-to-leaf path of the twig pattern. (2) Merge. The intermediate paths, which were constructed in step 1, are merged to solutions for the twig pattern. When all the edges in the twig pattern is of type A-D, TwigStack is optimal. In this case, all the constructed intermediate paths compose solutions for the twig pattern. However, TwigStack is sub-optimal if there are P-C nodes relationships in the twig pattern. In other words, the method may still generate redundant intermediate paths in the presence of P-C nodes relationships in the twig pattern.

Several algorithms ([138, 146, 77, 76, 85]) were introduced to handle this sub-optimality. The TwigTA algorithm handles this sub-optimality in a different way. The TwigTA join algorithm can be viewed as an extension of the TwigStack join algorithm. Like the TwigStack, it operates in two main phases. The first phase constructs the same type of intermediate paths that TwigStack constructs and the second phase
performs the same Merge operation as TwigStack. TwigTA extends the TwigStack operation by querying the input twig pattern $Q$ without generalizing $Q$ into $Q'$. In this way, the intermediate paths selection in the first phase becomes more accurate. The TwigTA algorithm also extends TwigJoin by the number of nodes it extracts from streams in each iteration. The number of extracted nodes determines the accuracy of the intermediate paths selection. TwigStack extracts one node for each stream in every iteration. TwigTA extracts $K$ nodes from each stream where $K$ is configurable.

The TwigTA can be adopted to cowork with many of the algorithms that have been proposed to reduce the number of unnecessary intermediate paths in TwigStack. These algorithm optimize several aspects of the TwigStack: caching, generalization of the tree patterns, traversal order, indexing, labeling scheme and the number of traversals. TwigStackList ([85]) caches a limited number of nodes in lookahead lists in order to filter unnecessary intermediate results. TwigTA performs a similar operation when the number $K$ of extracted nodes increases by each stream. Extensions for handling generalized tree patterns were suggested by [39, 114, 76, 117]. TwigTA models the twig pattern as an automaton and, therefore, supports generalized twig pattern. Twig$^2$Stack [39] suggested using labels that are encoded in a bottom-up traverse order. TwigTA algorithm constructs a tree automaton of the labeled nodes. The constructed automaton is independent of the traversal order. Therefore, TwigTA can be adopted to handle labels in a bottom-up traverse order.

An important direction in XML query processing research is the construction of structural-indexes from XML documents to avoid unnecessary scanning of source documents. Several algorithms combine indexes with the TwigStack operation [138, 77, 145, 97]. iTwigJoin [138] is a general twig join algorithm, that combines a structural-index. The structural-index enables the iTwigJoin to reduce I/O cost by filtering irrelevant labels from the streams. iTwigJoin suggests a structural-index that is called a prefix path streaming (PPS). The PPS scheme is a special case of Dataguide structural-index ([119]). Like Dataguide, iTwigJoin optimizes the query processing of simple queries while degrading the performance of more complex queries. We conduct experiments that compare between TwigTA and iTwigJoin processing on a tree-structured XML data. Like any other join mechanism, TwigTA can be combined with a structural-indexing technique.

Several algorithms modify the labeling scheme [146, 118, 96]. BLAS ([146]) pro-
posed a dual labeling scheme: D-Label which is exactly the same as region encoding and P-Label for accelerating the P-C nodes relationship processing. This method decomposes a twig pattern into several P-C only path patterns and then the results are joined. Translation of XML nodes into a sequence of labels by Prufers method is proposed in [118]. Translation of the region encoding into Dewey numbers is suggested in [96]. TwigTA can be adopted to construct automaton that predicts the structure of a stored document which is encoded in various labeling schemes.

TwigStack and TwigTA perform a single traverse on the XML document, which is stored in a DB $D^T$ and maintain memory, which is bounded by the height of the XML tree. Other methods perform either additional traversals on $D^T$ ([117]) or consume internal memory of $D^T$ size ([39]) in order to construct solutions for the twig pattern without producing intermediate paths. Algorithm [117] is based on the $STA(T)$ technique ([43]) which is the basis for the $STA^A_{S}^{Twig}(T)$ operation in Chapter 4. The $STA^A_{S}^{Twig}(T)$ is adopted in section 5.3 to perform a twig pattern matching in two traverses of $D^T$ without constructing intermediate paths. Algorithm [117] models $D^T$ as a tree and performs multiple traverses on $D^T$. Modeling parts of $D^T$ as tree automata enables TwigTA to perform a single traversal on $D^T$ while keeping the memory size bounded to the height of $D^T$.

5.3 $STA^A_{S}^{Twig}(T)$ based join

Before we detail the TwigTA algorithm we describe a join algorithm that adopts the $STA^A_{S}^{Twig}(T)$ operation (Definition 4.3) for $D^T$, which is described in Section 3.1.1, in a straight-forward manner. Both this algorithm and the TwigTA use the following notations to describe the a solution ($Solution^T$).

$Path^T$: Given $D^T$ and $Path^T$, which is the sequence $R_{v_1}, \ldots, R_{v_n}$ of encoded nodes that match a path in $T$. For each $1 \leq i \leq n - 1$, $ancestor(R_{v_i}, start_{v_{i+1}}) = 1$.

$Solution^T$: Given $D^T$ and $A^{Twig} Q$. The $Solution^T: F^{A^{PC}_{PC}} \mapsto Nodes^T$ is a one-to-one function that maps each selecting state $q \in F^{A^{PC}_{PC}}$ to a label $R_v$ in $D^T$. $Solution^T$ is a match of $Q$ in $D^T$.

Example 5.1 demonstrates these notations.
Example 5.1 The XML tree $T$ in Fig. 3.2 illustrates an XML region encoding. The sequence $((0, 29, 1), (2, 5, 3), (3, 4, 4))$ is a Path$^T$. A Solution$^T_s$ for the ATwig in Example 4.2 and the tree $T$ in Fig. 3.2 have the following structure: $s(q_a) = (0, 29, 1)$, $s(q_b) = (19, 28, 2)$, $s(q_c) = (3, 4, 4)$, $s(q_d) = (20, 21, 3)$, $s(q_e) = (24, 25, 5)$. 

The join algorithm, which is described in this section, performs two traverses on $D^T$: bottom-up and top-down. This algorithm uses $O(|D^T|)$ memory because it stores in memory the $\rho_{ATwig}(T)$, which in size $|D^T| = |V^T|$, in between the bottom-up and top-down traverses. In this sense it can be viewed as a “primitive” version of TwigTA that uses a constant memory and performs a single traverse on $D^T$. On the other hand, the algorithm in this section does not produce intermediate solutions at all. In this sense it is more “sophisticated” then TwigTA.

The STAs$^{ATwig}_S(T)$ has two phases: 1. The $ATwig(T)$ operation applies $UUTA$ $ATwig$ bottom-up to a tree $T$; 2. The top-down application of $FSA$ $ATwig$ to $\rho_{ATwig}(T)$ and $T$. The join algorithm also operates in two phases. Each phase performs a single traverse on $D^T$. Algorithm 8 describes the first phase of the join algorithm. The algorithm adopts the $ATwig(T)$ operation (Definition 4.2) to be applied to $D^T$. We denote this operation by $ATwig(D^T)$. The algorithm converts the tree automaton $ATwig$ into a string automaton by adding the parentNodes and childrenStates stacks. Given stack $S$, the algorithm applies the operations: $push(S, e)$, $pop(S)$, and $top(S)$. $push(S, e)$ inserts element $e$ to stack $s$. $pop(S)$ removes and returns element $e$ from the top of stack $S$ while $top(S)$ just returns the element $e$. Given a node $v_p$, instead of applying the transition $\delta_{ATwig}$ to $\cup_{v_c \in \text{children}(T, v_p)} \rho_{ATwig}(T)(v_c)$ at once, the algorithm gathers $\rho_{ATwig}(D^T)(v_c)$ into childrenStates during the traversal. When the node $v_p$ is popped from the parentNodes stack, the algorithm applies the transition $\delta_{ATwig}$ to childrenStates.
Algorithm 8: $A^{T_{wig}}(D^T)$ - the join bottom-up traverse

**Input:** $D^T$, $UUT\ A^{T_{wig}}$

**Output:** $\rho^{A^{T_{wig}}(D^T)}$

**Data:** Stack $parentNodes$, Stack $childrenStates$

1. begin
2. \hspace{1em} while $D^T \neq \emptyset$ do
3. \hspace{2em} $R_{v_c} \leftarrow \text{min\_future\_node}(D^T)$;
4. \hspace{2em} $R_{v_p} \leftarrow \text{top}(parentNodes)$;
5. \hspace{2em} if $\text{parent}(R_{v_p}, R_{v_c}) = 1$ then
6. \hspace{3em} $\text{push}(parentNodes, R_{v_c}); \text{push}(childrenStates, \emptyset)$;
7. \hspace{3em} $D^T(label^T(v_c)) \leftarrow \text{tail}(D^T(label^T(v_c)))$;
8. \hspace{2em} else
9. \hspace{3em} $S_{v_p}^{\text{children}} \leftarrow \text{pop}(childrenStates); \text{pop}(parentNodes)$;
10. \hspace{3em} $R_{v_p} \leftarrow \text{top}(parentNodes)$;
11. \hspace{3em} for all transitions $\delta^{A^{T_{wig}}}(S, label^T(v_p)) = q_p$ where $S \subseteq S_{v_p}^{\text{children}}$ do
12. \hspace{4em} Add $q_p$ to $\rho^{A^{T_{wig}}(D^T)}(v_p)$;
13. \hspace{4em} Add $q_p$ to top(childrenStates);
14. end

**Example 5.2** Example 4.2 described the $A^{T_{wig}}(T)$ operation. $D^T$, which stores $T$, was described in Example 3.1. Table 5.1 describes Algorithm 8 application of the $A^{T_{wig}}$ to the DB $D^T$. When we compare between the $A^{T_{wig}}(T)$ output in Example 4.2 and the $A^{T_{wig}}(D^T)$ output in Table 5.1 we see that $\rho^{A^{T_{wig}}(T)} = \rho^{A^{T_{wig}}(D^T)}$.  \hfill \Box
Holistic structural-join 115

Table 5.1: A flow of Algorithm $A^{T\text{wig}}(D_T)$ operation. Each row describes the data after the node $R_v$ was pushed to $\text{parentNodes}$. The $\text{start}_v$ column identifies the $R_v$. The ($\text{parentNodes}, \text{childrenStates}$) column describes the stacks content. Each pair $(\text{start}_v, \{q_1, \ldots, q_n\})$ describes two entries $R_v \in \text{parentNodes}$ and $\{q_1, \ldots, q_n\} \in \text{childrenStates}$ at the same position. The $(\text{start}_v, \rho^{A^{T\text{wig}}(D_T)}(v))$ column describes $\rho^{A^{T\text{wig}}(D_T)}$. Each pair $(\text{start}_v, \{q_1, \ldots, q_n\})$ describes $\rho^{A^{T\text{wig}}(D_T)}(v) = \{q_1, \ldots, q_n\}$ that was added since the last push of a node to the $\text{nodes}$ stack.

The top-down phase of the join algorithm, which is described in Algorithm 9, extends Algorithm 4. Algorithm 9 maintains the $\text{nodes}$ stack, the $\text{solutions}$ stack and the sorted list $L$ of solutions. Each entry in the $\text{solutions}$ stack contains a list of solutions references in the format $j_{qp}^k$ where $j, k$ are references to solutions in $L$. The states $q_p \in Q_{\text{wig}}$ and $q_a \in Q_{\text{wig}}$ derive nodes in solutions $L_j$ and $L_k$, respectively. A reference part $j_{qp}$ is used twice during the application of $A_{\text{wig}}$ to $D_T$ and $\rho^{A^{T\text{wig}}(D_T)}$. The first time the reference part $j_{qp}$ is used is when $R_v \in D_T$ and $R_v \in \text{nodes}$ satisfy $\text{parent}(R_{vp}, R_v) = 1$. The algorithm checks the reference part $j_{qp}$ in a reference $j_{q_{vp}}$ of $v_p$. If there is a transition $\delta A_{\text{wig}}$ from $q_p$ to $q_c \in \rho^{A^{T\text{wig}}(D_T)}(v_c)$ then $L_j$ is updated as follows. If $q_c$ is a selecting state then $R_v$ is added to the solution $L_j$ and $L_j(q_c) = R_v$.

Part II Querying of XML in a DB

<table>
<thead>
<tr>
<th>$\text{start}_v$</th>
<th>($\text{parentNodes}, \text{childrenStates}$)</th>
<th>$(\text{start}_v, \rho^{A^{T\text{wig}}(D_T)}(v))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(-\infty, \emptyset), (0, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$(-\infty, \emptyset), (0, \emptyset), (1, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$(-\infty, \emptyset), (0, \emptyset), (1, \emptyset), (2, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$(-\infty, \emptyset), (0, \emptyset), (1, \emptyset), (2, \emptyset), (3, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset)$</td>
<td>$(3, {q_c}), (2, {q_c^u}), (1, {q_c^u})$</td>
</tr>
<tr>
<td>8</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset), (8, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset), (8, \emptyset), (9, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset), (8, {q_c}), (11, \emptyset)$</td>
<td>$(9, {q_c})$</td>
</tr>
<tr>
<td>12</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset), (8, {q_c}), (11, \emptyset), (12, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>$(-\infty, \emptyset), (0, {q_a^u}), (7, \emptyset), (8, {q_c}), (11, {q_d}), (14, \emptyset)$</td>
<td>$(12, {q_d})$</td>
</tr>
<tr>
<td>19</td>
<td>$(-\infty, \emptyset), (0, {q_a^u, q_c^u, q_c^u}), (19, \emptyset)$</td>
<td>$(14, {q_c}), (11, {q_b, q_c^u}), (8, {q_c^u, q_c^u, q_c^u}), (7, {q_a^u, q_c^u, q_c^u})$</td>
</tr>
<tr>
<td>20</td>
<td>$(-\infty, \emptyset), (0, {q_a^u, q_c^u, q_c^u}), (19, \emptyset), (20, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>$(-\infty, \emptyset), (0, {q_a^u, q_c^u, q_c^u}), (20, {q_a^u}), (22, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>$(-\infty, \emptyset), (0, {q_a^u, q_c^u, q_c^u}), (20, {q_a^u}), (22, \emptyset), (23, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>$(-\infty, \emptyset), (0, {q_a^u, q_c^u, q_c^u}), (20, {q_a^u}), (22, \emptyset), (23, \emptyset), (24, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>$\infty$</td>
<td>$(-\infty, {q_a, q_b^u, q_c^u, q_c^u}), (\infty, \emptyset)$</td>
<td>$(24, {q_c}), (23, {q_c^u}), (22, {q_c^u}), (20, {q_b, q_c^u}), (0, {q_a, q_b^u, q_c^u, q_c^u})$</td>
</tr>
</tbody>
</table>
But if \( L_j(q_c) \) was already added to \( L_j \) then a new solution \( L_i \) is added to the list and \( L_i(q_c) = R_{v_c} \). In both cases, the references \( i_{q_c}^{j_{q_p}} \) and \( j_{q_p}^{i_{q_c}} \) are added to the solutions stack and node \( R_{v_c} \) is added to nodes. The second time the reference part \( j_{q_p} \) is used is when \( parent(R_v, R_{v_c}) = 0 \) and node \( R_{v_c} \) is popped from nodes. The references \( i_{q_c}^{j_{q_p}} \) and \( j_{q_p}^{i_{q_c}} \) of \( R_{v_c} \) are also popped from the solutions. Here, the algorithm uses the reference part \( j_{q_p} \) in order to update the parent node \( R_{v_p} \) in top(nodes) with references \( L_{v_p} \) that were created in \( v_c \) subtree. The algorithm uses \( j_{q_p} \) to backtrack the reference \( j_{q_p}^{i_{q_c}} \) of node \( R_{v_p} \) that originated \( j_{q_p}^{i_{q_c}} \). The update adds the reference \( i_{q_c}^{j_{q_p}} \) to the solutions of \( R_{v_p} \).

Algorithm 9 uses the following functions:

1. **all-but-subtree** \( : Solutions^T \times Q^{AT_w} \rightarrow Solutions^T \). Given \( Solutions^T \)
   \[ S \triangleq \{(q_{v_1}^{T_w}, v_1^T), \ldots, (q_{v_n}^{T_w}, v_n^T)\} \] and state \( q_{v_i}^{T_w} \in Q^{AT_w} \) all-but-subtree\( (S, q_{v_i}^{T_w}) \subseteq S \). If \( v_i^{T_w} \) is not an ancestor of \( v_j^{T_w} \) in \( T^{T_w} \) then \( (q_{v_i}^{T_w}, v_i^T) \in all-but-subtree(S, q_{v_i}^{T_w}) \).

2. **full** \( : Solutions^T \rightarrow \{0, 1\} \). Given \( Solutions^T \ S \), \( full(S) = 1 \) if every selecting state \( q_{v_i}^{T_w} \) satisfies \( (q_{v_i}^{T_w}, v_i^T) \in S \). Otherwise \( full(S) = 0 \).

For example, the solution \( S \triangleq \{(q_a, (0, 29, 1)), (q_b, (19, 28, 2)), (q_c, (3, 4, 4)), (q_d, (20, 21, 3)), (q_e, (24, 25, 5))\} \) which matches the twig pattern ‘//a[//c]/b[//e]/d’ on \( D^T \) in Example 4.2 satisfies \( full(S) = 1 \) because every selecting state \( q \in F^{AT_w} = \{q_a, q_b, q_c, q_d, q_e\} \) is mapped in \( S \). On the other hand, \( full(all-but-subtree(S, q_b)) = 0 \) because the mappings \( q_c \) and \( q_d \) are removed from all-but-subtree\( (S, q_b) = \{(q_a, (0, 29, 1)), (q_b, (19, 28, 2)), (q_c, (3, 4, 4))\} \).
Algorithm 9: The top-down traverse of the STA($T$)-based join.

Input: $DT$, $\rho_{A^T}A_{pc}$, $FSA A_{pc}^T$
Output: sorted list of Solution$^T$: $L$

Data: Stack : nodes, Stack : solutions where each entry contains solution references in the format $i_{q_p}$

begin

$L_0 \leftarrow (R_{\infty}, \ldots)$;
$push(\text{nodes}, R_{\infty}); push(\text{solutions}, \{i_{q_i}\})$ where $i = 0$;

while $DT \neq \emptyset$ do

$R_{vc} \leftarrow \text{min\_future\_node}(a); DT(\text{label}(v_c)) \leftarrow \text{tail}(DT(\text{label}(v_c)));$

while parent($R_{vp}, R_{vc}$) = 0 where $R_{vp} = \text{pop}(\text{nodes})$ do

forall $i_{q_c} \in \text{pop(\text{solutions})}$ do

if full($L_i$) = 0 then

Add $i_{k_{qa}q_p}$ to top(solutions) where $j_{k_{qa}q_p} \in \text{top(solutions)}$;

end

Let solutions$_c \leftarrow \emptyset$;

forall $j_{k_{qa}q_p} \in \text{top(solutions)}$ do

forall $\delta_{A_{pc}^T}(q_p, q_c) \in \rho_{A^T}(DT(v_c))$ do

Let $i \leftarrow j$;

if $q_c \in F_{A_{pc}^T}$ then

if exists $L_j(q_c)$ then

$i \leftarrow |L|$;
$L_i \leftarrow \text{all-but-subtree}(L_j, q_c)$;
Add $L_i$ to list;
$L_i(q_c) \leftarrow R_{vc}$;

if full($L_i$) = 0 then

Add $i_{q_p}$ to solutions$_c$;

end

push(\text{nodes}, R_{vc}); push(\text{solutions}, \text{solutions}_c);

end

Example 5.3 Table 5.2 describes Algorithm 9 application of the $A_{pc}^T$, which was described in Fig. 5.9(b) to $DT$, which was described in Example 4.2 and $\rho_{A^T}(DT)$ in Ex-
The solutions $L_1$ and $L_2$ are added to the list because $L_0(q_a) = (-\infty, \infty, 0)$. The solution $L_3$ is added to the list because $L_1(q_c) = (3, 4, 4)$. 

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
start$_v$ & nodes & solutions & $L$ \\
\hline
$-\infty$ & 0 & \(-\infty, -\infty, -\infty, -\infty, -\infty\)$_0$ & 0 \\
0 & $-\infty, 0$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
1 & $-\infty, 0, 1$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
2 & $-\infty, 0, 1, 2$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
3 & $-\infty, 0, 1, 2, 3$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
4 & $-\infty, 0, 7, 8$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
5 & $-\infty, 0, 7, 8, 9$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
6 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
7 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
8 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
9 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
10 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
11 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
12 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
13 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
14 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
15 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
16 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
17 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
18 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
19 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
20 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
21 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
22 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
23 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
24 & $-\infty, 0, 7, 8, 9, 10$ & \(1_{\top}^{\top}, 0_{\top}^{\top})_0, 0_{\top}^{\top})_1 & (0, 0, \ldots, 1) \\
\hline
\end{tabular}
\caption{Description of Algorithm $A_{pc}^{\mathcal{T}w}$ application of the $D^T$, which was described in Fig. 5.9b), to $D^T$, which was described in Example 4.2} Each row describes the data after a push of $R_v$ to nodes. The start$_v$ column identifies $R_v$. The nodes column describes the content of the nodes stack. The solutions column describes the top(solutions). L describes the sorted list. For compactness of representation, a solution in $L$ is denoted by the five tuple $(\text{start}_v, \ldots, \text{start}_v)$ which identify solution $\{(q_a, R_{v_1}), (q_b, R_{v_2}), (q_c, R_{v_3}), (q_d, R_{v_4}), (q_e, R_{v_5})\}$.

### 5.4 The TwigTA Algorithm

This section has the following structure. Section 5.4.1 extends the STA operations for the TwigTA algorithm. Section 5.4.2 describes the TwigTA algorithm.

#### 5.4.1 Extending the STA operations for the TwigTA algorithm

This section has the following structure. Section 5.4.1.1 extends the $A_{Tree}(T)$ operation with a partition $P$. Section 5.4.1.2 extends the $STA_S^{Selecting}(A_{Tree})$ operation by using the function history.
5.4.1.1 $A^{Tree}(T,P)$ operation

$TwigTA$ iterates over nodes of an XML tree $T$ that are stored in a DB $D^T$. In each iteration, the $TwigTA$ algorithm constructs a $UUTA$ $A^{Tree}$ and a partition $P : V^T \mapsto Q^{A^{Tree}}$ from a set of extracted nodes. Definition 5.1 constrains the run of $UUTA$ $A^{Tree}$ on a tree $T$ with partition $P : V^T \mapsto Q^{A^{Tree}}$. The partition $P$ reflects the knowledge of the $TwigTA$ algorithm on the run $A^{Tree}(T)$.

**Definition 5.1** Given a tree $T$, a $UUTA$ $A^{Tree}$ and a partition $P : V^T \mapsto Q^{A^{Tree}}$. An application (run) of $A^{Tree}$ on the tree $T$ and on the partition $P$ is defined as $A^{Tree}(T,P) \triangleq A^{Tree} \cdot (T,P) \rightarrow \rho^{A^{Tree}(T,P)}$. The output of the run is the map $\rho^{A^{Tree}(T,P)} : V^T \mapsto Q^{A^{Tree}}$. A node $v \in V^T$ satisfies $\rho^{A^{Tree}(T,P)}(v) = \begin{cases} P(v), & \text{if } P(v) \in \rho^{A^{Tree}(T)}(v) \\ \text{not exist}, & \text{otherwise} \end{cases}$. The run $A^{Tree}(T,P)$ is called accepting if $\rho^{A^{Tree}(T,P)}(\root(T)) \in F^{A^{Tree}}$. The automaton $A^{Tree}$ accepts $T$ and the partition $P$ if there is an accepting run $A^{Tree}(T,P)$.

From Definition 5.1 we get Lemma 5.1.

**Lemma 5.1** Given a tree $T$, $UUTA$ $A^{Tree}$ and a partition $P : V^T \mapsto Q^{A^{Tree}}$. Every $v \in V^T$ satisfies $\rho^{A^{Tree}(T,P)}(v) \in \rho^{A^{Tree}(T)}(v)$.

**Proof** If $\rho^{A^{Tree}(T,P)}(v) = P(v)$ then according to Definition 5.1 $P(v) \in \rho^{A^{Tree}(T)}(v)$.\]

**Example 5.4** $A^{Tree} = (Q^{A^{Tree}}, \Sigma, F^{A^{Tree}}, \delta^{A^{Tree}})$ where $\Sigma = \{a,b,c\}$, $Q^{A^{Tree}} = \{q_1,q_2,q_3\}$, $F^{A^{Tree}} = \{q_1\}$, $\delta^{A^{Tree}}(\emptyset,b) = q_3$, $\delta^{A^{Tree}}(\emptyset,c) = q_3$, $\delta^{A^{Tree}}(\emptyset,b) = q_2$, $\delta^{A^{Tree}}(\emptyset,c) = q_2$, $\delta^{A^{Tree}}(\{q_2,q_3\},a) = q_1$. Figure 5.2(a) illustrates $\rho^{A^{Tree}(T)}$. Given the partition $P = \{(1,q_1),(2,q_2),(3,q_3)\}$, Fig. 5.2(b) illustrates $\rho^{A^{Tree}(T,P)}$. Both runs $A^{Tree}(T)$ and $A^{Tree}(T,P)$ are accepting. We see that Lemma 5.1 is satisfied. For example, $\rho^{A^{Tree}(T,P)}(2) = q_2 \in \rho^{A^{Tree}(T)}(2)$.
Figure 5.2: (a) and (b) describe $\rho^{A_{Tree}(T)}$ and $\rho^{A_{Tree}(T,P)}$, respectively. A tree node $v$ is denoted by a circle. The label of node $v$ has two lines. The top line in each circle has the syntax ‘$v; \text{label}^T(v)$’. The bottom line in each circle has the format ‘$q_1, \ldots, q_n$’ where $q_1, \ldots, q_n = \rho^{A_{Tree}(T)}(v)$ in (a) and $q_1, \ldots, q_n = \rho^{A_{Tree}(T,P)}(v)$ in (b).

Definition 5.2 constraint the $STA^T_{A_{Tree}}(T)$ operation with a partition $P$.

Definition 5.2 Given a tree $T$, a STA $(A_{Tree}, S)$, and a partition $P : V^T \mapsto Q^{A_{Tree}}$. The application of the STA $(A_{Tree}, S)$ to tree $T$ and $P$ is defined as $STA^T_{A_{Tree}}(T, P) \triangleq (A_{Tree}, S) \cdot (T, P) \mapsto \rho^{STA^T_{A_{Tree}}(T,P)}$. The output of the $STA^T_{A_{Tree}}(T, P)$ operation $\rho^{STA^T_{A_{Tree}}(T,P)} : V^T \mapsto S$ maps the selected nodes $v \in V^T$ to states in $S$ that derive an accepting run $\rho^{A_{Tree}(T,P)}$. The $STA^T_{A_{Tree}}(T, P)$ operation has two phases: 1. The bottom-up traversal of $A_{Tree}(T, P)$ that outputs $\rho^{A_{Tree}(T,P)}$; 2. The top-down traversal of $A_{pc}^{Tree}$ on $T$ and $\rho^{A_{Tree}(T,P)}$ as in the $STA^T_{A_{Tree}}(T)$ operation.

Theorem 5.1 describes the relation between $A_{Tree}(T, P)$, $STA_{A_{Tree}}^{Q_{A_{Tree}}}(T, P)$ and $STA_{A_{Tree}}^{Q_{A_{Tree}}}(T)$.

Theorem 5.1 Given a tree $T$, UUTA $A_{Tree}$ and a partition $P : V^T \mapsto Q^{A_{Tree}}$ where $A_{Tree}(T, P)$ is an accepting run. Then, every $v \in V^T$ satisfies $P(v) = \rho^{STA_{A_{Tree}}^{Q_{A_{Tree}}}(T,P)}(v) \in \rho^{STA_{A_{Tree}}^{Q_{A_{Tree}}}(T)}(v)$. 

Proof If $A_{Tree}(T, P)$ is an accepting run then every $v \in V^T$ is annotated by $P(v) = \rho^{A_{Tree}(T,P)}(v)$. The top-down traversal of $STA_{A_{Tree}}^{Q_{A_{Tree}}}(T, P)$ prunes the deriving states of $P(root(T))$. Since $S = Q^{A_{Tree}}$, then, every state $q \in Q^{A_{Tree}}$ is a selecting states. Therefore a node $v$ is annotated by a single selecting state $P(v)$. As a result, $P(v)$ is pruned. Therefore, $P(v) = \rho^{STA_{A_{Tree}}^{Q_{A_{Tree}}}(T,P)}(v)$. From Lemma 5.1 we know that $\rho^{A_{Tree}(T,P)}(v) \in \rho^{A_{Tree}(T)}(v)$. The top-down traversal of both $STA_{Q_{A_{Tree}}}(T)$ and $STA_{A_{Tree}}^{Q_{A_{Tree}}}(T, P)$ is the same. Therefore, $\rho^{STA_{A_{Tree}}^{Q_{A_{Tree}}}(T,P)}(v) \in \rho^{STA_{A_{Tree}}^{Q_{A_{Tree}}}(T)}(v)$. 

Part II Querying of XML in a DB
In each iteration the TwigTA algorithm constructs $A^\text{Tree}$ and $P$ where $A^\text{Tree}(T, P)$ is an accepting run. The fact that $A^\text{Tree}(T, P)$ is an accepting run enables the TwigTA to filter the tree $T$ nodes $v$ where $\rho^{\text{STA}_S^{\text{Twig}}(A^\text{Tree})}(P(v)) = \emptyset$. Lemma 5.2 describes the relations between $\text{STA}_S^{\text{Twig}}(A^\text{Tree})$, $\text{STA}_S^{\text{Twig}}(T)$ and $P$.

**Lemma 5.2** Given a STA $(A^\text{Selecting}, S)$, a tree $T$, a UUTA $A^\text{Tree}$ and a partition $P : \mathcal{V} \to Q^{A^\text{Tree}}$ where $A^\text{Tree}(T, P)$ is an accepting run. If a node $v$ satisfies $q_{\text{Selecting}} \in \rho^{\text{STA}_S^{\text{Twig}}(A^\text{Tree})}(v)$ then $q_{\text{Selecting}} \in \rho^{\text{STA}_S^{\text{Twig}}(A^\text{Tree})}(P(v))$.

**Proof** We extend Theorem 4.4. If a $A^\text{Tree}(T, P)$ is an accepting run then from Theorem 5.1 we get $P(v) \in \rho^{A^\text{Tree}v_{\text{Selecting}}}(A^\text{Tree})(v)$. From the lemma claim we know that $q_{\text{Selecting}} \in \rho^{\text{STA}_S^{\text{Twig}}(T)}(v)$. From Lemma 4.7 we know that $\langle P(v), q_{\text{Selecting}} \rangle \in \rho^{\text{STA}_S^{\text{Twig}}(T)}$. Because of the top down traversal in $\text{STA}_S^{A^\text{Tree}\cap A^\text{Twig}}$, the FSA $(A^\text{Tree} \cap A^\text{Selecting})_p$ derives the state $\langle P(v), q_{\text{Selecting}} \rangle$ by one of the states in $F(A^\text{Tree} \cap A^\text{Selecting})_p$. Therefore, $q_{\text{Selecting}} \in \rho^{\text{STA}_S^{\text{Selecting}}(A^\text{Tree})}(P(v))$.

The TwigTA algorithm uses Lemma 5.2 in negation: If $\rho^{\text{STA}_S^{\text{Twig}}(A^\text{Tree})}(P(v)) = \emptyset$ then $q_{\text{Selecting}} \notin \rho^{\text{STA}_S^{\text{Twig}}(T)}(v)$. Therefore, the node $v$ is not part of any solution of the twig pattern and the TwigTA algorithm filters the node $v$.

The TwigTA algorithm constructs the partition $P$ from a node labeled tree $T \triangleq \langle \mathcal{V}^T, E^T, \text{label}^T \rangle$ and from a finite set of nodes extracted $\subseteq \mathcal{V}^T$. The partition is constructed using the minimal_ancestor relation. Let $v_i \in \mathcal{V}^T$, $v_j \in \text{extracted}$. minimal_ancestor(extracted, $v_i$) = $v_j$ if $v_j$ is the ancestor of $v_i$ that has the maximal level in extracted. The TwigTA algorithm constructs for $v_i \in \mathcal{V}^T$ the partition $P(v_i) = \{ q_v^{\text{self}}, q_v^{\text{desc}} \}$ where $v_j = \text{minimal_ancestor(extracted, } v_i \text{)}$ is the ancestor of $v_i$ that has the maximal level. The partition $P$ constructs two states for each $v \in \text{extracted}$: $q_v^{\text{self}}$ and $q_v^{\text{desc}}$. We denote the state $q_v$ as either $q_v^{\text{self}}$ or $q_v^{\text{desc}}$.

For example, we describe the partition $P$, which is constructed from the tree $T$ in Fig. 3.2 and from the set of nodes labels $\{(0, 29, 1), (1, 6, 2), (3, 4, 4), (12, 13, 5), (14, 15, 5)\}$. Given an encoded node $R_v$, we denote $v$ by $\text{start}_v$ for the simplicity of the presentation. The partition is: $P(0) = q_0^{\text{self}}$, $P(1) = q_0^{\text{self}}$, $P(2) = q_0^{\text{desc}}$, $P(3) = q_1^{\text{self}}$, $P(7) = q_1^{\text{desc}}$, $P(11) = q_2^{\text{desc}}$, $P(12) = q_2^{\text{self}}$, $P(14) = q_3^{\text{self}}$, $P(19) = q_0^{\text{desc}}$, $P(20) = q_0^{\text{desc}}$, $P(22) = q_0^{\text{desc}}$, $P(23) = q_0^{\text{desc}}$, $P(24) = q_0^{\text{desc}}$. Part II Querying of XML in a DB
5.4.1.2 \( STAS_{TA}^{Twig}(ATree, history) \) operation

The \( TWIGTA \) algorithm repeatedly constructs the \( UUTA \) \( ATree \). Each \( ATree \) accepts a subtree of the stored tree \( T \), which is not yet extracted from the streams. Different automata \( ATree \) share a common state \( q_{self} \in QA^{Tree} \) that annotates a common node \( v \in VT \) in different subtrees. The different automata \( ATree \) run on different subtrees that are rooted in \( v \). The \( TWIGTA \) algorithm extends the \( STAS_{TA}^{Twig}(ATree) \) functionality by the function \( history : Q^{ATree} \rightarrow 2Q^{ASelecting} \). The \( history \) function maps a common state \( q_{self} \in QA^{Tree} \) to selecting states \( q_{Selecting} \in \rho^{STAS_{TA}^{Twig}(ATree)}(q_{self}) \) where \( v_c \in children(T, v) \). The extended functionality of \( STAS_{TA}^{Twig}(ATree, history) \) ensures that the repeated \( STAS_{TA}^{Twig}(ATree) \) operates on the whole tree \( T \) and not only on its subtree which has not yet extracted from the DB. Definition 5.3 defines the application of the STA query to \( UUTA \) \( ATree \) and to a \( history \).

**Definition 5.3** Given a STA \( (ATwig, S) UUTA, ATree \) and \( history : Q^{ATree} \rightarrow 2Q^{ASelecting} \). The application of the STA to \( UUTA \) \( ATree \) and to \( history \) is \( STAS_{Selecting}(ATree, history) △ (A_{Selecting}, S) ∙ (ATree, history) \rightarrow \rho^{STAS_{Selecting}(ATree,history)} \). The output of \( STAS_{Selecting}(T) \) operation
\[
\rho^{STAS_{Selecting}(ATree,history)} : Q^{ATree} \rightarrow 2S \text{ maps the selected states } q \in Q^{ATree} \text{ to states in } S. \text{ The selected states are the query answer.}
\]

The \( STAS_{TA}^{Twig}(ATree) \) operation on a tree-\( UUTA \) \( ATree \) has two steps: 1. FSA \( (ATree \cap A_{Selecting})_{pc} \) construction that is given by the \( ConstructIntersectedFSA(ATree, STA) \) function in Algorithm 6; 2. A top-down traversal on \( (ATree \cap A_{Selecting})_{pc} \) that prunes reachable states. The \( STAS_{TA}^{Twig}(ATree, history) \) operation extends the FSA construction in step 1 by using the \( history \) function. The extended FSA construction \( ConstructIntersectedFSA(ATree, STA, history) \) uses the \( history \) function in the internal function \( children-search \). Algorithm 10 describes the modification in the \( children-search \) internal function, which is described in Algorithm 7 for the \( STAS_{TA}^{Twig}(ATree, history) \) operation. Algorithm 10 extends the \( children-search \) for the states \( q_{Selecting} \in QA^{Selecting} \) and for \( q_{Tree} \in QA^{Tree} \) with the states \( q_{Selecting} \in history(q_{Tree}) \). The \( STAS_{TA}^{Twig}(ATree, history) \) operation adds the \( history \) mapping as an input parameter to the functions \( ConstructIntersectedFSA \) and \( CreateTransitions \) in Algorithm 6.
Algorithm 10: The children-search function that receives history.

\begin{align*}
\text{children-search} & (\langle \rho_{\text{Tree}}, \rho_{\text{Selecting}} \rangle \in Q(A_{\text{Tree} \cap A_{\text{Selecting}}})_{pe}, \\
\text{maybe_children} & \subseteq Q(A_{\text{Tree} \cap A_{\text{Selecting}}})_{pe}, \text{ history } : Q_{A_{\text{Tree}}} \mapsto 2Q_{A_{\text{Selecting}}})
\end{align*}

Data: $A_{\text{Tree}}, A_{\text{Selecting}}$

Output: children-sets $\subseteq 2Q(A_{\text{Tree} \cap A_{\text{Selecting}}})_{pe}$

1 begin
2 \hspace{1em} forall $\delta_{A_{\text{Tree}}} (S_{\text{Tree}}, a) = \rho_{\text{Tree}}$ do
3 \hspace{2em} forall $\delta_{A_{\text{Selecting}}} (S_{\text{Selecting}}, a) = \rho_{\text{Selecting}}$ do
4 \hspace{3em} if exists $S \subseteq \text{maybe_children}$ and $S' \subseteq \text{history} (\rho_{\text{Tree}})$ where
5 \hspace{4em} $\cup (\rho_{\text{Tree}}, \rho_{\text{Selecting}}) \in S \text{Tree} = S_{\text{Tree}}$ and
6 \hspace{5em} $S' \cup \cup (\rho_{\text{Tree}}, \rho_{\text{Selecting}}) \in S \text{Selecting} = S_{\text{Selecting}}$ then
7 \hspace{4em} Add $S$ to children-sets;
8 end

Example 5.5 The $A_{\text{Twig}}$, which is constructed from the twig pattern ‘/a[/b/d]/c/d’, is $(Q_{A_{\text{Twig}}}, \Sigma, F_{A_{\text{Twig}}}, \delta_{A_{\text{Twig}}})$ where the alphabet is $\Sigma = \{a, b, c, d, \top\}$, the states of $A_{\text{Twig}}$ are $Q_{A_{\text{Twig}}} = \{q_\bot, q_\top, q_a, q_b, q_c, q_d, q_{d1}, q_{d2}\}$, the accepting state is $F_{A_{\text{Twig}}} = \{q_\top\}$. The transitions are: $\delta_{A_{\text{Twig}}}(\{q_\bot\}, \Sigma) = q_\bot$, $\delta_{A_{\text{Twig}}}(\emptyset, \Sigma) = q_\bot$, $\delta_{A_{\text{Twig}}}(\{q_\bot\}, d) = q_{d1}$, $\delta_{A_{\text{Twig}}}(\emptyset, d) = q_{d1}$, $\delta_{A_{\text{Twig}}}(\{q_\bot, q_{d1}\}, b) = q_b$, $\delta_{A_{\text{Twig}}}(\{q_\bot\}, d) = q_{d2}$, $\delta_{A_{\text{Twig}}}(\emptyset, d) = q_{d2}$, $\delta_{A_{\text{Twig}}}(\{q_\bot, q_{d2}\}, c) = q_c$, $\delta_{A_{\text{Twig}}}(\{q_\bot, q_b, q_c\}, a) = q_a$. $\delta_{A_{\text{ Twig}}}(\{q_\bot, q_a\}, \top) = q_\top$. Figure 5.3(a) describes the tree $T$ from which the $A_{\text{Tree}}$ is constructed. $A_{\text{Tree}}$ accepts the subtree that includes nodes 1, 4 and 5. $A_{\text{Tree}}$ is the tuple $(Q_{A_{\text{Tree}}}, \Sigma, F_{A_{\text{Tree}}}, \delta_{A_{\text{Tree}}})$ where the states are $Q_{A_{\text{Tree}}} = \{q_{\text{self}}, q_1^{\text{self}}, q_4^{\text{self}}, q_5^{\text{self}}\}$, the alphabet is $\Sigma = \{\top, a, c, d\}$, the accepting state is $F_{A_{\text{Tree}}} = \{q_{\top}\}$ and the transition function includes the following transitions: $\delta_{A_{\text{Tree}}}(\emptyset, d) = q_5^{\text{self}}$, $\delta_{A_{\text{Tree}}}(\{q_5^{\text{self}}\}, c) = q_4^{\text{self}}$, $\delta_{A_{\text{Tree}}}(\{q_4^{\text{self}}\}, a) = q_1^{\text{self}}$, $\delta_{A_{\text{Tree}}}(\{q_1^{\text{self}}\}, \top) = q_{\top}$. The previous $A_{\text{Tree}}$ which that was constructed from nodes 1, 2 and 3, assigned the history($q_1^{\text{self}}$) = $\{q_b\}$ where $q_b \in \rho_{\text{STA}(A_{\text{Tree}})}(\{2^{\text{self}}\}^2)$ and $2 \in \text{children}(T, 1)$. Figure 5.3(b) illustrates the $(A_{\text{Tree}} \cap A_{\text{Twig}})_{pe}$ that is constructed by the $\text{STA}_{\text{Tree}}(A_{\text{Tree}}, \text{history})$ operation. The state $\langle q_1^{\text{self}}, q_a \rangle \in Q(A_{\text{Tree} \cap A_{\text{Twig}}})_{pe}$ because the state $q_a, q_\bot$ are joined with the state $q_b \in \text{history}(q_1^{\text{self}})$ in the children-search in CreateTransitions function operation when states $\langle q_1^{\text{self}}, q_b \rangle$ is the input parameter $(\rho_{\text{Tree}}, \rho_{\text{Selecting}})$ to Algorithm 10. In this example the final reachable states of $Q(A_{\text{Tree} \cap A_{\text{Selecting}}})_{pe}$ are: $\rho_{\text{STA}_{\text{Tree}}(A_{\text{Tree}}, \text{history})}(q_1^{\text{self}}) = q_a$, $\rho_{\text{STA}_{\text{Tree}}(A_{\text{Tree}}, \text{history})}(q_1^{\text{self}}) = q_c$ and $\rho_{\text{STA}_{\text{Tree}}(A_{\text{Tree}}, \text{history})}(q_1^{\text{self}}) = q_{d2}$. \hfill \Box
Figure 5.3: (a) Demonstrates the tree $T$ from which $A^{Tree}$ and $history$ in example 5.5 are constructed. A node $v \in V^T$ is denoted by a circle. A black circle denotes nodes from which the current $A^{Tree}$ is constructed and has a label in the format ‘$v$; label$^T(v)$’. A gray circle denotes a node from which the previous $A^{Tree}$ was constructed and has a two lines label. The top line is in the format of the black circle labels. The bottom line is in the format ‘$q$’ where $q \in \rho^{STA^{\text{ Twig}}} (A^{Tree}_{old})(q_{self}^T)$. (b) Demonstrates the construction of $(A^{Tree} \cap A^{Twig})_{pc}$ by the $STA^{\text{ Twig}}(A^{Tree}, history)$ operation where the $A^{Tree}$ and the $history$ are described in example 5.5. A state $\langle \langle q^T_{Tree}, q^T_{twig} \rangle \rangle$ is denoted by a labeled circle. The label denotes the state Id. A transition $\delta((A^{Tree} \cap A^{Twig})_{pc}) (\langle q^T_{Tree}, q^T_{twig} \rangle, \langle q^T_{Tree}, q^T_{twig} \rangle) = \langle q^T_{Tree}, q^T_{twig} \rangle$ is denoted by an edge from $\langle q^T_{Tree}, q^T_{twig} \rangle$ circle to $\langle q^T_{Tree}, q^T_{twig} \rangle$ circle. The start state $q^T_0$ is denoted by an incoming edge. The states in $F((A^{Tree} \cap A^{Twig})_{pc}$ are denoted by double circles.

### 5.4.2 Description of the TwigTA Algorithm

The TwigTA algorithm iteratively traverses an XML document that is stored in a DB $D^T$. In each iteration, it extracts a finite set of nodes labels (extracted) from $D^T$. Figure 5.4 illustrates the nodes extraction process. The grey nodes 0, 19, 20, 22, 24 in Fig. 5.4 are the extracted set in the current iteration. The current iteration ex-
amines whether to construct intermediate paths for the present-nodes $v_{\text{present}}$ where $\text{min\_present} \leq \text{start}_{v_{\text{present}}} < \text{min\_future}(D^T)$. Nodes 19, 20, 22 in Fig. 5.4 are the present-nodes where $\text{min\_present} = 19$ and $\text{min\_future}(D^T) = 23$. If $\text{min\_present} \leq \text{start}_{v_{\text{present}}} < \text{min\_future}(D^T)$ then $v_{\text{present}} \in \text{extracted}$. A past-node $v_{\text{past}}$, which was extracted and processed in previous iterations, satisfies $\text{start}_{v_{\text{past}}} < \text{min\_present}$. There are two types of past-nodes: If a past-node $v_{\text{past}}$ is an ancestor of a present-node $v_{\text{present}} \in \text{extracted}$ then $v_{\text{past}} \in \text{extracted}$. For example, node 0 in Fig. 5.4 is an ancestor of a present-node 19. Otherwise, a past-node was removed from the $\text{extracted}$ set in previous iterations like nodes 1, 2, . . ., 14 in Fig. 5.4 which are denoted by white circles with dashed boundaries. There are also two types of a future-node $v_{\text{future}}$ where $\text{start}_{v_{\text{future}}} \geq \text{min\_future}(D^T)$. A $v_{\text{future}} \in \text{extracted}$ is an extracted node like node 24 in Fig. 5.4. Otherwise, $v_{\text{future}}$ has not yet extracted from $D^T$. For example, node 23 in Fig. 5.4 which is denoted by a white circle with continuous boundaries.

Figure 5.4: Illustration of the status of $D^T$ nodes during the TwigTA operation. A node is denoted by a circle. A label of node $v$ has the format ‘label$^T(v); \text{start}_v$’. A gray circle denotes a node that is currently extracted. A white circle with a dashed boundary denotes a past-node that contributed an intermediate-path in a previous iteration. A white circle with a continues boundary denotes a future-node that will be extracted in a future iteration. $\text{min\_present}$ points to the minimal extracted node that is currently examined by the algorithm. $\text{min\_future}(D^T)$ points to the minimal node in $D^T$.

Given $D^T$ and $\text{extracted} \subseteq \text{Nodes}^T$. We define two subtrees of tree $T$ that is stored in $D^T$:

$$\text{present\_future\_subtree}(D^T, \text{extracted}) \triangleq (V_{\text{present\_future}}, E_{\text{present\_future}})$$

is a subtree of $T$. $V_{\text{present\_future}}$ contains all the nodes $v$ where $R_v \in D^T$ and $R_v \in \text{extracted}$. If two nodes $v_i, v_j \in V_{\text{present\_future}}$ satisfy $\text{minimal\_ancestor}(V_{\text{present\_future}}, \text{start}_{v_j}) = R_{v_i}$ then $(v_i, v_j) \in E_{\text{present\_future}}$. 

Part III Querying of XML in a DB
*present-subtree* ($D^T, \text{extracted}$) $\triangleq (V^{\text{present}}, E^{\text{present}})$ is the subtree of T. $V^{\text{present}}$ contains all the past-nodes and the present-nodes $v$ where $R_v \in \text{extracted}$. If two nodes $v_i, v_j \in V^{\text{present}}$ satisfy $\text{minimal\_ancestor}(V^{\text{present}}, \text{start}_{v_j}) = R_{v_i}$ then $(v_i, v_j) \in E^{\text{present}}$.

The $A^{\text{Tree}}$, which is constructed by the TwigTA algorithm in the present iteration, accepts $\text{present-\_future-subtree}(D^T, \text{extracted})$. Nodes 0, 19, 20, 22, 23, 24 in Fig. 5.4 compose $\text{present-\_future-subtree}(D^T, \text{extracted})$. The $\text{present-subtree}(D^T, \text{extracted})$ is a subtree of the $\text{present-\_future-subtree}(D^T, \text{extracted})$. When TwigTA constructs intermediate paths, it constructs intermediate paths that compose $\text{present-subtree}(D^T, \text{extracted})$ of the present iteration. Figures 5.5(a) and 5.5(b) illustrate the $\text{present-\_future-subtree}(D^T, \text{extracted})$ and the $\text{present-subtree}(D^T, \text{extracted})$, respectively.

![Diagram](image)

Figure 5.5: Illustration of the $\text{present-\_future-subtree}(D^T, \text{extracted})$ and $\text{present-subtree}(D^T, \text{extracted})$ which are constructed from $D^T$ and $\text{extracted}$ in Fig. 5.4. A node is denoted by a circle. A label of node $v$ is in the format ‘$\text{label}^T(v); \text{start}_v$’. A gray circle denotes an extracted node. A white circle denotes a future-node.

Part II: Querying of XML in a DB
Algorithm [11] describes the TwigTA operation. Initially, the algorithm constructs the STA \((A^{Twig}, S) \triangleq F_{pc}^{Twig}\), from the twig pattern \(Q\). Then, the TwigTA algorithm iteratively traverses \(D_T\). In each iteration, the TwigTA extracts a maximum of \(K\) nodes from each stream \(D_T(a)\) \((extracted)\). We call this algorithm \(K\)-prediction where \(K\) is the number extracted nodes. For example, extraction of a single node and extraction of two nodes from each stream are called 1-prediction TwigTA algorithm and 2-prediction TwigTA algorithm, respectively. In each iteration, the algorithm constructs the partition \(P\) from the extracted nodes. Then, TwigTA constructs a prediction-tree \(T_{Prediction}\) from \(P\). The prediction-tree reflects the structure of the \(present\)-\(future\)-subtree \((D_T, extracted)\). It is called a prediction-tree because it predicts the structure of the future-nodes in the \(present\)-\(future\)-subtree \((D_T, extracted)\) from the nodes in the \(extracted\) set and the state of DB \(D_T\). Next, TwigTA constructs a UUTA ATree from \(T_{Prediction}\) and computes \(\rho^{STA_S^{Twig}(ATree, history)}(P(v)) = \emptyset\) then \(v\) is removed and another node is extracted from \(D_T\) instead. Otherwise, TwigTA constructs the intermediate paths that are in \(present\)-\(subtree\) \((D_T, extracted)\) and then removes the nodes \(R_v\) where \(v \in E_{present}\).
Algorithm 11: The TwigTA algorithm.

**Input:** Twig Q, DT, K ∈ N

**Data:** UUTA ATwig, FSA APc ATwig, UUTA ATree,

\[ T^{Prediction} \triangleq (V^{Prediction} \triangleq Q^{Tree}, E^{Prediction}, label^{Prediction}) \]

where

- label^{Prediction} : V^{Prediction} → 2^{2^{\mathbb{N}}}, extracted ⊆ Nodes^{T},
- history : Q^{Tree} → 2^{Q^{Tree}}, \rho^{STA_{ATwig}(ATree,history)} : Q^{ATree} → 2^{Q^{ATwig}},
- min_present ∈ Positions^{T}, P : Positions^{T} → Q^{ATree},
- paths : Q^{ATwig} → (Path^{T}_1, ..., Path^{T}_n)

**Output:** solutions : Solution^{T}_1, ..., Solution^{T}_n

1 begin
2 \( history ← \emptyset; min\_present ← 0; \)
3 construct ATwig, APc ATwig from Q ; \( \quad \) /* See Chapter 4 */
4 Let STA be (ATwig, S);
5 extracted ← \{R_vroot\} where \( R_vroot = (-\infty, \infty, 0) \) and \( label^{T}(v\_root) = \top ; \)
6 while ExtractNodes(DT, extracted, P, min_present, K) > 0:
7 \( \quad \) /* Algorithm.16 */
8 do
9 \( P ← MakePartition(DT, extracted); \) \( \quad \) /* Algorithm.12 */
10 \( T^{Prediction} ← ConstructTree(DT, P); \) \( \quad \) /* Algorithm.13 */
11 \( ATree ← ConstructAutomaton(T^{Prediction}); \) \( \quad \) /* Algorithm.14 */
12 \( \rho^{STA_{ATwig}(ATree,history)} ← STA_{ATwig}(ATree, history); \) \( \quad \) /* See Chapter 4 */
13 \( \quad \) /* Algorithm.15 */
14 if
15 \( RemoveUnselected(extracted, P, \rho^{STA_{ATwig}(ATree,history)}, history) = 0 \) then
16 \( \quad \) /* Algorithm.17 */
17 \( IntermediatePaths(root(T^{Prediction}), ATwig, APc ATwig, (\), \rho^{STA_{ATwig}(ATree,history)}, ATwig, T^{Prediction}, DT, min\_present, paths); \)
18 \( \quad \) /* Algorithm.18 */
19 \( RemovePresent(DT, extracted, P, \rho^{STA_{ATwig}(ATree,history)}, history, T^{Prediction}); \)
20 \( min\_present = min\_future(DT); \)
21 solutions ← MergeAllPathSolutions(paths); \( \quad \) /* See [106] */
22 end

Part II: Querying of XML in a DB
5.4.3 Partition construction

The partition $P : V^T \mapsto Q^{A_{\text{Tree}}}$ in Definition 5.3 maps tree $T$ nodes to $A^{\text{Tree}}$ states. The TwigTA partition $P : Positions^T \mapsto Q^{A_{\text{Tree}}}$ maps start$_v$ and end$_v$ encodings of node $v$ to $A^{\text{Tree}}$ states. The domain of partition $P$ is changed from $V^T$ into $Positions^T$ because TwigTA traverses nodes labels $R_v$ in $D^T$ instead of nodes $v$ of tree $T$. The partition content, however, does not change. The partition constructs for $R_v \in \text{Nodes}^T$, where $v$ is a node in the present-future-subtree($D^T$, extracted),

$$P(\text{start}_v) = P(\text{end}_v) = \begin{cases} q_v^{\text{self}}, & R_v \in \text{extracted} \\ q_v^{\text{descendent}}, & R_v = \text{minimal_ancestor}(\text{extracted}, \text{start}_v). \end{cases}$$

Algorithm 12 describes the partition construction.

Algorithm 12: The MakePartition algorithm.

```
MakePartition($D^T$,extracted)
Output: $P : Positions^T \mapsto Q^{A_{\text{Tree}}}$
begin
    forall pos $\in Positions^T$ do
        if pos = start$_v$ or pos = end$_v$ where $R_v \in \text{extracted}$ then
            Add $P(pos) = q_v^{\text{self}}$;
        else if pos $\geq$ min_future($D^T$) then
            Let $R_v \leftarrow \text{minimal_ancestor}(\text{extracted}, pos)$;
            Add $P(pos) = q_v^{\text{descendent}}$;
    end
```

Example 5.6 The first iteration in the run of 1-prediction TwigTA on $D^T$, which stores tree $T$ in Fig. 3.2, extracts the following nodes from the streams: head($D^T(a)$) = (0, 29, 1), head($D^T(b)$) = (1, 6, 2), head($D^T(c)$) = (3, 4, 4), head($D^T(d)$) = (12, 13, 5) and head($D^T(e)$) = (14, 15, 5). The virtual stream of the root $D^T(\top)$ contributes the label $(-\infty, \infty, 0)$. Therefore, $\text{extracted} = \{(-\infty, \infty, 0), (0, 29, 1), (1, 6, 2), (3, 4, 4), (12, 13, 5), (14, 15, 5)\}$ and the partition is: $P(0) = P(29) = q_0^{\text{self}}$, $P(1) = P(6) = q_1^{\text{self}}$, $P(2) = P(5) = q_0^{\text{descendent}}$, $P(3) = P(4) = q_3^{\text{self}}$, $P(7) = P(8) = \ldots = P(11) = q_0^{\text{descendent}}$, $P(12) = P(13) = q_2^{\text{self}}$, $P(14) = P(15) = q_4^{\text{self}}$, $P(16) = P(17) = \ldots = P(28) = q_0^{\text{descendent}}$. Note that we do not map $q_{-\infty}^{\text{descendent}}$ in partition $P$ because positions 30, $\ldots$, $\infty$ are not encoded by any node in present-future-subtree($D^T$, extracted).
5.4.4 Prediction-tree construction

\( T_{\text{Prediction}} \) is a node-labeled tree \( T = (V_{\text{Prediction}} \overset{\Delta}{=} Q^{A_{\text{Tree}}}, E_{\text{Prediction}}, \text{label}_{\text{Prediction}}) \)

where

\( \text{label}_{\text{Prediction}} : V_{\text{Prediction}} \mapsto 2^{{2^T}_{\text{Twig}}} \). A node \( q_{v}^{\text{descendants}} \) can have multiple labels because it maps multiple future-nodes from the original XML document.

Algorithm\[13\] describes the construction of \( T_{\text{Prediction}} \). In each iteration of \( \text{TwigTA} \), the \( T_{\text{Prediction}} \) is constructed from the partition \( P \). For all \( v_{p}, v_{c} \in V_{\text{Prediction}} \), the construction satisfies that if \( P(\text{start}_{v_{p}}) \) is a parent of \( P(\text{start}_{v_{c}}) \) in \( T_{\text{Prediction}} \) then \( v_{p} \) is an ancestor of \( v_{c} \) in the \( \text{present-future-subtree}(D, \text{extracted}) \). The following notations extends the \text{minimal\_ancestor} definition for states in a partition. These notations are used by Algorithm\[13\]

**positions**: Given partition \( P : \text{Positions}^{T} \mapsto Q^{A_{\text{Tree}}} \), \( q \in Q^{A_{\text{Tree}}} \). The positions function is defined by \( \text{positions}(P, q) = \{ \text{pos} \in \text{Positions}^{T} \mid P(\text{pos}) = q \} \).

**ancestor**: Given partition \( P : \text{Positions}^{T} \mapsto Q^{A_{\text{Tree}}} \), \( q_{a}, q_{d} \in Q^{A_{\text{Tree}}} \). The ancestor function is defined by \( \text{ancestor}(P, q_{a}, q_{d}) = 1 \) if \( \text{min}(\text{positions}(P, q_{a})) < \text{min}(\text{positions}(P, q_{d})) \) and \( \text{max}(\text{positions}(P, q_{a})) > \text{max}(\text{positions}(P, q_{d})) \). Otherwise,

\[
\text{ancestor}(P, q_{a}, q_{d}) = 0.
\]

**minimal\_ancestor**: Given partition \( P : \text{Positions}^{T} \mapsto Q^{A_{\text{Tree}}} \), \( q_{a}, q_{d} \in Q^{A_{\text{Tree}}} \).

\( \text{minimal\_ancestor} \) is defined by \( \text{minimal\_ancestor}(P, q_{d}) = q_{a} \) where \( \text{ancestor}(P, q_{a}, q_{d}) = 1 \) and if another state, \( q_{o} \in Q^{A_{\text{Tree}}} \), also satisfies \( \text{ancestor}(P, q_{o}, q_{d}) = 1 \) then \( \text{ancestor}(P, q_{o}, q_{a}) = 1 \).

Given \( \text{pos} \in \text{Positions}^{T}, D^{T} \) and \( a \in \Sigma. \, \text{labels}_{\text{future}} : D^{T} \times \text{Positions}^{T} \mapsto 2^\Sigma \) is defined by \( a \in \text{labels}_{\text{future}}(D^{T}, \text{pos}) \) if \( \text{pos} \geq \text{start}_{v} \) where \( R_{v} = \text{head}(D^{T}(a)) \).

Example\[5.7\] demonstrates these notation.

**Example 5.7** The XML tree \( T \) in Fig. 3.2 illustrates an XML region encoding. \( b \in \text{labels}_{\text{future}}(D^{T}, 8) \) because \( \text{head}(D^{T}(b)) = (1, 6, 2) \) and \( 8 \geq 1. \, d \notin \text{labels}_{\text{future}}(D^{T}, 8) \) because \( \text{head}(D^{T}(d)) = (12, 13, 5) \) and \( 8 < 12. \) □
Algorithm 13: The ConstructTree algorithm.

Input: $T^{Prediction}$
Output: $T^{Prediction} = (V^{Prediction}, Q^{Tree}, E^{Prediction}, label^{T^{Prediction}})$
begin
  forall $q^{self}_v \in Q^{Tree}$ do
    label$^{T^{Prediction}}(q^{self}_v) \leftarrow \{\text{label}(v)\}$;
  forall $q^{descendent}_v \in Q^{Tree}$ do
    label$^{T^{Prediction}}(q^{descendent}_v) \leftarrow$ labels$^{T^{Prediction}}(q^{descendent}_v)$;
  forall $q_p, q_c \in V^{Prediction}$ where $q_p = \minimal_ancestor(P, q_c)$ do
    Add $(q_p, q_c)$ to $E^{T^{Prediction}}$;
end

Example 5.8 Figure 5.6 describes the prediction-tree that is constructed from the partition in Example 5.6 by Algorithm 13. Algorithm 13 first assigns the labels to $q^{self}_v \in Q^{Tree}$. Next, it assigns the labels to $q^{descendent}_v \in Q^{Tree}$. We examine the labels that were assigned to state $q^{descendent}_1$. positions$(P, q^{descendent}_1) = \{2, 5\}$ and, therefore, $\max(positions(P, q^{descendent}_v)) = 5$. The algorithm assigns only the labels of $\max(positions(P, q^{descendent}_v))$ because, when given $pos_i, pos_j \in Positions^T$ if $pos_i > pos_j$ then labels$^{T^{Prediction}}(D^T, pos_i) \subseteq labels^{T^{Prediction}}(D^T, pos_j)$. Therefore, labels$^{T^{Prediction}}(D^T, max(positions(P, q^{descendent}_v)))$ because $\text{start}_v = 2 < 5$ where $R_v = \text{head}(D^T(a)) = (2, 5, 3)$. Therefore, $a \in labels^{T^{Prediction}}(q^{descendent}_1)$ and future-nodes in positions 2 and 5 can have the label ‘a’. start$_v$ where $R_v = \text{head}(D^T(\sigma))$ for $\sigma: b, c, d$ and $e$ are 7, 9, 20 and 24, respectively. All these positions are greater than 5. Therefore, future-nodes in positions 2 and 5 do not have the labels ‘b’, ‘c’, ‘d’ and ‘e’. Algorithm 13 ends by adding edges to $T^{Prediction}$. For example, $(q^{descendent}_7, q^{self}_{12}) \in E^{T^{Prediction}}$ because positions$(P, q^{descendent}_7) = \{7, \ldots, 11, 16, \ldots, 28\}$, positions$(P, q^{self}_{12}) = \{12, 13\}$. Therefore, $\max(P, q^{descendent}_7) = 28 > 13 = \max(P, q^{self}_{12})$ and $\min(P, q^{descendent}_7) = 7 < 12 = \min(P, q^{self}_{12})$. On the way, we can see that $\text{ancestor}(P, q^{self}_0, q^{descendent}_7) = 1$ and $\text{ancestor}(P, q^{self}_{-\infty}, q^{descendent}_7) = 1$. Therefore, $\minimal_ancestor(P, q^{self}_v) = q^{descendent}_7$.

Part II: Querying of XML in a DB
Figure 5.6: Illustration of the 1-prediction $T^{Prediction}$ that is constructed from the partition in Example 5.6. A white circle and a gray circle denote $q^{descendants}_v$ nodes and $q^{self}_v$ nodes, respectively. The label of node $v$ has two lines. The top line is in the format ‘$v; label^{Prediction}(v)$’ where $label^{Prediction}(v) = label_1, \ldots, label_n$. The bottom line is in the format ‘positions($P, v$)’ where positions($P, v$) = pos_1, \ldots, pos_n.

Figure 5.7 describes 2-prediction $T^{Prediction}$ which is constructed from the partition $P$ where $P$ is constructed from $extracted = \{(-\infty, \infty, 0), (0, 29, 1), (2, 5, 3), (1, 6, 2), (7, 18, 2), (3, 4, 4), (9, 10, 4), (12, 13, 5), (20, 21, 3), (14, 15, 5) (24, 25, 5)\}$. 

Part II: Querying of XML in a DB
Figure 5.7: Illustration of the 2-prediction $T^{\text{Prediction}}$. A white circle and a gray circle denote a $q^{\text{descendants}}$ node and a $q^\text{self}$ node, respectively. The label of node $v$ is in the format ‘$v; \text{label}^{T^{\text{Prediction}}}(v)$’ where $\text{label}^{T^{\text{Prediction}}}(v) = \text{label}_1, \ldots, \text{label}_n$.

5.4.5 $A^\text{Tree}$ construction

Algorithm 14 constructs $A^\text{Tree}$ from $T^{\text{Prediction}}$. The states of $A^\text{Tree}$ are the $T^{\text{Prediction}}$ vertices $V^{T^{\text{Prediction}}}$. Two construction rules translate the prediction-tree into an automaton $A^\text{Tree}$.

1. $q^\text{self}$ rule: Assume $T = \text{present-future-subtree}(D, \text{extracted})$. This rule constructs transitions which annotate a node $v$ in $T$, $R_v \in \text{extracted}$, by $P(\text{start}_v) = q^\text{self}_v$. Node $v$ is annotated when all the $v_c \in \text{children}(T, v)$ are annotated by $P(\text{start}_{v_c}) \in \text{children}(T^{\text{Prediction}}, q^\text{self}_v)$ (see lines 3–4 in Algorithm 14).

2. $q^{\text{descendent}}$ rule: Assume $T = \text{present-future-subtree}(D, \text{extracted})$. The rule constructs transitions which annotate a future-node $v$ in $T$ by $P(\text{start}_v) = q^{\text{descendent}}_u$ when a subset of $v_c \in \text{children}(T, v)$ are annotated by $P(\text{start}_{v_c}) \in \text{children}(T^{\text{Prediction}}, q^{\text{self}}_v)$, or by $q^{\text{descendent}}_u$ itself (see lines 5–8 in Algorithm 14).

Definition 4.7 defines the $A^\text{SMax}$. Each $q \in Q^{A^\text{SMax}}$ has a maximal children-set $S_q^{\text{Max}}$. If there is a transition $\delta^{A^\text{SMax}}(S_q, a) = q$ then $S_q \subseteq S_q^{\text{Max}}$. When $A^\text{Tree}$ is a $A^\text{SMax}$, the $\text{ST}A^A_{\text{Twig}}(A^\text{Tree})$ operation is optimized. From the $A^\text{Tree}$ construction rules we see that in $\text{TwigTA}$ algorithm $A^\text{Tree}$ is a $A^\text{SMax}$. $q^\text{self}_v$ rules construct a single transition $\delta^{A^\text{Tree}}(\text{children}(T^{\text{Prediction}}, q^\text{self}_v),\text{label}^{T^{\text{Prediction}}}(v)) = q^\text{self}_v$. Therefore, $S_q^{\text{Max}} = \text{children}(T^{\text{Prediction}}, q^\text{self}_v)$ is a maximal children-set. $q^{\text{descendent}}_v$ rule constructs transitions where $S_q^{\text{Max}} = \text{children}(T^{\text{Prediction}}, q^{\text{descendent}}_v) \cup \{q^{\text{descendent}}_v\}$. Therefore,
Example 5.9 The automaton, which is constructed from the \( STA_{S}^{Twig}(ATree, history) \) operation is optimized in the TwigTA algorithm. The run \( ATree(T', P) \), where \( T' \overset{\Delta}{=} present-future-subtree(D, extracted) \) in the present iteration and \( P \) is constructed in the present iteration, is an accepting run as shown in Example 5.9.

Algorithm 14: The ConstructAutomaton algorithm.

<table>
<thead>
<tr>
<th>Input: ( T^{Prediction} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: ( ATree \overset{\Delta}{=} (Q_{ATree}, \Sigma, F_{ATree}, \delta_{ATree}) )</td>
</tr>
</tbody>
</table>

1. \( \delta_{ATree} \leftarrow \emptyset; \) 2. \( \text{forall } q_{v}^{\text{self}} \in Q_{ATree} \) do
3. \( \delta_{ATree}(\text{children}(T^{Prediction}, q_{v}^{\text{self}}), \text{label}(v)) = q_{v}^{\text{self}} \)
4. \( \text{forall } q_{v}^{\text{descendent}} \in Q_{ATree} \) do
5. \( \text{forall } a \in \text{label}(T^{Prediction}(q_{v}^{\text{descendent}})), S \subseteq \text{children}(T^{Prediction}, q_{v}^{\text{descendent}}) \cup \{ q_{v}^{\text{descendent}} \} \) do
6. \( \delta_{ATree}(S, a) = q_{v}^{\text{descendent}} \)
7. \( F_{ATree} \leftarrow \{ \text{root}(T^{Prediction}) \} \);

Example 5.9 The automaton, which is constructed from the 1-prediction \( T^{Prediction} \) in Fig. 5.6 by Algorithm 14 is \( ATree = (Q_{ATree}, \Sigma, F_{ATree}, \delta_{ATree}) \) where \( \Sigma = \{a, b, c, d, e\}, Q_{ATree} = \{q_{\text{self}}, q_{0}^{\text{self}}, q_{0}^{\text{descendent}}, q_{1}^{\text{self}}, q_{1}^{\text{descendent}}, q_{3}^{\text{self}}, q_{12}^{\text{self}}, q_{14}^{\text{self}}\}, F_{ATree} = \{q_{\text{self}}\}. \)

The construction of the transitions is as follows:
1. \( \delta_{ATree}(\emptyset, c) = q_{3}^{\text{self}} \), 2. \( \delta_{ATree}(\emptyset, d) = q_{12}^{\text{self}} \), 3. \( \delta_{ATree}(\emptyset, a) = q_{1}^{\text{descendent}} \), 4. \( \delta_{ATree}(\{q_{3}^{\text{self}}\}, a) = q_{1}^{\text{descendent}} \), 5. \( \delta_{ATree}(\emptyset, e) = q_{14}^{\text{self}} \), 6. \( \delta_{ATree}(\{q_{3}^{\text{self}}, q_{1}^{\text{descendent}}\}, a) = q_{14}^{\text{self}} \), 7. \( \delta_{ATree}(\{q_{1}^{\text{descendent}}\}, a) = q_{1}^{\text{descendent}} \), 8. \( \delta_{ATree}(\{q_{1}^{\text{descendent}}\}, b) = q_{3}^{\text{self}} \), 9. \( \delta_{ATree}(\emptyset, \Sigma) = q_{0}^{\text{descendent}} \), 10. \( \delta_{ATree}(\{q_{0}^{\text{descendent}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 11. \( \delta_{ATree}(\{q_{12}^{\text{self}}, q_{14}^{\text{self}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 12. \( \delta_{ATree}(\{q_{1}^{\text{descendent}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 13. \( \delta_{ATree}(\{q_{12}^{\text{self}}, q_{14}^{\text{self}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 14. \( \delta_{ATree}(\{q_{14}^{\text{self}}, q_{0}^{\text{descendent}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 15. \( \delta_{ATree}(\{q_{12}^{\text{self}}, q_{14}^{\text{self}}, q_{0}^{\text{descendent}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 16. \( \delta_{ATree}(\{q_{12}^{\text{self}}, q_{14}^{\text{self}}, q_{0}^{\text{descendent}}\}, \Sigma) = q_{0}^{\text{descendent}} \), 17. \( \delta_{ATree}(\{q_{1}^{\text{self}}, q_{0}^{\text{descendent}}\}, a) = q_{0}^{\text{self}} \), 18. \( \delta_{ATree}(\{q_{0}^{\text{self}}\}, \Sigma) = q_{\text{self}} \). Figure 5.8 illustrates the run \( ATree(T', P) \) where \( T' \overset{\Delta}{=} present-future-subtree(D, extracted) \) is given in Example 5.6. \( ATree \) is constructed in this example and \( present-future-subtree(D, extracted) \) is illustrated in Fig. 5.2. In the first iteration of the TwigTA operation \( T' = T \) where \( T \) is the XML tree that is stored in \( D^{T} \). Figure 5.8 illustrates that, as requested by Lemma 5.2, the run \( ATree(T, P) \) is accepting. \( \square \)
Figure 5.8: Illustration of the run of $A^{Tree}(T, P)$ on the tree in Fig. 3.2 where $A^{Tree}$ is described in example 5.9 and $P$ is given in Example 5.6. A white circle and a gray circle denote stored nodes and extracted node, respectively. The label of node $v$ has two rows. The top row is in the format ‘$\text{label}^T(v)$’. The bottom row is in the format ‘$\rho_{A^{T\text{ree}}(T, P)}(v); Id$’ where $Id$ is the Id of $\delta_{A^{T\text{ree}}}(T)$ in example 5.9 that annotates $v$.

5.4.6 Selecting nodes

We give an example of the $\rho_{ST A^{T\text{wig}}(A^{Tree}, \text{history})}$ that is returned in the first iteration of the 1-prediction $T\text{wig}TA$ operation on the twig pattern ‘//a//c//b//e//d’ and $D^T$ in Example 3.1. The $\text{history}$ function is still empty in the first iteration and, therefore, $\rho_{ST A^{T\text{wig}}(A^{Tree})} = \rho_{ST A^{T\text{wig}}(A^{Tree}, \text{history})}$.

Example 5.10 Figure 5.9 describes the automata, which are constructed, in the $ST A^{T\text{wig}}(A^{Tree})$ operation that is described in Chapter 4. The STA is $\mathcal{A}^{T\text{wig}}, S$ where $S \triangleq F_{A^{T\text{wig}}}$, the $A^{T\text{wig}}$ is described in Example 4.2 and the $A^{T\text{pc}}$ is described in Fig. 5.9(b). The $A^{Tree}$ is described in Example 5.9. The $A^{T\text{pc}}$ is described in Fig. 5.9(a). Figure 5.9(c) illustrates $(A^{Tree} \cap A^{T\text{wig}})_{pc}$ that is constructed by Algorithm 6. For the simplicity of the illustration, we removed the states $q_\perp$ and $(q, q_\perp)$ where $q \in Q_{A^{T\text{ree}}}$ from Figs. 5.9(b) and 5.9(c), receptively. $\rho_{ST A^{T\text{wig}}(A^{Tree})}$ is constructed from the accepting states in $F(A^{Tree} \cap A^{T\text{wig}})_{pc}$. In this example $\rho_{ST A^{T\text{wig}}(A^{Tree})}(q_{0\text{self}}) = \{q_a\}$.
\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{self}_3) = \{q_c\}, \]
\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{descendent}_0) = \{q_a, q_b, q_c, q_d, q_e\}, \]
\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{descendent}_{12}) = \{q_d\}, \]
\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{descendent}_{14}) = \{q_e\}. \]

We see that \( \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{self}_1) = \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{descendent}_1) = \emptyset \). If \( \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{Tree}) \neq \emptyset \) then \( \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{Tree}) \) is enriched by states \( q^\text{Tree} \) where \( \langle q^\text{Tree}, q^\text{Twig} \rangle \in Q(A\text{T}\text{wig} \cap A\text{T}\text{wig})_{\text{pc}} \) is an accessible state in \( (A\text{T}\text{tree} \cap A\text{T}\text{wig})_{\text{pc}} \) and \( \langle q^\text{Tree}, q^\text{Twig} \rangle \notin F(A\text{T}\text{tree} \cap A\text{T}\text{wig})_{\text{pc}} \). The enrichment is done in order to support the \textit{history} function update in Algorithm 18 and the construction of intermediate paths in Algorithm 17. TwigTA enriches \( \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree}) \) as follows:

\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{descendent}_0) = \{q_a, q_u_a, q_b, q_c, q_u_c, q_d, q_e, q_u_e\} \]
\[ \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree})(q^\text{self}_0) = \{q_a, q_u_a\}. \]

This enriched \( \rho_{\text{ST A}\text{T} \text{wig}}(\text{ATree}) \) is returned in the first iteration of the 1-prediction TwigTA operation on the twig pattern ‘//a[///c]/b[///e]/d’ and \( D^T \) in Example 3.1.

Part II: Querying of XML in a DB
Figure 5.9: (a) Illustrates the $A^{Tree}_{pc}$ which is constructed from the $A^{Tree}$ that is detailed in Example 5.9. (b) Illustrates the $A^{Twig}_{pc}$ which is constructed from the $A^{Twig}$ that is detailed in Example 4.2. (c) Illustrates the $(A^{Tree} \cap A^{Twig})_{pc}$ after the removal of unaccessible states. A state $q$ is denoted by a labeled circle. The label denotes the state $Id$. A transition from $q_p$ to $q_c$ is denoted by an edge from $q_p$ circle to $q_c$ circle. The label of the edge $(q_p, q_c)$ in (a) and (b) is $a$ where $\delta^{A^{pc}} (q_p, a) = q_c$ exists. An edge $(q_p, q_c)$ with label $\Sigma$ denotes transitions $\delta^{A^{pc}} (q_p, a) = q_c$ where $a \in \Sigma$. The start state $q_0^{A^{pc}}$ is denoted by an incoming edge. The accepting states in $F^{A^{pc}}$ are denoted by double circles.
5.4.7 Removal of unselected nodes

Algorithm 15 describes the function $\text{RemoveUnselected}$. If $\rho_{\text{STATwig}}(A_{\text{Tree}}, \text{history})(q_v) = \emptyset$ then we know from Lemma 5.2 that $q_{\text{Selecting}} \notin \rho_{\text{STATwig}}(T)(u)$ where $P(u) = q_v$ and $T$ is the present-future-subtree$(D_T, \text{extracted})$. If $q_v = q_v^{\text{self}}$ then $u = v$. Therefore, $R_v$ is filtered from the extracted nodes. Otherwise, $q_v = q_v^{\text{descendent}}$ and a future-node $u$, where $P(u) = q_v^{\text{descendent}}$, is not part of any twig pattern solution in $T$. Therefore, node $u$ should not be extracted. In this case, the algorithm removes the entries $P(\text{start}_u)$ and $P(\text{end}_u)$. As a result, the execution of the $\text{ExtractNodes}$ function in the next iteration (see Algorithm 16) will filter the label $R_v$.

\begin{algorithm}
\caption{The $\text{RemoveUnselected}$ algorithm}
\begin{algorithmic}
\Input $\text{extracted}, P, \rho_{\text{STATwig}}(A_{\text{Tree}}, \text{history}), \text{history}$
\Output $\text{num} : \mathbb{N}$
\State $\text{num} \leftarrow 0$
\ForAll $\rho_{\text{STATwig}}(A_{\text{Tree}}, \text{history})(q_v) = \emptyset$
\State remove $P(\text{pos})$ were $\text{pos} \in \text{positions}(P, q_v)$
\If $q_v = q_v^{\text{self}}$
\State remove $R_v$ from $\text{extracted}$
\State remove $\text{history}(q_v^{\text{self}})$
\State $\text{num} \leftarrow \text{num} + 1$
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

Example 5.11 The $\rho_{\text{STATwig}}(A_{\text{Tree}})$, which is constructed in the first iteration in the 1-prediction $\text{TwigTA}$ operation on the twig pattern ‘//a//l//c//b//e//d’ and on $D_T$ in Example 3.1, is described in Example 5.10. $\rho_{\text{STATwig}}(A_{\text{Tree}})(q_1^{\text{self}}) = \emptyset$. Therefore, Algorithm 15 removes the label $(1, 6, 2)$ from the $\text{extracted}$ set. After the removal, we get $\text{extracted} = \{(-\infty, \infty, 0), (0, 29, 1), (3, 4, 4), (12, 13, 5), (14, 15, 5)\}$. $\rho_{\text{STATwig}}(A_{\text{Tree}})(q_1^{\text{descendent}}) = \emptyset$. Therefore, $P(1) = P(6) = q_1^{\text{self}}$ and $P(2) = P(5) = q_1^{\text{descendent}}$ are removed from $P$. After the removal the partition is: $P(0) = P(29) = q_0^{\text{self}}, P(3) = P(4) = q_3^{\text{self}}, P(7) = P(8) = \ldots = P(11) = q_0^{\text{descendent}}, P(12) = P(13) = q_2^{\text{self}}, P(14) = P(15) = q_2^{\text{self}}, P(16) = P(17) = \ldots = P(28) = q_0^{\text{descendent}}$.

5.4.8 Nodes extraction from streams

Algorithm 16 describes the function $\text{ExtractNodes}$. This function extracts a maximum of $K$ present-nodes or future-nodes from each stream. The size of the $\text{extracted}$ set is...
where $H$ is the height of the stored XML tree $T$. $H$ bounds the number of past-nodes in the extracted set. The function ExtractNodes replaces the nodes that were removed in the previous iteration. For each extracted node $R_v$, this function checks if $P$$(\text{start}_v)$ exists. If $P$$(\text{start}_v)$ does not exist, then, it was removed by Algorithm 15 because it was not part of any twig solution. In this case, $R_v$ is filtered. The algorithm returns the number of streams which contain future-nodes.

Algorithm 16: The ExtractNodes algorithm.

Input: $D^T, \text{extracted}, P, \text{min\_present}, K$

Output: $\text{streams} : \mathbb{N}$

begin
  $\text{streams} \leftarrow 0$;
  forall $a \in \Sigma^{T_{\text{twig}}}$ do
    if $D(a) \neq ()$ then
      $\text{streams} \leftarrow \text{streams} + 1$;
      $n = |\{R_v \in \text{extracted}|\text{label}^T(v) = a \text{ and } \text{start}_v \geq \text{min\_present}\}|$;
    while $D^T(a) \neq ()$ do
      Let $R_v$ be head($D^T(a)$);
      if exists $P$$(\text{start}_v)$ then
        if $n = K$ then
          break;
        extracted_nodes $\leftarrow$ extracted_nodes $\cup \{R_v\}$;
        $n \leftarrow n + 1$;
        $D^T(a) \leftarrow \text{tail}(D^T(a))$;
  end
end

Example 5.12: Nodes removal in the first iteration of the 1-prediction TwigTA, which operates on the twig pattern ‘//a[c]/b[e]/d’ and on $D^T$ in Example 3.1 is described in Example 5.11. The second iteration starts by calling Algorithm 16. The input is the extracted set and the partition $P$ in Example 5.11 $\text{min\_present} = 0$ and $\text{min\_future}(D^T) = 2$ and, therefore, (0, 29, 1) is a present-node and nodes (3, 4, 4), (12, 13, 5), (14, 15, 5) are future-nodes. $D(b)$ is the only stream where $n = 0 < 1 = K$. The algorithm extracts head$(D^T q(b)) = (2, 5, 3)$. But $P(2)$ and $P(5)$ were removed in the previous iteration. Therefore, (2, 5, 3) is filtered. Next, the function ExtractNodes extracts the head$(D^T q(b)) = (7, 18, 2)$. After the extraction we get extracted $= \{(-\infty, \infty, 0), (0, 29, 1), (3, 4, 4), (7, 18, 2), (12, 13, 5), (14, 15, 5)\}$. Algorithm 16 returns $\text{streams} = 1$ as the number of streams.
5.4.8.1 Using a DB index

Fetching and filtering an unselected label \( R_v \) when \( P(\text{start}_v) \) does not exist is inefficient. When the streams are used with a DB index, like XB-tree index [106], Algorithm 16 can skip fetching of unselected labels from the streams. We sketch the change in Algorithm 16 that is needed to support the use of a DB index. Instead of iterating over a stream by \( \text{head}(D^T(a)) \) as done in Algorithm 16, we use the index to select the minimal \( R_v \) where \( \text{start}_v \geq \min(\bigcup_{q \in Q_A^{Tree}\text{ positions}(P,q)}) \). The modified Algorithm 16 receives \( T^{\text{Prediction}} \) as an input.

Example 5.13 In Example 5.12, the ExtractNodes in the second iteration of the 1-prediction TwigTA, which operates on the twig pattern `//a//c/b//e//d` and the \( D^T \) in Example 3.1, filters encoded label \((2, 5, 3)\) from \( D^T(b) \). In this example, the encoded label \((2, 5, 3)\) is not fetched from \( D^T(b) \). The state \( q_0^{\text{descendent}} \) is the only state that satisfies \( b \in \text{positions}(P,q_0^{\text{descendent}}) = \{7, 8, \ldots, 11, 15, \ldots, 28\} \). Therefore, \( Q_A^{Ttree} = \{q_0^{\text{descendent}}\} \) and \( \min(\text{positions}(P,q_0^{\text{descendent}})) = 7 \). Fetching of \((2, 5, 3)\) is skipped because \( 2 < 7 \) and the label \((8, 17, 3)\) is fetched from \( D^T(b) \).

5.4.9 Intermediate paths construction

Algorithm 17 describes the construction of intermediate paths. The \( \text{IntermediatePaths} \) function constructs intermediate paths \( p \), which compose the \( \text{present-subtree}(D^T, \text{extracted}) \) and ends in \( R_v \) where \( v \) is a present-node. \( \text{IntermediatePaths} \) is a recursive function that receives the states \( q_{v_p}^{\text{self}} \in Q_A^{Ttree} \) and \( q_p \in Q_A^{Ttwig} \) as inputs. The function finds transitions \( \delta_A^{Ttwig}(q_p,p_c) = q_c \) from \( q_p \in \rho_{STA}^{Ttwig}(T^{tree,\text{history}})(q_{v_p}^{\text{self}}) \) to \( q_c \in \rho_{STA}^{Ttwig}(T^{tree,\text{history}})(q_{v_p}^{\text{self}}) \) where \( v_c \) is a child of \( v_p \) in the \( \text{present-subtree}(D^T, \text{extracted}) \). Therefore, \( q_{v_c}^{\text{self}} \) is a child of \( q_{v_p}^{\text{self}} \) in \( T^{\text{Prediction}} \). The algorithm applies the recursion to \( q_{v_c}^{\text{self}} \in Q_A^{Ttree} \) and \( q_c \in Q_A^{Ttwig} \). When \( q_c \in F^{Ttwig} \) then \( R_{v_c} \) is concatenated to the path \( p \). If \( v_c \) is a present-node then path \( p \) is added to \( \text{paths}(q_c) \).
Algorithm 17: The IntermediatePaths algorithm.

**Input:**\(q_{self}^{p} \in Q^{T_{\text{Tree}}}, q_{p} \in Q^{T_{\text{Twig}}}, Path^{T_{p}}, \rho^{STAS_{3}^{T_{\text{Twig}}}(A^{T_{\text{Tree}}, history})}, A_{pc}^{T_{\text{Twig}}}, T^{\text{Prediction}}, D^{T}, \text{min}_{-} \text{present}, \text{paths}\)

**begin**

forall \(q_{c} \in \rho^{STAS_{3}^{T_{\text{Twig}}}(A^{T_{\text{Tree}}, history})}(q_{self}^{p})\) where \(q_{c}^{p} \in \text{children}(T^{\text{Prediction}}, q_{self}^{p})\)
do

if \(\text{start}_{v_{c}} < \text{min}_{-} \text{future}(D^{T})\) then

if exists \(\delta^{A_{pc}^{T_{\text{Twig}}}}(q_{p}, q_{c}) = q_{c}\) then

if \(q_{c} \in F^{A_{pc}^{T_{\text{Twig}}}}\) then

Concatenate \(q_{c}\) to \(p\);

if \(\text{start}_{v_{c}} \geq \text{min}_{-} \text{present}\) then

Concatenate \(p\) to \(\text{paths}(q_{c})\);

IntermediatePaths\(\left(q_{c}^{p}, q_{c}, p, \rho^{STAS_{3}^{T_{\text{Twig}}}(A^{T_{\text{Tree}}, history})}, A_{pc}^{T_{\text{Twig}}}, T^{\text{Prediction}}, D^{T}, \text{paths}\right)\);

**end**

---

**Example 5.14** Figure 5.10 describes \(T^{\text{Prediction}}\) and \(\rho^{STAS_{3}^{T_{\text{Twig}}}(A^{T_{\text{Tree}}, history})}\) that are constructed in the fourth iteration of the 1-prediction TwigTA operation on the twig pattern ‘//a[//c]/b[//e]/d’ and \(D^{T}\) that was described in Example 3.1. The \(A_{pc}^{T_{\text{Twig}}}\) that is constructed from the twig pattern ‘//a[//c]/b[//e]/d’ was described in Fig. 5.9(b). In the fourth iteration, \(\text{min}_{-} \text{present} = 8\) and \(\text{min}_{-} \text{future}(D^{T}) = 19\). Table 5.3 describes the run of Algorithm 17 in the fourth iteration of the TwigTA. The Algorithm adds the following paths: \(((8, 17, 3))\) to \(\text{paths}(q_{d}), ((8, 17, 3), (11, 16, 4))\) to \(\text{paths}(q_{b}), ((0, 29, 1), (9, 10, 4))\) to \(\text{paths}(q_{c}), ((8, 17, 3), (9, 10, 4))\) to \(\text{paths}(q_{e}), ((8, 17, 3), (11, 16, 4), (12, 13, 5))\) to \(\text{paths}(q_{d}), ((8, 17, 3), (11, 16, 4), (14, 15, 5))\) to \(\text{paths}(q_{c})\). The Algorithm does not add \(((0, 29, 1))\) to \(\text{paths}(q_{b})\), because \(0 < 8 = \text{min}_{-} \text{present}\) and therefore, \((0, 29, 1)\) is a past-node that was outputted in previous iterations.

\(\square\)
Figure 5.10: $T^{\text{Prediction}}$ and $\rho^{\text{STA}_{\text{Twig}}}(A^{\text{Tree}, \text{History}})$, which were constructed in the fourth iteration of the 1-prediction TwigTA operation on the twig pattern ‘//a[///c]/b[///e]/d’ and on $DT$, which was described in Example 3.1. A circle denotes a node $q \in V^{T^{\text{Prediction}}}$. The label of $q$ includes two lines. The top line has the format ‘$q; \text{label}^{T^{\text{Prediction}}}(q)$’ where $\text{label}^{T^{\text{Prediction}}}(q) = a_1, \ldots, a_n$. The bottom line denotes $\rho^{\text{STA}_{\text{Twig}}}(A^{\text{Tree}, \text{History}})(q) = q_1, \ldots, q_n$. 

Part III: Querying of XML in a DB
Table 5.3: Description of the run of Algorithm 17 in the fourth iteration of the 1-prediction TwigTA that operates on the twig pattern ‘//a//c//b//e//d’ and on \( D^T \) that was described in Example 3.7. The Stack column is the stack of the recursive function calls in Algorithm 17. Each function call is identified by the pair \((q_{self}, q_p)\) where \( q_{self} \in QA^{Tree}_T \) and \( q_p \in QA^{Twig}_T \) are the input parameters with whom the function is being called. The Intermediate path column is the variable \( p \) when the function execution ends. The Paths column describes \( q \in P^{\mathcal{A}_{pc}^{Twig}} \) where \( paths(q) \) is being concatenated with \( p \) during function execution.

## 5.4.10 Removal of present-nodes

Algorithm 18 describes the removal of the present-subtree\((D^T, \text{extracted})\) nodes that constructs the intermediate paths in Algorithm 17. The algorithm removes all the past-nodes and present-nodes that are not ancestors of future-nodes. If \( \text{positions}(P, q_v^{\text{descendent}}) = \emptyset \) then all the descendants of \( v \) are either past-nodes or present-nodes. Therefore, \( v \) is not an ancestor of any future-node. In this case, the function removes \( R_v \) from extracted and removes \( q_v^{\text{self}} \) from history and partition functions. The parent of \( q_v^{\text{self}} \) in \( T^{\text{Prediction}} \) is \( q_v^{\text{op}} \). If node \( v \) is in \( \text{present-subtree}(D^T, \text{extracted}) \) then node \( v_p \) is also in \( \text{present-subtree}(D^T, \text{extracted}) \) because \( \text{start}_v > \text{start}_{v_p} \). If \( v_p \) is not removed then TwigTA adds the state \( q \in p^{\mathcal{STAG}^{Twig}_S}(ATree, \text{history}) (q_v^{\text{self}}) \) to history\((q_v^{\text{self}})\).
Algorithm 18: The RemovePresent algorithm.

\[
\text{Input: } D^T, \text{extracted, } P, \rho_{\text{STA}_3^{\text{Twig}}, (\text{ATree, history})}, \text{history, } T^{\text{Prediction}}
\]
\begin{algorithm}
\begin{algorithmic}
\State forall \( R_v \in \text{extracted} \) where \( \text{start}_v < \text{min\_future}(D^T) \) do
\If{\( \text{positions}(P, q_v^{\text{descendent}}) = \emptyset \)}
\State remove \text{history}(q_v^{\text{self}});
\State remove \( R_v \) from \text{extracted};
\State remove \( P(\text{start}_v) \) and \( P(\text{end}_v) \);
\State Let \( q_v^{\text{self}} \leftarrow \text{parent}(T^{\text{Prediction}}, q_v^{\text{self}}) \);
\If{\( \text{positions}(P, q_v^{\text{descendent}}) \neq \emptyset \)}
\State add \( q \in \rho_{\text{STA}_3^{\text{Twig}}, (\text{ATree, history})}(q_v^{\text{self}}) \) to \text{history}(q_v^{\text{self}});
\EndIf
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

Example 5.15 Example 5.14 describes the intermediate paths construction in the fourth iteration of the 1-prediction TwigTA operation on the twig pattern ‘//a//c]/b[//e]/d’ and on \( D^T \) that was described in Example 3.1. The extracted nodes in the fourth iteration are \( \{ (0, 29, 1), (8, 17, 3), (9, 10, 4), (11, 16, 4), (12, 13, 5), (14, 15, 5) \} \). Both \( \rho_{\text{STA}_3^{\text{Twig}}, (\text{ATree, history})} \) and \( T^{\text{Prediction}} \) were described in Fig 5.10. \( \text{min\_future}(D^T) = 19 \) and \( \text{min\_present} = 8 \). The partition \( P \) is \( P(0) = P(29) = q_0^{\text{self}}, P(19) = \ldots = P(28) = q_0^{\text{descendent}}, P(8) = P(17) = q_8^{\text{self}}, P(9) = P(10) = q_9^{\text{self}}, P(11) = P(16) = q_1^{\text{self}}, P(12) = P(13) = q_1^{\text{self}}, P(14) = P(15) = q_4^{\text{self}} \). \( \text{positions}(P, q_0^{\text{descendent}}) = \text{positions}(P, q_0^{\text{descendent}}) = \text{positions}(P, q_1^{\text{descendent}}) = \text{positions}(P, q_4^{\text{descendent}}) = \emptyset \). Therefore, the labels \( (8, 17, 3), (9, 10, 4), (11, 16, 4), (12, 13, 5), (14, 15, 5) \) are removed. \( \text{positions}(P, q_0^{\text{descendent}}) = \{ 19, \ldots, 28 \} \). Therefore, label \( 0, 29, 1 \) is not removed. Node \( q_8^{\text{self}} \), which is removed, is a child of \( q_0^{\text{self}} \), which is not removed, in \( T^{\text{Prediction}} \). Therefore, the states \( \{ q_a, q_u_c \} = \rho_{\text{STA}_3^{\text{Twig}}, (\text{ATree, history})}(q_8^{\text{self}}) \) are added to \text{history}(q_0^{\text{self}}) \). 

5.4.11 Output of twig pattern solutions

This section sketches the MergeAllPathSolutions function in Algorithm 11. The intermediate paths, which are constructed in the 1-prediction TwigTA operation on the twig pattern ‘//a//c]/b[//e]/d’ and on \( D^T \) that was described in Example 3.1, are given in the Table 5.4. There are three twig pattern solutions. Two of the solutions have roots in node \( (0, 29, 1) \). The other solution has a root in node \( (8, 17, 3) \). We see in Table 5.4 that the intermediate paths are not ordered according to the traversal order.
For example, paths(q_b) starts with path \((8, 17, 3)\) and only then it moves to path \(((0, 29, 1), (19, 28, 2))\). The first action that \textit{MergeAllPathSolutions} performs is to sort the intermediate paths as shown in Table 5.5. In Table 5.5, the intermediate paths, which are in paths(q_b), paths(q_d) and paths(q_e), are sorted by the traversal order. The second action that \textit{MergeAllPathSolutions} performs is application of the merge-join algorithm (for more describes, see [106]) to the sorted intermediate paths. The merge-join traverses all the lists of intermediate paths and joins paths with common prefixes. In this way, three solutions are returned for the twig pattern ‘//a[/c]/b[/e]/d’ and for \(D^T\) that was described in Example 3.1. Table 5.6 describes the three solutions.

| \(q_a\)     | \(((0, 29, 1)), ((8, 17, 3))\) |
| \(q_b\)     | \(((8, 17, 3), (11, 16, 4)), ((0, 29, 1), (19, 28, 2))\) |
| \(q_c\)     | \(((0, 29, 1), (3, 4, 4)), ((0, 29, 1), (9, 10, 4)), ((8, 17, 3), (9, 10, 4))\) |
| \(q_d\)     | \(((8, 17, 3), (11, 16, 4), (12, 13, 5)), ((0, 29, 1), (19, 28, 2), (20, 21, 3))\) |
| \(q_e\)     | \(((8, 17, 3), (11, 16, 4), (14, 15, 5)), ((0, 29, 1), (19, 28, 2), (24, 25, 5))\) |

Table 5.4: Description of \(paths: Q^{A_{Twig}} \mapsto (Path^T_1, \ldots, Path^T_n)\) in Algorithm 11.

The first column is the selecting state \(q \in S\) and the second column is the \(path(q) = Path^T_1, \ldots, Path^T_n\). A \(Path^T_i\) is denoted by \((R_{v_1}, \ldots, R_{v_n})\).

| \(q_a\)     | \(((0, 29, 1)), ((8, 17, 3))\) |
| \(q_b\)     | \(((0, 29, 1), (19, 28, 2)), ((8, 17, 3), (11, 16, 4))\) |
| \(q_c\)     | \(((0, 29, 1), (3, 4, 4)), ((0, 29, 1), (9, 10, 4)), ((8, 17, 3), (9, 10, 4))\) |
| \(q_d\)     | \(((0, 29, 1), (19, 28, 2), (20, 21, 3)), ((8, 17, 3), (11, 16, 4), (12, 13, 5))\) |
| \(q_e\)     | \(((0, 29, 1), (19, 28, 2), (24, 25, 5)), ((8, 17, 3), (11, 16, 4), (14, 15, 5))\) |

Table 5.5: Description of \(paths: Q^{A_{Twig}} \mapsto (Path^T_1, \ldots, Path^T_n)\) in Algorithm 11 where \(Path^T_1, \ldots, Path^T_n\) are sorted by the \(DFS\) traversal order. The first column is the selecting state \(q \in S\) and the second column is the \(path(q) = Path^T_1, \ldots, Path^T_n\). A \(Path^T_i\) is denoted by \((R_{v_1}, \ldots, R_{v_n})\).
Table 5.6: Description of three solutions: $\text{Solution}^T_1, \ldots, \text{Solution}^T_3$ from Algorithm 11. The second, third and fourth columns are $\text{Solution}^T_1$, $\text{Solution}^T_2$ and $\text{Solution}^T_3$, respectively. The first column is the selecting state $q \in S$. $Row_{i,j} = R_v, 2 \leq j \leq 4$, and $R_v = \text{Solution}^T_{j-1}(Row_{i,1})$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_a$</td>
<td>(0, 29, 1)</td>
<td>(0, 29, 1)</td>
<td>(8, 17, 3)</td>
</tr>
<tr>
<td>$q_b$</td>
<td>(19, 28, 2)</td>
<td>(19, 28, 2)</td>
<td>(11, 16, 4)</td>
</tr>
<tr>
<td>$q_c$</td>
<td>(3, 4, 4)</td>
<td>(9, 10, 4)</td>
<td>(9, 10, 4)</td>
</tr>
<tr>
<td>$q_d$</td>
<td>(20, 21, 3)</td>
<td>(20, 21, 3)</td>
<td>(12, 13, 5)</td>
</tr>
<tr>
<td>$q_e$</td>
<td>(24, 25, 5)</td>
<td>(24, 25, 5)</td>
<td>(14, 15, 5)</td>
</tr>
</tbody>
</table>

5.4.12 Example of a TwigTA run

This section describes the $1$-prediction TwigTA operation on the twig pattern ‘//a[//c]/b[//e]/d’ and on $D^T$ that were described in Example 3.1. The TwigTA algorithm traverses the document in six iterations (from $a$ to $f$). The $T_{\text{prediction}}$ is reconstructed in every iteration. Figures 5.11(a) to 5.11(f) describe the $T_{\text{prediction}}$ that is constructed in iterations $a$ to $f$, respectively. Table 5.7 describes the TwigTA data in different iterations.

<table>
<thead>
<tr>
<th>Step</th>
<th>Filtered Nodes</th>
<th>Extracted Nodes</th>
<th>Removed Unselected Paths</th>
<th>Intermediate Paths</th>
<th>Removed Present</th>
<th>$\text{min}_{\text{present}}$</th>
<th>$\text{min}_{\text{future}(D^T)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0, 1, 3</td>
<td>12, 14</td>
<td>1</td>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td></td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>11</td>
<td>(0), (0, 3)</td>
<td>3</td>
<td>(0, 9), (8), (8, 9), (8, 11), (8, 11, 12), (8, 11, 14)</td>
<td>8</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>8, 9</td>
<td>22</td>
<td>19</td>
<td></td>
<td>0</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>19, 20, 22, 24</td>
<td>(0, 19), (0, 19, 20), (0, 19, 24)</td>
<td>19</td>
<td></td>
<td>0, 19, 20, 24</td>
<td>19</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 5.7: TwigTA internal data from the run over the XML document in Fig. 3.2 and from the twig pattern in Fig. 2.5. In each iteration step, the table describes the following data: the node labels that Algorithm 16 filters, the labels that Algorithm 16 adds to extracted, the node labels that Algorithm 15 removes, the intermediate paths that Algorithm 17 constructs, the node labels that Algorithm 18 removes, the $\text{min}_{\text{present}}$ and the $\text{min}_{\text{future}(D^T)}$ in Algorithm 11. To enable a compact summary of the TwigTA run, a node label $R_v$ is denoted by $\text{start}_v$. 

Part II Querying of XML in a DB
Figure 5.11: The $T^{\text{Prediction}}$ structure in the $1$-prediction $\text{TwigTA}$ run over XML document in Fig. 5.2 and the twig pattern in Fig. 2.5. (a) - (f) describe the $T^{\text{Prediction}}$ structure in iterations $a$-$f$, respectively. A gray and white circles denote $q_v^{self}$ and $q_v^{descendent}$ nodes, respectively. The labels inside the nodes have the syntax $'q; label^{T^{\text{Prediction}}} (q)$ where $label^{T^{\text{Prediction}}} (q) = label_1, \ldots, label_n$.'
5.4.13 Time analysis

The core operation in the TwigTA algorithm is the $STA^S_{AT}$ operation in Algorithm 1. Section 4.5.2 bounds the time of the $STA^S_{AT}$ operation, which is also the runtime of the $STA^S_{AT}$ operation, to be $O(|ATree| \cdot |V^{Twig}|)$. Next, we compute $|ATree|$. From the construction of the partition $P$ in section 5.4.4, we know that $|Q^{Tree}| \leq 2 \cdot |extracted|$ because each $R_v \in extracted$ denotes a maximum of two states: $q_v^{self}$ and $q_v^{descendent}$. $|extracted| \leq H + K \cdot |V^{Twig}|$ where $H$ is the height of the tree $T$ that is stored in $D_T$. Therefore, $|Q^{Tree}| \leq 2 \cdot (H + K \cdot |V^{Twig}|)$. For the $ATree$ of $TwigTA$, which is constructed by the $TwigTA$ algorithm, $|ATree| \leq |\Sigma_t^{Twig}| \cdot |Q^{Tree}|$. Therefore, $|ATree| \leq 2 \cdot |\Sigma_t^{Twig}| \cdot (H + K \cdot |V^{Twig}|)$ and the $STA^S_{AT}$ operation in Algorithm 1 is bounded by $O((2 \cdot |\Sigma_t^{Twig}| \cdot (H + K \cdot |V^{Twig}|)) \cdot |V^{Twig}|) = O(K \cdot |\Sigma_t^{Twig}| \cdot |V^{Twig}|^2) = O(K \cdot |V^{Twig}|^3)$. Therefore, the runtime is bounded by $O(|D_T| \cdot K \cdot |V^{Twig}|^3)$ where $|D_T|$ is the size of the DB.

5.4.14 1-prediction TwigTA vs.TwigStack

The TwigStack algorithm resembles the 1-prediction TwigTA algorithm. The main difference is in the twig pattern $Q$ that each algorithm uses. TwigTA uses the input twig pattern $Q$. TwigStack generalizes $Q$ into $Q'$ that is matched by more intermediate paths than $Q$. Given $Q = (V^{Twig}, E^{Twig}, label^{Twig}, type^{Twig})$ TwigStack constructs $Q' = (V^{Twig}, E^{Twig}, label^{Twig}, type^{Twig}_{A-D})$. If $type^{Twig}(e)$ exists in $Q$ then $type^{Twig}_{A-D}(e) = A-D$ exists in $Q'$.

Instead of constructing $ATree$ from $T^{Prediction}$ and applying the STA to the $ATree$, the TwigStack removes labels, which are part of any twig solutions, by recognizing a simple pattern in the extracted labels using the map $MAP$ and $T^{Prediction}$. TwigStack constructs $T^{Prediction}$ from $R_v$ nodes. Since it is a 1-prediction algorithm, each twig node $v^{Twig}_i \in V^{Twig}$ is mapped into a single $R_v_i$ in extracted. We denote this map by $MAP : V^{Twig} \rightarrow Nodes_T$. In each iteration, TwigStack checks whether to construct an intermediate path for $R_v \in extracted$ where $start_v = min_{present}$. At each iteration, TwigStack analyzes the $T^{Prediction}$ subtree whose root is $R_v$. The pattern of $Q'$ guarantees that the $T^{Prediction}$ subtree satisfies the following: (i) $MAP(v^{Twig}_v) = R_v$, $start_v = min_{present}$. $MAP(v^{Twig}_c) \in children(T^{Prediction}, MAP(v^{Twig}_c))$, $v^{Twig}_c \in children(Q', v^{Twig}_c)$. In other words, the A-D node relationship in $Q'$ is
mapped into A-D node relations in $T^{Prediction}$. (ii) Each of the twig vertices $v^{'Twig}$ recursively satisfies the first property. Labels, which do not fit this pattern, are removed. The $history$ function is not needed for $TwigStack$ because once an intermediate path is constructed for a node $v$, then this path is part of a solution for the twig pattern. Therefore, all additional intermediate paths, which will be constructed next and will contain the node $v$, will also be part of a twig pattern solution. Therefore, the future intermediate paths will not need the $history$ function.

**Example 5.16** Figure 5.12 describes the $TwigStack$ operation on an XML document, which was described in Fig. 3.2 and by the twig pattern in Fig. 2.5. A comparison between the $TwigJoin$ and the $TwigTA$ algorithms exhibits some resemblance. Figure 5.12(b) describes the iterative $TwigStack$ prediction-tree when it constructs the path $(((0, 29, 1), (3, 4, 4))$. Figure 5.11(c) describes the $TwigTA$ prediction-tree in the iteration when it outputs the same path $(((0, 29, 1), (3, 4, 4))$. We can see that both prediction-trees are similar. The difference is in $R_v = (8, 17, 3)$. $TwigTA$ has not extracted yet label $(8, 17, 3)$. $TwigTA$ extracted less symbols because it does not construct an intermediate path in each iteration. It outputs the whole present-subtree($D^T$, extracted) when it must. Figure 5.12(e) describes the $TwigStack$ prediction-tree when it constructs the intermediate path $(((0, 29, 1), (9, 10, 4))$. Figure 5.11(d) describes the $TwigTA$ prediction-tree when it constructs the same intermediate path $(((0, 29, 1), (9, 10, 4))$. We can see that both prediction-trees are similar. The $TwigStack$ prediction-tree includes two additional labels $(7, 18, 2)$ and $(22, 27, 3)$. $TwigStack$ extracted the label $(22, 27, 3)$. $ TwigStack$ extracted the label $(7, 18, 2)$ because it was not selected at iteration $b$ (see Fig. 5.11(b)). $TwigStack$ mistakenly constructed an intermediate path for the label $(7, 18, 2)$ at iteration $c$ (see Fig. 5.12(c)) because it matches the twig pattern $Q'$. This example demonstrates the differences between the two algorithms.
Figure 5.12: The TwigStack operation on an XML document, which is described in Fig. 3.2 and on the twig pattern in Fig. 2.5. The boxes denote vertices in the stacks. The trapezoidal boxes denote the labels $R_v$ where $\text{start}_v = \text{min}_{\text{present}}$. The circles denote label $R_v$, $R_v \in V_T^{\text{Prediction}}$. The labels inside the vertices have the syntax ‘$\text{start}_v; \text{label}^{T}(v)$’.

### 5.5 Experimental results

In this section, we present experimental results by applying the TwigTA to XML documents with different characteristics to demonstrate its applicabilities to handle different XML documents types. Next, we conduct a comprehensive study on the performances of a twig pattern processing that are based on different prediction sizes. Our experimental results show that TwigTA outperforms the TwigStack and iTwigJoin based approaches.
5.5.1 Experimental settings for XML datasets

We implemented all the algorithms in Java 1.5. All our experiments were performed on a PC with 2.4GHz Pentium 4 processor and 1024MB RAM running Windows XP. We used the TreeBank [9] and the synthetic XMark [10] datasets in our experiments: (1) XMark XML dataset is synthetic and was generated by an XML data generator. It contains auction site information. Its DTD is recursive. (2) The TreeBank dataset is obtained from the University of Washington XML repository. The DTD of Treebank is also recursive. TreeBank consists of encrypted English sentences that were taken from the Wall Street Journal and tagged with speech parts. The deep recursive structure of this data makes it ideal for experiments with twig pattern matching algorithms. The datasets main characteristics are given in Table 5.8. We select the above two XML datasets because they represent two important types of data: XMark is more “information oriented” and has many repetitive structures and fewer recursions whereas TreeBank has an inherent tree structure because it encodes natural language parsing trees.

<table>
<thead>
<tr>
<th></th>
<th>XMark</th>
<th>Treebank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>113MB</td>
<td>77MB</td>
</tr>
<tr>
<td>Nodes</td>
<td>2.0 million</td>
<td>2.4 million</td>
</tr>
<tr>
<td>Tags</td>
<td>77</td>
<td>251</td>
</tr>
<tr>
<td>Max Depth</td>
<td>12</td>
<td>36</td>
</tr>
<tr>
<td>Average Depth</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.8: The characteristics of the XML Datasets that were used for testing the performances of TwigTA, TwigStack and iTwigJoin algorithms

5.5.1.1 Queries

We select representative queries (shown in Table 5.9) which cover several classes of twig pattern queries. The selected queries on the XMark dataset include: (1) A Path query (XMark1). (2) An A-D only query (XMark2). (3) A P-C only query (XMark3). (4) 1-branchnode (that is neither A-D nor P-C) query (XMark4). (5) A Query (XMark5) which does not fall in the above four types. The selected queries for the TreeBank dataset include: (1) An A-D only query (Tree1). (2) Two P-C only queries (Tree2 and Tree3). (3) Two queries (Tree4 and Tree5) which do no fall in the above two categories.
5.5.2 Performance measurements

We implemented three XML twig join algorithms: *TwigTA* (Algorithm 11), *TwigStack* [106] and *iTwigJoin* [138] using the file system as a simple storage engine. The reason that we choose to compare *TwigTA* to these two algorithms is that *TwigStack* and *iTwigJoin* are proved to be efficient for different classes of twig patterns. The classes are: 1. Only P-C twig pattern for which $\text{type}_{\text{Twig}}(e) = \text{P-C}$ for every $e \in E_{\text{Twig}}$; 2. Only A-D twig pattern for which $\text{type}_{\text{Twig}}(e) = \text{A-D}$ for every $e \in E_{\text{Twig}}$; 3. 1–branchnode twig pattern which has only one node $v \in V_{\text{Twig}}$ such that $|\text{children}(T_{\text{Twig}}, v)| > 1$. *TwigStack* performance is optimal for processing only A-D twig patterns. *iTwigJoin* with a prefix-path streaming (PPS) is optimal when processing only A-D, only P-C and 1–branchnode twig patterns. *TwigTA* processing is compared with the optimal processing of different patterns by *TwigStack* and *iTwigJoin*. The results show that *TwigTA* is near optimal on all types of queries.

Another difference between the algorithms is the use of structural indexing by *iTwigJoin*. The use of index reduces the number of produced intermediate paths and the number of scanned elements to near optimal but *iTwigJoin* performance can be degraded by the preprocessing of the index structure. *TwigTA* does not use a structural-index. The results show that unlike *iTwigJoin*, the *TwigTA* processing time is similar to all query types.

We consider the following performance metrics to compare between the performance of twig pattern matching algorithms that are based on three statistics: (1) The

Table 5.9: Queries used in the *TwigTA* experiments.
number of scanned nodes (scanned bytes). (2) The number of produced intermediate paths. (3) The running time. Figure 5.13 compares between the performance of these algorithms for Xmark and Treebank datasets.

Figure 5.13: Comparison between the performance of TwigTA and other algorithms

As we can see from Fig. 5.13, TwigTA can prune up to 30% from the number of the scanned nodes labels in the processing of XMark dataset (Fig. 5.13(d)). iTwigJoin prunes 40% of the irrelevant data. In the processing of Treebank dataset (Fig. 5.13(a)), we can see that TwigTA prunes up to 99% of the irrelevant data. iTwigJoin prunes 77% of the irrelevant data. The 99% pruning is achieved for Treemak5 query which

Part II Querying of XML in a DB
does not return any solution. Due to its accuracy, TwigTA extracts only 8 nodes and filters the rest.

With respect to the numbers of intermediate paths output by the different algorithms, TwigTA avoids redundant intermediate paths that were produced by TwigStack. For the XMark dataset (Fig. 5.13(e)), the reduction ratio goes up to 25% (XMark5) and for Treebank (Fig. 5.13(b)) as high as 1:98 (Tree2). iTwigJoin reduction ratio goes up to 25% (XMark5) and for Treebank as high as 1:2750 (Tree2).

In terms of running time, TwigTA is about ten times slower than TwigStack. The implementation of the algorithm is not optimized and currently it works without utilizing the indexing method. For XMark (Fig. 5.13(f)), iTwigJoin was always faster than TwigStack. For Treebank (Fig. 5.13(c)), iTwigJoin was faster for a small number of streams. For large number of streams, the preprocessing of the structural-index can take about 30 minutes! 30 times more than TwigTA processing time. In this case, preprocessing of the structural-index takes more time than the twig pattern processing itself.

5.5.2.1 $K > 1$ Performance Measures

In this section, we compare between the performance of the $K$-prediction TwigTA algorithm for different prediction sizes $K$. We check the impact of increasing the prediction size $K$ on the number of produced intermediate paths. We choose two queries, one from each dataset (XMark4, Treebank2). We experiment withTwigTA that uses $K = 1, 2, 4, 8, 16, 32, 64, 128, 256$. Figure 5.14 shows that as $K$ grows less intermediate paths are produced. Treebank2 reduces the intermediate paths from 254 ($K = 1$) to 58 ($K = 8$). XMark4 reduces the intermediate paths from 49505 ($K = 1$) to 46547 ($K = 8$). But a closer look reveals that a large prediction-size $K$ can damage the selection of intermediate paths. Treebank2 increases the intermediate paths from 58 ($K = 8$) to 119 ($K = 256$).
5.6 Conclusion and Future Work

XML twig pattern matching is a key issue in XML query processing. In this chapter, we introduced the TwigTA holistic join algorithm. The TwigTA is an efficient algorithm in terms of the requirements in using secondary memory. It uses a novel tree automata theory to address the twig pattern matching problem. We show that TwigTA improves the TwigStack algorithm’s performance while having a similar performance to the iTwigJoin algorithm. The iTwigJoin performance is achieved by combining the holistic join with a structural-index mechanism. As part of a future work, we plan to improve the TwigTA algorithm performance by combining it with an indexing mechanism.

We also plan to design a scalable version of TwigTA that extracts a varying number of nodes from a DB in each iteration. We hope that such a scalable version can construct no intermediate redundant paths for “real” XML data. If it is true then the scalable TwigTA could output the final twig pattern solutions in a single processing phase.

Currently, the run time of TwigTA is slower than other join algorithms. The core operation of the TwigTA is the \( STA_S^{\text{Twig}}(A^{\text{Tree}}) \) operation (See Chapter 4). We plan to design a faster implementation for this \( STA_S^{\text{Twig}}(A^{\text{Tree}}) \) operation.
Chapter 6
Holistic structural-index
6.1 Introduction

Given a structural-pattern $Q$ then structural-indexes provide quick jump points to nodes that satisfy $Q$. The existing structural-indexes for all DB Management Systems (DBMS) types are path-based. The input for the existing structural-indexes are structural-patterns $Q$ that describe the relationships between the nodes in the graph paths. Tree is a natural form to describe structural-patterns. Each branch in the tree defines a logical operator of either AND or OR. This chapter suggests a structural-index that processes a structural-pattern in a tree representation. We call this index type an holistic structural-index because it processes the whole structural-pattern together instead of decomposing it into paths. Our experiments show that the proposed holistic structural-index algorithm reduces the data retrieval by 50% on average in comparison to the performance of path-based structural indexes. The reduction becomes more evident when the structural-patterns are more complex.

6.1.1 The basic operation

Indexes maintain data structures that store references to the actual data. In order to describe a general structural-index, which fits relational, native and object-oriented DBMSs, we use in this chapter the following abstraction of a structural-index. A structural-index for a semi-structure data $T = (V^T, E^T, label^T)$ maintains two data structures: clusters and summary. The $clusters : C \mapsto 2^{V^T}$ function divides the tree nodes $V^T$ into clusters $C$. Nodes that share that same cluster $c \in C$ have a similar structure. The summary describes the relations between different clusters. The general structural-index operation is described in Fig. 6.1. The input for the structural-index algorithms is a structural-pattern $Q$ and a tree $T$. We denote the structural-indexes algorithms by $SelectIndex(T, Q)$. The $SelectIndex(T, Q)$ algorithms return selected nodes $v \in V^T$. The structural-index algorithms have offline and online phases. The offline phase ($CreateIndex(T)$ function in Fig. 6.1(a)) receives $T$ as an input and constructs the clusters and summary data structures that the index maintains. The offline phase is called once. Figure 6.1(b) describes the $CreateIndex(T)$ algorithm. The $CreateIndex(T)$ first constructs the summary from the data ($CreateSummary(T)$ in Fig. 6.1(b)) and then applies summary to $T$ in order to create the clusters data structure ($CreateClusters(T, summary)$ in Fig. 6.1(b)). The online phase ($PruneIndex...
function in Fig. 6.1(a)) prunes the data according to the structural-pattern $Q$. First, the online phase prunes the clusters $C_{\text{pruned}} \subseteq C$ in summary that matches $Q$ (PruneClusters functions in Fig. 6.1(c)). Then, the online phase (PruneNodes functions in Fig. 6.1(c)) returns the nodes that are mapped into the collection of the pruned clusters $\bigcup_{c \in C_{\text{pruned}}} \text{clusters}(c)$.

![Diagram](image)

(a) $\text{SelectIndex}(T, Q)$  (b) $\text{CreateIndex}(T)$  (c) $\text{PruneIndex}(\text{summary}, \text{clusters}, Q)$

Figure 6.1: (a) General flow of the $\text{SelectIndex}(T, Q)$ algorithm. (b) The $\text{CreateIndex}(T)$ algorithm, which is a routine in $\text{SelectIndex}(T, Q)$.
(c) The $\text{PruneIndex}(\text{summary}, \text{clusters}, Q)$ algorithm, which is a routine in $\text{SelectIndex}(T, Q)$. The ellipses denote the operations by the $\text{SelectIndex}(T, Q)$ algorithm. The boxes denote the data-structures that are the inputs and outputs of these operations.

This chapter defines three clusters properties. These clusters properties are the minimal requirements for structural-index. The $\text{SelectIndex}(T, Q)$ algorithms, which are presented in this chapter, satisfy the following clusters properties:

**Definition 6.1** The clusters properties are:

1. **Total**: Every tree node $v \in V^T$ has a cluster $c \in C$, $v \in \text{clusters}(c)$. This property ensures that all the nodes $v$ can be selected by the $\text{SelectIndex}(T, Q)$ algorithm.

2. **Unique**: The cluster of a node $v \in V^T$ is unique. If different clusters $c_i, c_j \in C$ do not share the same node $v \in V^T$ then $\text{clusters}(c_i) \cap \text{clusters}(c_j) = \emptyset$. This property ensures that a node $v$ is represented by a single cluster $c \in C$ in the $\text{SelectIndex}(T, Q)$ algorithm.
3. Shared label: All the nodes \( v_i, v_j \) in a cluster in \( C \) share the same label. Formally, if \( v_i, v_j \in \text{clusters}(c) \) then \( \text{label}^T(v_i) = \text{label}^T(v_j) \). This property ensures that the cluster \( c \in C \) is not selected by patterns that end with different labels.

The \( \text{SelectIndex}(T, Q) \) algorithms are applied to improve the pattern matching of \( Q \) for the data \( T \). The pattern matching is denoted by \( \text{SelectData}(T, Q) \). Instead of applying \( \text{SelectData}(T, Q) \), the \( \text{SelectIndex}(T, Q) \) algorithms apply \( \text{PruneIndex}(\text{summary}, \text{clusters}, Q) \) online to \( \text{clusters} \) and \( \text{summary} \) index data-structures that were constructed offline. Definition 6.2 establishes the relations between \( \text{SelectData}(T, Q) \) and \( \text{SelectIndex}(T, Q) \).

**Definition 6.2** The \( \text{SelectIndex}(T, Q) \) algorithm is safe if for every structural-pattern \( Q \) and a tree \( T \) \( \text{SelectData}(T, Q) \subseteq \text{SelectIndex}(T, Q) \). A \( \text{SelectIndex}(T, Q) \) algorithm is sound if for every pattern \( Q \) and a tree \( T \) \( \text{SelectIndex}(T, Q) \subseteq \text{SelectData}(T, Q) \).

The \( \text{SelectIndex}(T, Q) \) algorithm must be safe in order to replace \( \text{SelectData}(T, Q) \) properly. Figure 6.2 describes several families of the \( \text{SelectIndex}(T, Q) \) algorithms that are safe. The families of \( \text{SelectIndex}(T, Q) \) algorithms differ from the \( \text{summary} \) data-structures they maintain. The \( \text{SelectIndex}^{\text{Graph}}(T, Q) \) algorithm, which maintains a \( \text{summary} \) in a graph form, denoted by \( G^{\text{summary}} \), is called a graph-index.

The \( \text{SelectIndex}^{\text{FA}}(T, Q) \) algorithm, which maintains a \( \text{summary} \) in a finite automaton (FA) form, denoted by \( A^{\text{summary}} \), is called a FA-index. A \( \text{SelectIndex}^{\text{TA}}(T, Q) \) algorithm, which maintains a \( \text{summary} \) in a tree automaton (TA) form, denoted by \( A^{\text{summary}} \), is called a TA-index. Another difference between the families of indexes is by which the structural-patterns \( Q \) preform the selection. Both graph-indexes and FA-indexes use path structural-patterns to perform the selection. The TA-indexes are holistic. Therefore, tree structural-patterns are used to perform the selection. This chapter proves that graph-indexes are equivalent to FA-index and focuses on how the TA-indexes extend both the graph-indexes and the FA-indexes.
Figure 6.2: General flow of the three families from the \( SelectIndex(T, Q) \) algorithms. The ellipses denote the operations of the \( SelectIndex(T, Q) \) algorithm. The boxes denote the data-structures that are used as the inputs and the outputs for these operations.

The chapter, which describes in details the different families of \( SelectIndex(T, Q) \) algorithms, has the following structure: Section 6.2 provides preliminaries on graph-indexes, automata theory which is needed for understanding the automata-based indexes and related work. The \( FA \)-indexes and the holistic \( TA \)-indexes are described in Section 6.3. Section 6.4 provides experimental results.

### 6.2 Preliminaries and Related work

#### 6.2.1 SelectData(G,Q)

Given a path pattern \( Q \) and a node-labeled rooted graph \( G \), \( SelectData(G, Q) \) returns a node \( v \in SelectData(G, Q) \) if \( L(G, v) \cap L(Q) \neq \emptyset \). The node \( v \) is a match of \( Q \) in \( G \).

**Example 6.1** Figure 3.1 describes a node-labeled tree \( T = (V^T, E^T, label^T) \) where \( \Sigma = \{a, b, c, d, e\} \). The path from node 1 to node 4 is the nodes sequence (1, 2, 3, 4). The label-path of path (1, 2, 3, 4) is the string ‘abac’. The language of \( T \) in Fig. 3.1 is \( L(T) = \{a, ab, aba, abab, ababd, ababe, abac, abd\} \). The path pattern \( Q = \Sigma^*b\Sigma^*b \) where \( \Sigma^* \) denotes an arbitrary sequence of zero or more symbols from the alphabet \( \Sigma \). The language \( L(Q) \) includes all the words that include ‘b’ and end with another ‘b’. \( L(T) \cap L(Q) = \{abab\} \). \( L(T, 8) \cap L(Q) = L(T, 14) \cap L(Q) = \{abab\} \neq \emptyset \). Therefore, \( SelectData(T, Q) = \{8, 14\} \).

Part II: Querying of XML in a DB
Given a twig pattern $Q$ and a node-labeled rooted graph $G$, $v_{G_i} \in SelectData(G, Q)$ if it is part of a match of the twig pattern $Q$ in the graph $G$. A match of the twig pattern $Q$ in the graph $G$ is a set of distinct labels $v_{G_1}, \ldots, v_{G_n}$. Match maps each node $v_{G_i}$ to a query node $V_{Q_i} \in V^{TTwig}$ in the twig pattern $Q$ where the nodes relationships between node $v_{Q_i}$ and the other query nodes $v_{Q_j} \in V^{TTwig}$ are satisfied by the corresponding tree nodes $v_{G_i} \in V^G$ and $v_{G_j} \in V^G$.

**Example 6.2** There are three matches for twig pattern ‘//a[/l/c]/b[/l/e]/d’, in $T$, which is presented in Fig. 3. The matches, which are presented in the format $(v_1, \ldots, v_5)$ and are mapped into twig pattern nodes with labels $(a, b, c, d, e)$, are $(1, 11, 4, 12, 15)$, $(1, 11, 7, 12, 15)$, and $(6, 8, 7, 9, 10)$. $SelectData(T, Q) = \{1, 4, 6, 7, 8, 9, 10, 11, 12, 15\}$.

### 6.2.2 Graph-indexes

In this section, we introduce the family of graph indexes algorithms. A graph-index is a structural-index with a node-labeled rooted graph summary that is denoted by $G_{\text{summary}} = (V_{G_{\text{summary}}}, E_{G_{\text{summary}}}, \text{label}_{G_{\text{summary}}}, \text{root}_{G_{\text{summary}}})$ where the clusters are the summary nodes $V_{G_{\text{summary}}} \supseteq C$, $E_{G_{\text{summary}}}: C \mapsto C$ are the summary edges, $\text{label}_{G_{\text{summary}}}: C \mapsto \Sigma$ maps each cluster to a label and $\text{root}_{G_{\text{summary}}}$ is the root. Algorithm 19 describes the $SelectIndex_{\text{Graph}}(T, Q)$ of the graph-index.

**Algorithm 19:** The $SelectIndex_{\text{Graph}}$

**Input:** $T \triangleq (V^T, E^T, \text{label}^T)$, path pattern $Q$

**Output:** $\text{selected} \subseteq V^T$

begin

$G_{\text{summary}} \leftarrow \text{CreateSummary}(T)$; /* 1a in Fig. 6.1(b) */

Map $v \in \text{clusters}(c), w_v \in L(G_{\text{summary}}, c)$; /* CreateClusters 2a in Fig. 6.1(b) */

$C_{\text{pruned}} \leftarrow \text{SelectData}(G_{\text{summary}}, Q)$; /* PruneClusters 2a in Fig. 6.1(c) */

$\text{selected} \leftarrow \bigcup_{c \in C_{\text{pruned}}} \text{clusters}(c)$; /* PruneNodes 2b in Fig. 6.1(c) */

end

Lemma 6.1 describes the summary-properties of a graph-index which establishes the clusters properties that were described in Definition 6.1.

**Lemma 6.1** Given a graph-index which maintains $G_{\text{summary}}$ and clusters data structures. If $L(T) \subseteq L(G_{\text{summary}})$ then the clusters mapping is Total. If for every two different clusters $c_i, c_j \in C L(G_{\text{summary}}, c_i) \cap L(G_{\text{summary}}, c_j) = \emptyset$ then the clusters mapping is Unique. If $G_{\text{summary}}$ is a node-label graph then the clusters mapping has Shared Labels.

Part II Querying of XML in a DB
Theorem 6.1 proves that graph-indexes are safe as was defined in Definition 6.2.

**Theorem 6.1** A graph-index, which was constructed by Algorithm 19 and has summary-properties, which were described in Lemma 6.7, is safe.

**Proof** If \( v \in SelectData(T, Q) \) then \( w_v \in L(Q) \). From Algorithm 19 and because \( L(T) \subseteq L(G_{\text{summary}}) \), we know that \( v \in \text{clusters}(c) \) where \( w_v \in L(G_{\text{summary}}, c) \). Therefore, \( L(G_{\text{summary}}, c) \cap L(Q) \neq \emptyset \) and \( c \in SelectData(G_{\text{summary}}, Q) \). Therefore, \( v \in SelectIndex_{\text{Graph}}(T, Q) \) and \( SelectData(T, Q) \subseteq SelectIndex_{\text{Graph}}(T, Q) \). □

A variety of XML structural-indexes have been proposed in the literature. These indexes include the classical Dataguide [119], Lore [115] and 1-index [100], as well as more recent followups such as Index Fabric [49], stream partitions in Holistic Twig Joins [106] and A(K) indexes [87]. These indexing techniques work on relational, object-oriented and native DBMSs. These indexes use different \( G_{\text{summary}} \) graphs.

The F&B-Index [142] structural summary also uses a \( G_{\text{summary}} \) graph. But the F&B-Index clusters node \( v \in V_T \) according to both the top-down information in \( w_v \) and the bottom-up information which is the label paths in the subtree of \( v \). However, an F&B-Index suffers from the following problems:

1. Its size is usually too large in practice to be accommodated in memory;

2. Lack of efficiency: like TA-index, the F&B-Index input is twig patterns. But its twig pattern processing is path based and as such is not fully optimized. Even for the simple twig pattern that only contains P-C relations multiple branches need to be searched.

**Example 6.3** An example for the F&B-Index problems is given by the F&B-Index that is constructed from \( T \) in Fig. 3.2. The F&B-Index is \( T \) itself. Matching the twig pattern ‘/a[/c]/b[/e]/d’ on the F&B Index is the same as matching the twig pattern on \( T \). This examples the problems of the F&B-Index.

□

Part II Querying of XML in a DB
The following sections describe two CreateSummary(T) algorithms that are relevant for this chapter. Section 6.2.2.1 describes the Dataguide and 1-index algorithms and Section 6.2.2.2 describes the A(k)-indexes. Their performance will be compared with the proposed algorithms. 1-index is compared with the TA-index, which is based on A(0)-index

6.2.2.1 CreateSummary(T) of 1-index (Dataguide)

When a graph is in a tree form, the 1-index and the Dataguide CreateSummary(T) algorithms output the same summary. In this chapter, we refer to the graph-index, which maintains this summary, as 1-index. The 1-index is the most accurate index in terms of nodes selection. Given a node-labeled tree T, the CreateSummary(T) algorithm of the 1-index constructs a cluster \( c_w \) in \( G_{summary} \) from a word \( w \) of the node \( v \in V_T \).

Every edge \((v_i, v_j) \in E_T \) contributes an edge \((c_{w_{v_i}}, c_{w_{v_j}}) \) to \( E_{G_{summary}} \).

Like \( T \), the 1-index summary is a tree \( T_{summary} \). As a result, \( w_v = w_c \) where \( c \in V_{T_{summary}} \) and \( v \in clusters(c) \). The 1-index summary is sound because if \( v \in SelectIndex_{Graph}(T, Q) \) then \( w_c \in L(Q) \) where \( v \in clusters(c) \). Since \( w_c = w_v \) then \( w_v \in L(Q) \) and \( v \in SelectData(T, Q) \). Because it is sound, the 1-index algorithm tends to maintain large data-structures. In some cases, \(|T_{summary}| \approx |T|\). In this case, SelectIndex_{Graph}(T, Q) processing is not cost-effective in comparison to SelectData(T, Q). This is the reason why the A(k)-indexes were developed.

Example 6.4 Given a path pattern \( Q = \Sigma^{*}b\Sigma^{*}b' \) and the 1-index, which was constructed from the tree \( T \) in Fig. 3.1 that is illustrated in Fig. 6.3 then SelectIndex_{Graph}(T, Q) = \{8, 14\}. From Example 6.1 we can see that SelectIndex_{Graph}(T, Q) = SelectData(T, Q). \( \square \)
Figure 6.3: Illustration of the 1-index that is constructed from the tree in Fig. 3.1. A cluster $c$ is denoted by a circle. A label of a cluster $c$ includes two lines. The top line has the format ‘$\text{label}^{G_{\text{summary}}}(c)$’. The bottom line has the format ‘$\&i_1, \ldots, \&i_n$’ where $\text{clusters}(c) = \{v_{i_1}, \ldots, v_{i_n}\}$.

6.2.2.2 CreateSummary($T$) of A($k$)-index

$A(k)$-indexes are based on local similarities. Given a node-labeled tree $T$, the CreateSummary($T$) algorithm of the $A(k)$-index constructs from a tree node $v \in V_T$ a summary node $c_{w^k_v} \in V_{G^{G_{\text{summary}}}}$ where $w^k_v$ is the suffix of $w_v$ that contains a maximum of $k$ labels. Formally, the $A(k)$-index constructs from the tree node $v$, where $w_v = a_1, \ldots, a_n$, a cluster $c_{w^k_v}$ where $w^k_v = \begin{cases} a_1, \ldots, a_n & n < k \\ a_{n-k}, \ldots, a_n & \text{Otherwise} \end{cases}$. Every edge $(v_i, v_j) \in E_T$ contributes an edge $(c_{w^k_{v_i}}, c_{w^k_{v_j}})$ to $E_{G^{G_{\text{summary}}}}$.

Example 6.5 Given the path pattern $Q= \Sigma^*b\Sigma^*$, $A(0)$-index and $A(1)$-index, which were constructed from the tree $T$ in Fig. 3.1 which are illustrated in Fig. 6.4 and $\text{SelectIndex}^{\text{Graph}}(T, Q) = \{2, 5, 8, 11, 14\}$. From the Example 6.1 we can see that $\{8, 14\} = \text{SelectData}(T, Q) \subset \text{SelectIndex}^{\text{Graph}}(T, Q)$. $A(k)$-index is a generalization of the 1-index because $A(\infty)$-index $=$ 1-index.

Part II: Querying of XML in a DB
Figure 6.4: a. The $A(1)$-index. b. The $A(0)$-index. Both are constructed from the tree $T$ in Fig. 3.1. A cluster $c$ is denoted by a circle. A label of a cluster $c$ includes two lines. The top line has the format ‘$\text{label}^{G_{\text{summary}}}(c)$’. The bottom line has the format ‘$\&i_1, \ldots, \&i_n$’ where $\text{clusters}(c) = \{v_{i_1}, \ldots, v_{i_n}\}$.

### 6.2.2.3 Automata learning

This section describes how $\text{ConstructSummary}(T)$ algorithms apply automata learning methods. The $\text{ConstructSummary}(T)$ of a $FA$-index constructs a $FA$ $A_{\text{summary}}$ from $L(T)$. It can apply $FA$ learning techniques. For example, $L^*$ algorithm [19]. A $\text{ConstructSummary}(T)$ of a $TA$-index can construct $UUTA$ $A_{\text{summary}}$ in three steps:

1. Applies schema extraction methods to construct an XML schema from $T$. Schema extraction is addressed in [108, 107, 75, 29, 122];
2. Translates the XML schema to an ordered $TA$. Different XML schema languages as ordered $TA$ were defined in [105];
3. Constructs a $UUTA$ from the ordered $TA$ (see section 6.3.2.1).

### 6.3 Algorithm - automata based structural indexing

This section describes the automata based structural-indexes. Section 6.3.1 and Section 6.3.2 describe $FA$-indexes and $TA$-indexes, respectively.

#### 6.3.1 FA-indexes

The $FA$-index is a structural-index that maintains a summary $FA_{\text{summary}} \triangleq (Q_{\text{summary}}, \Sigma, q_0_{\text{summary}}, F_{\text{summary}}, \delta_{\text{summary}})$ where the clusters are the summary states $Q_{\text{summary}} \triangleq$
$C, \Sigma$ is the alphabet, $q_0^{A_{\text{summary}}} \in C$ is the start state, $F^{A_{\text{summary}}} \subseteq C$ are the accepting states and $\delta^{A_{\text{summary}}}: C \times \Sigma \mapsto C$. Algorithm 20 describes the SelectIndex$^{FA}(T, Q)$ algorithm.

**Algorithm 20:** The SelectIndex$^{FA}$ algorithm

**Input:** $T \triangleq (V^T, E^T, \text{label}^T)$, path pattern $Q$

**Output:** $\text{selected} \subseteq V^T$

begin

$FA^{A_{\text{summary}}} \leftarrow \text{CreateSummary}(T)$;  
/* CreateSummary(T) la 
 in Fig. 6.1(b) */

Map $v \in \text{clusters}(c)$ where $\delta^{A_{\text{summary}}}(w_v) \mapsto c$
;  
/* CreateClusters 2a in Fig. 6.1(b) */

Construct $FA^{AQ}$ from path pattern $(RE) Q$;

$C_{\text{pruned}} \leftarrow c$ where $\langle c, q \rangle \in F^{A_{\text{summary}}} \cap A^Q$ and $\langle c, q \rangle$ is accessible 
;  
/* PruneClusters 2a in Fig. 6.1(c) */

$\text{selected} \leftarrow \bigcup_{c \in C_{\text{pruned}}} \text{clusters}(c)$;  
/* PruneNodes 2b in Fig. 6.1(c) */

end

Lemma 6.2 describes the $FA$-index summary properties which establishes the clusters properties that were described in Definition 6.1.

**Lemma 6.2** Given a $FA$-index that maintains $A_{\text{summary}}$ and clusters data structures. If $L(T) \subseteq L(A_{\text{summary}})$ then clusters mapping is Total. If $A_{\text{summary}}$ is deterministic then clusters mapping is Unique. If $A_{\text{summary}}$ is a CFA then clusters mapping has Shared Labels.

**Proof** If $L(T) \subseteq L(A_{\text{summary}})$ then for every $v \in V^T$ $w_v \in L(A_{\text{summary}})$. Therefore, $\delta^{A_{\text{summary}}}(w_v) \mapsto c$ where $c \in F^{A_{\text{summary}}}$, $v \in \text{clusters}(c)$ and clusters mapping is Total. If $A_{\text{summary}}$ is deterministic then for every word $w \in L(A_{\text{summary}})$ there is a single derivation $\delta^{A_{\text{summary}}}(w) \mapsto c$ where $c \in F^{A_{\text{summary}}}$. Therefore, word $w_v$ has also a single derivation and $v$ is mapped into a single cluster $c$. If $A_{\text{summary}}$ is a CFA then all the transitions to the same cluster $c$ accept the same label $\sigma$. Therefore, if $\delta^{A_{\text{summary}}}(w) \mapsto c$ then $w \in \Sigma^* \sigma$. Therefore, if $v \in \text{clusters}(c)$ then $w_v \in \Sigma^* \sigma$. Therefore, label$^T(v) = \sigma, v \in \text{clusters}(c)$ shares label $\sigma$ and the clusters mapping has Shared Labels.

Theorem 6.2 proves that $FA$-indexes, which were constructed by Algorithm 20 are safe as was defined in Definition 6.2.

**Theorem 6.2** Given a tree $T$, a $FA$-index, which was constructed by Algorithm 20 and established the summary properties in Lemma 6.2 is safe.
Therefore, if \( v \in \text{SelectData}(T, Q) \) then \( w_v \in L(Q) \). Therefore \( w_v \in L(A^Q) \) and \( \delta^{A^Q}(w_v) \rightarrow q \) where \( q \in F^{A_Q} \). But \( w_v \in L(A^{\text{summary}}) \) because \( w_v \in L(T) \) and \( L(T) \subseteq L(A^{\text{summary}}) \). From Algorithm 20, we know that \( v \in \text{clusters}(c) \) where \( \delta^{A^{\text{summary}}}(w_v) \rightarrow c \) and \( c \in F^{A^{\text{summary}}} \). Therefore, \( w_v \in L(A^{\text{summary}} \cap A^Q) \) and exists \( \delta^{A^{\text{summary}} \cap A^Q}(w_v) \rightarrow \langle c, q \rangle \) such that \( \langle c, q \rangle \in F^{A^{\text{summary}} \cap A^Q} \) is an accessible state. Therefore, \( c \in \text{PruneClusters}(T, Q) \) and \( v \in \text{SelectIndex}^{\text{FA}}(T, Q) \).

6.3.1.1 CreateSummary(T) of the FA-index

The FA \( A^{\text{summary}} \) can be constructed in several ways. \( A^{\text{summary}} \) can be inferred from \( T \) by a grammatical inference techniques that are detailed in Section 6.2.2.3. The \( A^{\text{summary}} \) can also be inferred from an XML schema. See for example [113]. The FA \( A^{\text{summary}} \) can also be constructed from a graph-index. In this section we prove that every graph-index, which maintains data-structures \( G^{\text{summary}} \) and \( \text{clusters} \), can be represented as a FA-index by replacing the graph \( G^{\text{summary}} \) with \( FA^{\text{summary}} \). Definition 6.3 details how to construct a \( FA^{\text{summary}} \) from graph \( G^{\text{summary}} \). Each node in the graph contributes a state and each edge contributes a transition. All the states are accepting.

**Definition 6.3** Let \( G^{\text{summary}} = (V^{G^{\text{summary}}}, E^{G^{\text{summary}}}, \text{label}^{G^{\text{summary}}}, \text{root}^{G^{\text{summary}}}) \) be a summary. We construct \( CFA^{A^{\text{summary}}} = (Q^{A^{\text{summary}}}, \text{label}^{A^{\text{summary}}}, q_0^{A^{\text{summary}}}, F^{A^{\text{summary}}}, \delta^{A^{\text{summary}}}) \) where \( Q^{A^{\text{summary}}} = C \cup \{q_0^{A^{\text{summary}}}\} \), \( \text{label}^{A^{\text{summary}}} = \text{label}^{G^{\text{summary}}} \), \( F^{A^{\text{summary}}} = C \) are the accepting states and every edge \( (c_p, c_c) \in E^{G^{\text{summary}}} \) contributes a transition \( \delta^{A^{\text{summary}}}(c_p) = c_c \). An additional transition \( \delta^{A^{\text{summary}}}(q_0^{A^{\text{summary}}}) = \text{root}^{G^{\text{summary}}} \) is constructed.

Theorem 6.3 proves that \( \text{SelectIndex}^{\text{Graph}}(T, Q) \), which maintains data-structure \( G^{\text{summary}} \) and \( \text{clusters} \), is equal to \( \text{SelectIndex}^{\text{FA}}(T, Q) \) algorithm that maintains \( A^{\text{summary}} \), which is constructed from \( G^{\text{summary}} \) according to Definition 6.3.

**Lemma 6.3** Given \( T \), \( G^{\text{summary}} \) and \( \text{clusters} \), which was constructed from \( T \) by Algorithm 19 and \( CFA^{A^{\text{summary}}} \), which was constructed from \( G^{\text{summary}} \) according to Definition 6.3. Then, \( \delta^{A^{\text{summary}}}(w_v) \rightarrow c \) if and only if \( w_v \in L(G^{\text{summary}}, c) \).

**Proof** By induction on the \( \text{level}(T, v) \). The induction step assumes that the Lemma’s claim is true for \( v \), where \( \text{level}(T, v) \leq k \). It will be proved for \( v \) where \( \text{level}(T, v) = k + 1 \). If \( \delta^{A^{\text{summary}}}(w_v) \rightarrow c \) then \( w_v = w_{v_p} \sigma \) where \( \delta^{A^{\text{summary}}}(w_{v_p}) \rightarrow c_p \), \( v \in \text{children}(T, v_p) \), \( \text{label}^T(v) = \text{label}^{A^{\text{summary}}}(c) = \sigma \) and exists transition \( \delta^{A^{\text{summary}}}(c_p) = c \). \( \text{level}(T, v_p) = k \). Therefore, from the induction claim we get \( w_{v_p} \in L(G^{\text{summary}}, c_p) \). From definition 6.3 we know that \( (c_p, c) \in E^{G^{\text{summary}}} \). Therefore, \( w_v \in L(G^{\text{summary}}, c) \). The other direction is similarly proved.

From Lemma 6.3, Algorithm 19 and Algorithm 20 we see that graph-index, which maintains \( G^{\text{summary}} \) and \( CFA \)-index, which maintains the \( FA^{\text{summary}} \) that was constructed from \( G^{\text{summary}} \) by Definition 6.3 maintain the same \( \text{clusters} \).
Theorem 6.3 Given $G_{\text{summary}}$, clusters, $A_{\text{summary}}$, which was constructed from $G_{\text{summary}}$ according to Definition 6.3 and path pattern $Q$. Then, $\text{PruneIndex}(G_{\text{summary}}, \text{clusters}, Q) = \text{PruneIndex}(A_{\text{summary}}, \text{clusters}, Q)$. □

Proof if $v \in \text{PruneIndex}(G_{\text{summary}}, \text{clusters}, Q)$ then $\delta_{A_{\text{summary}} \cap A^Q}(w_v) \rightarrow \langle c, q \rangle$ where $\langle c, q \rangle \in F_{A_{\text{summary}} \cap A^Q}$ and $v \in \text{clusters}(c)$. From this we get that 1. $\delta_{A_{\text{summary}}}(w_v) \rightarrow c$ where $c \in F_{A_{\text{summary}}}$; 2. $\delta_{A^Q}(w_v) \rightarrow q$ where $q \in F_{A^Q}$. If $\delta_{A_{\text{summary}}}(w_v) \rightarrow c$ and $c \in F_{A_{\text{summary}}}$ then, according to Lemma 6.3, $w_v \in L(G_{\text{summary}}, c)$. If $\delta_{A^Q}(w_v) \rightarrow q \in F_{A^Q}$ then $w_v \in L(A^Q) = L(Q)$. Therefore, $w_v \in L(G_{\text{summary}}, c) \cap L(Q)$ and $L(G_{\text{summary}}, c) \cap L(Q) \neq \emptyset$. Therefore, $c \in \text{PruneClusters}(G_{\text{summary}}, Q)$ and $v \in \text{PruneIndex}(G_{\text{summary}}, \text{clusters}, Q)$. The other direction is similarly proved. ■

Example 6.6 Figure 6.5 describes different automata. Figure 6.5(a) describes the FA $A_{\text{summary}}$, which was constructed from the $G_{\text{summary}}$ in Fig. 6.3 by the construction in Definition 6.3. $A_{\text{summary}} = (Q_{\text{summary}}, \text{label}_{\text{summary}}, q_0_{\text{summary}}, F_{A_{\text{summary}}}, \delta_{A_{\text{summary}}})$ where $Q_{\text{summary}} = \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8 \}$, $\text{label}_{\text{summary}} = \{ (q_1, a), (q_2, b), (q_3, a), (q_4, c), (q_5, b), (q_6, d), (q_7, c), (q_8, d) \}$. $q_0$ is the start state. All the other states are accepting. The transitions are: $\delta_{A_{\text{summary}}}(q_0) = q_1$, $\delta_{A_{\text{summary}}}(q_1) = q_2$, $\delta_{A_{\text{summary}}}(q_2) = q_3$, $\delta_{A_{\text{summary}}}(q_3) = q_4$, $\delta_{A_{\text{summary}}}(q_4) = q_5$, $\delta_{A_{\text{summary}}}(q_5) = q_6$, $\delta_{A_{\text{summary}}}(q_6) = q_7$. Figure 6.5(b) describes the FA $A^Q$ that was constructed from the path pattern $Q = \Sigma^*b\Sigma^*b^*$. $A^Q$ is the tuple $(Q_{A^Q}, \Sigma, q_0_{A^Q}, F_{A^Q}, \delta_{A^Q})$ where $Q_{A^Q} = \{ q_0, q_{o1}, q_{o2}, q_{o3}, q_{o4}, q_{o5}, q_{o6} \}$, $\Sigma = \{ a, b \}$, $F_{A^Q} = \{ q_{o5} \}$, $q_0_{A^Q} = q_{o5}$. The transitions are: $\delta_{A^Q}(q_{o5}, \Sigma) = q_{o1}$, $\delta_{A^Q}(q_{o1}, b) = q_{o1}$, $\delta_{A^Q}(q_{o1}, \Sigma) = q_{o1}$, $\delta_{A^Q}(q_{o1}, b) = q_{o1}$, $\delta_{A^Q}(q_{o1}, \Sigma) = q_{o5}$. The intersection of $A_{\text{summary}}$ and $A^Q$ after the removal of the unaccessible states is described in Figure 6.3(c). $F_{A_{\text{summary}} \cap A^Q} = \{ \langle q_{o5}, q_{o5} \rangle \}$. Therefore, $\text{PruneIndex}(A_{\text{summary}}, \text{clusters}, Q) = \{ q_5 \}$. The clusters($q_5$) = $\{ 8, 14 \}$. Therefore, $\text{SelectIndex}(T, Q) = \{ 8, 14 \}$. Note that $\text{SelectIndex}_{T, \text{Graph}}(T, Q) = \{ 8, 14 \}$, which was described in Example 6.4, as was proved in Theorem 6.3. □
Figure 6.5: Description of three $FA$: (a) The $A^{summary}$, which was constructed from the $G^{summary}$ in Fig. 6.3. (b) The $A^Q$ of path pattern $Q = \Sigma^* b \Sigma^* b \Sigma^* b$. (c) The $A^{summary} \cap A^Q$ after the removal of unaccessible states. The automata are described as state machines. A state $q \in Q^A$ is denoted by a blank circle. $q^0_A$ is denoted by an incoming arrow. $q \in F^A$ is denoted by a double circle. The label of state $q$ in (b) and (c) has the format ‘$q$’. The label of state $q$ in (a) has two lines: the top has the format $q; label^A(q)$. The bottom line has the format ‘&$i_1$, . . . , &$i_n$’ where $clusters(q) = v_{i_1}, . . . , v_{i_n}$. A transition is denoted by an edge between states $q_s$ and $q_t$.

### 6.3.2 TA-indexes

A $TA$-index is a structural-index with a $UUTA$ $A^{summary} \overset{\Delta}{=} (Q^{A^{summary}}, \Sigma, F^{A^{summary}}, \delta^{A^{summary}})$ where the clusters are the summary states $Q^{A^{summary}} = C$, $\Sigma$ is the alphabet, $F^{A^{summary}} \subseteq C$ are the accepting states and $\delta^{A^{summary}} : 2^C \times \Sigma \mapsto C$. Algorithm 21 describes the $TA$-index construction.
Algorithm 21: The $SelectIndex^{TA}$ algorithm

**Input:** $T \triangleq (V^T, E^T, label^T)$, twig pattern $Q$

**Output:** $selected \subseteq V^T$

begin

| $CUUT A A^{summary} \leftarrow CreateSummary(T)$; /* CreateSummary($T$) |
| 1a in Fig. 6.1(b) */ |
| Map $v \in clusters(c)$ where $c \in \rho_{TA^{summary}}^{ST A^{summary}}(T)(v)$ |
| ; /* CreateClusters 2a in Fig. 6.1(b) */ |
| Construct $UUT A A^{Q}$ from twig pattern $Q$; |
| $C_{pruned} \leftarrow c$ where $\langle c, q \rangle \in F^{A^{summary} \cap A^{Q}}_{pc}$ is accessible |
| ; /* PruneClusters 2a in Fig. 6.1(c) */ |
| $selected \leftarrow \bigcup_{c \in C_{pruned}} clusters(c)$; /* PruneNodes 2b in Fig. 6.1(c) */ |

end

The online and the offline phases of the $TA$-index $SelectIndex^{TA}(T, Q)$ in Algorithm 21 are similar to the online and the offline phases of the $FA$-index $SelectIndex^{FA}(T, Q)$ in Algorithm 20 while providing top-down and bottom-up information.

The offline phase $CreateIndex(T)$ of the $FA$-index clusters node $v$ to $clusters(c)$ where $\delta^{A^{summary}}(w_v) \rightarrow c$ and $w_v \in \Sigma^*$. The clustering is done according to the top-down information in $w_v$. The offline phase $CreateIndex(T)$ of the $TA$-index clusters node $v$ to $clusters(c)$ where $\delta^{A^{pc}}_{pc}(w^q_v) \rightarrow c$ and $w^q_v \in Q^{A^{summary} \star}$ is the deriving-states string of the path from $root(T)$ to $v$ (See Definition 4.4). The clustering is done according to both the bottom-up information and the top-down information in $w^q_v$. The bottom-up information is given by the deriving states $q \in Q^{A^{summary}}$, which compose $w^q_v$, and are annotated by the bottom-up run $A^{summary}(T)$. The top-down information is given by the top-down operation of the $FA A^{pc}$. The online phase $PruneIndex(A^{summary}_{FA}, clusters, Q_{paths})$ of the $FA$-index prunes a cluster $clusters(c)$ where $\langle c, q \rangle \in F^{A^{summary} \cap A^{Q}_{paths}}$ and $\langle c, q \rangle$ is accessible. The clustering is done according to the top-down information. The online phase $PruneIndex(A^{summary}_{TA}, clusters, Q)$ of the $TA$-index prunes a cluster $clusters(c)$ where $\langle c, q \rangle \in F^{A^{summary}_{TA} \cap A^{Q}}_{pc}$ and $\langle c, q \rangle$ is accessible. The clustering is done according to both the bottom-up and the top-down information. The bottom-up information is given by the bottom-up construc-
tion of $Q(Q_{\text{summary}}(A^Q))_{pc}$. The top-down information is given by the fact that $(c, q)$ is accessible.

Lemma 6.4 describes the summary properties, which establish the clusters properties that were described in Definition 6.1.

**Lemma 6.4** Given a TA-index that maintains UUTA $A^\text{summary}$ and clusters data structures. If $T \in L(A^\text{summary})$ then the clusters mapping is Total. If every $v \in V^T$ satisfies $|\rho^\text{ST A}_{Q^\text{summary}}(T)(v)| = 1$ then the clusters mapping is Unique. If $A^\text{summary}$ is a CUUTA then the clusters mapping has Shared Labels.

**Proof** Lemma 4.8 proved that for a given tree $T$. If $T \in L(A^\text{summary})$ then every node $v \in V^T$ satisfies $v \in \rho^\text{ST A}_{Q^\text{summary}}(T)(c)$ for at least one $c \in Q^\text{summary}$. Therefore, according to the CreateClusters in Algorithm 21 $v \in \text{clusters}(c)$ for at least one $c \in Q^\text{summary}$ and the clusters mapping is Total. According to the CreateClusters in Algorithm 21 $\rho^\text{ST A}_{Q^\text{summary}}(T)(v) = \{c\}$. Therefore, $v$ is mapped uniquely to a clusters($c$) and the clusters mapping is unique. If $A^\text{summary}$ is a CUUTA then all the transitions to the cluster $c$ accept the same label $A^\text{summary}(c) = \sigma$. Therefore, only the nodes $v \in V^T$, where label$^T(v) = \sigma$, can satisfy $c \in \rho^\text{ST A}_{summary}(T)(v)$. Since $\rho^\text{ST A}_{Q^\text{summary}}(T)(v) \subseteq \rho^\text{summary}(T)(v)$ only nodes $v \in V^T$, where label$^T(v) = \sigma$, can satisfy $c \in \rho^\text{ST A}_{Q^\text{summary}}(T)(v)$. Therefore, $v \in \text{clusters}(c)$ share label $\sigma$ and clusters has Shared Labels.

Theorem 6.4 proves that FA-indexes, which were constructed by Algorithm 21, are safe as was defined in Definition 6.2.

**Theorem 6.4** Given a twig pattern $Q$ and a tree $T$ then TA-index, which was constructed by Algorithm 21 that established the summary properties in Lemma 6.4 is safe.

**Proof** If $v \in \text{SelectData}(T, Q)$ then $q_{\text{Selecting}} \in \rho^\text{ST A}_{Q^\text{summary}}(T)(v)$ where $S = F^Q_A$. From the summary properties we know that $T \in L(A^\text{summary})$. Therefore, from Theorem 4.4 we know that there exists a cluster $q_{\text{Tree}} \in C$ where $q_{\text{Selecting}} \in \rho^\text{ST A}_{Q^\text{summary}}(A^Q)(q_{\text{Tree}})$. Therefore, there is an accessible state $(c, q) \in F(A^\text{summary}(A^Q)_{pc})$. From Algorithm 21 we get that $q_{\text{Tree}} \in \text{PruneClusters}(A^\text{summary}, Q)$. According to the Lemma, $q_{\text{Tree}} \in \rho^\text{ST A}_{Q^\text{summary}}(T)(v)$ and because $|\rho^\text{ST A}_{Q^\text{summary}}(T)(v)| = 1$ then $\rho^\text{ST A}_{Q^\text{summary}}(T)(v) = \{q_{\text{Tree}}\}$ and $v \in \text{clusters}(q_{\text{Tree}})$. Therefore, $v \in \text{PruneNodes}(C_{\text{pruned}}, \text{clusters})$ and $v \in \text{SelectIndex}^TA(T, Q)$.

The next sections describe several variants of $A^\text{summary}$ that are constructed by different CreateSummary($T$) algorithms. Section 6.3.2.1 constructs CUUTA $A^\text{summary}$ from an XML schema. Section 6.3.2.2 constructs CUUTA $A^\text{summary}$ from a FA $A^\text{summary}$. This section describes the relations between the FA-index, which maintains $A^\text{summary}$,
with the $TA$-index, which maintains $A_{TA}^{summary}$. Section [6.3.2.3] refines the $CUUTA$ $A_{TA}^{summary}$ construction from a $FA$ $A_{FA}^{summary}$ in order to make the nodes selection efficient.

6.3.2.1 CreateSummary($T$) from XML Schema

The $CreateSummary(T)$ from XML schema is done in three steps. 1. A XML schema is extracted from $T$ using a schema extraction method that was described in Section [6.2.2.3] 2. The XML schema is translated into an ordered $TA$. 3. This step induces $UUTA$ $A_{TA}^{summary}$ from the ordered $TA$.

Algorithm [22] constructs a $UUTA$ that accepts, without order considerations, the same trees as the input unranked ordered $TA$ does. The $UUTA$ construction uses the same states while the transitions are changed. The algorithm constructs from each ordered $TA$ transition a set of $UUTA$ transitions. Each $UUTA$ transition contains a different set of children states which can be expressed by the $RE$ that expresses the children states of the $TA$ transition.

Part I] Querying of XML in a DB
Algorithm 22: The \textit{ConstructUUTA} algorithm from \(TA\).

\begin{verbatim}
begin
forall \(\delta^{\text{in}}(R) = q\) do
\hspace{1em}forall \(S \in \text{ConstructSets}(R)\) do
\hspace{2em}Add \(\delta^{\text{out}}(S) = q\);
end

\textbf{ConstructSets}(\textit{RE} \textit{R})
\begin{verbatim}
begin
switch \textit{R} do
\hspace{1em}case \textit{Symbol} \textit{a} do
\hspace{2em}Add \{a\} to \textit{Sets};
\hspace{1em}case (\textit{R}_1|\textit{R}_2) do
\hspace{2em}Add \text{ConstructSets}(\textit{R}_1) \cup \text{ConstructSets}(\textit{R}_2) to \textit{Sets};
\hspace{1em}case \textit{R}_1\textit{R}_2 do
\hspace{2em}forall \(S_1 \in \text{ConstructSets}(\textit{R}_1), S_2 \in \text{ConstructSets}(\textit{R}_2)\) do
\hspace{3em}Add \(S_1 \cup S_2\) to \textit{Sets};
\hspace{1em}case \textit{(R}_1)^* do
\hspace{2em}forall \(\text{Subsets} \subset \text{ConstructSets}(\textit{R}_1)\) do
\hspace{3em}Add \(\bigcup_{S \in \text{Subsets}} S\) to \textit{Sets};
end
end
end
\end{verbatim}

\end{verbatim}

Example 6.7 An example for the application of Algorithm\textsuperscript{22} to an ordered \(TA\) transition \(\delta^{\text{out}}(\langle qa, qb \rangle^* qc) = q_d\). The algorithm builds the UUTA transitions: \(\delta^{\text{out}}(\{q_c\}) = q_d\), \(\delta^{\text{out}}(\{qa, q_c\}) = q_d\), \(\delta^{\text{out}}(\{qb, q_c\}) = q_d\) and \(\delta^{\text{out}}(\{qa, qb, q_c\}) = q_d\). The following table describes the operation of the \textit{ConstructSets} function in Algorithm\textsuperscript{22}. The columns of the table describe the input (\textit{R}) and the output (\textit{Sets}) of the recursive calls to the \textit{ConstructSets} function. The input for the initial call to \textit{ConstructSets} function is the \textit{RE} \(\langle qa, qb \rangle^* qc\). The output are the children sets that compose the constructed \textit{CUUTA} transitions.
6.3.2.2 CreateSummary(T) from a FA

The CreateSummary(T) can be done in two steps. 1. A CFA $A_{FA}^{summary}$ is extracted from $T$ using a grammatical induction methods, which are were described in Section 6.2.2.2 or from a graph-index summary that was constructed in Section 6.2.2.2. This step induces the CUUTA $A_{TA}^{summary}$ from the CFA $A_{FA}^{summary}$. Algorithm 23 constructs the CUUTA $A_{TA}^{summary}$ from a CFA $A_{FA}^{summary}$.

Algorithm 23: The ConstructUUTA algorithm from FA.

Input: $CFA$

$A_{FA}^{summary} \triangleq (Q_{FA}^{summary}, label_{FA}^{summary}, q_0_{FA}^{summary}, F_{FA}^{summary}, \delta_{FA}^{summary})$

Output: $CUUTA A_{TA}^{summary} \triangleq (Q_{TA}^{summary}, label_{TA}^{summary}, F_{TA}^{summary}, \delta_{TA}^{summary})$

1. begin
2. $F_{TA}^{summary} \leftarrow \{q|\delta_{TA}^{summary}(q) = q\}$
3. forall $q \in Q_{FA}^{summary}$ do
4. Let $S_q \leftarrow \{q_d|\delta_{FA}^{summary}(q) = q_d\}$
5. forall $S \subseteq S_q$ do
6. Add $\delta_{TA}^{summary}(S) = q$
7. end

Example 6.8 Algorithm 23 constructs from the FA $A_{FA}^{summary}$, which was described in Example 6.8 and was constructed from the $G_{summary}$ in Fig. 6.3, the following CUUTA $A_{TA}^{summary} = (Q_{TA}^{summary}, label_{TA}^{summary}, F_{TA}^{summary}, \delta_{TA}^{summary})$. $Q_{TA}^{summary} = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$, $label_{TA}^{summary} = \{(q_1, a), (q_2, b), (q_3, c), (q_4, a), (q_5, b), (q_6, d), (q_7, e), (q_8, d)\}$, $F_{TA}^{summary} = \{q_1\}$. The transitions are:

$\delta_{TA}^{summary}(\emptyset) = q_1$, $\delta_{TA}^{summary}(\emptyset) = q_2$, $\delta_{TA}^{summary}(\emptyset) = q_3$, $\delta_{TA}^{summary}(\emptyset) = q_4$, $\delta_{TA}^{summary}(\emptyset) = q_5$, $\delta_{TA}^{summary}(\emptyset) = q_6$, $\delta_{TA}^{summary}(\emptyset) = q_7$, $\delta_{TA}^{summary}(\emptyset) = q_8$, $\delta_{TA}^{summary}(\{q_6\}) = q_5$, $\delta_{TA}^{summary}(\{q_7\}) = q_5$, $\delta_{TA}^{summary}(\{q_8\}) = q_5$, $\delta_{TA}^{summary}(\{q_1, q_5\}) = q_5$, $\delta_{TA}^{summary}(\{q_1, q_7\}) = q_5$, $\delta_{TA}^{summary}(\{q_2, q_3\}) = q_2$, $\delta_{TA}^{summary}(\{q_2, q_4\}) = q_2$, $\delta_{TA}^{summary}(\{q_3, q_5\}) = q_2$, $\delta_{TA}^{summary}(\{q_1, q_5\}) = q_5$

Given a FA-index, which maintains a CFA $A_{FA}^{summary}$ and a clusters $F_{FA}$, we construct a TA-index that maintains CUUTA $A_{TA}^{summary}$, which is constructed from the $A_{FA}^{summary}$.
according to Algorithm 23 and \( \text{clusters}_{TA} \). Lemma 6.5 proves that \( \text{clusters}_{TA}(c) \subseteq \text{clusters}_{FA}(c) \).

**Lemma 6.5** Given a FA \( A_{FA}^{\text{summary}} \) and a \( A_{TA}^{\text{summary}} \), which is constructed from \( A_{FA}^{\text{summary}} \) according to Algorithm 23, and a tree \( T \in L(A_{TA}^{\text{summary}}) \). If \( q \in \rho^{ST_{FA}^{A_{FA}^{\text{summary}}}(T)}(v) \) then \( \delta_{FA}^{A_{FA}^{\text{summary}}}(w_v) \rightarrow q \).

**Proof** We prove the Lemma by induction on the nodes in \( \text{level}(T,v) \leq k \). The basis of the induction is proved for \( \text{root}(T) \) where \( \text{level}(T,v) = 1 \). Since \( T \in L(A_{TA}^{\text{summary}}) \) and \( |\rho^{ST_{A_{TA}^{\text{summary}}}(T)}(\text{root}(T))| = 1 \) then \( \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(\text{root}(T)) = \{q\} \) where \( q \in F_{FA}^{A_{FA}^{\text{summary}}} \). From Algorithm 23 we know that \( \delta_{FA}^{A_{FA}^{\text{summary}}}(q_0^{A_{FA}^{\text{summary}}}) = q \). Therefore, \( \delta_{FA}^{A_{FA}^{\text{summary}}}(w_v) \rightarrow q \) where \( v = \text{root}(T) \). The Lemma’s assumption is proved for \( \text{root}(T) \). The induction step assumes that the Lemma’s claim is true for nodes \( v \in V_T \) in \( \text{level}(T,v) \leq k \) and proves the claim for nodes where \( \text{level}(T,v) = k + 1 \). If \( \{v_c\} = \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(q_c) \) where \( \text{level}(T,v_c) = k + 1 \) then there is a node \( v_p = \text{parent}(T,v_c) \) where \( \text{level}(T,v_p) = k \) and \( \{q_p\} = \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v_p) \). From the induction assumption we know that \( \delta_{FA}^{A_{FA}^{\text{summary}}}(w_{v_p}) \rightarrow q_p \). Since both \( \{q_p\} = \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v_p) \) and \( \{q_c\} = \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v_c) \), we know that there is a transition \( \delta_{FA}^{A_{FA}^{\text{summary}}}(S) = q_p \) where \( q_c \in S \). From the TA construction in Algorithm 23 we know that there is a transition \( \delta_{FA}^{A_{FA}^{\text{summary}}}(q_p) = q_c \). Therefore, there is derivation \( \delta_{FA}^{A_{FA}^{\text{summary}}}(w_{v_c}) \rightarrow q_c \). □

Given a FA-index, which maintains the CFA \( A_{FA}^{\text{summary}} \) and the \( \text{clusters}_{FA} \). We construct a TA-index that maintains the CUUTA \( A_{TA}^{\text{summary}} \), which is constructed from \( A_{FA}^{\text{summary}} \) according to Algorithm 23 and \( \text{clusters}_{FA} \). Theorem 6.5 proves that \( \text{SelectIndex}_{TA}(T,Q) \) is a subset of the \( \text{SelectIndex}_{FA}(T,Q_{paths}) \).

**Theorem 6.5** Given a twig pattern \( Q \) and a FA-index, which maintains the CFA \( A_{FA}^{\text{summary}} \) and the \( \text{clusters}_{FA} \). We construct a TA-index that maintains the CUUTA \( A_{TA}^{\text{summary}} \), which was constructed from the \( A_{FA}^{\text{summary}} \) according to Algorithm 23 and \( \text{clusters}_{TA} \). Then, \( \text{PruneIndex}(A_{TA}^{\text{summary}}, \text{clusters}_{TA}, Q) \subseteq \text{PruneIndex}(A_{FA}^{\text{summary}}, \text{clusters}_{FA}, Q_{paths}) \).

**Proof** If \( v \in \text{PruneIndex}(A_{TA}^{\text{summary}}, \text{clusters}_{TA}, Q) \) then according to Algorithm 23 \( v \in \text{clusters}(c) \) where \( q \in \rho^{ST_{A_{TA}^{\text{summary}}}(A_{FA}^{\text{summary}})}(c) \) and \( S = F_{PC}^{AQ} \). If \( q \in \rho^{ST_{A_{TA}^{\text{summary}}}(A_{TA}^{\text{summary}})}(c) \) then from Lemma 4.5 we know that there is a tree \( T \triangleq (V_T, E_T, \text{label}_T) \) and a node \( v \in V_T \) where \( T \in L(A_{TA}^{\text{summary}} \cap A^{Q}), q \in \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v) \) and \( c \in \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v) \). If \( q \in \rho^{ST_{A_{TA}^{\text{summary}}}(T)}(v) \) then, according to Theorem 4.3, \( v \) is part of twig pattern solution. Therefore, \( \delta_{paths}^{AQ}(w_v) \rightarrow q_{paths} \). According to Lemma 6.5 if \( c \in \rho^{ST_{A_{TA}^{\text{summary}}}(A_{TA}^{\text{summary}})}(v) \) then \( \delta_{FA}^{A_{FA}^{\text{summary}}}(w_v) \rightarrow c \). Therefore, \( \langle c, q_{paths} \rangle \in F_{FA}^{AQ} \cap Q_{paths} \) and \( c \in \text{PruneClusters}(A_{FA}^{\text{summary}}, \text{clusters}_{FA}, Q_{paths}) \). According to Lemma 6.5 if
\( v \in \text{clusters}_{TA}(c) \) then \( v \in \text{clusters}_{FA}(c) \). Therefore, \( v \in \text{PruneIndex}(A_{FA}^{summary}, \text{clusters}_{FA}, Q_{paths}) \).

Example 6.9 describes the \( \text{SelectIndex}^{FA}(T, Q_{paths}) \), which maintains the \( CF_{FA} A_{FA}^{summary} \), and the \( \text{SelectIndex}^{TA}(T, Q) \) operation that maintains the \( CUUT_{TA} A_{TA}^{summary} \), which was constructed from \( A_{FA}^{summary} \) by Algorithm 23. Theorem 6.5 proved that \( \text{SelectIndex}^{TA}(T, Q) \) is a subset of the \( \text{SelectIndex}^{FA}(T, Q_{paths}) \). But Example 6.9 shows that in practice, \( \text{SelectIndex}^{TA}(T, Q) = \text{SelectIndex}^{FA}(T, Q_{paths}) \). Therefore, the \( TA \)-index is not effective. Section 6.3.2.3 describes a more effective way to create the \( TA \)-indexes from the \( FA A_{FA}^{summary} \).

Example 6.9 Figure 6.6 describes the \( \text{PruneClusters}(A_{FA}^{summary}, \text{clusters}_{FA}, Q_{paths}) \) of the \( FA \)-index, which was constructed in Example 4.2. The \( \text{PruneClusters}(A_{FA}^{summary}, \text{clusters}_{FA}, Q_{paths}) \) returns the clusters that compose the final states in Fig. 6.6(b). The final states are: \( \langle q_1, q_a \rangle, \langle q_2, q_b \rangle, \langle q_3, q_a \rangle, \langle q_4, q_c \rangle, \langle q_5, q_b \rangle, \langle q_6, q_d \rangle, \langle q_7, q_c \rangle, \langle q_8, q_d \rangle \). Therefore, the \( \text{PruneClusters}(A_{FA}^{summary}, \text{clusters}_{FA}, Q_{paths}) \) returns all the clusters \( c \in C \). As a result, all the nodes in \( T \) are returned by \( \text{SelectIndex}^{FA}(T, Q_{paths}) \). The \( FA \)-index does not optimize the node selection for the path pattern \( Q_{paths} \). Figure 6.7 describes the \( \text{PruneClusters}(A_{TA}^{summary}, \text{clusters}_{TA}, Q_{paths}) \) of the \( TA \)-index, which maintains the \( CUUT_{TA} A_{TA}^{summary} \) that was constructed from the \( A_{FA}^{summary} \) in Example 6.6 using Algorithm 23 and the twig pattern \( Q \) in Fig. 2.5. The \( \text{PruneClusters}(A_{TA}^{summary}, \text{clusters}_{TA}, Q_{paths}) \) returns the final state \( \langle c, q \rangle \in F(A_{TA}^{summary} \cap A'^{paths})_{pc} \). The final states, which are detailed in Fig. 6.7(b), are: \( \langle q_1, q_a \rangle, \langle q_2, q_b \rangle, \langle q_3, q_a \rangle, \langle q_4, q_c \rangle, \langle q_5, q_b \rangle, \langle q_6, q_d \rangle, \langle q_7, q_c \rangle, \langle q_8, q_d \rangle \). Therefore, the \( \text{PruneClusters}(A_{TA}^{summary}, \text{clusters}_{TA}, Q) \) returns all the clusters \( c \in C \). As a result, all the nodes in \( T \) are returned by \( \text{SelectIndex}^{TA}(T, Q) \). The \( TA \)-index does not optimize the nodes selection for twig pattern \( Q \).
Figure 6.6: Description of two $FA$: (a) The $A^{paths}$ of the path pattern $Q^{paths}$ in Example 3.2. (b) The $A^{summary} \cap A^{paths}$ after the removal of unaccessible states. $A^{summary}$ is constructed in Fig. 6.5(a) and $A^{paths}$ is described in (a). The automata are described as state machines. A state $q \in Q^{A}$ is denoted by a blank circle. $q_{0}^{A}$ is denoted by an incoming arrow. $q \in F^{A}$ is denoted by a double circle. The label of state $q$ has the format ‘$q$‘.
6.3.2.3 Improving CreateSummary(T) from FA $A_{FA}^{summary}$

Section 6.3.2.2 described a method to construct a $CUUT A_{TA}^{summary}$ from a $A_{FA}^{summary}$. But the $TA$-index, which was constructed from this $A_{TA}^{summary}$, is not effective. The reason is that the clusters of the $TA$-index do not utilize the full strength of the $TA$ methodology. The construction in Algorithm 23 satisfies for each cluster $c$, $clusters_{TA}(c) \subseteq clusters_{FA}(c)$. The $clusters_{FA}(c)$ clusters a node $v \in V_T$ according to the top-down information $w_v$. The $TA$ methodology works for both the bottom-up information and for the top-down information in $T$. Therefore, the cluster $c$ of a node $v$ should contain a top-down information on $w_v$ and a bottom-up information on the labels in the subtree$(T, v)$. Given a $FA A_{FA}^{summary}$, a straightforward method is to split the cluster $c$ into clusters $c_S$, where $S_v \in \Sigma^*$ is a set of labels of children$(T, v)$. The labels in the set $S_v$ contribute a bottom-up information. Algorithm 24 describes the improved $A_{TA}^{summary}$ construction.

Figure 6.7: Description of two $FA$: (a) The $A_{pc}^Q$ of twig pattern $Q = 'a[c//b][//e]/d'$. (b) The $(A_{TA}^{summary} \cap A^Q)_{pc}$ after the removal of unaccessible states. $A_{TA}^{summary}$ was constructed in Example 4.2 and $A^Q$ was described in Example 4.1. The automata are described as state machines. A state $q \in Q^A$ is denoted by a blank circle. $q^A_0$ is denoted by an incoming arrow. $q \in F^A$ is denoted by a double circle. The label of state $q$ has the format ‘$q$’.
Algorithm 24: The improved ConstructUUTA algorithm from FA.

\[
\text{Algorithm } A_{\text{summary}}^\text{FA} \triangleq (Q_{\text{summary}}^A, \delta_{\text{summary}}^A, q_0^A, F_{\text{summary}}^A, \delta_{\text{summary}}^A)
\]

\[
\text{Algorithm } A_{\text{summary}}^\text{TA} \triangleq (Q_{\text{summary}}^A, \delta_{\text{summary}}^A)
\]

\[
\text{Data: } \text{states} : V^T \mapsto Q_{\text{summary}}^A
\]

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{Input:} Tree \( T \triangleq (V^T, E^T, \text{label}^T) \), \( T \) \( F_A \)
\STATE \textbf{Output:} CUUTA \( A_{\text{summary}}^\text{TA} \)
\STATE \textbf{Data:} states : \( V^T \mapsto Q_{\text{summary}}^A \)
\FORALL \( v \in V^T \)
\STATE Let \( c \in Q_{\text{summary}}^A \) where \( \delta_{\text{summary}}^A(v_c) \rightarrow c ; \)
\STATE Let \( S_v \leftarrow \bigcup_{v_c \in \text{children}(T,v)} \text{label}^T(v_c) ; \)
\STATE \( Q_{\text{summary}}^A \leftarrow Q_{\text{summary}}^A \cup \{c_s \} ; \)
\STATE \( \text{label}^A \leftarrow \text{label}^T(v) ; \)
\STATE \( \text{states}(v) \leftarrow c_s ; \)
\STATE \( F_{\text{summary}}^A \leftarrow \{\text{states(root}(T))\} ; \)
\ENDFOR
\STATE \text{Add transition} \( \delta_{\text{summary}}^A((\bigcup_{v_c \in \text{children}(T,v)} \text{states}(v_c)) = \text{states}(v) ; \)
\end{algorithmic}
\end{algorithm}

Example 6.10 Algorithm 24 constructs from the tree \( T \), which was described in Fig. 3.1 and \( A_{\text{summary}}^\text{FA} \), which was constructed in Example 6.6, the following \( \text{CUUTA} \)
\( A_{\text{summary}}^\text{TA} = (Q_{\text{summary}}^A, \delta_{\text{summary}}^A) \) where \( Q_{\text{summary}}^A = \{q_{1(b)}, q_{2(a)}, q_{3(b)}, q_{4(c)}, q_{5(d)}, q_{6}, q_{7}, q_{8}\} \), \( \text{label}^A = \{(q_{1(b)}, a), \}
\( (q_{2(a)}, b), (q_{3(b)}, a), (q_{4(c)}, a), (q_{5(d)}, b), (q_{6}, d), (q_{7}, e), (q_{8}, d)\} \) and \( F_{\text{summary}}^A = \{q_{1(b)}\} \). The transitions are (the brackets besides each transition denote the nodes that construct this transition):
\[
\delta_{\text{summary}}^A(\{q_{2(a)}, q_{2(d)}\}) = \{q_{1(b)}, q_{2(a)}, q_{3(b)}\},
\delta_{\text{summary}}^A(\{q_{3(b)}\}) = \{q_{2(a)}, q_{4(c)}, q_{5(d)}\},
\delta_{\text{summary}}^A(\{q_{4(c)}\}) = \{q_{5(d)}\},
\delta_{\text{summary}}^A(\{q_{5(d)}\}) = \{q_{6}\},
\delta_{\text{summary}}^A(\{q_{6}\}) = \{q_{7}\},
\delta_{\text{summary}}^A(\{q_{7}\}) = \{q_{8}\}.
\]

Figure 6.8 describes the PruneClusters(\( A_{\text{summary}}^\text{TA} \), \( \text{clusters}_{\text{TA}} \), \( Q \) of the \( A_{\text{summary}}^\text{TA} \), which is constructed in this example, the \( \text{clusters}_{\text{TA}} \), which was described in Fig. 6.8(a), and the twig pattern \( Q \) in Fig. 2.5. The PruneClusters(\( A_{\text{summary}}^\text{TA} \), \( \text{clusters}_{\text{TA}} \), \( Q \)) returns the clusters that compose the final states in Fig. 6.8(b). The final states are: \( \{q_{1(b)}, q_{2(a)}, q_{3(b)}, q_{4(c)}, q_{5(d)}, q_{6}, q_{7}, q_{8}\} \). Therefore, the PruneClusters(\( A_{\text{summary}}^\text{TA} \), \( \text{clusters}_{\text{TA}} \), \( Q \)) returns the clusters: \( q_{1(b)}, q_{2(a)}, q_{3(b)}, q_{4}, q_{5(d)}, q_{6}, q_{7}, q_{8} \). Clusters \( q_{2(a)}, q_{3(b)}, q_{4} \) and \( q_{5(d)} \) are not se-

Part II Querying of XML in a DB
lected. Therefore, nodes 2, 3, 5, 13 and 14 are filtered by the $SelectIndex^{TA}(T, Q)$ operation of the improved $TA$-index. The $TA$-index optimize the node selection for twig pattern $Q$ in 33% in comparison to the $SelectIndex^{FA}(T, Q)$ in Example 5.6.
Figure 6.8: Description of two $FA$: (a) The $(A_{TA}^{summary})_{pc}$. (b) The $(A_{TA}^{summary} \cap A^{Q})_{pc}$ after the removal of the unaccessible states. The automata are described as state machines. A state $q \in Q^{A}$ is denoted by a blank circle. $q_{0}^{A}$ is denoted by an incoming arrow. $q \in F^{A}$ is denoted by a double circle. The label of state $q$ has the format ‘$q$’.

The constructed $UUTA$ satisfies the clusters properties. We can see that $T \in L(A_{TA}^{summary})$ because in the construction of Algorithm 24, the transition $\delta^{A_{TA}^{summary}}$ was constructed.
for every branch in \( T \). Algorithm 24 constructs a \( A_{TA}^{\text{summary}} \) as a \( \text{CUUTA} \). Lemma 6.7 proves that \( |\rho_{\text{STA}(T)}(v)| = 1 \). Lemma 6.7 uses Lemma 6.6 for the proof.

Lemma 6.6 Given a tree \( T \) and \( FA \)-index, which was constructed from \( T \) and maintains \( CFA \) \( A_{FA}^{\text{summary}} \), we construct the \( \text{CUUTA} \) \( A_{TA}^{\text{summary}} \) from \( A_{FA}^{\text{summary}} \) and \( T \) according to Algorithm 24. Then, for \( v \in V^T \), if \( q_{i,v} \in \rho_{\text{STA}(T)}(v) \) then \( \delta_{A_{FA}^{\text{summary}}}(w_v) \rightarrow q_i \).

\[ \text{PROOF: } \text{We prove the Lemma by induction on the nodes in } \text{level}(T,v) \leq k. \text{ The proof is similar to proof of Lemma 6.5.} \]

Lemma 6.7 Given a tree \( T \) and a \( FA \)-index, which was constructed from \( T \) and maintains \( CFA \) \( A_{FA}^{\text{summary}} \), we construct the \( \text{CUUTA} \) \( A_{TA}^{\text{summary}} \) from \( A_{FA}^{\text{summary}} \) and \( T \) according to algorithm 24. Then, for \( v \in V^T \) \( |\rho_{\text{STA}(T)}(v)| = 1 \).

\[ \text{PROOF: } \text{From the construction of Algorithm 24 we know that if two states } q_{i,S_m}, q_{j,S_n} \in Q_{A_{FA}^{\text{summary}}} \text{ satisfy } q_{i,S_m} \in \rho_{A_{FA}^{\text{summary}}}(T)(v) \text{ and } q_{j,S_n} \in \rho_{A_{FA}^{\text{summary}}}(T)(v) \text{ then } i \neq j. \text{ Otherwise, if } i = j \text{ then the node } v \text{ is annotated by two different transitions: to } q_{i,S_m} \text{ and to } q_{i,S_n}. \text{ This is impossible because, according to Algorithm 24 construction, } S_m \neq S_n. \text{ Therefore, either } S = S_m \text{ or } S = S_n \text{ where } S = \bigcup_{v \in \text{children}(v)} \text{label}^T(v_c). \text{ Therefore, } i \neq j \text{ but } S_m = S_n. \text{ If } q_{i,S_n}, q_{j,S_n} \in \rho_{A_{FA}^{\text{summary}}}(T)(v) \text{ then according to Lemma 6.6 } \delta_{A_{FA}^{\text{summary}}}(w_v) \rightarrow q_i \text{ and } \delta_{A_{FA}^{\text{summary}}}(w_c) \rightarrow q_j. \text{ But this contradicts the } FA \text{ determinism.} \]

Example 6.10 selected the nodes accurately. But it was constructed form a \( FA \) \( A_{FA}^{\text{summary}} \) that is based on the 1-index graph-summary. The size of the 1-index graph-summary is large. It can be in the size of \( |\Sigma| \). Algorithm 24 constructed from each original cluster \( c \) up to \(|\Sigma|^2 \) the clusters \( c_S \). One for each subset of the set of children symbols. Example 6.11 applies the Algorithm 24 to a \( A_{FA}^{\text{summary}} \) that is based on \( A(0) \)-index graph-summary. Example 6.11 removes the top-down information of the 1-index and adds the bottom-up information of Algorithm 24. The number of clusters \( c \) in the \( A(0) \)-index is \(|\Sigma| \). Therefore, the number of clusters \( c_S \), which were constructed by Algorithm 24, is \(|\Sigma|^3 \) and thus it is independent of the \( T \) size. Example 6.11 shows that the constructed \( TA \)-index is effective. In this example, most of the top-down information is regained because the bottom-up information reflects a top-down information as well. In our experiments, we use this type of holistic \( TA \)-indexes.

Example 6.11 Figure 6.9(a) describes a \( FA \)-index that maintains the clusters \( c_{FA} \) and the \( CFA \) \( A_{FA}^{\text{summary}} \) which was constructed from the \( G_{\text{summary}} \) in Fig. 6.4(b) by the construction in Definition 6.3. \( A_{FA}^{\text{summary}} = (Q_{A_{FA}^{\text{summary}}}, \text{label}_{A_{FA}^{\text{summary}}}, A_{FA}^{\text{summary}}, F_{A_{FA}^{\text{summary}}}, \delta_{A_{FA}^{\text{summary}}}) \). \( Q_{A_{FA}^{\text{summary}}} = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \text{label}_{A_{FA}^{\text{summary}}} = \{(q_1, a), (q_2, b), (q_3, c), (q_4, d), (q_5, e)\} \), \( q_0 = q_0 \), \( F_{A_{FA}^{\text{summary}}} = \{q_1, q_2, q_3, q_4, q_5\} \) and the transitions
Holistic structural-index

are: \( \delta_{A_{FA}^{summary}}(q_0) = q_1, \delta_{A_{FA}^{summary}}(q_1) = q_2, \delta_{A_{FA}^{summary}}(q_2) = q_1, \delta_{A_{FA}^{summary}}(q_1) = q_3, \delta_{A_{FA}^{summary}}(q_2) = q_4, \delta_{A_{FA}^{summary}}(q_2) = q_5. \)

Figure [6.9(b)] describes the PruneClusters\((A_{FA}^{summary}, clusters_{FA}, Q_{paths})\) of the FA-index, where \( A_{FA}^{summary} \) and \( clusters_{FA} \) are described in Fig. [6.9(a)], and the path pattern \( Q_{paths} \) was given in Example [3.2]. The PruneClusters\((A_{FA}^{summary}, clusters_{FA}, Q_{paths})\) returns the clusters that compose the final states \( \langle c, q \rangle \in FA_{summary} \cap A_{FA}^{paths} \) in Fig. [6.9(b)]. All the states \( c \in Q_{A_{FA}^{summary}}^{summary} \) compose the final states in Fig. [6.10(b)]. The final states are: \( \langle q_1, q_2, q_3, q_4, q_5 \rangle \). As a result, all the nodes \( \nu \in V^T \) are returned by SelectIndex\(^{FA}(T, Q_{paths})\).

Algorithm [24] constructs from tree \( T \), which was described in [3.1] and \( A_{summary}^{FA} \), which was constructed above, the following \( UUTA_{T,A}^{summary} = (Q_{T,A}^{summary}, label_{T,A}^{summary}, \delta_{T,A}^{summary}) \) where \( Q_{T,A}^{summary} = \{q_{1(b)}, q_{1(b,c)}, q_{1(c)}, q_{2(a)}, q_{2(a,d)}, q_{2(d,c)}, q_{2(e)}, q_{4}, q_{5}\}, \) \( label_{T,A}^{summary} = \{(q_{1(b)}, a), (q_{1(b,c)}, a), (q_{1(c)}, a), (q_{2(a)}, b), (q_{2(a,d)}, b), (q_{2(d,c)}, b), (q_{2(e)}, b), (q_{3}, c), (q_{4}, d), (q_{5}, e)\}, \delta_{T,A}^{summary} = \{q_{1(b)}\}. \) The transitions are (the brackets beside each transition denote the nodes that construct the transition): \( \delta_{T,A}^{summary}\left(\{q_{2(a)}\}, \{q_{2(d,c)}\}\right) = q_{1(b)} \). \( \delta_{T,A}^{summary}\left(\{q_{4}\}, \{q_{5}\}\right) = q_{1(b)} \). \( \delta_{T,A}^{summary}\left(\{q_{3}\}, \{q_{2(d,c)}\}\right) = q_{1(b,c)} \). \( \delta_{T,A}^{summary}\left(\{q_{1(b,c)}\}, \{q_{1(c)}\}\right) = q_{2(a)} \). \( \delta_{T,A}^{summary}\left(\{q_{1(c)}\}, \{q_{2(d,e)}\}\right) = q_{2(d,e)}. \)

Figure [6.10(b)] describes the PruneClusters\((A_{T,A}^{summary}, clusters_{T,A}, Q)\) where the \( A_{T,A}^{summary} \) and the \( clusters_{T,A} \) are detailed in Fig. [6.10(a)] and the twig pattern \( Q \) in Fig. [2.5]. The PruneClusters\((A_{T,A}^{summary}, clusters_{T,A}, Q)\) returns the clusters that compose the final states in Fig. [6.10(b)]. The final states are: \( \langle q_{1(b)}, q_{2(d,c)}, q_{3}, q_{4}, q_{5}, q_{6}\rangle, \langle q_{1(b,c)}, q_{7}\rangle, \langle q_{2(a)}, q_{8}\rangle, \langle q_{2(d,e)}, q_{9}\rangle, \langle q_{3}, q_{10}\rangle, \langle q_{4}, q_{11}\rangle, \langle q_{5}, q_{12}\rangle. \) Therefore, the PruneClusters\((A_{T,A}^{summary}, clusters_{T,A}, Q_{paths})\) returns the clusters: \( q_{1(b)}, q_{1(b,c)}, q_{2(a,d)}, q_{2(d,e)}, q_{3}, q_{4}, q_{5}. \) Clusters \( q_{1(c)}, q_{2(a)} \) and \( q_{2(e)} \) are filtered. Therefore, nodes 2, 3, 5 and 14 are filtered by SelectIndex\(^{T,A}(T, Q)\) operation. The TA-index optimize the node selection for twig pattern \( Q \) in 25% in comparison to the SelectIndex\(^{FA}(T, Q)\) in Example [6.6].

Part II Querying of XML in a DB
Figure 6.9: Description of two $FA$: (a) $A_{FA}^{\text{summary}}$, which is constructed from the $A(0)$-index. (b) The intersection $A_{FA}^{\text{summary}} \cap A_{Q, \text{paths}}^{\text{paths}}$, where $A_{\text{summary}}$ was described in (a) and $A_{Q, \text{paths}}$ was described in Fig. 6.6(a), after the removal of unaccessible states. The automata are described as state machines. A state $q \in Q^A$ is denoted by a blank circle. $q_0^A$ is denoted by an incoming arrow. $q \in F^A$ is denoted by a double circle. The label of state $q$ in (a) has two lines. The top line has the format ‘$q$; label$^A(q)$’ and the bottom line has the format ‘&i_1, \ldots, &i_n’ where clusters($q$) = $v_{i_1}, \ldots, v_{i_n}$. The label of the state $q$ in (b) is ‘$q$’. The label of the edge $(v_p, v_c)$ is ‘$a$’ where $\delta^A(v_p, a) = v_c$. 

Part II Querying of XML in a DB
Figure 6.10: Description of two $FA$: (a) The $(A_{TA}^{summary})_{pc}$. (b) The $(A_{TA}^{summary} \cap A_{Q}^{summary})_{pc}$ after the removal of unaccessible states. $A_{TA}^{summary}$ was constructed in Example 6.11 and $A_{Q}^{summary}$ was described in Example 4.2. The automata are described as state machines. A state $q \in Q^{A}$ is denoted by a blank circle. $q^{0}_{x}$ is denoted by an incoming arrow. $q \in F^{A}$ is denoted by a double circle. The label of the state $q$ has the format ‘$q$’.
6.4 Experimental results for the holistic TA-index operation

We apply the TA-index, which were described in Section 6.3.2.3 to XML documents with different characteristics. Our experimental results show that the TA-index outperforms the 1-index that was described in Section 6.2.2.1. The 1-index is the most accurate structural-index that is currently used.

6.4.1 Experimental settings for XML datasets

We implemented all the algorithms in Java 1.5. All our experiments were performed on a PC with 2.4GHz Pentium 4 processor and 1024MB RAM running Windows XP. We used the following real-world (DBLP [51]) and synthetic datasets (XMark [10]) in our experiments: (1) XMark XML dataset is synthetic and was generated by an XML data generator. It contains auction site information. Its schema is recursive. (2) The DBLP dataset is obtained from the University of Trier and consists of computer sciences references to articles and books. The schema of DBLP is not recursive and contains three levels of elements. The bottom level contains a mixture of more than ten elements. We select the above two XML datasets because they represent two important types of data: XMark is a syntactic example and has many repetitive structures whereas the DBLP is more real-word example and its bottom level is a mixture of elements. We used 100 MB datasets with about two million element nodes in each XML tree. Given XML tree $T$, we constructed the TA-index as follows: 1. We constructed $A(0)$-index from $T$, which maintains $G^{summary}$, as was described in Section 6.2.2.2. 2. We constructed $FA^{summary}_{A}$ from the $G^{summary}$ of the $A(0)$-index using Definition 6.3. 3. We constructed $TA^{summary}_{FA}$ from the $FA^{summary}_{A}$ using Algorithm 24. 4. We constructed the TA-index, which maintains $A^{summary}_{TA}$, using Algorithm 21. The 1-index was constructed from $T$ as was described in Section 6.2.2.1.

Table 6.1 summarizes the characteristics of the data files and indexes that were constructed from these files. Table 6.1 shows that the summary sizes of both indexes are relatively small in comparison to the size of the data. Each state $q \in Q^{A^{summary}_{FA}}$ of the 1-index $FA^{summary}_{A}$-index clusters has about ten thousand nodes on average. Each state $q \in Q^{A^{summary}_{TA}}$ in the TA-index clusters has about one thousand elements nodes on average. The online processing of the 1-index is the size of $A^{summary}_{FA} \cap A^{paths}_{Q}$. From
table 6.1 we learn that the size of $A_{FA}^{summary}$ for both DBLP and XMark data is relatively small. The number of states $|Q_{FA}^{summary}|$ ($|1-index|$) is from dozens up to couple of hundred. Therefore, the 1-index online processing is not significant in comparison to the saving in I/O operations. The online processing of the $TA$-index depends on the number of states $|Q_{TA}^{summary}|$ ($|TA-index|$). The number of states $|Q_{TA}^{summary}|$ for both DBLP and XMark data is relatively small (couple of hundred states). Therefore, the $TA$-index processing is not significant in comparison to the saving in I/O operations.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Nodes number</th>
<th>$1-index$</th>
<th>$TA$-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark</td>
<td>135 MB</td>
<td>1.81 million</td>
<td>411</td>
<td>1630</td>
</tr>
<tr>
<td>DBLP</td>
<td>79 MB</td>
<td>1.65 million</td>
<td>56</td>
<td>820</td>
</tr>
</tbody>
</table>

Table 6.1: Datasets used in our experiments. The table describes the following characteristics of two XML files (Data) XMark and DBLP: The data files sizes in MB (Size); the number in millions (Nodes number) of element nodes in the XML data tree form; the number of states $|Q_{FA}^{summary}|$ of the 1-index ($|1-index|$); the number of states in $|Q_{TA}^{summary}|$ of the $TA$-index ($|TA-index|$).

6.4.1.1 Queries

We developed a random twig pattern generator. The pattern generator receives the XML Schema as an input. The pattern generator uses Eclipse [53] XML Schema parser to randomly traverses the XML schema. The twig pattern generator adds children and descendent elements, which are defined by the traversed XML schema type definitions, to the pattern. The XML Schema traversal produces twig patterns $Q$ where $SelectData(T, Q) \neq \emptyset$. The twig patterns generator is configured to produce pattern with different number of nodes. We call the number of nodes in a twig pattern a pattern size. In the experiments, we generated one hundred queries for each pattern size $k$ where $2 \leq k \leq 8$. The 1-index processes the path pattern $Q_{paths}$ which was constructed from the generated twig pattern $Q$.

6.4.2 Performance measurements

We implemented the two XML structural index algorithms: The $TA$-index and the 1-index using the file system as a simple storage engine. The reason that we choose to
compare the TA-index to the 1-index is that the 1-index is the most known selective structural index for XML. The suggested TA-index outperforms 1-index selectiveness.

In the experiments, we generated one hundred queries \(1 \leq i \leq 100\) for each twig pattern size \(2 \leq K \leq 8\). We denote the pattern \(i\) of size \(k\) as \(Q^k_i\). For each pattern \(Q^k_i\), we gathered number of nodes that were selected by 1-index (\(|Q^k_i|^{1\text{-index}}\)) and by TA-index (\(|Q^k_i|^{TA\text{-index}}\)).

For each twig pattern size \(2 \leq k \leq 8\), we considered the following performance metrics to compare between the selectiveness of the TA-index and the 1-index: (1) The improvement in the average coverage of the TA-index is the average percentage of queries \(Q^k_i\) that satisfy \(|Q^k_i|^{1\text{-index}} > |Q^k_i|^{TA\text{-index}}\); (2) The average improvement percentage in node selection of the TA-index over 1-index. The result is \(\frac{\sum_i |Q^k_i|^{TA\text{-index}}}{|Q^k_i|^{1\text{-index}}} \times 100\) where \(1 \leq i \leq 100\); (3) The average gain in node selection of the TA-index over 1-index. The result is \(\sum_i (|Q^k_i|^{1\text{-index}} - |Q^k_i|^{TA\text{-index}}) \times \frac{1}{100}\) where \(1 \leq i \leq 100\); (4) The maximal improvement percentage in node selection of the TA-index over 1-index. The result is \(\min(\frac{|Q^k_i|^{TA\text{-index}}}{|Q^k_i|^{1\text{-index}}} \times 100)\) where \(1 \leq i \leq 100\).

Table 6.2 compares between the performance matrices of XMark and DBLP datasets. From the table we can see that TA-index substantially outperforms the 1-index selectivity. The TA-index selects on average less than half the nodes of the 1-index for the same twig patterns. For example, see the average improvement of the twig patterns with eight nodes using the DBLP data. For certain twig patterns, the TA-index can return 500 times less nodes than the 1-index pattern! For example, see the maximal improvement of patterns with more than four nodes in the DBLP dataset. From Table 6.2 we see that the average gain in node selection of the TA-index is high in comparison to \(|Q^{summary}\text{T}_A|\) which depends on the TA-index online processing. Two additional conclusions about the TA-index can be inferred from Table 6.2 about the improvement in node selection of the TA-index: 1. The real-world data (DBLP) is more affected by the improvement of the TA-index than the syntactic examples (XMark). It can be explained by the fact that the real-world data contains a mixture of elements that is not expressed by \(Q^{paths}\) but expressed by \(Q\); 2. The improvement is more significant when the pattern is more complex. For larger twig patterns, the average coverage increases and both the average improvement and the maximal improvement decrease.
<table>
<thead>
<tr>
<th>Data</th>
<th>K</th>
<th>Average coverage (%)</th>
<th>Average improvement (%)</th>
<th>Average gain</th>
<th>Maximal improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>2</td>
<td>57</td>
<td>95</td>
<td>2887</td>
<td>11.44</td>
</tr>
<tr>
<td>DBLP</td>
<td>3</td>
<td>86</td>
<td>87</td>
<td>8095</td>
<td>38.37</td>
</tr>
<tr>
<td>DBLP</td>
<td>4</td>
<td>95</td>
<td>81</td>
<td>16424</td>
<td>11.18</td>
</tr>
<tr>
<td>DBLP</td>
<td>5</td>
<td>9</td>
<td>72</td>
<td>20283</td>
<td>0.19</td>
</tr>
<tr>
<td>DBLP</td>
<td>6</td>
<td>100</td>
<td>58</td>
<td>53726</td>
<td>0.15</td>
</tr>
<tr>
<td>DBLP</td>
<td>7</td>
<td>100</td>
<td>44</td>
<td>60168</td>
<td>0.15</td>
</tr>
<tr>
<td>DBLP</td>
<td>8</td>
<td>100</td>
<td>47</td>
<td>55841</td>
<td>0.16</td>
</tr>
<tr>
<td>XMark</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>XMark</td>
<td>3</td>
<td>28</td>
<td>93</td>
<td>1936</td>
<td>58.2</td>
</tr>
<tr>
<td>XMark</td>
<td>4</td>
<td>46</td>
<td>78</td>
<td>5717</td>
<td>1.27</td>
</tr>
<tr>
<td>XMark</td>
<td>5</td>
<td>17</td>
<td>92</td>
<td>1810</td>
<td>1.16</td>
</tr>
<tr>
<td>XMark</td>
<td>6</td>
<td>22</td>
<td>96</td>
<td>627</td>
<td>8.9</td>
</tr>
<tr>
<td>XMark</td>
<td>7</td>
<td>28</td>
<td>87</td>
<td>3640</td>
<td>0.42</td>
</tr>
<tr>
<td>XMark</td>
<td>8</td>
<td>60</td>
<td>73</td>
<td>9184</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Table 6.2: Performance metrics. Each row contains performance metrics measurements for a specific data (XMark or DBLP) and a specific twig pattern size $2 \leq k \leq 8$. The performance metrics measurements are: the average coverage, the average improvement, the average gain and the maximal improvement.

6.5 Conclusion and future work

In this chapter, we explained and demonstrated why $TA$-index node selection is more accurate than the 1-index which is an accurate structural-index for a semi-structured data in a tree form. We suggested a straightforward way to cluster the XML tree nodes according to both a bottom-up and a top-down information. In the future, more sophisticated clustering techniques, which will capture both the bottom-up and the top-down information in a more accurate way, should be developed. The suggested $TA$-index works on unordered tree patterns. If the suggested $TA$-methodology will be extended for an ordered $TA$ then the suggested $TA$-index will be able to operate on ordered tree patterns as well. If the suggested $TA$-methodology will be extended for graph automata, then the $TA$-index will be able to operate on graphs as well.
Part III

Accessing XML in a limited memory device
Chapter 7
MinStAX XML streaming API

7.1 Introduction

XML is a common standard for the interoperable document format. As XML becomes widespread, current trends in cellular telephone networks [70], financial transaction services [71] and identity management systems [36] are pushing the adoption of XML by smart cards and other devices with very limited memory. This chapter proposes to use the MinStAX API for accessing XML in limited memory devices. MinStAX focuses on reducing the memory utilization of the XML parsing. Reduction of memory utilization by XML parsers is not a simple task. XML has a textual format, therefore, its parsing is memory intensive. XML is not a binary format because its advantages lie in its verboseness. For example, a well-formed pairs of named marking-up tags for XML elements contribute its human-friendliness and vendor-neutrality. The design goals of the MinStAX API are:

**Small footprint** is the major goal for the API design. It has three aspects: 1. Small memory footprint (the memory size should not exceed 10 KB); 2. Small storage footprint (the source code should be less than 100 KB); 3. Static allocation of memory, which reduces the memory usage when the heap structure is not constructed, eliminates RAM fragmentation and enables the application to control the XML memory utilization. A heap based XML API, which processes a highly nested XML document with large text elements on a smart card, returned unexpected semi-random allocation failures. Such errors are hard to monitor and analyze.

**Fast enough runtime** means that the parsing time of the XML API should not be sub-
MinStAX XML streaming API

stentially degraded despite memory optimizations.

**Preserving the interoperability of the XML documents in open environments:** XML is mainly used in open environments, where there may be many participants conforming to the XML specifications [32]. An optimized uncompressed XML parser should neither reject nor broke by the optimization of XML messages in unexpected forms.

**Seamless support of standard XML and propriety XML** for open environment and closed environment, respectively. In close environment, propriety XML binary formats can increase the optimized memory utilization of XML. This chapter suggests a API that handles compressed XML in closed environments.

MinStAX was implemented on G&D company [2] smart cards platform. A design constraint of MinStAX was to implement the API on JavaCard [22]. JavaCard is the tiniest of Java targeted for embedded devices. G&D plans to deliver a JavaCard framework with their smart cards and MinStAX supposes to be in it.

We redesigned and re-implemented the Java open source of the StAX API. We choose to re-implement an existing API in order to preserve the interoperability of the XML documents by using a source code that is already running in many open environments. The redesign, which met the first three goals, included: 1. Removal of large StAX footprint features; 2. Redesign the internal data structures; 3. Using text compression. Section 7.2.1 describes the removal of StAX features. Section 7.3 describes the redesign of the data structures and section 7.4 describes the text compression for a standard XML parsing.

The fourth design goal of MinStAX is a seamless support for standard and propriety XML formats. In this thesis, we implement a propriety compressed format of XML. The compression scheme, which is used to compress the propriety XML binary format, is called Tagged Suboptimal Compression (TSC) [27]. TSC is natural for compression of XML on smart cards since it uses a limited amount of memory to produce a reasonable compression rate and since it enables to parse in the compressed domain. Figure 7.1 provides an example for a seamless support of both standard XML and propriety TSC XML over HTTP. The smart card sends an HTTP request for an XML based message with a propriety TSC encoding. If the WEB Server supports the propriety encoding

**Part III: Accessing XML in a limited memory device**
then the propriety XML format is sent as a response. Otherwise, the standard XML message is sent as a response. Section 7.4 describes how parsing of propriety format is supported.

![Diagram](image)

Figure 7.1: An example of a seamless support of both standard XML and propriety TSC XML over HTTP

### 7.2 XML background and related work

#### 7.2.1 XML parsers classifications

This section describes different XML parsers characteristics and explains the parsers characteristics of MinStAX. XML is a set of rules for encoding documents electronically. It has a textual data format which defines the structure of a document via a set of nested textual tags called elements. XML specification defines an XML document as a text which is well-formed, i.e., it satisfies a list of syntax rules provided in the specification. The list is fairly lengthy; some key points are: 1. It contains only properly-encoded legal Unicode characters; 2. None of the special syntax characters such as “<” and “&” appear except when performing their markup-delineation roles; 3. The begin, end, and
empty-element tags, which delimit the elements, are correctly nested with none missing and none overlapping; 4. The element tags are case-sensitive; the beginning and end tags must match exactly; 5. There is a single ‘root’ element which contains all the other elements. The definition of an XML document excludes texts which contain violations of well-formed rules; they are simply not XML. An XML API, which encounters such a violation, is required to report such errors and to cease normal processing. Checking if rules are well-formed is a major task of an XML-API

MinStAX degrades the interoperability by restricting the Unicode to be UTF-8 \[147\], which is a variable-length character encoding for Unicode. UTF-8 is able to represent any character in the Unicode standard, yet it is backwards compatible with ASCII. For these reasons, it is becoming the preferred encoding for e-mail, WEB pages and other places where characters are stored or streamed. The restriction to UTF-8 saves memory because it enables to represent characters as bytes (8 bits) and not as Java Chars (32 bits). This enables to reduce the internal data structures size by four.

In addition to being well-formed, an XML document may be valid. This means that the XML document contains a reference to a schema and its elements follow the grammatical rules which the schema specifies. XML APIs are classified as validating or non-validating depends on whether or not they check XML documents for validity. The validation consumes memory since all the declarations of the schema must be stored in memory during the validation. Therefore, MinStAX API is non-validating.

XML defines entities, which are simply a data-replacement facility. Any entity can be internal or external. An external entity points to data in an external file, so it functions like an “import” or an “include” statement. The value of an internal entity is defined in-line, so it functions like a macro. External entities consume memory because internal buffers are kept for each entity. XML API is classified as “standalone” if it does not allow external entities. MinStAX is a standalone XML API.

XML defines name prefixes called namespaces in order to resolve tag names conflicts. A namespace is defined by the ‘xmlns’ attribute in the start tag of an element. The namespace declaration has the syntax ‘xmlns:prefix=URI’. When a namespace is defined for an element, all nested elements with the same prefix are associated with the same namespace. Defining a default namespace for an element saves us from using prefixes in all the child elements. Default namespace has the syntax ‘xmlns=URI’. A major task of XML API is to support namespaces and in particular default namespaces.

Part [III]: Accessing XML in a limited memory device
MinStAX XML streaming API returns each element tag part as a Qualified Name (QName). A QName is the tuple \((\text{name}, \text{prefix}, \text{URI})\) where name is the element part name, prefix is the prefix of the namespace, and URI is the namespace URI.

XML API supplies access and serialization methods for XML documents. An XML API parses the XML document in three parsing steps. In the first two steps, character conversion and lexical analysis are usually invariant among different parsing models, while the third step, syntactic analysis creates data representations that is based on the used parsing model. These data representations result in different operational and performance requirements. XML-based DB and networking applications have unique requirements with respect to access and modification of the parsed data. Database applications must be able to access and modify the document structure back and forth; the parsed document resides in the DB server to receive multiple incoming queries and update instructions. The dominant data representation for DB access is a tree. The most popular API for DB application is Document Object Model (DOM) \([1]\). Networking applications rely on one-pass access and modification during parsing; they pass the unparsed document through the node to match the parsed queries and update instructions that reside in the node. The dominant data representation for network applications access is a sequence of events. The most popular API for network application is the Simple API for XML (SAX) \([3]\). The MinStAX API goal is to supply accesses methods for network applications in limited memory devices. Therefore, an implementation of an event-based API is a natural choice. Moreover, event-based APIs require a small memory footprint that matches our goals.

To summarize this section, we can define the MinStAX API as a non-validating standalone event based XML API that checks the well-formedness of the accessed and serialized XML documents.

### 7.2.2 An introduction to StAX

Event-based APIs like SAX are push-based. The application registers to the API in order to receive events. As elements are encountered within the source document, the API pushes events to the user application. On the other hand, a tree-based API allows a random access to the document. One can think of these two access metaphors as polar opposites. A tree-based API allows unlimited and random access and manipulation, while an push-based API is “one shot” pass through the source document.
StAX was designed as a median between these two opposites. In the StAX metaphor, the programmatic entry point is a cursor that represents a point within the document. The application moves the cursor forward (i.e., “pulling” the information from the parser as it needs). This is different from a SAX like API, which “pushes” data to the application, requiring the application to maintain a state between events as necessary to keep track of the location within the document.

StAX uses the pull approach. The developer requests events rather than having event information from StAX that is pushed onto the user application. This results in a more natural and readable code without sacrificing neither memory nor performance. Figure 7.2 shows an example of the StAX API operation. Figure 7.2(a) describes an XML document. This XML carries two information types: about a piece of furniture and about an HTML table. Both information use the table element. The tag name conflict is resolved by adding namespaces to the furniture and to HTML sections. The furniture section defines a default namespace. The HTML section defines a namespace with prefix ‘h’. Figure 7.2(b) illustrates the output from the StAX API when this XML document is processed.

Part III: Accessing XML in a limited memory device
1. <root> 
  SE: root
2. <table > 
  SE: table
3. xmlns="http://f">
4. <name> 
  SE: name
5. Coffee Table  
  T: Coffee Table
6. </name> 
  EE: name
7. <width> 
  SE: width
8. 80   
  T: 80
9. </width> 
  EE: width
10. <length> 
  SE: length
11. 120 
  T: 120
12. </length> 
  EE: length
13. </table> 
  EE: table
14. <h:table > 
  SE: table
15. xmlns:h="http://h" 
16. border="1">
17. <h:tr> 
  SE: tr
18. <h:td> 
  SE: td
19. Apples 
  T: Apples
20. </h:td> 
  EE: td
21. <h:td> 
  SE: td
22. Bananas 
  T: Bananas
23. </h:td> 
  EE: td
24. </h:tr> 
  EE: tr
25. </h:table> 
  EE: table
26. </root> 
  EE: root
(a) (b)

Figure 7.2: (a) An XML document where each line begins with its line number. (b) A description of the events that are pulled when the XML document in (a) is parsed by the StAX API. Each line is in the format ‘Event Type: Element name’ where the event type is either Start Element (SE) or, End Element (EE) or Text(T).

Figure 7.3(a) describes the processing steps of XML APIs as was detailed in [90]. Figure 7.3(b) describes the main software modules of the StAX API and relates them to the process steps Fig. 7.3(a). Figure 7.3(b) describes two processing use cases: access and serialization. The first use case is accessing an XML document via the API. The XML document is handled by the Entity module which caches the document in an internal buffer. The Entity uses the Java API to read the document and to

Part III: Accessing XML in a limited memory device
perform Unicode character conversions. StAX supplies the \textit{XMLChar} module with additional handling of Unicode characters. The \textit{XMLChar} module supplies functions that check the type of a Unicode character \( c \). The functions check whether \( c \) can compose an element name, a free text or whether \( c \) is a space character. Reduction of the Unicode to \textit{UTF}-8 substantially reduces the amount of memory that is allocated by the \textit{XMLChar} module. The lexical analysis involves partition of the characters stream into subsequences called tokens. The lexical analysis is separated between two modules. The \textit{DocumentScanner} maintains an ‘outer’ state machine that decides on the type of the next token which appears in the stream. The \textit{EntityScanner} module supplies for each token type a function that extracts a token according to an ‘inner’ state machine. The \textit{DocumentScanner} also applies the syntactic analysis by pushing and popping elements to \textit{Elements} and \textit{Namespaces} stacks. The \textit{DocumentScanner} uses the \textit{Elements} stack to check if begin and end tags, which delimit the elements, are correctly nested. The \textit{DocumentScanner} uses \textit{Namespaces} stack to refer to the current name space. The top of the \textit{Namespaces} stack contains the current namespace. The \textit{XMLReader} model supplies the API itself. It exports data structures, which are filled by the \textit{DocumentScanner} lexical analysis, to the user application.

The second use case is serialization of XML document via the API. The user application calls the \textit{XMLWriter} API in order to write XML tokens into a document. The \textit{XMLWriter} uses the \textit{Elements} stack to check the well-formedness of the document. The \textit{XMLWriter} also uses the \textit{Namespaces} stack to serialize the default namespace. The \textit{XMLWriter} also performs a simple syntactic analysis. It uses an “outer” state-machine to validate the type of the given token. The serialization is not needed to preform a complete lexical analysis. Tokens are given to it. The tokens are serialized into XML documents via the \textit{Entity} module.
7.3 Data structures redesign

The StAX internal data structures include Elements and Namespaces stack. They also include the XMLAttributes and the Symbol Table. The XML Attributes store the current attributes names and values. The Symbol Table is a hash table that stores...
all the tokens of the XML structure: elements, attributes, namespaces URIs etc. The goal of the Symbol Table is to increase the processing speed by enabling fast object based compressions instead of string based comparisons. When a token is analyzed from a document or given by the user application it is first matched against the symbols table. A reference to the string object, which matches the token, is given to the API. Then, all comparisons are made against the returned reference. The StAX counts on the garbage collection to manage references handling. Figure 7.4 illustrates the data structures of StAX. The Elements stack contains the object references of the qualified names that nests the currently parsed element. In Fig. 7.4 there are two references to QNames (‘root’, ) and (‘table’ ‘h’, ‘http://h’). The Namespaces stack contains the pairs (prefix, URI) where prefix and URI are declared by an ‘xmlns’ attribute of an element that nests the currently parsed element. The Namespaces stack in Fig. 7.4 contains a single pair (‘h’, ‘http://h’). The XML Attributes stores two attributes QNames: (‘xmlns’, ‘h’, ‘http://h’) and (‘border’, ‘h’, ‘http://h’). The Symbol Table contains all the referenced tokens: ‘border’, ‘h’, ‘http://h’, ‘root’, ‘table’, ‘xmlns’. But the Symbol Table also contains tokens that are not referenced by any data structure. Objects like ‘name’, ‘f’ and ‘http://f’, which were referenced by the data structures in the processing of previous tokens, still exist in Symbol Table. StAX symbol table is an “insert only”. Symbol Table does not remove objects, therefore, it becomes big.
Figure 7.4: Illustration of StAX internal data structures after processing the XML document up to line 16 in Fig. 7.2(a). The Elements and Namespaces stacks and the Symbol Table hash table are denoted by a column of boxes. The XML Attributes array is denoted by a row of boxes. Arrows denote object references. A box with a dashed line contains a tuple where each internal dash box denotes a tuple component.

The internal data structures redesign includes: 1. A redesign of the Symbol Table; 2. Replacing the object references by numeric indexes; 3. Making the internal data structures static. Figure 7.5 illustrates the internal data structure of MinStAX. The
memory consumption of StAX’s original Symbol Table is too big to be implemented on smart cards. We considered an option of removing the Symbol Table and to use string comparisons instead. But this option can degrade the runtime performance of the stack. Instead, we enable objects removal from the Symbol Table. The MinStAX stores in the Symbol Table only objects that are currently referenced by other data structures. These elements, which nest the currently parsed element, are referenced by the internal data structures. Therefore, the number of symbols in the Symbol Table depends on the number of elements that nests the currently parsed element. This number is typically small. Therefore, the updated Symbol Table can be implemented in smart cards. The Symbol Table still enables object based comparisons but there is a runtime overhead in removing objects from the stacks each time a namespace and an element are popped from the Namespaces and Elements stacks, respectively.

Another change in the Symbol Table is the method how string objects are stored. The StAX stores each string object separately. MinStAX, on the other hand, stores all the strings in a single byte array. By storing only tokens of elements, in which the current parsed element is nested, we enable to manage all the strings together in a stack-like structure. Each time a token is inserted to the Symbol Table, it is pushed to the SymbolsString stack. Each time a token is removed from the Symbol Table, it is popped from the SymbolsString stack. The Symbol Table entry in MinStAX contains the offset of the token in the SymbolsString and the tokens sizes. Table 7.1 shows an example of the stack of the SymbolsString for the processing of XML document in Fig. 7.2(a).
<table>
<thead>
<tr>
<th>Event</th>
<th>Line</th>
<th>SymbolsString</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE:root</td>
<td>1</td>
<td>root</td>
</tr>
<tr>
<td>SE:tr</td>
<td>17</td>
<td>root.xmns.h.<a href="http://h.table.tr">http://h.table.tr</a></td>
</tr>
<tr>
<td>SE:td</td>
<td>18</td>
<td>root.xmns.h.<a href="http://h.table.tr.td">http://h.table.tr.td</a></td>
</tr>
<tr>
<td>SE:td</td>
<td>21</td>
<td>root.xmns.h.<a href="http://h.table.tr.td">http://h.table.tr.td</a></td>
</tr>
</tbody>
</table>

Table 7.1: An example of the stack structure of the SymbolsString for processing of the XML document in Fig. 7.2(a). The Event column describes the start events, which the StAX outputs, as appeared in Fig. 7.2(b). The Line column describes the lines in the XML document which originated the event. The SymbolsString column describes the content of the SymbolsString byte array after the SE event occurred. For user readability, the tokens are delimited by dots characters which are not part of the SymbolsString byte array.

The stack structure of the SymbolsString enables fast update of the Symbol Table. It also makes the implementation of the Symbol Table a static data structure straightforward. A disadvantage of the SymbolsString implementation is that we can not use the same SymbolsString for access (XMLReader) and for serialization (XMLWriter). The combination of insertion and deletion of both parsed tokens and serialized tokens can break the First In First Out (FIFO) order of tokens insertion and deletion to and from the Symbol Table. Here, we prefer the speed over memory size. This is acceptable since the average size of the Symbol Table is a couple of hundreds KB. Another minor disadvantage is that we can not use the garbage collection to manage a reference count. Instead we manage internal reference count. Each Symbol Table entry contains a reference count. A token is pushed to SymbolsString and popped from SymbolsString only when the reference count is zero. All the dynamic internal data structures are redesigned as static data structures. The Symbol Table is implemented as a close universal hash-table over a byte array. The Elements, Namespaces and SymbolsString stacks and the XML Attributes vector are implemented as byte arrays with predefined sizes. Implementing data structures with predefined sizes degrades the interoperability because it limits the number of nested elements, the number of de-
clared namespaces and the number of attributes in an element. When a data structure is full then MinStAX outputs an error. On the other hand, using dynamic data structure is problematic because when the memory limits are exceeded then a general error is outputted which do not explain the nature of the problem. Predefined sizes give the smart card application a way to control its memory allocation. All the object references to the \textit{Symbol Table}, which the data structures contain, are replaced by the numeric indexes of the entries in the close hash table. The entries refer to tokens in the \textit{SymbolsString}. The object based comparisons are replaced by numerical comparisons of the indexes.

In the MinStAX implementation, we limited the \textit{Symbol Table} size to 255 and used numeric byte indexes. This reduces the internal data structure sizes by a factor of eight. Instead of long (64 bit) object reference, the data structures store byte (8 bit) indexes. Limiting the \textit{Symbol Table} size to 255 degrades the interoperability because highly nested XML documents can not be parsed by MinStAX even if each element defines its own namespace then nesting of more then eighty elements, which is a lot, is possible. Figure 7.5 illustrates the data structures of MinStAX. The \textit{Elements} and the \textit{XML Attributes} contain QNames \((\text{name}, \text{prefix}, \text{URI})\) where \text{name}, \text{prefix} and \text{URI} are numeric indexes for entries in the \textit{Symbol Table}. The \textit{Namespaces} stack contains the pairs \((\text{prefix}, \text{URI})\) where \text{prefix} and \text{URI} are numeric indexes of entries in the \textit{Symbol Table}.

An entry of MinStAX \textit{Symbol Table} is a tuple \((\text{offset}, \text{length}, \text{count})\) where the \text{offset} is the start position of the token in \textit{SymbolsString}, \text{length} is the length of the token and \text{count} is the reference count of the token. For example, entry 4 in the \textit{Symbol Table} in Fig. 7.5 is the tuple \((15, 8, 4)\) which refers to the substring ‘http://’ in positions 15..22 in \textit{SymbolsString}. The \text{count} = 4 indicates on four references by the other data structures: one by \textit{Namespaces} stack, one by \textit{Elements} stack and two references by the \textit{XML Attributes}. The MinStAX \textit{Symbol Table} does not contain entries of tokens which are not referenced by any data structure. Objects like ‘name’, ‘f’ and ‘http://f’, which exist in StAX \textit{Symbol Table} in Figure 7.4 do not exist in the MinStAX \textit{Symbol Table}.

Part III: Accessing XML in a limited memory device
Figure 7.5: Description of MinStAX internal data structures after processing up to line 16 of the XML document in Fig. 7.2(a). The Elements, Namespaces and Symbols String stacks and the Symbol Table hash table are denoted by a column of boxes. The XML Attributes array is denoted by a row of boxes.

The final change is in the `javax.xml.namespace.QName` class which encapsulates qualified names. The StAX API exports qualified names of elements and attributes via the `QName` class. The `QName` class contains the tuple `(name, prefix, URI)` of Java strings. The StAX `QName` instances contain references to string objects in the Symbol Table. In order to supply the same methodology as in MinStAX, we create `StringPart` as a new class. `StringPart` data members are a byte array, an offset and a length. The `StringPart` handles the sequence of bytes `offset...offset + length` in the array as a string. The `StringPart` supplies all the string functions. Function like `compareTo` and `indexOf`, which do not change the data members, are applied to the `StringPart` object. Function like `concat`, which changes the data members, returns a new string object. MinStAX rewrites `QName` to contain the tuple
(name, prefix, URI) as StringParts. The name, prefix and URI StringParts are initialized by the data in the Symbol Table. The byte array is set to the SymbolsString reference. The offset and length are assigned by the offset and the length of the entry, which contains the relevant token, in the Symbol Table. In this way, content search, which is returned by the MinStAX API, does not allocate new memory. Only when the content, which is returned by the API, is changed, then a new string object is allocated. This API change does not affect the interoperability of the API because String and StringPart have the same functionality.

### 7.4 Data compression

Standard compression together with propriety XML compression that is based on a new XML document parsing were used. In a standard XML parser, text compression is needed because the size of a single text element in a typical SOAP based communication can reach to be more than 5KB. This size is problematic when having 10KB RAM for the whole XML API processing. In order to reduce the texts sizes, the standard XML parsing encodes the text by using TSC, which is a fast Huffman-like compression method. TSC consumes less than 0.5 KB RAM of internal data structure and its entropy is 4.3 bits per character for English texts. We used a semi-static version of TSC. The symbols frequencies are updated after every character encoding and the symbols codes are updated according to these frequencies only at the end of each document parsing. We assume that the two sequentially parsed XML documents are highly correlated. Therefore, characters frequencies of the current parsed document foretell the characters frequencies of the next document to be parsed. The TSC compression enables MinStAX to store 5 KB text elements. Compression of 5 KB of text by TSC saves up to 2 KB RAM. TSC enables MinStAX to process XML documents with fairly large text elements. Of course, compressing and decompressing texts slow down the processing rate of MinStAX. In order to avoid decompression, MinStAX applies the TSC which enables Compressed Pattern Matching (CPM). CPM is the process of pattern searching in a compressed file without decompression of the compressed file. CPM enables to search 5KB long text elements without decompressing them. We developed a CompressedString class that supplies the Java String an API for handling strings which were compressed by TSC. Function like ‘compareTo’ and ‘indexOf’, which do
not change the string, are applied to the compressed domain. Function like ‘concat’, which changes the string, returns a new string object.

In order to support TSC with the propriety XML parsing, we created the modules CompressedEntity, CompressedSymbolTable and CompressedSymbolsString. The Entity and the CompressedEntity supply a common read interface that reads bytes from the current position. The CompressedEntity decompresses the bytes from the encoded XML document stream. The EntityScanner was updated to support the common read API. When a compressed token is parsed by EntityScanner, it is searched in the CompressedSymbolTable as a compressed token. The CompressedSymbolTable hash function is applied to the compressed token. The CompressedSymbolsString represents a binary string. Each entry in the CompressedSymbolTable is the tuple \((offset, length, count)\) where the \(offset\) and \(length\) refer to a substring of bits \(offset \ldots offset + length\) in the CompressedSymbolsString that contains the compressed token. If the compressed token is not found then it is pushed to the CompressedSymbolsString as a substring. Figure 7.6 describes the compressed data structures. We can see that the XML Attributes, the Namespaces and the Elements stacks are not changed in comparison to the data structure of the standard XML parsing in Fig. 7.5. The QName is updated in order to represent the tuple \((name, prefix, URI)\) as CompressedStringParts. The CompressedStringPart handles the sequence of bits \(offset \ldots offset + length\) in the bits array as a CompressedString.
Figure 7.6: Description of the compressed MinStAX internal data structures after processing the compressed version of the XML document up to line 16 in Fig. 7.2(a). The Elements, Namespaces and CompressedSymbolsString stacks and the CompressedSymbolTable hash table are denoted by a column of boxes. The XML Attributes and CompressedEntity array are denoted by rows of boxes.

### 7.5 Experimental results

We implemented all the algorithms in Java 6. All our experiments were performed on a PC with 2.4GHz Pentium 4 processor with 1024MB RAM running Windows XP. We used Sun XMLTest benchmark [5]. XMLTest is an XML processing test which is designed to mimic the processing that takes place in the life cycle of an XML document that typically involves the following steps: parse, access, modify and serialize. The XMLTest dataset is based on a business invoice document. XML documents in formats like UBL [4] are part of the XMLTest dataset.

We updated the XMLTest to work with MinStAX. We tested MinStAX and StAX processing time where Processing = Parse + Access + Serialize. We compared between the total processing time of StAX and MinStAX on the XMLTest dataset. Standard XML processing of MinStAX is three times slower, on average, than StAX processing time. The compressed XML processing time of MinStAX is seven times slower, on average, than StAX processing time. We used YourKit 7.0 [6] to measure the average sizes of the allocated memory. StAX allocates 7.3 MB of RAM during the processing of XMLTest dataset. When processing XMLTest dataset standard doc-
documents, MinStAX allocates 7.6KB of RAM. When processing XMLTest compressed documents, MinStAX allocates 5.8KB of RAM.

7.6 Conclusions and future work

In this chapter we introduced the MinStAX, which is a memory efficient redesign of the Java StAX API. MinStAX supports both standard and propriety XML formats. The propriety XML format is encoded by TSC. In the current implementation, the compressed data is decompressed by the CompressedEntity in the read function. But TSC supports CPM, which enables to parse XML document in the compressed domain. In the future, we intend to update the EntityScanner in order to parse the compressed XML document without decompressing it first. We plan to implement a fast byte-wise CPM that is suggested in 8. We hope to dramatically improve the run time of MinStAX when compressed XML documents are processed.
Chapter 8

Tagged Suboptimal Code (TSC)

8.1 Introduction

This chapter shows how to compress (encode) losslessly, search and decompress (decode) textual data in a machine/device that has a limited memory (several kilobytes). Data processing is assumed to take place in a steaming mode. Development of a fast compression scheme that fits limited memory and supports of text search are a non-trivial task. This thesis proposes such a technique.

The goal of such a compression method is to enable parsing of textual streams in a limited memory device. For example, assume we have a micro-processor with only several kilobytes of RAM which parses a TCP/IP stream that contains an XML message. The microprocessor compresses the XML message and then searches for specific XML tags in the compressed text (See Chapter 7). The compression enables the microprocessor to process large XML messages without affecting its limited memory utilization. The compressed domain pattern matching enables the microprocessor to parse large XML messages.

State-of-the-art text compression techniques cannot operate with several kilobytes of memory. For example, dictionary based methods such as LZW [148] cannot support a dictionary data structure on a small size memory. Context based methods such as PPM [46] cannot maintain contexts. Even block-sorting compression algorithm [59] cannot be used because the block sizes, which the algorithm uses, are too big (100 KB and above).

The constraint of a limited memory barely suffices to store the frequencies of the alphabet. Therefore, we need a prefix-code that supports text searching in a compressed domain. The Tagged Suboptimal Code TSC [27] is a near-optimal prefix-code that was

Part III: Accessing XML in a limited memory device
Tagged Suboptimal Code (TSC) designed for this task. TSC code-words boundaries are identified instantaneously. This feature enables the TSC to perform pattern matching in the compressed domain.

In this chapter, we show how to represent integers by a TSC code. We prove that TSC is a universal code. We introduce the TSC\textsuperscript{k} family of universal codes where TSC\textsuperscript{0} is the original TSC. Instead of constructing an optimal-code such as Huffman, we choose the best near-optimal-code from the TSC\textsuperscript{k} family of universal prefix-codes. Both TSC\textsuperscript{0} and TSC\textsuperscript{1} fit English text compression. Experiments show that TSC\textsuperscript{1} code compression scheme improves the original TSC\textsuperscript{0} compression rate. We introduce a fast decoding technique that uses compact tables in order to decode the compressed data as bytes. We adopt the Aho-Corasick (AC)[14] pattern matching algorithm to operate on the TSC\textsuperscript{k} compressed domain. The adopted AC algorithm uses the same compact tables, which are used in the decoding process, to perform a fast pattern matching in the TSC\textsuperscript{k} compressed domain. We also introduce a new adaptive text compression scheme that maps characters to universal codes.

All these enhancements increase the compression rate to enable fast pattern matching in the compressed domain. These enhancements improve simultaneously the speed and the compression quality of the codec without increasing substantially its memory utilization. These enhancements enable textual stream parsing in limited memory devices. A smart card is one typical device that has a limited memory.

The rest of the chapter has the following structure. Section 8.2 provides preliminaries and related work. Section 8.3 describes the TSC compression algorithm. Section 8.4 describes the compressed string matching algorithm. Section 8.5 provides theoretical analysis of the compression. Experimental results of the compression and the compressed string matching algorithms are given in section 8.6. The appendix in section 8.8 proves a tight bound less than 2 for the optimal Huffman coding.

8.2 Preliminaries and related work

8.2.1 Encoding

This chapter presents a new family of prefix-codes. A prefix-code, which uniquely maps an alphabet symbols to code-words, is given in definition 8.1.

**Definition 8.1** Given an alphabet \( a_1, \ldots, a_n \). The encoding constructs a code \( c_1, \ldots, c_n \) by translating \( a_i \) into \( c_i \) whereas the decoding translates \( c_i \) back into \( a_i \). A prefix-code is a code \( c_1, \ldots, c_n \) that satisfies the prefix property: code-word \( c_i \) is not a prefix of any other code-word \( c_j \) when \( i \neq j \).

Part III: Accessing XML in a limited memory device
Not all prefix-codes transformations compress data. Optimal codes define prefix-codes that are best for compression. Definition 8.2 defines an optimal-code.

**Definition 8.2** Given an alphabet $a_1, \ldots, a_n$ and a set of probabilities $p_1, \ldots, p_n$. The optimal-code $c_1, \ldots, c_n$ is a prefix-code which brings the weighted average code-word length $\sum_{1 \leq i \leq n} (p_i \cdot |c_i|)$ to minimum where $|c_i|$ is the size in bits of the code-word $c_i$. □

This chapter examines the TSC coding that is near-optimal for English text. Huffman encoding is a well known method to produce optimal-codes according to alphabet frequencies. Huffman encoding, which is used in this chapter, is given in definition 8.3.

**Definition 8.3** An alphabet $a_1, \ldots, a_n$, where $a_i$ is the $i$-th most common symbol, is assigned with a monotonic probability distribution $p_1, \ldots, p_n$. **Huffman encoding** constructs an optimal-code $c_1, \ldots, c_n$ by a labeled tree $T = (V, E, \text{label})$ where $\text{label} : E \mapsto \{0, 1\}$. Each symbol $a_i$ is attached to a leaf in $T$. The symbol $c_i$ is a concatenation of $\text{label}(e)$ of edges from the root to the $i$-th leaf. □

The TSC codes are represented in this chapter as universal codes. A universal code encodes positive integers as defined in definition 8.4.

**Definition 8.4** Given positive integers $1, \ldots, n$ and a set of probabilities $p_1, \ldots, p_n$. A **universal-code** is a prefix-code $c_1, \ldots, c_n$ that satisfies two conditions: 1. $i$ is mapped into $c_i$ for every $i = 1, \ldots, n$. 2. If the probability distribution is monotonic ($p_i \geq p_{i+1}$, $1 \leq i \leq n$) then there is a $k$ such that the optimal-code $c_1^0, \ldots, c_n^0$ with $p_1, \ldots, p_n$ satisfies $|c_i| \leq k, 1 \leq i \leq n$. Each universal code has an implied probability for which it is optimal and $|c_i| = |c_i^0|$, $1 \leq i \leq n$. □

A well known example for a universal code is the **Elias gamma code** [55]. Coding a number $n$ by a gamma code is done in the following way: 1. $n$ is written in a binary representation; 2. 1 is subtracted from the number of bits written in step 1 and that many zeros are padded. For example, the number 3 is encoded as 011 because 11 is the binary representation of 3 and a single 0 is padded. Our experiments show that the TSC compression rate is higher than the gamma codes compression rate.

Neither TSC nor Huffman encoding are the state-of-the-art compression techniques for text. The state-of-the-art text compression techniques are primarily being used to store large amounts of data on high-end servers with large sizes of installed RAM. Therefore, memory utilization is not considered by these algorithms. In many cases, the memory usage of these state-of-the-art compression techniques is $O(N)$ where $N$ is the size of the uncomressed text $T'$. Dictionary based methods such as LZW [148] maintain a dictionary data structure that is of $O(N)$ size. Context based methods, such as PPM* [46], maintain unbound contexts of the current symbol. Contexts data structure are of size $O(N)$. Even the size of blocks in the **block-sorting** compression algorithm

**Part III**: Accessing XML in a limited memory device
is big (100 KB and above). These techniques do not fit a limited memory device
that operates in a streaming mode.

One technique that may fit compression in limited resources devices is the Byte Pair
Encoding (BPE) [65]. BPE is a simple form of data compression which supports fast
compressed string matching. BPE replaces the most common pair of consecutive sym-

bols of data with a symbol that does not occur within that data. A table of replacements
is required to rebuild the original data. For example, the string “aabaab” is translated
into the string “XbXb” after replacing ‘aa’ with X. The string “XbXb” is translated into
string “YY” after replacing ‘Xb’ with Y. The BPE encoding algorithm is a multi-pass
and requires that all the data must be stored in memory. Therefore, the algorithm cannot
handle streams. Buffering small blocks of data and compressing each block separately
partially solves this problem. We compare the TSC against BPE with block sizes of 1
KB that fit limited memory processing. The decoding rates of BPE are similar to these
of TSC but the encoding rate of BPE is far slower than the encoding rate of TSC.

8.2.1.1 Tagged Suboptimal Code (TSC)

TSC is a suboptimal coding technique that can be used as a general-purpose compres-
sion scheme. The code-words are generated from a tree traverse. The tree can be rep-
resented by a quad tree (degree 4) as shown in Fig. 8.1. The code-words are generated
in every level. Each edge in the tree contributes two bits to the symbol. The first two
code-words, which are in level 1 (length of a code-word), are 01 and 10. The next
level generates four different code-words. Branching the tree with 00 or 11 bits is not
considered as an end of a code-word. Therefore, the next pair of bits must be included
in the code-word until the sequence of bits are delimited with 01 or 10 bits. TSC is
a variable-length prefix-code. The code-words sizes depend upon the number of rep-
resented codes (2 to 14 bits long for 254 codes). The TSC code-words are ‘tagged’
because its boundaries are instantaneously detected.
8.2.2 Compressed string matching

String matching problem is well researched and developed. Given a pattern $P$ and a much larger text $T'$, pattern matching locates the first or all the occurrences of $P$ in $T'$. The state-of-the-art string-matching algorithms such as [25] requires a preprocessing of $T'$. Therefore, it does not fit streaming. An efficient string matching algorithm, which does not preprocess $T'$, is the **Boyer-Moore** (BM) string search algorithm [30]. The execution time of BM algorithm can be sub-linear: it does not require that every character in the string is checked. It can skip some of the characters. BM uses the information, which is gained from unsuccessful attempts to find a match, to rule out as many positions of the text as possible where the string cannot be matched. Other known algorithms like **AC** ([14]) use FSMs to match $P$ in $T'$ in linear time.

The compressed string matching problem was introduced in [17]. It is a variant of the string matching problem in which the text $T$ is in some compressed form. Compressed string matching algorithms were suggested in many encoding techniques. Algorithms for compressed string matching, which use the run length compression, are proposed in [54]. An algorithm for a compressed string matching, which uses a variant of the adaptive LZW compression algorithm, is given in [17]. Algorithms for compressed string matching, which use word-based Huffman compression, were suggested in [131][33]. The compressed string matching, which uses Huffman compression, adopts pattern-matching algorithms such as [30, 14] and others to search in a compressed domain. They are considered as state-of-the-art.

Two goals for compressed string matching were stated in [131]: 1. Perform it faster than decompression of the compressed text $T$ into the uncompressed text $T'$ followed by a search of $P$ in $T'$; 2. Perform it faster than searching $P$ in $T'$. Compressed string matching, which uses word-based Huffman compression, is attractive because
it achieves goal 2. Another compressed string matching, which achieves goal 2 and operates by using word-based BPE compression, was suggested in [131].

Due to the number of symbols in a word-based compression, it is impractical for use in limited memory devices. Therefore, we can not achieve goal 2. In this chapter, we focus on achieving goal 1. In order to achieve goal 1, [131] states two problems that a compressed string matching method must overcome: 1. **Synchronization** between the pattern and the text code-words; 2. **Byte-wise processing**. A compressed string matching that uses TSC, which is described in [27], do not supply a byte-wise processing. The suggested compressed string matching that uses TSC, which is suggested in this chapter, overcomes these problems while supplying a byte-wise processing.

The compressed string matching that uses Huffman compression, which is suggested in [131], constructs transition tables to perform byte-wise processing. Similar transition tables construction is also suggested in [132]. The size of these transition-table is big. Therefore, a byte-wise compressed string matching, which uses Huffman compression, is impractical for use in a limited memory device. In this chapter, we present a byte-wise compressed string matching that uses TSC and compact transition tables. Therefore, it fits a device with a limited memory.

The TSC algorithm has an additional desired feature. The TSC code-words are tagged which means that their boundaries are immediately detectable. When the code-words are tagged, we can adopt the BM algorithm as a compressed string matching algorithm because the tagged code-words enable code-words skipping. Therefore, we can achieve a sub-linear run time. Unfortunately, not all the $TSC^k$ codes produce tagged code-words. Therefore, in this chapter, we adopt the $AC$ as a string matching algorithm that uses TSC instead of BM.

### 8.3 The TSC compression algorithm

In this section, we describe a fast and compact text compression algorithm. Section 8.3.1 formalizes the TSC as a universal code. Section 8.3.2 extends TSC into a family of universal codes. Section 8.3.3 describes a fast and compact TSC decoding mechanism. Section 8.3.4 describes how to use a static and an adaptive text compression schemes with universal codes.
8.3.1 TSC as a universal code

We consider TSC as a universal code. An integer number \( n \) can be transformed into TSC codes in a straightforward way by using a composition of the \textit{trim} and \textit{transform} functions.

The TSC code-word \( c_i \) represents the number \( n = i - 1 \) with \( \lceil \log_2 i + 1 \rceil \) bits. The integer \( n \) is represented by \( \lfloor \log_2 i \rfloor \) bits where \( \lfloor \log_2 i \rfloor - \lceil \log_2 i + 1 \rceil \triangleq \begin{cases} 0 & \text{when } i = 2^k \text{ or } i = 2^k - 1 \\ 1 & \text{otherwise}. \end{cases} \)

From the above we know that if \( n = 2^k - 2 \) or \( n = 2^k - 1 \) then both the integer bit representation of \( n \) and the TSC code-word \( c_{n+1} \) represent the same number of bits. Therefore, no bit in \( n \) is trimmed. Otherwise, the most significant bit in \( n \) is trimmed. When \( n = 2^k - 2 \) or \( n = 2^k - 1 \) then \( n \) has the binary format \( 1^+(1|0) \). Definition 8.5 formalizes the \textit{trim} function.

**Definition 8.5** Given a binary string \( s \in \{0,1\}^* \) where \( s = s_1, \ldots, s_k \) is a binary representation of an integer \( n \) such that the most significant bit is \( s_1 = 1 \). If \( s \) is in the format \( 1^+(0|1) \) then the function returns \( s \). Otherwise, the \textit{trim} function trims the most significant bit \( s_1 = 1 \) and returns \( s_2, \ldots, s_k \). The inversion function \( \text{trim}^{-1} \) pads \( s \) with \( s_1 = 1 \) if \( s \) is not in the format \( 1^+(1|0) \).

The \textit{trim} function satisfies \( \text{trim}(\text{trim}^{-1}(s)) = s \) for every binary string \( s \) due to the fact that \( s_1 = 1 \) in the binary representation \( s_1, \ldots, s_k \) of any integer \( n \). Therefore, \( s_1 \) can be trimmed and padded.

Definition 8.6 defines the \textit{transform} function. This function replaces each bit in \( \text{trim}(n) \) with two bits. The resulted bits are a valid TSC code-word.

**Definition 8.6** Given a binary string \( s \in \{0,1\}^* \), where \( s = s_1, \ldots, s_k \). The \textit{transform} function replaces each binary symbol \( s_i \), \( 1 \leq i \leq k \), by two bits according to the following decision table:

<table>
<thead>
<tr>
<th>( s_i \backslash i )</th>
<th>( &lt; k )</th>
<th>( = k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

TSC codes can be used for compression because both functions \textit{trim} and \textit{transform} have inversion functions, therefore, \( n = \text{trim}^{-1}(\text{transform}^{-1}(\text{transform}(\text{trim}(n)))) \).

Table 8.1 illustrates the construction of TSC universal codes for the numbers 0, \ldots, 8.
$n$  trim($n$)  transform(trim($n$))
\hline
0 0 01 \\
1 1 10 \\
2 10 0001 \\
3 11 0010 \\
4 100 1101 \\
5 101 1110 \\
6 110 000001 \\
7 111 000010 \\
8 1000 111101 \\
\hline

Table 8.1: An example of TSC universal code. The example describes the code-words $c_1, \ldots, c_9$ for the numbers 0, \ldots, 8. The trim($n$) column describes the binary format of $n$. The bolded binary digits are the outputs from trim($n$). The column transform(trim($n$)) describes the encoded TSC code-words.

8.3.2 \textit{TSC}^k \textit{universal code}

TSC code is a part of a family of codes that we denote by TSC$^k$. The TSC$^k$ code-words are similar to TSC code-words except that they have an additional suffix in a fixed length. More formally, a TSC$^0$ code-word has the format (00|11)*(01|10). A TSC$^0$ code-word is a special case of a TSC$^k$ code-word $c_i \in \{0,1\}^*$ that has the format (00|11)*(01|10)(0|1)$^k$. In this perspective, the TSC code is a TSC$^0$ code. Lemma 8.1 proves that TSC$^k$ is a prefix-code.

\textbf{Lemma 8.1} \textit{TSC}^k \textit{is a prefix-code.} \hfill \Box

\textbf{Proof} We assume in negation that $c_i$ is a prefix of $c_j$ where $i < j$ and both $c_i$ and $c_j$ are TSC$^k$ code-words. If $|c_i| = |c_j|$ then the codes cannot be used as prefixes, therefore, $|c_i| < |c_j|$. We examine $c_j$ as a binary string $s_1, \ldots, s_m$. Since $c_i$ is a prefix of $c_j$, we know that the substring $s_{|c_i|−k−1}, s_{|c_i|−k} = (01|10)$. This fact contradicts the $c_j$ construction where the first (01|10) appears in $s_{|c_j|−k−1}, s_{|c_j|−k}$.

In this chapter, we experiment with English text compression that uses TSC$^0$ and TSC$^1$ codes because the implied probabilities of both codes resemble English characters probabilities. Therefore, TSC$^0$ and TSC$^1$ codes fit English text compression. Theoretical analysis is given in section 8.5.

Part III: Accessing XML in a limited memory device
$TSC^1$ is defined by different $trim$ and $trasfrom$ functions. The $trim$ is computed according to the difference in bits between the code-word $c_i$ and the integer $n = i - 1$. The $TSC^1$ code-word $c_i$ is represented by $\lfloor \log_2 i + 3 \rfloor$ bits. The integer $n$ is represented by $\lceil \log_2 i \rceil$ bits.

$$\lfloor \log_2 i \rfloor - \lfloor \log_2 i + 3 \rfloor = \begin{cases} 0 & \text{when } i = 2^k - j, \ 0 \leq j \leq 3 \\ 1 & \text{otherwise.} \end{cases}$$

From this equation we know that if $n = 2^k - j$, $1 \leq j \leq 4$, then the integer bit representation of $n$ and the code-word representation $c_{n+1}$ have the same size. In this case, $n$ has the binary format $1^+(1|0)^2$. Definition 8.7 formalizes the function $trim$ for $TSC^1$ code-words.

**Definition 8.7** Given the string $s \in \{0,1\}^*$, where $s = s_1, \ldots, s_k$ is a binary representation of an integer $n$ such that the most significant bit is $s_1 = 1$. If $s$ is in the format $1^+(0|1)^2$ then the trim function returns $s$. Otherwise, $trim$ trims the most significant bit $s_1 = 1$ and returns $s_2, \ldots, s_k$. The inversion function $trim^{-1}$ pads $s$ with $s_1 = 1$ if $s$ is not in the format $1^+(1|0)^2$.

The $transfrom$ function is modified to handle the additional bit in the code-word as defined in Definition 8.8.

**Definition 8.8** Given a binary string $s \in \{0,1\}^*$, where $s = s_1, \ldots, s_k$. The $transform$ function replaces each binary symbol $s_i$, $1 \leq i < k$, by one or two bits according to the following decision table:

<table>
<thead>
<tr>
<th>$s_i \backslash i$</th>
<th>$&lt; k - 1$</th>
<th>$= k - 1$</th>
<th>$= k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.2 illustrates the construction of $TSC^1$ code. The example shows the code-words $c_1, \ldots, c_9$ that represent the integers $0, \ldots, 8$.

Part III: Accessing XML in a limited memory device
Tagged Suboptimal Code (TSC) 220

<table>
<thead>
<tr>
<th>n</th>
<th>trim(n)</th>
<th>transform(trim(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00010</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>00011</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>00100</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>11010</td>
</tr>
</tbody>
</table>

Table 8.2: An example of TSC\(^1\) universal codes for the numbers 0, \ldots, 8. The trim(n) column describes the binary format of \(n\). The bolded binary digits are the outputs from the trim(n) function. The column transform(trim(n)) shows the encoded code-words.

### 8.3.3 Fast and compact TSC\(^0\) decoding

The complexity of a prefix-code decoding is linear in the size of the compressed data stream and bounded by the number of symbols it has to output. However, this theoretical cost estimation does not take into account the actual CPU time that was used to process and decode every single compressed data bit. There is a difference in the decoding speed if every single bit has to be processed individually, compared to bytes processing (8 bits at once) or even larger machine words.

A method for byte processing of prefix-codes is described in [132, 131]. This method requires main memory storage space for the input-words transition table and for the output symbols table. For a given set of \(m\) encoded source symbols in the alphabet and an input-word size of \(k\) bits, the input-words transition table is of size \(O(m \cdot 2^k)\) and the output symbols table has a worst case space of \(O(m \cdot 2^k \cdot k)\) since at most \(k\) symbols may be decoded by a bit-sequence of length \(k\) of prefix-codes. For \(k = 8\) and \(m = 256\), the size of the output symbols table is 64 KB in the worse case. Limited resources devices do not have 64 KB of memory. We propose a method for byte processing of TSC\(^0\) code-words. The memory bound of the suggested method is \(O(2^{k+2} \cdot k)\). Therefore, for \(k = 8\), the size of the memory is 1 KB, which is \(\frac{4}{m} = \frac{1}{64}\) from the fast prefix-code approach in [132]. For texts with a larger size of \(m\) encoded source symbols,
Tagged Suboptimal Code (TSC)

for example Japanese text, the compactness of the tables is even more evident.

We construct, like [132], a transition input-subwords table and an output sub-symbols table. We use the fact that TSC can decode a code-subword into an output sub-symbol, which is an integer part, without processing the complete code-word in order to reduce the sizes of the tables. Fast TSC0 decoding process uses the tables to translate the encoded data according to the transform−1 function in definition 8.6.

An entry in the input-subwords table represents an input-subword. An input-word is also an input-subword. Two encoded bits in a TSC0 code-word contribute one decoded bit of an output symbol, therefore, there are \( \sum_{0 \leq i \leq \frac{k}{2}} 2^{2i} \) input-subwords. An entry in the input-subwords table stores two values: 1. A reference to the output sub-symbol that is decoded from the current code-subword; 2. A reference to the next input-subword after removing the current code-subword from the current input-word. The number of input-subwords is bounded by \( \sum_{0 \leq i \leq \frac{k}{2}} 2^{2i} < \sum_{0 \leq i \leq k} 2^i < 2^{k+1} \). Therefore, a reference to the next input-subword can be represented by \( k + 1 \) bytes. We now calculate the size of the reference to an output sub-symbol entry. There are four code-subwords of size of two. Each addition of two bytes to the code-subwords multiplies the number code-subwords by two. The total number of code-subwords is \( \sum_{2 \leq i \leq \frac{k}{2} + 1} 2^i \). From \( \sum_{2 \leq i \leq \frac{k}{2} + 1} 2^i < \sum_{0 \leq i \leq \frac{k}{2} + 1} 2^{2i} < 2^{k+2} \) we get that a reference to a code-subword is represented by \( \frac{k}{2} + 2 \) bits. An entry in the input-subwords table needs \( k + 1 \) bits for the reference of the next input-subword plus \( \frac{k}{2} + 2 \) bits for the reference to the current output sub-symbol. All in all, an entry needs \( \frac{3k}{2} + 3 \) bits. If \( k > 6 \) then \( \frac{3k}{2} + 3 < 2k \). Therefore, the size of the input-subwords table is bounded by \( O(2^{k+1} \cdot 2k) = O(2^{k+2} \cdot k) \).

The output sub-symbols table contains an entry for each code-subword. A code-word is also a code-subword. Each entry defines an output sub-symbol i.e an integer part that is decoded from a code-subword. An entry contains the integer part, the size of the integer part in bits and whether or not the integer part completes the integer decoding. The number of entries in the output sub-symbols table is bounded by the number of code-subwords which is \( O(2^{\frac{k}{2}+2}) \). Each entry stores the integer part in \( \frac{k}{2} \) bits, the size of the integer part in \( \log(\frac{k}{2}) \) bits and one bit for the complete boolean flag. If \( k > 2 \) then \( \frac{k}{2} + \log(\frac{k}{2}) + 1 < k \). Therefore, the size of the output-subwords table is bounded by \( O(2^{\frac{k}{2}+2} \cdot k) \). Table 8.2 describes the input-subwords table and the output sub-symbols table for the nibble input-words (\( k = 4 \)).

Part [III]: Accessing XML in a limited memory device
Tagged Suboptimal Code (TSC) 222

<table>
<thead>
<tr>
<th>input-subword</th>
<th>id of</th>
<th>id of</th>
<th>id of output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binary)</td>
<td>input-subword</td>
<td>input-subword</td>
<td>sub-symbol</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—00</td>
<td>16</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>—01</td>
<td>17</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>——</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) The input-subwords table.

<table>
<thead>
<tr>
<th>code-subword</th>
<th>id of output</th>
<th>integer part</th>
<th>size of complete integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binary)</td>
<td>sub-symbol</td>
<td>(binary)</td>
<td>integer part (y=yes, n=no)</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>0001</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0010</td>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>4</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) The output sub-symbols table

Figure 8.2: Description of the tables that are constructed for $TSC^0$ decoding in nibble input-words. We omitted several entries from the tables to simplify the presentation. The values of the entries are integers unless described otherwise in brackets. A bolded columns name denotes the column name in Algorithm 25. A column name, which is not bolded, is added for a better understanding of Algorithm 25 and it is not part of the implementation.

Fast implementation of $transform^{-1}$ is described by Algorithm 25. The decode operation $decode(input) \triangleq trim^{-1}(transform^{-1}(input))$ decodes the current code-word in an input stream that is composed of input-words of size $k$. $transform^{-1}$ iterates over the code-subwords of the current code-word until the whole code-word in the stream is decoded. In each iteration, it decodes the current code-subword to an integer part and adds it to the integer that was decoded in previous iterations. The function $next_{byte}$
returns the next input-subword entry or the next input-word entry when the next input-subword is empty.

**Algorithm 25: TSC\(^0\) byte-oriented transform**

\[\text{transform}^{-1}(\text{stream} : \text{input})\]

**Output:** int : number, int : bitscount

**Data:** input-subwords-table : \(\text{in}._\text{table}, \text{byte} : \text{inbyte}\),

\[\text{output-sub-symbols-table} : \text{out}._\text{table}, \text{byte} : \text{outbyte}\]

\[
\begin{align*}
\text{begin} & \quad \text{bitscount} \leftarrow 0; \\
\text{repeat} & \quad \text{inbyte} \leftarrow \text{next_byte}(\text{input}); \\
& \quad \text{outbyte} \leftarrow \text{in}_\text{table}[\text{inbyte}].\text{output} ; \\
& \quad \text{number} \leftarrow \text{number} \lor \text{out}_\text{table}[\text{outbyte}].\text{integer} \ll \text{bitscount} ; \\
& \quad \text{bitscount} \leftarrow \text{bitscount} + \text{out}_\text{table}[\text{outbyte}].\text{size} ; \\
\text{until} & \quad \text{out}_\text{table}[\text{outbyte}].\text{complete} ;
\end{align*}
\]

**end**

\[
\text{int} \quad \text{next}_\text{byte}(\text{stream} : \text{input}) \\
\begin{align*}
\text{begin} & \quad \text{if} \quad \text{in}_\text{table}[\text{inbyte}].\text{next} = \text{NULL} \text{ then} \\
& \quad \_ \quad \text{return} \quad \text{input.read}(); \\
& \quad \text{else} \\
& \quad \_ \quad \text{return} \quad \text{in}_\text{table}[\text{inbyte}].\text{next} ;
\end{align*}
\]

**Example 8.1** Figure 8.3(a) illustrates an input stream with three nibbles. The input stream contains three code-words ‘000001’, ‘0001’ and ‘01’. The separation to nibbles creates five input-subwords that correspond to five code-subwords ‘0000’, ‘01’, ‘00’, ‘01’ and ‘01’. Each input-subword table transition decodes a single code-subword. There are extra three transitions to a NULL state that mark the end of a nibble. Overall, there are eight states transitions. Figure 8.3(b) illustrates the application of the input-subwords table, which is described in Fig. 8.2(a), to the input stream in Fig. 8.3(a). 

Part III: Accessing XML in a limited memory device
8.3.3.1 Fast decoding by $TSC^1$

$TSC^1$ input-word processing uses the same tables types as $TSC^0$. $TSC^1$ input-word processing has two additional operations: 1. Bit pair alignment; 2. Suffix trimming. The $TSC^1$ code-words bit pairs, i.e. $(11|00|10|01)$, can be split between two sequential bytes of the input stream. $TSC^0$ code-word bit pairs are always in the same byte because the byte size (8) is divided by the pair size (2) without a residual whereas the $TSC^1$ code-words have an extra bit suffix. Bit pair alignment overcomes this complexity by “remembering” the code-word bit pair prefix that ended the previous byte. The $TSC^1$ input-subwords table, which is constructed for input-words with $k$ bits, contains input-subwords with $i$ bits where $0 \leq i \leq k+1$. The additional input-subwords with $k+1$ bits are used to align bit pairs that are split between two sequential bytes. The $next\_byte$ function in Algorithm 25 is extended to handle single bit input-subwords. When an
Tagged Suboptimal Code (TSC)

input-subword of a single bit is encountered, then the single bit is concatenated with the $k$ bits of the next input-word in the stream. The extended input-subwords table handles the concatenated input-subword that has $k + 1$ bits. When the input-subword table is carefully constructing then the concatenation operation can be replaced by a fixed offsetting of the input-word. Note that the size of the input-subword table for the $TSC^1$ table is $O(2^{k+3} \cdot k)$ due to the support in the $k + 1$ input-subwords.

The suffix of $TSC^1$ can be split between two sequential bytes of the input stream. Suffix trimming operation handles this case. When a single bit suffix of the $TSC^1$ code-word is processed in a separated input-word, then the $next_{\text{byte}}$ function is extended to trim this bit from the input-word and directly appends it to the decoded integer. Due to the $TSC^k$ encoding scheme, a $k$ bits suffix of a $TSC^k$ code-word is an integer part that can be outputted without translation. When the input-subword table is carefully constructing, then the trimming operation can be replaced by a fixed offsetting of the input-word.

Example 8.2 Demonstration of the bit pair alignment operation. Figure 8.4(a) illustrates an input stream that contains two $TSC^1$ code-words ‘010’ and ‘00100’. The input stream has two nibbles. Both nibble input-words ‘0100’ are the same. But different byte-pair prefixes (NULL for the first nibble and ‘0’ for the second) cause different transitions to be derived. Figure 8.4(b) illustrates the application of the input-subwords table, which is described in table 8.3, to the input stream in Fig. 8.4(a). This example decodes the integers 0 and 6 from table 8.2.

Part III: Accessing XML in a limited memory device
Tagged Suboptimal Code (TSC)

<table>
<thead>
<tr>
<th>input-subword (binary)</th>
<th>id of input-subword</th>
<th>id of next input-subword</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>4</td>
<td>62</td>
</tr>
<tr>
<td>–0100</td>
<td>36</td>
<td>60</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>——0</td>
<td>60</td>
<td>–</td>
</tr>
<tr>
<td>——1</td>
<td>61</td>
<td>–</td>
</tr>
<tr>
<td>———</td>
<td>62</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 8.3: Description of a partial input-subwords table that is constructed for TSC\textsuperscript{1} decoding of nibble input-words. The values of the entries are integers unless described otherwise in brackets. A bolded column name denotes the column name in Algorithm 25. A column name, which is not bolded, is added for a better understanding of Algorithm 25 and it is not part of the table implementation.

![Diagram](image)

Figure 8.4: Decoding integers 0 and 6 in table 8.2 (a) Description of the input-words. The continues boundaries denote nibbles input-words. There are two nibbles in the input. The dashed boundaries denote the code-words \(c_7\) and \(c_1\). (b) Description of the operation of the input-subwords table. The input-subwords table is operated in the extended \(\text{next}_\text{byte}\) function that was described in section 8.3.3.1. A circle denotes the current input-subword entry in the transition table. The label of the circle denotes the id of the input-subword. A double circle denotes an input-subword that completes a TSC\textsuperscript{1} code-word. A bolded circle denotes an empty or a single byte of the input-subword.

### 8.3.4 Text compression by TSC universal codes

The \(TSC\) text compression scheme, which was presented in [27], is static. When \(TSC\) is treated as a universal code, the static compression scheme is straightforward. We
store the characters in the array \( \text{char} : \text{char}_1, \ldots, \text{char}_n \) where \( \text{char}_i \) is the \( i \)-th most common character in the text. The static decoding of a symbol is done by the operation \( \text{char}_{\text{decode}}(\text{input}) \). Figure 8.5 illustrates the \( TSC^0 \) static compression.

![Figure 8.5: Illustration of the \( TSC^0 \) codec context. Row 0 shows the \( TSC^0 \) code-words. Row 1 shows the \( \text{char} \) array. If the decoded code-word is \( \text{table}[0][j] \) then the symbol \( \text{char}_j = \text{table}[1][j] \) is the decoded symbol. The binary string “10,0001,0001,0001” is decoded to become the string “bcce”.

An adaptive text compression scheme, which uses universal codes, is also straightforward. The algorithm stores the frequencies of the \( \text{freq} \) array. The \( \text{freq} \) array stores each frequency in \( K \) bits. After decoding each character, the compression uses an \( \text{adaptive} \) function that increases the decoded character frequency by one and sorts the \( \text{char} \) array according to the characters frequencies where the most common character is in index 1. The adaptive text decoding scheme returns \( \text{char}_i \) where \( i = \text{adaptive}(\text{decode}(\text{input})) \).

This is described in Algorithm 26.

Algorithm 26 describes the \( \text{adaptive} \) function. This function does not sort all the characters. It only updates the decoded character. Given a decoded character in index \( i \), the algorithm replaces the decoded character \( \text{char}_i \) with the character \( \text{char}_{i'} \) where \( \text{freq}_i = \text{freq}_{i'} \) and \( i' \) is minimal. Note that \( i \) can be equal to \( i' \). A single replacement is sufficient to sort characters according to their frequencies because after a frequency update \( \text{freq}_i \) does not change, \( \text{freq}_i < \text{freq}_{i'} \) and \( \text{freq}_{i'} \leq \text{freq}_{i'-1} \). The \( \text{index}\_\text{finder} \) function finds \( i' \). In the worse case, it may iterate on the whole array to find \( i' \). Therefore, its complexity is \( O(|\Sigma|) \).

After the symbol array is updated, the \( \text{overflow}\_\text{check} \) function is called. The \( \text{overflow}\_\text{check} \) checks if the most common frequency is \( 2^K - 1 \) which is the maximum frequency that \( K \) bits enable. In this case, it divides all the frequencies by \( 2^M \) where \( M \leq K \). The division is done using the shift right bit operation ‘\( \gg \)’. After the division, the \( M \) most significant bits in each frequency are zero and each frequency can be increased by at least \( 2^M \) before another division will occur. This di-
vision takes $O(|\Sigma|)$ time. If we choose $M \approx \log_2|\Sigma|$ then it takes $O(1)$ amortized. $M = K = 1$ is a special case in which the frequency is stored in a single bit. In this case, the compression scheme is similar to combining move-to-front transform [133] with TSC universal codes. After the decoding of every symbol, the overflow_check resets the freq to zeros. The index_finder in each iteration returns the index 0 because $freq_1 = freq_2 = \ldots = freq_n = 0$.

Algorithm 26: The adaptive algorithm

Data: char[], freq[], M

int adaptive (int : i)

begin
  $i' = \text{index\_finder}(i)$;
  char$_{temp} = \text{char}_i$; freq$_{temp} = \text{freq}_i$;
  char$_i = \text{char}_i$; freq$_i = \text{freq}_i + 1$;
  index_finder();
  return $i'$;
end

int index_finder (int : i)

begin
  $i' = i$;
  while $\text{freq}_{i'-1} = \text{freq}_{i'}$ do
    $i' \leftarrow i' - 1$
  return $i'$;
end

overflow_check()

begin
  if $\text{freq}_0 = 2^K - 1$ then
    forall $j$ do
      forall $j$ do $\text{freq}_j \leftarrow \text{freq}_j \gg M$;
  end
end

Figure 8.6 is an example that illustrates the adaptive compression of TSC$^0$.
Figure 8.6: The table illustrates how the context changes by $TSC^0$ when the string “bccc” is decoded. Row 0 shows the code-words. Column 0 shows the decoded symbols. Row $i$ shows the context after decoding the characters in $table[1][0], \ldots, table[i][0]$. The shadowed upper row shows the $char^i$ array and the bottom row shows the $freq^i$ array. If the character in $table[i][0] = char^i_j$ in each row $i$ then $table[0][j + 1]$ is the decoded code-word. The encoding of the string “bccc” becomes the binary string “10,0001,10,01”.

### 8.4 Compressed string matching in TSC

The $TSC^0$ string matching technique, which is described in [27], utilizes the fact that $TSC^0$ code-word is ‘tagged’ in the sense that its boundaries are identified at any position in the string.

Unfortunately, when $k$ is an odd number, the $TSC^k$ code is not ‘tagged’. $TSC^k$ code-words boundaries cannot be identified at any position of the string. Therefore, $TSC^0$ string matching technique, which is described in [27], cannot be applied to $TSC^k$ code when $k$ is an odd number.

Instead, we use the compact tables, which were constructed in section 8.3.3 to optimize the string matching process. When $k$ is an even number, $TSC^k$ uses the tables that were constructed for $TSC^0$ decoding. When $k$ is an odd number, $TSC^k$ uses the tables that were constructed for $TSC^1$ decoding. These constructed tables parse code-subwords. We suggest a string matching process that 1. Translates the encoded binary data into a string of code-subwords symbols; 2. Translates the string pattern into a
code-subwords symbols string pattern; 3. Matches the pattern in the data in the code-subwords symbols domain.

8.4.1 Encoded data translation

The output sub-symbols table, which was constructed in section 8.3.3 translates code-subwords into integer parts. In order to translate the code-subwords into the code-subwords symbols domain, we assign to each entry in the output sub-symbols table a symbol identifier. This attribute uniquely identifies the current parsed code-subword in the encoded string.

Example 8.3 Figure 8.7 illustrates a data translation of a binary string, which encodes the English text “werewere”, into the $TSC^0$ code-subwords domain. Figure 8.7(a) describes the symbols that are assigned to the code-subwords $00,01,10,11,0001$ and $0010$ in the output sub-symbols table. Each code-subword symbol is denoted by a Greek letter. Figure 8.7(b) describes a char array of a static $TSC^0$ English text compression. The char array translates the letters $w, e, r, x, y, z$ into $TSC^0$ code-words. Figure 8.7(c) describes the data translation in three rows. Row 1 describes the uncompressed string. Row 2 describes the binary compressed string. The continuous lines denote composition into bytes. The dashed lines denote composition into code-words. Row 3 describes the translated code-subwords symbols string. 

Part III: Accessing XML in a limited memory device
8.4.2 Pattern string translation

In order to translate a pattern string into a code-subword symbols domain, the following take place. 1. It is encoded. 2. Each code-word is decomposed into code-subwords according to the way by which the code-words were divided into bytes. For example, we examine the shortest pattern in example 8.3. We see that ‘e’ is mapped into the code-word 01. Its size is two bits which is also the shortest code-subword size. The character ‘e’ is chosen as a pattern. It is translated into its code-subword symbol Ξ. There are five code-subword symbols Ξ in the translated data in Fig. 8.7(c). But there are only four ‘e’ characters in the uncompressed data. The translated data contains an extra Ξ symbol because the code-word ‘0001’, which is the code-word of the first character ‘r’ in the

Figure 8.7: Illustration of data translation into code-subwords symbols domain
Tagged Suboptimal Code (TSC)

data, is divided into two code-subwords between the first byte 00 and the second byte 01. Therefore, the second byte starts with the code-subword 01 that is translated into the symbol Ξ.

This example demonstrates why the suggested translation in section 8.4.1 is insufficient for pattern matching. The synchronization problem, which was introduced in section 8.2.2, is not solved by the translation that was introduced in section 8.4.1. In order to match a pattern in the data in the code-subwords symbols domain, we need to refine the data translation. We have to differentiate between code-subwords that start a code-word and code-subwords that occur in the middle of a code-word. The initial code-subword of a matched pattern always starts a code-word. Figure 8.8 illustrates the refined data translation of the data in example 8.3. The code-subwords, which occur in the middle of a code-word, are denoted by the same Greek letter as the code-subwords that start a code-word. But code-subword symbols, which occur in the middle of a code-word, are annotated by an extra prime. To implement the refined translation, we add two code-subwords symbols identifiers to every output sub-symbol entry in the table.

![Figure 8.8: Illustration of a refined data translation into a code-subwords domain](image)

When the refined translation is used, the ‘e’ characters are translated into the symbol Ξ, where the extra 01 code-subword symbol is translated into Ξ’. When we match the pattern Ξ in the refined data, we match only the four characters ‘e’ that exist in the original text. In this way, the exact pattern is matched. Up till now, we match the shortest pattern available. When a longer pattern is matched, its string code-words division into code-subwords is determined by the bits offset of the encoded pattern string in the encoded data. There are eight possible offsets in a byte. Therefore, there are up to eight different code-subwords symbols patterns to match. In TSC₀, the offsets positions are even. Therefore, there are only four different code-subword symbols patterns that we

Part III: Accessing XML in a limited memory device
need to match. Figure 8.9 illustrates the translation of the pattern string ‘were’ into code-
subword symbols strings. The first column (code-words string) describes the encoded
pattern string with different possible start offsets. The second column (code-subwords
string) describes the translation of the pattern into a code-subwords symbols string. Row
1 describes the pattern string encoding with offset start of 0. Row 2 describes the pattern
string encoding with offset start of 2. Row 3 describes the pattern string encoding with
offset start of 4 and row 4 describes the pattern string encoding with offset start of 6.
The translation outputs three code-subwords patterns. Offsets 2 and 4 are translated into
the same string. From the data in Fig. 8.8, we see a concatenation of two code-subwords
symbol patterns: The pattern in offset 0 followed by the pattern in offset 2. Therefore, the
pattern matching of ‘were’ in the data ‘werewere’ in the code-subword symbols domain
returns the same results as in the uncompressed domain.

<table>
<thead>
<tr>
<th>#</th>
<th>Code-words string</th>
<th>Code-subwords string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0 0 1 0 0 1 0 0 1 0 1</td>
<td>Ω Ξ φ Ξ' Ξ</td>
</tr>
<tr>
<td>2.</td>
<td>0 0 1 0 0 1 0 0 0 1 0 1</td>
<td>Ω Ξ Δ Ξ</td>
</tr>
<tr>
<td>3.</td>
<td>0 0 1 0 0 1 0 0 0 1 0 1</td>
<td>Ω Ξ Δ Ξ</td>
</tr>
<tr>
<td>4.</td>
<td>0 0 1 0 0 1 0 0 0 1 0 1</td>
<td>φ Γ' Ξ Δ Ξ</td>
</tr>
</tbody>
</table>

Figure 8.9: Illustration of the translation of the pattern string ‘were’ into code-subword
symbols patterns according to the translation tables in Fig. 8.7

8.4.3 Pattern matching

After the text and the pattern were translated into code-subwords symbols domain, pat-
tern matching can be achieved by any known algorithm for string matching that is
able of: 1. Forward string traversal. 2. Multiple strings matching support. Forward
traversal is needed because backward traversal is inefficient on un-tagged variable-
length code-words. We have to use an algorithm that matches multiple strings because
each pattern is translated into multiple patterns in the code-subword symbols domain. In

Part III: Accessing XML in a limited memory device
this chapter, we implement the Aho-Corasick (AC) string matching algorithm [14]. AC was chosen because the construction of the AC machine is compact. Therefore, it fits limited resources devices. The AC machine construction algorithm stays unchanged. The AC search algorithm is described in Algorithm 27. It receives the encoded input stream and the current state of the AC-machine. The AC machine is constructed from the code-subwords symbols patterns that were constructed in section 8.4.2. Algorithm 27 traverses the data as code-subwords symbols. The function \( \text{next\_symbol} \) preforms the code-subwords symbols traversal. It resembles the \( \text{next\_byte} \) function in Algorithm 25. When an input-word traversal ends, it checks if it ends a complete code-word. If it does not end where a code-word ends then the next code-subword symbol is a middle code-subword (its attribute in the output sub-symbols table is \( \text{symbol}' \)). In all other cases, the next code-subword symbol is a start code-subword (its attribute in the output sub-symbols table is \( \text{symbol} \)).

At each step, the algorithm finds the AC machine next state that accepts the code-subword symbol. If no transition exists then the failed transition is used until the root state is reached. The function returns the id of the AC state that was found or \(-1\) if no pattern was found.

Part III: Accessing XML in a limited memory device

```c
int next_symbol(stream : input)

Data: input-subwords-table : in_table, byte : inbyte,
      output-sub-symbols-table : out_table

begin
  Let out_entry be out_table[in_table[inbyte].output];
  if table[inbyte].next = NULL then
    if out_entry.complete then
      inbyte ← input.read();
      return out_entry.symbol;
    else
      inbyte ← input.read();
      return out_entry.symbol';
  else
    inbyte ← table[curbyte].next;
    return out_entry.symbol;
end

int ACsearch(aho_corasick_state_t : state, stream : input)

begin
  while input exists do
    let code ← next_symbol(input);
    while state.data(code) = FAIL do
      state ← state.fail;
      state ← state.data(code);
    if state.output > 0 then
      return state.id ;
  return -1
end
```

8.5 Theoretical analysis of $TSC^k$ compression for $k=0,1$

$TSC$ code is universal. Definition 8.4 states two conditions for a prefix-code to be universal. Section 8.3.1 described the first condition that maps each code-word to a
number. The second condition states that the code-words length are bounded by the optimal-code-word lengths. Theorem 8.1 formalizes the second condition.

**Lemma 8.2** Given a binary tree $T$ with at least $n$ nodes. Then, $\text{height}(T) \geq \log_2 n$. □

**Proof** The minimal height is achieved when the tree is fully balanced. In this case, $\text{height}(T) = \log_2 n$. ■

**Lemma 8.3** Given probabilities $p_1, \ldots, p_n$ for an alphabet $a_1, \ldots, a_n$. Then, the Huffman code-word length is $|c_i| \geq \log_2 (i + 1)$ for every $i$. □

**Proof** $|c_i| = \text{depth}(T, v_{c_i})$ where $v_{c_i}$ is the leaf that represents the code-word $c_i$ in the Huffman tree $T$. The Huffman subtree $T'$, which contains a node $v$ such that $\text{depth}(T', v) \geq \text{depth}(T', v_{c_i})$, includes the leaves $v_{c_1}, \ldots, v_{c_i}$. Otherwise, there exists $j \leq i$ such that $|c_j| > |c_i|$. This contradicts the Huffman construction and its optimality because by switching between $c_j$ and $c_i$ the code-words produce a better code. $\text{root}(T) = \text{root}(T')$. Therefore, $T'$ contains at least $i + 1$ nodes. $\text{depth}(T, v_{c_i}) = \text{height}(T')$. Therefore, we know from Lemma 8.2 that $|c_i| = \text{depth}(T, v_{c_i}) = \text{height}(T') \geq \log_2 i + 1$.

**Theorem 8.1** Given probabilities $p_1, \ldots, p_n$ for an alphabet $a_1, \ldots, a_n$. We construct the Huffman optimal-code $c_1^o, \ldots, c_n^o$ and the TSC₀ suboptimal code $c_1, \ldots, c_n$. Then, $\frac{|c_i|}{|c_i^o|} \leq 2$ for every $i$. □

**Proof** From the TSC₀ construction in section 8.3.1, we know that $|c_i| = 2 \cdot \lfloor \log_2 (i + 1) \rfloor$. From Lemma 8.3 we know that $|c_i^o| \geq \log_2 (i + 1) \geq \lfloor \log_2 (i + 1) \rfloor$. Therefore, $\frac{|c_i|}{|c_i^o|} \leq 2$. ■

The ratio 2 in Theorem 8.1 is general. A better factor can be shown for specific probabilities. The Appendix in section 8.8 shows the existence of better factors for exponential and for Fibonacci probabilities functions. The universality of the TSC code becomes clearer when the TSC code is compared to Elias gamma universal code. The size of the Elias gamma code for integer $i$ is $2 \cdot \lfloor \log_2 i \rfloor + 1$. Theorem 8.2 defines the relation between the Elias gamma and the TSC code sizes.
Theorem 8.2 Given probabilities \( p_1, \ldots, p_n \) for integers 1, \ldots, \( n \). The Elias gamma code \( c^0_1, \ldots, c^0_n \) and the TSC\(^0\) code \( c^1_1, \ldots, c^1_n \) are constructed. Then, \( \text{abs}(|c^1_i| - |c^0_i|) = 1 \) for every \( i \).

**Proof** \( \lfloor \log_2 i + 1 \rfloor - \lfloor \log_2 i \rfloor = \begin{cases} 
1 & \text{when } i + 1 = 2^k \\
0 & \text{otherwise.} 
\end{cases} \)

Therefore, \( |c^1_i| - |c^0_i| = 2 \cdot (\lfloor \log_2 i + 1 \rfloor - \lfloor \log_2 i \rfloor) - 1 = \begin{cases} 
1 & \text{when } i + 1 = 2^k \\
-1 & \text{otherwise.} 
\end{cases} \)

We can see from Theorem 8.2 that \( |c^1_i| = |c^0_{i+1}| - 1 \). When the probability distribution is uniform, then, the average code size of TSC\(^0\) is smaller by almost one bit from the average code size of Elias gamma as shown by the following equation:

\[ \frac{1}{n} \sum_{i=1}^{n} |c^1_i| = \frac{1}{n} \sum_{i=2}^{n+1} (|c^0_i| - 1) = \frac{1}{n} (\sum_{i=1}^{n} c^0_i - n + 2 \cdot \log_2 (n + 1) + 1 - 1) = \frac{1}{n} \sum_{i=1}^{n} |c^0_i| - \frac{n - 2 \cdot \log_2 (n + 1)}{n}. \]

The universality of the TSC\(^1\) code becomes clearer when the TSC\(^1\) code is compared to the universality of the Elias gamma code. Theorem 8.3 defines the relation between the Elias gamma and the TSC\(^1\) code sizes.

Theorem 8.3 Given probabilities \( p_1, \ldots, p_n \) for integers 1, \ldots, \( n \). The Elias gamma code \( c^0_1, \ldots, c^0_n \) and the TSC\(^1\) code \( c^1_1, \ldots, c^1_n \) are constructed. Then, \( \text{abs}(|c^1_i| - |c^0_i|) = 2 \) for every \( i \).

**Proof** The size of TSC\(^1\) code-word of integer \( i \) is \( 2 \cdot \lfloor \log_2 i + 3 \rfloor - 1 \).

\[ \lfloor \log_2 i + 3 \rfloor - \lfloor \log_2 i \rfloor = \begin{cases} 
2 & \text{when } i = 1 \\
1 & \text{when } i + j = 2^k \text{ where } 1 \leq j \leq 3 \\
0 & \text{otherwise.} 
\end{cases} \]

Therefore,

\[ |c^1_i| - |c^0_i| = 2 \cdot (\lfloor \log_2 i + 3 \rfloor - \lfloor \log_2 i \rfloor) - 2 = \begin{cases} 
-2 & \text{when } i = 1 \\
0 & \text{when } i + j = 2^k \text{ where } 1 \leq j \leq 3 \\
2 & \text{otherwise.} 
\end{cases} \]

We can see from Theorem 8.3 that the first TSC\(^1\) code-word is longer than the first Elias gamma code-word by two bytes but the other code-words are shorter in two bytes or equal to the gamma code-word.

The difference in the code-word sizes originated from the different implied probabilities of the codes. The implied probability of a gamma code \( c^0_i \) is \( p^0_i = \frac{1}{2^2 \cdot \log_2 (i + 1)}. \) The

Part III: Accessing XML in a limited memory device
implied probability of a TSC code $c_t^i$ is $p_t^i = \frac{1}{2^{F_i^2+1}}$ and the implied probability of a TSC$^1$ code $c_t^i$ is $p_t^i = \frac{1}{2^{F_i^2+3}}$. The Elias gamma code is more suitable for distorted probabilities than TSC$^0$. TSC$^0$ is more suitable for distorted probabilities than TSC$^1$. In general, TSC$^k$ is more suitable for distorted probabilities than TSC$^{k+1}$. The English letters probability is not highly distorted. Therefore, TSC$^0$ and TSC$^1$ codes implied frequencies are more similar to English letters frequencies than the Elias gamma code implied frequencies. This similarity enables us to utilize TSC codes for English text compression. Table 8.4 examines the English letters frequencies in comparison to TSC and Elias gamma codes implied frequencies.

<table>
<thead>
<tr>
<th>Elias</th>
<th>TSC$^0$</th>
<th>TSC$^1$</th>
<th>English</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>25</td>
<td>12.5</td>
<td>12.7</td>
<td>e</td>
</tr>
<tr>
<td>12.5</td>
<td>25</td>
<td>12.5</td>
<td>9.1</td>
<td>t</td>
</tr>
<tr>
<td>12.5</td>
<td>6.1</td>
<td>12.5</td>
<td>8.2</td>
<td>a</td>
</tr>
<tr>
<td>3.1</td>
<td>6.1</td>
<td>12.5</td>
<td>7.5</td>
<td>o</td>
</tr>
<tr>
<td>3.1</td>
<td>6.1</td>
<td>3.1</td>
<td>7.0</td>
<td>i</td>
</tr>
<tr>
<td>3.1</td>
<td>6.1</td>
<td>3.1</td>
<td>6.7</td>
<td>n</td>
</tr>
<tr>
<td>3.1</td>
<td>1.6</td>
<td>3.1</td>
<td>6.3</td>
<td>s</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>3.1</td>
<td>6.1</td>
<td>h</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>3.1</td>
<td>6.0</td>
<td>r</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>3.1</td>
<td>4.3</td>
<td>d</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>3.1</td>
<td>4.0</td>
<td>l</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>3.1</td>
<td>2.8</td>
<td>c</td>
</tr>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>0.8</td>
<td>2.8</td>
<td>u</td>
</tr>
</tbody>
</table>

Table 8.4: Comparison between English letters frequencies that use TSC and Elias gamma codes implied frequencies

8.6 Experimental results

The Basic Compression Library (BCL) [67] was chosen as a benchmark. BCL is a library of well known compression algorithms implemented in portable ANSI C code. We extended the BCL library for our testing purposes. We added the BPE compression scheme, the TSC compression variants and the string matching over TSC$^0$ encoded data. The experiment was performed on Intel dual core processor with 1.83GHz CPU
We tested the TSC compression scheme on the well known Calgary corpus [26] and on the Canterbury corpus [21].

Table 8.5 summarizes the compression ratios (CR) for different methods. We compare among the static $TSC^0$ and $TSC^1$, the adaptive $TSC^0 (TSC^0_A)$ and $TSC^1 (TSC^1_A)$ compression methods. We compared these methods against the static Huffman encoding [78] and Byte Pair Encoding (BE) [65]. The BE was adjusted to run on a limited memory. Its bucket size was reduced to 1 KB and its symbols hash table size was reduced to 3 KB. In this way, its memory utilization matched the memory utilization of the TSC compression schemes. Table 8.5 shows the CR from different compression schemes that were applied to ASCII files from the Calgary corpus that were created by humans. The table shows the average achieved CR for these files and the average achieved CR for all the ASCII files from the Calgary corpus. It also shows the average achieved CR on the ASCII files from the Canterbury corpus. The experiments on both data sets produce similar results. The CR achieved by Huffman outperforms by 3% – 4% the CR achieved. $TSC^1$. The CR achieved by $TSC^1$ outperforms by 2% – 3% the CR achieved by $TSC^0$. The CR achieved by the static $TSC$ outperforms by another 2% – 3% the adaptive CR achieved by $TSC$. The CR from the adaptive $TSC$ is equal or better than the CR that the $BPE$ produced.
Table 8.5: Summary of the compression ratios in percentages from different encoding methods. \( TSC^0 \) and \( TSC^1 \) are static. The adaptive \( TSC^0 \) and \( TSC^1 \) are denoted by \( TSC^0_A \) and \( TSC^1_A \), respectively. Huffman encoding and Byte Pair Encoding are denoted by \( Huffman \) and \( BPE \), respectively. These methods were applied to ASCII files from the Calgary corpus that were created by humans. The average achieved CR on these files is denoted by \( average \). The achieved average CR on all the ASCII files from the Calgary corpus is denoted by \( average+ \). These methods were also applied to the ASCII files from the Canterbury corpus and the results are denoted by \( Canterbury \).

Table 8.6 shows the encoding and decoding time and the string matching time in milliseconds measured by different methods. We compare among the encoding/decoding time using Huffman, BE and static \( TSC^0 \). We also compare among the encoding/decoding time for text such as Gamma Rice codes [55]. The CR for the Gamma Rice codes is close to 100%. Therefore, it is not shown in table 8.5. This experiment was added to compare between the \( TSC \) universal code processing and the gamma codes processing which is another universal code. The table also contains the Aho-Corasick string matching time in the \( TSC^0 \) compressed domain (\( S_{TSC^0} \)) and the string matching time in the uncompressed text denoted (\( S_{text} \)). The experiments show that the encoding and decoding times of the \( TSC^0 \) is three times faster than Huffman encoding and decoding, respectively. The encoding and decoding times of the \( TSC^0 \) is five times faster than the Gamma-Rice encoding and decoding times, respectively. The \( BPE \) decoding time is also similar to the \( TSC^0 \) encoding and decoding times. The \( BPE \) encoding time is
13 times slower than the $TSC^0$ encoding time. The performance of the string matching was also tested. The matched strings were taken from the AC original paper [14]: ‘his’, ‘hers’ and ‘she’. The $TSC^0$ compressed string matching time is 25% faster than the $TSC^0$ decoding time and even slightly faster than the $TSC^0$ encoding time. The search time in the uncompressed text is about two times faster than the search in the compressed domain. The search times, which were measured, did not include I/O operations. On platforms where the I/O operation is expansive, the search in the compressed domain may be more cost-effective.

<table>
<thead>
<tr>
<th>FILE</th>
<th>$E_{Huffman}$</th>
<th>$D_{Huffman}$</th>
<th>$E_{BPE}$</th>
<th>$D_{BPE}$</th>
<th>$E_{Elias}$</th>
<th>$D_{Elias}$</th>
<th>$E_{TSC^0}$</th>
<th>$D_{TSC^0}$</th>
<th>$S_{TSC^0}$</th>
<th>$S_{text}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>5.26</td>
<td>8.4</td>
<td>32.87</td>
<td>1.78</td>
<td>11.0</td>
<td>10.2</td>
<td>2.06</td>
<td>2.4</td>
<td>1.9</td>
<td>0.7</td>
</tr>
<tr>
<td>book1</td>
<td>32.26</td>
<td>47.15</td>
<td>201.44</td>
<td>11.72</td>
<td>72.8</td>
<td>66.9</td>
<td>13</td>
<td>15.4</td>
<td>11.5</td>
<td>5.1</td>
</tr>
<tr>
<td>book2</td>
<td>25.77</td>
<td>39.55</td>
<td>178.37</td>
<td>9.81</td>
<td>57.8</td>
<td>53.2</td>
<td>10.4</td>
<td>12.5</td>
<td>9.3</td>
<td>3.9</td>
</tr>
<tr>
<td>news</td>
<td>16.93</td>
<td>26.87</td>
<td>99.97</td>
<td>5.53</td>
<td>35.7</td>
<td>27.2</td>
<td>6.6</td>
<td>8.0</td>
<td>5.9</td>
<td>2.4</td>
</tr>
<tr>
<td>paper1</td>
<td>2.37</td>
<td>3.65</td>
<td>15.51</td>
<td>0.86</td>
<td>5.0</td>
<td>4.6</td>
<td>0.99</td>
<td>1.1</td>
<td>0.8</td>
<td>0.3</td>
</tr>
<tr>
<td>paper2</td>
<td>3.54</td>
<td>4.97</td>
<td>23.6</td>
<td>1.31</td>
<td>7.7</td>
<td>7.0</td>
<td>1.47</td>
<td>1.6</td>
<td>1.2</td>
<td>0.5</td>
</tr>
<tr>
<td>paper3</td>
<td>2.03</td>
<td>2.92</td>
<td>13.12</td>
<td>0.73</td>
<td>4.4</td>
<td>4.0</td>
<td>0.87</td>
<td>1.0</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td>paper4</td>
<td>0.63</td>
<td>0.85</td>
<td>3.91</td>
<td>0.21</td>
<td>1.3</td>
<td>1.1</td>
<td>0.3</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>paper5</td>
<td>0.60</td>
<td>0.83</td>
<td>3.58</td>
<td>0.2</td>
<td>1.2</td>
<td>1.1</td>
<td>0.3</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>paper6</td>
<td>1.75</td>
<td>2.59</td>
<td>11.13</td>
<td>0.62</td>
<td>3.7</td>
<td>3.4</td>
<td>0.73</td>
<td>0.8</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>9.11</td>
<td>13.78</td>
<td>58.35</td>
<td>3.28</td>
<td>20.06</td>
<td>18.4</td>
<td>3.68</td>
<td>4.34</td>
<td>3.2</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 8.6: Summary of the encoding, decoding and string matching times in milliseconds for different methods. The decoding encoding times for Huffman, BE, static $TSC^0$, Gamma Rice codes are denoted by $E_{Huffman}$, $E_{BPE}$ and $D_{BPE}$, $E_{TSC^0}$ and $D_{TSC^0}$, $E_{Elias}$ and $D_{Elias}$, respectively. The Aho-Corasick string matching time in the $TSC^0$ compressed domain and in the uncompressed text are denoted by $S_{TSC^0}$ and $S_{text}$, respectively.

Table 8.7 describes the encoding and decoding times in milliseconds for several $TSC$ based schemes. The results show that the encoding and decoding times of the static $TSC^0$ are similar to the encoding and decoding times of the static $TSC^1$. The encoding and decoding times of the static $TSC$ schemes are 2-3 times faster than the encoding and decoding times of the adaptive $TSC$ schemes.

Part III: Accessing XML in a limited memory device
Table 8.7: Summary of the encoding and decoding times in milliseconds for different TSC methods: static $TSC^0$, static $TSC^1$, adaptive $TSC^0$, adaptive $TSC^1$ are denoted by $E_{TSC^0}$ and $D_{TSC^0}$, $E_{TSC^1}$ and $D_{TSC^1}$, $E_{TSC^0\alpha}$ and $D_{TSC^0\alpha}$, $E_{TSC^1\alpha}$ and $D_{TSC^1\alpha}$, respectively

<table>
<thead>
<tr>
<th>FILE</th>
<th>$E_{TSC^0}$</th>
<th>$D_{TSC^0}$</th>
<th>$E_{TSC^1}$</th>
<th>$D_{TSC^1}$</th>
<th>$E_{TSC^0\alpha}$</th>
<th>$D_{TSC^0\alpha}$</th>
<th>$E_{TSC^1\alpha}$</th>
<th>$D_{TSC^1\alpha}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>2.06</td>
<td>2.4</td>
<td>1.9</td>
<td>2.5</td>
<td>4.2</td>
<td>6.1</td>
<td>5.8</td>
<td>7.3</td>
</tr>
<tr>
<td>book1</td>
<td>13.0</td>
<td>15.4</td>
<td>11.9</td>
<td>16.0</td>
<td>29.3</td>
<td>40.3</td>
<td>26.5</td>
<td>32.5</td>
</tr>
<tr>
<td>book2</td>
<td>10.4</td>
<td>12.5</td>
<td>9.6</td>
<td>12.9</td>
<td>23.2</td>
<td>32.1</td>
<td>20.9</td>
<td>25.9</td>
</tr>
<tr>
<td>news</td>
<td>6.6</td>
<td>8.0</td>
<td>6.2</td>
<td>8.3</td>
<td>17.4</td>
<td>24.6</td>
<td>13.0</td>
<td>16.2</td>
</tr>
<tr>
<td>paper1</td>
<td>0.99</td>
<td>1.1</td>
<td>0.9</td>
<td>1.1</td>
<td>2.0</td>
<td>2.9</td>
<td>1.9</td>
<td>2.3</td>
</tr>
<tr>
<td>paper2</td>
<td>1.47</td>
<td>1.6</td>
<td>1.4</td>
<td>1.7</td>
<td>5.5</td>
<td>7.5</td>
<td>2.8</td>
<td>3.5</td>
</tr>
<tr>
<td>paper3</td>
<td>0.87</td>
<td>1.0</td>
<td>0.8</td>
<td>1.0</td>
<td>1.8</td>
<td>2.5</td>
<td>2.9</td>
<td>3.7</td>
</tr>
<tr>
<td>paper4</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td>1.4</td>
<td>0.9</td>
<td>1.1</td>
</tr>
<tr>
<td>paper5</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.9</td>
<td>1.2</td>
<td>0.8</td>
<td>1.0</td>
</tr>
<tr>
<td>paper6</td>
<td>0.73</td>
<td>0.8</td>
<td>0.7</td>
<td>0.8</td>
<td>2.8</td>
<td>3.8</td>
<td>2.5</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>3.68</td>
<td>4.34</td>
<td>3.40</td>
<td>4.48</td>
<td>8.8</td>
<td>12.23</td>
<td>7.8</td>
<td>9.6</td>
</tr>
</tbody>
</table>

8.7 Conclusion

In this chapter, we introduced the TSC compression scheme that is fast and compact. Its decoding speed is gained due to its ability to perform a byte-wise processing. The byte-wise processing is enabled by its operation on compact transition tables. The current byte-wise fast compression methods [132, 131, 33] use much larger transition tables to translate a bit-wise processing into a byte-wise processing. We suggested the code-subword domain as a new domain for processing. The TSC in this chapter uses transition tables to translate a code-subword-wise processing into a byte-wise processing. This enables us to perform byte-wise processing via compact transition-tables. We introduced a fast string matching algorithm that performs byte-wise processing via the same compact transition-tables. We also formalized the TSC codes as universal codes. We extended the original code [27] into a family $TSC^k$ of universal codes that can be processed in a compact byte-wise manner. The codes in the $TSC^k$ family are suitable for several applications such as word-wise compression and network protocols compression for limited resources devices.
8.8 APPENDIX I: Extended theory

The goal of this section is to show that we can get a tight bound that is less than 2 on the optimal Huffman coding.

For two given sequences of real numbers $\alpha_1, \alpha_2, \ldots, \alpha_m$ and $\beta_1, \beta_2, \ldots, \beta_m$ with $m \in \mathbb{N}$, denote $B_k = \sum_{i=1}^{k} \beta_i$, $1 \leq k \leq m$, then

**Definition 8.9** $S = \sum_{i=1}^{m} \alpha_i \beta_i = \alpha_m B_m - \sum_{i=1}^{m-1} B_i (\alpha_{i+1} - \alpha_i)$.

It is called the Abel transformation.

8.8.1 Case I, Input grows like $2^n$

Compare between $S_1 = \sum_{i=1}^{n} 2^{n-i} \cdot i$ and $S_2 = \sum_{i=1}^{n} 2^{n-i} \cdot 2 \log_2(i + 1)$. Since $S_1 = 2^n \cdot \sum_{i=1}^{n} 2^{-i} \cdot i$ and $S_2 = 2^n \cdot \sum_{i=1}^{n} 2^{-i} \cdot 2 \log_2(i + 1)$, we can compare between the sums $S_1^* = \sum_{i=1}^{n} 2^{-i} \cdot i$ and $S_2^* = \sum_{i=1}^{n} 2^{-i} \cdot 2 \log_2(i + 1)$.

$S_1^* = \sum_{i=1}^{n} 2^{-i} \cdot i$: Define the sequences $\alpha_i = i$, $\beta_i = 2^{-i}$, then $B_k = \sum_{i=1}^{k} 2^{-i} = \frac{\frac{1}{2}}{\frac{1}{2} - 1} = 1 - \left(\frac{1}{2}\right)^k$. By using Eq. in Definition 8.9 the sum $S_1^*$ can be written as $S_1^* = \sum_{i=1}^{n} 2^{-i} \cdot i = n \cdot (1 - \left(\frac{1}{2}\right)^n) - \sum_{i=1}^{n-1} (1 - \left(\frac{1}{2}\right)^i)((i + 1) - i) = n \cdot (1 - \left(\frac{1}{2}\right)^n) - (n - 1) + 2(n - 1) = n - \frac{n}{2^{2n}} - n + 1 + 1 - \frac{1}{2^{2n-1}}$. By taking the limit $n \to \infty$ we get $S_1^* = 2$.

$S_2^* = \sum_{i=1}^{n} 2^{-i} \cdot 2 \log_2(i + 1)$; Define the sequences $\alpha_i = 2 \log_2(i + 1)$, $\beta_i = 2^{-i}$, then $B_k = \sum_{i=1}^{k} 2^{-i} = 1 - \left(\frac{1}{2}\right)^k$. By using Eq. in Definition 8.9 the sum $S_2^*$ can be written as $S_2^* = \sum_{i=1}^{n} 2^{-i} \cdot 2 \log_2(i + 1) = 2 \log_2(n + 1) \cdot (1 - \left(\frac{1}{2}\right)^n) - \sum_{i=1}^{n-1} (1 - \left(\frac{1}{2}\right)^i)2(\log_2(i + 2) - \log_2(i + 1)) = 2 \log_2(n + 1) - \frac{2 \log_2(n + 1)}{2^n} - \sum_{i=1}^{n-1} 2(\log_2(i + 2) - \log_2(i + 1)) + \sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1)) = 2 \log_2(n + 1) - \frac{2 \log_2(n + 1)}{2^n} - 2 \log_2(n + 1) + 2 + \sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1))$.

We want to bound the sum $\sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1))$. Since the first term in the sum is the largest, we take it out of the sum and bound the residual. $\sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1)) = \frac{1}{2} 2(\log_2(3) - \log_2(2)) + \sum_{i=2}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1))$. It can be shown (induction) that for $i \geq 2$, $0 < \log_2(i + 2) - \log_2(i + 1) \leq 0.42$. This yields $\sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i \cdot 2(\log_2(i + 2) - \log_2(i + 1)) \leq \log_2(\frac{3}{2}) + 2 \cdot 0.42 \cdot \frac{1}{2^n} = \log_2(\frac{3}{2}) + 0.42 \cdot 2^{1-n} = 1.00496$. Taking the limit $n \to \infty$ we get $2 < S_2^* \leq 2 + 1.00496$.

Part III: Accessing XML in a limited memory device
\[ \lim_{n \to \infty} \frac{S_n}{\phi} = \lim_{n \to \infty} \frac{S_n}{\phi} = \lim_{n \to \infty} \frac{S_n^2}{S_n} = l \text{ and } 1 < l \leq 1.503. \]

### 8.8.2 Case II, Input grows like \( F(n) \), where \( F(n) \) is the \( n^{th} \) value of the Fibonacci series

Compare between \( S_3 = \sum_{i=1}^{n} F(n-i) \cdot i \) and \( S_4 = \sum_{i=1}^{n} F(n-i) \cdot 2\log_2(i+1) \). We use the identity \( F(n) = \left\lfloor \frac{\phi^n}{\sqrt{5}} \right\rfloor \), where \( \phi = \frac{1}{2}(1 + \sqrt{5}) \) is the golden ratio. Since \( \frac{\phi^n}{\sqrt{5}} - 1 \leq \left\lfloor \frac{\phi^n}{\sqrt{5}} \right\rfloor \leq \frac{\phi^n}{\sqrt{5}} + 1 \), the two series \( S_3 = \sum_{i=1}^{n} \frac{\phi^{n-i}}{\sqrt{5}} \cdot i \) and \( S_4 = \sum_{i=1}^{n} \frac{\phi^{n-i}}{\sqrt{5}} \cdot 2\log_2(i+1) \) can be bounded by \( S_3^A = \sum_{i=1}^{n} \left( \frac{\phi^{n-i}}{\sqrt{5}} - 1 \right) \cdot i \leq S_3 \leq \sum_{i=1}^{n} \left( \frac{\phi^{n-i}}{\sqrt{5}} + 1 \right) \cdot i = S_3^B, \)

\[ S_4^A = \sum_{i=1}^{n} \left( \frac{\phi^{n-i}}{\sqrt{5}} - 1 \right) \cdot 2\log_2(i+1) \leq S_4 \leq \sum_{i=1}^{n} \left( \frac{\phi^{n-i}}{\sqrt{5}} + 1 \right) \cdot 2\log_2(i+1) = S_4^B. \]

We analyze the limits \( \lim_{n \to \infty} \frac{S_3}{S_3^A}, \lim_{n \to \infty} \frac{S_4}{S_3^A}. \)

\[ \lim_{n \to \infty} \frac{S_4}{S_3^A} = \lim_{n \to \infty} \frac{\sqrt{5} \sum_{i=1}^{n} \frac{\phi^n \phi^{n-i} - (1+n)^n}{\sqrt{5} \sum_{i=1}^{n} 2\log_2(i+1) - 2\log_2((n+1)!)}}{\sqrt{5} \sum_{i=1}^{n} \frac{\sqrt{5}(1+n)^n}{\phi^n \sum_{i=1}^{n} 2\log_2((i+1)!)}} = \lim_{n \to \infty} \frac{\sum_{i=1}^{n} \frac{\phi^n \phi^{n-i} - (1+n)^n}{\sqrt{5} \sum_{i=1}^{n} 2\log_2(i+1) - 2\log_2((n+1)!)}}{\sum_{i=1}^{n} \frac{\sqrt{5}(1+n)^n}{\phi^n \sum_{i=1}^{n} 2\log_2((i+1)!)}}. \]

Since \( \frac{\sqrt{5}(1+n)^n}{\phi^n \sum_{i=1}^{n} 2\log_2((i+1)!)} \to 0 \) and \( \frac{\sqrt{5}2\log_2((n+1)!)}{\phi^n \sum_{i=1}^{n} 2\log_2((i+1)!)} \to 0 \) when \( n \to \infty \), we can find the limits \( \lim_{n \to \infty} \sum_{i=1}^{n} \frac{\phi^n}{\phi^n \sum_{i=1}^{n} 2\log_2(i) - 2\log_2((i+1)!)}. \) Denote these sums as \( \tilde{S}_3^A = \sum_{i=1}^{n} \phi^{-i} \cdot i \) and \( \tilde{S}_4^A = \sum_{i=1}^{n} \phi^{-i} \cdot 2\log_2(i+1) \).

\[ \tilde{S}_3^A = \sum_{i=1}^{n} \phi^{-i} \cdot i: \] Define the sequences \( \alpha_i = i, \beta_i = \phi^{-i}, \) then \( B_k = \sum_{i=1}^{k} \phi^{-i} = \frac{1}{\phi-1} (1 - (\frac{1}{\phi})^k). \) Definition[8.9] yields \( \sum_{i=1}^{n} \frac{\phi^{-i}}{\phi^n} i = n \cdot \frac{1}{\phi-1} (1 - (\frac{1}{\phi})^n) = \frac{1}{\phi-1} \sum_{i=1}^{n} (1 - (\frac{1}{\phi})^i) = \sum_{i=1}^{n} (1 - (\frac{1}{\phi})^i) - \frac{1}{\phi-1} \sum_{i=1}^{n} (1 - (\frac{1}{\phi})^i) = \frac{n}{\phi-1} (1 - (\frac{1}{\phi})^n) - \frac{1}{\phi-1} \sum_{i=1}^{n} (1 - (\frac{1}{\phi})^i). \]

Taking the limit \( n \to \infty \) we get \( \tilde{S}_3^A = \frac{1}{\phi-1} + \frac{1}{(\phi-1)^2} \approx 4.237. \)

\[ \tilde{S}_4^A = \sum_{i=1}^{n} \phi^{-i} \cdot 2\log_2(i+1): \] Define the sequences \( \alpha_i = 2\log_2(i+1), \beta_i = \phi^{-i}, \) then \( B_k = \sum_{i=1}^{k} \phi^{-i} = \frac{1}{\phi-1} (1 - (\frac{1}{\phi})^k). \) Equation in Definition[8.9] yields \( \tilde{S}_4^A = 2\log_2(n+1) \cdot \frac{1}{\phi-1} (1 - (\frac{1}{\phi})^n) - \frac{1}{\phi-1} \sum_{i=1}^{n} (1 - (\frac{1}{\phi})^i) 2(\log_2(i+2) - \log_2(i+1)) + \frac{1}{\phi-1} \sum_{i=1}^{n} (\frac{1}{\phi})^i \cdot 2(\log_2(i+2) - \log_2(i+1)) = 2(\log_2(n+1) - \frac{1}{\phi-1} \log_2((\phi+1)^n) - \frac{1}{\phi-1} \sum_{i=1}^{n} 2(\log_2(i+2) - \log_2(i+1)) + \frac{1}{\phi-1} \sum_{i=1}^{n} (\frac{1}{\phi})^i \cdot 2(\log_2(i+2) - \log_2(i+1)) = 2(\log_2(n+1) - \frac{1}{\phi-1} \log_2((\phi+1)^n) - \frac{1}{\phi-1} \sum_{i=1}^{n} (\frac{1}{\phi})^i \cdot 2(\log_2(i+2) - \log_2(i+1)). \]

As before, we want to bound the last sum:

\[ \frac{1}{\phi-1} \sum_{i=1}^{n} (\frac{1}{\phi})^i \cdot 2(\log_2(i+2) - \log_2(i+1)) = \frac{2}{\phi-1} (\log_2(\frac{3}{2}) + \frac{2}{\phi-1} \sum_{i=1}^{n} (\frac{1}{\phi})^i (\log_2(i+2) - \log_2(i+1))) \leq \frac{2\log_2(\frac{3}{2})}{\phi-1} + \frac{2 \cdot 4}{\phi-1} \cdot \frac{1}{(\phi-1)^n} \cdot \log_2(\frac{3}{2}). \]

Substituting this bound into the original sum yields

---

Part [III]: Accessing XML in a limited memory device
\[
\tilde{S}_4^A \leq \frac{2}{\phi-1} - \frac{2 \log_2 (n+1)}{(\phi-1)\phi^n} + \frac{2 \log_2 \left(\frac{3}{2}\right)}{(\phi-1)\phi} + \frac{0.42}{\phi-1} \left(1-\frac{1}{2}\right)^{n-1}.
\]

Taking the limit \( n \to \infty \) we get
\[
\frac{2}{\phi-1} + \frac{2 \log_2 \left(\frac{3}{2}\right)}{(\phi-1)\phi} < \tilde{S}_4^A \leq \frac{2}{\phi-1} + \frac{2 \log_2 \left(\frac{3}{2}\right)}{(\phi-1)\phi} + \frac{2 \log_2 \left(\frac{3}{2}\right)}{(\phi-1)\phi} \approx 5.77.
\]

\( \lim_{n \to \infty} \frac{S_4^A}{S_3^A} \) and \( \lim_{n \to \infty} \frac{S_4^B}{S_3^B} \): \( \lim_{n \to \infty} \frac{S_4^A}{S_3^A} = l^A, 1 < l^A \leq 1.363 \). Calculation of
\( \lim_{n \to \infty} \frac{S_4^B}{S_3^B} \) is similar and yields \( \lim_{n \to \infty} \frac{S_4^B}{S_3^B} = l^B, 1 < l^B \leq 1.363 \).

\( \lim_{n \to \infty} \frac{S_4^A}{S_3^A} \): Since \( S_3^A \leq S_4^A \leq S_4^B \) and \( S_4^A \leq S_4^A \leq S_4^B \) then if we denote \( \lim_{n \to \infty} \frac{S_4^A}{S_3^A} = \bar{l}, 1 < \bar{l} \leq 1.363 \).

Part [III]: Accessing XML in a limited memory device
Part IV

Compressed validation of XML documents
Chapter 9

DPDT algorithm

9.1 Introduction

9.1.1 Motivation

Extensible Markup Language (XML) is the standard format for content representation (presentation) and sharing on the Web. Communication of information on machine level will ultimately be carried out through XML. XML is a highly verbose language, especially regarding the duplication of meta-data in the form of elements and attributes. As the level of XML traffic grows so is the demand to compresses XML data volume in order to reduce XML traffic bandwidth. XML on cellular communications networks [7] is a good example for the need to compress XML data.

Storing massive XML content before it is shared or presented on the Web is another need to have lossless XML compression. Again, the XML verbose nature significantly enlarges the volumes of the stored data.

It is clear that a lossless compression scheme for reducing XML volume is needed. In this chapter, we treat XML in its most basic form - as a language. Each language has a grammar. Every grammar has a parser which recognizes it. But for XML languages, this assumption is not straightforward since there is no clear definition what is an XML parser. In the XML literature, the term XML parser actually means a lexical analyzer not a parser. There is no standard way to generate XML parsers for general purposes. There is also a difficulty to determine how to transform a syntactic XML dictionary into a formal grammar definition. We use the term syntactic dictionary to address the existing XML meta data description formats that contains DTD [31], XML-Schema

Part IV Compressed validation of XML documents
Our algorithm suggests how to generate automatically an XML parser according to a given dictionary. This XML parser generator can be used in a wide variety of XML applications such as validators, converters, editors, etc.

### 9.1.2 The basic idea

A lossless compression scheme for XML data is needed. This chapter suggests a fully syntax based XML compression. We treat XML in its most general form - as a language whose underlying grammar is a variant of a context-free grammar (CFG). This is why we can benefit from twenty years of experience on the study of CFG source compression models and to implement and utilize a similar approach towards XML.

In the chapter, we exploit the common form of syntactic dictionary to produce a new XML parsing technique. Our parser construction starts from a new grammar model, which we call a dictionary grammar (D-grammar). It is similar to a CFG with the following modifications:

1. Each non-terminal symbol appears as the right-hand-side of one production;
2. The left-hand-side of a production includes a RE that is enclosed by a unique pair of tagging characters.

This is a general approach towards XML manipulation. It creates a generic framework for XML processing. The XML parser accepts the D-grammar of documents described by this dictionary, which is the input dictionary. We call this process, which constitutes the core of this chapter, XML parser generation. This framework is used to achieve XML compression.

The work in [94] suggested to use specific syntactic compressors that are planted inside the XML compression. When an XML document type is specified by a CFG, its definition can easily be expanded to include other CFG grammars. For example, if we want to syntactically encode URL addresses inside an XML document, we can expand the XML grammar with the URL grammar. URL address definition is even more restrictive than XML. It can be defined as a RE. The following RE illustrates the URL address structure:
The 'free-text' is a predefined lexical symbol of the free text. Most of the structures that reside inside XML documents such as numbers, dates, IP addresses etc., will be compressed by XML lossless compression.

The proposed XML parser can be used for applications other than compression. The fact that this is a simple and fast generator of parsers, makes this parser generation technique very practical. Unlike common parsers that use prediction table for parsing, our XML parser uses a FSM instead of a table to determine the next production rule to be used for derivation. The FSM, which has a reduced number of states, serves as a compact prediction table. The parser takes into consideration the XML structure and its operation becomes efficient. The suggested XML parser generator can fit a wide variety of XML applications such as validators, converters, editors, etc (see [84]).

9.1.3 Outline of the lossless XML compression algorithm

The flow of the algorithm is given in Fig. 9.1. It contains three sub-modules:

1. **Dictionary conversion** - converts the dictionary to D-grammar (see section 9.3.3);

2. **XML parser generator** - creates an XML parser from its D-grammar;

3. **XML encoding** - encodes the XML parsers' moves.
Each element in the dictionary can be rephrased as a RE. This translation to D-grammar representation precedes the parser generation. We construct a Dictionary Deterministic Pushdown Transducer (DPDT) that acts as a parser for the given D-grammar (see section 9.3.3).

The third phase of the encoding algorithm uses the Partial Prediction Matching (PPM [47]), which is considered to be the state-of-the-art for text encoding. The encoder uses the XML parsing process to decide which are the lexical symbols that are relevant to the current elements’ state. Only these symbols participate in the encoding process.

The decoder decodes the lexical symbols and sends them to the XML parser. The parser transforms it to its original XML format and writes it to a file.

A preliminary version of the basic DPDT-L algorithm was described in [74]. It neither provides the theoretical infrastructure nor updated benchmarks which are described here. This chapter details the encoding scheme, formalizes the theory of DPDT generation and operation (see section 9.3) and new compression algorithms are applied to new benchmarks.

Figure 9.1: Flow of the XML lossless compression algorithm: the main components

Part IV Compressed validation of XML documents
9.1.4 Main results

The comparison between the performance of our algorithm (DPDT-L) and the XMLPPM algorithm [40] is given in [74]. In this chapter, we update and enhance the set of compression tools for which DPDT-L is compared against. Special emphasis is given to comparison with the compression techniques such as XMLPPM [40] and SCMPPM [13] that use the same PPM encoder as we do. We also compare with two DTD conscious encoders that are also based on PPM encoder: DTDPPM [81] and XAUST [73]. On average, our codec outperforms the other methods.

In [74] we evaluate the compression performance on small dataset (1MB). In this chapter, the datasets are extended to medium (10MB) and large (100MB) sizes datasets.

The structure of the chapter is as follows. Related compression and parsing algorithms are given in section 9.2. Section 9.3 describes the XML compression algorithm. The results after the completion of the application of the algorithm on standard benchmark datasets are given in section 9.4.

9.2 Related work

In this section, we mention the main current XML compression methods. They are compared with our XML encoder design philosophy.

9.2.1 Context-Free-Grammar (CFG) encoding models

Over the past twenty years there have been attempts to find the best CFG encoding scheme. Two compression techniques emerged: the derivational and the guided-parsing techniques. The core of the derivational technique [37, 52, 86] is a step-by-step transmission of the derivation of a string from the goal symbol. At each step, the leftmost non-terminal is rewritten according to the grammar. Each non-terminal can only be rewritten by certain production rules. The derivational technique encodes the production rules choices.

The guided-parsing encoding method [16, 58, 134] is based on recording the moves a parser makes while parsing the text. Stone (and Al-Hussaini) choose LR(1) parsers for their broad coverage and thorough exploitation of grammatical information. Evans [58] applied it to both LR(1) and LL(1) parsers. Evans pointed out that the derivational
metaphor is actually the same as the guided parsing metaphor, since e.g., the derivational method replays the LL(1) parser’s moves. In the rest of the chapter, we refer to both techniques as **LL guided parsing** and **LR guided parsing** encoding methods.

Section 9.2.1.1 describes the LL guided-parsing encoding technique. We focus on this technique because it is the basis for our encoding method. Section 9.2.1.2 compares between LR guided parsing and LL guided parsing techniques. Section 9.2.1.3 describes how the guided-parsing encoding methods are used.

### 9.2.1.1 LL guided parsing encoding models

The encoder in LL guided parsing, sends a series of production rules that derives the encoded string. The production rules series can be extracted from the LL(1) parsing process. Each time the top of the stack contains a non-terminal, a decision using a **decision table** is made on the next production rule to execute the derivation. LL guided-parsing encodes these decisions. We demonstrate the LL guided parsing encoding process on the XHTML document in Fig. 1.1. We use a single XHTML document (continues example) through this chapter to demonstrate our encoding concepts. Figure 1.1 shows a simple XHTML example document. Figure 1.1a shows the textual XML syntax of the example. Figure 1.1b illustrates how the XML document is represented on the WEB.

Figure 2.3 shows the DTD of the XHTML example introduced in Fig. 1.1. This DTD defines a subset of the XHTML. We use this DTD to demonstrate our encoding principles. DTD is one example for an XML syntactic dictionary. It can be shown to fit XML Schema. Figure 9.2 defines the $CFG$ of an XHTML subset. We leave out the attributes definitions to simplify the presentation.

Part IV Compressed validation of XML documents
Figure 9.2: A \textit{CFG} definition of the XHTML subset that was declared in Fig. 2.3. Only the elements are defined in this grammar. A \texttt{html} element (PR.1) with an header and a body elements are defined. The header element (PR.2-3) has an optional title element (PR.4). The body element (PR.5-7) contains multiple paragraph elements (PR.8-11). Each paragraph contains a mixture of image elements (PR.12) and a free text.

The decision table in Fig. 9.2 grammar is given in Fig. 9.3

<table>
<thead>
<tr>
<th>Index</th>
<th>Rule Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR.1</td>
<td>\texttt{html} \to \texttt{html&gt;head body \textless /html&gt;;}</td>
</tr>
<tr>
<td>PR.2</td>
<td>\texttt{head} \to \texttt{\textless head&gt; title \textless /head&gt;;}</td>
</tr>
<tr>
<td>PR.3</td>
<td>\texttt{head} \to \texttt{\textless head&gt; \textless /head&gt;;}</td>
</tr>
<tr>
<td>PR.4</td>
<td>\texttt{title} \to \texttt{\textless title&gt; #PCDATA \textless /title&gt;;}</td>
</tr>
<tr>
<td>PR.5</td>
<td>\texttt{body} \to \texttt{\textless body&gt; body_c \textless /body&gt;;}</td>
</tr>
<tr>
<td>PR.6</td>
<td>\texttt{body_c} \to \texttt{p body_c}</td>
</tr>
<tr>
<td>PR.7</td>
<td>\texttt{body_c} \to \texttt{p \textless /p&gt;;}</td>
</tr>
<tr>
<td>PR.8</td>
<td>\texttt{p} \to \texttt{\textless p&gt; p_c \textless /p&gt;;}</td>
</tr>
<tr>
<td>PR.9</td>
<td>\texttt{p_c} \to \texttt{\textless img&gt; p_c}</td>
</tr>
<tr>
<td>PR.10</td>
<td>\texttt{p_c} \to \texttt{#PCDATA p_c}</td>
</tr>
<tr>
<td>PR.11</td>
<td>\texttt{p_c} \to \texttt{p_c}</td>
</tr>
<tr>
<td>PR.12</td>
<td>\texttt{\textless /img&gt;}</td>
</tr>
</tbody>
</table>

Figure 9.3: A decision table of the \textit{CFG} that is defined in Fig. 9.2. Each terminal symbol that is a lookahead symbol defines a row. Each non-terminal symbol defines a column. When the LL-parser has a non-terminal symbol at the top of its stack, it extracts the production rule from the cell denoted by this non-terminal and the lookahead symbol.

The LL parsing process is illustrated in Fig. 9.4

Part IV Compressed validation of XML documents
Figure 9.4: The parsing process of the XHTML document that was defined in Fig. 1.1. The parser recognizes the grammar that is defined in Fig. 9.2. The **lookahead** column describes the lookahead terminal symbols. The **stack** column shows the contents of the stack during the parsing. Each cell shows the stack as a set of strings delimited by commas. The gray strings are terminal symbols and the black strings are non-terminals symbols. This stack symbol is the leftmost string in the top of the stack. When the top of the stack is a non-terminal symbol (black), the parser decides by using Fig. 9.3 decision table which production rule to apply. The **rule** column describes this production rule. This illustration is not complete. The second paragraph of the body element is missing. Its parsing is the same as the first paragraph. It applies the production-rules: PR.6, PR.10, PR.9, PR.12, PR.11 and PR.7.

The LL guided-parsing compression encodes the production rules choices which the LL parser applies. In the parsing example of Fig. 9.4, the rules column content is being encoded. The naive approach is to enumerate all the production rules globally and to use the global production number (GPN) \[137\] as the encoder’s symbols. In the above example, the GPN of each production rule is its index, as appears in the index column in Fig. 9.2. The encoded symbols are:

**GPN:** PR.1, PR.3, PR.5, PR.6, PR.10, PR.9, PR.12, PR.11, PR.7.

Part IV Compressed validation of XML documents
The compression performance of \( GPN \) is not sufficiently good. Cameron\[^{[37]}\] suggested to use a local production rule number (LPN)\[^{[137]}\]. The LPN sequencing disposes wider determinism level. Each non-terminal has a limited production set that can derive it. The production rules, in which it appears in the left side, are enumerated. The matched LPN number is encoded each time this non-terminal is derived. For example, when the decision table columns in Fig.\[^{[9.3]}\] is examined, we see that there are three non-terminals which have a choice of multiple production rules: ‘head’, ‘body\(_c\)’ and ‘\(p\_c\)’. We sort the production-rules of each non-terminal by its indices and enumerate them. For example, for the non-terminal ‘head’, the local enumeration is: 1(PR.2) and 2(PR.3). This enumeration is the local production number. The local encoded symbols of the above example are:

\[
\text{LPN: } -, 2[2], -, 1[2], -, 2[3], 1[3], -, 3[3], -. \]

The ‘-’ character denotes a missing symbol that is encoded globally but not locally. The square brackets indicate the number of local enumerations that each symbol has.

### 9.2.1.2 LR vs. LL guided-parsing encoding models

LR guided parsing encoding is based on the information the parser has when a grammatical conflict occurs. Two types of conflicts are handled:

1. **Shift/Shift** - The encoder has to supply the lookahead symbol.

2. **Reduce/Reduce** - The encoder indicates the production rule.

The shift/reduction conflicts are not allowed in a legal LR grammar.

LR guided parsing exploits determinism whenever it occurs. The disadvantage of LR guided parsing is that during encoding top-down information is lost because of the bottom up nature of the LR parsing process. Because of its top down nature, the LL guided-parsing encoding exposes dependencies in the text that would otherwise remain hidden. Encoding of production rules implies that several terminals, which are part of the production rule derivation strings, are encoded by one symbol. But LL guided-parsing can also separate terminals by encoding the non-terminals in between neighboring terminals symbols. This phenomena is known as order-inflation\[^{[89]}\]. Worse than order inflation, it even unclear whether additional non-terminals are needed. This phenomena is called redundant categorization\[^{[89]}\].
redundant categorization, poorly affect the encoding quality. Our encoding algorithm is a top down in its nature. But it encodes terminals instead of production rules. The encoding of terminals prevent the order inflation and redundant categorization phenomena to occur.

9.2.1.3 Encoding methods for CFG models

A chronological view of related works identifies the evolution of encoding methods. In the 1980s, [134], [16], [48], [86] used Huffman coding to compress Pascal source-file corpus. In the late 1980, [37] targeted Pascal programs and used arithmetic coding. During the 1990s, programming languages have has been changed from Pascal to Java. [58] applied arithmetic coder to both Java and Pascal sources. [64] applied LZW on Java files. [137] used PPM algorithm to reduce the size of Pascal sources. In recent years, the CFG compression goal has changed from compression of static archives to reduced throughput of dynamic XML and Java byte codes transmissions. [52] compressed Java mobile code with arithmetic coder. [135] adopted PPM for the same purpose. [40] encoded XML lexical symbols using a PPM algorithm. [89] used PPM to encode Scheme source code. Our encoding algorithm follows the trail of CFG source encoding methods and use PPM to encode the text in XML documents.

9.2.2 XML parsing

Current XML parsing theory is based upon regular tree grammars. In regular tree grammars, perspective XML documents are handled as textual representations of trees. Therefore, a dictionary specifies the structure of the trees. Various automata were introduced to implement tree grammars for XML parsing.

Three restrictive classes of regular tree grammars and their automata are defined in [104]. Each class defines and exposes the expressive power of a different XML schema language:

1. Local tree class defines the expressive power of the DTD schema language [31];
2. Single type class defines the expressive power of the W3C XML Schema language [139];

Part IV Compressed validation of XML documents
3. Regular tree grammar defines the expressive power of RelaxNG schema language \[45\].

Parsing of regular tree grammar is not deterministic. It may provides more than one interpretation of a document. As a result, its parsing time is not bounded by the length of the document. Therefore, it is impractical.

D-grammar parsing, which is described in section \[9.3.3\] is deterministic. Its expressive power equals to local tree class expressiveness. However, it can easily be adopted to express single class type of languages. Therefore, we can use most XML syntactic dictionaries that rely on this subclass: DTD \[31\], XML-schema \[139\] and deterministic RELAX NG \[45\] documents. Therefore, although the chapter uses DTD as its underlying explanatory demonstration, it can fit other syntactic dictionaries.

### 9.2.2.1 Parsing of XML streams

Most of the proposed tree grammars automata have a major disadvantage. They are incapable to process XML streams. Neumann \[109\] constructed a top-down automaton for regular tree grammars, which parses XML streams.

Our D-grammar automaton, called DPDT, fits to process XML streams. DPDT resembles the Neumann’s automaton in its use of the REs in the automata construction. But non-determinism complicates Neumann’s automaton. Neumann’s automaton has three sets of states. DPDT has a standard single set of states. This makes the DPDT construction more compact. Terminals participation in the production rules writings makes D-grammar a natural way to describe XML attributes in particular and XML in general. In summary, we show that DPDT is a natural parser for XML documents.

### 9.2.2.2 XML validation

DTD validation of streaming XML documents under memory constraints was investigated in \[95\]. They showed the existence of an automaton with a bounded stack that is related to the depth of the XML document. This automaton has \(2^{|\Sigma|}\) states. DPDT also produces a strong XML validation. DPDT’s automaton stack size is bounded by the depth of the XML document. The state space of the DPDT is more compact than the automaton in \[95\]. In most cases, the state space of the DPDT is linear in the size of the D-grammar.
The bounded stack size of the DPDT enhances the compression. It bounds the PPM context that predicts the encoded symbol. It makes D-grammar optimal for compression of XML documents that are grammar based.

### 9.2.3 XML Compression

XML compression is important mainly for two WEB applications: storage and transmission. The verbose nature of XML is disturbing for both. The static nature of storage usually allows it to use general encoders to achieve high compression ratios \[94, 40, 140, 50, 83, 20, 91, 13, 81, 73\]. XML DB compressions have two variants: generic compression \[94, 40, 50, 91, 13, 81, 73\] and query enabling compression \[140, 83, 20\]. Query enabling compression takes into consideration a query mechanism which is applied to the stored XML data.

The encoding models in XML compression differ in several parameters:

1. The compressor can be either streaming or not.

2. They have different ways to compress the document’s content and its structure. XML’s content contains the text (\#CDATA and \#PCDATA) of the XML document. XML structure contains all the tags, attributes and special characters in the XML document. Cheney [82] defined two models for content encoding:
   - Multiplexed Hierarchical Modeling (MHM). The MHM approach switches among several PPM models.
   - Structural Context Modeling (SCM). In SCM, rather than switching among a small number of models that are based on the syntactic class of the data, the compressor uses a separate model to compress the content under each element symbol.

3. The encoding model in the XML compressors differs in the underlying encoding algorithm. It can utilize byte codes, LZW, Huffman, arithmetic coder and PPM.

4. How the compression exploits the structural information in the DTD.

This chapter presents a new streaming compressor called DPDT-L. It is a generic DB XML encoder that uses MHM approach with an underlying PPM encoder. The presented compressor switches between two PPM models: a structural model that encodes
the XML validation decisions and a content model. Section 9.2.3.1 describes how other XML encoders model the structure and the content.

The DPDT-L encapsulates the structural information of the DTD in the validator operation. Several other encoding methods are also aware of the DTD structural information. Section 9.2.3.2 describes how different compressors exploit the structural information in the DTD.

### 9.2.3.1 XML encoding models

Transmission applications use byte codes to transfer the encoded source. It can be either a fixed byte-code [7, 69, 111] or a variable length byte-code [84, 136]. The most advanced encoding for transmission application was presented in [69, 136].

In order to be able to query the structure, most query enabled encoders separate structural compression from content compression. XMLzip [50] splits its content according to a certain depth of the XML tree structure and uses LZW to compress each sub-tree. XQueC [20] even separates between each path encoding. XQueC uses Huffman coding for encoding the structure and ALM for encoding the content. XGrind [140] uses Huffman coding to encode the structure and arithmetic coding for encoding the content.

Generic DB XML encoders use variety of encoding methods. XMill [94] splits the text of the XML document into containers and compresses each container using a text compressor such as gzip, bzip2 or PPM. XMill also uses semantic compressors to encode data items with a particular structure. The semantic compressors are based on a parser for a regular grammar.

XMLPPM [40] is a streaming compressor that uses an MHM encoding approach. The XMLPPM switches among several PPM models, one for element, attribute, character, and miscellaneous data, and “injects” element context symbols into the other models to recover lost accuracy due to model splitting. XMLPPM uses PPM as its underlying compressor.

SCMPPM [13] is a variant of the XMLPPM that uses the SCM encoding approach. AXECHOP [91] uses XMill’s container approach to encode text content and grammar based compression to encode the element structure of the document. XAUST [73] compressor takes advantage of the DTD information to compress the element structure and uses SCM encoding approach to compress the content (albeit using order-4 arithmetic.
coding rather than PPM). DTDPPM [81] is a DTD conscious extension of XMLPPM.

9.2.3.2 DTD awareness

The initial XML compression algorithms [94, 40, 50, 69] ignored the DTD information. Xcompress [92] and XGrind [140] extract the list of expected elements from the DTD and encodes the index of the element instead of the element itself. More sophisticated approach is used in the Millau project [136]. It creates a tree structure for each element that is specified in the DTD. The tree includes the relation to other elements, such as the special operator nodes for the RE operators that define the element content. The XML data is also represented as a tree structure. The DTD and the XML trees, are scanned in parallel and only the delta between the two representations is encoded. This method is called differential DTD. The same compression method was addressed more formally in [92]. Differential DTD does not extract the whole information from the DTD. Attribute definition of the DTD is not used by this method.

DTDPPM use of DTD is primarily to provide information about the element and attribute structure while supplying little information about text content. It removes whitespace from the XML document. The presented algorithm also removes whitespace from the XML document.

AXECHOP [91] generates a CFG that is capable of deriving this XML structure. This grammar is passed through an adaptive arithmetic coder before being written as a compressed file. The DPDT-L approach also generates a grammar that is capable of deriving this XML structure. But we use the D-grammar that is dedicated for describing the XML structure. A CFG description is too general for XML description.

XAUST [73] creates a FSM for each element in the DTD. The FSM describes the element content. In each encoding step, XAUST encodes the current element and the current state in the FSM of the element. The DPDT-L algorithm generalizes the XAUST algorithm. It combines the set of FSMs to a single automaton called DPDT. It encodes a single DPDT state in each encoding step instead the pairs \(<element, state>\). Furthermore, it encodes the state locally and not globally as XAUST does. This generalization enables the DPDT-L algorithm to combine the validation with the encoding process. The DPDT-L algorithm was developed independently of XAUST. The patent ([23]), which is based on the DPDT-L algorithm, was filed before the publication of XAUST.
9.2.4 Prediction by Partial Matching (PPM) encoding

A context is a finite length suffix of the current symbol. A context model is a conditional probability distribution over the alphabet that is computed from the contexts. PPM\(^\text{47}\) is a finite context model encoding. The context model encoding uses the context model to predict the current symbol. The prediction is encoded and sent to the decoder. The context model is then updated by the current symbol and the encoding continues. A finite context model limits the length of contexts by which it predicts the current symbol. When the current context does not predict the current symbol, a special ‘escape’ event signals this fact to the decoder and the compression process continues with the context that is one event shorter. If zero length context does not predict the current symbol, the PPM uses an unconditional ‘order-1’ model as its baseline model.

We use in our encoding algorithm a variant of the PPM\(^\text{D}\)\(^\text{143}\) that improves the basic PPM compression twofold: escape probability assignment and scaling. The ‘D’ escape probability assignment method treats the escaping events as a symbol. When a symbol occurs it increments both the current symbol and the ‘escape’ symbol counts by 1/2. ‘D’ method is generally used as the current standard method, due to its superior performance.

The ‘+’ term insinuates a scaling technique that the algorithm uses. Scaling here means distortion of the probabilities measurements in order to emphasis certain characteristics in the context. Two characteristics are scaled: if the current symbol was recently predicted in this context (recent scaling) or if no other symbol is predicted in this context (deterministic scaling).

The PPM\(^D\) algorithm uses arithmetic coder to encode its predicted symbols.

9.3 XML compression: the DPDT-L algorithm

The XML compression algorithm has two sequential components:

1. Generation of XML parser from its dictionary. Throughout the rest of the chapter we use the DTD as an illustrative example of a dictionary. The same works for XML Schema and others.

2. XML compressor that uses the parser from the first component.
In the first component, the dictionary is converted into a set of REs. Each XML element is described as a single RE - see section 9.3.1. Then, an XML parser is generated from this description in the following way. A Deterministic Pushdown Transducer, which produces a leftmost parse, is generated - see section 9.3.3. This parser is similar to a LL parser. The output of the parser - namely the leftmost parse - is used as an input to the guided parsing compressor, which constitutes the second component of the algorithm - see section 9.3.6.

The guided parsing compression has three components:

1. The XML tokenizer accepts the XML source and outputs lexical tokens;
2. The XML parser parses the lexical tokens;
3. The PPM encodes the lexical symbols using information from the parser.

The algorithm’s flow is given in Fig. 9.5. The vertical flow describes the sequential stages. The horizontal flow describes the iterative parsing and the encoding process. Two parsers, XML parser (3b in Fig. 9.5) and the parser’s generation (2c in Fig. 9.5) operate independently. They contain the same iterative process.
In the next sections, we give detailed descriptions for various components in the XML compression algorithm as they appear in Fig. 9.5.

### 9.3.1 Dictionary conversion

We describe now the flow of 1a (dictionary conversion) in Fig. 9.5. The dictionary is translated into a set of REs. An XML element is described as a concatenation of a start tag string, attributes list, the element’s content and the end tag string. The RE syntax is given as:

```
"<element" attributes ">" element-content ">"
```

Figure 9.6 describes the RE description of the XHTML subset. The RE is converted from the original DTD (Fig. 9.6a). The attributes are described as a concatenation of the pair attribute and value. Implied attributes are described with the optional operator character ‘?’. Text free attribute values are described with the reserved string CDATA. A selection of attribute values is described as in the DTD. Figure 9.6b shows all the attributes that were converted to RE:

Part IV: Compressed validation of XML documents
1. The ‘src’ attribute of the ‘img’ element is an explicit attribute with a free text value. Its RE conversion is ‘src CDATA’.

2. The ‘name’ attribute of the ‘img’ element is an implicit attribute with a free text value. Its RE conversion is ‘?(name CDATA)’.

3. The ‘text’ attribute of the ‘body’ element is an explicit attribute with selection of the values ‘black’ or ‘white’. Its RE conversion is ‘text (black|white)’.

The reserved PCDATA string is used for free text elements. See for example the title element content.

```xml
<!ENTITY color (black|white)>
<!ENTITY code #CDATA>
<!ELEMENT html head body >
<!ELEMENT head title? >
<!ELEMENT title #PCDATA >
<!ELEMENT body p* >
<!ATTLIST body
text %color
background %color
>
<!ELEMENT p (img | #PCDATA)*
>
<!ELEMENT img p*>
<!ATTLIST img
src %code
name %code
>
```

Figure 9.6: DTD conversion of XHTML subset. Left: DTD description of its HTML subset. Right: Regular expression description of the HTML subset

Part IV: Compressed validation of XML documents
9.3.2 The RE lexer

We describe now the flow of 2b (RE lexer) in Fig. 9.5. The RE has three tokens types: 1. RE operator’s characters; 2. XML reserved character; 3. Textual tokens.


The XML reserved character > marks the end of element character. It distinguishes between elements and attributes to enable the tokenizer to determine which symbol to produce.

The RE lexer has three functions: 1. Tokenizes a RE; 2. Generates a lexical symbol from tokens; 3. Classifies textual token by its XML entity types which are element, attribute and attribute’s value.

A FSM with three states is being used to tokenize the RE (see Fig. 9.7). Each state fits a different XML entity type. Each token is replaced with a lexical symbol. The lexical symbol is given to the XML parser generator as an input symbol. It is saved in the lexer for a future use by the next analyzed tokens and by the XML lexer. The XML lexer inherits its symbols’ table from the RE lexer. The XML entity type, which is known according to the current lexer state, is also saved. The XML entity type will be used by the XML lexer (see section 9.3.4) in order to correctly represent a decoded token.

Part IV Compressed validation of XML documents
9.3.3 Parser Generator

This section presents the parsing algorithm of an XML file. Note that we use the term parsing as it appears in Computer Science literature (e.g. Formal Language Theory, Compilers, etc.). This is in contrast to the use of the term parsing in some of the XML literature, as noted in Section 9.2.2.

We rely on the fact that the dictionaries of an XML file constitute an Extended Backus Normal Form (EBNF) grammar for the rest of the file. EBNF grammars are not strictly CFGs, because they use some form of REs in the right-hand-side of their productions. On the other hand, each XML element is delimited by a unique pair of start tag and end tag (in angled brackets). This fact is used to simplify the parsing process.

For example, `<html>` is the right bracket of the first RE in Fig. 9.6 and `</html>` is the left bracket. None of them appear elsewhere in the grammar.

In our presentation, we will consider the special form for a dictionary grammar, which we call D-grammar. We assume that the reader is familiar with the basics of Automata, Language and Parsing Theory ([15]). Its notation is adopted here.
Definition 9.1 A D-grammar is a 4-tuple \( G = (N, \Sigma, P, A_1) \) where \( N = \{A_1, A_2, \ldots, A_n\} \) is a finite non-empty set of non-terminals, \( \Sigma \) is a finite non-empty set of terminal symbols, divided between two disjoint subsets \( \Sigma = \{a_1, a_2, \ldots, a_n, \bar{a}_1, \bar{a}_2, \ldots, \bar{a}_n\} \cup \Sigma' \) where \( \Sigma' \) is a collection of attributes. \( A_1 \) is the start symbol, and \( P \) is a non-empty set of bracketed productions, with the following form: each non-terminal \( A_i \) has a unique production \( A_i \to a_i R_i \bar{a}_i \), where \( a_i, \bar{a}_i \in \Sigma \) are the left and right bracket for \( A_i \), respectively, and \( R_i \) is a RE over \( N \cup \Sigma' \) (we will call it \( A_i \)’s RE). Note that the brackets of different non-terminals are distinct.

For example, in the grammar of Figure 9.6 \( N = \{html, head, title, body, p, img\} \), \( A_6 = \text{img} \), \( a_6 = \text{"<img"} \), \( \bar{a}_6 = \text{"</img>"} \), and \( R_6 = \text{src CDATA name CDATA >p *}. \)

A D-grammar is used to derive words in \( \Sigma^* \) by repeatedly applying production to a non-terminal symbol. This is similar to the way a CFG is used, except that the right hand side of a production is not a fixed word, like in a CFG, so when a production of \( A_i \to a_i R_i \bar{a}_i \) of a D-grammar is applied to \( A_i \), \( A_i \) is replaced by an arbitrary word \( a_i \beta \bar{a}_i \), such that \( \beta \in R_i \).

More formally, we define

Definition 9.2 Let \( G = (N, \Sigma, P, A_1) \) be a D-grammar. We define the relation \( \Rightarrow \) (read “derives”) on words over \( N \cup \Sigma \) as follows. If \( A \in N \), \( \alpha, \gamma \in (N \cup \Sigma)^* \), \( A \to a R \bar{a} \in P \) and \( \beta \in R \), then \( \alpha A \gamma \Rightarrow \alpha \beta \gamma \). We will also say that \( \alpha A \gamma \Rightarrow \alpha \beta \gamma \) uses the production \( A \to a R \bar{a} \in P \). If \( \alpha \in \Sigma^* \), then we call the derivation leftmost, and denote it by \( \alpha A \gamma \Rightarrow_L \alpha \beta \gamma \). (Henceforth we will be interested only in leftmost derivations). We use the usual notation for the reflexive transitive closure of the derives relation to indicate derivation of any length: If \( \delta_0 \Rightarrow_L \delta_1 \Rightarrow_L \ldots \Rightarrow_L \delta_m \) for some \( m \geq 0 \), then we write \( \delta_0 \Rightarrow^*_L \delta_m \).

Further, if for each \( j, 1 \leq j \leq m-1 \), \( \delta_j \Rightarrow_L \delta_{j+1} \) uses production \( A_{i_j} \to a_{i_j} R_{i_j} \bar{a}_{i_j} \in P \), then the leftmost parse of the derivation \( \delta_0 \Rightarrow^*_L \delta_m \) is the sequence of production numbers \( i_0 i_1 \ldots i_{m-1} \) which we will denote \( \pi(\delta_0 \Rightarrow^*_L \delta_m) \).

The language defined by a non-terminal symbol \( A_i \), is \( L(A_i) = \{w \in \Sigma^*|A_i \Rightarrow^*_L w\} \). The language defined by the grammar is simply the language defined by the start symbol \( A_1 \).

We will now show how to construct a Deterministic Pushdown Transducer (DPDT) that acts as a parser for the given D-grammar. A DPDT is a pushdown automaton with output. First we present a definition of a DPDT adapted from [15], but simplified: For our purpose, we need not be concerned with \( \epsilon \) moves.

Definition 9.3 A (\( \epsilon \) free) Deterministic Pushdown Transducer, (henceforth simply DPDT) is a 8-tuple \( M = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F) \) where \( Q \) is a finite set of states, \( \Sigma \) is a finite
input alphabet, $\Gamma$ is a finite pushdown alphabet, $\Delta$ is a finite output alphabet, $\delta$ is a function from $Q \times \Sigma \times \Gamma$ to $Q \times \Gamma^* \times \Delta^*$ called the transition function, $q_0 \in Q$ is the initial state, $Z_0$ is the initial stack symbol, and $F \subseteq Q$ is the set of final or accepting states.

A configuration of $M$ is a 4-tuple $(q, w, \gamma, v)$ in $Q \times \Sigma^* \times \Gamma^* \times \Delta^*$, where $q$ is the current state of $M$, $w$ is the unread portion of the input, $\gamma$ is the content of the stack, (its leftmost symbol is the top of the stack), and $v$ is the output produced so far.

A move of $M$ is represented by a relation $\vdash$ between configurations, defined as follows: $(q, aw, Z \alpha, v) \vdash (p, w, \gamma \alpha, vu)$ if $\delta(q, a, Z) = (p, \gamma, u)$, for some $q, p \in Q, a \in \Sigma, w \in \Sigma^*, Z \in \Gamma, \gamma, \alpha \in \Gamma^*$ and $v, u \in \Delta^*$.

We use $\vdash^*$ to denote a computation of any length.

A word $w$ is accepted by $M$ and translated into $v$ if $(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, v)$ for some $p \in F$: when $M$ is started in its initial state, with the stack containing the initial symbol, and with $w$ in its input, it terminates in a final state, with an empty stack, having consumed all its input, and produced $v$ as its output.

We will now present the DPDT $M$ that is constructed to act as a parser for a given D-grammar. Given a word $w \in \Sigma^*$, if $w$ is generated by the D-grammar, then given $w\$$ as input, (where $\$$ is a special end marker), $M$ will read the input to completion, terminate in an accepting state and empty the stack, and produce as output the leftmost parse $\pi(A_1 \Rightarrow^*_L w)$. Otherwise the DPDT will reject $w\$$ - it will not terminate as described.

The construction of $M$ is defined as follows.

**Definition 9.4** Let $G = (N, \Sigma, P, A_1)$ be a D-grammar, and let $M_0, M_1, M_2, \ldots, M_n$ be Finite State Automata (FSA), so that for $i \geq 1$, $M_i$ accepts the language $R_i, A_i$’s RE. The FSA $M_0$ is added to simplify the construction. It accepts the language $\{A_1\}$.

In particular, $M_i = (Q_i, N \cup \Sigma', \delta_i, q_{0i}, F_i)$. For $M_0$, specifically, $Q_0 = \{q_{00}, f_0\}$, $F_0 = \{f_0\}$, $\delta_0(q_{00}, A_1) = f_0$ and $\delta_0$ is undefined elsewhere. We assume, without loss of generality, that the sets of states $Q_i$ are disjoint.

We now define a DPDT as follows: $M = (Q, \Sigma \cup \{\$$\}, \Gamma, \Delta, \delta, q_0, Z_0, \{f_0\})$ where $Q = \bigcup_{i=0}^n Q_i$, $\Gamma = \{Z_0\} \cup \{[q, a_i] | q \in Q, 0 \leq i \leq n\}$. The output alphabet $\Delta = \{1, 2, \ldots, n\}$ represents production numbers. The transition function $\delta$ has four types of rules, depending on the type of input symbol:

1. For all $1 \leq i \leq n, 0 \leq j \leq n, Z \in \Gamma$ and $q \in Q_j$, we have $\delta(q, a_i, Z) = (q_{0i}, [\delta_j(q, A_i), a_i]Z, i)$ (left bracket).

2. For all $1 \leq i \leq n, q \in Q$, and $p \in F_i$, we have $\delta(p, a_i, [q, a_i]) = (q, \epsilon, \epsilon)$ (right bracket).
3. For all $0 \leq i \leq n$, $q \in Q_i$, $a \in \Sigma'$ and $Z \in \Gamma$, we have $\delta(q, a, Z) = (\delta_i(q, a), Z, \epsilon)$ (non bracket symbol).

4. $\delta(f_0, \$, Z_0) = (f_0, \epsilon, \epsilon)$ (end marker).

$\delta$ is undefined for all other values of its arguments.

In the sequel, we will use $^i_\top$ (and $^i_\top*$) to denote a computation step (sequence of steps) of type $i$.

It can easily be seen that $M$ is deterministic, and has no $\epsilon$ moves.

$M$ operates as follows. When given non bracket symbols, $M$ simulates the behavior of an individual FSM in its state, each time following a word $\beta$ to see if it belongs to a specific $R_j$ (type 3 moves). Whenever a left bracket $a_i$ appears in the input, the DPDT must suspend its simulation of the current FSM $M_j$, pushing onto the stack a symbol that combines the state $q \in Q_j$ from which this simulation is to be resumed later (explained below), and the left bracket $a_i$. $M$ then starts a simulation of the RE $R_i$ by changing its state to the initial state $q_{0i}$ of the corresponding FSM $M_i$ (type 1 move). Whenever a right bracket $\overline{a}_i$ is read, $M$ must be in an accepting state $p \in F_i$ of the current FSM being simulated $M_i$. Further, the right bracket being read $\overline{a}_i$ must match the left bracket $a_i$ on the stack. If these conditions hold, then the stack symbol $[q, a_i]$ is popped and the simulation resumes from the state $q \in Q_j$ (type 2 move).

The state $q \in Q_j$ from which simulation is to be resumed (which is pushed onto the stack along with the right bracket) is computed as follows. The right bracket $a_i$ that causes suspension uniquely determines the non-terminal symbol $A_i$ for which a derivation step is considered. When the simulation of $M_i$ is completed in an accepting state, and followed by the appearance of $\overline{a}_i$ in the input, this corresponds to completion of the right hand side of the production $A_i \rightarrow a_iR_i\overline{a}_i$. As far as the FSM $M_j$, whose operation have been suspended, this amounts to viewing the symbol $A_i$, so the state in which the simulation should be resumed should be $\delta_j(q, A_i)$, where $q$ was the state in which the simulation of $M_j$ was suspended. (This justifies the definition of a type 1 move).

One can see that the DPDT traverses the derivation tree left to right, top down. It moves down when processing left brackets (type 1), right when processing non bracket symbols (type 3), and up when processing right brackets (type 2). It pushes a symbol on the stack while going down, and pops a symbol while going up. It produces an output
symbol only when it goes down – it outputs the production number \( i \) when reading \( a_i \). After reading a word \( w \in A_1 \), \( M \) will be in its accepting state, and the stack will contain the initial stack symbol only. Reading the end marker will now empty the stack (type 4), terminating the computation successfully. One can see that if the computation terminates successfully, the resulting output is exactly the left parse of the input word.

We demonstrate the DPDT operation on the XHTML introduced in section \[9.2\]. Figure 9.8 illustrates the FSA \((M_i)\) constructed from the DTD of Fig. 2.3.

<table>
<thead>
<tr>
<th>XML ELEMENTS</th>
<th>FSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>start ((M_j))</td>
<td><img src="start_diagram.png" alt="Start State Diagram" /></td>
</tr>
<tr>
<td>html ((M_j))</td>
<td><img src="html_diagram.png" alt="HTML State Diagram" /></td>
</tr>
<tr>
<td>head ((M_j))</td>
<td><img src="head_diagram.png" alt="Head State Diagram" /></td>
</tr>
<tr>
<td>title ((M_j))</td>
<td><img src="title_diagram.png" alt="Title State Diagram" /></td>
</tr>
<tr>
<td>body ((M_j))</td>
<td><img src="body_diagram.png" alt="Body State Diagram" /></td>
</tr>
<tr>
<td>paragraph ((M_j))</td>
<td><img src="paragraph_diagram.png" alt="Paragraph State Diagram" /></td>
</tr>
<tr>
<td>img ((M_j))</td>
<td><img src="img_diagram.png" alt="Image State Diagram" /></td>
</tr>
</tbody>
</table>

Figure 9.8: The DPDT that accepts the XHTML elements in Fig. 9.6 is constructed from the RE. There are seven FSA, one for each of the six non-terminals \((M_1-M_6)\), and \( M_0 \) which is used to start the transcoding. The circles are states of the FSA. Accepting states are denoted by a thick circle, while start states are denoted by an incoming arrow.

Figure [9.9] describes the DPDT operation.

Part IV: Compressed validation of XML documents
## DPDT algorithm

<table>
<thead>
<tr>
<th>Lookahead</th>
<th>Type</th>
<th>State (Q)</th>
<th>Output</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;html&gt;</td>
<td>1</td>
<td>q_{50}</td>
<td>q_{50}</td>
<td></td>
</tr>
<tr>
<td>&lt;head&gt;</td>
<td>1</td>
<td>q_{40}</td>
<td>q_{40}</td>
<td>{t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;/head&gt;</td>
<td>2</td>
<td>q_{50}</td>
<td>q_{50}</td>
<td>[q_{40}, &lt;head&gt;] \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;body&gt;</td>
<td>1</td>
<td>q_{11}</td>
<td>q_{11}</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>3</td>
<td>q_{40}</td>
<td>q_{40}</td>
<td>[q_{40}, &lt;body&gt;] \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&gt;</td>
<td>3</td>
<td>q_{44}</td>
<td>q_{44}</td>
<td>[q_{44}, &lt;body&gt;] \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;p&gt;</td>
<td>1</td>
<td>q_{50}</td>
<td>q_{50}</td>
<td>[q_{50}, &lt;p&gt;] \cdot {q_{50}, &lt;p&gt;} \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&quot;don’t be&quot;</td>
<td>3</td>
<td>q_{40}</td>
<td>q_{40}</td>
<td>[q_{50}, &lt;p&gt;] \cdot {q_{50}, &lt;p&gt;} \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;img&gt;</td>
<td>1</td>
<td>q_{40}</td>
<td>q_{40}</td>
<td>[q_{50}, &lt;img&gt;] \cdot {q_{44}, &lt;p&gt;} \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;/img&gt;</td>
<td>2</td>
<td>q_{44}</td>
<td>q_{44}</td>
<td>[q_{50}, &lt;img&gt;] \cdot {q_{44}, &lt;p&gt;} \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;/p&gt;</td>
<td>2</td>
<td>q_{50}</td>
<td>q_{50}</td>
<td>[q_{50}, &lt;p&gt;] \cdot {q_{44}, &lt;p&gt;} \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;/body&gt;</td>
<td>2</td>
<td>q_{44}</td>
<td>q_{44}</td>
<td>[q_{44}, &lt;body&gt;] \cdot {t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>&lt;/html&gt;</td>
<td>2</td>
<td>q_{12}</td>
<td>q_{12}</td>
<td>{t, &lt;html&gt;} \cdot z_1</td>
</tr>
<tr>
<td>$</td>
<td>4</td>
<td>t_0</td>
<td>t_0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.9: DPDT parsing of the XHTML document which appears in Figure 1.1. The table contains five columns. The **lookahead** lexical symbol, the transition **type** (1-4), the current transcoder **state** and the current **stack** content and the output.

The proof that the DPDT indeed works as expected, will proceed by proving a series of theorems:

The first lemma shows how to partition a derivation tree into its top production and a collection of subtrees.

**Lemma 9.1** Let $w$ be a word in $a_i\Sigma^*\bar{a}_i$ for some $i, 1 \leq i \leq n$. Then $w \in L(A_i)$ if and only if $w$ can be partitioned as $w = a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i$ for some $k \geq 0$, such that

- for all $1 \leq j \leq k + 1, x_j \in \Sigma'$
- For all $1 \leq j \leq k, y_j \in L(A_{ij})$ for some $A_{ij} \in N$, and
- $\hat{w} = x_1 A_{i1} x_2 A_{i2} \ldots x_k A_{ik} x_{k+1} \in R_i$.

Furthermore, $\hat{w}$ is uniquely determined from $w$.

Part IV: Compressed validation of XML documents
**Lemma 9.2** For all \( i, 1 \leq i \leq n, x \in \Sigma^*, Z \in \Gamma, \)

1. If there exists \( z, \) such that \( xz \in R_i, \) then \((q_{0i}, x, Z, \epsilon) \vdash_3 (\delta_i(q_{0i}, x), \epsilon, Z, \epsilon)\)

2. If \((q_{0i}, x, Z, \epsilon) \vdash_3 (p, \epsilon, \gamma, v)\) for some \( p \in Q, \gamma \in \Gamma^*, \) and \( v \in \Delta^* \) then \( p = \delta_i(q_{0i}, x), \gamma = Z, v = \epsilon \) and the derivation uses type 3 moves only.

**Proof** Each direction may be proved by a straightforward induction on the length of \( x, \) omitted.

We can now show that each word derived from a non-terminal induces a certain computation of \( M. \)

**Lemma 9.3** For all \( 1 \leq i \leq n, q \in Q, Z \in \Gamma \) and \( w \in L(A_i) \)

\[
(q, w, Z, \epsilon) \vdash_3 (\delta_i(q, A_i), \epsilon, Z, \pi(A_i \Rightarrow_L w))
\]

where \( q \in Q_i. \)

**Proof** We will prove the lemma by induction on the height of the derivation tree.

**Basis:** The height of the derivation tree is 1. Then \( w \in L(A_i) \) implies that \( w = a_i x_1 \bar{a}_i, x_1 \in \Sigma^*, \bar{w} = x_1 \in R_i \) and \( A_i \rightarrow a_i R_i \bar{a}_i \rightarrow P. \) By construction of \( M, \) for all \( l, 1 \leq l \leq n, q \in Q_i \)

\[
(q, a_i x_1 \bar{a}_i, Z, \epsilon) \vdash_1 (q_{0i}, x_1 \bar{a}_i, [\delta_i(q, A_i), a_i] Z, i) \vdash_3
\]

\[
(\delta_i(q_{0i}, x_1), \bar{a}_i, [\delta_i(q, A_i), a_i] Z, i) \vdash_2 (\delta_i(q, A_i), \epsilon, Z, i)
\]

Part [IV] Compressed validation of XML documents
We used Lemma 9.2 for the middle part of the computation (type 3 moves). The last step (type 2 move) is valid since $x_1 \in R_i$ implies that $\delta_i(q_0, x_1) \in F_i$.

To complete the basis, we just note that $\bar{i} = \pi(A_i \Rightarrow L, a_i R_i \bar{a}_i)$.

**Induction step:** Assume the lemma holds for all $w'$ and all $i'$ such that the height of the derivation tree for $A_{i'} \Rightarrow_L^{*} w'$ is at most $h$ for some $h > 0$. Now assume $A_i \Rightarrow_L^{*} w$ with a derivation tree of height $h + 1$. By Lemma 9.1 the derivation can be rewritten as

$$A_i \Rightarrow L, a_i x_1 A_i x_2 A_i \ldots x_k A_i x_{k+1} \bar{a}_i = \Rightarrow_L^{*} a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i$$

where for each $j, 1 \leq j \leq k + 1, A_i \Rightarrow_L^{*} y_j$. Furthermore, the derivation trees of all $A_{i_j} \Rightarrow_L^{*} y_j$, have height at most $h$, so we can use the induction hypothesis for each of them.

In order to complete the proof of the induction step, we need the following claim.

**Lemma 9.4** Let $w = a_i x_1 y_1 x_2 y_2 \ldots x_m y_m x_{m+1}$, such that $x_j \in \Sigma^*$ for $1 \leq j \leq m + 1$, $A_{i_j} \Rightarrow_L^{*} y_j$, for all $1 \leq j \leq m$, and assume that Lemma 9.3 holds for these derivations. Let $\bar{w} = x_1 A_i x_2 A_i \ldots x_m A_m x_{m+1}$, and suppose there exists $z$ such that $\bar{w} z \in R_i$.

Then for all $Z \in \Gamma$

$$(q, w, Z, \epsilon) \vdash^{*} (\delta_i(q_0, \bar{w}), \epsilon, [\delta_i(q, A_i), a_i] Z, i\pi(A_i \Rightarrow_L^{*} y_1) \pi(A_{i_2} \Rightarrow_L^{*} y_2) \ldots \pi(A_{i_m} \Rightarrow_L^{*} y_m))$$

**Proof:** The proof will be by induction on $m$.

**Basis:** $m = 0$. Then $w = a_1 x_1, \bar{w} = x_1 \in \Sigma^*$ and there exists $z$ such that $x_1 z \in R_i$.

Then by construction, for any $q \in Q, Z \in \Gamma, (q, a_i x_1, Z, \epsilon) \vdash^{1} (q_0, x_1, [\delta_i(q, A_i), a_i] Z, i)$ where $q \in Q_i$. Further, by Lemma 9.2 we get

$$(q_0, x_1, [\delta_i(q, A_i), a_i] Z, i) \vdash^{2} (\delta_i(q_0, x_1), \epsilon, [\delta_i(q, A_i), a_i] Z, i)$$

which completes the basis.

**Induction step:** Suppose the claim holds for all $m < m_0$, for some $m_0 > 0$. Now let $m = m_0$. Let $w = a_i x_1 y_1 x_2 y_2 \ldots x_m y_m x_{m+1}$, such that $x_j \in \Sigma^*$ for $1 \leq j \leq m + 1$, $A_{i_j} \Rightarrow_L^{*} y_j$, for all $1 \leq j \leq m$, and assume that Lemma 9.3 holds for these derivations. Suppose there exists $z$, such that $\bar{w} z \in R_i$ where $\bar{w} = x_1 A_i x_2 A_i \ldots x_m A_m x_{m+1}$. Let $w_1 = a_i x_1 y_1 x_2 y_2 \ldots x_{m-1} y_{m-1} x_m$. By the induction hypothesis for all $Z \in \Gamma$

$$(q, w_1, Z, \epsilon) \vdash^{*} (\delta_i(q_0, \bar{w}_1), \epsilon, [\delta_i(q, A_i), a_i] Z, i) \pi(A_{i_2} \Rightarrow_L^{*} y_2) \ldots \pi(A_{i_{m-1}} \Rightarrow_L^{*} y_{m-1}))$$

Since $w = w_1 y_m x_{m+1}$, we can write

$$(q, w_1 y_m x_{m+1}, Z, \epsilon) \vdash^{*} (q, w_1 y_m x_{m+1}, Z, \epsilon)$$

Part IV Compressed validation of XML documents
We now consider derivation \( A_{i_m} \Rightarrow \delta\), and use Lemma 9.3 to extend \( M \)'s computation as follows:

\[
(\delta(q_0, w_1)(y_m x_{m+1}, [\delta(q, A_i), a_i] Z, i\pi(A_{i_1} \Rightarrow \delta) y_1) \ldots \pi(A_{i_{m-1}} \Rightarrow \delta) y_{m-1}))
\]

We now use Lemma 9.2 and apply the equation \( \delta(q_0, u_1, w_1) = \delta(q, u_1 u_2) \) twice to extend the computation further.

\[
(\delta(q_0, w_1 A_i, x_{m+1}, [\delta(q, A_i), a_i] Z, i\pi(A_{i_1} \Rightarrow \delta) y_1) \ldots \pi(A_{i_{m-1}} \Rightarrow \delta) y_{m-1}))
\]

This establishes the entire computation, and completes the proof of the induction step. Thus, Lemma 9.3 has been established.

We can now complete the induction step in the proof of Lemma 9.3. Consider again the word \( w = a_i x_1 A_{i_1} x_2 A_{i_2} \ldots x_k A_{i_k} x_{k+1} \tilde{a}_i \) and the derivation

\[
A_i \Rightarrow \delta A_i x_1 A_{i_1} x_2 A_{i_2} \ldots x_k A_{i_k} x_{k+1} \tilde{a}_i \Rightarrow \delta a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \tilde{a}_i
\]

where for each \( j, 1 \leq j \leq k + 1, A_{i_j} \Rightarrow \delta y_j. \) Let \( w = w' \tilde{a}_i. \) Then the conditions of Lemma 9.4 apply to \( w' = a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1}, \) (with \( z = \epsilon \)) and from the lemma we get the computation

\[
(q, w, Z, \epsilon) \Rightarrow (\delta(q_0, w), \tilde{a}_i, [\delta(q, A_i), a_i] Z, i\pi(A_{i_1} \Rightarrow \delta) y_1) \pi(A_{i_2} \Rightarrow \delta) y_2) \ldots \pi(A_{i_m} \Rightarrow \delta) y_m))
\]

By definition, the leftmost parse of a derivation is the production used in its first step, followed by the leftmost parses of the subtrees from left to right. Hence

\[
i\pi(A_{i_1} \Rightarrow \delta) y_1) \pi(A_{i_2} \Rightarrow \delta) y_2) \ldots \pi(A_{i_m} \Rightarrow \delta) y_m)) = \pi(A_{i_1} \Rightarrow \delta) w)
\]

Also, since \( w \in R_i, \delta(q_0, w) \in F_i, \) the computation may be extended by

\[
(\delta(q_0, w), \tilde{a}_i, [\delta(q, A_i), a_i] Z, \pi(A_{i_1} \Rightarrow \delta) w)) \Rightarrow (\delta(q, A_i), \epsilon, Z, \pi(A_{i_1} \Rightarrow \delta) w))
\]

This completes the induction step and the entire proof of Lemma 9.3.

The next Lemma is the converse of Lemma 9.3.

Part IV: Compressed validation of XML documents
Lemma 9.5 If \((q, w, Z, \epsilon) \vdash^* (p, \epsilon, Z, v)\) for some \(q, p \in Q, Z \in \Gamma, \) and \(v \in \Delta^*\) so that all intermediate configurations in this computation have stack height larger than 1, then there exist \(i\) and \(l\), such that \(1 \leq i \leq n, 0 \leq l \leq n, w \in L(A_i), q \in Q_l, p = \delta_i(q, A_i), \) and \(v = \pi(A_i \Rightarrow_L w)\). \(\square\)

PROOF Since all intermediate configurations in this computation have stack height larger than 1, it follows that the first step must be a type 1 move, and the last step a type 2 move. So \(w = a_i x_i \bar{a}_i\). Let \(q \in Q_l\), for some \(0 \leq l \leq n\), and let \(p = \delta_i(q, A_i)\).

We proceed by an induction on the maximal stack height during the computation.

**Basis:** The maximal stack height is 2, so the computation can be written as

\[
(q, a_i x_1 \bar{a}_i, Z, \epsilon) \vdash^1 (q_0i, x_1 a_i, [p, a_i]Z, i) \vdash^3 (p_1', \bar{a}_i, [p, a_i]Z, i) \vdash^2 (p, \epsilon, Z, i)
\]

where \(p_1' = \delta_i(q_0i, x_1)\) (by Lemma 9.2), and \(p_1' \in F_i\) (to allow for the type 2 move). Clearly also \(i = i'\). It follows that \(x_1 \in R_i\), so that \(w = a_i x_1 a_i \in L(A_i)\) with \(\pi(A_i \Rightarrow_L w) = i\) (a single step derivation). This completes the basis.

**Induction step:** Assume the lemma holds for computations of maximal stack height less than \(h\), for some \(h > 2\). Now consider a computation with maximal stack height \(h\).

Since the height of the stack can be changed by at most 1 in each step, we can identify the longest subcomputations that occur at a fixed stack height of 2, and decompose the computation as follows, using the fact that moves that do not change the stack height are of type 3, which do not change the content of the stack and do not produce output. As in the basis, the left and right bracket symbols must match, so one can write \(w = a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i\) and decompose the computation as

\[
(q, a_i x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i, Z, \epsilon) \vdash^1 (p_1, x_1 y_1 x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i) \vdash^3 (p_2, x_2 y_2 \ldots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i) \vdash^2 (p, \epsilon, Z, i)
\]

where intermediate configuration in the subcomputations on the words \(y_j\) have stack height larger than 2, so they are not dependent on the actual stack symbols. Hence we can say that for all \(1 \leq j \leq k\) and \(Z' \in \Gamma\) \((p_j', y_j, Z', \epsilon) \vdash^* (p_{j+1}, \epsilon, Z', v_j)\), where the maximal stack height of these computations is less than \(h\). The type 1 move (the first step in the derivation) implies that \(p_1 = q_0i\).

Applying the induction hypothesis to the computations \((p_j', y_j, Z', \epsilon) \vdash^* (p_{j+1}, \epsilon, Z', v_j)\) for all \(1 \leq j \leq k\), we get that \(y_j \in L(A_i)\), \(p_j' \in Q_{i_j}\), \(p_{j+1} = \delta_{i_j}(p_j', A_{i_j}), v_j = \pi(A_{i_j} \Rightarrow_L y_j)\). Looking at the type 3 subcomputations, we get from Lemma 9.2 that

Part IV Compressed validation of XML documents
\[ p'_j = \delta_i(p_j, x_j) \] for all \( 1 \leq j \leq k \). In addition, since each of the type 3 subcomputations is followed by a type 1 move (the computations on \( y_j \) start by increasing the size of the stack), we must have \( p'_j \in F_{i_j} \).

By combining all the above, we can see that all \( l_j \) are identical, and equal to \( l \).

For all \( 1 \leq j \leq k \), \( v_j = \pi(A_{i_j} \Rightarrow^*_L y_j) \). Hence \( iv_1v_2 \ldots v_k = i\pi(A_{i_1} \Rightarrow^*_L y_{i_1}) \ldots \pi(A_{i_k} \Rightarrow^*_L y_{i_k}) = \pi(A_i \Rightarrow^*_L w) \).

**Lemma 9.6** Given a D-grammar, one can construct a DPDT \( M \) that works as follows. For each \( w \in \Sigma^* \), \( M \) accepts \( w \) if and only if \( w \in L(A_1) \). Furthermore, if \( w \in L(A_1) \), then \( M \) produces as output the left parse of \( w \). \( M \) has no \( \epsilon \) moves, so its running time is linear in the length of \( w \).

**Proof** Follows from Lemma 9.3 and Lemma 9.5.

If \( w \in L(A_1) \) then by Lemma 9.3 \((q_0, w, Z_0, \epsilon) \vdash^* (f_0, \epsilon, Z_0, \pi(A_i \Rightarrow^*_L w))\), since \( \delta_0(q_0, A_1) = f_0 \). Adding the end marker, and a type 4 move we get \((q_0, \$, Z_0, \pi(A_i \Rightarrow^*_L w)) \vdash (f_0, \epsilon, \epsilon, \pi(A_i \Rightarrow^*_L w))\).

Conversely, if \( w \$ \) is accepted by \( M \), then its computation must be of the form

\[ (q_0, w\$, Z_0, \epsilon) \vdash^* (f_0, \$, Z_0, v) \vdash^4 (f_0, \epsilon, \epsilon, v). \]

We can now use Lemma 9.5, noting that \( q_0 \in Q_0 \), \( f_0 = \delta_0(q_0, A_1) \) and \( \delta_0 \) is undefined elsewhere, to conclude that \( w \in L(A_1) \), and \( v = \pi(A_i \Rightarrow^*_L w) \).

The linear running time follows from the construction of \( M \) as \( \epsilon \) free.

We can therefore construct a parser generator, that constructs the parsing tables (a variation of the DPDT shown above) while reading the dictionary portion of the XML file. Then, the parser is applied to the rest of the XML file, producing the leftmost parse as explained (see Section 9.3.5).

The size of the parser (the number of states) may, in the worst case, be exponential in the size of the original grammar, because the construction involves conversion of non-deterministic FSA to deterministic FSA. However, in practice, we can expect, the parser is not much larger than the original grammar. The running time of the parser’s generator may therefore be exponential in the worst case, but it is linear in practice. In any event, the running time of the parsing is linear in the size of the input.

**9.3.4 XML lexer**

The flow 3a (XML tokenizer) in Fig. 9.5 is described now. The XML lexical analyzer (lexer) inherits its symbols table from the RE lexer. The table maps symbols to XML to-
kens. The XML lexer reads XML tokens from an XML source. It retrieves its matched lexical symbol from the symbol table and sends it to the XML parser. The lexer uses two types of predefined symbols: Free text element is wrapped with the PCDATA lexical symbol, and free text attribute value is wrapped with the CDATA lexical symbol. Figure 9.10 illustrates the XML lexer FSM. It has five states to determine which string is currently tokenized: start tag or end tag or attribute or free text attribute value or selection list attribute value.

![XML lexer FSM diagram](image)

Figure 9.10: The XML lexer FSM

The XML lexer also supplies a reverse functionality. It receives a lexical symbol from the decoder and writes the matched XML token to the output XML source. In order to represent the token correctly it must know its XML entity type. The XML entity type of each symbol is inherited from the RE lexer as part of the symbol table. The following XML representation occurs in the decoding process:

attribute: attribute =

start element: <element>

end element: </element>

attribute value: ”value”
9.3.5 The DPDT parser

We describe now the flow of 3b (XML parser) in Fig. 9.5. The DPDT, generated as described in section 9.3.3, is applied to the stream of XML tokens, producing the leftmost parse as explained. Since the DPDT has no \( \epsilon \) moves, it works in linear time. (Its operation is similar to the LL parser operation - working top down with no backtracking).

As noted in section 9.3.3, the output of the DPDT is the left parse of the input word, namely, a list of the production numbers used in the parse tree, listed top down, left to right.

9.3.6 DPDT guided encoding

The DPDT-L encoding method multiplexes the content model encoding and the structure model encoding using the same PMM model. The structure model symbols are the DPDT finite output alphabet symbols \( \Delta \). The DPDT-L algorithm executes the DPDT on the input XML document and encode the output symbols \( a \in \Delta \). Its encoding is locally guided by the DPDT. Section 9.2.1.1 describes local LL-guided-parser encoding that encodes the relevant production rules. Relevant production rules can derive the non-terminal at the top of the stack.

The DPDT guided encoding, encodes the output symbols instead of production rules. Local DPDT guided encoding, encodes the DPDT output symbols that are relevant for the current DPDT state. The relevant DPDT output symbols are determined by the DPDT transition function. Each transition type assigns a relevancy type symbol as follows:

**Type 1:** For all \( 1 \leq i \leq n, 0 \leq j \leq n \) and \( q \in Q_j \), if \( \delta_j(q, A_i) \) is defined, then \( a_i \) is relevant to \( q \) (left bracket).

**Type 2:** For all \( 1 \leq i \leq n \) and \( q \in F_i \), \( \bar{a}_i \) is relevant to \( q \) (right bracket).

**Type 3:** For all \( 0 \leq i \leq n, q \in Q_i, a \in \Sigma' \), if \( \delta_i(q, a) \) is defined, then \( a \) is relevant to \( q \) (non-bracket symbol).

A single relevant symbol is ignored by the encoding algorithm. In the XHTML example, the relevant symbols are shown in Fig. 9.11. It is constructed from the REs in Fig. 9.8.

Part IV Compressed validation of XML documents
Figure 9.11: Table of XHTML relevant symbols which are constructed from the transitions in Fig. 9.8. The list of relevant symbols is detailed for each state. The square brackets to the right of each symbol mark its relevancy-type.

When the XHTML example in Fig. 9.9 is encoded, we receive the following local encoded symbols:

- •, "don’t be", <img, ..., - , </p>, </body>, -

The character ‘-‘ marks deterministic lexical symbols that are ignored by the encoder. The ‘...’ marks the places in the example where the parsing details are not shown.

Implementation of a local DPDT encoding by PPMD+ is straightforward. PPMD+ implementation uses an exclusion bit mask that refers to symbols that are excluded during the symbol encoding process (see section 9.2.4). Normally, the PPMD+ initializes an empty exclusion mask for every encoded symbol. In local DPDT encoding, when a symbol is encoded, we mark the non-relevant symbols in the exclusion mask. Thus, the PPMD+ encoder ignores the non-relevant symbols and only the relevant symbols are encoded.

The content encoding model is plain. It has 255 ASCII symbols. All character symbols are relevant for encoding of an ASCII symbol.

### 9.4 XML compression results

In this section, we evaluate the performance of the encoding algorithm that was introduced in section 9.3. In section 9.4.1, the XML benchmarks documents (XML corpus)
used in the experiment are described. The performance of the compression algorithm (DPDT-L) is compared in section 9.4.2 to other XML compression methods.

9.4.1 The source data

Our XML corpus contains four available XML benchmarks files [93, 120, 125, 57]. The different benchmarks are XML documents with a range of distinct structural characteristics. The XML corpus has different sizes: large (~100MB), medium (~10MB) and small (~1MB).

Table 9.1 provides structural information of the XML corpus.

<table>
<thead>
<tr>
<th>Document</th>
<th>Structure (%)</th>
<th>Average depth</th>
<th>Average freedom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xmark</td>
<td>28</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>007</td>
<td>78</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Michigan</td>
<td>17</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>XMach-1</td>
<td>14</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9.1: Characteristics of the benchmark files that are used by the DPDT-L compression algorithm. Column 2 (Structure) is the number (in percentage) of characters in the dataset that are XML tags. The average depth of the stack in the parser is given in column 3 (Average depth). The average number of relevant symbols (Average freedom) is given in column 4.

Table 9.1 contains an additional statistics that was gathered by the DPDT-L algorithm during the parsing of the XML documents such as the average XML tree depth (Average depth). Relevant symbols were introduced in section 9.3.6. Relevant symbols are symbols that are accepted by the current DPDT state. The average number of relevant symbols (Average freedom) is given in column 4.

The XML corpus contains four documents. The characteristics of these documents (datasets) are:

**XMark** benchmark document [125]. The XMark data generator produces XML documents modeling an auction website, which is a typical e-commerce application.

**007** benchmark document [93]. The x007 data generator produces structure centric synthetic XML documents with a simple structure.
Michigan benchmark document [120]. The Michigan data generator produces content centric XML documents with a highly redundant content and a simple structure.

XMach-1 benchmark document [57]. The XMach-1 data generator produces synthetic XML documents. It randomly creates its content from a given set of words.

9.4.2 Compression ratios

The current XML compression methods to which we compare the performance of the DPDT-L were described in section 9.2.3. We compare the presented DPDT local encoding schemes (DPDT-L) to existing methods that use the same underlying PPM encoder. DTDPPM [81] and XAUST [73] are DTD aware encoders. XMLPPM [40] and SCMPPM [13] ignore the DTD.

Table 9.2 summarizes the bytes per character (bpc) computed by different compression methods on small sizes (≈1MB) datasets.

<table>
<thead>
<tr>
<th>File</th>
<th>Size (KB)</th>
<th>DPDT-L</th>
<th>XAUST</th>
<th>DTDPPM</th>
<th>XMLPPM</th>
<th>SCMPPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xmark</td>
<td>1155</td>
<td>1.63</td>
<td>1.75</td>
<td>1.87</td>
<td>1.61</td>
<td>1.68</td>
</tr>
<tr>
<td>007</td>
<td>1085</td>
<td>0.10</td>
<td>0.18</td>
<td>0.20</td>
<td>0.18</td>
<td>0.14</td>
</tr>
<tr>
<td>Michigan</td>
<td>909</td>
<td>0.61</td>
<td>0.63</td>
<td>0.60</td>
<td>0.61</td>
<td>0.54</td>
</tr>
<tr>
<td>XMach-1</td>
<td>1091</td>
<td>2.11</td>
<td>2.12</td>
<td>2.28</td>
<td>2.11</td>
<td>2.09</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>1.143</td>
<td>1.204</td>
<td>1.275</td>
<td>1.160</td>
<td>1.145</td>
</tr>
</tbody>
</table>

Table 9.2: bpc from different compression methods operated on small datasets (≈1MB). The average is weighted.

Table 9.3 summarizes the bpc from different compression methods applied to medium sizes (≈10MB) datasets.

Part IV Compressed validation of XML documents
Table 9.3: bpc from different compression methods operated on medium sizes (~10MB) datasets

Table 9.4 summarizes the bpc from different compression methods applied to large sizes (~100MB) datasets.

Table 9.4: bpc from different compression methods applied to large sizes (~100MB) datasets

Table 9.5 summarizes the improvements in % using DPDT-L algorithm in comparison with other compression methods. This is achieved by taking the weighted average, which is related to the file size, of the compression ratios.

Table 9.5: Improvements in % of the DPDT-L against different compression methods.

The results in Table 9.5 clearly show that local DPDT encoded guided parser (DPDT-L algorithm) outperforms, on average, the rest. Both DTD aware compressors achieve
lower CR on every file in the corpus. DPDT-L improves XAUST CR by 6% on average. 25% on the average is the improvement over the performance of the DTDPPM algorithm.

The results are less evident on DTD unaware compressors. The content of the XML files in the corpus can be roughly divided into: simple and complex. Simple content is highly redundant (Michigan) or its content portion in the file is redundant in comparison to the structure portion of the file (007). Files with simple content are best compressed by SCMPPM. However, files with complex content (XMach-1 and XMark) are best compressed by MHM based encoders such as XMLPPM and DPDT-L.

On the average, XMLPPM and SCMPPM achieve similar CR. The advantage and disadvantage of each compression method are balanced.

MHM based compression methods such as DPDT-L and XMLPPM, converge to similar CR on complex content. But DPDT-L CR is better (up to 50%) on files with simple content files. On the average, this advantage makes DPDT-L CR better than XMLPPM and SCMPPM methodologies.

The results show that medium size (~10MB) files achieve better CR improvement when DPDT-L is used.

9.5 Future work

This research can be extended into the following directions:

- No special attention was paid to efficient and standard implementation of the DPDT-L XML compression algorithm. We plan to replace the propriety XML lexical analyzer with a fast and standard XML parser. The PPM model should be replaced by a dynamic allocation mechanism in order to reduce memory utilization.

- From understanding the advantages of SCM based encoders, it is useful to combine the DPDT-L encoding structure with the SCM content encoding.
Part V

XML Schema extraction
Chapter 10

Two Levels State Machine algorithm

10.1 Introduction

An XML schema can play an important role in XML that manages queries and storage. Automatic schema extraction can be used in many applications. It can be useful for WEB developers that need to write an XML schema for WEB sites. Software developers can use XML Schema for automatic generation of source code for WEB applications while reducing possible coding mistakes.

XML files are often distributed without any XML Schema attached to them. This raises the problem how to extract schematic information from XML documents. A schema should tightly represents the data while being compact. These two requirements conflict each other. This makes the schema extraction a difficult task.

We propose a new type of a state machine called two layer state machine (TLSM). TLSM accepts a grammar of ordered trees. TLSM holds in each state a FSM that describes the internal order of the children of each node in the ordered tree. A new algorithm, called XTLSM, which extracts TLSM from a given XML sample, is presented. The output from TLSM is tight and compact. Finally, a method to create a compact XML Schema from TLSM is also introduced. The XTLSM algorithm processes XML documents and creates an XML schema. In order to create an XML schema, it converts the XML document into several data modules. In each stage, the algorithm produces the next data module from previous data modules. In the last stage, it produces the XML Schema.

Stage 1 - creation of a prefix tree acceptor from an XML document: An XML Doc-
Two Levels State Machine algorithm

Document is presented as an ordered labeled tree (OLT). Repetitions of elements and sub-trees are merged in this stage. A prefix tree acceptor (PTA) is constructed. The size of a PTA is usually relatively small since in general an XML document has high number of repetitions.

Stage 2 - the PTA is converted into a FSM: The next data module, which the algorithm creates, is a FSM. The PTA is inspected. Similar sub-trees are marked to be merged. Merging similar sub-trees of the PTA converts it into a FSM. The new FSM does not have a tree structure any more that may include infinite loops.

Stage 3 - the inner FSM is added to each outer state: The FSM from Stage 2 is used as a grammar that accepts all the paths in the original XML document. We need a better data module that can be used as a grammar for an OLT.

In this stage, each state in the FSM turns into an inner FSM. This new property enables the new data module to distinguish between an OLT according to the order of the sibling nodes in the OLT. In addition, how to validate an OLT using a two layer state machine (TLSM) is demonstrated.

Stage 4 - each outer state is converted into an XML schema element: In this final optional stage, TLSM is converted into an XML schema. Each inner FSM is chopped vertically and horizontally into smaller FSMs until chopping is impossible. Each vertical chop creates an XSD sequence element. Each horizontal chop creates an XSD choice element. The parts of the FSM, which cannot be chopped anymore, are converted into XSD elements using a brute force approach; The new grammar, which the XSD provides, describes a language for an XML document. This language may be bigger than the language that was described by the TLSM.

The chapter has the following structure. The related work and the data models that are used throughout the chapter are described in section 10.2. The XTLSM algorithm is given in section 10.4. Experimental results are given in section 10.5.

10.2 Preliminaries

This section defines preliminaries for the XTLSM algorithm: Section 10.2.1 describes a Prefix Tree Acceptor (PTA), which is a known construction in grammatical inference.
algorithms. Section 2.1.1 describes the XML schema language that is inferred by the \textit{XLTSM} algorithm.

\section{10.2.1 Prefix Tree Acceptor (PTA)}

\textit{Prefix Tree Acceptor (PTA)} is a tree structured FSM. Any state in a PTA has a single parent. The PTA is a known construction in grammatical inference. It is introduced by ALERGIA \cite{38}, which is an algorithm for learning stochastic regular grammars by means of a state merging method. In this chapter, the PTA concept is simplified. The PTA concept here resembles DataGuide \cite{119}. The \textit{XTLSM} in section 10.4 produces a PTA from an OLT that represents an XML document.

\textbf{Definition 10.1} Assume $T = (V^T, E^T, label^T)$ is an OLT. $PTA_T = (\Sigma, S, s_0, \delta, F)$ is the PTA of $T$ if $T$ and $PTA_T$ have the same exact language $L_{\text{leaf}}(T) = L(PTA_T)$ (therefore, $L(PTA_T)$ is finite) and $PTA_T$ has a tree structure $\delta(s_i, \alpha) = \delta(s_j, \alpha) \implies i = j$.

\textbf{Example 10.1} The PTA in Fig. 10.1 is the $PTA_T$ of $T$ which is illustrated in Fig. 2.4. $L(PTA_T) = L_{\text{leaf}}(T) = \{abce, abcfaa, abd, abde, abd'a'aaa\}$.
Figure 10.1: Illustration of a PTA. Single circles denote non-accepting states. Double circles denote accepting states. The state-transition function is denoted by arrows - source and destination states are the source and destination circles connected to the arrow. A label, which is attached to each arrow, is its transition symbol. The start state is the destination circle of the arrow that has no source circle.

10.2.2 XSD Π Notation

Since XML Schema standard is complex and large, we take only a subset of this standard to define a data module that fits our needs.

**Definition 10.2 (XSD Π notion)** Let $X = (\Pi, \pi_0 \in \Pi)$ be a XSD. $\Pi = \{\pi_0, \pi_1, \pi_2, \ldots\}$ is a non-empty set of type definitions. Each type definition $\pi = (index^\pi, \omega^\pi)$ is a pair of index and $\omega$ expression. The expression $\omega$ is defined recursively using one of the following forms:
empty expression:  λ empty type definition means that such an XML element does not have any sub-structures;

reference expression:  \((\alpha \rightarrow index)^*\) is a reference for another type definition.  \(\alpha\) is an element name and \(index\) is the index of the other type.

sequence expression:  \((\omega_1, \omega_2, \ldots)^*\) is a sequence of expressions. It means that all the sub-structures must appear in this order under the current element.

choice expression:  \((\omega_1|\omega_2|\ldots)^*\) is a choice between expressions. It means that one of the sub-structures must appear under the current element.

\(*\) : The quantity of each element can be one of the following:

- \(0 - 1\): zero or one appearance. (minOccuers="0" maxOccuers="1");
- \(0 - \infty\): any number of appearances. (minOccuers="0" maxOccuers="unbounded");
- \(1\): one appearance. (minOccuers="1" maxOccuers="1");
- \(1 - \infty\): one or more appearances. (minOccuers="1" maxOccuers="unbounded").

If not specified, the quantity will be 1. In the Π notation module, the quantity of each element can be only one of the four choices defined here. In real XSD definition, minOccuers and maxOccuers can also be any integer. As a result, the Π notation module will not be able to describe the exact quantity of each XML element, but only whether the element is mandatory, single or multiple. But such a module will be simpler to process.

\(\pi_0\) is the root element of the XML document that the schema describes. Since XML documents allow only a single root element, it must be in the form (\(\alpha \rightarrow \pi\))^1.

\[
\begin{align*}
\pi_0 &: a \rightarrow \pi_1 \\
\pi_1 &: (b \rightarrow \pi_2)^{1-\infty} \\
\pi_2 &: (e \rightarrow \pi_3|d \rightarrow \pi_3) \\
\pi_3 &: (e \rightarrow \pi_4, f \rightarrow \pi_5)^{0-1} \\
\pi_4 &: \lambda \\
\pi_5 &: (a \rightarrow \pi_5)^{0-\infty}
\end{align*}
\]

line 1 in Fig. 2.2
line 2-4 in Fig. 2.2
lines 5-10 in Fig. 2.2
lines 11-16 in Fig. 2.2
lines 17-18 in Fig. 2.2
lines 19-21 in Fig. 2.2

Figure 10.2: Example of XSD written in Π notation

Example 10.2 Figure 10.2 demonstrates the XSD Π notation. The elements in this example are the same elements as in Fig. 2.2. Each \(\pi\) expression in this example matches to a single element definition in Fig. 2.2.

Part XML Schema extraction
10.3 Related work: XML schema extraction

The section defines related XML schema inference methods. The existing XML schema inference methods are:

Schema inference: Schema for semi-structured data are defined in [35, 60]. The inference of semi-structured data is addressed in [119, 108]. Derivation of a graph summary structure, called Dataguide, is described in [119]. This data structure contains all the paths in the DB. XSD-like schemas cannot be derived from it.

The schema’s data structure, which is based on a labeled directed graph, is described in [108]. Yet, XSD-like schemas cannot be derived from this data structure type. A schema extraction method, which is based on the powerful model of extended context-free grammars, is introduced in [41].

DTD inference: Several approaches for DTD inference were proposed. Xtract [66, 101] uses Minimum Description Length (MDL) principle to select REs for each element name. The methods in [28] generate a finite automaton for each element. Each automaton is rewritten into a RE. Several approaches to generate probabilistic string automata representing RE are proposed in [123]. There are no methods to transform these into corresponding REs.

XSD inference: Scalable derivation of schema’s in XML syntax, which does not incorporate expressiveness beyond DTDs, is given in [75]. A complete inference algorithm, which can infer XSD from a corpus of XML documents, is provided in [29].

Learning of tree automata: A framework of function distinguishably to learn tree automata for ranked trees is showed in [61]. It also presents a generic regular tree languages inference algorithm.

A learnable subclass of regular unranked tree languages, called the \((k, l)\)-contextual tree languages, is defined in [116]. It also describes the use of this subclass to induce wrappers for information extraction from structured documents. The techniques in [116] are geared toward inferring queries not XSDs.
10.4 The XTLSM algorithm

10.4.1 Two Layers State Machine (TLSM)

To distinguish between two OLTs, we need a more expressive grammar than the grammar that FSM provides. We introduce the two layer state machine (TLSM). This type of state machine describes the order of the children in a OLT. TLSM extends a FSM by adding inner FSMs. Each inner FSM is attached to a state in the outer layer. Both layers of the TLSM are deterministic and finite.

**Definition 10.3** $B = (Q, q_0 \in Q, \delta, F, \Sigma)$ is a TLSM if $\Sigma$ is a finite alphabet, $Q = \{q_0, q_1, q_2, \ldots\}$ is a non-empty set of outer states (these states are the outer layer states), $q_0$ is the start state of the outer layer, $\bot$ is an error state, $\delta = (Q \times \Sigma) \mapsto Q \cup \{\bot\}$ is the outer layer state-transition function and $F \subset Q$ is a non-empty set of final (accept) states of the outer layer. Each outer state $q = (\Sigma^q, S^q, s^q_0, \delta^q, F^q)$ is an inner FSM. The following notation is used for each inner FSM: the alphabet of the inner FSM is the same ($\Sigma^q = \Sigma$) as the alphabet of the outer FSM, $S^q = \{s^q_0, s^q_1, s^q_2, \ldots\}$ is a set of states (these states are the states of the inner FSM $q$), $s^q_0$ is the start state of the inner FSM $q$, $\delta^q = (S^q \times \Sigma^q) \mapsto S^q \cup \{\bot\}$ is a state transition function of the inner FSM $q$, $\delta^q(s, \alpha) = \bot$ means that there is no transition from state $p$ using the symbol $\alpha$ and $F^q \subset S^q$ is a non-empty set of final (accepting) states of the inner FSM $q$. 

Part [XML Schema extraction]
Figure 10.3: Illustration of a TLSM. The outer circles are the outer states. The final outer states are denoted by double circles. The arrows between the outer circles represent the outer layer state-transition function. The inner circles are the states of the inner FSMs. Each inner state is drawn inside the circle that represents its outer state. The final inner states are also drawn as double circles. The inner start states $q_0$ are not shown. Since the start inner state $s_0^q$ is final if and only if $q$ is final (i.e. $q \in F \iff s_0^q \in F^q$), then, we use the outer circle to represent both the outer state $q$ and the start inner state $s_0^q$.

Example 10.3 Figure [10.3] illustrates the TLSM $B$. The outer layer of $B$ is the same as the FSM in Fig. 2.6. $B = (Q, q_0 \in Q, \delta, F, \Sigma)$ where $\Sigma = \{a, b, c, d, e, f\}, Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \delta =$

<table>
<thead>
<tr>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>b</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_3$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>c</td>
<td>$q_3$</td>
<td>$q_3$</td>
<td>$q_3$</td>
<td>$q_4$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>d</td>
<td>$q_4$</td>
<td>$q_4$</td>
<td>$q_4$</td>
<td>$q_4$</td>
<td>$q_5$</td>
</tr>
<tr>
<td>e</td>
<td>$q_5$</td>
<td>$q_5$</td>
<td>$q_5$</td>
<td>$q_5$</td>
<td>$q_5$</td>
</tr>
</tbody>
</table>

and $F = \{q_3, q_4, q_5\}$. The inner states are: $S^{q_0} = \{s_0^{q_0}, s_1^{q_0}\}, S^{q_1} = \{s_0^{q_1}, s_1^{q_1}\}, S^{q_2} = \{s_0^{q_2}, s_1^{q_2}\}$.
Two Levels State Machine algorithm

\{s_0^{q_2}, s_1^{q_2}\}, S^{q_3} = \{s_0^{q_3}, s_1^{q_3}, s_3^{q_3}\}, S^{q_4} = \{s_0^{q_4}\} and S^{q_5} = \{s_0^{q_5}, s_1^{q_5}\}. The inner transition functions are:

\[\delta_0^{q_0} =\]

\[\begin{array}{|c|c|c|}
\hline
s_0^{q_0} & s_1^{q_0} & \bot \\
\hline
a & s_1^{q_0} & \bot \\
b & \bot & \bot \\
c & \bot & \bot \\
d & \bot & \bot \\
e & \bot & \bot \\
f & \bot & \bot \\
\hline
\end{array}\]

\[\delta_1^{q_1} =\]

\[\begin{array}{|c|c|c|c|}
\hline
s_0^{q_1} & s_1^{q_1} & \bot & \bot \\
\hline
a & \bot & \bot & \bot \\
b & s_1^{q_1} & s_1^{q_1} & \bot \\
c & \bot & \bot & \bot \\
d & \bot & \bot & \bot \\
e & \bot & \bot & \bot \\
f & \bot & \bot & \bot \\
\hline
\end{array}\]

\[\delta_2^{q_2} =\]

\[\begin{array}{|c|c|c|c|}
\hline
s_0^{q_2} & s_1^{q_2} & \bot & \bot \\
\hline
a & \bot & \bot & \bot \\
b & \bot & \bot & \bot \\
c & s_1^{q_2} & \bot & \bot \\
d & s_1^{q_2} & \bot & \bot \\
e & \bot & \bot & \bot \\
f & \bot & \bot & \bot \\
\hline
\end{array}\]

\[\delta_3^{q_3} =\]

\[\begin{array}{|c|c|c|c|}
\hline
s_0^{q_3} & s_1^{q_3} & \bot & \bot \\
\hline
a & \bot & \bot & \bot \\
b & \bot & \bot & \bot \\
c & \bot & \bot & \bot \\
d & \bot & \bot & \bot \\
e & \bot & \bot & \bot \\
f & \bot & \bot & \bot \\
\hline
\end{array}\]

\[\delta_4^{q_4} =\]

\[\begin{array}{|c|c|c|c|}
\hline
s_0^{q_4} & s_1^{q_4} & \bot & \bot \\
\hline
a & \bot & \bot & \bot \\
b & \bot & \bot & \bot \\
c & \bot & \bot & \bot \\
d & \bot & \bot & \bot \\
e & \bot & \bot & \bot \\
f & \bot & \bot & \bot \\
\hline
\end{array}\]

\[\delta_5^{q_5} =\]

\[\begin{array}{|c|c|c|c|}
\hline
s_0^{q_5} & s_1^{q_5} & \bot & \bot \\
\hline
a & \bot & \bot & \bot \\
b & \bot & \bot & \bot \\
c & \bot & \bot & \bot \\
d & \bot & \bot & \bot \\
e & \bot & \bot & \bot \\
f & \bot & \bot & \bot \\
\hline
\end{array}\]

The inner final states are:

\[F^{q_0} = \{s_0^{q_0}\}, F^{q_1} = \{s_1^{q_1}\}, F^{q_2} = \{s_1^{q_2}\}, F^{q_3} = \{s_0^{q_3}, s_2^{q_3}\}, F^{q_4} = \{s_0^{q_4}\}, F^{q_5} = \{s_0^{q_5}, s_1^{q_5}\}.\]

10.4.2 Ordered Labeled Tree Validation

TLSM recognizes the OLT. We introduce a recursive algorithm that can validate the OLT TLSM. This algorithm scans once the tree. Therefore, its complexity is \(O(|V^T|)\) where \(|V^T|\) is the number of nodes in the tree.

Part XML Schema extraction
Algorithm 28: Validation of OLT using TLSM.

\[ \text{ValidateOrderedTree} (OLT T = (V^T, E^T, label^T), TLSM M = (Q, q_0, \delta, F, \Sigma)); \]

Result: true / false

1 begin
2 return ValidateNode \( (T, M, \text{root}(T), q_0) \)
3 end
4 ValidateNode \((OLT T = (V^T, E^T, label^T), TLSM M = (Q, q_0, \delta, F, \Sigma), node v \in V, \text{outer state } q \in Q); \)
5 Result: true / false

begin
6 \[ p \leftarrow s_0^q; \]
7 while \( v \neq \lambda \) do
8 \[ q_{\text{next}} \leftarrow \delta(q, label^T(v)); \]
9 \[ p \leftarrow \delta^q(p, label^T(v)); \]
10 if \( p = \perp \) or \( q_{\text{next}} = \perp \) then return false;
11 if MostLeft \((v) = \lambda \) then
12 \quad if \( q_{\text{next}} \notin F \) then return false;
13 \quad else if ValidateNode \((T, M, \text{MostLeft}(v), q_{\text{next}}) = \text{false} \) then
14 \quad \quad return false;
15 \end
16 \[ v \leftarrow \text{Right}(v); \]
17 end
18 return \( p \in F^q; \)
19 end

Algorithm 28 validates the OLT according to a given TLSM. The algorithm contains two methods:

1. ValidateOrderedTree (lines 1-3) is a service method that invokes the recursive method.

2. ValidateNode (lines 4-19) is a recursive method. Lines 6-7 validate that the current state has a transition that uses the label of \( v \). Lines 8-9 validate that \( q \) and \( s_0^q \) are the final states if node \( v \) is a leaf. Lines 11-18 validate that the children order...
of the original node $v$ is acceptable by the inner FSM $q$. Line 15 is a recursive call that validates the sub-tree of $v$.

**Example 10.4** We try to validate the two OLTs $t_4$ and $t_5$ in Fig. 2.7 using the TLSM $B$ in Fig. 10.3. $t_4$ is valid since all the words in $t_4$ are acceptable by the outer layer of $B$. For each $v$ in $t_4$, $\text{Children}(v)$ is acceptable by the appropriate inner FSM. $t_5$ is not valid since the sequence $fe$ is not acceptable by the inner FSM of $q_3$.

### 10.4.3 Main Steps of the XTLSM Algorithm

This section describes the main steps of the XTLSM algorithm. The XTLSM algorithm should satisfy the following properties:

**Separate between elements with same label but have different roles:** An XML schema allows to use the same element name for several element types. Figure 2.1 illustrates such a scenario where not all the elements, which are labeled as $a$, can be classified to have the same type. It is clear that the root element has a different type than the rest of the $a$ elements. The XTLSM algorithm finds elements that have the same type. This property is needed in order to generate a tight grammar.

**Merging elements with different labels that have the same role:** XML Schema allows to use type definition of elements with different labels. In Fig. 2.1 the elements $c$ and $d$ should share the same type definition. This property is needed to create a compact grammar.

**Finding the order of XML elements:** The XTLSM algorithm can provide two data modules as its output: TLSM and XSD. Both modules are ordered labeled tree grammars. This means that children order is part of both grammars. This property can provide more information to applications that use their grammars.

**Converting TLSM into XSD:** Although the TLSM is well defined and easy to use in a computer program, XSD is a standard way to describe an XML in addition to the fact that many tools already know this grammar.
Two Levels State Machine algorithm

Algorithm 29: XML to XSD algorithm

\[
\text{XmlToXsd} (OLT \ T = (V^T, E^T, label^T), \text{precision parameter } \varepsilon) \; ; \\
\text{Result: } \text{XSD } \Pi \text{ notation } - xsd = (\Pi, \pi_0) \; \text{[section 10.2]}
\]

1 \begin{align*}
    &\text{Let } \text{Origin} : S \rightarrow 2^V \text{ be a structure that maps states to } V \text{ subsets;} \\
    &\quad PTA_T \leftarrow \text{TreeToPta} (T, \{\text{root}(T)\}, \text{Origin}) \; ; \quad \text{[alg. 30]} \\
    &\quad fsm \leftarrow \text{Generalize} (PTA_T, \varepsilon, \text{Origin}) \; ; \quad \text{[alg. 31]} \\
    &\quad tlsm \leftarrow \text{AddOrderLayer} (fsm, T, \text{Origin}) \; ; \quad \text{[alg. 34]} \\
    &\quad xsd \leftarrow \text{TwoLayerStateMachineToXsd} (tlsm) \; ; \quad \text{[alg. 35]} \\
    &\text{return } xsd;
\end{align*}

8 \end{align*}

The \text{XmlToXsd} method is the main method of the algorithm. An XML document given in an \textit{OLT} form, is the input of this method. The algorithm contains four steps:

1. \textbf{PTA is created from OLT} (line 3): Creates \textit{PTA} from \textit{T} using the \text{TreeToPta} method (Algorithm 30).

2. \textbf{PTA is generalized into a FSM} (line 4): Generalizes \textit{PTA} into \textit{sm} using the \text{Generalize} method (Algorithm 31).

3. \textbf{Adding inner FSMs to the outer FSM} (line 5): Converts \textit{sm} into \textit{tlsm} by adding \textit{inner FSMs} according to \textit{T}. This operation is done by calling the \text{AddOrderLayer} method.

4. \textbf{XSD is created from TLSM} (line 6): Converts \textit{tlsm} into \textit{xsd} using the \text{TwoLayerStateMachineToXsd} method. The final step is optional since the \textit{TLSM} can be used as an output of the algorithm.

The \textit{Origin} data structure (line 2) maps states in \textit{PTA} and in \textit{sm} into the original nodes in \textit{T} that are associated with them. The \textit{Origin} structure is constructed in Step 1, updated in Step 2 and used in Step 3.

Part [V] XML Schema extraction
10.4.4 Tree to PTA

**Algorithm 30**: Generation of the PTA from a given XML document

\[ \text{TreeToPTA}(OLT = (V^T, E^T, \text{label}^T), D \subseteq 2^V, \text{Origin} : S \mapsto 2^{V^T}); \]

**Result**: \( PTAT = (S, s_0, \delta, F, \Sigma) \) - a tree shaped FSM, \( \text{Origin} \) is updated

begin

1. let \( PTAT = (S = \{s_0\}, s_0, \delta = \emptyset, F = \emptyset, \Sigma); \)

2. if exist \( v \in D \) where \( v \) is a leaf then \( F \leftarrow \{s_0\}; \)

3. \( \text{Origin}[s_0] \leftarrow D; \)

4. foreach \( \alpha \in \Sigma \) do

5. let \( D^\alpha \) be a set of nodes;

6. \( D^\alpha \leftarrow \{\text{all } v \in V^T \text{ where } \text{parent}(v) \in N \text{ and } \text{label}^T(v) = \alpha\}; \)

7. if \( D^\alpha \neq \emptyset \) then

8. let \( PTAt^\alpha = (S^\alpha, s_0^\alpha, \delta^\alpha, F^\alpha, \Sigma) \) be a PTA;

9. \( PTAT^\alpha \leftarrow \text{TreeToPTA}(T, N^\alpha, \text{Origin}); \)

10. \( \delta \leftarrow \delta \cup \delta^\alpha \cup \{(s_0, \alpha) \rightarrow s_0^\alpha\}; \)

11. \( S \leftarrow S \cup S^\alpha; \)

12. \( F \leftarrow F \cup F^\alpha; \)

end

end

return \( PTAT \)

end

Step 1 in Algorithm 30 generates a PTA from the XML document. TreeToPta is a recursive algorithm that generates a new PTA from a given OLT \( T \). The input parameter \( D \) has two goals:

1. It is a set of nodes that contains only the single root node of the tree when Algorithm 30 is called from the main algorithm.

2. When the method is called recursively, \( D \) is a set of nodes that have the same label and level. These nodes denote together a single state in the new PTA.

Since all the tree nodes are visited only once during the execution of this method, the time complexity of Algorithm 30 is \( O(n) \).

Part V XML Schema extraction
Figure 10.1 illustrates a $PTA^T$ that TreeToPTA generated from the tree $T$ in Fig. 2.4. Figures 10.4 and 10.5 illustrate the steps that generate a $PTA^T$ using Algorithm 30.

In each step, the tree extends a single level. The next recursive construction calls are illustrated in each step. The content of $D$ in these illustrations is the parameter of TreeToPta. In some step, there are several recursive calls. The Origin of a state in the new $PTA^T$ are nodes from $T$ that participated in the creation of the state. These are the nodes in $D$. 

Part VII XML Schema extraction
Figure 10.4: Steps 1-5 that create a $\mathcal{PTA}^T$ from OLT $T$ from Fig. 2.4. In each step, the braces contain all the nodes in $D$. 

Part [V] XML Schema extraction
Figure 10.5: Steps 6-9 that create $PTA^{T}$ from $OLT^{T}$ from Fig. 2.4. In each step, the braces contain all the nodes in $D$
10.4.5 Generalization of the PTA

Algorithm 31: Generalization of the PTA into FSM

Generalize($PTA^T = (S, s_0, \delta, F, \Sigma)$, precision parameter $\varepsilon$,
Origin : $S \mapsto 2^V$);

begin
  Data: Similarity matrix $A$ - all values are uninitialized
  Result: $PTA^T$ becomes generalized FSM
  foreach $p, q \in S$ where $label^T(p) = label^T(q)$ do
    Similarity($PTA^T, p, q, A$); [Alg. 32]
  end
  foreach $p, q \in S$ where $label^T(p) = label^T(q)$ and $A[p, q] < \varepsilon$ do
    MergeStates($PTA^T, p, q, \text{Origin}$); [Alg. 33]
  end
  foreach $p, q \in S$ where for all $\alpha \in \Sigma$ applies $\delta(p, \alpha) = \delta(q, \alpha)$ do
    MergeStates($PTA^T, p, q, \text{Origin}$); [Alg. 33]
  end
end

Algorithm 4 is the core of the XTLSM algorithm. It transforms a PTA into a FSM. This algorithm is called once to create the outer FSM. The algorithm has three loops:

1. Calculation (lines 1-3): This loop calculates the similarity between sub-trees in the PTA that have origins with the same label. In this loop, the PTA still has a tree structure. We can assume that inside the similarity function that the PTA has no loops;

2. Threshold merging (lines 4-6): By using the previous calculation, each pair of states, which has similarity value bigger than a threshold, is merged into a single state. Each time such a merge occurs, the PTA can lose the tree structure property. As a result, the PTA becomes a FSM that may contain loops;

3. Parent merging (lines 7-9): It merges all the states that have the same transition function.

Example 10.5 Figure[10.6] is an example of a typical parent merging. The two states $s_i$ and $s_j$ have the same transition function: $\delta^{s_i} = \{(c \rightarrow s_n), (d \rightarrow s_m)\}$ and $\delta^{s_j} = \{(c \rightarrow s_n), (d \rightarrow s_m)\}$.

Part [V] XML Schema extraction
Figure 10.6: Illustration of parent merging (lines 7-9 in Algorithm 31). (a): Before the merge occurs. $s_i$ and $s_j$ have the same transition function. (b): After the merge occurred. $s_i$ and $s_j$ were joined to a single new state that is called $s_{ij}$.
10.4.5.1 Sub-Tree Similarity

**Algorithm 32**: Similarity calculation between two sub-trees in the PTA

\[
\text{Similarity} (PTA_T = (S, s_0, \delta, F, \Sigma)), \text{ states } p, q \in S, \text{ similarity matrix } A) ;
\]

**Result**: \( A \) is updated by the similarity of \( p \) and \( q \)

```plaintext
begin
  sum ← 0.0;
  contributors ← 0.0;
  if for each \( \alpha \in \Sigma : \delta(p, \alpha) = \delta(q, \alpha) = \perp \) then \( A[p, q] ← 1.0; \)
  if \( A[p, q] \) already calculated then return;
  foreach \( \alpha \in \Sigma \) do
    if \( \delta(p, \alpha) \neq \perp \) and \( \delta(q, \alpha) \neq \perp \) then
      \text{Similarity} (pta, \delta(p, \alpha), \delta(q, \alpha), A);
      sum ← sum + \( A[\delta(p, \alpha), \delta(q, \alpha)] + 1; \)
  end
  if \( \delta(p, \alpha) \neq \perp \) or \( \delta(q, \alpha) \neq \perp \) then
    contributors ← contributors + 1.0;
  end
  end
  \( A[p, q] ← \frac{\text{sum}}{\text{contributors}}; \)
  mark \( A[p, q] \) as already calculated;
  return;
end
```

The similarity recursion algorithm calculates the similarity between two sub-trees. If the sub-tree root labels are not the same, the similarity value is always 0.0. This method uses a dynamic programming approach that uses the calculated values.

Line 4 is the recursion base. The similarities between sub-trees, which are leafs, are maximal (1.0).

Line 5 uses dynamic programming calculation. Lines 6-13 are the loop that sums the similarities between children of the current sub-trees roots. The children pairs are chosen according to the symbols of the transition function. Line 8 is a recursion call. Line 15 completes the calculation of the similarity averaging.

This calculation is not a regular averaging of children similarity. Each value, which
Two Levels State Machine algorithm came from two valid states, is modified in line 9. The modification is done by the function $F(x) = \frac{x + 1}{2}$ that increases the value to 1.0. The reason for this modification is to make the differences in distance in the sub-trees less important relatively to the distance.

Figure 10.7: Example of low similarity value between two states in the PTA. The similarity is $Similarity(s_i, s_j) = \frac{1.0 + 0.0 + 0.0 + 0.0 + 0.0}{5} = 0.2$. This similarity value is low since most of the children are non-similar.
Figure 10.8: The similarity is \[ \text{Similarity}(s_m, s_n) = \frac{1.0 + 1.0 + 0.0}{3} = 0.66. \] The similarity value is high since most of the children are similar.

### 10.4.5.2 Merging Two States

Algorithm 6 merges two states into a single state. After each merge, the algorithm looks for non-deterministic transitions. When such a transition is found, the algorithm makes the FSM deterministic by merging ambiguous states.

Part [✔] XML Schema extraction
### Algorithm 33: Merging two states of a FSM

**MergeStates** ($fsm = (S, s_0, \delta, F, \Sigma)$, $p, q \in S$, Origin : $S \rightarrow 2^V$) ;

**Result:** The FSM will be updated with the merge of $p$ and $q$ and makes the FSM deterministic.

1 begin
2     $S \leftarrow S \setminus \{q\}$;
3     Origin[$p$] $\leftarrow$ Origin[$p$] $\cup$ Origin[$q$];
4     foreach $s \in S, \alpha \in \Sigma$ where $s \neq q$ and $\delta(s, \alpha) = q$ do
5         $\delta \leftarrow (\delta \setminus \{(s, \alpha) \rightarrow q\}) \cup \{(s, \alpha) \rightarrow p\}$;
6     end
7     foreach $s \in S, \alpha \in \Sigma$ where $s \neq q$ and $\delta(q, \alpha) = s$ do
8         $\delta \leftarrow (\delta \setminus \{(q, \alpha) \rightarrow s\}) \cup \{(p, \alpha) \rightarrow s\}$;
9     end
10    if $q \in F$ then $F \leftarrow (F \setminus \{q\}) \cup \{p\}$;
11    if $s_0 = q$ then $s_0 \leftarrow p$;
12    foreach $\alpha \in \Sigma$ do
13        foreach $((p, \alpha) \rightarrow s_n) \in \delta, ((p, \alpha) \rightarrow s_m) \in \delta$ where $s_n \neq s_m$ do
14            MergeStates($fsm, s_n, s_m, \text{Origin}$);
15        end
16    end
17 end

---

**Example 10.6** Figure [10.9] demonstrates how to merge non-deterministic states. (a) Before MergeState function was called. $q_i$ and $q_j$ are different states. (b) MergeStates merged them into a single state $q_{ij}$ (lines 2-11). The FSM is non-deterministic in this stage. (c) In order to fix it, MergeStates merges $q_n$ and $q_m$ into a single state $q_{mn}$ (lines 12-16).
Figure 10.9: Example of merging two states using the MergeStates (Algorithm 33). In every step, the same part of the FSM is illustrated. (a) Just before MergeState is called. (b) After merging the wanted states. (c) After fixing the non-deterministic states.

Example 10.7 Figure 10.10 demonstrates how three states are merged into a single state. (a) Before the MergeStates function was called. $q_i$ and $q_j$ are different states. (b) MergeStates merged them into a single state $q_{ij}$ (lines 2-11). The FSM is non-deterministic in this stage. (c) In order to fix it, MergeStates merges $q_{ij}$ and $q_k$ into a single state $q_{ijk}$ (lines 12-16). This causes all the three states to collapse into a single state $q_{ijk}$. The accepting property is not lost. The single state is an accepting (final) state.

Figure 10.10: Example of merging two states using the MergeStates function. In every step, the same part of the FSM is illustrated. (a) Before the MergeStates is called. (b) After merging the wanted states. (c) After fixing the non-deterministic states.
10.4.6 Adding of Order Layer

Algorithm 34: Adding order layer

```
AddOrderLayer(fsm = (Q, q0, δ, F, Σ), T = (V^T, E^T, label^T),
Origin : S ↦→ 2^V^T);

Result: fsm is extended into TLSM
```

```
begin
  foreach state q ∈ Q do
    extend q to be inner FSM q = (S^q, s^q_0, δ^q = δ^↓, F^q, Σ);
    [δ^↓ = \bigcup_{α∈Σ} \{(s^q_0, α)→⊥\}]
    foreach node v ∈ Origin[q] do
      state s ← s^q_0;
      node v_{child} ← MostLeft(v);
      while v_{child} ≠ λ do
        α ← label^T(v_{child});
        if δ^q(s, α) = ⊥ then
          if exists s_{src} ∈ S^q where δ^q(s_{src}, α) ≠ ⊥ then
            p ← δ^q(s_{src}, α);
          else
            p ← new inner state;
            S^q ← S^q \cup \{p\};
          end
        δ^q ← δ^q \cup \{(s, α)→p\};
        s ← δ^q(s, α);
        v_{child} ← Right(v_{child});
        if v_{child} = λ then F^q ← F^q \cup \{s\};
      end
    end
end
```

Algorithm 34 extends the FSM into a TLSM by converting each state in the original
outer FSM into an inner FSM. The states in the inner FSM are built by using all the nodes that are the origins of the outer state.

The algorithm takes each node and builds the inner FSM to accept all the strings that match the children order. In the inner FSM, each inner state will be associated with a single symbol from $\Sigma$. This means that for the same symbol, all the transitions go into the same inner state $\delta^q(s_i, \alpha) = \delta^q(s_j, \alpha), \forall s_i, s_j \in S$.

**Example 10.8** Figure [10.11] illustrates part of a state machine. Figure [10.12] illustrates the relevant nodes in the OLT that are associated with the state $q_1$ in Fig. [10.11]. The connection between the two figures is the origin of state $q_1$: $\text{Origin}(q_1) = \{n_1, n_{10}, n_{20}, n_{30}\}$. Figure [10.13] is the inner FSM that Algorithm 34 created from the given state machine and the tree.

![Figure 10.11: Illustration of several states in a FSM](image)

Part [XML Schema extraction]
Figure 10.12: Tree nodes that are the origin of state $q_1$ in Fig. 10.11

Figure 10.13: *Inner FSM* created for the state $q_1$ from the nodes in Fig. 10.12 by Algorithm 34

Part V: XML Schema extraction
10.4.7 Extract $\Pi$ Notation XSD from FSM

**Algorithm 35**: TLSM to $\Pi$ notation XSD

StateMachineToXSD ($\text{TLSM} = (Q, q_0, \delta, F, \Sigma)$);

begin
  Data: state machine $sm$
  Result: xsd definition $\Pi$
  foreach $q \in Q$ do
    $\pi_q \leftarrow \text{SMXSD}(q)$;
    [alg. 36]
    $\Pi \leftarrow \Pi \cup \{\pi_q\}$;
    if $q = q_0$ then $\pi_0 = \pi_q$;
  end
  return $\Pi$;
end

Algorithm [35] is the main method for converting the TLSM into an XSD $\Pi$ notation. The main method generates a single $\pi$ expression from each inner FSM. Each $\pi$ expression depends only on a single inner FSM.

Algorithm [36] is based on the algorithm in [72] that obtains REs from finite-state automata. The idea is similar. We just replaced the RE with the XSD $\Pi$ notation element definition. The XMXSD method operates on the inner FSM $q$ and produces a XSD $\Pi$ notation element definition that accepts the same language that the inner FSM accepts. We assume that the given inner FSM has non acceptable start state $s_0 \notin \Sigma$ and there is no transition into the start state: there are no $s \in S^q, \alpha \in \Sigma$ where $\delta(s, \alpha) = s_0$.

**Line 2**: If global detours exits they are removed. As a result, $q$ is updated to have no detour and $\text{min}$ becomes 0 if such a detour exists or 1 otherwise.

**Line 3**: Global loop is removed. $q$ is updated not to have this loop and $\text{max}$ is $\infty$ if such loop removed and 1 otherwise.

**Lines 4-8**: The recursive base. When there are only two states, the returned expression will be a simple XSD reference expression.

**Lines 9-12**: The inner FSM is chopped (if possible) into vertical chops. An XSD sequence expression is built from these vertical chops.
**Lines 13-16:** The same as in lines 9-12. Instead of vertical chops the inner FSM is chopped into horizontal chops and an XSD choice expression is generated.

**Lines 17-21:** When no chopping is possible, the last approach is a brute force approach. A $\Pi$ expression is generated from any possible state sequence using state elimination. The minimal $\Pi$ expression is used.

---

**Algorithm 36:** Convert inner FSM to $\Pi$ notation XSD element

$$\text{SMXSD}(\text{inner FSM } q = (S^q, s^q_0, \delta^q, F^q, \Sigma));$$

**Result:** xsd element definition $\pi$

```plaintext
begin

1 $min \leftarrow \text{RemoveGlobalDetour}(q); \quad \text{[alg. 37]}$
2 $max \leftarrow \text{RemoveGlobalLoop}(q); \quad \text{[alg. 38]}$
3 if $|S^q| = 2$ then
4     Locate $\alpha$ where $\delta^q(s^q_0, \alpha) \in F^q$;
5     $q_{next} \leftarrow \delta(q, \alpha)$;
6     return $\alpha \rightarrow \pi_{q_{next}}$;
7 end
8 $chops \leftarrow \text{VerticalChop}(q); \quad \text{[alg. 39]}$
9 if $|chops| > 1$ then
10     return (SMXSD(chops[1]), ..., SMXSD(chops[n]))$^{min-max}$;
11 end
12 $chops \leftarrow \text{HorizontalChop}(q); \quad \text{[alg. 42]}$
13 if $|chops| > 1$ then
14     return (SMXSD(chops[1])|...|SMXSD(chops[n]))$^{min-max}$;
15 end
16 find shortest $\omega$ expression using state elimination.; \quad \text{[sec. 10.4.7.5]}
17 return $\omega^{min-max}$;
18 end
```

---

**10.4.7.1 Remove Global Detour method**

The remove global detour method detects and removes detours in the FSM. detours are associated with the $\Pi$ notation zero quantity.
Algorithm 37: Remove global detour from FSM

RemoveGlobalDetour ($FSM \ A = (S, s_0, \delta, F, \Sigma)$);

Result: $A$ is updated - global detour is removed. The return value is 0 if such
detour is removed or 1 otherwise.

1 begin
2    if $s_0 \in F$ and $|F| > 1$ then
3        $F \leftarrow F \setminus \{s_0\}$;
4        return 0;
5    end
6    return 1;
7 end

Figure 10.14: Illustration of the removal by the global detour method: $s_0$ is the start
state. All the other states are showed as a cloud. (a) The FSM as a start state that it is
also an accepting state. This means that there is a detour for all the states machine. (b)
Illustrates the same FSM after the application of the detour removal. $s_0$ is no longer an
accepting state. To compensate for it, the $\Pi$ notation quantity becomes zero which is
the equivalent for the removed detour.

Part [✓] XML Schema extraction
10.4.7.2 Remove Global Loop

The Remove global loop method detects and removes a global loop from the FSM. These loops are associated with the $\Pi$ notation infinite quantity.

**Algorithm 38**: Remove global loop from FSM

RemoveGlobalLoop($FSM\ A = (S, s_0, \delta, F, \Sigma)$);

**Result**: $A$ is updated - global loop is removed. The return value is $\infty$ if such loop is removed or 1 otherwise.

```plaintext
begin
1 if for all $s \in F, \alpha \in \Sigma : \delta(s_0, \alpha) = \delta(s, \alpha)$ then
2     foreach $s \in F, \alpha \in \Sigma$ do
3         $\delta(s, \alpha) \leftarrow \bot;$
4     end
5     return $\infty$;
6 end
7 return 1;
8 end
```

Part [XML Schema extraction]
Figure 10.15: Illustration of the Remove global loop method: $s_0$ is the start state. The A cloud represents all the states that are the children of the start state. The F cloud represents all the states that are the accept states. (a) All the transitions from all states in $F$ are similar to the transitions from $s_0$. This means that the language of the FSM contains endless words. (b) Illustration of the same FSM after the application of the Remove Global Loop was applied. All the states in $F$ have only transitions to $\perp$. The loop in the FSM is replaced by $\Pi$ notation infinite quantity.

10.4.7.3 Vertical Chopping

The vertical chopping method chops the FSM into several sub-state machines. This chopping is associated with the $\Pi$ notation sequence expression.
Algorithm 39: FSM vertical chopping - main method

```
Algorithm 39: FSM vertical chopping - main method

VerticalChop(FSM A = (S, s₀, δ, F, Σ));

Result: chops is array of FSMS vertically chopped from A

1 begin
2 Find a path from s₀ to a state f ∈ F;
3 Let C = (s₀, b₁, b₂, ...f) be a sequence of states of the path;
4 Let min, anc, max be arrays, all initiated to #;
5 Let visited be an empty set;
6 CalculateMinAncMax(s₀, C, min, anc, max, visited); [alg. 40]
7 Let bridges be the set of states from C;
8 visited ← ∅;
9 FindBridges(s₀, bridges, C, min, anc, max, visited); [alg. 41]
10 Build chops so every sub-FSM will be contained between two states in bridges;
11 return ;
12 end
```

The VerticalChop algorithm (Algorithm 39) decomposes A into several sub-automata A₁, A₂, ..., Aₙ such that L(A) = L(A₁) · L(A₂) · ... · L(Aₙ). A is chopped vertically in the bridge states where the bridge states are neither a start nor a final state. The accepting path will pass through every bridge state at least once for every word that is accepted by the FSM. The path never passes through any states between several visits of the same bridge. To find the bridge states, the algorithm uses a single path (line 3).

Line 6 calculates: anc, min and max values of the states using DFS where

ancᵢ: The index i of the state bᵢ ∈ C such that there is a path from bᵢ to s and there is no path from bⱼ ∈ C to s for j > i. Then, ancᵢ = i.

minᵢ: The index i of the state bᵢ ∈ C such that there is a path from s to bᵢ and there is no path from s to bⱼ for h < i in the FSM without visiting any states in C.

maxᵢ: The index i of the state bᵢ ∈ C such that there is a path from s to bᵢ and there is no path from s to bⱼ for i < j in the FSM without visiting any states in C.

All the non-bridge states are eliminated according to the calculated values (line 9). At the last step, the algorithm processes the set of bridges and decomposes the FSM into
several FSMs such that every sub-state machine will be contained between two bridges. (line 10).

Figure 10.16: Illustration of the vertical chopping operation. (a) The states are marked with indexes from 1 to 9. (b) All the sub-FSM after the chopping according to the bridges (states 2, 6 and 7).
Part [V] XML Schema extraction
Algorithm 40: FSM vertical chopping - calculate min anc and max

CalculateMinAncMax(q, C = (s_0, b_1, b_2, ..., f), min, anc, max, visited, ancIndex);

Result: min, anc, max are updated with true values

begin

if q = f or q ∈ visited then return;

visited ← visited ∪ {q};

if q ∈ C then

let next be the next state in C after q;

CalculateMinAncMax(next, C, min, anc, max, visited, ancIndex + 1); end

minNotInPath ← #;

maxNotInPath ← #;

minAncInPath ← #;

maxAncInPath ← #;

foreach α ∈ Σ where δ(q, α) ≠ ⊥ do

q_{next} ← δ(q, α);

CalculateMinAncMax(q_{next}, C, min, anc, max, visited, ancIndex);

if q_{next} ∈ C then

minAncInPath ← Minimum(minAncInPath, anc[q_{next}]);

maxAncInPath ← Maximum(maxAncInPath, anc[q_{next}]);

dend

else

minNotInPath ← Minimum(minAncInPath, min[q_{next}]);

maxNotInPath ← Maximum(maxAncInPath, max[q_{next}]);

dend

min[q] ← Minimum(minAncInPath, minNotInPath);

anc[q] ← ancIndex;

max[q] ← Maximum(maxAncInPath, maxNotInPath);

return;

dend
The CalculateMinAncMax method recursively calculates the \textit{min anc} and \textit{max} values of all the states in the FSM.

![Diagram of a selected path and the calculated min, anc, and max values](image)

Figure 10.17: Illustration of a selected path and the calculated min, anc, and max values. The selected path is shown with heavy arrows \((s, b_1, b_2, ..., f)\). The values of min, anc, and max are in parentheses near each state \((\text{min, anc, max})\).
Algorithm 41: FSM vertical chopping - removal of non bridges

\[
\text{FindBridges}(q, \text{bridges}, \text{C}, \min, \text{anc}, \max, \text{visited});
\]

**Result:** All states that are not bridges will be removed from the set \text{bridges}

```
begin
if \( q \in \text{visited} \) then return ;
\text{visited} \leftarrow \text{visited} \cup \{q\};
index \leftarrow \#;
foreach \( s \in \text{C} \) do
index \leftarrow \text{anc}\_s;
if index > \min[q] \text{ and } index \leq \text{anc}_q \text{ and } \min[q] \neq \# \text{ then}
\text{bridges} \leftarrow \text{bridges} \setminus \{s\};
end
if index > \text{anc}[q] \text{ and } index < \max[q] \text{ then}
\text{bridges} \leftarrow \text{bridges} \setminus \{s\};
end
end
foreach \( \alpha \in \Sigma \text{ where } \delta(q, \alpha) \neq \bot \) do
\text{FindBridges}(\delta(q, \alpha), \text{bridges}, \text{C}, \min, \text{anc}, \max, \text{visited});
end
return ;
end
```

The \text{FindBridges} method removes all the states that violate any requirement to be a bridge from the list of bridges.

10.4.7.4 Horizontal Chopping

The horizontal chopping method chops the FSM into several sub-states machine. This chopping is associated with the Π notation choice expression.
Algorithm 42: FSM horizontal chopping

HorizontalChop(FSM \( A = (S, s_0, \delta, F, \Sigma) \));

Result: \( chops \) is array of sub-automaton that were chopped horizontally from \( A \)

begin
  let \( clusters \) be a map from \( S \) to \( \Sigma \);
  initiate \( clusters \) to map every state to the error cluster \( \# \);
  \textbf{foreach} \( \alpha \in \Sigma \) \textbf{do}
    MarkCluster(\( A, \delta(s_0, \alpha), \alpha, clusters \)); \[alg. 43\]
  \textbf{end}
  Split \( A \) to several sub-automaton \( \{A_1, A_2, ..., A_n\} \) so every sub-automaton will contain states with the same cluster mark;
  \( chops = \{A_1, A_2, ..., A_n\} \);
  \textbf{return} \( chops \);
end

Algorithm 43: Mark Cluster

MarkCluster(FSM \( A, state \ s, symbol \ \alpha, clusters : S \mapsto \Sigma \));

Result: mark

begin
  if \( s \neq \bot \) then
    if \( clusters[s] = \# \) then
      \( clusters[s] \leftarrow \alpha \);
      \textbf{foreach} \( \beta \in \Sigma \) \textbf{do}
        MarkCluster(\( A, \delta(s, \beta), \alpha, clusters \));
      \textbf{end}
    else if \( clusters[s] \neq \alpha \) then
      \textbf{foreach} \( q \in S \ where \ clusters[q] = clusters[s] \) \textbf{do}
        \( clusters[q] \leftarrow \alpha \);
      \textbf{end}
    \textbf{end}
  \textbf{end}
end

Part V: XML Schema extraction
The Horizontal chopping method is based on DFS. Marked clusters are maintained for each state according to the symbol we used to exit from the start state during the application of the DFS. When the DFS detects a state, which was already marked with a different cluster, the old cluster is merged with the current one. After the DFS is completed, the cluster’s mark is used to chop horizontally the FSM into several sub-FSMs.
Figure 10.18: Illustration of horizontal chopping. (a) FSM $A$ before the application of the horizontal chopping. (b) Step 1: The marking of the $\alpha$ cluster (states 4, 5, and 6). (c) Step 2: The marking of the $\beta$ cluster (states 8 and 9). (d) Step 3: The marking of the $\theta$ cluster (states 8, 9, 10 and 11) In this step, the new cluster takes over all the $\beta$ cluster states due to the transition from state 11 to state 9. The $\beta$ cluster is now obsolete. (e) Step 4: The two sub-FSMs $A_1$ and $A_2$ created after the application of the horizontal chopping. It is obvious that the language of the FSM in (a) is the same as the union set of the two languages of the FSMs in (b).
10.4.7.5 State Elimination

State elimination is a method that converts FSMs into REs. It was introduced in [34] to compute REs from FSM. The idea is to remove states except for start and final states. When a state is removed then the alphabet of the FSM is expended to compensate for the missing state. Since the original state elimination converts the FSM into REs, the alphabet was expended to contain REs. In order to convert FSMs into Π notation ω expressions, the alphabet of the FSM is expended to contain ω expressions.

The original state elimination algorithm produces RE. In order to modify the algorithm to produce ω expression we introduce the following illustration:

![Diagram](image)

Figure 10.19: Illustration of reducing a single state $p_j$ from a FSM by the state elimination approach. $p_i$ and $p_k$ are two of the states that have transitions from or to $p_j$. After $p_j$ is removed, the transitions of all these states become more complex. The alphabet symbols on the transitions become Π notation ω expressions.

Part [✓] XML Schema extraction
10.5 Experimental results

In this section, we present experimental results. This section has the following structure: section 10.5.1 provides some synthetic examples that demonstrate the products of the XTLSM algorithm. Section 10.5.2 describes the experimental settings. Section 10.5.3 conducts a comprehensive study of time and space performances of the XTLSM algorithm. Section 10.5.4 studies the generalization rate of the XTLSM algorithm. Our experimental results show that XTLSM is a fast and memory efficient Schema extraction algorithm. The generalization experiments shows that XTLSM extracts the exact schema from a small portion of XML documents.

10.5.1 Synthetic examples

In this section, we demonstrate two features of the XTLSM algorithm. The ability to detect an infinite element and represents it in the XML Schema is described in section 10.5.1.1. The ability to differentiate between element definitions for elements with the same name is described in section 10.5.1.2.

10.5.1.1 FSMs with Loops

This section demonstrates the ability of the algorithm to detect and represents in an XML Schema the infinite element definition.

1. `<x>`
2. `<a><b><c></c></b></a>`
3. `<a><b><c><a><b><c></c></b></a></c></b></a>`
4. `</x>`

Figure 10.20: An example of an XML document that contains a loop: an XML element contains instances of its own type (line 3).
Figure 10.21: The TLSM that was created from the XML in Fig. 10.20. The recursion in the XML document generated three steps length loop in the external FSM.
Figure 10.22: The XML Schema that was generated from the XML document in Fig. 10.20. It also contains recursive definition of elements: a contains b. b contains c. c contains a.

10.5.1.2 Different element definitions for elements with the same name

1. <x>
2. <a><c/></a><c/></a><c><d/></c></a>
3. <b><c><c/></c></b></c></b>
4. </x>

Figure 10.23: An example of XML document that contains two elements with same name with different purposes

Part □ XML Schema extraction
Figure 10.23 shows an XML document with several elements where all have the same name (c) that do not share the same role. In line 2, c elements can contain d elements, but in line 3 c elements contain c elements.

Figure 10.24: The TLSM was created from the XML in Fig. 10.23. There are two external states, one for each role of the c element. $q_4$ is the external state for the c elements in line 2, while $q_5$ is the external state for the c elements in line 3.

Part V: XML Schema extraction
Figure 10.25: An example of XSD that contains two elements definition with the same element name. The XML Schema in Fig. [10.25] was generated from the XML document in Fig. [10.23]. There are two types of c element definitions: c1 for the c elements in line 2 and c2 for the c elements in line 3.
10.5.2 Experimental settings for XML datasets

We implemented all the algorithms in C++. All our experiments were performed on a PC with 2.8GHz Intel Pentium core duo processor with 2048MB RAM running Windows XP. We used the following synthetic datasets for our experiments: 1. XMark [124]; 2. Randomly generated XML. The XMark dataset is synthetic and is generated by an XML data generator. It contains auction site information. XMark by itself is too limited for experimenting with the XTLSM algorithm. The XMark data is generated from a single schema. Therefore, the XTLSM algorithm can only generate a single XML Schema from the XMark dataset. Furthermore, the XMark generates a single data file. Therefore, the generalization can not be measured on the XMark dataset.

In order to conduct comprehensive experiments, we produced a new XML data generator. We denote, hereinafter, this XML data generator by XMLGen+ . The XMLGen+ operates in two sequential phases: The first phase generates a random XML Schema and the second phase generates a collection of XML documents that are valid for the generated XML Schema which was generated in the first phase. The generated XML Schema contains a random number of complex and simple XSD element definitions. The number of complex element definitions is distributed randomly between XSD sequence definitions and XSD choice definitions. The number of XSD elements, which are referred by a complex element definition, is random. This construction generates a complex and nested XML schema.

10.5.3 Computational complexity (Time and space) of the XTLSM Algorithm

In this section, we present the performance of Algorithm 29 for processing XML documents. The XMark [124] dataset was used as a source for processing huge XML documents. The program, which we tested, contained only an implementation that converts an XML into a TLSM. Since the TLSM we got was very small, we can estimate that the time and space needed to convert the TLSM into XSD are negligible. Five XML documents with sizes from 12 to 224 MB were converted into TLSMs. Although the theoretical analysis produces worse results, the experimental analysis exhibits linear time and space complexities (scalability).
<table>
<thead>
<tr>
<th>XML size</th>
<th>XML elements</th>
<th>time</th>
<th>space</th>
<th>elements/second</th>
<th>space/element</th>
</tr>
</thead>
<tbody>
<tr>
<td>12MB</td>
<td>167863</td>
<td>1.08 sec.</td>
<td>45MB</td>
<td>155k</td>
<td>281 bytes</td>
</tr>
<tr>
<td>23MB</td>
<td>336242</td>
<td>2.09 sec.</td>
<td>90MB</td>
<td>160k</td>
<td>280 bytes</td>
</tr>
<tr>
<td>56MB</td>
<td>832909</td>
<td>5.12 sec.</td>
<td>237MB</td>
<td>162k</td>
<td>298 bytes</td>
</tr>
<tr>
<td>112MB</td>
<td>1666309</td>
<td>10.24 sec.</td>
<td>450MB</td>
<td>162k</td>
<td>283 bytes</td>
</tr>
<tr>
<td>224MB</td>
<td>3337647</td>
<td>20.48 sec.</td>
<td>915MB</td>
<td>162k</td>
<td>287 bytes</td>
</tr>
</tbody>
</table>

Table 10.1: XML size: The size of the XML documents in megabytes, XML elements: the number of XML elements in the XML document, time: seconds to process the XML documents into TLSM, space: megabytes to process the XML document into TLSM, \(\text{elements/second}\): the average number of processed elements per second, \(\text{space/element}\): the average space needed to process a single element.

We see from Table 10.1 that both time and space complexities are linear (scale well) since the two right most columns are invariant. One of the reasons is the nature of the processed XML document. Although the sizes of the XML documents vary, the sizes of their PTA were almost the same. The scalability in time and space complexities are explained by the following fact: most of the calculations were performed on the PTA of the XML documents and the similarity matrix, which is considered to be the most significant data structure, was generated on the PTA module.

10.5.4 Generalization measurements of the XTLSM algorithm

While the computational performance (in terms of time and space) is an important factor in our work, the goal of the experiments is to measure the quality of the results. For this we used extensively synthetic data that is attractive for evaluating the quality of the typing because we were able to compare between the types produced by XTLSM with the intended type in the data specification. We measure the generalization of the XTLSM algorithm by the following testing procedure. First, the procedure used the XMLGen+, which is described in section 10.5.2 to produce the random sets \(S_i\), \(1 \leq i \leq 100\), of XML documents. The set \(S_i\) contains XML documents \(D_{i,j}\), \(1 \leq j \leq 100\), that are valid for a specific random XML schema. For each set \(S_i\), the testing procedure used Algorithm 29 to generate XSD \(X_{i,k}\) from the documents \(D_{i,1}, \ldots, D_{i,k}\), \(1 \leq k \leq 100\). The documents \(D_{i,1}, \ldots, D_{i,k}\) are joined into a single document that is given as an input to Algorithm 29. The testing procedure calculates the number...
$n_{i,k}$ of documents in $D_{i,j} \in S_i$ that are valid for $XSD X_{i,k}$. The testing procedure calculates the \textbf{generalization parameter} $n_k = \Sigma_{1 \leq i \leq 100} n_{i,k}/(100 \times 100)$, $1 \leq k \leq 100$. The generalization parameter measures the quality of the induction of the $XTLSM$ algorithm as a function of the size of the training set. Figure 10.26 shows the results. The graph illustrates the generalization parameter $n_k$, $1 \leq k \leq 100$. The results shows that an XML schema, which is inducted from less than 10% of the data, is sufficient to validate all the XML documents in the dataset.

Figure 10.26: Graph of the $XTLSM$ generalization parameter $n_k$ that is achieved from induction of 1 to 100 XML documents.

### 10.6 Conclusions

$XTLSM$ is a data module that can be used to describe ordered tree languages. We presented an algorithm that takes an XML document as an input to construct a TLSM. We were able to construct a TLSM from XML documents in linear time and space complexities (preserve scalability). The algorithm converts TLSM into an XML Schema like a data module. The algorithm clustered elements from the XML document according to the elements role. An XSD element definition is generated for each cluster.

Part [X] XML Schema extraction
Bibliography


[51] DBLP. "http://www.informatik.uni-trier.de/~ley/db/index.html".


[90] Tak Cheung Lam. *Xml document parsing: Operational and performance characteristics*.


