Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Paweł Parys

Application of automata theory to processing of XML documents

PhD dissertation

Supervisor
dr hab. Mikołaj Bojańczyk

Institute of Informatics
Warsaw University

May 2011

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

May 30, 2011 . . . .

date *Paweł Parys*

Supervisor's declaration:
the dissertation is ready to be reviewed

May 30, 2011 . . . .

date *dr hab. Mikołaj Bojańczyk*

Abstract

In this thesis we study the problem of evaluating XPath queries in XML documents. We present an algorithm that, given an XPath node selecting query φ and an XML document t, returns the set of nodes in t that satisfy φ.

We consider a fragment of XPath 1.0 called FOXPath, where attribute and text values may be compared. We also consider an extension of FOXPath, called Regular XPath, in which arbitrary regular expressions may appear as path expressions. Our algorithms are constructed basing on an automata-theoretic approach: fragments of the input query are translated into finite automata.

We present four variants of the algorithm, having different properties and time complexities. The first variant of the algorithm works for queries from the Regular XPath fragment, and has linear time data complexity. The constant in the linear time of this algorithm is exponential in the query size. This algorithm uses deterministic automata. The second variant of the algorithm also has linear time data complexity, but it has polynomial time combined complexity. This algorithm uses the special form of path expressions in FOXPath, which in fact are less expressive than regular expressions. Hence the algorithm does not work for Regular XPath, only for FOXPath. The third variant of the algorithm is the simplest one. It has $O(|t| \log |t|)$ time complexity in the document size $|t|$, polynomial combined complexity, and works for the whole Regular XPath as well. Probably among the four algorithms this one may be most useful in the practice. It is easier to understand and implement, which probably compensates for the additional $\log |t|$ factor. Finally, we present a variant of the algorithm, which works for Regular XPath in time linear in the document size, and polynomial in the query size and the document height. So its running time depends also on the document height, but in practice the document height is usually very small.

Moreover, we present an algorithm evaluating binary queries of XPath, i.e. queries which return pairs of nodes. This algorithm first does a precomputation on the document in linear time and then outputs the selected pairs with constant delay between them. This algorithm also have four variants, corresponding to variants of the first algorithm.

As a side effect, we show a solution to the following problem of quick infix evaluation. There is given a word $w = a_1 a_2 \dots a_n$, and a finite automaton $\mathcal{A}$. We are allowed to preprocess the word in time linear in n. Then we have to answer in constant time queries of the form: given two indices $i \leq j$, does $\mathcal{A}$ accepts the subword $a_i a_{i+1} \dots a_j$?

We also develop the theory of constant height factorization forests, introduced by I.Simon. We show an algorithm which calculates such factorization forests for the monoid of binary relations over some finite set Q. Our algorithm works in time linear in the length of the word (size of the forest), and polynomial in the size of the set Q (hence logarithmic in the monoid size). All previously known algorithms were polynomial in the monoid size, hence exponential in the size of Q. This construction is then used in one of the variants of the XPath evaluation algorithms.

Finally, we perform some experiments on existing software performing XPath evaluation. This experiments show that these programs are very slow for some queries, hence that the algorithms used there are significantly worse than the algorithms known from theoretical papers.

Streszczenie

W poniższej rozprawie badamy problem wyliczania zapytań XPath w dokumentach XML. Przedstawiamy algorytm, który dla danego zapytania XPath φ wybierającego wierzchołki dokumentu oraz danego dokumentu t znajduje zbiór wierzchołków t spełniających φ.

Rozważamy fragment języka XPath 1.0 nazywany FOXPath, w którym wartości atrybutów i wartości tekstowe mogą być porównywane. Rozważamy również rozszerzenie fragmentu FOXPath, nazywane Regular XPath, w którym dowolne wyrażenia regularne mogą pojawiać się jako wyrażenia ścieżkowe. Nasze algorytmy skonstruowane zostały w oparciu o podejście korzystające z teorii automatów: fragmenty wejściowego zapytania są tłumaczone do automatów skończonych.

Przedstawiamy cztery warianty algorytmu, mające różne własności i złożoności czasowe. Pierwszy wariant algorytmu działa dla zapytań z fragmentu Regular XPath i ma liniową złożoność czasową ze względu na rozmiar dokumentu. Jednak stała przy czasie liniowym w tym algorytmie jest wykładnicza ze względu na rozmiar zapytania. Algorytm ten używa automatów deterministycznych. Drugi wariant algorytmu również ma liniową złożoność czasową ze względu na rozmiar dokumentu, lecz ma wielomianową złożoność ze względu na rozmiar zapytania. Ten algorytm korzysta ze szczególnej postaci wyrażeń ścieżkowych używanych w FOXPath, które w rzeczywistości mają mniejszą siłę wyrazu niż dowolne wyrażenia regularne. Zatem algorytm ten nie działa dla fragmentu Regular XPath, tylko dla FOXPath. Trzeci wariant algorytmu jest najprostszy. Ma złożoność czasową $O(|t| \log |t|)$ ze względu na rozmiar dokumentu $|t|$, wielomianową złożoność ze względu na rozmiar zapytania i działa także dla całego fragmentu Regular XPath. Przypuszczalnie pośród opisywanych czterech algorytmów właśnie ten jest najbardziej użyteczny w praktyce. Jest łatwiejszy do zrozumienia i zaimplementowania, co prawdopodobnie rekompensuje dodatkowy czynnik $\log |t|$. Z kolei czwarty wariant algorytmu działa dla fragmentu Regular XPath w czasie liniowym ze względu na rozmiar dokumentu i wielomianowym ze względu na rozmiar zapytania i wysokość dokumentu. Zatem jego czas działania zależy także od wysokości dokumentu, jednak ona w praktyce jest bardzo niewielka.

Ponadto przestawiamy algorytm wyliczający zapytania binarne z XPath, tzn. zapytania które zwracają pary wierzchołków. Algorytm ten najpierw przetwarza dokument w czasie liniowym, a następnie zaczyna znajdować pary spełniające zapytanie, każda kolejna para jest znajdowana w czasie stałym po poprzedniej. Ten algorytm także ma cztery warianty, odpowiadające czterem wariantom pierwszego algorytmu.

Jako efekt uboczny, podajemy także rozwiązanie następującego problemu szybkiego wyliczania podsłów. Początkowo na wejściu dane jest słowo $w = a_1 a_2 \dots a_n$ oraz automat skończony $\mathcal{A}$. Możemy wykonać pewne obliczenia w czasie liniowym ze względu na n. Następnie powinniśmy w czasie stałym odpowiadać na zapytania postaci: dane są dwa indeksy $i \le j$, czy $\mathcal{A}$ akceptuje podsłowo $a_i a_{i+1} \dots a_j$?

Rozwijamy również teorię lasów rozkładu stałej głębokości, wprowadzonych przez I.Simona. Podajemy algorytm, który oblicza takie lasy rozkładu dla przypadku półgrupy relacji binarnych nad pewnym zbiorem skończonym Q. Przedstawiany algorytm działa w czasie liniowym ze względu na długość słowa (która jest jednocześnie rozmiarem lasu rozkładu) i wielomianowym ze względu na rozmiar zbioru Q (zatem logarytmicznym ze względu na rozmiar półgrupy). Wszystkie wcześniej znane algorytmy były wielomianowe ze względu na rozmiar półgrupy, czyli wykładnicze ze względu na rozmiar zbioru Q. Konstrukcja ta używana jest następnie w jednym z wariantów algorytmu obliczającego zapytania XPath.

Przeprowadzamy także pewne eksperymenty na istniejącym oprogramowaniu służącym do wyliczania zapytań z języka XPath. Pokazują one, iż badane programy dla pewnych zapytań działają bardzo powoli, zatem użyte w nich algorytmy są znacznie gorsze niż algorytmy znane z prac teoretycznych.

Contents

1 Introduction **7**
1.1 Basic definitions and used facts . . . 8
1.1.1 Trees . . . 8
1.1.2 Binary relations . . . 9
1.1.3 Edge labelled trees . . . 10
1.2 XML documents and XPath . . . 10
1.2.1 Data model . . . 10
1.2.2 Regular XPath . . . 11
1.3 Real world XPath systems . . . 12

2 Factorization forests **15**
2.1 Monoid of binary relations . . . 16
2.2 First step: $\mathcal{J}$-homogeneous forest . . . 18
2.3 From a $\mathcal{J}$-homogeneous forest to a homogeneous one . . . 20

3 Fast evaluation of paths **23**
3.1 Logarithmic querying . . . 23
3.2 Tape construction . . . 24
3.2.1 Tape construction for words . . . 24
3.2.2 Tapes in a tree . . . 25
3.3 Polynomial combined complexity for words . . . 26
3.3.1 Accelerating pointers . . . 26
3.3.2 Logarithmic querying . . . 28

4 A problem of simplifying snippets **29**
4.1 Linear-logarithmic algorithm . . . 30
4.2 Tape construction . . . 31
4.3 Polynomial combined complexity for words . . . 32

5 Evaluating Regular XPath node tests **35**
5.1 Proof strategy . . . 35
5.2 Preparing the tree . . . 36
5.3 From path expressions to automata . . . 38
5.4 Inequalities . . . 41
5.5 Equality tests . . . 41
5.6 Small height of a document . . . 46

6 FOXPath **47**
6.1 Basic automata . . . 47
6.2 Precomputing automaton runs . . . 48
6.3 Simplifying the snippets . . . 50

7 Regular XPath with aggregation **53**
7.1 Unnested aggregates 54
7.2 Arbitrary node tests and numeric expressions 54

8 Evaluation of path expressions **56**
8.1 An auxiliary problem 56
8.1.1 Linear-logarithmic algorithm 58
8.1.2 Linear algorithm for Regular XPath 58
8.1.3 Polynomial combined complexity for words 59
8.1.4 Polynomial combined complexity for FOXPath 60
8.2 Path expressions 61

Chapter 1

Introduction

In this thesis, we present an algorithm that, given an XPath node selecting query φ and an XML document t, returns the set of nodes in t that satisfy φ. XPath evaluation algorithms that are built into browsers are very inefficient, and may have running times that are exponential in the size of the query and high-degree polynomial in the size of the queried XML document [GKP05]. The existing papers devoted to improving XPath evaluation can be grouped into two main approaches, as is explained next (see e.g. [BK09] for a survey).

One idea, as used in e.g. [GKP02] and improved in [GKP03], is to use dynamic programming; see also [GKP05]. This gives evaluation algorithms that are polynomial (but not linear) in both the node test (we use this term for node selecting queries, although the terms predicate or filter are sometimes used in the literature) φ and the size of the document t. The best known algorithms for full XPath 1.0 [GKP03] have running time $O(|\varphi|^2|t|^4)$.

Another idea is to compile queries into finite-state tree automata, see [Nev02] for a survey. This approach works if the node test does not refer to attribute or text values (a fragment called CoreXPath), and therefore an XML document can be identified with a finitely labeled tree (the label of a node is its tag name). In this setting, an XPath node test can be compiled into a finite-state automaton; and this automaton can be evaluated on the tree in linear time. In general, the automaton may be exponential in the size of the query. (It is worth noting that using dynamic programming, one can evaluate CoreXPath node tests in time linear in both query and document, see [GKP05].)

This thesis, together with the conference papers on which it is based, [BP08], [Par09], [BP10], and [BP] can be seen as a generalization of the automata-theoretic framework to node tests that use attribute and text values. In the terminology of [BK09], we study a fragment of XPath called FOXPath (however without node identifiers). The first algorithm with linear time data complexity for this fragment was given in [BP08]. The constant in the linear time of this algorithm was exponential in the query size. However, the algorithm could handle an extension of XPath in which arbitrary regular expressions may appear as path expressions. We use the name *Regular XPath* for this extension of XPath, as opposed to *FOXPath*, which stands for XPath where path expressions are not allowed to use the Kleene star, as in the XPath specification [CD99]. The algorithm in [BP08] uses algebraic methods like finite monoids and Simon decompositions. We present here a different algorithm with the same complexity, which uses deterministic automata instead of monoids.

Then in [Par09], an algorithm with linear time data complexity and polynomial time combined complexity was given. This algorithm used the special form of path expressions in FOXPath, which in fact are less expressive than regular expressions. Hence the algorithm does not work for Regular XPath, only for FOXPath.

There is also a third, unpublished algorithm, which is a simpler version of these in [BP08] and [Par09]. It has $O(|t|\log|t|)$ time complexity in the document size $|t|$, polynomial combined complexity, and works for the whole Regular XPath as well. Probably among the four algorithms this one may be most useful in the practice. It is easier to understand and implement, which

probably compensates for the additional $\log|t|$ factor.

Finally, we have an algorithm presented in [BP10], which works for Regular XPath in time linear in the document size, and polynomial in the query size and the document height. Notice that a typical XML document (even very big) has a very small height.

The four algorithms described above are the content of this thesis. They are presented in the following theorem.

Theorem 1.1
Let t be an XML document and φ a node test of Regular XPath (as defined in Section 1.2.2). The set of nodes of t that satisfy φ can be computed in time

- $O(|\varphi|^3|t|\log|t|)$, *or*
- $O(2^{O(|\varphi|)}|t|)$, *or*
- *linear in $|t|$, polynomial in $|\varphi|$ and the height of t, or*
- *when φ is from FOXPath—in time $O(|\varphi|^3|t|)$.*

The theorem above talks about evaluating node tests. What about path expressions? In principle, path expressions can not be evaluated in time linear in the tree size, as sometimes quadratically many pairs satisfy a path expression. However it is possible to do the evaluation in time linear in the number of selected pairs or in the tree size, whatever is bigger. Even more, we give a constant delay algorithm: it finds some first pair satisfying α in time linear in the document, and each next pair in constant time. Hence, when someone wants to find just one pair, or just a linear number of pairs in the size of the document, this can be done in linear time.

Theorem 1.2
Let t be an XML document and α a path expression of Regular XPath. All pairs of nodes of t satisfying α can be computed one after another in time

- *first pair:* $O(|\alpha|^3|t|\log|t|)$, *each next pair:* $O(|\alpha|^3\log|t|)$, *or*
- *first pair:* $O(2^{O(|\alpha|)}|t|)$, *each next pair:* $O(2^{O(|\alpha|)})$, *or*
- *first pair: linear in $|t|$, polynomial in $|\alpha|$ and the height of t, each next pair: polynomial in $|\alpha|$ and the height of t, or*
- *when α is from FOXPath—first pair:* $O(|\alpha|^3|t|)$, *each next pair:* $O(|\alpha|^3)$.

The thesis is structured as follows. In the remaining part of Chapter 1, we present preliminary definitions, the data model, and we define the fragment of XPath considered in this thesis. We also present results of some experiments. In Chapters 3 and 4 we present solutions to two problems, which are parts of the XPath evaluation algorithm, but can be also seen separately. Then, in Chapter 5, we present an algorithm answering to Regular XPath node tests, i.e. we prove the first three variants of Theorem 1.1. Chapter 6 is devoted to the fourth variant of the theorem: we consider there node tests of FOXPath. In Chapter 7 we consider an extension of Regular XPath, which contains aggregation. Finally, in Chapter 8 we give an algorithm evaluating path expressions, i.e. we present a proof of Theorem 1.2.

1.1 Basic definitions and used facts

1.1.1 Trees

A *binary tree* is a tree in which every node has zero, one, or two children. We distinguish between left and right child, in particular if a node has one child, it is either left child or right child. Trees will be denoted by letters t, s. Nodes will be denoted by x, y, z. We write $x \leq y$ to denote that x is an ancestor of y. Whenever we use words descendant or ancestor, they need not to be proper.

We say that a tree *forms a word* if every its node has at most one child.

Let x and y be two nodes in a binary tree t. The *closest common ancestor* (CCA) of x and y is the (unique) node z that is an ancestor of both x and y, and has a minimal possible distance from x and y (equivalently, maximal level).

Throughout the thesis we use the following fact.

Fact 1.3
For a tree t, after preprocessing in time $O(|t|)$, we can answer, in time $O(1)$, queries of the form: given two nodes x and y,

1. *where is the closest common ancestor of x and y?*
2. *is x an ancestor of y?*

Harel and Tarjan [HT84] show an algorithm for queries of type 1 (a simpler algorithm doing the same was given later by Bender and Farach-Colton [BFC00]). Queries of type 2 follow immediately from queries of type 1: it is enough to check if the CCA of x and y is equal to x.

Notice that the fact becomes significantly simpler, if we ask for $O(|t| \log |t|)$ preprocessing time and $O(\log |t|)$ query time. We just keep from each node a pointer to the node 2^k edges above it, for each k. Then the closest common ancestor can be found in time $O(\log |t|)$ using some kind of the binary search algorithm.

Unranked trees

We rarely use also unranked trees. This are trees in which every node is allowed to have arbitrary number of children; the children are ordered. We have the following fact.

Fact 1.4
For an unranked tree t, after preprocessing in time $O(|t|)$, we can answer, in time $O(1)$, queries of the form: for two nodes $x < y$, which child of x is an ancestor of y?

Proof
This is a consequence of Fact 1.3. We unravel t into a binary tree s using the first child / next sibling encoding: the leftmost child of a node becomes its left child, while its next sibling becomes its right child. We perform the preprocessing of Fact 1.3 for s. Additionally we remember the rightmost child of each node. In the query step we have to find in s the closest common ancestor of y and the rightmost child of x, which is done by one query to Fact 1.3. □

1.1.2 Binary relations

The set of binary relations over a set Q is denoted R_Q. This set forms a monoid, where the monoid operation is relation composition. For two relations $r, s \in R_Q$, their composition is denoted by $r \circ s$, or simply rs. Moreover, for $P \subseteq Q$, we also use the notation

$$P \circ r = \{q : \exists p \in P, (p, q) \in R\}, \qquad \text{and}$$
$$r \circ P = \{p : \exists q \in P, (p, q) \in R\}.$$

We have the following trivial fact about complexities of these operations.

Fact 1.5
Given $r, s \in R_Q$ and $P \subseteq Q$, we can calculate

- *the composition $r \circ s$ in time*[1] $O(|Q|^3)$,
- *the sets $P \circ r$ and $r \circ P$ in time $O(|Q||P|)$, which is $O(|Q|^2)$ in general, and $O(|Q|)$ when $|P| = 1$,*
- *the transitive closure of the relation r in time $O(|Q|^3)$.*

[1] It can be done even in a better complexity, namely $O(|Q|^{2.38})$, by a more sophisticated algorithm, see [CW87].

1.1.3 Edge labelled trees

Throughout the next two chapters we consider binary trees with labels on edges. The set of such trees labeled by elements of a set A is denoted $etrees(A)$.

Assume we have a tree $t \in etrees(R_Q)$, for some set Q. Then for any its node x and its descendant y of such a tree t we define $val_t(x, y)$ as the composition of relations written on the simple path from x to y. When t is clear from the context, we simply write $val(x, y)$.

1.2 XML documents and XPath

1.2.1 Data model

In this section we define the data model. We represent an XML document as a binary tree, called a *data tree*. The tree is binary, i.e. a node may have two children: left and right, one child: left or right, or no children. Although an XML document is typically seen as an unranked tree, it can be also interpreted as a binary tree, using the first child / next sibling encoding: the leftmost child of a node becomes its left child, while its next sibling becomes its right child.

There are two reasons why we use binary trees. One reason is to simplify the complexity analysis: for many operations it is obvious that processing two children takes constant time, but it is less obvious that for many children it takes time proportional to their number. A second reason is more important: the horizontal axes of XPath do not correspond to any edge of an unranked tree; however each axis can be simulated by a combination of axes going along edges of a binary tree.

In a data tree there are three types of nodes: element nodes, attribute nodes and text nodes. Attribute and text nodes always have no left child (i.e. they are leaves in the unranked tree). Every element and attribute node is assigned a *label* which is a tag name or an attribute name, respectively, and which is taken from a finite alphabet. Text nodes do not have names, we assume that their label is `text`. We call the whole alphabet A—every node has a label from the set A. Moreover every node has a *string value*. A string value of an attribute node is the value of the corresponding attribute, which is a string. A string value of a text node is just a text. But, what causes some difficulties, to get the string value of an element node one has to *concatenate* the string values of all text node descendants of the left child of the element node,[2] in document order. The total length of all string values may be quadratic in the input size. So, the string values of element nodes are not remembered explicitly. Since in chapters about XPath most of the time we will be dealing with data trees, we will sometimes write tree instead of data tree. String values will be denoted by d.

Consider for instance the following XML document:

```
⟨a⟩
  ⟨b⟩abc⟨/b⟩xyz
  ⟨b at1 = "01" at2 = "0101"⟩⟨/b⟩
⟨/a⟩
```

The data tree representing this document uses labels $A = \{\mathtt{a}, \mathtt{b}, \mathtt{at1}, \mathtt{at2}, \mathtt{text}\}$. The first two are tag names, the next two attribute names and the last one is the special label for text nodes. The data tree is presented in Figure 1.2.1.

The size of a data tree is the number of nodes plus the sum of lengths of string values of its attribute and text nodes. This size measure is linear in the size of the text file representation, since the only difference is in the special characters like ⟨ or ".

[2] This stands for all text node descendants of the element node, when the document is interpreted as an unranked tree.

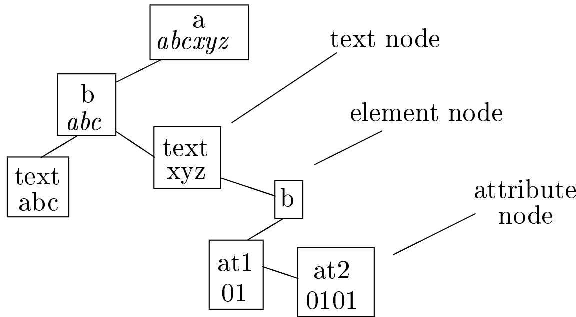


Figure 1.1: Example data tree (string values in italic are not remembered)

1.2.2 Regular XPath

In the thesis we mainly consider two fragments of XPath, called here *FOXPath* and *Regular XPath* (additionally in Chapter 7 we consider one more fragment, which allows counting). The FOXPath fragment is almost the fragment called FOXPath in [BK09]. Basically, it contains queries that may navigate in a tree and compare string values. The specification [CD99] of XPath 1.0 contains a lot of constructs, which can be easily added (like type conversions, etc.), but we omit them from this thesis to avoid going into technicalities. The constructs of full XPath 1.0 which are important for evaluation complexity, and which are not contained in Regular XPath, are: aggregates, manipulating integers and position arithmetic. The first two of them are addressed in Chapter 7.

The only difference between FOXPath and Regular XPath is that the second allows Kleene star. Regular XPath is not in the XPath specification, but it is an often considered extension. In this section we define these two fragments of XPath.

In XPath, the primitives employed for navigation along the tree structure are called *axes*. We consider the following *one-step* axes: `to−left`, `to−right` and their inverses `from−left`, `from−right`. They correspond to going to and from the left and the right child. We also have the transitive-reflexive closures of the one-step axes, called *multistep* axes: `to−left`*, `to−right`*, `from−left`*, `from−right`*, (`to−left`+`to−right`)*, (`from−left`+`from−right`)*. We comment on the relation to XPath with the original set of axes below.

There are two types of expressions: path expressions and node tests. We may look at them as on functions, for every node returning respectively: node sets and booleans. Another way for looking at a *path expression* is that it is a binary query. In each tree, a path expression will select a set of pairs (x, y) of nodes. Intuitively a path expression will describe the path from x to y, although the path might not be the shortest one. A typical path expression is `to−left`*, it selects a pair (x, y) if y can be reached from x by going several times to the left child, possibly $x = y$. A *node test* is a unary query: it selects a set of nodes. A typical node test is a, it selects nodes that have label a. In general in XPath, the two types of expression are mutually recursive, as defined below:

- Every label $a \in A$ is a node test, which selects nodes with a label a.
- Node tests admit negation, conjunction and disjunction.
- If α, β are path expressions, c is a string constant, and `RelOp` $\in \{=, \leq, <, >, \geq, \neq\}$, then

  $$\alpha \ \texttt{RelOp} \ \beta \qquad \text{and} \qquad \alpha \ \texttt{RelOp} \ c$$

  are node tests. The first of them selects a node x if there exist nodes y, z such that (x, y) is selected by α and (x, z) is selected by β and that the string values of y and z satisfy the relation `RelOp`. The second of them selects a node x if there exists a node y such that (x, y) is selected by α and the string value of y and the constant c satisfy the relation `RelOp`. The inequalities $\leq, <, >, \geq$ correspond to the lexicographic order of strings.

- There are two types of *atomic* path expressions. Every axis, including the multistep axes, is an atomic path expression. Furthermore, a node test φ may be interpreted as an atomic path expression $[\varphi]$, which holds in pairs (x, x) such that φ holds in x.

- In general, a path expression is a concatenation (composition) or union of simpler path expressions. In particular an empty concatenation is allowed, denoted ε. Moreover in Regular XPath (but not in FOXPath) a path expression may be a Kleene star of a simpler path expression.

Note that the operators $=$ and $\neq$ in node tests α `RelOp` β and α `RelOp` c are not mutually exclusive. A node may satisfy none or one or both of $\alpha = \beta$ and $\alpha \neq \beta$ (similarly for $<, \geq$, etc.). Note also that in Regular XPath the multistep axes are not necessary, as they can be expressed using a star and the one-step axes; this is not the case for FOXPath, since Kleene star is not available.

For a node test φ or a path expression α, by $|\varphi|$ and $|\alpha|$ we denote their size, understood as the length of their text representations.

Relation to XPath with the original set of axes

All standard axes, navigating in the unranked tree of a document, can be expressed by a combination of our axes. For example, the `child` axis is $\texttt{to-left} \cdot \texttt{to-right}^*$; the `ancestor` axis is $(\texttt{from-left} + \texttt{from-right})^* \cdot \texttt{from-left}$, and the `self` axis is ε (the empty path expression). Moreover, if the original expression does not use Kleene star, then our new expression does not as well (although of course it uses multistep axes).

1.3 Real world XPath systems

It was already shown in [GKP02, GKP05] that the working time of real-world XPath engines may be very bad, namely $O(|t|^{|\varphi|})$, where $|t|$ is the document size and $|\varphi|$ is the query size. As the experiments from these papers are already quite old, we repeat a part of them on the newest versions of XPath engines. In general, the results of our experiments show that nothing has changed.

We evaluate three XPath engines, namely Xalan, version 2.7.1, Saxon, version 9.3.0.4 (HE), and libxml2, version 2.6.31. These are some of the most popular freely available XPath engines. The first two of them are written in Java, and the last one is a C++ library. We have performed our experiments on a computer with 2.4 GHz Intel Core 2 processor and 4 GB RAM memory, running Linux.

For our experiments we generated simple, flat XML documents. Each document was of the form

$$\langle a \rangle \underbrace{\langle b/\rangle \ldots \langle b/\rangle}_{i \text{ times}} \langle /a \rangle.$$

For Saxon and Xalan we construct queries using the following simple pattern. The first query was '//a/b' (using the original XPath syntax). The i+1-th query was obtained by taking the i-th query and appending '/parent::a/b'. For instance, the third query was '//a/b/parent::a/b/parent::a/b'. For libxml2 we have used another queries. The first three were
//*[parent::a/child::*]
//*[parent::a/child::*[parent::a/child::*]]
//*[parent::a/child::*[parent::a/child::*[parent::a/child::*]]]
and then the sequence continues in the same way.

The results of the experiments are presented on Figures 1.2-1.4. They show that in all three cases the running time is indeed $O(|t|^{|\varphi|})$. Additionally, while running larger queries in Saxon, an „out of memory" error was occuring.

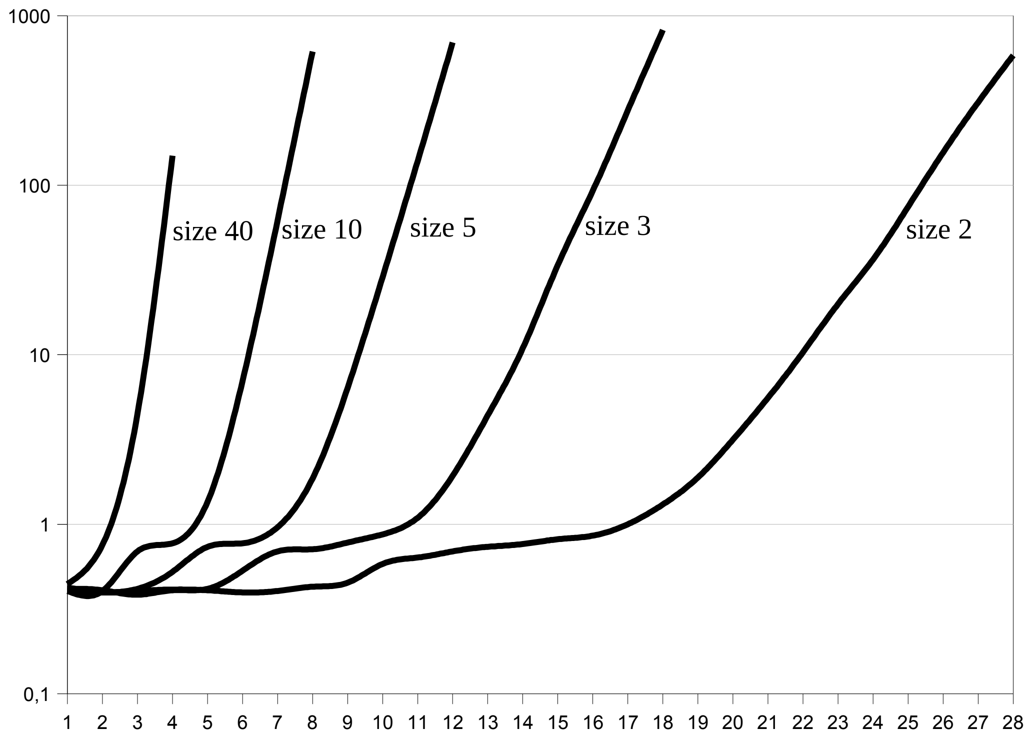


Figure 1.2. Running times of Xalan (in seconds), depending on query size (X axis) and document size (each line represents different document size)

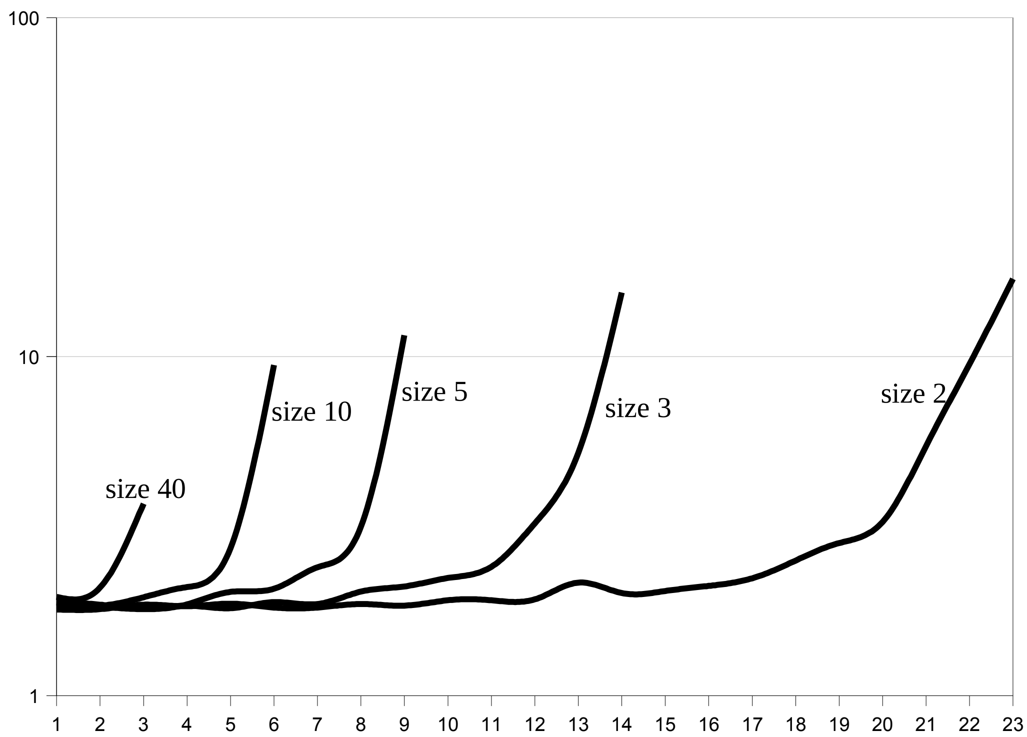


Figure 1.3. Running times of Saxon (in seconds), depending on query size (X axis) and document size (each line represents different document size)

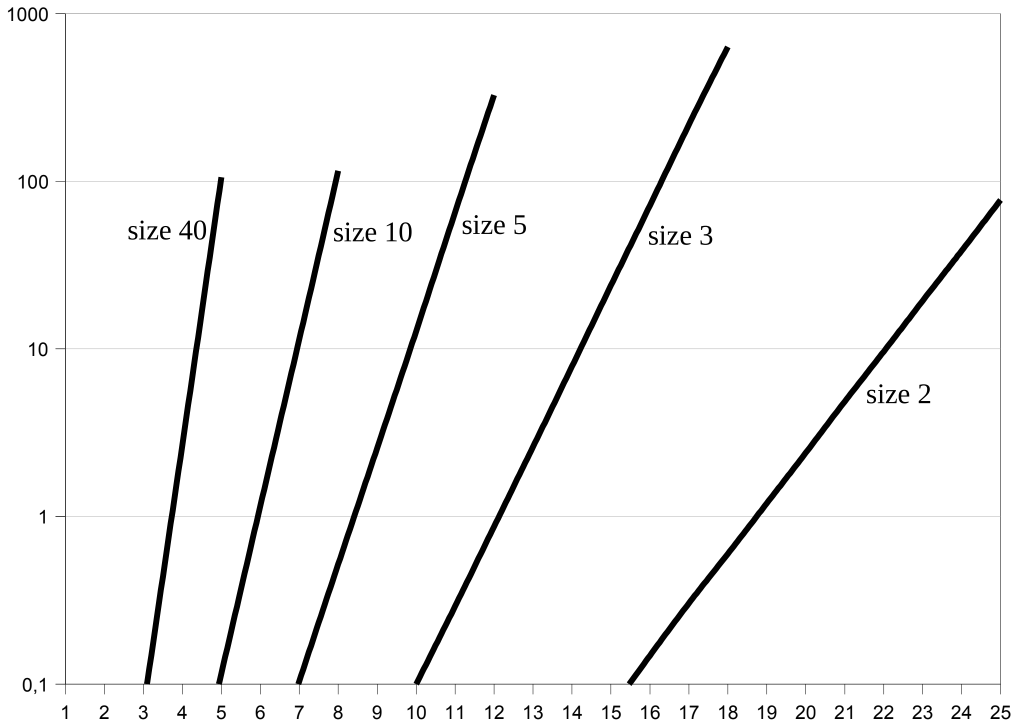


Figure 1.4. Running times of libxml2 (in seconds), depending on query size (X axis) and document size (each line represents different document size)

Chapter 2

Factorization forests

This chapter can be seen independently from the rest of the thesis. We present here the notion of factorization forests and we show a new algorithm calculating them (faster than previously known algorithm). This algorithm is then used in the next chapter (Section 3.3) to evaluate infix values for words (i.e. to prove the third variant of Theorem 3.1).

The basic object in this chapter are words, not trees.

Instead of specifying an infix by its first x and last position y, we use the set of all of its positions $F = \{x, x+1, \dots, y\}$. This way we can use set operations on infixes. If F is a set of positions in a word w, we write $w[F]$ for the subsequence of w consisting of positions from F, e.g. $a_1a_2a_3[\{1,2\}] = a_1a_2$. We use the name *factor* of w for a connected set of positions (all our sets of positions are connected), and the name *infix* for the word $w[F]$ when F is a factor. (Of course, the algorithms represent factors by just keeping the first and last position). We write x, y, z for positions, and F, G, H for factors.

In this chapter as an alphabet we use a finite monoid[1]. An *evaluation* of a word $w \in M^*$ is the element of M we get as the multiplication of all letters of w. For a factor F, the evaluation of $w[F]$ is denoted $val_w(F)$.

Factorization forests

A *factorization forest for a word* w is a family of factors that contains $\{x\}$ for every position x in w, and where every two factors are either disjoint, or one is contained in the other. There is a natural forest structure on the factors, so we can talk about descendants, parents, children and siblings, etc. The *level* of a factor is the number of its ancestors (including itself).

Suppose that $F_1, \dots, F_n$ are consecutive factors of a word $w \in M^*$ (i.e. the first position of F_{i+1} is the next position after the last position of F_i). A *collation* of these factors is any union of these factors that is also a factor, i.e. any $F_i \cup \dots \cup F_j$ for $i \le j$. We say that $F_1, \dots, F_n$ are homogeneous if all of their collations have the same value under val_w. A factorization forest is called homogeneous if any choice of at least three consecutive siblings is homogeneous. (It is important that we do not require this for only two consecutive siblings—otherwise for most words any factorization forest would not exist). In other words, in a homogeneous factorization forest we have two kinds of non-leaf factors: factors which have two children, and factors which have arbitrarily many homogeneous children.

First observe that it is very easy to get a factorization forest of logarithmic height: we split the whole word into two factors (of approximately the same length), then each of them into two smaller factors, and so on. As every factor has only two children, such forest is automatically homogeneous.

[1] Alternatively, we can use words over arbitrary alphabet A, and a morphism $\alpha\colon A^* \to M$.

Fact 2.1
For any word $w \in M$ we can find, in time $O(|w|)$, a homogeneous factorization forest for w of height at most $\log |w|$.

However the height may be constant in the word length. This is a beautiful result of Imre Simon, called the Factorization Forest Theorem [Sim90]. It says that, for each word over a monoid M, a homogeneous factorization forest of height $O(|M|)$ exists. From the proof, it is not difficult to see that the forest can be constructed in time linear in the word length and polynomial in the monoid size.

In this thesis, we study homogeneous factorization forests for words over R_Q, i.e. when M is the monoid of binary relations over a set Q. The size of the monoid R_Q is exponential in the size of Q. The main result is that we can build a factorization forest without worrying about this exponential blowup.

Theorem 2.2
For any word $w \in R_Q^*$ we can find, in time $O(|Q|^3|w|)$, a homogeneous factorization forest for w of height at most polynomial[2] in $|R_Q|$.

It is important, that the complexity in $|Q|$ is polynomial, not exponential. We describe the proof of this theorem in the next subsections.

2.1 Monoid of binary relations

For the rest of this section, we fix the monoid R_Q. We write r, s, t for the binary relations which are elements of R_Q.

Green's relations

Let r, s, t, t_1, t_2 below be elements of R_Q.

- r is called a prefix of s, written $r \geq_{\mathcal{R}} s$, if there is some t with $r \circ t = s$.
- r is called a suffix of s, written $r \geq_{\mathcal{L}} s$, if there is some t with $t \circ r = s$.
- r is called an infix of s, written as $r \geq_{\mathcal{J}} s$, if there are t_1, t_2 with $t_1 \circ r \circ t_2 = s$.
- If r is both a prefix and a suffix of s, we write $r \geq_{\mathcal{H}} s$.

These relations are called Green's relations; they come from [Gre51]. It is easy to see that each of Green's relations is a pre-order: it is both transitive and reflexive. The relations are not necessarily antisymmetric and therefore it makes sense to consider their connected components. For instance, we say that r and s are $\mathcal{R}$-equivalent, written $r \sim_{\mathcal{R}} s$, if both $r \geq_{\mathcal{R}} s$ and $s \geq_{\mathcal{R}} r$. An equivalence class is called an $\mathcal{R}$-class. Likewise for $\mathcal{L}$, $\mathcal{J}$ and $\mathcal{H}$.

In the algorithm, we will need to perform operations on R_Q in time $O(|Q|^3)$. One such operation is to calculate composition $r \circ s$, this is easy to do (see Fact 1.5). A problem that we will have to work around is that we do not know how to test $\mathcal{J}$-equivalence in time polynomial in $|Q|$. However, we can do this in some special cases, as stated in the following lemma.

Lemma 2.3
Given $r, s \in R_Q$, we can check in time $O(|Q|^3)$ if

$$r \circ s \overset{?}{\sim}_{\mathcal{J}} r, \qquad \textit{and if} \qquad r \circ s \overset{?}{\sim}_{\mathcal{J}} s.$$

To see this we need two auxiliary lemmas.

[2] The height can be even linear in $|R_Q|$, but it requires more care in the proof.

Lemma 2.4
For $r, s \in R_Q$ the two equivalences below hold:

$$r \circ s \sim_{\mathcal{J}} r \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{R}} r \qquad \textit{and} \qquad r \circ s \sim_{\mathcal{J}} s \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{L}} s.$$

Proof
This is a classic fact from the theory of Green's relations, but we prove it here for the sake of completeness. We only prove the part concerning $\mathcal{R}$-classes, namely

$$r \circ s \sim_{\mathcal{J}} r \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{R}} r.$$

The proof for $\mathcal{L}$-classes is the same. Only the implication from left to right is nontrivial. By the assumption $r \circ s \sim_{\mathcal{J}} r$ there must be some $t, u \in M$ such that

$$r = t \circ r \circ s \circ u$$

By substituting n times the right side instead of r, we get

$$r = t^n \circ r \circ (s \circ u)^n$$

If we choose n so that $(s \cdot u)^n$ is idempotent (this is always possible in a finite monoid), we get

$$r = t^n \circ r \circ (s \circ u)^n = t^n \circ r \circ (s \circ u)^n \circ (s \circ u)^n = r \circ (s \circ u)^n$$

which shows that $r \circ s$ is a prefix of r, and hence $r \circ s \sim_{\mathcal{R}} r$. □

Lemma 2.5
Let r_1, r_2 be two elements of R_Q. We define

$$Q_1(q_2) = \{q_1 \colon r_1 \circ \{q_1\} \subseteq r_2 \circ \{q_2\}\}.$$

It holds $r_1 \geq_{\mathcal{R}} r_2$ if and only if $r_1 \circ (Q_1(q_2)) = r_2 \circ \{q_2\}$ for each $q_2 \in Q$.

Proof
First assume that $r_1 \geq_{\mathcal{R}} r_2$, i.e. $r_1 \circ r = r_2$ for some $r \in R_Q$. Fix some element $q_2 \in Q$. Of course $r_1 \circ (Q_1(q_2)) \subseteq r_2 \circ \{q_2\}$ because $q_1 \in Q_1(q_2)$ only if $r_1 \circ \{q_1\} \subseteq r_2 \circ \{q_2\}$. Now take $q \in r_2 \circ \{q_2\}$, which means that $q \in r_1 \circ \{q_1\}$ for some $q_1 \in r \circ \{q_2\}$. But then $r_1 \circ \{q_1\} \subseteq r_2 \circ \{q_2\}$, so $q_1 \in Q_1(q_2)$ and $q \in r_1 \circ (Q_1(q_2))$.

For the other direction assume that $r_1 \circ (Q_1(q_2)) = r_2 \circ \{q_2\}$ for each $q_2 \in Q$; we need to find $r \in R_Q$ such that $r_1 \circ r = r_2$. Let r contain pairs (q_1, q_2) such that $q_1 \in Q_1(q_2)$. Then for any $q_2 \in Q$ it holds

$$(r_1 \circ r) \circ \{q_2\} = r_1 \circ r \circ \{q_2\} = r_1 \circ (Q_1(q_2)) = r_2 \circ \{q_2\},$$

which shows that $r_1 \circ r = r_2$. □

Proof (of Lemma 2.3)
Lemma 2.4 shows that all we need to do is to test $\mathcal{R}$-equivalence and $\mathcal{L}$-equivalence. Lemma 2.5 give a criterion for deciding whether $r_1 \geq_{\mathcal{R}} r_2$, which may be checked in time $O(|Q|^3)$. $\mathcal{L}$-equivalence is done the same (symmetric) way. □

Calculating compositions and performing checks from Lemma 2.3 is our only interface to the monoid R_Q. In particular if we consider a monoid M where these operations can be performed in time T, then our algorithm computes a factorization forest in time $O(T|w|)$.

2.2 First step: $\mathcal{J}$-homogeneous forest

Proof strategy

We present the proof strategy for Theorem 2.2.

The definition of homogeneous factors or factorization forests requires equality of some monoid elements (values of collations of factors). By weakening this requirement we define notions of $\mathcal{J}$-homogeneity and $\mathcal{H}$-homogeneity. Let $F_1, \ldots, F_n$ be consecutive factors. We say the factors are $\mathcal{J}$-homogeneous if their collations have $\mathcal{J}$-equivalent values under val_w. Likewise we define a $\mathcal{J}$-homogeneous factorization forests, and the same for $\mathcal{H}$.

Our proof strategy is to first compute a $\mathcal{J}$-homogeneous factorization forest, then upgrade it to an $\mathcal{H}$-homogeneous one, and then upgrade that one to a homogeneous one. The main difficulty is in the first step – computing a $\mathcal{J}$-homogeneous forest; we do this below in Lemma 2.6. The other steps are done using basically the same techniques as in the proof of the factorization forest theorem from [Kuf], or the proofs of [Sim90, Col10].

Lemma 2.6
Let $w \in R_Q^*$. One can compute a $\mathcal{J}$-homogeneous factorization forest $\mathcal{F}$ in time $O(|Q|^3|w|)$. The forest has height linear in $|R_Q|$.

The rest of the section is devoted to proving this lemma. The algorithm processes word positions from left to right. We begin by describing the invariant.

The invariant

After processing position x, the algorithm will have computed a factorization forest $\mathcal{F}_x$ for the prefix $1, \ldots, x$. With each factor we remember one additional bit: if the factor is *open* or *closed*. All open factors have to contain the last processed position x. Open factors might grow when processing new positions. Once a factor becomes closed, it does not change. All singleton factors are closed. Suppose $F_1, \ldots, F_n \in \mathcal{F}_x$ is a maximal set of siblings (written from left to right). The invariant is that they satisfy the following property $\star$:

- $\star$ The factors $F_1, \ldots, F_{n-1}$, and the factor $F_1 \cup \cdots \cup F_{n-1}$ are all $\mathcal{J}$-equivalent.

Additionally, when they are children of an open factor $F \in \mathcal{F}_x$, the following property $\star\star$ is satisfied:

- $\star\star$ F and F_1 are $\mathcal{J}$-equivalent.

The invariant is satisfied by the initial configuration $\mathcal{F}_1 = \{\{x\}\}$.

Once we have processed the whole word, it is not difficult to get a $\mathcal{J}$-homogeneous factorization forest from the one produced by the algorithm. For each maximal set of siblings $F_1, \ldots, F_n \in \mathcal{F}_x$, it is enough to add a factor $F_1 \cup \cdots \cup F_{n-1}$.

Updating the forest

Suppose we have computed $\mathcal{F}_{x-1}$, and we want to compute $\mathcal{F}_x$. Consider the factors open in $\mathcal{F}_{x-1}$:

$$x - 1 \in F_1 \subsetneq F_2 \subsetneq \cdots \subsetneq F_n.$$

There are also closed factors containing $x-1$, at least one: $\{x-1\}$. Let C be the biggest of them. We obtain $\mathcal{F}_x$ from $\mathcal{F}_{x-1}$ as follows.

- Add $\{x\}$.
- If C and F_1 are not $\mathcal{J}$-equivalent, or $n = 0$, add open factor $G_0 = C \cup \{x\}$.
- Replace the factors F_i by $G_i = F_i \cup \{x\}$, for $i \in \{1, \ldots, n\}$.

- When $G_i \backslash \{x\}$ and G_i are not $\mathcal{J}$-equivalent close G_i, for every i (i.e. for $i \in \{0, \dots, n\}$ if G_0 was added, and for $i \in \{1, \dots, n\}$ otherwise).

The test on $\mathcal{J}$-equivalence in the second and the last step is done using Lemma 2.3, since we are testing $\mathcal{J}$-equivalence of a factor and its suffix or prefix. Below we argue that the invariant is preserved. Then, we show why the algorithm runs in the required time, and why the factorization forest has height linear in R_Q.

Correctness

Extending a factor does not impact on property $\star$, as it does not talk about a last sibling. Property $\star$ has to be checked only for the siblings of the newly added factor $\{x\}$. If G_0 is created, $\{x\}$ has only one sibling, so $\star$ is satisfied. Otherwise C is no longer the last sibling. This happens only when C is $\mathcal{J}$-equivalent to its parent F_1. As F_1 is open, it is $\mathcal{J}$-equivalent to its first child (from $\star\star$), hence to all its children (from $\star$), which gives $\star$ in the new forest.

Now let check the property $\star\star$ for open factors. Factor G_0 stays open only when G_0 and $G_0 \backslash \{x\} = C$ are $\mathcal{J}$-equivalent, which is exactly $\star\star$. Any other G_i stays open when it is $\mathcal{J}$-equivalent to F_i, which (from $\star\star$) is equivalent to its first child (which is also the first child of G_i).

Running time

A potential problem is the last step. Potentially we have to do n tests for $\mathcal{J}$-equivalence. However notice that when $G_i \backslash \{x\}$ and G_i are $\mathcal{J}$-equivalent for some i, then they are $\mathcal{R}$-equivalent (Lemma 2.4), hence also $G_j \backslash \{x\}$ and G_j are $\mathcal{R}$-equivalent ($\mathcal{J}$-equivalent) for any $j > i$. Thus we may stop testing greater i when we detect an equivalence. The number of tests for $\mathcal{J}$-equivalence is bounded by the number of factors becoming closed (plus one). Since the total number of factors in a factorization forest is at most twice the length of the word, we have a limit on the total number of operations in the last step of the algorithm.

Two implementation problems remain. First, where do we get the images of the factors $F_1, \dots, F_n$ that are used in the tests for $\mathcal{J}$-equivalence? The answer is that our algorithm maintains for each open factor F_i, the image of its closed part $F_i - F_{i-1}$. Second, what is the cost of adding x to the factors F_i? The answer is that this can achieved for free, if we do not store the ends of open factors, but we only keep in mind that they all end in the currently processed position x.

The dependence in $|Q|$ is $O(|Q|^3)$, as each single step is either done in constant time, when it manipulates the forest, or in $O(|Q|^3)$, when we compose some elements of R_Q or when we perform checks from Lemma 2.3.

Height of the forest

Why is the height of the factorization forest linear in R_Q? It would be useful to look at the $\mathcal{J}$-class of the first child of each non-singleton factor. The following invariant is preserved by the algorithm: whenever a factor F in the factorization forest is the parent of a non-singleton factor G, then the first child of F has a strictly smaller $\mathcal{J}$-class than the first child of G. It guarantees that the level of a factor is bounded by the position of its first child in the $\leq_{\mathcal{J}}$ order.

Why is the invariant satisfied? First observe an auxiliary property of the forest: every closed factor in the factorization forest (except singletons) has a different (smaller) $\mathcal{J}$-class than its first child. Indeed, when a factor G_i becomes closed, it has a different $\mathcal{J}$-class than $G_i \backslash \{x\}$, which contains the first child of G_i.

To prove the invariant notice that during execution of the algorithm, the first child of a factor is never modified. Hence it is enough to analyze each moment when a new pair of a parent and its child is created. It happens only in the second step, when G_0 is created (creating $\{x\}$ does not matter, as the invariant does not talk about singleton factors). First compare G_0 with its only non-singleton child C. As C is closed, from the above we know that its first child has greater

$\mathcal{J}$-class than C itself, which is the first child of G_0. Now compare G_0 with its parent G_1. The factor G_0 is created only when C (the first child of G_0) has greater $\mathcal{J}$-class than F_1. Because F_1 is open, from $\star\star$ we get that it is $\mathcal{J}$-equivalent with its first child (which is also the first child of G_1).

2.3 From a $\mathcal{J}$-homogeneous forest to a homogeneous one

In this section we finish the proof of Theorem 2.2. Recall that thanks to Lemma 2.6, we have a $\mathcal{J}$-homogeneous factorization forest for our input word, which has height linear in $|R_Q|$.

Our proof is in two steps. First, we upgrade the $\mathcal{J}$-homogeneous factorization forest to an $\mathcal{H}$-homogeneous one. Then, we upgrade the $\mathcal{H}$-homogeneous factorization forest to a homogeneous one required by Theorem 2.2.

First we state a technical lemma, which will be used in both steps.

First-letter-homogeneous factorization forests

As a tool in the proof, we show that some other kind of factorization forests can be efficiently computed. We say that factors $F_1, \ldots, F_n$ are *first-letter-homogeneous*, when the first letter of each of these factors is the same. We say that a factorization forest is *first-letter-homogeneous* if any choice of at least three consecutive siblings is first-letter-homogeneous.

Lemma 2.7
Let A be a set whose elements can be represented using k bits. For each word $w \in A^*$, in time $O(k|w|)$ we can compute a first-letter-homogeneous factorization forest of height at most $2|A|$.

Proof
For $a \in A$, consider the set X_a of positions in the word w which are labeled by a. We want to represent each nonempty set X_a by a list. Then for each position x in the word w we can find the next position having the same letter as x, in constant time. To find such representation it is enough to sort all pairs (letter, position) using any order on letters. When it is the lexicographic order on bit representations, we may use the lexicographic sorting, which works in time $O(k|w|)$.

We construct a factorization forest $\mathcal{F}$ as follows. First we add to it the factor containing the whole word. Then each newly added non-singleton factor F is split in the following way. Let a be the first letter of $w[F]$, and let $x_1, \ldots, x_n$ be the positions in F having the same letter a (in particular x_1 is the first position of F). As observed above, these positions can be found in time $O(n)$. When $n > 1$, for each $1 \leq i < n$ we add to $\mathcal{F}$ the factor from x_i to $x_{i+1} - 1$, and we add the factor from x_n to the end of F. When $n = 1$ we add to $\mathcal{F}$ the singleton factor containing x_1, and the factor containing all the other positions of F.

Directly from the construction we see that $\mathcal{F}$ is a first-letter-homogeneous factorization forest. The processing take time linear in the size of $\mathcal{F}$, hence linear in $|w|$. It remains to bound the height of $\mathcal{F}$. Notice that if a factor F is divided in the first way (i.e. it has $n > 1$), then its children are divided in the second way: their first letter does not appear anywhere else. But when we divide in the second way, the number of different letters appearing in the second child is strictly smaller than the number of different letters in the parent. Thus the height is at most $2|A|$. □

From a $\mathcal{J}$-homogeneous forest to an $\mathcal{H}$-homogeneous one

We first present an auxiliary lemma. A *partial factorization forest* is defined like a factorization forest, but its leaves need not be singletons. We also require that any factor is the union of its children, a requirement which is redundant when leaves are singletons.

Lemma 2.8
Let $F_1, \ldots, F_k$ be consecutive factors that are $\mathcal{J}$-homogeneous. In time $O(|Q|^2 k)$, we can construct an $\mathcal{R}$-homogeneous partial factorization forest with leaves $F_1, \ldots, F_n$ and height at most $|R_Q|$.

Proof
Let F be the union $F_1 \cup \cdots \cup F_k$. We will treat the factors $F_1, \ldots, F_k$ as letters in a word

$$v = r_1 \cdots r_k \in (R_Q)^*,$$

where r_i is the value of F_i. Apply Lemma 2.7 to the word v, yielding a factorization forest $\mathcal{G}'$. By expanding the i-th letter of v to the factor F_i, we can convert the factorization forest $\mathcal{G}'$ into a partial factorization forest $\mathcal{G}$ with leaves $F_1, \ldots, F_k$ and root F.

We claim that $\mathcal{G}$ satisfies the statement of the lemma. Its height is at most $|R_Q|$ by Lemma 2.7, so we only need to show that it is $\mathcal{R}$-homogeneous.

Consider a set of at least three siblings $G_1, \ldots, G_m$ in $\mathcal{G}$. From the way $\mathcal{G}$ was constructed, we know that for each $i \in \{1, \ldots, n\}$ the value of G_i is the same. Take any collation $G_i \cup \cdots \cup G_j$ of these factors. Note that G_i is a prefix of $G_i \cup \cdots \cup G_j$, and both are $\mathcal{J}$-equivalent. Consequently, by Lemma 2.4, they must be $\mathcal{R}$-equivalent. It follows that the collations of $G_1, \ldots, G_m$ are all $\mathcal{R}$-equivalent. □

We use the above lemma to upgrade a $\mathcal{J}$-homogeneous factorization forest $\mathcal{F}$ to an $\mathcal{R}$-homogeneous one, call it $\mathcal{G}$. We will simply add factors to $\mathcal{F}$. Initially, $\mathcal{G} = \mathcal{F}$. We process each maximal set of siblings $\mathcal{S} = \{F_1, \ldots, F_k\}$ from $\mathcal{F}$. (In most cases, $\mathcal{S}$ consists of all the children of a common parent, the exception is when $\mathcal{S}$ contains the roots of $\mathcal{F}$.) If $\mathcal{S}$ has at most two factors, we do not need to do anything. Otherwise, by assumption on $\mathcal{J}$-consistency of $\mathcal{F}$, we can apply the above lemma to the factors in $\mathcal{S}$, and add all factors of the resulting factorization forest $\mathcal{G}_\mathcal{S}$ to $\mathcal{G}$. Note that the added factors from $\mathcal{G}_\mathcal{S}$ are all included in $\bigcup \mathcal{S}$, so $\mathcal{G}$ is a factorization forest. The processing time needed to compute $\mathcal{G}$ is linear in the number of factors in all the sets $\mathcal{S}$, which is simply the number of factors in $\mathcal{F}$. Finally, if $\mathcal{G}$ contains a set of at least three siblings, then these siblings were added in some $\mathcal{G}_\mathcal{S}$, and hence they all have the same $\mathcal{R}$-class.

By a symmetric argument we upgrade the factorization forest $\mathcal{G}$ to an $\mathcal{L}$-homogeneous one, call it $\mathcal{H}$. But $\mathcal{H}$ is also $\mathcal{R}$-homogeneous, as already $\mathcal{G}$ was such. (If a factorization forest $\mathcal{G}$ is $\mathcal{R}$-homogeneous, and a factorization forest $\mathcal{H}$ contains more factors than $\mathcal{G}$, then $\mathcal{H}$ is also $\mathcal{R}$-homogeneous.) Thus $\mathcal{H}$ is $\mathcal{H}$-homogeneous.

From $\mathcal{H}$-homogeneouity to homogeneouity

In this section we show how to upgrade an $\mathcal{H}$-homogeneous factorization forest to an homogeneous one. The structure of the proof is the same as in the previous case, we only need a new version of Lemma 2.8.

Lemma 2.9
Let $F_1, \ldots, F_k$ be consecutive factors that are $\mathcal{H}$-homogeneous. In time $O(|Q|^3 \cdot k)$, we can construct an homogeneous partial factorization forest with leaves $F_1, \ldots, F_n$ and height at most $|R_Q|$.

Proof
We use the same approach as in Lemma 2.8. Let F be $F_1 \cup \cdots \cup F_k$. Let $r(F_i) = F_i \cup \cdots \cup F_k$. We treat each of the factors $F_1, \ldots, F_k$ as a letter in a word

$$v = r_1 \cdots r_k \in (R_Q)^*$$

where r_i is the value of $r(F_i)$. Apply Lemma 2.7 to the word v with $M = R_Q$, yielding a factorization forest $\mathcal{G}'$. Replacing each letter r_i by the factor F_i, we convert $\mathcal{G}'$ into a partial factorization forest $\mathcal{G}''$ with leaves $F_1, \ldots, F_k$. Then for each maximal set of siblings $G_1, \ldots, G_m$ (for $m > 2$) we add the factor $G_1 \cup \cdots \cup G_{m-1}$, getting a partial factorization forest $\mathcal{G}$.

We claim that $\mathcal{G}$ is homogeneous. We argue as in Lemma 2.8: consider a maximal set of at least three siblings. The only possibility is that these are $G_1, \ldots, G_{m-1}$ among some maximal set of siblings $G_1, \ldots, G_m$ from $\mathcal{G}''$. From the way $\mathcal{G}''$ was constructed, we know that for each i the value of $r(G_i)$ is the same. Let us write $g_1, \ldots, g_m$ for the values of $G_1, \ldots, G_m$; these satisfy

$$g_i \circ r(G_{i+1}) = r(G_i) = r(G_{i+1}) \qquad \text{for all } i < m.$$

The following well known lemma on Green's relations completes the proof of Lemma 2.9, since it shows that all the elements $g_1, \ldots, g_{m-1}$ must be equal, as they all represent the group identity. (In a group, if $g \circ h = h$ holds, then g must be the group identity.) $\square$

Fact 2.10
Let H be a $\mathcal{H}$-class in a finite monoid M. If there exist $s, t \in H$ such that $s \cdot t \in H$, then H is a group.

Proof
Take $a, b \in H$ such that $a \cdot b \in H$ and take any $d \in H$. Since $b \sim_{\mathcal{R}} d$, then $a \cdot b \sim_{\mathcal{R}} a \cdot d$, so even more $d \sim_{\mathcal{J}} a \cdot b \sim_{\mathcal{J}} a \cdot d$. On the other hand $a \cdot b \sim_{\mathcal{L}} d$ (because both are in H) and $d \sim_{\mathcal{L}} a \cdot d$ (from Lemma 2.4). Hence $a \cdot b \sim_{\mathcal{R}} a \cdot d$ and $a \cdot b \sim_{\mathcal{L}} a \cdot d$, so $a \cdot d \in H$. Symmetrically we may show that if $a \cdot d \in H$ for some $a, d \in H$, then also $c \cdot d \in H$ for any $c \in H$. This shows that $c \cdot d \in H$ for any $c, d \in H$.

Take any $a \in H$. Since M is finite it has to be $a^n = a^{2n}$ for some positive n. It holds $a^n \in H$. Denote a^n as $\mathbf{1}_H$ (for some fixed a). We have $\mathbf{1}_H \cdot \mathbf{1}_H = \mathbf{1}_H$. This will be the neutral element in H. Indeed, take any $b \in H$. We may write $b = m \cdot (b \cdot \mathbf{1}_H)$ for some $m \in M$. Then $b = m \cdot b \cdot \mathbf{1}_H = m \cdot b \cdot \mathbf{1}_H \cdot \mathbf{1}_H = b \cdot \mathbf{1}_H$. Symmetrically $\mathbf{1}_H \cdot b = b$.

To conclude that H is a group it is enough to show that each element has an inverse. Take any $a \in H$. For some positive m it holds $a^m = a^{2m}$. As above $b \cdot a^m = b$ for any $b \in H$. So $a^m = \mathbf{1}_H \cdot a^m = \mathbf{1}_H$, hence a^{m-1} is an inverse of a. $\square$

Chapter 3

Fast evaluation of paths

Assume we have a binary tree t with edges labelled by binary relations over some set Q. In this section we show that after appropriate preprocessing for such tree t we can quickly answer queries about $val_t(x, y)$ for any node x and its descendant y (recall that $val_t(x, y)$ is the composition of the relations written on the simple path from x to y). Here quickly means in time constant in the size of the tree, or at most logarithmic.

Theorem 3.1
For a binary tree $t \in etrees(R_Q)$ we can, after preprocessing, answer queries of the form: for any node x and its descendant y compute $val_t(x, y)$. This can be done in time

- *preprocessing: $O(|Q|^3|t| \log |t|)$, query: $O(|Q|^3 \log |t|)$, or*
- *preprocessing: $O(2^{O(|Q|)}|t|)$, query: $O(2^{O(|Q|)})$, or*
- *when t forms a word—preprocessing: $O(|Q|^5|t|)$, query: $O(|Q|^5)$.*

This theorem will be used later for XPath evaluation. It gives also a quick method of calculating runs of an automaton reading paths of a tree. Assume we have an automaton $\mathcal{A}$ with states Q and input alphabet A. Assume that we also have a binary tree t_A with edges labelled by elements of A. Using the above theorem we can quickly answer queries of the form: for any node x and its descendant y does $\mathcal{A}$ accept the word written on the path from x to y. Indeed, in the preprocessing stage we create a tree t, which has the same nodes as t_A, but on each edge, instead of a letter $a \in A$ it has the transition relation of $\mathcal{A}$ while reading the letter a. This is a tree with edges labelled by elements of R_Q, we make the preprocessing stage of Theorem 3.1 for it. Now observe that $val_t(x, y)$ for any node x and its descendant y is the transition relation of the automaton while reading the word written on the path from x to y. So during the query it is enough to calculate $val_t(x, y)$ and check if it contains a pair with an initial state on the first coordinate and an accepting state on the second coordinate.

In the following sections we prove the three variants of this theorem, using three different techniques.

3.1 Logarithmic querying

In this section we show the first variant of Theorem 3.1. This is a straightforward divide and conquer approach.

Fix a set Q and a binary tree t with edges labelled by elements of R_Q. First, for each node remember its level (distance from the root). Let K be the greatest number such that 2^K is not greater than the height of the tree t. It holds $K = O(\log |t|)$. For every node y of t and every $0 \leq k \leq K$ we remember a pointer to its ancestor x which is 2^k edges above y. Together with the pointer we remember $val_t(x, y)$. This information can be easily calculated in time $O(|Q|^3|t|K)$: to

find a node 2^k edges above from y, we twice go 2^{k-1} edges up using previously calculated pointers. Also $val_t(x, y)$ is the composition of these values remembered for $k-1$.

Now consider a query step: consider any node x and its descendant y. Consider the nodes $y = x_0 > x_1 > \cdots > x_n = x$, where x_{i+1} is 2^k edges above x_i for the greatest number k such that $x_{i+1} \geq x$. In other words we go from y to x using our pointers: we always use a pointer to the highest ancestor which is still a descendant of x. Recall that with each node we also remember its level in the tree, so finding this sequence of nodes is easy. At each step we use smaller k, so it holds $n \leq K+1$. For each i we have $val_t(x_{i+1}, x_i)$ written in our data structure. To get $val_t(x, y)$ it is enough to compose them all; this takes time $O(|Q|^3 \log |t|)$.

3.2 Tape construction

In this section we solve the same problem as in the previous section, but in better complexity in the tree size: we eliminate the $\log |t|$ factor. Unfortunately, the complexity in $|Q|$ becomes exponential.

In Subsection 3.2.1, we describe the main idea, which we call the tape construction. An immediate application of the construction is a fast string-matching algorithm, as described below. Fix a regular word language $L \subseteq A^*$, recognized by a deterministic automaton $\mathcal{D}$. For any word $a_1 \cdots a_n \in A^*$ one can do a preprocessing stage in time $O(|\mathcal{D}|n)$ (linear in the word length), such that later on, any query $a_i \cdots a_j \in L$? can be answered in time $O(|\mathcal{D}|)$ (not depending on n or $j-i$). Then, in Subsection 3.2.2 we show how the results can be applied in a tree and we prove the second variant of Theorem 3.1. The tape construction is used also in Section 4.2, where we prove another theorem.

3.2.1 Tape construction for words

We use deterministic automata. Such an automaton is denoted by letter $\mathcal{D}$, its set of states by letter D and its particular states by letter d (we use a non-standard notation to distinguish them from states q of a nondeterministic automaton $\mathcal{A}$). The input alphabet of such an automaton is denoted by A.

Consider a word $w = a_1 \cdots a_n \in A^*$. A *node in w* is any number $i = 0, \ldots, n$, which is identified with the space between position i and $i+1$. So we think about a word in a way that the letters are written on the edges of a path connecting $n+1$ nodes. (This definition is meant to be extended to trees with letters on edges.)

Given nodes $x \leq y$ in such a word, the *word from x to y in w* consists of the letters $a_{x+1} \cdots a_y$. In other words, these are the letters that are on the path between x and y. In particular, the word from x to x is the empty word. By $run_w(x, d, y)$ we denote the state of the automaton $\mathcal{D}$ after reading the word from x to y, assuming that it begins in state d in node x (note that there is exactly one such state $run_w(x, d, y)$, as $\mathcal{D}$ is deterministic).

Let $K = |D|$. For an input word, we will create K *tapes*, numbered from 1 to K, on which we will be writing runs of the automaton. More precisely, we create a two-dimensional array, indexed by a tape number (rows) and by a node number (columns). In each cell of this array we remember two pieces of information. First, each cell stores a state of $\mathcal{D}$. In each node, every tape stores a different state, so every state appears in some tape. Second, the cell stores the number j of some tape, possibly j is undefined. If at node x on the i-th tape a number j is written, we say that the i-th tape *joins* the j-th tape at that node and that the i-th tape is *reset*. If there is no number, we say that this tape is not reset at that node. We define the contents of the tapes by an algorithm, which for each node, from the first to the last, does the following, see Figure 3.1 for an illustration.

1. If we are at the first node we write the states on the tapes arbitrarily (but preserving the rule that on each tape there is a different state).

2. Otherwise, let $d_1, \ldots, d_K$ be the states written on tapes $1, \ldots, K$ at the previous node (they are already calculated). Let a be the letter written on the edge between the previous and

the current node.

3. To each of these states we apply the transition function of the automaton, using input letter a. We get some states $d'_1, \ldots, d'_K$ (i.e. for each i the automaton goes from d_i to d'_i when reading a). Some of these states might become equal.

4. When some state d'_i is not equal to any of the earlier states $d'_1, \ldots, d'_{i-1}$, we write it on its tape and we remember that this tape is not reset.

5. For each other i, we take the smallest $j < i$ such that $d'_i = d'_j$ (in other words: j such that d'_i is already written on the j-th tape). We remember that the i-th tape joins the j-th tape at that node.

6. All the other states, which are not listed in $d'_1, \ldots, d'_K$, are written on the reset tapes (in an arbitrary order).

The contents of the tapes can be calculated in one left-to-right pass through the word; when it is done carefully, it takes time $O(|D||w|)$. Additionally at each node we remember a pointer to the closest node to the right where this tape is reset (or that there is no such node). This can be calculated in one right-to-left pass.

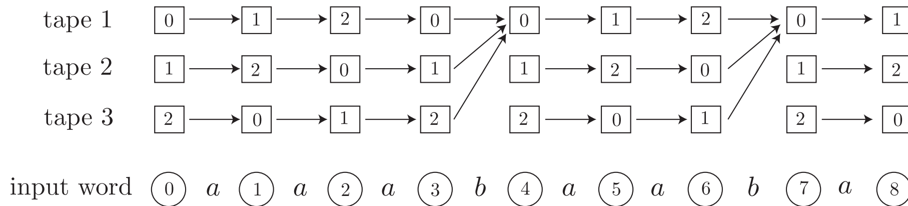


Figure 3.1: The tape construction. In this example, the automaton $\mathcal{D}$ has input alphabet $\{a, b\}$ and its state $d \in \{0, 1, 2\}$ holds the number of a's since the last b, modulo 3. The arrows show which tape joins which tape. Note how in node 4 (also in node 7), both tapes 2 and 3 join tape 1.

Consider a run of $\mathcal{D}$ starting in a state d at some node x, and ending in some position $y > x$. We find the tape i_1 on which this state is written. Then the run is written on that tape until the tape joins another tape i_2. It is important that $i_2 < i_1$, as a tape may only join an earlier tape. Then the run is written on i_2, until it joins tape $i_3 < i_2$ and so on. When position y is reached, the run is on some tape i_k, with $k < K$. The tape number i_k can be determined by following, k times, the pointers to the resets, which are stored in the data structure. (Each time we follow such a pointer, we test if the reset is still before y.) To find the state reached at node y, it is enough to read the state on tape i_k. Summing up, we can determine the state in y in time $O(K) = O(|D|)$.

3.2.2 Tapes in a tree

Fix a deterministic automaton $\mathcal{D}$, an alphabet A and a binary tree t with a label from A on every edge. In this section we use the tape construction to find the value of runs of the automaton on downward paths in t.

We extend the mapping run_w to trees in the following way. For two nodes $x \leq y$ in the tree t, the *word from x to y* is obtained by reading the labels on the (shortest) path from x to y. The mapping run_t is defined analogously to the word case.

We do the tape construction on each path from the root to some node. The contents of the tapes (and places where a tape joins some other tape) depend only on a prefix of a word. So the tapes can be calculated by doing a single top-down pass through the tree, we will be using this heavily later on.

We have to modify slightly which pointers are remembered, as keeping pointers to the next place where the tape is reset may be too costly. Instead we keep the following information for each $1 \leq k \leq K$:

A. A tree s_k consisting of nodes x at which the k-th tape is reset (a node is a child in s_k of another node if it is its proper descendant in t and at no node between them the k-th tape is reset). The tree s_k is not necessarily binary.

B. For each node x a pointer to the nearest ancestor y at which the k-th tape is reset.

We say that all tapes are also reset in the root of the whole tree t. All this information can be easily completed during a top-down pass, in time $O(|D||t|)$. We also perform the preprocessing step of Fact 1.4 on the trees s_k; it allows us to find in constant time a child of a node which is an ancestor of another node.

The key property of the information above is that it allows to compute run_t in constant time. Assume we have two nodes $x \leq y$ and a state $d \in D$ and we want to calculate $run_t(x, d, y)$. We find which tape in node x contains state d. As in the word case (Section 3.2.1), it is enough to find the nearest descendant of x on the path to y in which the tape joins some other tape; we move in that way until we reach y. As before there are at most K changes of the current tape. Although now we do not have a direct pointer to such descendants, they still can be computed in constant time: we move to the nearest ancestor in which the current tape is reset and then to its child in an appropriate tree s_k. We have to choose the child, which is an ancestor of y, this can be done in constant time using Fact 1.4 (as y may be not a node of s_k, we first need to move to its nearest ancestor which is in s_k, using the pointer from point B). This proves the following lemma.

Lemma 3.2
For any two nodes $x \leq y$ and a state $d \in D$ the state $run_t(x, d, y)$ can be found in time $O(|D|)$.

Now see how the second variant of Theorem 3.1 follows from this lemma. Assume we have a binary tree t labelled by binary relations over a set Q. We take $D = \mathcal{P}(Q)$, and an automaton $\mathcal{D}$, which from a state $d \subseteq Q$ after reading a letter $r \in R_Q$ goes to the state $d \circ r$. Then we construct the tapes data structure for t and $\mathcal{D}$. Observe that $run_t(x, d, y) = d \circ val_t(x, y)$. In the query step to find $val_t(x, y)$ it is enough to find $run_t(x, \{q\}, y)$ for each $q \in Q$. So we simply make $|Q|$ queries to Lemma 3.2, each in time $O(|D|) = O(2^{|Q|})$.

3.3 Polynomial combined complexity for words

In this section we prove Theorem 3.1 in the case when the tree t forms a word. Recall that a tree forms a word when each node has only one child.

Fix a binary tree $t \in etrees(R_Q)$ which forms a word. A corresponding word w over alphabet R_Q can be constructed: the word can be read along the only path in the tree. Nodes of the tree correspond to places between letters of the word. For any two nodes $x < y$, we have a corresponding factor F consisting of letters between x and y. Observe that $val_t(x, y) = val_w(F)$.

Suppose that w is a word with a homogeneous factorization forest $\mathcal{F}$. Colcombet observed in [Col07] that the value of any infix $w[F]$ of w, not necessarily from $\mathcal{F}$, can be calculated in time linear in the height of $\mathcal{F}$. Our work builds on this observation. We show that the time can be even logarithmic in the height of $\mathcal{F}$.

3.3.1 Accelerating pointers

For the algorithms, we represent a factorization forest $\mathcal{F}$ as follows. Each factor is represented by a record with its first and last position, and its value. Each position x contains a pointer to the record of the factor $\{x\}$.

Each factor record stores a pointer to its parent factor record, but also to some other ancestors, as described below. Let n be a number from 0 to the logarithm of the height of the factorization

forest. Consider a factor $F \in \mathcal{F}$. We create a pointer from the record of F to the record of the 2^n-parent of F, call it G. (The 2^n-parent is the ancestor 2^n levels above.) This pointer is called the *accelerating pointer of length* 2^n. It is decorated by two elements of R_Q, which are the values of the two factors below.

- $\mathit{left}(F, G)$: positions from G that are strictly before all positions from F.
- $\mathit{right}(F, G)$: positions from G that are strictly after all positions from F.

When we say that a factor F is *decomposed* into some factors, we mean that the factors are disjoint and their union is F. By $\mathcal{P}(\mathcal{F})$ we denote the family of all factors from $\mathcal{F}$ and the factors $\mathit{left}(F, G)$ and $\mathit{right}(F, G)$ for each accelerating pointer from F to G. Note that this family satisfies the following property $\sharp$: for each factor X in $\mathcal{P}(\mathcal{F})$,

- $X = \{x\}$ for some position x, or
- X is decomposed into two factors $X_1, X_2 \in \mathcal{P}(\mathcal{F})$. Moreover, $\mathcal{P}(\mathcal{F})$ is organized so that X_1 and X_2 can be found in constant time.

Indeed, a non-singleton factor $F \in \mathcal{F}$ can be decomposed into its first child F_1 and the factor $\mathit{right}(F_1, F)$ (we have an accelerating pointer of length 1 from F_1 to F). A factor $\mathit{left}(F, G)$ for an accelerating pointer of length $2^k > 1$ from F to G can be decomposed into $\mathit{left}(F, F')$ and $\mathit{left}(F', G)$, where from F to F' and from F' to G we have pointers of length 2^{k-1}. When the length is 1, we decompose $\mathit{left}(F, G)$ into the previous sibling F' of F and $\mathit{left}(F', G)$. Similarly for $\mathit{right}(F, G)$.

The number of accelerating pointers, and the time required to compute them, is $O(|\mathcal{F}| \cdot \log h)$, where h is the height of $\mathcal{F}$. From the property $\sharp$ we immediately get that also the values remembered with each accelerating pointer can be computed in that time. From now on, we assume in our algorithms that factorization forests are equipped with accelerating pointers.

The benefit of accelerating pointers is that one can go from a factor to any of its ancestors by following a number of accelerating pointers that is logarithmic in the height of the forest. Moreover, the following lemma holds.

Lemma 3.3
Any factor X can be decomposed into several factors $X_1, \dots, X_m$ from $\mathcal{P}(\mathcal{F})$ and at most one factor X' being a collation $F_1 \cup \dots \cup F_n$, where $F_1, \dots, F_n$ are homogeneous siblings in $\mathcal{F}$. Both the number of factors and the time to compute it are logarithmic in the height of the forest.

Notice however that the number n of factors $F_1, \dots, F_n$ can be arbitrarily big; we do not find these factors, only their collation X'.

Proof
Let F be the smallest factor in $\mathcal{F}$ that contains X, and let $F_0, \dots, F_{n+1}$ $(n \geq 0)$ be the children of F that intersect X, written from left to right. The records of F, F_0 and F_{n+1} can be found by following the pointers in the forest, starting with the leftmost and rightmost positions in X. If we use the accelerating pointers, we only need time logarithmic in the height of the forest.

The factor X is decomposed as

$$X = \mathit{left}(F_1, X) \cup (F_1 \cup \dots \cup F_n) \cup \mathit{right}(F_n, X).$$

Whenever $n \geq 1$, the siblings $F_0, \dots, F_{n+1}$ are homogeneous, so $F_1 \cup \dots \cup F_n$ can be taken as X'. Moreover, $\mathit{left}(F_1, X)$ can be decomposed into several factors of the form $\mathit{right}(F, G)$ with an accelerating pointer from F to G. Namely, we take such factor for each accelerating pointer used to find F_0. Their number is logarithmic in the height of $\mathcal{F}$. Similarly for $\mathit{right}(F_n, X)$. □

3.3.2 Logarithmic querying

Lemma 3.3, together with the original idea of using homogeneous factorization forests to calculate values of infixes, gives the following result.

Lemma 3.4
Let $\mathcal{F}$ be a homogeneous factorization forest for a word $w \in R_{Q'}^*$. Using the accelerating pointers, the value of any factor can be calculated in time cubic in $|Q|$ and logarithmic in the height of $\mathcal{F}$.

Proof
Let X be a factor whose value we want to calculate. We decompose X using Lemma 3.3 into factors from $\mathcal{P}(\mathcal{F})$ and a collation. It is enough to find the value for each of them, and then compose. For the factors from $\mathcal{P}(\mathcal{F})$ the value is remembered in the data structure. It remains to find the value of the factor $X' = F_1 \cup \cdots \cup F_n$. Because $F_1, \ldots, F_n$ are homogeneous, the value of the collation $F_1 \cup \cdots \cup F_n$ is the same as the value of, say F_1, which is stored in its record. □

Combining the above lemma with a divide and conquer approach from Fact 2.1 (which gives us a factorization forest of height $O(\log |w|)$), we get an easy solution for the infix evaluation problem in the word case[1] that has preprocessing time $O(|Q|^3|w|)$ and query time $O(|Q|^3 \log(\log |w|))$. However using Theorem 2.2 to construct a factorization forest, we get preprocessing time $O(|Q|^3|w|)$ and query time $O(|Q|^5)$. This shows the third variant of Theorem 3.1. The complexity in $|Q|$ is $|Q|^5$, because $\log(R_Q) = |Q|^2$.

[1] We don't know if this method can be generalized to the tree case.

Chapter 4

A problem of simplifying snippets

In this chapter we solve another problem, which we call a problem of simplifying snippets. This problem appears naturally, when one wants to calculate an XPath node test of the form $\alpha = \beta$. As it will become clear in the next chapter, solving this problem is the main difficulty of Theorem 1.1.

Like previously, we have a binary tree t with edges labelled by binary relations over a set Q. A *snippet* is a tuple (x, y, Q_x, Q_y), where x is an ancestor of y in the tree t, and Q_x and Q_y are any subsets of Q. The nodes x and y are called the *high node* and the *low node* of a snippet. When each of the sets Q_x and Q_y contains just one element q_x and q_y, we write the snippet in the form (x, y, q_x, q_y); such snippets are called *single-state* snippets. A snippet is called *trivial* when it is a single-state snippet in which the high node is equal to the low node. The basic notion is the equivalence of two sets of snippets.

Definition 4.1 We say that a snippet (x, y, Q_x, Q_y) *selects* a pair $(p, q) \subseteq Q \times Q$ in a node z if $x \leq z \leq y$, and $p \in Q_x \circ val_t(x, z)$, and $q \in val_t(z, y) \circ Q_y$. We say that a set of snippets *selects* a pair $(p, q) \subseteq Q^2$ in a node z if at least one of snippets in the set selects it.

Definition 4.2 We say that two sets of snippets are *equivalent* if they select the same pairs in the same nodes.

The main result of this chapter is the following theorem. It is used in the next chapter regarding XPath evaluation.

Theorem 4.3
Let $t \in etrees(R_Q)$, and let S be a set of snippets in t. We can calculate an equivalent set S' of trivial snippets in time

- $O(|Q|^3(|t| + |S|) \log |t|)$*, or*
- $O(2^{O(|Q|)}(|t| + |S|))$*, or*
- *when t forms a word—*$O(|Q|^5(|t| + |S|))$.

We prove the individual variants of the theorem in the next sections. Before that, we give a few ways how the snippets can be split, following directly from the definitions.

Proposition 4.4
Let $x \leq z \leq y$. Then a snippet (x, y, Q_x, Q_y) is equivalent to the set of two snippets

$$(x, z, Q_x, val(z, y) \circ Q_y) \qquad \textit{and} \qquad (z, y, Q_x \circ val(x, z), Q_y).$$

To see that this is true, consider any node z', and two elements $p, q \in Q$. Assume first that $x \le z' \le z$. Then $p \in Q_x \circ val_t(x, z')$ and $q \in val_t(z', y) \circ Q_y$ if and only if $p \in Q_x \circ val_t(x, z')$ and $q \in val_t(z', z) \circ val_t(z, y) \circ Q_y$, because $val_t(z', z) \circ val_t(z, y) = val_t(z', y)$. It means that (p, q) at z' is selected by (x, y, Q_x, Q_y) if and only if it is selected by $(x, z, Q_x, val(z, y) \circ Q_y)$ (moreover it is never selected by $(z, y, Q_x \circ val(x, z), Q_y)$). Similarly when $z \le z' \le y$. When neither $x \le z' \le z$ nor $z \le z' \le y$, no pair can be selected by any of these snippets at z'.

We can also do the split in another, slightly stronger way.

Proposition 4.5
Let $x \le z_1 \le z_2 \le y$ be such that z_1 is the parent z_2. Then a snippet (x, y, Q_x, Q_y) is equivalent to the set of two snippets

$$(x, z_1, Q_x, val(z_1, y) \circ Q_y) \qquad \textit{and} \qquad (z_2, y, Q_x \circ val(x, z_2), Q_y).$$

This is true for the same reasons; notice that for any z' between x and y we either have $x \le z' \le z_1$ or $z_2 \le z' \le y$.

The next proposition allows us to remove redundant snippets.

Proposition 4.6
Let $x_1 \le x_2 \le y$ and let $Q_2 \subseteq Q_1 \circ val(x_1, x_2)$. Then the set of two snippets (x_1, y, Q_1, Q_y) and (x_2, y, Q_2, Q_y) is equivalent to the first of these snippets.

It follows from Proposition 4.4: The snippet (x_1, y, Q_1, Q_y) is equivalent to $(x_1, x_2, Q_1, val(x_2, y) \circ Q_y)$ and $(x_2, y, Q_1 \circ val(x_1, x_2), Q_y)$. But because $Q_2 \subseteq Q_1 \circ val(x_1, x_2)$, all pairs selected by the snippet (x_2, y, Q_2, Q_y) are also selected by the snippet $(x_2, y, Q_1 \circ val(x_1, x_2), Q_y)$.

Finally, we have yet another easy property, saying that each snippet can be replaced by single-state snippets.

Proposition 4.7
Any snippet (x, y, Q_x, Q_y) is equivalent to the set of snippets (x, y, q_x, q_y) for all $q_x \in Q_x$, $q_y \in Q_y$.

4.1 Linear-logarithmic algorithm

In this section we prove the first variant of Theorem 4.3. Fix a tree t and a set of snippets S. During the processing, every snippet is remembered in its low node. We create the structure of pointers as in Section 3.1. Then we process the snippets in two steps.

Step 1

After this step we want to have single-state snippets in which the distance between the low and high node is 2^k for some k (i.e. there is a pointer between them in our data structure).

Consider any snippet (x, y, Q_x, Q_y). Like in Section 3.1, we find the nodes $y = x_0 > x_1 > \cdots > x_n = x$ where x_{i+1} is 2^k edges above x_i for the greatest number k such that $x_{i+1} \ge x$. It holds $n = O(\log |t|)$. We consecutively calculate the sets $Q_\downarrow^i = val(x_i, y) \circ Q_y$ and $Q_i^\uparrow = Q_x \circ val(x, x_i)$ observing that

$$\begin{aligned} Q_\downarrow^i &= val(x_i, x_{i-1}) \circ Q_\downarrow^{i-1} && \text{for } 0 < i \le n, \quad Q_\downarrow^0 = Q_y, \text{ and} \\ Q_i^\uparrow &= Q_{i+1}^\uparrow \circ val(x_{i+1}, x_i) && \text{for } 0 \le i < n, \quad Q_n^\uparrow = Q_x. \end{aligned}$$

For each i it is done in time $O(|Q|^2)$. Then we replace the original snippet by snippets $(x_{i+1}, x_i, q_{i+1}^\uparrow, q_\downarrow^i)$ for all $q_{i+1}^\uparrow \in Q_{i+1}^\uparrow$, $q_\downarrow^i \in Q_\downarrow^i$, $0 \le i < n$. It easily follows from Propositions 4.4 and 4.7 that the new set of snippets is equivalent to the original snippet. This step is done in time $O(|Q|^2 |t| \log |t|)$.

Step 2

After this final step we should have only trivial snippets.

We want to consequently replace big snippets by smaller snippets. We start from the biggest. Take any snippet (x, y, q_x, q_y) where the distance between x and y is 2^k for some $k > 0$. Let z be the node exactly in the middle between them (2^{k-1} edges above y). We replace our snippet by snippets (x, z, q_x, q) for all $q \in val(z, y) \circ \{q_y\}$ and by snippets (z, y, q, q_y) for all $q \in \{q_x\} \circ val(x, z)$; we see from Propositions 4.4 and 4.7 that it gives an equivalent set. These snippets are processed again later, when all snippets of size 2^k are already removed.

Finally we get only snippets for $k = 0$, i.e. snippets (x, y, q_x, q_y) in which x is the parent of y. We replace such snippet by snippets (x, x, q_x, q) for all $q \in val(x, y) \circ \{q_y\}$ and by snippets (y, y, q, q_y) for all $q \in \{q_x\} \circ val(x, y)$; we get a set of trivial snippets; it is equivalent due to Propositions 4.5 and 4.7.

It is important that we remember each snippet only once (we remove identical snippets). Thanks to that for each 2^k we have at most $O(|Q|^2|t|)$ snippets; the procedure works in time $O(|Q|)$ for each snippet, so the whole step takes time $O(|Q|^3|t| \log |t|)$.

4.2 Tape construction

In this section we use the tape construction to convert a set of arbitrary snippets into an equivalent set of trivial snippets, in time linear in $|t|$, i.e. to prove the second part of Theorem 4.3. First, we construct the same additional data structure as in the preprocessing stage of the algorithm described in Section 3.2.2. Recall that in each node we have $K = 2^{|Q|}$ tapes, each of them contains different subset of Q. Moreover, we distinguish places in which tapes are reset and places in which tapes are not reset. If a tape containing Q_x at node x is not reset until $y \geq x$, then at y this tape contains $Q_x \circ val(x, y)$. We use the following two steps to simplify the input set of snippets.

Step 1

After this step in the set there will be only trivial snippets and snippets (x, y, Q_x, Q_y) for which the tape containing Q_x at x is not reset between x and y.

Take any snippet (x, y, Q_x, Q_y) from the input set. If $x = y$, we replace this snippet by an equivalent set of trivial snippets, like in Proposition 4.7. Otherwise x is a proper ancestor of y. We find a tape containing Q_x at x. As in Section 3.2.2, using the additional information we can find a sequence of nodes

$$x = x_1 \leq y_1 < x_2 \leq y_2 < \cdots < x_n \leq y_n = y, \qquad n \leq K,$$

such that the tape containing $Q_x \circ val(x, x_i)$ at x_i is not reset until y_i, and that x_{i+1} is a child of y_i.

We replace the snippet (x, y, Q_x, Q_y) by the set of snippets $(x_i, y_i, Q_x \circ val(x, x_i), val(y_i, y) \circ Q_y)$ for all $1 \leq i \leq n$; we get an equivalent set due to Proposition 4.5. In the snippets of the first kind, by definition the tape containing $Q_x \circ val(x, x_i)$ at x_i is not reset until y_i, so the snippets are of the proper form. We calculate the sets $val(y_i, y) \circ Q_y$ using the second variant of Theorem 3.1, it takes time $O(2^{O(|Q|)}K) = O(2^{O(|Q|)})$. The sets $Q_x \circ val(x, x_i)$ can be simply read from the current tape at x_i (but of course they also could be calculated using Theorem 3.1).

Step 2

After this final step we should have only trivial snippets. We have to deal with snippets (x, y, Q_x, Q_y) in which the tape containing Q_x at x is not reset between x and y.

The key property is that when we have two snippets (x_1, y, Q_1, Q_y) and (x_2, y, Q_2, Q_y) where Q_1 at x_1 and Q_2 at $x_2 \geq x_1$ are on the same tape, then the second snippet can be removed and we get an equivalent set (assuming that this tape is not reset between x_1 and x_2, which is true for all the snippets we have now). This follows from Proposition 4.6, because $Q_1 \circ val(x_1, x_2) = Q_2$

(as Q_1 at x_1 and Q_2 at x_2 are on the same tape). Thus for each y we always keep only at most $K2^{|Q|} = O(4^{|Q|})$ snippets, at most one for every pair of a state set and a tape number, and we immediately remove the redundant ones.

We consider every y starting from the lowest nodes and ending in the root. Let z be the parent of y. We replace any snippet (x, y, Q_x, Q_y) by equivalent set of two snippets (Proposition 4.5)

$$(x, z, Q_x, val(z, y) \circ Q_y) \qquad \text{and} \qquad (y, y, Q_x \circ val(x, y), Q_y).$$

The second one has the high node equal to the low node, but the state sets are not singletons; it can be replaced by an equivalent set of trivial snippets (Proposition 4.7). The first one is processed again, when we are in the node z. Note that it still satisfies the property that the tape containing Q_x at x is not reset until z. The value $Q_x \circ val(x, y)$ is written at y on the tape containing Q_x at x, so it can be found in time $O(|Q|)$. The other value, $val(z, y) \circ Q_y$, is computed by hand in time $O(|Q|^2)$, as z is the parent of y. This gives the total complexity $O(4^{|Q|}|Q|^2|t|) = O(2^{O(|Q|)}|t|)$.

4.3 Polynomial combined complexity for words

In this section we prove the last variant Theorem 4.3. As the input we have a tree t which forms a word, as well as a set of snippets. We want to output an equivalent set of trivial snippets.

Let w be the word written on the edges of t. Let $\mathcal{F}$ be the homogeneous factorization forest for w, of height polynomial in $|R_Q|$, created by Theorem 2.2. We also use the data structure from Section 3.3.1, in particular $\mathcal{P}(\mathcal{F})$. Recall that the number of factors in $\mathcal{F}$ is $O(|t|)$ and the number of factors in $\mathcal{P}(\mathcal{F})$ is $O(|Q|^2|t|)$.

In this section we use slightly different notation for snippets: instead of writing (x, y, Q_x, Q_y) we write (F, Q_x, Q_y), where F is the factor consisting of the letters written on the edges between nodes x and y. The snippets (F, Q_x, Q_y) in which $F \in \mathcal{P}(\mathcal{F})$ (where $\mathcal{P}(\mathcal{F})$ is the structure defined in Section 3.3.1) will be called *structural* snippets and the snippets in which $F = F_1 \cup \dots \cup F_k$ for homogeneous siblings from $\mathcal{F}$ will be called *neighbor* snippets.

Before we come to the algorithm, we make an observation; it allows us to reduce the number of neighbor snippets.

Lemma 4.8
Let $F_1, \dots, F_n$ be consecutive homogeneous siblings in $\mathcal{F}$ and S a set of neighbor snippets $\sigma = (F^\sigma, Q_x^\sigma, Q_y^\sigma)$ for which $F^\sigma = F_{i(\sigma)} \cup \dots \cup F_k$, i.e. they all end at the end of the same F_k, but may begin at the beginning of different $F_{i(\sigma)}$. Then there exists a set S' of neighbor snippets ending at the end of F_k and a set S'' of structural snippets, such that S is equivalent to $S' \cup S''$, and $|S'| \leq |Q|^2$, and $|S''| \leq |S|$. Moreover, the sets can be calculated in time $O(|Q|^2|S|)$.

Proof
First we split each snippet σ into two snippets, as described by Proposition 4.4:

$$(F_{i(\sigma)}, Q_x^\sigma, val(F_{i(\sigma)+1} \cup \dots \cup F_k) \circ Q_y^\sigma) \qquad \text{and} \qquad (F_{i(\sigma)+1} \cup \dots \cup F_k, Q_x^\sigma \circ val(F_{i(\sigma)}), Q_y^\sigma).$$

Snippets of the first kind are taken to S''; these are structural snippets. Snippets of the second kind are taken to $\widetilde{S}'$; these are neighbor snippets, which end at the end of F_k. The problem is that $\widetilde{S}'$ is too big. In a second step, for each pair of states (p, q) we take to S' the longest snippet $(F_{i(\sigma)+1} \cup \dots \cup F_k, Q_x^\sigma \circ val(F_{i(\sigma)}), Q_y^\sigma)$ from $\widetilde{S}'$ (i.e. this with $i(\sigma)$ as small as possible) among those having $p \in Q_x^\sigma \circ val(F_{i(\sigma)})$, $q \in Q_y^\sigma$. If there is no such snippet, we do not take any; if there are many, we take any of them.

Recall that the values $val(F_{i(\sigma)+1} \cup \dots \cup F_k)$ and $val(F_{i(\sigma)})$ are remembered in our data structure (in particular they are equal, because $F_1, \dots, F_n$ are consecutive homogeneous siblings). Thus the natural implementation gives the $O(|Q|^2|S|)$ complexity.

We have to prove that $S' \cup S''$ is equivalent to $\widetilde{S}' \cup S''$ (which is equivalent to S). The only nontrivial direction is to prove that any node selected by $\widetilde{S}'$ is also selected by $S' \cup S''$. So take any

pair (p',q') selected at some node z by some snippet $(F_{i(\sigma)+1}\cup\dots\cup F_k, Q_x^\sigma\circ val(F_{i(\sigma)}), Q_y^\sigma)\in\widetilde{S}'$. This pair is selected by a snippet $(F_{i(\sigma)+1}\cup\dots\cup F_k,p,q)$ for some $p\in Q_x^\sigma\circ val(F_{i(\sigma)})$ and $q\in Q_y^\sigma$ (it follows from Proposition 4.7). For the pair (p,q) some snippet $(F_{i(\tau)+1}\cup\dots\cup F_k, Q_x^\tau\circ val(F_{i(\tau)}), Q_y^\tau)\in\widetilde{S}'$ was taken to S', which was created as a part of a snippet $\tau\in S$. It holds $i(\tau)\le i(\sigma)$, as well as $p\in Q_x^\tau\circ val(F_{i(\tau)})$ and $q\in Q_y^\tau$. From homogeneouity, $val(F_{i(\tau)}) = val(F_{i(\tau)}\cup\dots\cup F_{i(\sigma)})$, so $p\in Q_x^\tau\circ val(F_{i(\tau)}\cup\dots\cup F_{i(\sigma)})$. Using Proposition 4.6 we get that the set of two snippets $(F_{i(\tau)}\cup\dots\cup F_k, Q_x^\tau, q)$ and $(F_{i(\sigma)+1}\cup\dots\cup F_k,p,q)$ is equivalent to only the first of them. It means that our pair (p',q') is selected at z by $(F_{i(\tau)}\cup\dots\cup F_k,Q_x^\tau,q)$; hence also by $(F_{i(\tau)}\cup\dots\cup F_k,Q_x^\tau,Q_y^\tau)$. But this snippet is equivalent to a set consisting of the snippet $(F_{i(\tau)},Q_x^\tau, val(F_{i(\tau)+1}\cup\dots\cup F_k)\circ Q_y^\tau)$ from S'' and the snippet $(F_{i(\tau)+1}\cup\dots\cup F_k, Q_x^\tau\circ val(F_{i(\tau)}), Q_y^\tau)$, which is in S'; thus our pair is selected also by $S'\cup S''$. □

Like in the previous sections, we simplify the snippets in several steps: this time we have three steps.

Step 1

After this step there will be only structural and neighbor snippets.

Consider a snippet (F,Q_x,Q_y) from the input set. We decompose F according to Lemma 3.3 into $X_1,\dots,X_n,X'$. This gives an equivalent set of structural and neighbor snippets: for each i, we take the snippet $(X_i, Q_x\circ val(left(X_i,F)), val(right(X_i,F))\circ Q_y)$; similarly for X'. Hence we may replace the original snippet by the new ones; we do this for each of the snippets. We need to calculate the sets $Q_x\circ val(left(X_i,F))$ and $val(right(X_i,F))\circ Q_y$. Notice that, for each i, $val(X_i)$ is known (stored in the data structure), and $val(X')$ can be calculated using Lemma 3.4 in time $O(|Q|^5)$. The sets $Q_x\circ val(left(X_i,F))$ can be calculated from left to right (and $val(right(X_i,F))\circ Q_y$ from right to left), by decomposing $left(X_i,F)$ into the last factor Y among $X_1,\dots,X_n,X'$ contained in $left(X_i,F)$, and to $left(Y,F)$, for which the value is already calculated.

The running time is $O(|Q|^5|S|)$. The number of snippets increases by logarithm of the height of $\mathcal{F}$, which is $O(|Q|^2)$. Namely, after this step we have $O(|S|)$ neighbor snippets and $O(|Q|^2|S|)$ structural snippets.

Step 2

After this step there will be only structural snippets.

We process the neighbor snippets in a right-to-left pass through each sequence of siblings in $\mathcal{F}$. When we are in a factor F_k, we eliminate neighbor snippets ending at the end of F_k (and starting at the beginning of some sibling of F_k). First we reduce their number using Lemma 4.8, so that only $|Q|^2$ are left (this generates also some structural snippets). Then, using Proposition 4.4, we split each of them into F_k and the rest, which results in a structural snippet and a neighbor snippet ending at the end of F_{k-1}. The snippets of the second kind are processed again later, when we are in F_{k-1}. Lemma 4.8 ensures that the number of snippets is always small: for each factor F_k we have to process the original neighbor snippets from Step 1 which end at the end of F_k, and at most $|Q|^2$ new neighbor snippets. As the number of factors in $\mathcal{F}$ is $O(|t|)$, the running time is $O(|Q|^4(|t|+|S|))$. The number of newly created structural snippets is at most twice the number of processed neighbor snippets, hence at the end we have $O(|Q|^2(|t|+|S|))$ structural snippets.

Step 3

The only thing left is to simplify the structural snippets. We first replace each structural snippet by single-state structural snippets, using Proposition 4.7. Then we start from the longest snippets and we move towards shorter. For each X in $\mathcal{P}(\mathcal{F})$, we have at most $|Q|^2$ snippets (X,p,q) (at most one for each p and q). Thanks to the property $\sharp$, given in Section 3.3.1, if the factor X contains at least two positions, it can be divided into factors X_1 and X_2 from $\mathcal{P}(\mathcal{F})$. We replace the snippet (X,p,q) by snippets (X_1,p,q') for all $q'\in val(X_2)\circ\{q\}$ and snippets (X_2,p',q) for all

$p' \in \{p\} \circ val(X_1)$ (equivalent due to Propositions 4.4 and 4.7). For each snippet (X, p, q) it takes time $O(|Q|)$, we have $O(|Q|^2)$ snippets for each X, and we have $O(|Q|^2|t|)$ factors X in $\mathcal{P}(\mathcal{F})$, so the running time of this procedure is $O(|Q|^5|t|)$. Finally, we have only snippets (X, p, q) in which X contains one position, i.e. in which the high node is the parent of the low node. They can be replaced by trivial snippets, as described by Propositions 4.5 and 4.7.

Chapter 5

Evaluating Regular XPath node tests

The goal of this chapter is to present an algorithm evaluating Regular XPath queries (node tests) in an XML document, i.e. to prove Theorem 1.1. Recall that the theorem says about four algorithms, having four different complexities. In this chapter we concentrate on the first three of them. In fact, almost the only difference between the algorithms is which subroutine from the previous two chapters is used; this chapter describes the common part of these algorithms. The fourth variant of the theorem, concerning FOXPath queries, is addressed in Chapter 6.

5.1 Proof strategy

In this section we describe the high-level structure of our linear time algorithms.

To allow storage of intermediate results, we slightly extend the definition of node labels. Now a data tree t comes with some constant k and in every node of t there is an array of k labels from A. A node test that checks for a label is now of the form $\texttt{label}[i] = a$ where $1 \leq i \leq k$ is an integer constant and $a \in A$; it holds in nodes whose i-th label is a. We do not change the definition of the data tree size—the size of t is the number of nodes plus the sum of lengths of string values of its attribute and text nodes. In particular the size does not depend on k (and also the complexity of all the algorithms does not depend on k).

Consider a node test φ defined in Regular XPath. We will present an algorithm that selects the nodes of a data tree t satisfying φ. The algorithm is defined by induction on the structure of the query (which means that it is recursive and takes a subquery as a parameter).

There are a few easy cases: when φ just tests a label or when it is a negation, conjunction or disjunction of smaller node tests. For example to evaluate a node test $\varphi \vee \varphi'$, first we evaluate both φ and φ' from the induction assumption, which gives in every node of t two boolean values, and then in every node we check whether any of them is true.

Consider now the first nontrivial induction step: a node test α `RelOp` β. Let $\varphi_1, \ldots, \varphi_n$ be the node tests that appear in the path expressions α and β. Using the induction assumption, we run a linear time algorithm for each of these node tests, and label each node in the data tree with the set of node tests from $\varphi_1, \ldots, \varphi_n$ that it satisfies. Formally we enrich the alphabet A by constants `true` and `false` and we construct a new data tree t'. It is almost the data tree t, only the labels will be changed. In each node instead of one label we will have a label array consisting of $n + 1$ elements. The first element of the array contains the original label of this node from the data tree t. The $i + 1$-th element is `true` if the node satisfies φ_i and `false` otherwise. Due to our specific definition of size, the number of labels does not count to the size, so both data trees have the same size. Then we create new path expressions α' and β' by replacing every φ_i in α or β by a label test checking if the $i + 1$-th element of the label array is equal to `true` and we run the modified node test α' `RelOp` β' on the data tree t'—it will be true in exactly the same nodes as the original node test. Path expressions like α' and β' will be called unnested.

Definition 5.1 A path expression γ is *unnested*, when the only node tests appearing in atomic path expressions in γ are label tests.

The above discussion shows that, when the subqueries $\varphi_1, \ldots, \varphi_n$ are already evaluated, it is enough to give an algorithm for a node test where α' and β are unnested. Moreover note that $|\alpha' \texttt{ RelOp } \beta'| = O(|\alpha \texttt{ RelOp } \beta| - |\varphi_1| - \ldots - |\varphi_n|)$. The remaining sections of the article are devoted to evaluating node tests of the form $\alpha' \texttt{ RelOp } \beta'$ where the path expressions α' and β' are unnested.

The same approach succeeds with node tests $\alpha \texttt{ RelOp } c$: it is enough to evaluate all node tests which appear in α and then $\alpha' \texttt{ RelOp } c$ for some unnested α' on an appropriate data tree t'. We can even go further: the node test $\alpha' \texttt{ RelOp } c$ can be easily simulated by one of the other kind $\alpha'' \texttt{ RelOp } \beta$, where α'' and β are also unnested. We construct a data tree t'', which is a modified version of t': we add a new root above the current root of t'; it contains the constant c in a string value. The label array would be extended with an additional field, which is `true` in the new root and `false` in the nodes from t'. The node test $\alpha'' \texttt{ RelOp } \beta$ in t'' should return the same as $\alpha' \texttt{ RelOp } c$ in t: β just goes to the root, while α'' does the same as α' omitting the new root. To get such α'' after every axis in α' we add a label test checking that we are not in the new root. Note that under the natural assumption[1] $|t| \geq |c|$, we have $|t''| \leq |t| \cdot 2 = O(|t|)$. We also have $|\alpha'' \texttt{ RelOp } \beta| = O(|\alpha \texttt{ RelOp } c| - |\varphi_1| - \ldots - |\varphi_n|)$.

Concluding, only the construction $\alpha \texttt{ RelOp } \beta$, for various values of `RelOp`, is left for the next sections, and only in the case when α and β are unnested. Moreover the complexity of the whole algorithm is the same as a complexity of an algorithm for this case.

Corollary 5.2
Assume we have an algorithm which, for unnested α and β, evaluates the node test $\alpha \texttt{ RelOp } \beta$ in a data tree t in time $T(|\alpha \texttt{ RelOp } \beta|, |t|)$. Then there is an algorithm evaluating any Regular XPath node test φ in a data tree t in time $O(T(O(|\varphi|), O(|t|)))$.

5.2 Preparing the tree

Before we come to solving node tests $\alpha \texttt{ RelOp } \beta$ for unnested α and β, we describe data structures used to represent a data tree. The operations described in this section can be done without knowing the query; they prepare a tree to answer to any query. In particular we show in this section how one can quickly compare data in the nodes of a tree. We also define skeletons and we show how to construct them.

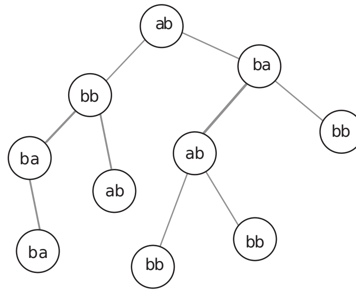
First, we say how a data tree is stored in memory by the algorithm. An initial situation is that we have a record for each node, called the *node record*. This record contains the array of node labels, the string value (in text and attribute nodes), as well as pointers to the node records of the left child, the right child, and the parent. Some of these may be empty, if the appropriate nodes do not exist. Moreover we remember the level of each node (i.e. the distance from the root).

Let the *class of* d be the set of all closest common ancestors of any two nodes x and y having string value d. In particular every node with a string value d is in the class of d (since a node x is the closest common ancestor of x, x). In the evaluation algorithm, it will be convenient to reason about classes. Therefore, for each string value, we keep a copy of the tree where only nodes from the class are kept, as described below.

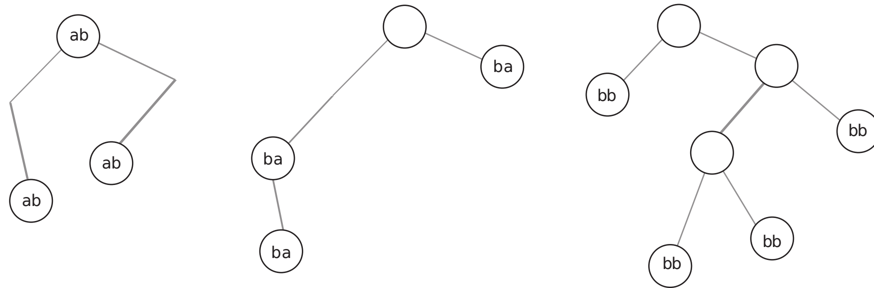
Let t be a data tree and let d be a string value. The d*-skeleton* of t, is a binary tree obtained by only keeping the nodes of t from the class of d. The tree structure in the d-skeleton is inherited from t. In particular, x is a child of y in the d-skeleton if in the tree t, x is a descendant of y, and no node between x and y belongs to the class of d.

For instance, consider the following document, where the picture shows the nodes and their string values.

[1]A more careful analysis shows that Theorem 1.1 stays true even without this assumption.



There are three string values ab, ba and bb. Below we show the d-skeleton for each of these classes. Note how these skeletons share nodes, e.g. all of them contain the root of the document.



The *skeleton representation* of a data tree t consists of the record representation of t and all of its d-skeletons. Furthermore, for each d-skeleton, each node record contains a pointer to the corresponding node in t and each node record in t contains a list of corresponding nodes in all d-skeletons to which it belongs.

Note that the sum of sizes of all skeletons in t is linear in $|t|$, since each node may be a leaf only in one skeleton. Moreover, the skeleton representation can also be calculated in linear time. The crucial operations are comparing the string values and finding the CCA of any two given nodes.

First, we discuss how string values of nodes can be quickly compared. If the sum of their lengths is bounded by the size of the document, we could simply sort them lexicographically. However the situation is complicated by the fact that the string values overlap: a string value in an element node is a concatenation of all text node descendants of its left child (see Section 1.2.1). Operations on string values needed by the algorithm are described in the following two propositions. The first one is used for calculating d-skeletons. The second one is useful during evaluation of node tests α `RelOp` β, where `RelOp` is one of the inequalities.

Proposition 5.3
For a data tree t we can group all its nodes into sets of nodes with the same string value, in time $O(|t|)$.

Proposition 5.4
For a data tree t, after preprocessing in time $O(|t|)$, we can answer, in time $O(1)$, queries of the form: for given two nodes x, y, is the string value in x lexicographically smaller than the string value in y?

Proof (of Propositions 5.3 and 5.4)
A *suffix array* is the lexicographically sorted array of the suffixes of a word (of course in this array we do not remember the whole suffixes, only their numbers). Kärkkäinen and Sanders [KS03] show how to construct the suffix array in linear time. Moreover they show that some additional

data can be calculated such that in constant time we can find a longest common prefix of any two suffixes.[2]

We use the algorithm in the following way: We concatenate the string values of all text nodes in the document order and after them the string values of all attribute nodes; we get some word w. Note that w contains the string values of all element nodes as infixes, however they overlap. For every node we calculate which infix it is (the start position and the length). This can be done during one traversal through the tree. Now we run the suffix array algorithm on the word w. We also calculate the so-called reversed suffix array: for each suffix we remember its position in the suffix array.

To get Proposition 5.3 we sort all nodes by the length of their string values—we can do this in linear time using counting sort (or bucket sort), because these lengths are bounded by the document size. Now we process every length of string values separately (only string values having the same length may be equal). For every string value we consider a suffix of w starting at the position where this string value starts. We process string values of a given length in the (already calculated) lexicographical order of these suffixes. We know (in constant time, from the Kärkkäinen and Sanders algorithm) what is the length of the common fragment of a suffix and the next suffix corresponding to a string value of the same length. If it is equal or longer than the length of the string values, then these string values are equal. If not, they are not equal and moreover the first one can not be equal to any further string value, due to the lexicographic ordering in the suffix array.

Now see that Proposition 5.4 is also true. Assume one comes with two nodes x and y. Their string values are prefixes of some suffixes of w. From the second part of the Kärkkäinen and Sanders algorithm we know the first position on which the two suffixes differ. When they differ further than the length of the shorter of our string values, then the shorter string value is a prefix of the longer one, so it is also lexicographically smaller. Otherwise the order of the string values is the same as the order of the suffixes, which we know from the reversed suffix array. □

Next, we show how to calculate d-skeletons.

Proposition 5.5
The skeleton representation of a data tree t can be calculated in time $O(|t|)$.

Proof
From Proposition 5.3 we already know leaves of all d-skeletons. We need to find other nodes in the skeletons and connect them appropriately. An almost naive use of Fact 1.3 allows us to calculate skeletons in linear time. We consider each skeleton separately, all leaves in the skeleton from left to right. At every moment we already have a skeleton for some subset of leaves and all other leaves are to the right of it. We want to add the next leaf to the skeleton. We find the closest common ancestor z of this new leaf y and the rightmost already processed leaf x. We need to add z in the appropriate place in the skeleton. We compare z with the nodes on the rightmost path of the skeleton, starting from x and going up. When z is between some node and its parent in the skeleton, we add it there, together with attached y. It is also possible that z is over the root of the current skeleton.

Why does it work in linear time? Potentially there are many nodes on the rightmost path of the current version of a skeleton. However always at most one of the visited nodes is an ancestor of z. Other visited nodes, which are not ancestors of z, no longer will be on the rightmost path after adding z, so every node may be visited only once in that role. □

5.3 From path expressions to automata

In this section we show how automata can be used to calculate path expressions. These will be word automata and they will be reading string descriptions of paths.

[2] If we want to get $O(|t| \log |t|)$ complexity, instead of $O(|t|)$, we can calculate the suffix array using a standard and a little bit simpler $O(|t| \log |t|)$ algorithm by Karp et al. [RKR72], instead of the linear time algorithm.

A *path* in a binary tree is a sequence of nodes $x_1, \ldots, x_n$ where each two consecutive nodes are connected (x_i is a child or parent of x_{i+1}). A path may loop. A *string description* of a path $x_1, \ldots, x_n$ is a word $A_1 m_1 A_2 m_2 \cdots A_{n-1} m_{n-1} A_n$ over the alphabet $(\{1, \ldots, k\} \times A) \cup \{\texttt{to-left}, \texttt{to-right}, \texttt{from-left}, \texttt{from-right}\}$, where k is the number of elements in the label array of every node of t. The m_i is a letter, which is the name of one of the four one-step axes depending on the relationship between x_i and x_{i+1} in t. So it is `to-left`, `to-right`, `from-left`, or `from-right` when the node x_{i+1} is the left child of x_i, the right child of x_i, x_i is the left child of x_{i+1} or the right child of x_{i+1}, respectively. The A_i is a word, which consists of some pairs (j, a) such that the j-th label of x_i is a. So a path has a lot of (infinitely many) different string descriptions, depending on which pairs (j, a) are included in it, allowing for reorderings and repetitions. In particular some words A_i may be empty.

A *simple path* between two nodes is the (unique) path on which no node appears more than once. A *simple string description* is a (not unique) string description in which every word A_i contains at most one letter.

We will use nondeterministic automata to read string descriptions. Let $\mathcal{A}$ be such an automaton, with states Q. Let x, y be any two nodes in a tree t. We write $trans^{all}_{\mathcal{A},t}(x, y)$ for the set of state pairs (p, q) such that some string description of some path from x to y can take the automaton $\mathcal{A}$ from a state p to a state q. Note that three objects are quantified existentially here: the path from x to y, the string description, and the run of the nondeterministic automaton. Similarly, we write $trans_{\mathcal{A},t}(x, y)$ for the set of state pairs (p, q) such that some simple string description of the simple path from x to y can take the automaton $\mathcal{A}$ from state p to state q. When both t and $\mathcal{A}$ are clear from the context, we simply write $trans(x, y)$.

An unnested path expression can be translated into an automaton reading string descriptions of paths, as described in the following lemma; this is the standard translation of regular expressions into nondeterministic automata.

Lemma 5.6
Let α be an unnested path expression. There exists an automaton $\mathcal{A}$ reading string descriptions such that a pair of nodes x, y of a data tree t is selected by α if and only if $(q_I, q_F) \in trans^{all}_{\mathcal{A},t}(x, y)$ for some initial state q_I and accepting state q_F. The automaton has $O(|\alpha|)$ states and can be constructed in time $O(|\alpha|^2)$.

Until now, our automata had to read string descriptions of all paths. We want to get rid of this and concentrate only on simple string descriptions of simple paths. This is described in the following definition.

Definition 5.7 Let t, s be two data trees with the same nodes (but with different labels) and let α be an unnested path expression. We say that an automaton $\mathcal{A}$ in the tree s *simulates* α in the tree t, when for any two nodes x, y of t (and simultaneously of s),

- $trans^{all}_{\mathcal{A},s}(x, y) = trans_{\mathcal{A},s}(x, y)$, and
- the pair x, y is selected by α in t if and only if $(q_I, q_F) \in trans_{\mathcal{A},s}(x, y)$ for some initial state q_I and accepting state q_F.

The main result of this section is the following theorem, which we are proving through the rest of the section:

Theorem 5.8
Let t be a data tree and α an unnested path expression. We can calculate, in time $O(|\alpha|^3|t|)$, a data tree s with the same nodes as t and an automaton $\mathcal{A}$ with $O(|\alpha|)$ states such that $\mathcal{A}$ in s simulates α in t.

To get the condition $trans^{all}_{\mathcal{A},s}(x, y) = trans_{\mathcal{A},s}(x, y)$, which says that we can consider only simple paths instead of all paths, we will calculate all possible loops which the automaton may do in the tree as described by the following lemma, proved below.

Lemma 5.9
For a nondeterministic automaton $\mathcal{A}$ and a tree t we can calculate, in time $O(|Q|^3|t|)$, for every node x of t the set

$$loop(x) = trans^{all}_{\mathcal{A},t}(x,x).$$

Once we have the *loop* sets, we can remember them in the label array of every node and modify the automaton, in such a way that it will be reading these values instead of making loops. The complexities in the number of states in the proofs below are $O(|Q|^3)$, which follows from Fact 1.5.
Proof (of Lemma 5.9)
This is a fairly standard construction. First, for each node x we calculate the subset $down(x)$ of state pairs in $loop(x)$ that correspond to paths that only visit descendants of x. The value of *down* for x depends only on the values of *down* in the two children of x, and the labels in x. Assume for a moment that having this information we can calculate $down(x)$ effectively. Then the values $down(x)$ can be calculated in a single bottom-up pass through the tree. Second, we calculate for each node x the subset $up(x)$ of $loop(x)$ that corresponds to paths that never visit proper descendants of x, but they may visit e.g. descendants of the sibling of x. The value of *up* in x depends only on the value of *up* in the parent of x, the value of *down* in the sibling of x, and on the labels in x. In particular, the values $up(x)$ can be calculated in a single top-down pass through the tree, after the values $down(x)$ are known for all nodes x. Once we have *down* and *up*, the function $loop(x)$ can easily be calculated, as the transitive closure of the union of $down(x)$ and $up(x)$.

The above algorithm would have the declared complexity, if we can calculate $down(x)$ basing on *down* in the two children x_1, x_2 of x in time $O(|Q|^3)$. In $down(x)$ there should be pairs (p,q) such that from p to q there is a transition reading letter (j,a) and the j-th label of x is a. There should be also pairs corresponding to runs which read a letter `to−left`, then do something from $down(x_1)$ and then read a letter `from−left`. Let R_c be the set of pairs (p,q) such that from p to q there is a transition reading `to−left`. Similarly R_p for `from−left`. Then to $down(x)$ we add the composition $R_c \circ down(x_1) \circ R_p$. Similarly for x_2 and the axes `to−right` and `from−right`. Then $down(x)$ is the transitive closure of all these pairs, since every string description of every path from x to x using only descendants of x can be divided into such fragments. The same way we can calculate the values of *up* in the two children of x basing on $up(x)$ and the values of *down* in the children of x. □

Proof (of Theorem 5.8)
First, let $\mathcal{A}'$ be the automaton constructed in Lemma 5.6 from the path expression α. Then, we use Lemma 5.9 to calculate the values of the *loop* function. We remember them in the tree t, getting a tree s: we forget about the labels from t, instead in the label array of every node x we put elements corresponding to all pairs (q_i, q_j), and we write there `true` or `false` depending on whether $(q_i, q_j) \in loop(x)$ or not. To get the automaton $\mathcal{A}$ we take the set of states, the set of initial states, and the set of accepting states from $\mathcal{A}'$. We remove all transitions reading labels, but we leave transitions reading axes. Moreover between every two states q_i, q_j we add a transition which reads `true` in the label corresponding to (q_i, q_j).

Take any two states p, q and any two nodes x, y. First see that if $(p,q) \in trans^{all}_{\mathcal{A},s}(x,y)$ then $(p,q) \in trans^{all}_{\mathcal{A}',t}(x,y)$. This is because the run reading a string description of some path in s from x to y may use a transition from q_i to q_j of the new type in a node z only when $(q_i, q_j) \in loop(z)$. So we can replace each such transition by the loop of $\mathcal{A}'$ from q_i in z to q_j in z and we get a run of $\mathcal{A}'$ in t. Conversely, observe that if $(p,q) \in trans^{all}_{\mathcal{A}',t}(x,y)$ then $(p,q) \in trans_{\mathcal{A},s}(x,y)$. The crucial observation is that any path from x to y has to use all the edges of the simple path. So we split the run of $\mathcal{A}'$ into fragments of two alternating types: loops staring/ending in a node of the simple path and edges of the simple path. Then each loop can be replaced by a singe transition of $\mathcal{A}$ in s of the new type; the transition is allowed in the node, because the corresponding loop exists. Moreover trivially $trans_{\mathcal{A},s}(x,y) \subseteq trans^{all}_{\mathcal{A},s}(x,y)$. Summing up, we have proved that $\mathcal{A}$ in s simulates α in t. □

When the tree s is created, we calculate and remember in the node records the following

additional information: $trans_{\mathcal{A},s}(x,x)$ for any node x and $trans_{\mathcal{A},s}(x,y)$ for y being the parent of x or the left or right child of x. The sets $trans_{\mathcal{A},s}(x,x) = loop(x)$ are indeed already calculated and stored, the sets $trans_{\mathcal{A},s}(x,y)$ for y being a child or a parent of x are compositions of three known sets, so they can be easily calculated.

In the next sections we will not be distinguishing between the trees t and s, because these are just two labeling of the same tree. Whenever we talk about the path expression α, it uses the labels defined by t, while the automaton $\mathcal{A}$ always uses the labels defined by s. Furthermore, we simply write $trans(x,y)$ for $trans_{\mathcal{A},s}(x,y)$.

5.4 Inequalities

In this section we deal with node tests of the form α `RelOp` β where `RelOp` is one of the inequalities: $\neq, <, >, \leq, \geq$ and α and β are unnested. These can be solved with linear time data complexity and polynomial time query complexity regardless of the XPath fragment.

The basic idea is as follows. If (x,y) is a node pair selected by the path expression α, a string value d of y is called a *representative for α in x*. Likewise for β. For each node x of a data tree t, we calculate the minimal and the maximal representative for α in x, or if there is no representative at all. Likewise for β. The „minimal" and „maximal" refers to the lexicographical order of string values. This information is sufficient to test if α `RelOp` β holds. For example a node x satisfies $\alpha < \beta$ if and only if there exist some representatives for α and for β and the minimal representative for α is less than the maximal representative for β. Similarly for the other inequalities. A node x satisfies $\alpha \neq \beta$ if and only if there exist some representatives for α and for β, but it is not the case that there is only one representative for α and only the same one for β.

It remains to show that the information about the representatives can be calculated efficiently. In order to do this, we slightly generalize the problem, so that a dynamic algorithm can be applied. Let $\mathcal{A}$ be an automaton with states Q. A representative for a state $q \in Q$ in a node x is a string value d of some node y with $(q, q_F) \in trans(x,y)$, where q_F is some accepting state.

Finding representatives (a minimal and a maximal representative) in this new sense is a generalization of the problem for path expressions, since any unnested path expression α or β can be simulated by an automaton reading simple string descriptions of simple paths (Theorem 5.8).[3]

In order to find the representatives, we use the standard two-step (first a bottom-up pass, then a top-down pass) approach. In the bottom-up pass we take into account only representatives which are in descendants of the current node. For example, to find the minimal such representative for a state q in a node x, we should consider: the string value of x if $(q, q_F) \in trans(x,x)$ for some accepting state q_F, and the minimal such representative in the left child y of x for any state p such that $(q,p) \in trans(x,y)$, similarly for the right child. Such a step can be done even in time $O(|Q|^2)$. It is important here that the string values can be compared in constant time due to Proposition 5.4 (we do not remember the string value itself, just a pointer to the node from which it comes). Similarly we do a top-down step, in which we look for the representatives in the rest of the tree (not being descendants of the current node), so the whole processing is done in time $O(|Q|^2|t|)$.[4] It is worth noting that we get this complexity even for Regular XPath. This contrasts with the node tests $\alpha = \beta$, which can be evaluated faster when the path expressions are from the FOXPath fragment rather that from the whole Regular XPath.

5.5 Equality tests

In this section, we show how to calculate node tests $\alpha = \beta$. We are going to use snippets and the methods to simplify them, described in Chapters 4. The strategy will be as follows: First we show how to find some set of snippets representing the solution of $\alpha = \beta$. Then, using the previous

[3]Recall that this is not only a translation of a path expression into an automaton, but we also need to relabel the tree.

[4]However the complexity of the whole algorithm is $O(|Q|^3|t|)$, because of the complexity of the preprocessing step described in the previous section.

results, we transform it into an equivalent set of trivial snippets. Finally we show that having a set of trivial snippets is enough to solve the node test $\alpha = \beta$.

From Theorem 5.8 we know that α and β can be recognized by automata. By inspecting the proof of the theorem it is easy to see that for both α and β we can use a common automaton, denoted $\mathcal{A}$, with states Q (being just the union of the automata for α and β). The set of accepting states Q_F can also be common. Only the initial states are different, say Q_I^α for α, and Q_I^β for β. Then a pair of nodes x, y is selected by α if and only if $(q_I^\alpha, q_F) \in trans(x, y)$ for some $q_I^\alpha \in Q_I^\alpha$ and $q_F \in Q_F$; similarly for β. Recall that during this translation we also need to change labels in the tree t, by adding state pairs to the labels. We use the same letter t for both the original and the relabeled tree, hoping that it will not introduce ambiguity.

A first component of the algorithm is a quick method of calculating possible automata runs between distinct nodes. This is described by the corollary, which follows from Theorem 3.1.

Corollary 5.10

For a data tree t and an automaton $\mathcal{A}$ with states Q we can, after preprocessing, answer queries of the form: given two nodes x, y such that[5] x is an ancestor or a descendant of y, and a set of states $Q_y \subseteq Q$ compute the set

$$trans(x, y) \circ Q_y.$$

This can be done in time

- *preprocessing: $O(|Q|^3|t|\log|t|)$, query: $O(|Q|^3 \log|t|)$, or*
- *preprocessing: $O(2^{O(|Q|)}|t|)$, query: $O(2^{O(|Q|)})$, or*
- *if t forms a word—preprocessing: $O(|Q|^5|t|)$, query: $O(|Q|^5)$.*

Like previously, here we mean that there are three algorithms, one for each of the listed complexities.

Proof

Fix a data tree t and an automaton $\mathcal{A}$. We create a binary tree t' which has the same nodes as t, and each its edge is labelled by a binary relation over $Q \times \{1, 2\}$. An edge from a node x to its child y is labelled by

$$\{((q, 1), (p, 1)) : (p, q) \in trans(y, x)\} \cup \{((p, 2), (q, 2)) : (p, q) \in trans(x, y)\}.$$

In other words such label is a pair $((trans(y, x))^{-1}, trans(x, y))$, appropriately encoded. Notice that then $val_{t'}(x, y)$ for any node x and its proper descendant y is also equal to

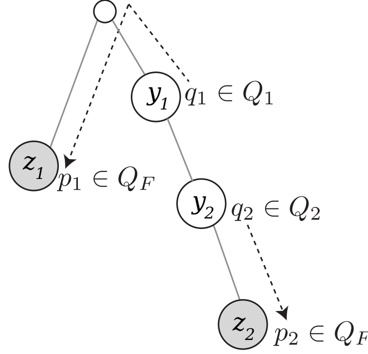
$$\{((q, 1), (p, 1)) : (p, q) \in trans(y, x)\} \cup \{((p, 2), (q, 2)) : (p, q) \in trans(x, y)\}.$$

If y is a proper descendant of x, using Theorem 3.1 we answer the query about $val_{t'}(x, y)$, then basing on it we calculate $trans(x, y) \circ Q_y$ in time $O(|Q|^2)$. Similarly if y is a proper ancestor of x, but we use the other part of $val_t(y, x)$. For $x = y$ we should simply return $trans(x, x)$, which is precalculated (notice that we need this special case: $val_{t'}(x, x)$ is always equal to identity, so we cannot extract $trans(x, x)$ from it). □

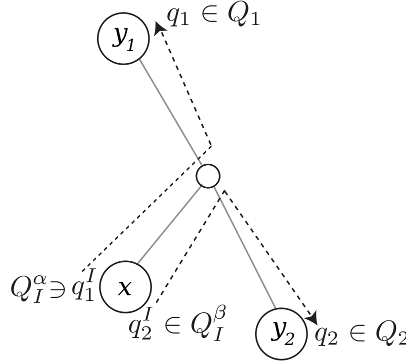
Now we come to representing the solution by snippets. Recall that a snippet is a tuple (x, y, Q'_x, Q'_y) where x is an ancestor of y in t', and Q'_x and Q'_y are subsets of some set, in our case it will be the set $Q \times \{1, 2\}$. We only use snippets of the form $(x, y, Q_x \times \{1\}, Q_y \times \{2\})$ for some $Q_x, Q_y \subseteq Q$.

A snippet $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$ represents a piece of information about the output of the query $\alpha = \beta$. The idea is that there are nodes z_1, z_2 which have the same string value, and such that for each $i = 1, 2$ and each state $q_i \in Q_i$ there is a path from y_i to z_i that takes the automaton from q_i to an accepting state $p_i \in Q_F$, as in the picture below (the dotted lines depict automaton paths, the highlighted nodes carry the same string value).

[5]The assumption that x is an ancestor or a descendant of y can be easily removed, but we do not need the stronger version of the lemma.



Namely, we say that the snippet $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$ *represents* a node x when $q_1^I \in trans(x, y_1) \circ Q_1$ and $q_2^I \in trans(x, y_2) \circ Q_2$ for some $q_1^I \in Q_I^\alpha$, $q_2^I \in Q_I^\beta$ or $q_1^I \in Q_I^\beta$, $q_2^I \in Q_I^\alpha$. In other words, from two initial states in x, one for α, one for β, $\mathcal{A}$ can reach a state from Q_1 in y_1 and a state from Q_2 in y_2, as in the picture below.



We say that a set of snippets is *sound* when all nodes selected by these snippets are also selected by $\alpha = \beta$. Conversely, a set of snippets is *complete* when all nodes selected by $\alpha = \beta$ are also selected by the set of snippets. Our algorithm will first create a sound and complete set of snippets. Then it will convert the snippets into trivial snippets, ensuring that the set of snippets is still sound and complete. Finally, after this transformation, we get only trivial snippets, from which the set of nodes selected by $\alpha = \beta$ will be calculated.

It would be very easy to construct some sound and complete set of snippets, if the assumption that x is an ancestor of y would not be present. We would simply take a snippet $(y_1, y_2, Q_F \times \{1\}, Q_F \times \{2\})$ for each pair y_1, y_2 of nodes with the same string value. It is obviously sound and complete. The assumption that x is the ancestor of y does not complicate too much: we can split every such snippet into two, in the closest common ancestor of y_1 and y_2. The more serious problem is that such set of snippets may be too big: it may have quadratic size, for example when every node has the same string value. Our first goal is to calculate a smaller set of snippets, as described by the following lemma.

Lemma 5.11
For a data tree t and a node test $\alpha = \beta$ given by an automaton $\mathcal{A}$ we can find some sound and complete set of snippets in time $O(|Q|^3|t| + T_P + |t|T_Q)$, where T_P and T_Q is the preprocessing time and the query time of an algorithm described by Corollary 5.10. Moreover, the set contains $O(|t|)$ snippets.

Proof
For any string value d and a node x in the class of d we define a set $class(x, d)$ of states p such that $(p, q_F) \in trans(x, y)$ for some $q_F \in Q_F$ and for some node y with the string value d. Note

that the requirement on x is weaker than that on y: y needs to have string value d, while x only needs to be in the class of d, so it may be a CCA of two nodes with the string value d.

We calculate all the sets $class(x, d)$. We do the calculation separately for every d-skeleton, in time proportional to its size. Once again we use here a bottom-up pass followed by a top-down pass. In the bottom-up pass for every node x of a d-skeleton we calculate the part $class^{down}(x, d)$ of $class(x, d)$ such that the node y from the definition is a descendant of x (which includes $y = x$). The crucial observation is that the set $class^{down}(x, d)$ depends only on these sets for its two d-children x_1, x_2, and on x itself: it is a union of $trans(x, x_i) \circ class^{down}(x_i, d)$ for $i = 1, 2$ and, if the string value of x is d, it is also a union with $trans(x, x) \circ Q_F$, where Q_F stands for the set of accepting states. Thus, for one node x of a d-skeleton we need to make three queries to Corollary 5.10. In total we have $O(|t|)$ nodes in all d-skeletons, hence we get the desired complexity.

In the top-down pass we calculate the part $class^{up}(x, d)$ of $class(x, d)$ such that the node y is not a descendant of x, this is very similar to the above. The desired set $class(x, d)$ is the union of $class^{down}(x, d)$ and $class^{up}(x, d)$.

We create our set of snippets as follows. For each data value d and each y_1, y_2 such that y_1 is the parent of y_2 in the d-skeleton we take to the set a snippet $(y_1, y_2, class(y_1, d), class(y_2, d))$. Additionally, for each d and each y in the d-skeleton we take a snippet $(y, y, class(y, d), class(y, d))$.[6]

Looking at the definitions it is easy to see that these snippets are sound (we also use here the fact that $trans(x, y) = trans^{all}(x, y)$ for any x, y). Now see that the set is complete. Take any node x selected by $\alpha = \beta$. Let z_α, z_β be nodes with the same string value d such that z_α (respectively, z_β) is reachable from x using α (β). Let y_α be the first node in the d-skeleton on the simple path from x to z_α; similarly for β. If $y_\alpha = y_\beta$ then x is selected by the snippet of the second kind for $y = y_\alpha = y_\beta$. Otherwise y_α is a parent or a child of y_β in the d-skeleton, because we have a path from y_α to y_β (through x) not going through any node from the d-skeleton. Then x is selected by the snippet of the first kind for $y_1 = y_\alpha$, $y_2 = y_\beta$ or $y_1 = y_\beta$, $y_2 = y_\alpha$. □

In the next stage, the algorithm simplifies the set of snippets, so that all snippets become trivial; this is described by the following corollary.

Corollary 5.12
Let t be a data tree, let $\mathcal{A}$ be an automaton with states Q, and let S be a set of snippets of the form $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$, where $y_1 \le y_2$ are nodes of t, and $Q_1, Q_2 \subseteq Q$. We can calculate a set S' of trivial snippets, which represents the same set of nodes as S, in time

- $O(|Q|^3(|t| + |S|) \log |t|)$, *or*
- $O(2^{O(|Q|)}(|t| + |S|))$, *or*
- *when t forms a word—*$O(|Q|^5(|t| + |S|))$.

Proof
We use here the same tree t' as in the proof of Corollary 5.10. It has the same set of nodes as t, hence every node of t can be simultaneously understood as a node of t'. Using Theorem 4.3 for S and t' we create an equivalent set S' of trivial snippets. It suffices to show that the set S' represents the same set of nodes as S.

Notice first that $(Q_1 \times \{1\}) \circ val_{t'}(y_1, z) \subseteq Q \times \{1\}$ and $val_{t'}(z, y_2) \circ (Q_2 \times \{2\}) \subseteq Q \times \{2\}$ for any nodes $y_1 \le z \le y_2$, so snippets of the form $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$ can select only pairs of the form $((p, 1), (q, 2))$. It means that the equivalent set of trivial snippets contains only snippets of the form $(z, z, (p, 1), (q, 2))$ (as other trivial snippets would select other pairs).

Next, we will show that any two equivalent sets S_1, S_2 of snippets of the form $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$ represent the same set of nodes. Indeed, take any node x represented by a snippet $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\}) \in S_1$. Consider the simple paths from x to y_1 and to y_2. Let z be the last common node on these paths; we have $y_1 \le z \le y_2$. Because x is represented, we have

[6] Equivalently, instead of the second kind of snippets, one could take a snippet (y, y, Q_F, Q_F) for each y.

$q_1^I \in trans(x, y_1) \circ Q_1$ and $q_2^I \in trans(x, y_2) \circ Q_2$ for some initial states q_1^I, q_2^I, one for α, one for β. We decompose

$$trans(x, y_1) = trans(x, z) \circ trans(z, y_1) \qquad \text{and} \qquad trans(x, y_2) = trans(x, z) \circ trans(z, y_2).$$

Thus there are $p, q \in Q$ such that

$$\begin{array}{llll} p \in trans(z, y_1) \circ Q_1 & \text{and} & q_1^I \in trans(x, z) \circ \{p\}, & \text{and} \\ q \in trans(z, y_2) \circ Q_2 & \text{and} & q_2^I \in trans(x, z) \circ \{q\}. & \end{array}$$

Moreover, if $z = y_1$, we can assume that $p \in Q_1$, and if $z = y_2$, we can assume that $q \in Q_2$. By definition of labels in the tree t', it means that

$$(p, 1) \in (Q_1 \times \{1\}) \circ val_{t'}(y_1, z) \qquad \text{and} \qquad (q, 2) \in val_{t'}(z, y_2) \circ (Q_2 \times \{1\}),$$

hence the pair $((p, 1), (q, 2))$ is selected at node z by the snippet $(y_1, y_2, Q_1 \times \{1\}, Q_2 \times \{2\})$. From equivalence, the same pair is selected by a snippet $(y_1', y_2', Q_1' \times \{1\}, Q_2' \times \{2\}) \in S_2$, from which we get $p \in trans(z, y_1') \circ Q_1'$ and $q \in trans(z, y_2') \circ Q_2'$. This time we do not know if z is on the simple path from x to y_1' and to y_2', but $trans(x, z) \circ trans(z, y_1') \subseteq trans^{all}(x, y_1') = trans(x, y_1')$; similarly for y_2'. Thus $q_1^I \in trans(x, y_1') \circ Q_1$ and $q_2^I \in trans(x, y_2') \circ Q_2$, which means that x is represented by a snippet from S_2. □

Finally, when we have only trivial snippets, we have to find nodes represented by them.

Lemma 5.13
For a data tree t, an automaton $\mathcal{A}$, and a set S of trivial snippets, we can calculate, in time $O(|Q|^3|t|)$, the set of nodes represented by the snippets.

Proof
Notice first that $|S| \leq |Q|^2|t|$, as for each node there are at most $|Q|^2$ snippets. For any node x we define a set $double(x)$ of state pairs (p_1, p_2) such that for some snippet $(y, y, (q_1, 1), (q_2, 2))$ from S it holds

$$(p_1, q_1) \in trans(x, y) \qquad \text{and} \qquad (p_2, q_2) \in trans(x, y).$$

Observe that a node x is represented by some of the snippets if and only if $(q_I^\alpha, q_I^\beta) \in double(x)$ or $(q_I^\beta, q_I^\alpha) \in double(x)$ for some initial states $q_I^\alpha \in Q_I^\alpha$ and $q_I^\beta \in Q_I^\beta$. Hence it is enough to calculate the sets $double$.

Here we also do a bottom-up pass followed by a top-down pass. In the bottom-up pass we calculate the part $double^{down}(x)$ of $double(x)$ such that the node y from the definition is a descendant of x. See how $double^{down}(x)$ depends on this value in its two children x_L, x_R. It should contain (for $i = L, R$) all pairs (p_1, p_2) such that for some states $(q_1, q_2) \in double^{down}(x_i)$ both pairs (p_1, q_1) and (p_2, q_2) are in $trans(x, x_i)$. We have to be a little careful to calculate them in time $O(|Q|^3)$: In a first step we calculate the set of state pairs (p_1, q_2) such that for some q_1 there is $(q_1, q_2) \in double^{down}(x_i)$ and $(p_1, q_1) \in trans(x, x_i)$. In the second step we calculate the required set. A straightforward implementation of both steps gives time $O(|Q|^3)$. To $double^{down}(x)$ we should also add all pairs (q_1, q_2) for snippets (x, x, q_1, q_2) from S. The top-down pass is similar. □

Summing up, by composing the above lemmas, we get an algorithm evaluating Regular XPath node tests. When we use first two variants of Corollary 5.10 and Theorem 4.3, we get first two variants of Theorem 1.1. If the third variant of Corollary 5.10 and Theorem 4.3 is used, we get an algorithm which works in time $O(|Q|^5|t|)$, assuming that the tree t forms a word. In the next section we deduce from this result the third variant of Theorem 1.1, which talks about trees of fixed height. The fourth variant of Theorem 1.1 is shown in the next chapter.

5.6 Small height of a document

In this section we give an algorithm evaluating Regular XPath queries in an XML document, which is linear in the document size, and polynomial in the query size and the document height, i.e. we prove the third variant of Theorem 1.1. Notice that a typical XML document (even very big) has a very small height. In the case when the tree forms a word, we already have an algorithm evaluating Regular XPath node tests in time $O(|\varphi|^5|t|)$, as shown in the previous sections. Recall that we say that a binary data tree forms a word, if each its node has only one child.

Of course in Theorem 1.1 we mean the height of the original, unranked tree. When we consider a document already converted to a binary data tree, such height can be calculated as follows. The *original level* of a node is the number of left children on each path from this node to the root, plus one. The *original height* of a tree is equal to the maximum of original levels of its nodes. Then the original height of a data tree is equal to the height of an XML document, which it represents, seen as an unranked tree.

Assume that t is a binary data tree having original height k, representing an XML document of height k. Such tree t over an alphabet A can be encoded, by writing the nodes in document order and decorating them with their original levels, as a data word (data tree which forms a word) $enc_k(t)$ over alphabet $A \times \{1, \dots, k\}$. This encoding can be decoded by Regular XPath in the following sense: for each node test φ we can compute in time polynomial in k and $|\varphi|$ a query $enc_k(\varphi)$ such that the set of nodes selected by φ in t can be recovered in linear time from the set of nodes selected by $enc_k(\varphi)$ in the data word $enc_k(t)$. The idea is to replace the axes: e.g. `to−right` (which goes to the next sibling in the original XML document) is replaced by a disjunction, over all $i \in \{1, \dots, k\}$, of the path expression which connects a position x having original level i with the first position $y > x$ such that y has original level i and all positions between x and y have original level at least $i + 1$. The Kleene star is needed to talk about the positions between x and y.

Chapter 6

FOXPath

In this chapter we present an algorithm evaluating a node test φ in a document t in time $O(|Q|^3|t|)$, assuming that φ is from FOXPath, i.e. we prove the fourth variant of Theorem 1.1. Recall that FOXPath is the fragment of Regular XPath in which Kleene star is not allowed: a path expression is a concatenation (composition) or union of simpler path expressions, but not a Kleene star of a simpler path expression (however multistep axes can still be used). In fact the XPath specification [CD99] does not allow Kleene star, so it is very natural to consider this restriction.

The general approach to the evaluation is the same as in Chapter 5. We can use all the results from there, as FOXPath is a part of Regular XPath. We only need to improve the complexity, using the fact that we have an expression from FOXPath.

6.1 Basic automata

We improve the results from Section 5.3. For path expressions from FOXPath in Section 5.3 we get automata of a special form, described by the following definition and theorem (which is an improved version of Theorem 5.8).

Definition 6.1 An automaton $\mathcal{A}$ is called *basic*, when its states can be numbered $Q = \{q_1, \dots, q_n\}$ in such way that transitions from q_i to q_j exist only for $i \leq j$.

Theorem 6.2
Let t be a data tree and α an unnested path expression of FOXPath. We can calculate, in time $O(|t||\alpha|^3)$, a data tree s with the same nodes as t and a **basic** *automaton $\mathcal{A}$ with $O(|\alpha|)$ states such that $\mathcal{A}$ in s simulates α in t.*

Proof
We inspect the proof of Theorem 5.8. Notice first that when α is an unnested path expression of FOXPath, the automaton constructed in Lemma 5.6 is basic. Indeed, when translating a regular expression into an automaton, only the Kleene star creates loops, and the Kleene star is forbidden in FOXPath. In FOXPath we have multistep axes, however the can cause only trivial loops.

We also slightly modify the construction of automaton $\mathcal{A}$ inside the proof of Theorem 5.8. Now we add to $\mathcal{A}$ only transitions between those states q_i, q_j for which $i \leq j$. Thanks to this $\mathcal{A}$ is basic. Because $\mathcal{A}'$ is basic, only such pairs can be in the set *loop*. Thus the other transitions would not be used at all; removing them does not change the behavior of the automaton. □

Notice that most of the parts of the algorithm in Chapter 5 work in time $O(|Q|^3|t|)$. Only Corollaries 5.10 and 5.12 have to be improved. Now on input to this corollaries, instead of arbitrary automaton, we get a basic automaton. Thus the binary tree created in the proofs of these corollaries is of a special form.

Definition 6.3 Let $t \in etrees(R_Q)$ for some set Q. We say that the labelling of t is *basic* if the elements of Q can be numbered $Q = \{q_1, \dots, q_n\}$ and there are two subsets Q^l, $Q^r \subseteq$ such that for each edge its label r satisfies

- $r \subseteq \{(q_i, q_j)\colon i \leq j\}$, and
- $r \cap \{(q,q)\colon q \in Q\} = \{(q,q)\colon q \in Q^l\}$ if the edge leads from a node to its left child, and
- $r \cap \{(q,q)\colon q \in Q\} = \{(q,q)\colon q \in Q^r\}$ if the edge leads from a node to its right child.

In other words only pairs (q_i, q_j) with $i \leq j$ are allowed, and each pair (q_i, q_i) appears either on every edge from a node to its left child or on none of them, and either on every edge from a node to its right child or on none of them. Recall that in the proofs of Corollaries 5.10 and 5.12 we create a binary tree t' which has the same nodes as the data tree t, and each its edge is labelled by a binary relation over $Q \times \{1, 2\}$. An edge from a node x to its child y is labelled by

$$\{((q,1),(p,1)) : (p,q) \in trans(y,x)\} \cup \{((p,2),(q,2)) : (p,q) \in trans(x,y)\}.$$

It is easy to see that if the automaton is basic, the labelling of t' is also basic. In particular, if $(q,q) \in trans(x,y)$, then there exists a transition from q to q reading an appropriate axis, so such pair appears simultaneously on every edge in given direction (for example, if there is a transition from q to q reading the `from−left` axis, the pair $((q,1),(q,1))$ appears on every edge from a node to its left child). Concluding, we have to prove the following two theorems.

Theorem 6.4
For a binary tree $t \in etrees(R_Q)$ with **basic** *labelling we can, after preprocessing in time $O(|Q|^3|t|)$, answer, in time $O(|Q|^3)$, queries of the form:*

- *for any node x, its descendant y, and a set Q_y compute $val_t(x,y) \circ Q_y$, and*
- *for any node x, its descendant y, and a set Q_x compute $Q_x \circ val_t(x,y)$.*

Theorem 6.5
Let $t \in etrees(R_Q)$ be a tree with **basic** *labelling, and let S be a set of snippets in t. We can calculate, in time $O(|Q|^3(|t| + |S|))$, an equivalent set S' of trivial snippets.*

6.2 Precomputing automaton runs

First we show a proof of Theorem 6.4, i.e. that after appropriate preprocessing we evaluate a value of a path in time not depending on its length, assuming that the tree has basic labelling. We remark that it is important that in the theorem we are calculating $val_t(x,y) \circ Q_y$ for given Q_y, and $Q_x \circ val_t(x,y)$ for given Q_x, not just $val_t(x,y)$. We suspect that our method would give $O(|Q|^4)$ query complexity if it would be used to calculate whole $val_t(x,y)$, while for $val_t(x,y) \circ Q_y$ and $Q_x \circ val_t(x,y)$ the query complexity is $O(|Q|^3)$. This is opposite to Chapter 3, in which $val_t(x,y)$ was calculated.

Fix a set Q and a binary tree $t \in etrees(R_Q)$ with basic labeling. A first component of our data structure is the following function. For every node y of t and every two elements $p, q \in Q$ we define $first^{up}(p,q,y)$ as a pointer to the nearest ancestor x of y such that $(p,q) \in val_t(x,y)$. It is possible that such an ancestor does not exist, in which case we remember an empty pointer instead. These pointers are stored in the node y. The following lemma shows that this function can be efficiently calculated.

Lemma 6.6
We can calculate the function $first^{up}$ in time $O(|Q|^3|t|)$.

Proof
Let x be the parent of y. Then $first^{up}(p,q,y)$ is equal to y, if $p = q$, otherwise it is the lowest from nodes $first^{up}(p,q',x)$ for all states q' such that $(q',q) \in val_t(x,y)$. We can calculate all the pointers in a single top-down pass, in every node we quantify over three states p, q', q, so it takes time $O(|Q|^3|t|)$. □

Before we come to the proof of Theorem 6.4, we give some intuitions behind it. Observe that every path expression selecting distant nodes has to use a multistep axis (since the Kleene star is not allowed in path expressions), which means that the basic automaton stays in some state q using a transition reading some axis, for example there is a transition from q to q reading `from−left`. Instead of considering an arbitrary run, we want to (for a run going upwards) reach the last such state q as quickly as possible (which is described by the $first^{up}$ function), then go up staying in this state and finally do only a few (at most $|Q|$) individual steps. Similarly for a run going downwards, we want to reach first such a state as quickly as possible (in at most $|Q|$ steps), then we go down staying in this state as long as possible, and finally do some transitions described by $first^{up}$. This approach succeeds completely, if from the starting node to the final node we can go using just one axis. In general, a path alternates between `from−left` and `from−right` axes. But again, if the number of such alternations is greater that $2|Q|$, the run has to stay in some state q for which there are transitions from q to q reading both `from−left` and `from−right` axes. Thus we can repeat the same argument for this state.

For any two nodes $x < y$ we say that x is a *direct ancestor* of y if x can be reached from y using only one of the `from−left`* or `from−right`* axes. We say that x is the (unique) *topmost direct ancestor* of y if additionally no node above x is a direct ancestor of y.

For every node y and its topmost direct ancestor x we remember in y the set $val_t(x,y)$. It is easy to calculate these values in a top-down pass. This is done in the preprocessing phase. This gives the following possibility in the query phase.

Proposition 6.7
When a node x is the topmost direct ancestor of a node y, we can calculate $Q_x \circ val_t(x,y)$ and $val_t(x,y) \circ Q_y$ in time $O(|Q|^2)$. When Q_x (Q_y) contains only one element, it can be done in time $O(|Q|)$.

We will now show how to calculate $Q_x \circ val_t(x,y)$ in the case when x is any direct ancestor of y. Suppose that x can be reached from y using the `from−left`* axis (the case of the `from−right`* is completely symmetric). Consider the sequence of nodes $y = x_0, x_1, \ldots, x_n = x$ in which x_{i+1} is the parent of x_i We are not allowed to find all of them and for example remember them on a list, as the complexity should be independent on n. When $n \leq |Q|$ we calculate $Q_x \circ val_t(x,y)$ step by step in, observing that $Q_x \circ val_t(x,x_i)$ is equal to $Q_x \circ val_t(x,x_{i+1}) \circ val_t(x_{i+1},x_i)$ for any $0 \leq i < n$, and that val_t between a node and its parent is stored in the tree. For each i it takes time $O(|Q|^2)$, so the whole computation takes time $O(|Q|^3)$.

Otherwise first we calculate sets $Q_i = Q_x \circ val_t(x,x_i)$ for $n - |Q| \leq i \leq n$ in time $O(|Q|^3)$. Recall the two sets Q^l, Q^r from Definition 6.3; an edge from a node to its left child has a pair (q,q) in its label if and only if $q \in Q^l$ (similarly for a right child and Q^r). We calculate a set Q_0: an element $q \in Q$ is in Q_0 if for some $n - |Q| \leq i \leq n$ and for some $p \in Q_i \cap Q^l$ it holds[1] $first^{up}(p,q,y) \geq x_i$ (which means that $(p,q) \in val_t(y',y)$ for some node y' below x_i); in particular $first^{up}(p,q,y)$ should be a nonempty pointer.

We will show that $Q_0 = Q_x \circ val_t(x,y)$.

First observe that $Q_0 \subseteq Q_x \circ val_t(x,y)$. Indeed, we always have $(p,q) \in val_t(first^{up}(p,q,y),y)$, from the definition of $first^{up}$. When $first^{up}(p,q,y) \geq x_i$ it also holds $(p,q) \in val_t(x_i,y)$, because $p \in Q^l$ and from $first^{up}(p,q,y)$ we can reach x_i using the `from−left`* axis.

To see that $Q_x \circ val_t(x,y) \subseteq Q_0$, take any $q_0 \in Q$ from $Q_x \circ val_t(x,y)$. This means that $(q_n,q_0) \in val_t(x,y)$ for some $q_n \in Q_x$. Moreover, there are elements $q_1, \ldots, q_{n-1}$ such that $(q_{i+1},q_i) \in val_t(x_{i+1},x_i)$ for each $0 \leq i < n$ (in general those elements can be chosen in multiple

[1] Here and below it is enough to compare levels of the nodes, because they are on the same path from the root.

ways, but fix some choice). Because there are only $|Q|$ elements and because the tree has a basic labelling, there has to be $q_r = q_{r+1}$ for some $n-|Q| \le r < n$. In particular $q_r \in Q^l$. This sequence of elements proves that $q_r \in Q_r$ and $first^{up}(q_r, q_0, y) \ge x_r$. This means that $q_0 \in Q_0$.

In the general case (when x is any ancestor of y) we calculate $Q_x \circ val_t(x, y)$ in a similar way. We define a *zig-zag* sequence from x to y: it is the (unique) sequence of nodes $y = x_0 > x_1 > \cdots > x_n = x$ such that x_{i+1} is a direct ancestor of x_i and that n is minimal. In other words, for any $0 \le i \le n-2$ the node x_{i+1} is the topmost direct ancestor of the node x_i; this is not the case for $i = n-1$, as there might be more direct ancestors of x_{n-1} above x. Like previously, we find only a few topmost nodes x_i, now $2|Q|+1$ of them, namely those for $n-2|Q| \le i \le n$. To allow this, during the preprocessing we should remember for every node z its bottommost descendant reachable by the `to−left`* axis and its bottommost descendant reachable by the `to−right`* axis. Then x_i is the closest common ancestor of y and this descendant of x_{i+1} (hence it can be calculated in constant time, using Fact 1.3).

For these topmost $2|Q|+1$ nodes we calculate the sets $Q_i = Q_x \circ val_t(x, x_i)$; first of them is calculated from the above special case in time $O(|Q|^3)$ (as x_n is just a direct ancestor of x_{n-1}), each next of them in time $O(|Q|^2)$ using Proposition 6.7 (as then x_{i+1} is the topmost direct ancestor of x_i). Then we calculate the set Q_0: an element $q \in Q$ is in Q_0 if for some $n-2|Q| \le i \le n$ and for some $p \in Q_i \cap Q^l \cap Q^r$ it holds $first^{up}(p, q, y) \ge x_i$. It holds $Q_0 = Q_x \circ val_t(x, y)$ for the same reasons as previously; the difference is that now we may go from both a left and a right child, but we consider elements from $Q^l \cap Q^r$. Since now we take $2|Q|+1$ nodes, for every sequence of states there have to be three consecutive nodes x_i, x_{i+1}, x_{i+2} with the same state and hence this state is in both Q^l and Q^r.

Although the algorithm calculating $val_t(x, y) \circ Q_y$ is not completely symmetric, it is similar. Once again we first solve the case of direct ancestor, and then the general case. Consider the case, when x is direct ancestor of y, say reachable by the `from−left`* axis. Take the sequence $y = x_0, x_1, \dots, x_n = x$ in which x_{i+1} is the parent of x_i. First for $n-|Q| \le i \le n$ we calculate sets $\widetilde{Q}_i$: element p is taken to $\widetilde{Q}_i$ if $p \in Q^l$ and for some $q \in Q_y$ there is $first^{up}(p, q, y) \ge x_i$. Then we do $Q_i = \widetilde{Q}_i \cup (val_t(x_i, x_{i-1}) \circ Q_{i-1})$ for $n-|Q| < i \le n$, starting from $Q_{n-|Q|} = \widetilde{Q}_{n-|Q|}$. The argument that $Q_n = val_t(x, y) \circ Q_y$ is very similar to the previous one. The general case is solved analogously.

6.3 Simplifying the snippets

We now come to the proof of Theorem 6.5. Recall that we have to transform a set of arbitrary snippets into an equivalent set of trivial snippets. This should be done in time $O(|Q|^3(|t|+|S|))$, assuming that the tree has a basic labeling. First we give two lemmas, which are used to simplify the snippets. The sets Q^l, Q^r below are the sets from Definition 6.3.

Lemma 6.8

For any snippet in which the high node is a direct ancestor of the low node we can find, in time $O(|Q|^3)$, an equivalent set of $O(|Q|^3)$ snippets (x, y, q_x, q_y) in which

(a) $x = y$ (trivial snippets), or

(b) $q_x \in Q^l$ and x is reachable from y by the `from−left`* *axis, or*

(c) $q_x \in Q^r$ and x is reachable from y by the `from−right`* *axis.*

Proof

Let (x, y, Q_x, Q_y) be the input snippet. Assume that x is reachable from y using the `from−left`* axis (the other case is symmetric). Consider the sequence $y = x_0, x_1, \dots, x_n = x$ where x_{i+1} is the parent of x_i. Let $k = \max(0, n-|Q|)$. For $k \le i \le n$ we calculate the nodes x_i and the sets

$Q_i^\uparrow = Q_x \circ val_t(x, x_i)$ and $Q_\downarrow^i = val(x_i, y) \circ Q_y$, observing that

$$Q_i^\uparrow = Q_{i+1}^\uparrow \circ val_t(x_{i+1}, x_i) \qquad \text{for } k \le i < n, \quad Q_n^\uparrow = Q_x, \text{ and}$$
$$Q_\downarrow^i = val_t(x_i, x_{i-1}) \circ Q_\downarrow^{i-1} \qquad \text{for } k < i \le n, \quad Q_\downarrow^k = val_t(x_k, y) \circ Q_y.$$

The set $Q_\downarrow^k$ is calculated using Theorem 6.4 in time $O(|Q|^3)$, each other of $O(|Q|)$ sets basing on the previous one in time $O(|Q|^2)$. Then we add trivial snippets $(x_i, x_i, q_i^\uparrow, q_\downarrow^i)$ for all $q_i^\uparrow \in Q_i^\uparrow$, $q_\downarrow^i \in Q_\downarrow^i$, $k \le i \le n$. We also add snippets $(x_i, y, q_i^\uparrow, q_y)$ for all elements $q_i^\uparrow \in Q_i^\uparrow \cap Q^l$ and $q_y \in Q_y$, where $k \le i \le n$. We get $O(|Q|^3)$ snippets of the allowed form.

We have to prove that the set of those snippets is equivalent to the original snippet. When $k = 0$ it is clear. Note that for $k > 0$ the set would be equivalent (by Propositions 4.5 and 4.7), if it would also contain snippets $(x_k, y, q_k^\uparrow, q_y)$ for all elements $q_k^\uparrow \in Q_k^\uparrow$, $q_y \in Q_y$ (not only these where $q_k^\uparrow \in Q^l$). Consider one such snippet. As $q_k^\uparrow \in Q_k^\uparrow = Q_x \circ val_t(x, x_k)$, we have $(q_x, q_k^\uparrow) \in val_t(x, x_k)$. Moreover, we have elements $q_{k+1}^\uparrow, \dots, q_{n-1}^\uparrow, q_n^\uparrow = q_x$ such that $(q_{i+1}^\uparrow, q_i^\uparrow) \in val_t(x_{i+1}, x_i)$ for $k \le i < n$. Because there are only $|Q|$ elements of Q, and the labelling is basic, there has to be $q_r^\uparrow = q_{r+1}^\uparrow$ for some $k \le r < n$. See that $q_r^\uparrow \in Q_r^\uparrow \cap Q^l$, so there is a snippet $(x_r, y, q_r^\uparrow, q_y)$ in our set, for which $q_k^\uparrow \in \{q_r^\uparrow\} \circ val_t(x_r, x_k)$. Thus, by Proposition 4.6 the snippet $(x_k, y, q_k^\uparrow, q_y)$ is not necessary and could be removed. □

Lemma 6.9
For any snippet we can find, in time $O(|Q|^3)$, an equivalent set of $O(|Q|^3)$ snippets (x, y, q_x, q_y) in which

(a), (b), (c) like above in Lemma 6.8, or

(d) x is the topmost direct ancestor of y, or

(e) $q_x \in Q^l \cap Q^r$.

Proof
The proof is very similar to the previous one. Now we take the zig-zag sequence between x and y and $k = \max(0, n - 2|Q|)$. The sets $Q_i^\uparrow$ and $Q_\downarrow^i$ are defined as previously; we calculate $Q_\downarrow^k$, $Q_\downarrow^n$, $Q_{n-1}^\uparrow$ using Theorem 6.4 in time $O(|Q|^3)$, each other using Proposition 6.7 in time $O(|Q|^2)$. We add snippets $(x_{i+1}, x_i, q_{i+1}^\uparrow, q_\downarrow^i)$ for all $q_{i+1}^\uparrow \in Q_{i+1}^\uparrow$, $q_\downarrow^i \in Q_\downarrow^i$, $k \le i \le n-2$ (they are of type (d)). For $i = n-1$ we cannot do the same, as x_n is not the topmost direct ancestor of x_{n-1}; instead we replace the snippet $(x_n, x_{n-1}, Q_n^\uparrow, Q_\downarrow^{n-1})$ by the set from the previous lemma. We also add snippets $(x_i, y, q_i^\uparrow, q_y)$ for all states $q_i^\uparrow \in Q_i^\uparrow \cap Q^l \cap Q^r$ and $q_y \in Q_y$, where $k \le i \le n$. The created set is equivalent to the original snippet for the same reasons as in the previous lemma. □

Now come to the proof of Theorem 6.5. First we apply Lemma 6.9 to each snippet from S, getting an equivalent set of $O(|Q|^3|S|)$ snippets of types (a)-(e). We want to eliminate snippets of types (b)-(e), leaving only trivial snippets. The key observation is that for each low node we have to remember only $3|Q|^2$ snippets; the other are redundant and can be removed. Indeed, in each node there are only $|Q|^2$ different trivial snippets; it is enough to remember each of them once. The same is true for (d) snippets, as the topmost direct descendant for a low node is unique. When we have two snippets (x_1, y, q_x, q_y) and (x_2, y, q_x, q_y) of type (b), (c), or (e), and x_1 is an ancestor of x_2, then the second snippet can be removed (Proposition 4.6), because $q_x \in q_x \circ val_t(x_1, x_2)$. Hence here also for each pair of states and each low node y we need at most one snippet.

We consider every node y starting from the lowest nodes and ending in the root. Let z be the parent of y. We replace a snippet (x, y, q_x, q_y) by snippets (x, z, q_x, q_z) for every $q_z \in val_t(z, y) \circ \{q_y\}$ (these snippets are processed again, when we are in the node z) and by trivial snippets (y, y, q, q_y) for every $q \in \{q_x\} \circ val_t(x, y)$; they are equivalent due to Propositions 4.5 and 4.7. Note that $\{q_x\} \circ val_t(x, y)$ can be computed in time $O(|Q|)$: for snippets of type (d) from Proposition 6.7; for

snippets of types (b), (c), or (e) because q is in $\{q_x\} \circ val_t(x, y)$ if and only if $first^{up}(q_x, q, y) \geq x$. The other set $val_t(z, y) \circ \{q_y\}$ is easy as well, as z is the parent of y. Since for each y we have $O(|Q|^2)$ snippets with low node at y, the whole processing takes time $O(|Q|^3|t|)$. The key point is that we remove redundant snippets whenever a new snippet is created.

Chapter 7

Regular XPath with aggregation

In this section we consider additional constructions from the XPath standard, namely aggregates. To define them, we need a third type of expressions, beside path expressions and node tests. These are numerical expressions. A numerical expression produces a real number for every node v. A typical numerical expression is `count(child)`, which for every node v calculates the number of children of v.

We extend the definition of Regular XPath from Section 1.2.2 by the following constructions:

- If α is a path expression, ϑ, ϑ' are numerical expressions, c is a real number, and $\texttt{RelOp} \in \{=, \leq, <, >, \geq, \neq\}$,

  $$\alpha\ \texttt{RelOp}\ \vartheta \qquad \text{and} \qquad \vartheta\ \texttt{RelOp}\ \vartheta' \qquad \text{and} \qquad \vartheta\ \texttt{RelOp}\ c$$

  are node tests. The first of them selects a node u if there exists a node v such that (u, v) is selected by α, and the string value of v represents a number, and this number and the value of ϑ calculated in u satisfy the relation `RelOp`. The second of them selects a node u if the values of ϑ and ϑ' calculated in u satisfy the relation `RelOp`. The third of them selects a node u if the value of ϑ calculated in u and the constant c satisfy the relation `RelOp`. The inequalities $\leq, <, >, \geq$ correspond to the linear order of real numbers.

- If α is a path expression,

  $$\texttt{count}(\alpha) \qquad \text{and} \qquad \texttt{sum}(\alpha)$$

  are numerical expressions. The first of them for a node u calculates the number of nodes v such that (u, v) is selected by α. The second of them for a node u calculates the sum of all string values (converted to numbers) of nodes v such that (u, v) is selected by α; if some of these string values does not represent a number, a special value NaN (Not-a-Number) is returned.

- Numerical expressions (representing numbers) may be added, multiplied, etc. (i.e. if ϑ, ϑ' are numerical expressions, $\vartheta + \vartheta'$, $\vartheta \cdot \vartheta'$, etc. also are).

In our analysis we assume that all operations on numbers are performed in constant time. This is somehow convergent with XPath specification [CD99], which says that all numbers should be represented as a floating point real numbers of fixed precision; hence we need not to represent very long numbers.

The main theorem of this chapter is that node tests of Regular XPath with aggregates, as well as numerical expressions, can be evaluated in time linear-logarithmic in the document size and exponential in the query size.

Theorem 7.1
Let t be an XML document, φ a node test of Regular XPath with aggregates, and ϑ a numerical expression of Regular XPath with aggregates. The set of nodes of t that satisfy φ can be computed

in time $O(2^{O(|\varphi|)}|t| + |\varphi||t|\log|t|)$. *The value of* ϑ *in each node of* t *can be computed in time* $O(2^{O(|\vartheta|)}|t| + |\vartheta||t|\log|t|)$.

7.1 Unnested aggregates

In this section we deal with numerical expressions of the form $\mathtt{sum}(\alpha)$ and $\mathtt{count}(\alpha)$ in which α is unnested.

First, for each node of the input data tree t, we remember the real number represented by the string value of that node, or that the string value does not represent any number. This can be calculated in linear time. Indeed, for text nodes and attribute nodes this is straightforward. However recall that the string value of an element node is a concatenation of the string values of all text node descendants of the left child of the element node, in document order (see Section 1.2.1). The total length of all string values may be quadratic in the document size. To deal with that we use the following observation: if for two strings u, v we know the natural numbers i, j represented by them, and we know $10^{|u|}$ and $10^{|v|}$, then we can compute in constant time the number $10^{|v|} \cdot i + j$ represented by the concatenation uv, as well as $10^{|uv|} = 10^{|u|} \cdot 10^{|v|}$. The same can be done for real numbers, but additionally we have to know if the string contains a minus or a period, and on which position (precisely, we need to remember 10^p if the period is on position p in the string).

Now we will show how to evaluate the numerical expression $\mathtt{sum}(\alpha)$ for unnested α. Recall that in each node u we have to calculate the sum of numbers in every node v such that (u, v) is selected by α. In particular these sums are commutative. As in the previous chapters we generalize the problem to automata and we use the automaton $\mathcal{A}$ simulating α, which reads string descriptions of simple paths (from Theorem 5.8). Let Q be the set of states of $\mathcal{A}$.

For each node u of a binary tree t and for each set of states $P \subseteq Q$ we define $\mathtt{sum}(u, P)$ as the sum of string values in every node v such that $(q, q_F) \in \mathit{trans}(u, v)$ for some accepting state q_F and some $q \in P$. We want to compute $\mathtt{sum}(u, P)$ for each node u in the tree, and each $P \subseteq Q$. As we consider each set of states, the algorithm is exponential in the size of α. In order to compute the function $\mathtt{sum}$ we first do a bottom-up pass, then a top-down pass. In the bottom-up pass we calculate the part $\mathtt{sum}_{down}(u, P)$ of $\mathtt{sum}(u, P)$ corresponding only to these nodes v, which are descendants of u. We see that $\mathtt{sum}_{down}(u, P)$ depends only on $\mathtt{sum}_{down}$ in its two children u_1, u_2. First we calculate sets $P_i = P \circ \mathit{trans}(u, u_i)$. Then $\mathtt{sum}_{down}(u, P)$ is equal to the sum of $\mathtt{sum}_{down}(u_i, P_i)$ for $i = 1, 2$ plus the number in u, if some accepting state is in P. Similarly we may do a top-down pass, calculating the part of $\mathtt{sum}(u, P)$ corresponding to these nodes v, which are not descendants of u. For both directions it is possible to process a node in time $O(2^{|Q|}|Q|^3)$, so the total time is $O(2^{|Q|}|Q|^3|t|)$. Finally, the result of $\mathtt{sum}(\alpha)$ in each node u is equal to $\mathtt{sum}(u, Q_I)$, where Q_I is the set of initial states.

Exactly the same approach succeeds for $\mathtt{count}(\alpha)$. Just instead of adding the number stored in a node, we add 1.

Note that the information just for singleton sets P is highly insufficient. For example if $\mathtt{sum}_{down}(u, \{q_1\}) = \mathtt{sum}_{down}(u, \{q_2\}) = 1$ we don't know whether these sums come from the same or different node, but it is important in the parent of u, for example if from some state q in the parent we may reach both q_1 and q_2 in u.

7.2 Arbitrary node tests and numeric expressions

We now show how to evaluate arbitrary node test and numeric expression of Regular XPath with aggregates, i.e. we prove Theorem 7.1. The general proof strategy is the same as described in Section 5.1. We only have to deal with the new types of constructions. Simultaneously to the algorithm evaluating node tests, we show an algorithm which for a numerical expression ϑ calculates its value in every node of a tree t. The algorithm works by induction on the size of φ or ϑ.

As previously, there are a few easy cases. To evaluate a node test ϑ `RelOp` ϑ', first we evaluate both ϑ and ϑ' from the induction assumption, which gives in every node of t two real numbers, and then in every node we check, whether the two results are equal or not. Similarly for a node test ϑ `RelOp` c, and for a numerical expression which is an arithmetical operation of smaller numerical expressions.

Consider now the nontrivial induction step: a numerical expression $\mathtt{sum}(\alpha)$. We proceed like in Section 5.1 for node tests of the form α `RelOp` β. Let $\varphi_1, \ldots, \varphi_n$ be the node tests that appear in the path expression α. Using the induction assumption, we run our algorithm for each of these node tests, and label each node in the tree with the set of node tests from $\varphi_1, \ldots, \varphi_n$ that it satisfies. Then we create new path expression α' replacing every φ_i by a label test checking if φ_i is satisfied. The numerical expression $\mathtt{sum}(\alpha')$ in the enriched tree has in each node exactly the same value as $\mathtt{sum}(\alpha)$. The gain is that α' is unnested, so it can be solved using the algorithm from the previous section. We have a bound for size: $|\mathtt{sum}(\alpha')| = O(|\mathtt{sum}(\alpha) - |\varphi_1| - \cdots - |\varphi_n|)$, which guarantees the complexity as required. In the same way we proceed with $\mathtt{count}(\alpha)$.

The only thing left is a node test of the form α `RelOp` ϑ. By the same argument as above we can assume that α is unnested. We want to simulate this query by α' `RelOp` β for some unnested path expressions α' and β. First of all, α' should select only pairs (u, v) such that the string value in v represents a number (as only such pairs are taken into account by α `RelOp` ϑ). To ensure that, we put an additional label in each node which says whether its string value represents a number or not. Then α' should check that label in the final node.

Second, we calculate the value of ϑ in each node. We store the results in the tree. If the tree would be unranked, not binary, we could add under each node a new attribute child, say at the leftmost position, and store the result in its string value. We do the same in our binary tree t: under each node u we create a new attribute node v as the left child of u, and we store there the result of ϑ from u. If u already had a left child, we attach it as a right child of v. The number of nodes in the new tree t' is twice the number of nodes in t. Now in α' we have to omit the new nodes, so each axis $\mathtt{to{-}left}$ in α is replaced by $\mathtt{to{-}left} \cdot \mathtt{to{-}right}$ in α', and each axis $\mathtt{from{-}left}$ in α is replaced by $\mathtt{from{-}right} \cdot \mathtt{from{-}left}$ in α'. The path expression β should just go to the left child.

Then α' `RelOp` β selects in the new tree t' exactly the nodes coming from the original tree t which would be selected there by α `RelOp` ϑ. This is true under the assumption that `RelOp` refers now to the linear order of numbers represented by the string values, not to the lexicographic order of string values. This is not a problem: instead of using Proposition 5.4, the algorithm evaluating α' `RelOp` β should just compare the real numbers represented by the string values. The other problem is how we write the results of ϑ in the string values of the newly created attribute nodes. We do not want to convert them to string, as these strings can be quite long. We should just remember them as floating point real numbers, slightly modifying the definition of the data tree, so that string value of an attribute node can be either a string or a number. After such modification of the definition, Proposition 5.3 is no longer true. To group all nodes into sets of nodes with the same string value, we have to sort the numbers which are in the string values; this takes time $O(|t| \log |t|)$, not $O(|t|)$ (this is the only place where the complexity in the document size becomes $O(|t| \log |t|)$, everywhere else it is linear).

Chapter 8

Evaluation of path expressions

This chapter is devoted to a proof of Theorem 1.2. Given a data tree and a Regular XPath path expression α, we want to find all pairs of nodes satisfying α, one after another. Before we come to that, in Section 8.1 we solve an auxiliary problem in the style of those in Chapters 3 and 4. Its solution will be used in Section 8.2 to evaluate path expressions.

8.1 An auxiliary problem

Let t be a binary tree labelled by binary relations over some set Q. Then for any node x of t, and two subsets $P, Q_x \subseteq Q$ we define $set_P(x, Q_x)$ as the set of all descendants y of x such that $val_t(x, y) \cap (Q_x \times P) \neq \emptyset$ (the set P is going to be fixed, so it is written in the subscript of set_P). We are going to enumerate the elements of $set_P(x, Q_x)$, which is described by the following theorem.

Theorem 8.1
Let $t \in etrees(R_Q)$ for some set Q, and let $P \subseteq Q$. After an appropriate preprocessing, we can, given a node x of t and a set $Q_x \subseteq Q$, compute all nodes in $set_P(x, Q_x)$ one after another in time

- *preprocessing: $O(|Q|^3|t| \log |t|)$, each element of $set_P(x, Q_x)$: $O(|Q|^3 \log |t|)$, or*
- *preprocessing: $O(2^{O(|Q|)}|t|)$, each element of $set_P(x, Q_x)$: $O(2^{O(|Q|)})$, or*
- *when t forms a word—preprocessing: $O(|Q|^5|t|)$, each element of $set_P(x, Q_x)$: $O(|Q|^5)$, or*
- *when the labelling of t is basic—preprocessing: $O(|Q|^3|t|)$, each element of $set_P(x, Q_x)$: $O(|Q|^3)$.*

To be precise: we first get t and P, for which we can do the preprocessing, and then we get x and Q_x, for which we should be able to enumerate the elements of $set_P(x, Q_x)$. We want to have a constant delay algorithm; we not only want to quickly find the whole set, but we need to find its first element as well as each next element in the declared time. The order in which the nodes of $set_P(x, Q_x)$ are output does not matter.

The rest of the section is devoted to a proof of the above theorem. Fix a set Q, a binary tree $t \in etrees(R_Q)$, and a set $P \subseteq Q$.

Recall that we write $x \leq y$ to denote that x is an ancestor of y. All ancestors and descendants need not to be proper, unless otherwise stated. We use here also the postfix order of nodes: for each x the nodes from the left subtree of x in t are before the nodes from the right subtree of x, and the nodes in both subtrees are before x. This order is similar to the order of the closing tags in an XML document, but it is slightly different, since it refers to the binary tree t. It is an important detail that a node is ordered after its proper descendants. To simplify comparing of nodes, in each node we remember its number in the postfix order.

Our algorithms will be returning the nodes of $set_P(x, Q_x)$ in the postfix order. Hence we define $first_P(x, Q_x)$ as the first node in the postfix order which is in $set_P(x, Q_x)$. For any $y \geq x$ we also define $next_P(x, Q_x, y)$ as the next node after y in the postfix order which is in $set_P(x, Q_x)$. Such a node may not exist, in which case we say that $first_P$ or $next_P$ returns an empty pointer. We remark that although at the end $next_P$ will be used only for nodes y from $set_P(x, Q_x)$, however inside the proofs it is used also for other nodes y, so it is defined for any descendant of x.

Observe two easy properties of $first_P$ and $next_P$, which will be useful during the calculation of these values.

Proposition 8.2
Let $x \leq z \leq y$ be three nodes and $Q_x \subseteq Q$. Assume we know the values of $next_P(x, Q_x, z)$ and $next_P(z, Q_x \circ val_t(x, z), y)$. Then $next_P(x, Q_x, y)$ can be calculated in time $O(1)$.

Proposition 8.3
Let $x \leq y$ be two nodes and $Q_x \subseteq Q$. Assume we know the value of $next_P(x, \{q_x\}, y)$ for every element $q_x \in Q$. Then $next_P(x, Q_x, y)$ can be calculated in time $O(|Q|)$. Similarly, assume we know the value of $first_P(x, \{q_x\})$ for every element $q_x \in Q$. Then $first_P(x, Q_x)$ can be calculated in time $O(|Q|)$.

Indeed, in the first proposition, as all descendants of z are before z in the postfix order, if $next_P(z, Q_x \circ val_t(x, z), y)$ is nonempty, it is the value of $next_P(x, Q_x, y)$; otherwise we should take $next_P(x, Q_x, z)$. In the second proposition $next_P(x, Q_x, y)$ is the first among the nodes $next_P(x, \{q_x\}, y)$ for all $q_x \in Q_x$; similarly $first_P(x, Q_x)$.

Now observe that the $first_P$ pointers can be easily calculated.

Lemma 8.4
The pointers $first_P(x, \{q_x\})$ can be calculated for each node x and each element $q_x \in Q$ in total time $O(|Q|^2|t|)$.

Proof
The calculation of $first_P(x, \{q_x\})$ can be easily done in a bottom-up pass, since it is $first_P(x_1, \{q_x\} \circ val_t(x, x_1))$, where x_1 is the left child of x; if this pointer is empty we should take $first_P(x_2, \{q_x\} \circ val_t(x, x_2))$ for the right child x_2 of x; if this pointer is also empty and $q_x \in P$, we should take x; otherwise we should return the empty pointer. □

Now see that the $next_P$ pointers can be all calculated when y is a child of x.

Lemma 8.5
The pointers $next_P(x, \{q_x\}, y)$ for each pair (x, y) of a parent and its child and for each element $q_x \in Q$ can be calculated in time $O(|Q|^2|t|)$.

Proof
We have two cases depending on whether y is the left or the right child of x. If it is the right child, $next_P(x, \{q_x\}, y)$ is either empty or equal to x (if $q_x \in P$). Otherwise let z be the right child of x; we have $next_P(x, \{q_x\}, y) = first_P(z, \{q_x\} \circ val_t(x, z))$ or, if this gives the empty pointer, we should take $next_P(x, \{q_x\}, y) = x$ if $q_x \in P$, and the empty pointer otherwise. □

Below, in the four subsections, we will show the following lemma, saying that it is possible to quickly compute $next_P$ for any arguments.

Lemma 8.6
For a tree $t \in etrees(R_Q)$ and a set $P \subseteq Q$ we can, after an appropriate preprocessing, answer queries of the form: for two nodes $x \leq y$ and a set $Q_x \subseteq Q$ compute $next_P(x, Q_x, y)$. This can be done in time

- *preprocessing: $O(|Q|^3|t| \log |t|)$, query: $O(|Q|^3 \log |t|)$, or*
- *preprocessing: $O(2^{O(|Q|)}|t|)$, query: $O(2^{O(|Q|)})$, or*

- *when t forms a word—preprocessing: $O(|Q|^5|t|)$, query: $O(|Q|^5)$, or*
- *when the labelling of t is basic—preprocessing: $O(|Q|^3|t|)$, query: $O(|Q|^3)$.*

It is easy to see that Theorem 8.1 follows from this lemma. Indeed, assume we have a tree $t \in etrees(R_Q)$ and a set $P \subseteq Q$. In the preprocessing step we do the preprocessing of Lemma 8.6, and we calculate $first_P(x, \{q_x\})$ for each node x and each $q_x \in Q$ (using Lemma 8.4). In the query step we are given some node x and set Q_x. The first element of $set_P(x, Q_x)$, which is $first_P(x, Q_x)$, can be calculated from Proposition 8.3. Having an element y of $set_P(x, Q_x)$, the next element, which is $next_P(x, Q_x, y)$, can be found using Lemma 8.6.

8.1.1 Linear-logarithmic algorithm

In this subsection we prove the first version of Lemma 8.6.

We need information like in Section 3.1 but slightly enriched: For every node x of the data tree t and every $0 \leq k \leq K$ we remember a pointer to its ancestor y which is 2^k edges above x (as previously, K is the greatest number such that 2^K is not greater than the height of the data tree t). Together with it we remember $val_t(x, y)$ as previously, but also $next_P(y, \{q_y\}, x)$ for each element $q_y \in Q$.

Now see how to find the pointers $next_P(y, \{q_y\}, x)$ in the preprocessing step. For $k = 1$ they can be calculated by Lemma 8.5. Then we inductively calculate the pointers for $k > 1$. Let z be the node halfway between x and y. The pointer $next_P(y, \{q_y\}, x)$ is easily calculated basing on the $next_P$ pointers for pairs (y, z) and (z, x), as described by Propositions 8.2 and 8.3.

Now come to the query step. We are given two nodes $x \leq y$ and a set of states Q_x. As in the previous subsections, we consider the nodes $y = x_0 > x_1 > \cdots > x_n = x$ $(n \leq K + 1)$ where x_{i+1} is 2^k edges above x_i for the greatest number k such that $x_{i+1} \geq x$. First for each i we calculate the sets $Q_i = Q_x \circ val_t(x, x_i)$ observing that

$$Q_i = Q_{i+1} \circ val_t(x_{i+1}, x_i) \qquad \text{for } 0 \leq i < n, \quad Q_n = Q_x.$$

As we know $next_P(x_i, \{q\}, x_{i+1})$ for each i and q, using Proposition 8.3 we calculate $next_P(x_i, Q_i, x_{i+1})$. Then Proposition 8.2 allows us to compose them into $next_P(x, Q_x, y)$.

8.1.2 Linear algorithm for Regular XPath

In this subsection we are going to prepare the data structure from Section 3.2.2 for queries about $next_P(x, Q_x, y)$. Recall first the important properties of this data structure. In each node we have $K = 2^{|Q|}$ tapes, each of them contains different subset of Q. Moreover, we distinguish places in which tapes are reset and places in which tapes are not reset. If a tape containing Q_x at node x is not reset until $y \geq x$, then at y this tape contains $Q_x \circ val(x, y)$. To simplify the notation, let Q_x^k be the set written on the k-th tape at node x. All what we need to know about the tapes data structure is the following. Let $x \leq y$ be two nodes and Q_x a subset of Q. We can find, in time constant in $|t|$, a sequence of nodes

$$x = x_1 \leq y_1 < x_2 \leq y_2 < \cdots < x_n \leq y_n = y, \qquad n \leq K = 2^{O(|Q|)},$$

such that the tape containing $Q_x \circ val(x, x_i)$ at x_i is not reset until y_i, and that x_{i+1} is a child of y_i. The numbers of tapes used in each fragment are also known. When a tape number k is used at node z, we know that $Q_z^k = Q_x \circ val_t(x, z)$.

The information collected in Section 3.2.2 will be enriched. For any node y, denote its parent as $par(y)$. For each node y (except the root) and for each subset $Q_{par(y)} \subseteq Q$ we remember $next_P(par(y), Q_{par(y)}, y)$. This is easily calculated using Lemma 8.5 and Proposition 8.3. Moreover, for each tape k and each node z we remember a pointer to its nearest ancestor y such that $next_P(par(y), Q_{par(y)}^k, y)$ is non-empty. This information is collected in a top-down pass for each tape. The preprocessing takes time $O(2^{O(|Q|)}|t|)$.

Consider first a query of a special kind: calculate $next_P(x, Q_x, y)$, where $x \leq y$ are such that the tape containing Q_x at x is not reset between x and y. Let k be the number of that tape. Note that for any node y' such that $x \leq y' \leq y$ we have $Q^k_{y'} = Q_x \circ val_t(x, y')$. This means that $next_P(x, Q_x, y)$ is equal to one of $next_P(par(y'), Q^k_{par(y')}, y')$ for $x < y' \leq y$; namely to the first of them in the postfix order. But we know that $next_P(par(y'), Q^k_{par(y')}, y')$ is between y' and $par(y')$ in the postfix order. So we need to find the nearest ancestor y' of y for which $next_P(par(y'), Q^k_{par(y')}, y')$ is not empty, and it gives $next_P(x, Q_x, y)$; a pointer to such y' is stored at y. It is also possible that the pointer shows y' which is already an ancestor of x; in this case $next_P(x, Q_x, y)$ is empty.

Consider now a general query: calculate $next_P(x, Q_x, y)$, where $x \leq y$ are arbitrary nodes. Using the data structure, we find the sequence $x_1, y_1, \dots, x_n, y_n$ mentioned at the beginning of this subsection. Let k_i be the tape used between x_i and y_i. Then from the above special case we know each $next_P(x_i, Q^{k_i}_{x_i}, y_i)$. We also know each $next_P(y_i, Q^{k_i}_{y_i}, x_{i+1})$. Note that $Q^{k_i}_{x_i} = Q_x \circ val_t(x, x_i)$ and $Q^{k_i}_{y_i} = Q_x \circ val_t(x, y_i)$. Thus, from Proposition 8.2 we can compose all these values into $next_P(x, Q_x, y)$.

8.1.3 Polynomial combined complexity for words

In this subsection we will prove Lemma 8.6 in the case when t forms a word. Let w be the word written on the edges of t. Notice that nodes closer to the root (earlier in the word) are later in the postfix order.

We use the same data structure as in Section 3.3. Recall that we have a homogeneous factorization forest $\mathcal{F}$ containing $O(|t|)$ factors, as well as a structure $\mathcal{P}(\mathcal{F})$ containing $O(|Q|^2|t|)$ factors. Additionally, for each element $q \in Q$, and each factor $F \in \mathcal{P}(\mathcal{F})$, starting at x and ending at y, we remember the pointer $next_P(x, \{q\}, y)$.

This information is calculated starting from the shortest factors, and going towards the longest. When F contains just one letter (x is the parent of y), then either $next_P(x, \{q\}, y) = x$, when $q \in P$, or $next_P(x, \{q\}, y)$ is empty. Recall property $\sharp$ from Section 3.3.1: every factor $F \in \mathcal{P}(\mathcal{F})$ containing at least two letters can be decomposed into two shorter factors $F_1, F_2 \in \mathcal{P}(\mathcal{F})$. Let z be the position between F_1 and F_2. Then we already know $next_P(x, \{q\}, z)$ and $next_P(z, \{q\}, y)$ for every $q \in Q$. Recall that we also know $val_t(x, z)$ (which is remembered together with F_1). So we can calculate $next_P(x, \{q\}, y)$ for every $q \in Q$ (see Propositions 8.2 and 8.3). For each q it works in time $O(|Q|)$, we have $|Q|$ values of q, and $O(|Q|^2|t|)$ factors, so the whole procedure works in time $O(|Q|^4|t|)$.

Consider a maximal sequence $F_1, \dots, F_n$ of homogeneous siblings in $\mathcal{F}$. Let x_{i-1} be the node just before F_i, and x_i the node just after F_i, for $1 \leq i \leq n$. Observe that for given $q \in Q$ and two indexes $0 \leq i < j \leq n$, we can find $next_P(x_i, \{q\}, x_j)$ in time $O(1)$. Indeed, observe that we have stored the values $next_P(x_i, \{q\}, x_{i+1})$ and $next_P(x_0, \{q\}, x_j)$ in the factors F_{i+1} and $F_1 \cup \dots \cup F_j$, which are in $\mathcal{P}(\mathcal{F})$ (in particular the second of them is either equal to $F_1 \cup \dots \cup F_n$, which is in $\mathcal{F}$, if $j = n$, or it is $left(F_1 \cup \dots \cup F_n, F_{j+1})$, which is one of the accelerating pointers). From homogeneouity we know that $val_t(x_0, x_{i+1}) = val_t(x_i, x_{i+1})$. Thus any node $z \geq x_{i+1}$ is in $set_P(x_0, \{q\})$ if and only if it is in $set_P(x_i, \{q\})$. So, if $next_P(x_0, \{q\}, x_j) \geq x_{i+1}$, we should take it as $next_P(x_i, \{q\}, x_j)$; otherwise $next_P(x_i, \{q\}, x_j)$ is after x_{i+1} in the postfix order (or is empty), so we should take $next_P(x_i, \{q\}, x_{i+1})$.

Now consider the query step: we want to quickly calculate $next_P(x, Q_x, y)$ for some $x < y$ and $Q_x \subseteq Q$. The factor consisting of letters between x and y can be decomposed (Lemma 3.3) into factors $X_1, \dots, X_m, X'$ such that $X_1, \dots, X_m \in \mathcal{P}(\mathcal{F})$ and $X' = F_1 \cup \dots \cup F_k$ for some homogeneous siblings $F_1, \dots, F_k$ from $\mathcal{F}$. For each of the factors $X_1, \dots, X_m, X'$, and each $q \in Q$ we know $next_I(x', \{q\}, y')$, when the factor starts at x' and ends at y' (for X' we use the observation from the previous paragraph); moreover we can also calculate $Q_x \circ val_t(x, x')$ (by composing the values of the factors from left to right). This is sufficient to calculate $next_P(x, Q_x, y)$ (see Propositions 8.2 and 8.3).

8.1.4 Polynomial combined complexity for FOXPath

In this subsection we will prove Lemma 8.6 in the case when the labelling of t is basic. We need to show how to quickly answer queries about $next_P(x, Q_x, y)$. As in the previous subsections, we have to remember some of these values. We use here the notions of direct ancestor, topmost direct ancestor, zig-zag sequence defined in Section 6.2. We also use the sets $Q^l, Q^r \subseteq Q$ from the definition of a basic labelling (Definition 6.3). Let $par(x, k)$ be the node which is reached from x by moving k times to the parent (a node k edges above x). Similarly, let $tda(x, k)$ be the node which is reached from x by moving k times to the topmost direct ancestor. We remember the following information:

A. for each element $q \in Q$, each node x, and each $1 \leq k \leq 2|Q|$ we remember the pointers $next_P(par(x, k), \{q\}, x)$ and $next_P(tda(x, k), \{q\}, x)$, if the appropriate node $par(x, k)$ or $tda(x, k)$ exists;

B. for each element $q \in Q$ and each node y we remember the nearest ancestor x of y such that $next_P(par(x, |Q|), \{q\}, x)$ is non-empty as well as the nearest ancestor x of y such that $next_P(tda(x, 2|Q|), \{q\}, x)$ is non-empty.

How to calculate this information? We start from the information in A for par and $k = 1$; it is calculated by Lemma 8.5. Then the values for bigger k are calculated by composition of smaller values, as described by Propositions 8.2 and 8.3 (namely, to calculate $next_P(par(x, k), \{q\}, x)$ we need to know $\{q\} \circ val_t(par(x, k), par(x, 1))$, $next_P(par(x, k), \{q\}, par(x, 1))$ and $next_P(par(x, 1), \{p\}, x)$ for each state p). For one x, q, and k the calculation takes time $O(|Q|)$, the total time consumed is $O(|Q|^3|t|)$.

Now we switch to the values for tda. For $k = 1$ we calculate them moving top-down in the tree. We either have $tda(x, 1) = par(x, 1)$, or we compose already calculated values of $next_P$ between the topmost direct ancestor of x and the parent of x with $next_P$ between the parent of x and x. For $k > 1$ we compose the values in the same way as for par. So this part of the preprocessing is also done in time $O(|Q|^3|t|)$.

The information in B can be collected in a top-down pass for each state.

Now we come to the query step; someone asks for $next_P(x, Q_x, y)$ for $x \leq y$. Concentrate first on queries in which x is a direct ancestor of y; assume that x is reachable from y by the `from−left`* axis (the case of the `from−right`* axis can be done symmetrically). Consider the sequence $y = x_0, x_1, \dots, x_n = x$, where x_{i+1} is the parent of x_i (we are not allowed to calculate all these nodes, as there is too many of them). The algorithm works as follows. At the beginning, we find the nodes x_i and calculate the sets $Q_i = Q_x \circ val_t(x, x_i)$ for $n - |Q| \leq i \leq n$ in time $O(|Q|^3)$. Basing on that, we have the following property, shown already in Section 6.2: For any state q and any x_i ($0 \leq i \leq n$) we can check whether $q \in Q_x \circ val_t(x, x_i)$ in time $O(|Q|)$. We were doing that in the following way: For each $p \in Q^l$ we look at $first^{up}(p, q, x_i)$, we find the smallest $n - |Q| \leq j \leq n$ such that $first^{up}(p, q, x_i) \geq x_j$ (in time $O(1)$ basing on the level), and we check whether $p \in Q_j$ (when it is true for any p, we have $q \in Q_x \circ val_t(x, x_i)$).

Next, the algorithm will consider some of the precomputed values of $next_P$, which are the possible candidates for $next_P(x, Q_x, y)$, and it will choose the leftmost of all these values (the first in the postfix order). When $n \leq |Q|$, we simply consider $next_P(x_i, Q_i, x_{i-1})$ for each $0 < i \leq n$. Otherwise we proceed as follows. First, as a potential value of $next_P(x, Q_x, y)$, we take $next_P(x_i, Q_i, x_{i-1})$ for each $n - |Q| < i \leq n$. Second, for every element $q \in Q$, using the information in B we find the lowest ancestor z of y such that $next_P(par(z, |Q|), \{q\}, z)$ is non-empty. When such z is already an ancestor of x, or does not exist, we omit this part. Otherwise, for each $0 \leq k < |Q|$, if $par(z, |Q| + k) \geq x$ and $q \in Q_x \circ val_t(x, par(z, |Q| + k))$, we take $next_P(par(z, |Q| + k), \{q\}, par(z, k))$. Third, for every element $q \in Q$, and every $1 \leq k \leq n$, if $q \in Q_x \circ val_t(x, par(y, k))$, we take $next_P(par(y, k), \{q\}, y)$. Finally we choose the leftmost among all the nodes taken as the candidates. This works in time $O(|Q|^3)$, as for $|Q|$ values of q and $|Q|$ values of k we perform operations in time $O(|Q|)$ (in particular we can check in that time whether $q \in Q_x \circ val_t(x, x_i)$, as described in the previous paragraph).

It is easy to see, that all among the candidates are in $set_P(x, Q_x)$. We have to prove, that we really select the leftmost of the elements of $set_P(x, Q_x)$ being to the right of y. Let z' be the value of $next_P(x, Q_x, y)$ and let x_c be the first of $x_1, \dots, x_n$ which is an ancestor of z'. Then we have a sequence of elements $q_c, \dots, q_n \in Q$ such that $q_n \in Q_x$, and $(q_{i+1}, q_i) \in val_t(x_{i+1}, x_i)$ (for each $c \le i < n$), and $\{q_c\} \circ val_t(x_c, z') \in P$. One case is that $c > n - |Q|$; then $z' = next_P(x_c, Q_c, x_{c-1})$, so z' is among the elements of the first kind. Similarly, if $c \le |Q|$, then z' is among the elements of the third kind. Otherwise, some element in the sequence $q_c, \dots, q_{c+|Q|}$ has to repeat. Because the labelling is basic, the element appears twice in a row, and it belongs to Q^l (i.e. $q_r = q_{r+1} \in Q^l$ for some $c \le r < c + |Q|$). Then $z' = next_P(x_r, \{q_r\}, x_{r-|Q|})$ and $q_r \in Q_x \circ val_t(x, x_r)$. But because $q_r \in Q^l$ (and the labelling is basic), we have $q_r \in Q_x \circ val_t(x, x_i)$ for every $0 \le i \le r$. Let z be the lowest ancestor of y such that $next_P(par(z, |Q|), \{q_r\}, z)$ is non-empty. If $x_{r-|Q|} \le par(z, |Q|)$ (i.e. $z = x_i$ for $i \le r - 2|Q|$), then $next_P(par(z, |Q|), \{q_r\}, z)$ is before (or equal to) $x_{r-|Q|}$ in the postfix order, which is strictly before $z' = next_P(x_r, \{q_r\}, x_{r-|Q|})$; additionally $next_P(par(z, |Q|), \{q_r\}, z)$ is in $set_P(x, Q_x)$, which contradicts with the assumption that z' is the first element after y which is in $set_P(x, Q_x)$. Thus $z = x_i$ for $i > r - 2|Q|$, so $next_P(x_r, \{q_r\}, x_{r-|Q|})$ is among the elements taken as candidates of the second kind.

The general situation when x is an arbitrary ancestor of y is very similar. We just consider the zig-zag sequence (as we considered in Section 6.2) instead of the sequence of parents, and we use the information for tda instead of this for par. We have to use the above restricted case between the last two nodes in the sequence, as x is a direct ancestor of x_{n-1}, but not the topmost direct ancestor.

8.2 Path expressions

In this section we prove Theorem 1.2, using the algorithms designed in the previous section. As for node tests, we evaluate all nested node tests in α and we mark in t whether they are satisfied. So we can assume that α is unnested. In particular, evaluating α does not depend on the data; the problem can be stated also for trees without data. We compile α to an automaton $\mathcal{A}$ using Theorem 5.8 (or Theorem 6.2 in the case of expression of FOXPath); this also changes the labels in the data tree t. Let Q be the set of states of $\mathcal{A}$.

Let x be a node and Q_x a subset of Q. We define $set^I(x, Q_x)$ as the set of descendants y of x such that $trans(y, x) \circ Q_x$ contains some initial state. Similarly, we define $set^F(x, Q_x)$ as the set of descendants y of x such that $Q_x \circ trans(x, y)$ contains some accepting state. It follows from Theorem 8.1 that we can enumerate these sets efficiently.

Corollary 8.7
Let t be a data tree and let $\mathcal{A}$ be an automaton with states Q. After an appropriate preprocessing, we can, given a node x of t and a set $Q_x \subseteq Q$, compute all nodes in $set^I(x, Q_x)$ one after another in time

- *preprocessing: $O(|Q|^3 |t| \log |t|)$, each element of $set^I(x, Q_x)$: $O(|Q|^3 \log |t|)$, or*
- *preprocessing: $O(2^{O(|Q|)} |t|)$, each element of $set^I(x, Q_x)$: $O(2^{O(|Q|)})$, or*
- *when t forms a word—preprocessing: $O(|Q|^5 |t|)$, each element of $set^I(x, Q_x)$: $O(|Q|^5)$, or*
- *when $\mathcal{A}$ is basic—preprocessing: $O(|Q|^3 |t|)$, each element of $set^I(x, Q_x)$: $O(|Q|^3)$.*

The same is true of $set^F(x, Q_x)$.

Proof
We use here the same binary tree t' as in the proofs of Corollaries 5.10 and 5.12. Recall that its labels are from the set $R_{Q \times \{1,2\}}$, and are assigned in such a way that $val_{t'}(x, y)$ for any node x and its proper descendant y is also equal to

$$\{((q,1),(p,1)) : (p,q) \in trans(y,x)\} \cup \{((p,2),(q,2)) : (p,q) \in trans(x,y)\}.$$

This means that (for a proper descendant y of x) $trans(y,x) \circ Q_x$ contains some initial state of and only if $val_{t'}(x,y) \cap ((Q_x \times \{1\}) \times (Q_I \times \{1\})) \neq \emptyset$, where Q_I is the set of initial states. Thus $set^I(x, Q_x)$ in t is almost equal to $set_{Q_I \times \{1\}}(x, Q_x \times \{1\})$ in t'. We say „almost", because it is possible that x is in $set^I(x, Q_x)$ and not in $set_{Q_I \times \{1\}}(x, Q_x \times \{1\})$ (as $val_t(x,x)$ is always equal to identity, unlike $trans(x,x)$). So, we can output the nodes using Theorem 8.1, possibly returning also x if $trans(x,x) \circ Q_x$ contains some initial state. The complexity is the same as in the variant of Theorem 8.1 which we use. Recall also from Section 6.1 that if $\mathcal{A}$ is basic, then the labelling of t' is basic. For $set^F(x, Q_x)$ we can do the same, but this time we use $set_{Q_F \times \{2\}}(x, Q_x \times \{2\})$, where Q_F is the set of accepting states. □

Now we show how to find all pairs of nodes (x,y) satisfying a path expression α. There are several types of such pairs, depending on the relationship between x and y:

1. $x = y$,
2. x is a proper ancestor of y,
3. x is a proper descendant of y,
4. x is neither an ancestor nor a descendant of y and it is before y in the postfix order,
5. x is neither an ancestor nor a descendant of y and it is after y in the postfix order.

First consider the pairs of type 1. This type is easy. In the preprocessing step we can check for each node $x = y$ if it satisfies α or not. This is the case when some pair (q_I, q_F) of an initial and an accepting state belongs to $trans(x,x)$, so the checking procedure is trivial (recall from Section 5.3 that for every node x the sets $trans(x,x)$ are already calculated and stored in the tree). We make a list of all such nodes satisfying α, and then return them one after another, reading from the list.

Pairs of type 2 are also not too difficult. Note that a pair (x,y) satisfies α if and only if $y \in set^F(x, Q_I)$, i.e. when from an initial state in x the automaton can reach a final state in y. In the preprocessing step we make a list of nodes x for which $set^F(x, Q_I)$ is nonempty. Then we take consecutive nodes x from the list and consecutive nodes y from $set^F(x, Q_I)$, using Corollary 8.7 to enumerate its elements. Symmetrically for pairs of type 3; this time we require that $x \in set^I(y, Q_F)$.

Now we come to the most complex type 4 (and 5). It is convenient to distinguish the part $set^I_{left}(x, Q_x)$ of $set^I(x, Q_x)$ consisting of only these nodes, which are in the left subtree of x. Similarly let $set^F_{right}(x, Q_x)$ contain only these nodes of $set^F(x, Q_x)$, which are in the right subtree of x. Observe that

$$set^I_{left}(x, Q_x) = set^I(y, trans(y,x) \circ Q_x), \qquad \text{where } y \text{ is the left child of } x, \text{ and}$$
$$set^F_{right}(x, Q_x) = set^F(y, Q_x \circ trans(x,y)), \qquad \text{where } y \text{ is the right child of } x.$$

Thus we can enumerate elements of $set^I_{left}(x, Q_x)$ and $set^F_{right}(x, Q_x)$ with the same delay between the elements as in Corollary 8.7 (recall that $trans(x,y)$ between a node and its child is remembered in the tree).

For each node x we also define two sets of states: $up^I_{left}(x)$ and $down^F_{right}(x)$. The first set contains all the states q which can be reached by the automaton in x, when it starts in an initial state somewhere in the left subtree of x. Similarly, $down^F_{right}(x)$ contains all the states q such that from q in x the automaton can reach an accepting state somewhere in the right subtree of x. These sets can be easily calculated for each x in one bottom-up pass in time $O(|Q|^2|t|)$.

The following lemma follows immediately from the definitions of all our sets.

Lemma 8.8

1. *Let z be any node. Then there exists some node x in the left subtree of z and some node y in the right subtree of z such that (x,y) satisfies α if and only if $up^I_{left}(z) \cap down^F_{right}(z) \neq \emptyset$.*

2. *Let z satisfy the above. Denote $Q_z = up^I_{left}(z) \cap down^F_{right}(z)$ and let y be any node in the right subtree of z. Then there exists some node x in the left subtree of z such that (x, y) satisfies α if and only if $y \in set^F_{right}(z, Q_z)$.*

3. *Let y and z satisfy the above and let x be any node in the left subtree of z. Then (x, y) satisfies α if and only if $y \in set^I_{left}(z, trans(z, y) \circ Q_F)$.*

This lemma gives us a method of returning pairs (x, y) of type 4 satisfying α with a constant delay. In the preprocessing step we create a list of all nodes z satisfying (1). Then we take consecutive nodes z from the list. For each of them we take consecutive y from $set^F_{right}(z, up^I_{left}(z) \cap down^F_{right}(z))$ and for each of them we take consecutive x from $set^I_{left}(z, trans(z, y) \circ Q_F)$. Then between consecutive pairs we make a constant number of queries to Corollaries 8.7 and 5.10, so the delay is small. Note that each pair (x, y) will be returned for exactly one z: for their closest common ancestor.

To return pairs of type 5 we do the same, but we replace left with right.

This finishes the proof of the first, second, and fourth variant of Theorem 1.2. To get the third variant of the theorem, we first encode the data tree t into a tree which forms a word, and we appropriately change the path expression, as described in Section 5.6. Then we use the above for a tree which forms a word.

Bibliography

[BFC00] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical Informatics*, pages 88–94, 2000.

[BK09] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys*, 41:3:1–3:54, January 2009.

[BP] M. Bojańczyk and P. Parys. XPath evaluation in linear time. *Journal of the ACM*. Accepted for publication.

[BP08] M. Bojańczyk and P. Parys. XPath evaluation in linear time. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '08, pages 241–250, New York, NY, USA, 2008. ACM.

[BP10] M. Bojańczyk and P. Parys. Efficient evaluation of nondeterministic automata using factorization forests. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 515–526, Berlin, Heidelberg, 2010. Springer-Verlag.

[CD99] J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation. Technical report, W3C, 1999.

[Col07] T. Colcombet. On factorization forests. Technical Report hal-00125047, Irisa Rennes, 2007.

[Col10] Thomas Colcombet. Factorization forests for infinite words and applications to countable scattered linear orderings. *Theoretical Computer Science*, 411:751–764, January 2010.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, pages 1–6, New York, NY, USA, 1987. ACM.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 95–106. VLDB Endowment, 2002.

[GKP03] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space eficiency. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE'03, pages 379–390, 2003.

[GKP05] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30:444–491, June 2005.

[Gre51] J. A. Green. On the structure of semigroups. *Annals of Mathematics*, 54:163–172, 1951.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *International Conference on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, 2003.

[Kuf] Manfred Kufleitner. The height of factorization forests. In *Mathematical Foundations of Computer Science*, MFCS'08.

[Nev02] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31:39–46, September 2002.

[Par09] Paweł Parys. XPath evaluation in linear time with polynomial combined complexity. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 55–64, New York, NY, USA, 2009. ACM.

[RKR72] R.E. Miller R.M. Karp and A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Symposium on Theory of computing*, pages 125–136, 1972.

[Sim90] I. Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.