XPath Evaluation in Linear Time

MIKOŁAJ BOJAŃCZYK and PAWEŁ PARYS
Warsaw University

We consider a fragment of XPath 1.0, where attribute and text values may be compared. We show that for any unary query φ in this fragment, the set of nodes that satisfy the query in a document t can be calculated in time $O(|\varphi|^3|t|)$. We show that for a query in a bigger fragment with Kleene star allowed, the same can be done in time $O(2^{O(|\varphi|)}|t|)$ or in time $O(|\varphi|^3|t|\log|t|)$. Finally, we present algorithms for binary queries of XPath, which do a precomputation on the document and then output the selected pairs with constant delay.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational Logic*; H.2.3 [**Database Management**]: Languages—*Query Languages*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Tree with data, XML, XPath

1. INTRODUCTION

In this paper, 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 [Gottlob et al. 2005]. The existing papers devoted to improving XPath evaluation can be grouped into two main approaches, as is explained next (see e.g. [Benedikt and Koch 2008] for a survey).

One idea, as used in e.g. [Gottlob et al. 2002] and improved in [Gottlob et al. 2003], is to use dynamic programming; see also [Gottlob et al. 2005]. 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 [Gottlob et al. 2003] have running time $O(|\varphi|^2|t|^4)$.

Another idea is to compile queries into finite-state tree automata, see [Neven 2002] 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

We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.
Work supported by Polish government grant no. N N206 380037.

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 [Gottlob et al. 2005].)

This paper, together with the conference papers on which it is based, [Bojańczyk and Parys 2008] and [Parys 2009], can be seen as a generalization of the automata-theoretic framework to node tests that use attribute and text values. In the terminology of [Benedikt and Koch 2008], 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 [Bojańczyk and Parys 2008]. 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 extension* for this extension of regular XPath, as opposed to the *basic fragment*, which stands for XPath where path expressions are not allowed to use the Kleene star, as in the XPath specification [Clark and DeRose 1999]. The algorithm in [Bojańczyk and Parys 2008] 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 [Parys 2009], an algorithm with linear time data complexity and polynomial time combined complexity was given. This algorithm used the special form of path expressions in the basic fragment, which in fact are less expressive than regular expressions. Hence the algorithm does not work for the regular extension, only for the basic fragment.

There is also a third, unpublished algorithm, which is a simpler version of these in [Bojańczyk and Parys 2008] and [Parys 2009]. It has $O(|t| \log |t|)$ time complexity in the document size $|t|$, polynomial combined complexity, and works for the regular extension as well. Probably among the three 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.

The three algorithms described above are the content of this paper. They are presented in the following theorem.

THEOREM 1.1. *Let t be an XML document and φ a node test of XPath (as defined in section 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*
—*when φ is from the basic fragment—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 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
—when α is from the basic fragment—first pair: $O(|\alpha|^3|t|)$, each next pair: $O(|\alpha|^3)$.

The paper is structured as follows. In Section 2, we present preliminary definitions, the data model, and we define the fragment of XPath considered in this paper. In Section 3, we present a high level overview of the algorithm from Theorem 1.1. The algorithm is then detailed in Sections 4 to 10. Sections 4 to 7 are common to all the three complexities; they reduce Theorem 1.1 to Theorems 7.1 and 7.8. These theorems are then shown in Sections 8, 9, and 10 in three different ways, which gives the three different complexities of the algorithms. Finally, in Section 11 we present a proof of Theorem 1.2.

Major contributions.. The major contributions of this paper are the three XPath evaluation algorithms mentioned in Theorem 1.1. There are some differences between the algorithms, but they all share the following properties: a) the queries are data-aware, i.e. the queries do not correspond to automata over a finite alphabet; and b) when the query is fixed, the evaluation algorithm on a document t runs in time $O(|t|)$ or $O(|t|\cdot\log(|t|))$. Previous algorithms for data-aware fragments of XPath would have at least quadratic data complexity, and conversely, previous algorithms with linear data complexity would ignore data.

2. DATA MODEL AND XPATH

2.1 Data model

In this section we define the data model. We represent an XML document as a 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 Σ—every node has a label from the set Σ. Moreover

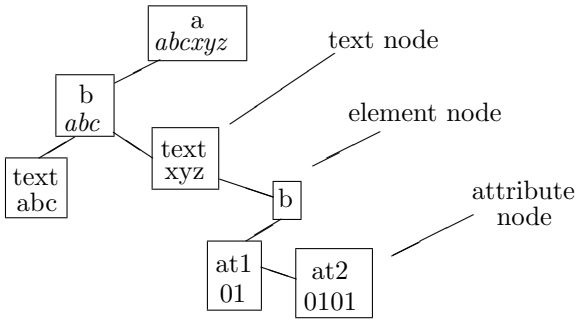


Fig. 1. Example data tree (string values in italic are not remembered)

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,[1] 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 most of the time we will be dealing with data trees, we will sometimes write tree instead of data tree.

Consider for instance the following XML document:

$$\begin{array}{l}\langle \mathtt{a}\rangle\\ \quad \langle \mathtt{b}\rangle \mathtt{abc} \langle /\mathtt{b}\rangle \mathtt{xyz}\\ \quad \langle \mathtt{b\ at1} = \text{"}\mathtt{01}\text{"}\ \mathtt{at2} = \text{"}\mathtt{0101}\text{"}\rangle \langle /\mathtt{b}\rangle\\ \langle /\mathtt{a}\rangle\end{array}$$

The data tree representing this document uses labels $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{at1}, \mathtt{at2}, \mathtt{text}\}$. 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.

Trees will be denoted by letters t, s. Nodes will be denoted by x, y, z. String values will be denoted by d. 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.

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 $\langle$ or ".

2.2 XPath

In this section we define the fragments of XPath that are used in the paper. There are two fragments: the *basic fragment* and the *regular extension*. The basic fragment is almost a fragment called FOXPath in [Benedikt and Koch 2008]. Basically, it contains queries that may navigate in a tree and compare string values. The spec-

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

ification [Clark and DeRose 1999] of XPath 1.0 contains a lot of constructs, which can be easily added (like type conversions, etc.), but we omit them from this paper to avoid going into technicalities. The constructs of full XPath 1.0 which are important for evaluation complexity, and which are not handled here, are: aggregates, manipulating integers and position arithmetic.

The only difference between the basic fragment and the regular extension is that the second allows Kleene star. The regular extension is not in the XPath specification, but it is an often considered extension.

In XPath, the primitives employed for navigation along the tree structure are called *axes*. We consider the following *one-step* axes: $\texttt{to}-\texttt{left}$, $\texttt{to}-\texttt{right}$ and their inverses $\texttt{from}-\texttt{left}$, $\texttt{from}-\texttt{right}$. They correspond to going to and from the left and the right child. Moreover we consider their transitive-reflexive closures, called *multistep* axes: $\texttt{to}-\texttt{left}^*$, $\texttt{to}-\texttt{right}^*$, $\texttt{from}-\texttt{left}^*$, $\texttt{from}-\texttt{right}^*$, $(\texttt{to}-\texttt{left} + \texttt{to}-\texttt{right})^*$, $(\texttt{from}-\texttt{left} + \texttt{from}-\texttt{right})^*$. 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 $\texttt{to}-\texttt{left}^*$, 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 $a \in \Sigma$ is a node test, which selects nodes with a label a.

— Node tests admit negation, conjunction and disjunction.

— If α, β are path expressions, ϑ is a string constant and $\texttt{RelOp} \in \{=, \leq, <, >, \geq, \neq\}$, then

$$\alpha \ \texttt{RelOp} \ \beta \quad \text{and} \quad \alpha \ \texttt{RelOp} \ \vartheta$$

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 $\texttt{RelOp}$. 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 ϑ satisfy the relation $\texttt{RelOp}$. The inequalities $\leq, <, >, \geq$ correspond to the lexicographic order of strings (all the results hold as well for the order of integer numbers).

— 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 the regular extension (but not in the basic fragment) 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` ϑ 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 the regular extension 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 the basic fragments, since Kleene star is not available.

When referring to XPath, we mean the fragments above. 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, even in the basic fragment. 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). The `following` axis can be written as $\texttt{parent}^* \cdot \texttt{next-sibling} \cdot \texttt{next-sibling}^* \cdot \texttt{child}^*$ in the unranked tree, which should translate to $(\varepsilon + (\texttt{from-left} + \texttt{from-right})^* \cdot \texttt{from-left}) \cdot \texttt{to-right} \cdot \texttt{to-right}^* \cdot (\varepsilon + \texttt{to-left} \cdot (\texttt{to-left} + \texttt{to-right})^*)$. Observe that the multistep axes $\texttt{to-left}^*$ and $\texttt{from-left}^*$ are not necessary in this translation, but we add them for symmetry.

3. PROOF STRATEGY

In this section we describe the high-level structure of our linear time algorithms. Recall that the three algorithms have a common part; the discussion below concerns this common part. The algorithms diverge after Section 7.

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 Σ. 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 $a \in \Sigma$; 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 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 Σ 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 3.1. 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 α' `RelOp` β' where the path expressions α' and β' are unnested.

The same approach succeeds with node tests α `RelOp` ϑ: it is enough to evaluate all node tests which appear in α and then α' `RelOp` ϑ for some unnested α' on an appropriate data tree t'. We can even go further: the node test α' `RelOp` ϑ can be easily simulated by one of the other kind α'' `RelOp` β, 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 ϑ 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 α'' `RelOp` β in t'' should return the same as α' `RelOp` ϑ 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[2] $|t| \geq |\vartheta|$, we have $|t''| \leq |t| \cdot 2 = O(|t|)$. We also have $|\alpha'' \texttt{ RelOp } \beta| = O(|\alpha \texttt{ RelOp } \vartheta| - |\varphi_1| - \ldots - |\varphi_n|)$.

Concluding, only the construction α `RelOp` β, 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 3.2. *Assume we have an algorithm which, for unnested α and β, evaluates the node test α* `RelOp` *β in a data tree t in time $T(|\alpha \texttt{ RelOp } \beta|, |t|)$. Then there is an algorithm evaluating any XPath node test φ in a data tree t in time $O(T(O(|\varphi|), O(|t|)))$.*

4. PREPARING THE TREE

Before we come to solving node tests α `RelOp` β 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.

[2]A more careful analysis shows that Theorems 1.1 and 1.2 stay true even without this assumption.

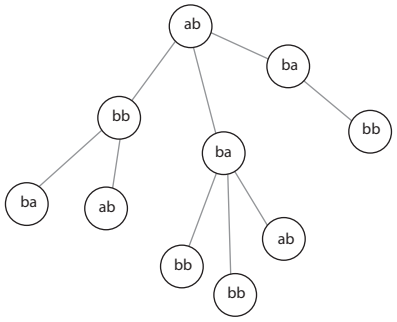
First, we say how a data tree is stored in memory by the algorithm. An initial situation is that we have a record for each node, called the *node record*. This record contains the array of node labels, the string value (in text and attribute nodes), as well as pointers to the node records of the left 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 x and y be two nodes in a data 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).

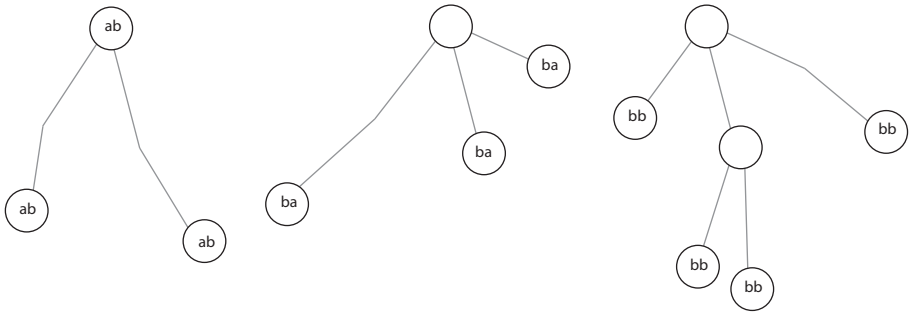
Let the *class of* d be the set of all closest common ancestors of any two nodes x and y having string value d. In particular every node with a string value d is in the class of d (since a node x is the closest common ancestor of x, x). In the evaluation algorithm, it will be convenient to reason about 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 only 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.



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. 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 4.1. *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 4.2. *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 4.1 AND 4.2. 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 [2003] 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.

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 4.1 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 with equal 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 4.2 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. □

To calculate d-skeletons we also need operations described by the following fact.

FACT 4.3. *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 [1984] show an algorithm for queries of type 1 (a simpler algorithm doing the same was given later by Bender and Farach-Colton [2000]). 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.

We are now ready to prove the following proposition.

PROPOSITION 4.4. *The skeleton representation of a data tree t can be calculated in time $O(|t|)$.*

PROOF. From Proposition 4.1 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 4.3 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. 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. The automata will be working on binary versions of trees.

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 $(\{1, \ldots, k\} \times \Sigma) \cup \{\texttt{to-left}, \texttt{to-right}, \texttt{from-left}, \texttt{from-right}\}$, 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 5.1. *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 $O(|\alpha|^2)$.*

For path expressions from the basic fragment we get automata of a special form, described by the following definition and lemma.

Definition 5.2. An automaton $\mathcal{A}$ is called *basic*, when its states can be numbered $Q = \{q_1, \ldots, q_n\}$ in such way that transitions from q_i to q_j exist only for $i \leq j$.

LEMMA 5.3. *When α is an unnested path expression from the basic fragment, the automaton constructed in Lemma 5.1 is basic.*

PROOF. When translating a regular expression into an automaton, only the Kleene star creates loops, and the Kleene star is forbidden in the basic fragment. The multistep axes causes trivial loops. □

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 5.4. 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 5.5. *Let t be a data tree and α an unnested path expression. We can calculate, in time $O(|t||\alpha|^3)$, 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. When α is from the basic fragment, $\mathcal{A}$ is basic.*

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 5.6. *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 follow from the following easy proposition, which is used implicitly also in the further sections.

PROPOSITION 5.7. *We can calculate in time $O(|Q|^3)$ the transitive closure of a given set of state pairs (understood as a relation on states) or the composition of two given sets of state pairs.*

PROOF OF LEMMA 5.6. 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 of R_c with $down(x_1)$ and with 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 5.5. First, let $\mathcal{A}'$ be the automaton constructed in Lemma 5.1 from the path expression α. Then, we use Lemma 5.6 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). In the case of a basic path expression (and a basic automaton), there are only two small differences. First, to the tree s we take the elements corresponding only to pairs (q_i, q_j) for $i \leq j$. Because $\mathcal{A}'$ is basic, only such pairs may be in the sets $loop$. Second, in $\mathcal{A}$ we add only transitions between states q_i, q_j for $i \leq j$, hence $\mathcal{A}$ is basic.

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 $trans_{\mathcal{A},s}(x, y)$.

6. 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 data 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 5.5).[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 4.2 (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

[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 $O(|Q|^3|t|)$, since this is the complexity of the preprocessing step described in the previous section.

noting that we get this complexity even for the regular extension. This contrasts with the node tests $\alpha = \beta$, which can be evaluated faster when the path expressions are from the basic fragment and not from the regular extension.

7. EQUALITY TESTS—THE COMMON PART

In this section, we identify the main difficulty in calculating node tests $\alpha = \beta$. The strategy will be as follows: first we define snippets and trivial snippets. Then we show how to find some set of snippets representing the solution of $\alpha = \beta$. Finally we show that having a set of trivial snippets is enough to solve the node test $\alpha = \beta$. Transformation of any snippets into trivial snippets is postponed to the next sections. We also require here some fast method of calculating automata runs, which also will be shown in the next sections.

From Theorem 5.5 we know that α and β can be recognized by automata (which are basic when α and β are from the basic fragment). 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 following theorem, which is proved in the Sections 8, 9, and 10; in each of them an algorithm with different complexity is given.

THEOREM 7.1. *For a data tree t and an automaton $\mathcal{A}$ we can, after preprocessing, answer queries of the form: for 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*[6]

$$prec(x, y, Q_y) = \{p \in Q : \exists_{q \in Q_y} (p, q) \in trans(x, y)\}.$$

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 the automaton $\mathcal{A}$ is basic—preprocessing:* $O(|Q|^3|t|)$, *query:* $O(|Q|^3)$.

Here we mean that there are three algorithms, one for each of the listed complexities. Observe that *prec* is compositional in the following sense.

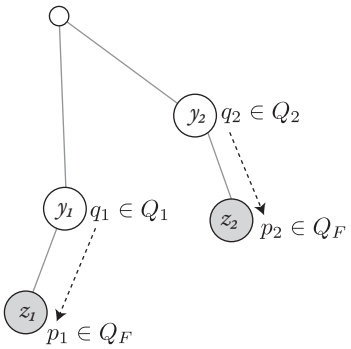
PROPOSITION 7.2. *Let z be any node on the simple path from x to y. Then*

$$prec(x, y, Q_y) = prec(x, z, prec(z, y, Q_y)).$$

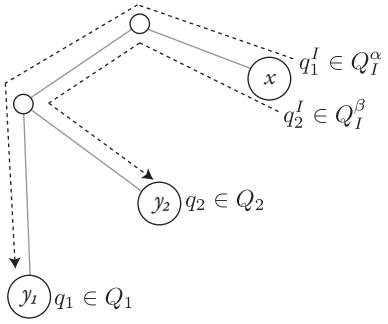
[5]The assumption that x is an ancestor or a descendant of y can be easily removed, but we do not need the stronger version of the lemma.

[6]One may wonder why we use the *prec* sets instead of simply calculating $trans(x, y)$. The reason is that in the third version of the algorithm it would be slower: there would be $|Q|^4$ instead $|Q|^3$ in the complexities.

Now we define snippets. A *snippet*[7] is a tuple (y_1, y_2, Q_1, Q_2) where y_1, y_2 are nodes of the tree t and Q_1 and Q_2 are sets of states of the automaton $\mathcal{A}$. A snippet 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 $p_i \in Q_F$. This is presented in the picture below (the dotted lines depict automaton paths, the highlighted nodes carry the same string value).



Namely, we say that the snippet *selects* a node x when $(q_1^I, q_1) \in trans(x, y_1)$ and $(q_2^I, q_2) \in trans(x, y_2)$ for some $q_1 \in Q_1$, $q_2 \in Q_2$ and $q_1^I \in Q_I^\alpha$, $q_2^I \in Q_I^\beta$ 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 often use snippets in which both state sets are singletons; in such case we simply write (y_1, y_2, q_1, q_2). In our snippets y_1 will often be an ancestor of y_2; we call such snippets *vertical*, y_1 a *high node* of the snippet, and y_2 its *low node*. A snippet is called *trivial* when $y_1 = y_2$ and both state sets are singletons.

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. Two sets of snippets

[7] In [Bojańczyk and Parys 2008] and [Parys 2009] we used the word *bracket* instead of *snippet*.

are *equivalent* if they select the same set of nodes. Our algorithm will first create a sound and complete set of snippets. Then it will be converting the snippets into simpler ones, ensuring that the set of snippets is equivalent to the previous one, so it is always 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 is very easy to construct some sound and complete set of snippets. We simply take a snippet (y_1, y_2, Q_F, Q_F) for each pair y_1, y_2 of nodes with the same string value. It is obviously sound and complete, however it 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 7.3. *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*

—$O(|Q|^3|t| \log |t|)$, or
—$O(2^{O(|Q|)}|t|)$, or
—when the automaton $\mathcal{A}$ is basic—in time $O(|Q|^3|t|)$.

Moreover, there will be $O(|t|)$ snippets, all of them vertical.

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 $prec(x, x_i, class^{down}(x_i, d))$ for $i = 1, 2$ and if the string value of x is d, it is also a union with $prec(x, x, Q_F)$, 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 Theorem 7.1. In total we have $O(|t|)$ nodes in all d-skeletons, hence we get the desired complexity (depending on which version of Theorem 7.1 is used).

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 $y^{\uparrow}, y_{\downarrow}$ such that $y^{\uparrow}$ is the parent of $y_{\downarrow}$ in the d-skeleton we take to the set a snippet $(y^{\uparrow}, y_{\downarrow}, class(y^{\uparrow}, d), class(y_{\downarrow}, d))$. Additionally, for each d and each y in the d-skeleton we take a snippet $(y, y, class(y, d), class(y, d))$.[8]

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

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

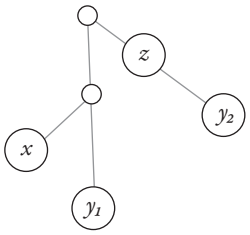
set is complete. Take any node x selected by $\alpha = \beta$. Let z_α, z_β be nodes with the same string value d such that z_α (respectively, z_β) is reachable from x using α (β). Let y_α be the first node in the d-skeleton on the simple path from x to z_α; similarly for β. If $y_\alpha = y_\beta$ then 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 $y^\uparrow = y_\alpha$, $y_\downarrow = y_\beta$ or $y^\uparrow = y_\beta$, $y_\downarrow = y_\alpha$. □

In the next stage, the algorithm should simplify the set of snippets, so that all snippets become trivial. First we give a few ways how the snippets can be split, following directly from the definitions.

PROPOSITION 7.4. *Let z be any node on the simple path from y_1 to y_2. Then a snippet (y_1, y_2, Q_1, Q_2) is equivalent to the set of two snippets*

$$(y_1, z, Q_1, prec(z, y_2, Q_2)) \qquad \text{and} \qquad (z, y_2, prec(z, y_1, Q_1), Q_2).$$

This is because when we have paths from some x to y_1 and to y_2, at least one of them has to lead through z. The situation is illustrated below, with one of the possible placements of the node x.

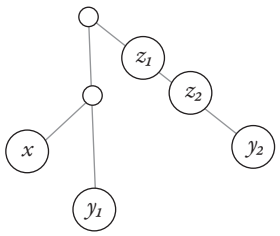


We can also do the split in another, slightly stronger way.

PROPOSITION 7.5. *Let z_1, z_2 be two nodes connected by an edge and both lying on the simple path from y_1 to y_2 in such way that y_1 is closer to z_1 than to z_2. Then a snippet (y_1, y_2, Q_1, Q_2) is equivalent to the set of two snippets*

$$(y_1, z_1, Q_1, prec(z_1, y_2, Q_2)) \qquad \text{and} \qquad (z_2, y_2, prec(z_2, y_1, Q_1), Q_2).$$

Now again, for any x, either the path from x to y_2 crosses z_1, or the path from x to y_1 crosses z_2. The situation is illustrated below.



The next property allows us to remove unnecessary snippets.

PROPOSITION 7.6. *Let $(y_1^\uparrow, y_\downarrow, Q_1^\uparrow, Q_\downarrow)$ and $(y_2^\uparrow, y_\downarrow, Q_2^\uparrow, Q_\downarrow)$ be two vertical snippets such that $y_1^\uparrow$ is an ancestor of $y_2^\uparrow$ and $Q_2^\uparrow \subseteq prec(y_2^\uparrow, y_1^\uparrow, Q_1^\uparrow)$. Then the first snippet is equivalent to the set of both these snippets.*

This is because each run ending in a state from $Q_2^\uparrow$ at $y_2^\uparrow$ can be extended to a run ending in a state from $Q_1^\uparrow$ at $y_1^\uparrow$. Finally, we have yet another easy property, saying that each snippet can be replaced by snippets with singleton state sets.

PROPOSITION 7.7. *Any snippet (y_1, y_2, Q_1, Q_2) is equivalent to the set of snippets (y_1, z_1, q_1, q_2) for all $q_1 \in Q_1$, $q_2 \in Q_2$.*

The following theorem will be shown in Sections 8, 9, and 10.

THEOREM 7.8. *For a data tree t, an automaton $\mathcal{A}$, and a set S of $O(|t|)$ vertical snippets we can calculate an equivalent set S' of $O(|Q|^2|t|)$ trivial snippets. It can be done in time*

—$O(|Q|^3|t| \log |t|)$, or
—$O(2^{O(|Q|)}|t|)$, or
—when the automaton $\mathcal{A}$ is basic—in time $O(|Q|^3|t|)$.

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

LEMMA 7.9. *For a data tree t, an automaton $\mathcal{A}$, and a set S of $O(|Q|^2|t|)$ trivial snippets we can calculate, in time $O(|Q|^3|t|)$, the set of nodes selected by the snippets.*

PROOF. First, for any node x we define a set $double(x)$ of state pairs $(p_\uparrow, p_\downarrow)$ such that for some snippet $(y, y, q^\uparrow, q_\downarrow)$ from S it holds

$$(p_\uparrow, q_\uparrow) \in trans(x, y) \qquad \text{and} \qquad (p_\downarrow, q_\downarrow) \in trans(x, y).$$

Observe that a node x is selected 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_1, x_2. It should contain (for $i = 1, 2$) all pairs $(p_\uparrow, p_\downarrow)$ such that for some states $(q_\uparrow, q_\downarrow) \in double^{down}(x_i)$ both pairs $(p_\uparrow, q_\uparrow)$ and $(p_\downarrow, q_\downarrow)$ 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 $(p_\uparrow, q_\downarrow)$ such that for some $q_\uparrow$ there is $(q_\uparrow, q_\downarrow) \in double^{down}(x_i)$ and $(p_\uparrow, q_\uparrow) \in trans(x, x_i)$. In a 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 $(q^\uparrow, q_\downarrow)$ for snippets $(x, x, q^\uparrow, q_\downarrow)$ from S. The top-down pass is similar. □

Summing up, by composing the algorithm given by the above lemmas and theorems, we get Theorem 1.1. All the lemmas are proved above, but Theorems 7.1 and 7.8 are not. Their proofs are given in Sections 8, 9, and 10. Each of these sections give a different complexity of the algorithm.

8. LINEAR-LOGARITHMIC ALGORITHM

In this section we show the first parts of Theorems 7.1 and 7.8: we give an algorithm with $O(|Q|^3|t|\log|t|)$ or $O(|Q|^3\log|t|)$ complexities.[9] This gives an $O(|Q|^3|t|\log|t|)$ algorithm for calculating node tests from the regular extension.

Before we do this, we complement the previous sections with two simplifications possible in a linear-logarithmic algorithm. First, in Propositions 4.1 and 4.2 we can calculate the suffix array using a standard and a little bit simpler $O(|t|\log|t|)$ algorithm by Karp et al. [1972], instead of the linear time algorithm. Second, we can also significantly simplify the algorithm hidden in Fact 4.3, which allows us to find the closest common ancestor of given nodes. 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.

8.1 Precomputing automaton runs

First we show a proof of Theorem 7.1, i.e. that after appropriate preprocessing we can run an automaton in time logarithmic in the length of its input. This is a straightforward divide and conquer approach.

Fix an automaton $\mathcal{A}$ with states Q and a tree t. Let K be the greatest number such that 2^K is not greater than the height of the data tree t. It holds $K = O(\log|t|)$. For every node x of t and every $0 \le k \le K$ we remember a pointer to its ancestor y which is 2^k edges above x. Together with the pointer we remember $trans(x,y)$ and $trans(y,x)$. This information can be easily calculated in time $O(|Q|^3|t|K)$: to find a node 2^k edges above from x, we twice go 2^{k-1} edges up using previously calculated pointers. Also $trans(x,y)$ is the composition of these values remembered for $k-1$.

Now consider a query step. First observe the following.

PROPOSITION 8.1. *When the distance between a node x and its ancestor or its descendant y is 2^k, we can calculate $prec(x,y,Q_y)$ in time $O(|Q|^2)$. When Q_y contains only one state, it can be done in time $O(|Q|)$.*

As the set $trans(x,y)$ is stored, it is enough to take all p such that $(p,q) \in trans(x,y)$ for some $q \in Q_y$.

Let now y be any ancestor of x (the case of a descendant is completely symmetric). Consider the nodes $x = x_0 > x_1 > \cdots > x_n = y$, where x_{i+1} is 2^k edges above x_i for the greatest number k such that $x_{i+1} \ge y$. In other words we go from x to y using our pointers: we always use a pointer to the highest ancestor which is still a descendant of y. 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 $n \le K+1$. We take $Q_n = Q_y$ and we consecutively calculate $Q_i = prec(x_i, x_{i+1}, Q_{i+1})$ for every i between $n-1$ and 0, using Proposition 8.1; this takes time $O(|Q|^2\log|t|)$. Due to Proposition 7.2 it holds $Q_0 = prec(x,y,Q_y)$.

[9] Some parts of the algorithm have $|Q|^2$ instead of $|Q|^3$ in the complexity. For us it is not important, as it does not change the overall complexity.

8.2 Simplifying the snippets

Now we come to a proof of Theorem 7.8. We have a set of $O(|t|)$ vertical snippets and we want to create an equivalent set of $O(|Q|^2|t|)$ trivial snippets. We do that in two steps. During the processing, every snippet is remembered in its low node.

Step 1. After this step we want to have trivial snippets and 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).

Snippets of the form $(y, y, Q^\uparrow, Q_\downarrow)$ are easily converted into at most $O(|Q|^2)$ trivial snippets: $(y, y, q^\uparrow, q_\downarrow)$ for every $q^\uparrow \in Q^\uparrow$, $q_\downarrow \in Q_\downarrow$.

Now we handle snippets $(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$ where $y^\uparrow$ is a proper ancestor of $y_\downarrow$. Like previously we find the nodes $y_\downarrow = y_0 > y_1 > \cdots > y_n = y^\uparrow$ where y_{i+1} is 2^k edges above y_i for the greatest number k such that $y_{i+1} \geq y^\uparrow$. It holds $n \leq K + 1$. We consecutively calculate the sets $Q^i_\downarrow = prec(y_i, y_\downarrow, Q_\downarrow)$ and $Q^\uparrow_i = prec(y_i, y^\uparrow, Q^\uparrow)$ observing that

$$Q^i_\downarrow = prec(y_i, y_{i-1}, Q^{i-1}_\downarrow) \qquad \text{for } 0 < i \leq n, \quad Q^0_\downarrow = Q_\downarrow, \text{ and}$$
$$Q^\uparrow_i = prec(y_i, y_{i+1}, Q^\uparrow_{i+1}) \qquad \text{for } 0 \leq i < n, \quad Q^\uparrow_n = Q^\uparrow.$$

For each i it is done in time $O(|Q|^2)$, as described by Proposition 8.1. Then we replace the original snippet by snippets $(y_{i+1}, y_i, q^\uparrow_{i+1}, q^i_\downarrow)$ for all $q^\uparrow_{i+1} \in Q^\uparrow_{i+1}$, $q^i_\downarrow \in Q^i_\downarrow$, $0 \leq i < n$; we get an equivalent set due to Propositions 7.4 and 7.7.

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 $(y^\uparrow, y_\downarrow, q^\uparrow, q_\downarrow)$ where the distance between $y_\downarrow$ and $y^\uparrow$ is 2^k for some $k > 0$. Let y be the node exactly in the middle between them (2^{k-1} edges above $y_\downarrow$). We replace our snippet by snippets $(y^\uparrow, y, q^\uparrow, q)$ for all $q \in prec(y, y_\downarrow, q_\downarrow)$ and by snippets $(y, y_\downarrow, q, q_\downarrow)$ for all $q \in prec(y, y^\uparrow, q^\uparrow)$; Propositions 7.4 and 7.7 cause the equivalence. This snippets are processed again later, when all snippets of size 2^k are already removed. Similarly snippets for $k = 0$ (where $y^\uparrow$ is the parent of $y_\downarrow$) are replaced by trivial snippets in $y^\uparrow$ and $y_\downarrow$ (Proposition 7.5 is used instead of 7.4).

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 $O(|Q|^3|t| \log |t|)$.

9. LINEAR ALGORITHM FOR THE REGULAR EXTENSION (TAPES CONSTRUCTION)

In this section, like in the previous one, we consider the regular extension, and not the basic fragment of XPath. We describe an algorithm with linear data complexity, but with exponential combined complexity. We prove the second parts of Theorems 7.1 and 7.8. This section is based on the techniques from [Bojańczyk and Parys 2008], but is different in that it uses deterministic automata instead of monoids. The tape construction that is used comes from [Bojańczyk 2009].

In Section 9.1, 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 $L \subseteq \Sigma^*$, recognized by a deterministic automaton $\mathcal{D}$. For any word $a_1 \cdots a_n \in \Sigma^*$ 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 Section 9.2 we show how the results can be applied in a tree and we prove Theorem 7.1. Finally, in Section 9.3 we prove Theorem 7.8.

9.1 Tape construction

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 $\mathcal{A}$). We do not use at all data values in this section, so letter d is used only for states, not for data values. 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 $val_w(x, d, y)$ 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 $val_w(x, d, y)$, 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 and by a node number. 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 2 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 $O(|Q||w|)$. 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.

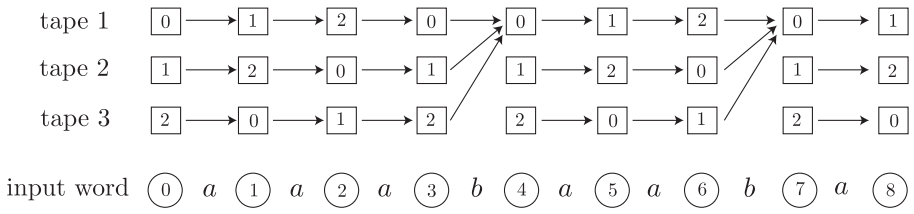


Fig. 2. The tape construction. In this example, the automaton $\mathcal{D}$ has input alphabet $\{a, b\}$ and its state $d \in \{0, 1, 2\}$ holds the number of a's since the last b, modulo 3. The arrows show which tape joins which tape. Note how in node 4 (also in node 7), both tapes 2 and 3 join tape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

9.2 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 *val* 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 *val* is defined analogously to the word case; we omit the subscript t since the tree t is fixed.

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 will use the following operation on the trees s_k.

FACT 9.1. *For an arbitrary (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 4.3. We unravel t into a binary tree s using the first child / next sibling encoding. Additionally we remember the rightmost child of each node. Then we have to find in s the closest common ancestor of y and the rightmost child of x. □

The key property of the information above is that it allows to compute *val* in constant time. Assume we have two nodes $x \leq y$ and a state $d \in D$ and we want to calculate $val(x, d, y)$. We find which tape in node x contains state d. As in the word case (Section 9.1), 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 9.1 (as y may be not a node of s_k, we first need to move to its nearest ancestor which is in s_k). This proves the following lemma.

LEMMA 9.2. *For any two nodes $x \leq y$ and a state $d \in D$ the value $val(x, d, y)$ can be evaluated in time $O(|D|)$.*

Now see how Theorem 7.1 follows from this lemma. We take

$$D = (P(Q) \times \{\emptyset\}) \cup (\{\emptyset\} \times P(Q)) \subseteq P(Q) \times P(Q) \qquad \text{and} \qquad A = P(Q^2) \times P(Q^2),$$

where Q is the set of states of the automaton $\mathcal{A}$ from Theorem 7.1. So an element of D is a pair of two state sets, one of which is empty. The first is used to simulate runs of $\mathcal{A}$ going down, the second—runs of $\mathcal{A}$ going up. We have $|D| = O(2^{|Q|})$. We label an edge from any x to its child y by a pair $(trans(x, y), trans(y, x))$. The transition in $\mathcal{D}$ is an appropriate application of these *trans* relations to the sets of states: the transition from (Q_1, Q_2) reading letter (R_1, R_2) leads to a state

$$(\{q : (p, q) \in R_1, p \in Q_1\}, \{p : (p, q) \in R_2, q \in Q_2\}).$$

Observe that one of these sets is empty, since one of Q_1, Q_2 was empty.

Now consider a query $prec(x, y, Q_y)$ from Theorem 7.1. When the nodes satisfy $x \leq y$, for each state q we can easily check, if it is in $prec(x, y, Q_y)$: we calculate $(Q_1, Q_2) = val(x, (\{q\}, \emptyset), y)$ and we check, whether the sets Q_1 and Q_y have nonempty intersection (which means that from q at x the automaton $\mathcal{A}$ can reach some state from Q_y at y). In the case when $y \leq x$ the set $prec(x, y, Q_y)$ can be found on the second coordinate of $val(y, (\emptyset, Q_y), x)$. The complexity is $O(|D||Q|) = O(2^{O(|Q|)})$. Notice that we need not to explicitely construct the automaton $\mathcal{D}$. Having its state and an input letter we may quickly find its next state directly from its definition.

9.3 Simplifying the snippets

In this section we use the tapes construction to convert a set of arbitrary snippets into an equivalent set of trivial snippets, in time linear in $|t|$. Here we take $\mathcal{D}$, the labeling of edges and all the information as in the previous subsection. Recall that the initial set contains $O(|t|)$ snippets. We use the following two steps to simplify the set of snippets.

Step 1. After this step in the set there will be only snippets $(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$ in which the tape containing $(\emptyset, Q^\uparrow)$ at $y^\uparrow$ is not reset between $y^\uparrow$ and $y_\downarrow$.

Take any original snippet $(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$. Recall that all snippets which we have are vertical, i.e. $y^\uparrow$ is an ancestor of $y_\downarrow$. We find a tape containing $(\emptyset, Q^\uparrow)$ at $y^\uparrow$. As in the previous subsection, using the additional information we can find a sequence of nodes

$$y^\uparrow = x_1 \leq y_1 < x_2 \leq y_2 < \cdots < x_n \leq y_n = y_\downarrow, \qquad n \leq K,$$

where the current tape of the run (staring from $(\emptyset, Q^\uparrow)$ at $y^\uparrow$) changes. Precisely, between x_i and y_i the run stays on the same tape, while between y_i and x_{i+1} it changes the tape (x_{i+1} is a child of y_i). Note that the run has $\emptyset$ on the first coordinate at each node. For each $1 \leq i \leq n$ we use Theorem 7.1 to calculate the sets

$$Q_i^\uparrow = prec(x_i, y^\uparrow, Q^\uparrow) \quad \text{and} \quad Q_\downarrow^i = prec(y_i, y_\downarrow, Q_\downarrow).$$

It takes time $O(2^{O(|Q|)}K) = O(2^{O(|Q|)})$, because each of them is calculated in time $O(2^{O(|Q|)})$.[10]

We replace the snippet $O(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$ by the snippets $(x_i, y_i, Q_i^\uparrow, Q_\downarrow^i)$ for all $1 \leq i \leq n$. They are equivalent due to Proposition 7.5. By definition the tape containing $(\emptyset, Q_i^\uparrow)$ at x_i is not reset until y_i, so the snippets are of the proper form.

Step 2. After this final step we want to have only trivial snippets.

The key property is that when we have two snippets $(y_1^\uparrow, y_\downarrow, Q_1^\uparrow, Q_\downarrow)$ and $(y_2^\uparrow, y_\downarrow, Q_2^\uparrow, Q_\downarrow)$ where $(\emptyset, Q_1^\uparrow)$ at $y_1^\uparrow$ and $(\emptyset, Q_2^\uparrow)$ at $y_2^\uparrow \geq y_1^\uparrow$ are on the same tape, then the second snippet can be removed (assuming that this tape is not reset between $y_1^\uparrow$ and $y_2^\uparrow$, which is true for all the snippets we have now). This property follows from

[10] In fact the sets $Q_i^\uparrow$ (but not $Q_\downarrow^i$) can be just read from the tapes: on the current tape at node x_i there is $(\emptyset, Q_i^\uparrow)$.

Proposition 7.6 because $prec(y_2^\uparrow, y_1^\uparrow, Q_1^\uparrow) = Q_2^\uparrow$ (as $\mathcal{D}$ from $(\emptyset, Q_1^\uparrow)$ at $y_1^\uparrow$ reaches $(\emptyset, Q_2^\uparrow)$ at $y_2^\uparrow$). Thus for each $y_\downarrow$ 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 $y_\downarrow$ starting from the lowest nodes and ending in the root. Let y be the parent of $y_\downarrow$. We replace any snippet $(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$ by two snippets

$$(y^\uparrow, y, Q^\uparrow, prec(y, y_\downarrow, Q_\downarrow)) \quad \text{and} \quad (y_\downarrow, y_\downarrow, prec(y_\downarrow, y^\uparrow, Q^\uparrow), Q_\downarrow) \ .$$

They are equivalent due to Proposition 7.5. The second one is almost trivial, but the state sets are not singletons; it can be replaced by trivial snippets due to Proposition 7.7. The first one is processed again, when we are in the node y. Note that it still satisfies the property that the tape containing $(\emptyset, Q^\uparrow)$ at $y^\uparrow$ is not reset until y. The value $prec(y_\downarrow, y^\uparrow, Q^\uparrow)$ is written at $y_\downarrow$ on the second coordinate of the tape containing $(\emptyset, Q^\uparrow)$ at $y^\uparrow$, so it can be found in time $O(|Q|)$. The other value, $prec(y, y_\downarrow, Q_\downarrow)$, is computed by hand in time $O(|Q|^2)$, as y is the parent of $y_\downarrow$. This gives the total complexity $O(4^{|Q|}|Q|^2|t|) = O(2^{O(|Q|)}|t|)$.

10. POLYNOMIAL COMBINED COMPLEXITY FOR THE BASIC FRAGMENT

Now we switch to the basic fragment. In this section we show Theorems 7.1 and 7.8 in the case when $\mathcal{A}$ is a basic automaton (it simulates path expressions from the basic fragment). This gives an $O(|Q|^3|t|)$ algorithm for calculating node tests from the basic fragment.

10.1 Precomputing automaton runs

First we show a proof of Theorem 7.1, i.e. that after appropriate preprocessing we can run a basic automaton in time not depending on the length of its input.

Fix a basic automaton $\mathcal{A}$ with states Q and a data tree t. A first component of our data structure are the following two functions. For every node x of t and every two states p, q we define $first^{up}(x, p, q)$ as a pointer to the nearest ancestor y of x such that $(p, q) \in trans(x, y)$. 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 x. Similarly let $first^{down}(x, p, q)$ be a pointer to the nearest ancestor y of x such that $(p, q) \in trans(y, x)$. Notice the broken symmetry here: although $first^{up}$ describes runs of the automata going up in the tree and $first^{down}$ these going down, both of them contain pointers to ancestors. Intuitively, pointers to descendants are too costly to store, because there are multiple branches of the tree. The following lemma shows that these functions can be efficiently calculated.

LEMMA 10.1. *We can calculate the functions $first^{down}$ and $first^{up}$ in time $O(|Q|^3|t|)$.*

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

Before we come to the proof of Theorem 7.1, we give some intuitions behind it. It will be important that a basic automaton does not have nontrivial cycles (since the Kleene star is not allowed in path expressions). Every run between distant nodes has to use a multistep axis, which means that it 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 individual steps. Similarly for a run going downwards, we want to reach such a state as quickly as possible, then we go down staying in this state as long as possible, and finally do some transitions described by $first^{down}$.

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

For every node x and its topmost direct ancestor y we remember in x the sets $trans(x, y)$ and $trans(y, x)$. 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 10.2. *When a node y is the topmost direct ancestor of a node x, or vice-versa, we can calculate $prec(x, y, Q_y)$ in time $O(|Q|^2)$. When Q_y contains only one state, it can be done in time $O(|Q|)$.*

We will now show how to calculate $prec(x, y, Q_y)$ in the case when y is any direct ancestor of x. Suppose that y can be reached from x using the `from−left`* axis (the case of the `from−right`* is completely symmetric). Consider the sequence of nodes $x = x_0, x_1, \ldots, x_n = y$ 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 of n. When $n \leq |Q|$ we calculate $prec(x, y, Q_y)$ step by step in time $O(|Q|^3)$, observing that $prec(x_i, y, Q_y)$ is equal to $prec(x_i, x_{i+1}, prec(x_{i+1}, y, Q_y))$ for any $0 \leq i < n$ (Proposition 7.2), and that *prec* between a node and its parent can be easily calculated.

Otherwise first we calculate sets $Q_i = prec(x_i, y, Q_y)$ for $n - |Q| \leq i \leq n$ in time $O(|Q|^3)$. We say that a state q *has a* `from−left` *loop*, when there is a transition from q to q reading the letter `from−left` (similarly for the other axes). We calculate a set Q_0: a state p is in Q_0 if for some $n - |Q| \leq i \leq n$ and for some state $q \in Q_i$ with a `from−left` loop there is[11] $first^{up}(x, p, q) \geq x_i$ (which means that the state q can be reached at some node below x_i, while going up from the state p at the node x); in particular $first^{up}(x, p, q)$ should be a nonempty pointer.

We will show that $Q_0 = prec(x, y, Q_y)$.

First observe that $Q_0 \subseteq prec(x, y, Q_y)$. Indeed, we always have $(p, q) \in trans(x, first^{up}(x, p, q))$, from the definition of $first^{up}$. When $first^{up}(x, p, q) \geq x_i$ there is also $(p, q) \in trans(x, x_i)$, because the state q has a `from−left` loop and from $first^{up}(x, p, q)$ we can reach x_i using the `from−left`* axis.

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

To see that $prec(x, y, Q_y) \subseteq Q_0$, take any state q_0 from $prec(x, y, Q_y)$. This means that on some string description of the simple path from x to y, the automaton can be taken from the state q_0 to some state $q_n \in Q_y$. Let $q_1, \ldots, q_{n-1}$ be the states of the run after the nodes $x_1, \ldots, x_{n-1}$. Because there are only $|Q|$ states and because a basic automaton has only trivial cycles, there has to be $q_r = q_{r+1}$ for some $n - |Q| \leq r < n$. In particular the state q_r has a $\texttt{from}-\texttt{left}$ loop. Because the run exists, there has to be $q_r \in Q_r$ and $first^{up}(x, q_0, q_r) \geq x_r$. This means that $q_0 \in Q_0$.

In the general case (when y is any ancestor of x) we calculate $prec(x, y, Q_y)$ in a similar way. We define a *zig-zag* sequence from x to y: it is the (unique) sequence of nodes $x = x_0 > x_1 > \cdots > x_n = y$ such that x_{i+1} is a direct ancestor of x_i and that n is minimal. Observe that for any $0 \leq i \leq n-2$ 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 direct ancestors of x_{n-1} above y. Like previously, we find only a few topmost of the nodes x_i, now $2|Q| + 1$ of them, namely these for $n - 2|Q| \leq i \leq n$. To allow this, during the preprocessing we should remember for every node z its bottommost descendant reachable by the $\texttt{to}-\texttt{left}^*$ axis and its bottommost descendant reachable by the $\texttt{to}-\texttt{right}^*$ axis. Then x_i is the closest common ancestor of x and this descendant of x_{i+1} (hence it can be calculated in constant time, using Fact 4.3).

For these topmost $2|Q| + 1$ nodes we calculate the sets $Q_i = prec(x_i, y, Q_y)$; 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 10.2 (as then x_{i+1} is the topmost direct ancestor of x_i). Then we calculate the set Q_0: a state p is in Q_0 if for some $n - 2|Q| \leq i \leq n$ and for some state $q \in Q_i$ with both $\texttt{from}-\texttt{left}$ and $\texttt{from}-\texttt{right}$ loops there is $first^{up}(x, p, q) \geq x_i$. It holds $Q_0 = prec(x, y, Q_y)$ 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 states with both $\texttt{from}-\texttt{left}$ and $\texttt{from}-\texttt{right}$ loops. Since now we take $2|Q| + 1$ nodes, for every run there have to be three consecutive nodes x_i, x_{i+1}, x_{i+2} with the same state and hence this state has the two loops.

Although the situation when y is a descendant of x 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 $\texttt{from}-\texttt{left}^*$ axis. Take the sequence $y = x_0, x_1, \ldots, x_n = x$ in which x_{i+1} is the parent of x_i. First for $n - |Q| \leq i \leq n$ we calculate sets $\widetilde{Q}_i$: state p is in $\widetilde{Q}_i$ if it has a $\texttt{to}-\texttt{left}$ loop and for some $q \in Q_y$ there is $trans^{down}(y, p, q) \geq x_i$. Then we do $Q_i = \widetilde{Q}_i \cup prec(x_i, x_{i-1}, Q_{i-1})$ for $n - |Q| < i \leq n$, starting from $Q_{n-|Q|} = \widetilde{Q}_{n-|Q|}$. The argument that $Q_n = prec(x, y, Q_y)$ is very similar to the previous one. The general case is solved analogously.

10.2 Simplifying the snippets

We now come to the proof of Theorem 7.8 for a basic automaton. Recall that we have to transform a set of $O(|t|)$ arbitrary vertical snippets into an equivalent set of $O(|Q|^2|t|)$ trivial snippets in time $O(|Q|^3|t|)$. First we give two lemmas, which are used to simplify the snippets.

LEMMA 10.3. *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 $(x^\uparrow, x_\downarrow, q^\uparrow, q_\downarrow)$ in which

(a) $x^\uparrow = x_\downarrow$ (trivial snippets), or

(b) $q^\uparrow$ has a `from−left` *loop and $x^\uparrow$ is reachable from $x_\downarrow$ by the* `from−left`* *axis, or*

(c) $q^\uparrow$ has a `from−right` *loop and $x^\uparrow$ is reachable from $x_\downarrow$ by the* `from−right`* *axis.*

PROOF. Let $(y^\uparrow, y_\downarrow, Q^\uparrow, Q_\downarrow)$ be the input snippet. Assume that $y^\uparrow$ is reachable from $y_\downarrow$ using the `from−left`* axis (the other case is symmetric). Consider the sequence $y_\downarrow = y_0, y_1, \ldots, y_n = y^\uparrow$ where y_{i+1} is the parent of y_i. Let $k = \max(0, n - |Q|)$. For $k \le i \le n$ we calculate the nodes y_i and the sets $Q_i^\uparrow = prec(y_i, y^\uparrow, Q^\uparrow)$ and $Q_\downarrow^i = prec(y_i, y_\downarrow, Q_\downarrow)$, observing that (Proposition 7.2)

$$Q_i^\uparrow = prec(y_i, y_{i+1}, Q_{i+1}^\uparrow) \qquad \text{for } k \le i < n, \quad Q_n^\uparrow = Q^\uparrow, \text{ and}$$
$$Q_\downarrow^i = prec(y_k, y_{i-1}, Q_\downarrow^{i-1}) \qquad \text{for } k < i \le n, \quad Q_\downarrow^k = prec(y_k, y_\downarrow, Q_\downarrow).$$

The set $Q_\downarrow^k$ is calculated using Theorem 7.1 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 $(y_i, y_i, q_i^\uparrow, q_\downarrow^i)$ 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 $(y_i, y_\downarrow, q_i^\uparrow, q_\downarrow)$ for all states $q_i^\uparrow \in Q_i^\uparrow$, $q_\downarrow \in Q_\downarrow$ such that $q_i^\uparrow$ has a `from−left` loop, $k \le i \le n$. We get $O(|Q|^3)$ snippets of the allowed form.

All the new snippets are sound, since these snippets obtained by applying Propositions 7.5 and 7.7. Now we prove that the new set of snippets is complete. When $k = 0$ it is clear. Note that for $k > 0$ the set would be complete (by Propositions 7.5 and 7.7), if it would also contain snippets $(y_k, y_\downarrow, q_k^\uparrow, q_\downarrow)$ for all states $q_k^\uparrow \in Q_k^\uparrow$, $q_\downarrow \in Q_\downarrow$ (not only these where $q_k^\uparrow$ has a `from−left` loop). Consider one such snippet. As $q_k^\uparrow \in Q_k^\uparrow = prec(y_k, y^\uparrow, Q^\uparrow)$, there is a run of $\mathcal{A}$ from $q_k^\uparrow$ at y_k to some $q_n^\uparrow \in Q^\uparrow$ at $y^\uparrow$. Let $q_{k+1}^\uparrow, \ldots, q_{n-1}^\uparrow$ be the states of the run after the nodes $y_{k+1}, \ldots, y_{n-1}$. Because there are only $|Q|$ states, there has to be $q_r^\uparrow = q_{r+1}^\uparrow$ for some $k \le r < n$. See that $q_r^\uparrow$ has a `from−left` loop and that $q_r^\uparrow \in Q_r^\uparrow$, so there is a new snippet $(y_r, y_\downarrow, q_r^\uparrow, q_\downarrow)$ such that $q_k^\uparrow \in prec(y_k, y_r, \{q_r^\uparrow\})$. Thus, by Proposition 7.6 the snippet $(y_k, y_\downarrow, q_k^\uparrow, q_\downarrow)$ is not necessary and can be removed. □

LEMMA 10.4. *For any snippet we can find, in time $O(|Q|^3)$, an equivalent set of $O(|Q|^3)$ snippets $(x^\uparrow, x_\downarrow, q^\uparrow, q_\downarrow)$ in which*

(a), (b), (c) like above in Lemma 10.3, or

(d) $x^\uparrow$ is the topmost direct ancestor of $x_\downarrow$, or

(e) $q^\uparrow$ has both `from−left` *and* `from−right` *loops.*

PROOF. The proof is very similar to the previous one. Now we take the zig-zag sequence between $y_\downarrow$ and $y^\uparrow$ 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 7.1 in time $O(|Q|^3)$, each other using Proposition 10.2 in time $O(|Q|^2)$. We add snippets

$(y_{i+1}, y_i, q^{\uparrow}_{i+1}, q^i_{\downarrow})$ for all $q^{\uparrow}_{i+1} \in Q^{\uparrow}_{i+1}$, $q^i_{\downarrow} \in Q^i_{\downarrow}$, $k \leq i \leq n-2$ (they are of type (d)). For $i = n-1$ we can not do the same, as y_n is not the topmost direct ancestor of y_{n-1}; instead we replace the snippet $(y_n, y_{n-1}, Q^{\uparrow}_n, Q^{n-1}_{\downarrow})$ by the set from the previous lemma. We also add snippets $(y_i, y_{\downarrow}, q^{\uparrow}_i, q_{\downarrow})$ for all states $q^{\uparrow}_i \in Q^{\uparrow}_i$, $q_{\downarrow} \in Q_{\downarrow}$ such that $q^{\uparrow}_i$ has both `from−left` and `from−right` loops, $k \leq i \leq n$. The new set is equivalent for the same reasons as in the previous lemma. □

First we apply Lemma 10.4 to each original snippet, getting an equivalent set of $O(|Q|^3|t|)$ 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 $(y^{\uparrow}_1, y_{\downarrow}, q^{\uparrow}, q_{\downarrow})$ and $(y^{\uparrow}_2, y_{\downarrow}, q^{\uparrow}, q_{\downarrow})$ of type (b), (c), or (e), and $y^{\uparrow}_1$ is an ancestor of $y^{\uparrow}_2$, then the second snippet can be removed (Proposition 7.6), because $q^{\uparrow} \in prec(y^{\uparrow}_2, y^{\uparrow}_1, q^{\uparrow})$. Hence here also for each pair of states and each $y_{\downarrow}$ we need at most one snippet.

We consider every $y_{\downarrow}$ starting from the lowest nodes and ending in the root. Let y be the parent of $y_{\downarrow}$. We replace a snippet $(y^{\uparrow}, y_{\downarrow}, q^{\uparrow}, q_{\downarrow})$ by snippets $(y^{\uparrow}, y, q^{\uparrow}, q)$ for every $q \in prec(y, y_{\downarrow}, \{q_{\downarrow}\})$ (these snippets are processed again, when we are in the node y) and by trivial snippets $(y_{\downarrow}, y_{\downarrow}, q, q_{\downarrow})$ for every $q \in prec(y_{\downarrow}, y^{\uparrow}, \{q^{\uparrow}\})$; they are equivalent due to Propositions 7.5 and 7.7. Note that $prec(y_{\downarrow}, y^{\uparrow}, \{q^{\uparrow}\})$ can be computed in time $O(|Q|)$: for snippets of type (d) from Proposition 10.2; for snippets of types (b), (c), or (e) because q is in $prec(y_{\downarrow}, y^{\uparrow}, \{q^{\uparrow}\})$ if and only if $first^{up}(y_{\downarrow}, q, q^{\uparrow}) \geq y^{\uparrow}$. The other set $prec(y, y_{\downarrow}, \{q_{\downarrow}\})$ is easy as well, as y is the parent of $y_{\downarrow}$. Since for each $y_{\downarrow}$ we have $O(|Q|^2)$ snippets, the whole processing takes time $O(|Q|^3|t|)$. The key point is that we remove redundant snippets whenever a new snippet is created.

11. PATH EXPRESSIONS IN CONSTANT DELAY

In this section, we prove Theorem 1.2. This theorem is about evaluating path expressions, as opposed to node tests that were considered previously. Recall that we want to return all pairs satisfying a path expression by a constant delay algorithm: first a linear-time preprocessing can be done and then each consecutive pair should be returned in constant time.

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 5.5; this also changes the labels in the data tree t.

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 refer 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.

Let x be a node and Q_x a set of states of $\mathcal{A}$. We are interested in the set of descendants of x, from which the automaton can reach a state from Q_x at x, starting in an initial state. We define $set_I(x, Q_x)$ as the set of nodes $y \geq x$ such that $prec(y, x, Q_x)$ contains some initial state. As we are constructing a constant delay algorithm, we want to go quickly from one node in $set_I(x, Q_x)$ to the next such node. The word ,,next" could potentially refer to any order, but in our case it will be the postfix order. Hence we define $first_I(x, Q_x)$ as the first node in the postfix order which is in $set_I(x, Q_x)$. For any $y \geq x$ we also define $next_I(x, Q_x, y)$ as the next node after y in the postfix order which is in $set_I(x, Q_x)$. Such a node may not exist, in which case we say that $first_I$ or $next_I$ returns an empty pointer. We remark that although at the end $next_I$ will be used only for nodes y from $set_I(x, Q_x)$, 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 $first_I$ and $next_I$, which will be useful during the calculation of these values.

PROPOSITION 11.1. *Let $x \leq z \leq y$ be three nodes and Q_x a set of states. Assume that we know $next_I(x, Q_x, z)$ and $next_I(z, prec(z, x, Q_x), y)$. Then $next_I(x, Q_x, y)$ can be calculated in time $O(1)$.*

PROPOSITION 11.2. *Let $x \leq y$ be two nodes and Q_x a set of states. Assume that we know $next_I(x, \{q_x\}, y)$ for every state q_x. Then $next_I(x, Q_x, y)$ can be calculated in time $O(|Q|)$. The same holds for $first_I$.*

Indeed, in the first proposition, as all descendants of z are before z in the postfix order, if $next_I(z, prec(z, x, Q_x), y)$ is nonempty, it is the value of $next_I(x, Q_x, y)$; otherwise we should take $next_I(x, Q_x, z)$. In the second proposition $next_I(x, Q_x, y)$ is the first among the nodes $next_I(x, \{q_x\}, y)$ for all $q_x \in Q_x$. Now observe that the $first_I$ pointers can be easily calculated.

LEMMA 11.3. *The pointers $first_I(x, \{q_x\})$ can be calculated for each node x and each state q_x in total time $O(|Q|^2|t|)$.*

PROOF. The calculation of $first_I(x, \{q_x\})$ can be easily done in a bottom-up pass, since it is $first_I(x_1, prec(x_1, x, \{q_x\}))$, where x_1 is the left child of x; if this pointer is empty we should take $first_I(x_2, prec(x_2, x, \{q_x\}))$ for the right child x_2; if this pointer is also empty and $prec(x, x, \{q_x\})$ contains some initial state, we should take x. □

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

LEMMA 11.4. *The pointers $next_I(x, \{q_x\}, y)$ for each pair (x, y) of a parent and its child and for each state q_x 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, $next_I(x, \{q_x\}, y)$ is either empty or equal to x (if $prec(x, x, \{q_x\})$ contains some initial state). Otherwise let z be the right

child of x; we have $next_I(x, \{q_x\}, y) = first_I(z, prec(z, x, \{q_x\}))$ or, if this gives the empty pointer and $prec(x, x, \{q_x\})$ contains some initial state, we should take $next_I(x, \{q_x\}, y) = x$. □

Below, in the three subsections, we will show the following theorem, saying that it is possible to quickly compute $next_I$ for any arguments.

THEOREM 11.5. *For a tree t and an automaton $\mathcal{A}$ we can, after an appropriate preprocessing, answer queries of the form: for two nodes $x \leq y$ and a set of states Q_x compute $next_I(x, Q_x, y)$. 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 the automaton $\mathcal{A}$ is basic—preprocessing: $O(|Q|^3|t|)$, query: $O(|Q|^3)$.

Now we show how Theorem 1.2 follows from Theorem 11.5.

Similarly to set_I, let $set_F(x, Q_x)$ be the set of descendants y of x such that $prec(x, y, Q_F)$ contains some state from Q_x (where Q_F is the set of accepting states). Basing on these sets we define $first_F(x, Q_x)$ and $next_F(x, Q_x, y)$ for $y \geq x$ as the first node or as the next node after y in the postfix order which is in $set_F(x, Q_x)$. Note that in Lemma 11.3 and in Theorem 11.5 we can replace $first_I$ and $next_I$ by $first_F$ and $next_F$, and they still will be true. This is because set_F is equal to set_I with respect to an automaton $\mathcal{A}'$ in which the direction of every transition is reversed and the sets of initial and accepting states are swapped.

We run the algorithm from Lemma 11.3 and the preprocessing step of Theorem 11.5 (for both $\mathcal{A}$ and $\mathcal{A}'$). It allows us to enumerate elements of any set_I and set_F (for $\mathcal{A}$) with a constant delay.

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.

Note that types 2 and 3 are symmetric: when one swaps the role of x and y and uses the subroutine for pairs of type 2 for the reversed automaton $\mathcal{A}'$, he gets exactly all pairs of type 3. Similarly for 4 and 5. So it is enough to concentrate on types 1, 2, and 4.

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 that the $trans(x, x)$ are already calculated). 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 a 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 Theorem 11.5 to calculate $next_F(x, Q_I, y)$.

Now we come to the most complex type 4. 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. Note that we can enumerate their elements with a small delay between them (a delay of one query to Theorem 11.5). To get elements of $set_I^{left}(x, Q_x)$ we start to enumerate elements of $set_I(x, Q_x)$, using Theorem 11.5 to calculate $next_I(x, Q_x, y)$, but we stop when we are already in the right subtree of x. Similarly for $set_F^{right}(x, Q_x)$: we move between its elements normally using $next_F(x, Q_x, y)$, but we start in the first descendant of the right child x_r of x which is in $set_F(x, Q_x)$, namely from $first_F(x_r, \{q : (p, q) \in trans(x, x_r), p \in Q_x\})$.

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 11.6. *(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 $y \in set_I^{left}(z, prec(z, y, Q_F))$.

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 $set_I^{left}(z, prec(z, y, Q_F))$. Then between consecutive pairs we make a constant number of queries to Theorems 11.5 and 7.1, so the delay is small. Note that each pair (x, y) will be returned for exactly one z: for their closest common ancestor. This finishes the proof of Theorem 1.2; in the three subsections we show three proofs of Theorem 11.5, giving the tree complexities.

11.1 Linear-logarithmic algorithm

In this subsection we prove the first version of Theorem 11.5.

We need information like in Section 8.1 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 $trans(x, y)$ as previously, but also $next_I(y, \{q_y\}, x)$ for each state q_y.

Now see how to find the pointers $next_I(y, \{q_y\}, x)$. For $k = 1$ they can be calculated by Lemma 11.4. Then we inductively calculate the pointers for $k > 1$. Let z be the node halfway between x and y. The pointer $next_I(y, \{q_y\}, x)$ is easily calculated basing on the $next_I$ pointers for pairs (y, z) and (z, x), as described by Propositions 11.1 and 11.2.

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 $Q_i = prec(x_i, x, Q_x)$ observing that

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

As we know $next_I(x_i, \{q\}, x_{i+1})$ for each i and q, using Proposition 11.2 we calculate $next_I(x_i, Q_i, x_{i+1})$. Then Proposition 11.1 allows us to compose them into $next_I(x, Q_x, y)$.

11.2 Linear algorithm for the regular extension

Now we are going to prepare the data structure from Section 9.2 for queries about $next_I(x, Q_x, y)$. Recall first the important properties of this data structure. We were considering a deterministic automaton $\mathcal{D}$ having the following property: For any set Q_x of states of $\mathcal{A}$ and two nodes $x \leq y$, $\mathcal{D}$ goes from state $(\emptyset, Q_x)$ at x to state $(\emptyset, prec(y, x, Q_x))$ at y. We remark that here we use only the second coordinate of states of the deterministic automaton $\mathcal{D}$ and only tapes with empty first coordinate, as we are interested only in runs of $\mathcal{A}$ going upward in the tree. To simplify the notation, let Q_x^k be the set of states written on the second coordinate of 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 $(\emptyset, Q_x)$ a state of $\mathcal{D}$. We can find, in time constant in $|t|$, a sequence of nodes

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

in which the run of $\mathcal{D}$ starting from $(\emptyset, Q_x)$ at x changes the current tape; between x_i and y_i the run uses the same tape and x_{i+1} is a child of y_i. The numbers of tapes used in each fragment are also known. When such run uses a tape k at node z, we know that $Q_z^k = prec(z, x, Q_x)$.

The information collected in Section 9.2 will be enriched. For any node y, denote its parent as $par(y)$. For each node y (except the root) and for each set of states $Q_{par(y)}$ we remember $next_I(par(y), Q_{par(y)}, y)$. This is easily calculated using Lemma 11.4 and Proposition 11.2. Moreover, for each tape k and each node z we remember a pointer to its nearest ancestor y such that $next_I(par(y), Q^k_{par(y)}, y)$ 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 $next_I(x, Q_x, y)$, where $x \leq y$ are such that the tape containing $(\emptyset, Q_x)$ 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 $Q^k_{y'} = prec(y', x, Q_x)$. This means that $next(x, Q_x, y)$ is equal to one of $next_I(par(y'), Q^k_{par(y')}, y')$ for $x < y' \leq y$; namely to the first of them in the postfix order. But we know that $next_I(par(y'), Q^k_{par(y')}, y')$ is between y' and $par(y')$ in the postfix order. So we need to find the nearest ancestor y' of y for which $next_I(par(y'), Q^k_{par(y')}, y')$ is not empty, and it gives $next_I(x, Q_x, y)$; 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 $next_I(x, Q_x, y)$ is empty.

Consider now a general query: calculate $next_I(x, Q_x, y)$, where $x \leq y$ are arbitrary nodes. Using the data structure, we find the sequence $x_1, y_1, \ldots, x_n, y_n$ 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 $next_I(x_i, Q^{k_i}_{x_i}, y_i)$. We also know each $next_I(y_i, Q^{k_i}_{y_i}, x_{x+1})$. Note that $Q^{k_i}_{x_i} = prec(x_i, x, Q_x)$ and $Q^{k_i}_{y_i} = prec(y_i, x, Q_x)$. Thus, from Proposition 11.1 we can compose all these values into $next_I(x, Q_x, y)$.

11.3 Polynomial combined complexity for the basic fragment

Now we will prove Theorem 11.5 in the case when $\mathcal{A}$ is a basic automaton. We need to show how to quickly answer queries about $next_I(x, Q_x, y)$. As previously, we have to remember some of these values. We use here the notions of direct ancestor, topmost direct ancestor, zig-zag sequence, and states with a loop defined in Section 10.1. 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 state q, each node x, and each $1 \leq k \leq 2|Q|$ we remember the pointers $next_I(par(x, k), \{q\}, x)$ and $next_I(tda(x, k), \{q\}, x)$, if the appropriate node $par(x, k)$ or $tda(x, k)$ exists;

B. for each state q and each node y we remember the nearest ancestor x of y such that $next_I(par(x, |Q|), \{q\}, x)$ is non-empty as well as the nearest ancestor x of y such that $next_I(tda(x, 2|Q|), \{q\}, x)$ 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 11.4. Then values for bigger k are calculated by composition of smaller values, as described by Propositions 11.1 and 11.2 (namely, to calculate $next_I(par(x, k), \{q\}, x)$ we need to know $prec(par(x, k-1), par(x, k), \{q\})$, $next_I(par(x, k), \{q\}, par(x, k-1))$ and $next_I(par(x, k-1), \{p\}, x)$ 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 $next_I$ between the topmost direct ancestor of x and the parent of x with $next_I$ 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 $next_I(x, Q_x, y)$ 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 $\texttt{from}-\texttt{left}^*$ axis (the case of the $\texttt{from}-\texttt{right}^*$ axis can be done symmetrically). Consider the sequence $y = x_0, x_1, \ldots, x_n = x$, 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). First see what may happen: Let z be the value of $next_I(x, Q_x, y)$ and let x_c be the first of $x_1, \ldots, x_n$ which is an ancestor of z. Consider a run from an initial state in z to a state from Q_x in x. Whenever $n - c \geq |Q|$, this run has to be in a state with a $\texttt{from}-\texttt{left}$ loop in some node x_i for $c \leq i \leq n$. Moreover, this would happen close to x_c, for some $c \leq i \leq c + |Q|$, as well as close to x_n, for some $n - |Q| \leq i \leq n$. It is also possible that $n - c < |Q|$. We now want to cover both these cases by some of the precomputed $next_I$ values. Then we will choose the leftmost of all these values.

The second case, $n - c \leq |Q|$, is easy to cover. We find the nodes x_i and calculate $Q_i = prec(x_i, x, Q_x)$ for $n - |Q| \leq i \leq n$ in time $O(|Q|^3)$. We take $next_I(x_i, Q_i, x_{i-1})$ as a potential value of $next_I(x, Q_x, y)$.

Before we come to the case $n - c \geq |Q|$ recall one property from Section 10.1: For any state q and any x_i ($0 \leq i \leq n$) we can check whether $q \in prec(x_i, x, Q_x)$ in time $O(|Q|)$, basing on the already calculated sets Q_j for $n - |Q| \leq j \leq n$. We were doing that in the following way: For each state p with a $\texttt{from}-\texttt{left}$ loop we look at $first^{up}(x_i, q, p)$, we find the lowest $n - |Q| \leq j \leq n$ such that $first^{up}(x_i, q, p) \geq x_j$ (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 $q \in prec(x_i, x, Q_x)$).

Now we want to cover the case $n - c \geq |Q|$ for a state q with a $\texttt{from}-\texttt{left}$ loop. Using the information in B we find the lowest ancestor z of y such that $next_I(par(z, |Q|), \{q\}, z)$ is non-empty. This $next_I$ is a potential value for $next_I(x, Q_x, y)$ when $par(z, |Q|)$ is a descendant of x (so $par(z, |Q|) = x_i$ for some i) and $q \in prec(par(z, |Q|), x, Q_x)$; we check this as described in the previous paragraph. We also take $next_I(par(z, |Q| + k), \{q\}, par(z, k))$ for $1 \leq k < |Q|$ when $par(z, |Q| + k) \geq x$ and $q \in prec(par(z, |Q| + k), x, Q_x)$, as potentially they may be earlier in the postfix order.We do not need to take into account values $next_I(par(z, |Q| + k), \{q\}, par(z, k))$ for $k \geq |Q|$, because they are after $par(z, |Q|)$ in the postfix order, which is after $next_I(par(z, |Q|), \{q\}, z)$, hence for sure $next_I(par(z, |Q|), \{q\}, z)$ is a better candidate. The values taken till now are such that the run reaches q in some x_i for $i \geq |Q|$. We also need to consider smaller i: for $1 \leq i < |Q|$ we take $next_I(par(y, i), \{q\}, y)$ as a potential value of $next_I(x, Q_x, y)$ when $q \in prec(par(y, i), x, Q_x)$. Finally, as it was already mentioned, we take the leftmost from all the potential values for $next_I(x, Q_x, y)$. The whole processing is done in time $O(|Q|^3)$.

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 10.1) 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.

12. CONCLUSION

In this paper, we have provided three kinds of algorithms for evaluating queries of XPath. Each kind of algorithm can be used to evaluate boolean, unary and binary queries. Two of the three algorithms run linear time in the size of the document, but the constants depend on the query in different ways (polynomial or exponential). One of the three algorithm runs in time $n \log n$ in the size of the document, and its constants are polynomial in the query.

We are currently working on implementing the algorithms, to see how they work on real examples.

The fragment of XPath studied in this paper is a restricted fragment of XPath 1.0. It seems to us that our techniques will fail for any significant departure from this restricted fragment. We give two examples below.

One possible departure is the use of IDREF. This boils down to studying XPath queries not on trees with data, but on arbitrary graphs. In the graph setting, the distinction of tree structure and data values becomes redundant, as data values can be encoded in the edge relation. Our techniques do not work for the graph extension, because they heavily draw on tree automata.

Another possible departure is the use of XPath 2.0. The syntax of XPath 2.0 subsumes first-order logic, and therefore evaluation of XPath 2.0 would subsume evaluation of first-order logic formulas on relational structures. The latter is a widely studied and very interesting topic, but one that seems to require wholly different techniques than the ones deployed here.

REFERENCES

Bender, M. A. and Farach-Colton, M. 2000. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*. Lecture Notes in Computer Science, vol. 1776. Springer, 88–94.

Benedikt, M. and Koch, C. 2008. XPath leashed. *ACM Comput. Surv. 41,* 1.

Bojańczyk, M. 2009. Factorization forests. In *Developments in Language Theory*. 1–17.

Bojańczyk, M. and Parys, P. 2008. XPath evaluation in linear time. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 241–250.

Clark, J. and DeRose, S. 1999. XML Path language (XPath) version 1.0, W3C recommendation. Tech. rep., W3C.

Gottlob, G., Koch, C., and Pichler, R. 2002. Efficient algorithms for processing XPath queries. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 95–106.

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

Gottlob, G., Koch, C., and Pichler, R. 2005. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst. 30,* 2, 444–491.

Harel, D. and Tarjan, R. E. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13,* 2, 338–355.

Kärkkäinen, J. and Sanders, P. 2003. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2719. Springer, 943–955.

Karp, R. M., Miller, R. E., and Rosenberg, A. L. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 125–136.

Neven, F. 2002. Automata theory for XML researchers. *SIGMOD Rec. 31,* 3, 39–46.

Parys, P. 2009. XPath evaluation in linear time with polynomial combined complexity. In *PODS '09: Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 55–64.

Received Month Year; revised Month Year; accepted Month Year