XPath Evaluation in Linear Time with Polynomial Combined Complexity

Paweł Parys*
Warsaw University, Poland
parys@mimuw.edu.pl

ABSTRACT

We consider a fragment of XPath 1.0, where attribute and text values may be compared. We show that for any unary query in this fragment, the set of nodes that satisfy the query can be calculated in time linear in the document size and polynomial in the size of the query. The previous algorithm for this fragment also had linear data complexity but had exponential complexity in the query size.

Categories and Subject Descriptors. F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; H.2.3 [**Database management**]: Languages—*Query languages*

General Terms. Algorithms, Languages, Theory

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 can have running times that are exponential in the size of the query and high-degree polynomial in the size of the queried XML document [5]. There have been a number of papers devoted to improving XPath evaluation, which can be grouped into two main approaches, see e.g. [8] for a survey.

One idea, as used in e.g. [5] and improved in [4], is to use dynamic programming. This allows 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 [4] have running time $|t|^4$.

Another idea is to compile queries into finite-state tree automata, see [9] for a survey. This approach only works if the node-test does not refer to attribute or text values (a fragment called CoreXPath), and therefore an XML document can be identified with a finitely labeled tree (the label of a node is its tag name). In this setting, an XPath node test can be compiled into a finite-state automaton; and this automaton can be evaluated on the tree in linear time. In general, the automaton may be exponential in the size of the query. (It is worth noting that using dynamic programming, one can evaluate CoreXPath node tests in time linear in both query and document, see [5].)

The only linear-time algorithm was proposed in [2]. This paper and [2] can be seen as a generalization of the automata-theoretic framework to node tests that use attribute values. The general structure of these two algorithms is similar. However in [2] monoids are used and compilation of XPath queries to monoids causes exponential blowup. As a side effect, the algorithm works also for an extension of XPath, in which the Kleene star may be used in path expressions. Here, instead, we use the fact, that the Kleene star is not allowed in XPath and we observe that automata recognizing XPath path expressions have a special form. This allows to do a special things in polynomial time directly with the automata, which in general are possible only for monoids, thus avoiding the exponential blowup.

The following aspects of the algorithm from [2] are improved here:

- The previous algorithm had exponential complexity in the size of the query. Here we have polynomial complexity in the size of the query (with the exception of queries using `sum` or `count`, which are not considered in [2] at all).
- The previous algorithm had linear complexity in the number of bits of the XML document (it was important that the alphabet had constant size). Here the complexity is linear in the number of nodes plus the total length of text, where the text length is measured in bytes. These bytes may be of size logarithmic in the

*Author supported by Polish government grant no. N206 008 32/0810.

input size, if we assume that some basic operations on them may be done in constant time. This is a more classical complexity measurement.

- The previous algorithm was able to compare only attribute values (or text values of leaf text nodes). However the XPath specification [3] also defines a string-value for inner nodes, this is the concatenation of the string-values of all its text node descendants in document order. It should be possible to compare these string-values. This definition means that the total length of string-values of all nodes may be quadratic in the document size, so calculating all of them explicitly is impossible in a linear time algorithm. We show how to handle them implicitly in linear time.
- The previous algorithm worked only for a limited fragment of XPath. Here we extend it to almost full Agg-XPath fragment (as defined in [8]), with the exception of `id()` function. So in addition to fragment in [2] we handle inequalities $<, >, \leq, \geq$ (which is an easy extension) and calculating aggregates `count` and `sum`, together with arithmetic operations on them. However when a query uses `count` or `sum`, the algorithm is exponential in the size of the query.

The only thing, which is better in [2], is that it allows to use the Kleene star in path expressions. This is an extension of XPath standard and we do not handle it.

The following are our main results:

Theorem 1.1. *Let φ be a node test of XPath (as defined in section 2.2) in which* `sum` *and* `count` *are not used, and t an XML document. The set of nodes of t that satisfy φ can be computed in time $O(|t| \cdot |\varphi|^3)$. When* `sum` *or* `count` *are used, the same may be done in time $O(|t| \cdot c^{|\varphi|})$ for some c.*

The length $|t|$ is the length of the text file representing the XML document. Note that the alphabet used in the text need not to be of fixed size (for example just 2 or 256 letters). We rather treat letters as numbers. We only require, that there are at most polynomially many letters (that the letters are of the logarithmic size) and that all standard operations (like comparing, arithmetic operations, etc.) on the letters are done in constant time. This is a standard complexity measurement. Even with such assumptions the algorithm works in time linear in $|t|$.

One may ask why we give a linear time algorithm only for node tests and not for path expressions. However for path expressions a linear time algorithm is impossible, since they may select quadratically many pairs of nodes. Algorithm given here allows to evaluate path expressions in quadratic time.

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. The algorithm is then detailed in Sections 4 to 11. In general terms, Section 4 and 5 show some necessary precomputations, Section 6 is devoted to node tests using inequalities, Section 7 to node tests using `sum` and Sections 8 to 11 to node tests using equalities. Finally, in Section 12, we discuss possible extensions of this work.

2. Data model and XPath

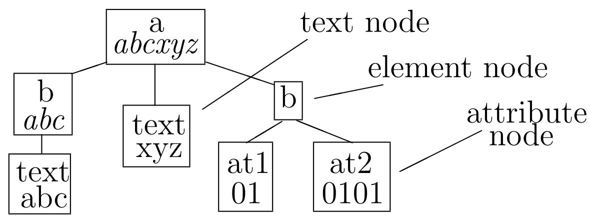
2.1 Data model

In this section we define the data model. We represent an XML document as a tree, called a *data tree*. The tree is unranked, i.e. a node may have any number of children, and the children are ordered. There are three types of nodes: element nodes, attribute nodes and text nodes. Attribute and text nodes are always leaves. Every element and attribute node is assigned a *name* which is a tag name or an attribute name, and which is taken from a finite alphabet. Text nodes do not have names, we assume that their name is `text`. We call the whole alphabet Σ—every node is labeled with a name from the set Σ. Moreover every node has a *string value*. A string value of an attribute node is the value of the corresponding attribute, which is a string. A string value of a text node is just a text. But, what is important, the string value of an element node is the *concatenation* of the string values of all text node descendants of the element node 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:

$\langle$a$\rangle$
  $\langle$b$\rangle$abc$\langle$/b$\rangle$xyz
  $\langle$b at1 = "01" at2 = "0101"$\rangle\langle$/b$\rangle$
$\langle$/a$\rangle$

The data tree representing this document will use names $\Sigma = \{$a, b, at1, at2, text$\}$. The data tree will look like this (string values in italic are not remembered):



Trees will be denoted by a letter t and binary trees by a letter $\widehat{t}$. Nodes will be denoted by u, v, w. String values will be denoted by d. 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 for 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 fragment of XPath that is used in this paper. This is almost a fragment called AggXPath. Basically, these are almost all queries, including value comparing, computing aggregates and manipulating integers. Comparing to the AggXPath fragment, only using of `id()` function is forbidden. From constructs important for evaluation complexity, in full XPath 1.0 there are also `position()` and `last()` functions, which are forbidden here too. In fact the specification [3] of XPath 1.0 contains a lot of other constructs. However these are technical details, such as type converting, etc. and they may be easily added.

In XPath, the primitives employed for navigation along the tree structure are called *axes*. We consider the following *one-step* axes (their names here are slightly different than usual): `child`, `next` and their inverses `parent`, `prev`. They correspond to going to a child and to the next sibling. Moreover we consider their transitive-reflexive closures, called *multistep* axes: $\texttt{child}^*, \texttt{next}^*, \texttt{parent}^*, \texttt{prev}^*$.

For simplicity we treat strings and numbers as the same. The idea is that all numbers are represented as strings. Since according to the XPath specification all numbers are constant-size floating point reals, there is no problem with complexity: we may assume, that all operations on numbers are done in constant time. We do not define what happens, when someone tries to do an arithmetic operation on a string, which does not represent any number.

There are three types of expressions: path expressions, node tests and string-typed expressions. We may look on them as on functions, for every node returning respectively: node sets, booleans and strings. Another way for looking on a *path expression* is that it is a binary query. In each tree, a path expression will select a set of pairs (u, v) of nodes. Intuitively a path expression will describe the path from u to v, although the path might not be the shortest one. A typical path expression is $\texttt{parent} \cdot \texttt{child}$, it selects a pair (u, v) if v is a sibling of u, possibly $u = v$. A *node test* is a unary query: it selects a set of nodes. A typical node test is a, it selects nodes that are labeled by the tag name a. A *string-typed expression* produces a string value i for every node v. A typical string-typed expression is $\texttt{count}(\texttt{child})$, which for every node v calculates a string representation of the number of children of v. In general in XPath, the three types of expression are mutually recursive, as defined below:

- Every name $a \in \Sigma$ is a node test, which holds in nodes with a name a.
- Node tests admit negation, conjunction and disjunction.
- If α, β are path expressions, ϑ, ϑ' are string-typed expressions and $\texttt{RelOp} \in \{=, \leq, <, >, \geq, \neq\}$,

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

  are node tests. The first of them selects a node u if there exist nodes v, w such that (u, v) is selected by α and (u, w) is selected by β and that the string values of v and w satisfy the relation `RelOp`. The second of them selects a node u if there exists a node v such that (u, v) is selected by α and that the string value of v and the value of ϑ calculated in u satisfy the relation `RelOp`. The third of them selects a node u if values of ϑ and ϑ' calculated in u satisfy the relation `RelOp`. The inequalities $\leq, <, >, \geq$ correspond to the linear order of numbers. Only $=$ and $\neq$ may be done on arbitrary strings.
- There are two types of *atomic* path expressions. Every axis, including the multistep axes, is an atomic path expression. Furthermore, a node test φ can be interpreted as an atomic path expression $[\varphi]$, which holds in pairs (u, u) such that φ holds in u.
- In general, a path expression is a concatenation (composition) or union of simpler path expressions.
- A string constant 'c' is a string-typed expression. It is equal to 'c' in every node u.
- If α is a path expression,

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

  are string-typed expression. For a node u it calculates the number of nodes v such that (u, v) is selected by α or, appropriately, the sum of all string values of nodes v such that (u, v) is selected by α.
- String-typed expressions (representing numbers) may be added, multiplied, etc.

Note that the operators $=$ and $\neq$ in node tests $\alpha \,\texttt{RelOp}\, \beta$ and $\alpha \,\texttt{RelOp}\, \vartheta$ are not mutually exclusive. A node may satisfy none or one or both of $\alpha = \beta$ and $\alpha \neq \beta$ (similarly for $<, \geq$, etc.). However always exactly one of the node tests $\vartheta = \vartheta'$ and $\vartheta \neq \vartheta'$ is satisfied in a node, as string-typed expressions produce exactly one value in every node.

Note that, following XPath specification, we do not allow the Kleene star in path expressions, which is different from [2]. Our algorithm does not work for path expressions with the Kleene star.

When referring to XPath, we mean the fragment above.

3. Proof strategy

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

To allow storing of intermediate results, we slightly extend the definition of node names. Now a tree t comes with some constant k and in every node of t there is an array of k names from Σ. A node test that checks for a name is now of the form $\texttt{name}[i] = a$ where $1 \leq i \leq k$ is an integer constant and $a \in \Sigma$; it holds in nodes whose i-th name is a. We do not change the definition of the 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 (so 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 tree t satisfying φ. Simultaneously we show an algorithm, which for a string-typed expression ϑ calculates its value in every node of a tree t (the result of ϑ may be only a number, in which case it has constant size by assumption, or a constant from the query, in which case its size is bounded by $|\vartheta|$). We want the algorithms to run in time linear in $|t|$. Although the constant in the linear time will depend on the size of node test or string-typed expression—it should be cubic when `sum` and `count` are not used anywhere inside the query, otherwise it may be exponential. The algorithm works by induction on the size of φ or ϑ.

There are a few easy cases: when φ just tests a name, when it is a negation, conjunction or disjunction of smaller node tests, when φ is of the form ϑ `RelOp` ϑ', when ϑ is just a constant or when ϑ is an arithmetical operation of smaller string-typed expressions. For example to evaluate a node test $\vartheta = \vartheta'$, first we evaluate both ϑ and ϑ' from the induction assumption, which gives in every node of t some number or some string constant (when one of ϑ, ϑ' is a constant), and then in every node we check, whether the two results are equal or not.

Consider now the first nontrivial induction step: a string-typed expression $\mathtt{sum}(\alpha)$. Let $\varphi_1, \dots, \varphi_n$ be the node tests that appear in the path expression α. Using the induction assumption, we run a linear time algorithm for each of these node tests, and label each node in the tree with the set of node tests from $\varphi_1, \dots, \varphi_n$ that it satisfies. Formally we enrich Σ by constants `true` and `false` and we construct a new data tree t'. It is almost the tree t, but the name array of every node consists of $n + k$ elements (instead of k). The first k elements of the array contain the original names of this node from the tree t. The $i + k$-th element is `true` if the node satisfies φ_i and `false` otherwise. Due to specific definition of size, both trees have the same size. Then we replace every φ_i in α by a name test checking if $i + k$-th element of the name array is equal to `true` and we run $\mathtt{sum}(\alpha)$ on the tree t'. In other words, we may assume without loss of generality that the only node tests appearing in atomic path expressions in α are name tests. This case is solved in Section 7.

In the same way we may reduce the node test of the form α `RelOp` β to the case when the only node tests appearing in atomic path expressions in α and β are name tests. Such node tests are solved in the farther sections.

The node test $\mathtt{count}(\alpha)$ may be easily simulated by a node test $\mathtt{sum}(\alpha')$. We construct a tree t', which is a modified version of t: under every node of t we add a new, rightmost attribute child with a string value 1. The name array would be extended with an additional field, which is `true` in the new children and `false` in the nodes from t. The node test $\mathtt{sum}(\alpha')$ in t' should return the same as $\mathtt{count}(\alpha)$ in t by summing the ones in the added children of the nodes selected by α. To get α' from α, we should append at its end a path expression going to a child and checking in its name that it is an added child. We also have to avoid using the new nodes elsewhere in α: after every axis we add a name test checking that we are not in a new child. Note that the tree size is at most multiplicated by some constant.

Similarly α `RelOp` ϑ may be simulated by α' `RelOp` β: We add a new, rightmost attribute child under every node, which would contain in a string value the result of the string-typed expression ϑ in that node; β just goes to the new child and α' does the same as α omitting the new children. Whenever ϑ is not a constant, then its result is a number, which we assume has constant size. When ϑ is a constant, then the tree t' could be too big, so we proceed in a slightly different way: we add a new child only under the root; β goes to the root and then to its new child. Then under the natural assumption $|t| \geq |\vartheta|$, we have $|t'| \leq |t| \cdot 2$.

Concluding, only the constructions α `RelOp` β and $\mathtt{sum}(\alpha)$ are left for the next sections, and only in the case, when the only node tests appearing in atomic path expressions in α and β are name tests.

4. Classifying nodes by string values

In this section we show the following result:

Proposition 4.1. *Nodes of a data tree t may be divided into classes with equal string values in time $O(|t|)$.*

We assign some natural number to each of these classes and in every node we remember what is the number of its string value. Thanks to that, we may later in constant time compare string values for equality. String values representing numbers will be also compared for inequalities ($<$, $\leq$, $>$, $\geq$), but such string values are of constant size.

As a side remark note, that the assignment of numbers to classes of equal string values may be done in such way, that the order on these numbers would agree with the lexicographical order of the string values (this follows from the proof below). Thanks to that, the operators $<$, $\leq$, $>$, $\geq$ may be also used to lexicographically compare string values. However this is not included in XPath standard.

Proof (of Proposition 4.1) The *suffix array* is a lexicographically sorted array of the suffixes of a string (of course in this array we do not remember the whole suffixes, only their numbers). Kärkkäinen and Sanders [7] show how to construct a 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. Note that they do not assume that the alphabet has constant size, their complexity measurement agrees with the one declared by us in the introduction.

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. Note that this string contains the string values of all element nodes as infixes, however they overlap. Moreover for every node we may calculate which infix it is (the start position and length). Now we run the suffix array algorithm on that string. Additionally we sort all nodes by 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 starting at the position where this string value starts. We process string values of given length in the (already calculated) lexicographical order of these suffixes. We know (in constant time) what is the length of the common fragment of a suffix and a 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, since the farther suffixes differ at at least the same or even more first positions. □

5. From path expressions to automata

In this section we show how automata may be used to calculate path expressions.

From an arbitrary data tree t we create its binary version $\widehat{t}$ (using the first child / next sibling encoding). It has the same set of nodes, with the same string values, but we change the way in which the nodes are connected. The leftmost child of a node u from t becomes its left child in $\widehat{t}$. The next sibling of u from t becomes its right child in $\widehat{t}$. Node names will also be changed in some way (more about this later). For nodes u, v of $\widehat{t}$, we say that u is a t-*child* (or t-parent, etc.) of v, when it is his child (or parent) in the original tree t. Writing just child or parent we mean the relation in the binary tree $\widehat{t}$.

A *path* in a binary tree is a sequence of nodes $u_1, \ldots, u_n$ where each two consecutive nodes are connected (one is a child of the other). A *string description* of a path $u_1, \ldots, u_n$ is a word $A_1 m_1 A_2 m_2 \cdots A_{n-1} m_{n-1} A_n$ over the alphabet $\{1, \ldots, k\} \times \Sigma \cup \{\overline{\texttt{child}}, \texttt{next}, \overline{\texttt{parent}}, \texttt{prev}\}$, where k is the number of elements in the name array of every node of $\widehat{t}$. The letter m_i is a name of one of the four one-step axes depending on the relationship between u_i and u_{i+1} in t. So it is $\overline{\texttt{child}}$, $\texttt{next}$, $\overline{\texttt{parent}}$ or $\texttt{prev}$ when in $\widehat{t}$ the node u_{i+1} is the left child of u_i, the right child of u_i, u_i is the left child of u_{i+1} or the right child of u_{i+1}, respectively. We use the new axis $\overline{\texttt{child}}$ instead of $\texttt{child}$ because a node is connected by the $\overline{\texttt{child}}$ axis only with its leftmost child from t, not with all children (similarly for $\overline{\texttt{parent}}$). The word A_i consist of some pairs (j, a) such that j-th name of u_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. 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 string description in which every word A_i contains at most one letter.

Let $\mathcal{A}$ be a nondeterministic automaton with states Q reading string descriptions. Let u, v be any two nodes in a binary tree $\widehat{t}$. We write $trans^{all}_{\mathcal{A},\widehat{t}}(u, v)$ for the set of state pairs (p, q) such that some string description of some path from u to v 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 u to v, the string description and the run of the non-deterministic automaton. Similarly we write $trans_{\mathcal{A},\widehat{t}}(u, v)$ for the set of state pairs (p, q) such that some simple string description of the simple path from u to v can take the automaton $\mathcal{A}$ from state p to state q. When both $\widehat{t}$ and $\mathcal{A}$ are clear from the context, we simply write $trans(u, v)$.

Definition 1. A nondeterministic word automaton $\mathcal{A}$ with states $Q = \{q_1, \ldots, q_n\}$ reading string descriptions is called an *XPath automaton* for a binary tree $\widehat{t}$ when:

1. transitions from q_i to q_j exist only for $i \leq j$;
2. if for some i there is a transition from q_i to q_i reading $\overline{\texttt{child}}$ (respectively, $\overline{\texttt{parent}}$) then there is also such transition reading $\texttt{next}$ (respectively, $\texttt{prev}$);
3. the automaton has $O(|Q|^2)$ transitions;
4. $trans^{all}_{\mathcal{A},\widehat{t}}(u, v) = trans_{\mathcal{A},\widehat{t}}(u, v)$ holds for any two nodes u, v.

The third condition just says that the number of transitions does not depend on the number of names in the name array of every node. Note that the last condition depends on the tree $\widehat{t}$; the definition talks about a pair: an automaton and a tree. The main result of this section is the following theorem:

Theorem 5.1. *Let t be a data tree and α a path expression such that the only node tests appearing in atomic path expressions in α are name tests. We may calculate a binary version $\widehat{t}$ of t and an XPath automaton $\mathcal{A}$ for $\widehat{t}$ with $O(|\alpha|)$ states such that a pair of nodes u, v is selected by α in t iff $(q_I, q_F) \in trans_{\mathcal{A},\widehat{t}}(u, v)$ for some initial state q_I and accepting state q_F. Moreover for any node u we may calculate (and remember in $\widehat{t}$) $trans_{\mathcal{A},\widehat{t}}(u, u)$ and $trans_{\mathcal{A},\widehat{t}}(u, v)$ for v being the parent of u or the left or right child of u. All this may be done in time $\Theta(|t||\alpha|^3)$.*

It may be proved using standard techniques. Condition 1 of Definition 1 comes from the fact, that Kleene star is not allowed in path expressions. We get condition 2, because there is no multistep axis going only to the leftmost child several times, we have to go to an arbitrary descendant. To get condition 4, which says that we may consider only simple paths instead of all, we use the following lemma:

Lemma 5.2. *For a nondeterministic automaton $\mathcal{A}$ and a binary tree $\widehat{t}$ we may in time $O(|\widehat{t}||Q|^3)$ calculate for every node u of $\widehat{t}$ the set*

$$loop(u) = trans^{all}_{\mathcal{A}',\widehat{t}'}(u, u)$$

Once we have the sets *loop*, we may remember them in the name array of every node and modify the automaton, in such a way that it will be reading these values instead of making loops.

6. Inequalities

In this section we deal with node tests of the form:

$$\alpha \; \texttt{RelOp} \; \beta$$

where RelOp is one of the inequalities: $\neq, <, >, \leq, \geq$. If (u, v) is a node pair selected by the path expression α, a string value d of v is called a *representative for α in u*. Likewise for β. For the relations $<, >, \leq, \geq$ only string values representing numbers may be representatives and there is a natural order on them. For $\neq$ we may use any linear order on all string values and we use the order on the numbers given to each string value in Section 4.

The basic idea is as follows. For each node u of a binary data tree $\widehat{t}$, we calculate the minimal and the maximal representative for α in u, or if there is no representative at all. Likewise for β. This information is sufficient to test if α RelOp β holds. For example a node u 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.

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 XPath automaton with states Q. A representative for a state $q \in Q$ in a node u is a string value d of some node v with $(q, q_F) \in trans(u, v)$, 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 path expression α or β can be simulated by an XPath automaton reading simple string descriptions of simple paths (Theorem 5.1). It is worth noting that in this section, as well as in Section 7, we do not use conditions 1 and 2 from the definition of XPath automaton (Definition 1). So these algorithms would also work for path expressions allowing the Kleene star. The special form of an XPath automaton is necessary for evaluating node tests $\alpha = \beta$ and will be used in Sections 9 and 11.

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 u, we should consider: the string value of u if q is accepting, and the minimal such representative in the left child v of u for any state p such that $(q, p) \in trans(u, v)$, similarly for the right child. Such a step may be done even in time $O(|Q|^2)$, similarly 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 $O(|t||Q|^2)$.

7. Aggregates

In this section we deal with string-typed expressions of the form $\texttt{sum}(\alpha)$. Recall that they take into account only string values representing numbers, and calculate sums of appropriate string values understood as numbers. In particular these sums are commutative. As in the previous section we generalize the problem to automata and we use it for the XPath automaton $\mathcal{A}$ with states Q corresponding to α reading string descriptions of simple paths (from Theorem 5.1).

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

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

8. Skeleton representation

Now we turn to node tests of the form $\alpha = \beta$. These are the most interesting node tests. Sections 8 to 11 are devoted to this case. At the beginning we show how nodes with the same string values are organized. Then in Section 9 we speed up calculations of automata runs. In Section 10 the whole problem is almost solved, but a most difficult theorem is postponed to Section 11.

In this section we show how a binary data tree is stored in memory by the algorithm while performing node tests of the form $\alpha = \beta$. An initial situation is that we have a record for each node, called the *node record*. This record contains the node name, the number of its string value, as well as pointers to the node records of the: parent, left and right child. Some of these may be empty, if the appropriate nodes do not exist. Additionally the node record contains the level of the node (i.e. distance from the root).

Let u and v be two nodes in a binary data tree $\widehat{t}$. The *least common ancestor* (LCA) of u and v is the (unique) node w that is an ancestor of both u and v, and has a minimal possible distance from u and v.

Let the *class of d* be the set of all least common ancestors of any two nodes u and v having string value d. In particular every node with a string value d is in the class of d (since a node u is the least common ancestor of u, u).

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

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

Note that the sum of sizes of all skeletons in $\widehat{t}$ is linear in $\widehat{t}$, since each node may be a leaf only in one skeleton. Moreover the skeleton representation can also be calculated in linear time. The crucial operation is finding the LCA of any two given nodes. Harel and Tarjan [6] show an algorithm, which first does a preprocessing on a tree $\widehat{t}$ in time $O(|\widehat{t}|)$ and then in time $O(1)$ can answer queries: „where is the least common ancestor of nodes u and v?". A much simpler algorithm doing the same was given later by Bender and Farach-Colton [1]. These algorithms allow us to prove the proposition:

Proposition 8.1. *The skeleton representation of a binary data tree can be calculated in time* $O(|\widehat{t}|)$.

9. Precomputing automaton runs

In this section we show that, after appropriate preprocessing, we may run an XPath automaton in time constant in the length of its input.

Fix an XPath automaton $\mathcal{A}$ with states Q and a binary tree $\widehat{t}$. For every node u and its t-parent v (parent in the original tree t) we remember in u sets $trans(u, v)$ and $trans(v, u)$. Additionally in every node u we remember a pointer to its rightmost t-child. It is easy to calculate these values while moving from left to right through all t-children of a fixed node.

For every node u of $\widehat{t}$ and every two states p, q we define $first^{up}(u, p, q)$ as a pointer to the lowest (farthest from the root) ancestor v of u such that $(p, q) \in trans(u, v)$. It is possible that such an ancestor does not exist, in such case we remember an empty pointer instead. These pointers are stored in the node u. Similarly let $first^{down}(u, p, q)$ be a pointer to the lowest ancestor v of u such that $(p, q) \in trans(v, u)$. Notice the broken symmetry here: although $first^{up}$ describes runs of the automata going up in the tree and $first^{down}$ these going down, but both of them contain pointers to nodes somewhere above in the tree. Intuitively, pointers to nodes below are impossible, because there are multiple branches of the tree. The following lemma shows, that these functions may be efficiently calculated.

Lemma 9.1. *We may calculate the functions* $first^{down}$ *and* $first^{up}$ *in time* $O(|t||Q|^3)$.

Proof Let v be the parent of u. Then $first^{up}(u, p, q)$ is equal to u, if $(p, q) \in trans(u, u)$, otherwise it is the lowest from nodes $first^{up}(v, p', q)$ for all states p' such that $(p, p') \in trans(u, v)$. We may 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 $O(|t||Q|^3)$. Similarly we calculate $first^{down}$. □

The second lemma says, that these functions may be used to speed up the automaton. Here is the first time when we use the fact, that XPath automaton does not have nontrivial cycles (that the Kleene star is not allowed in path expressions). The fact is used also in Section 11.

Lemma 9.2. *For any two nodes u, v such that one is an ancestor of the other, and for any set of states $Q_v \subseteq Q$ we may compute in time $O(|Q|^3)$ the set*

$$prec(u, v, Q_v) = \{p : \exists_{q \in Q_v} (p, q) \in trans(u, v)\}$$

Before we come to the proof, we give some intuitions staying behind it. 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. Instead of considering an arbitrary run, we may (for a run going upwards) reach last such state q as fast as possible (which is described by the $first^{up}$ function), then go up staying in this state and finally do only a few more arbitrary steps. Similarly for a run going downwards, we have to reach such state quite fast, then we go down staying in this state as long as possible, and finally do some transitions described by $first^{down}$.

Proof (of Lemma 9.2) First assume that v is an ancestor of u. Let w be this right t-sibling of v, which is a t-ancestor of u. To find w we go from v to his t-parent, then to his rightmost t-child w' and then w is the least common ancestor of u and w' in $\widehat{t}$, which we may find in constant time.[1] Then

$$prec(u, v, Q_v) = prec(u, w, prec(w, v, Q_v))$$

so its enough to solve the cases, when v is a t-ancestor of u and when v is a t-sibling of u.

Consider the case when v is a t-ancestor of u. Consider the nodes $u = u_0, u_1, \ldots, u_n = v$, where u_{i+1} is the t-parent of u_i (we are not allowed to find all of them and for example remember on a list, as the complexity should be independent on n). Recall, that we already have calculated $trans(u_i, u_{i+1})$, it is stored in the node u_i. So we may calculate $prec(u_i, u_{i+1}, \widetilde{Q})$ for any set $\widetilde{Q}$, even in time $O(|Q|^2)$. When $n \leq |Q|$ we may calculate $prec(u, v, Q_v)$ step by step

[1]It is possible to avoid using the LCA algorithm in this lemma and in Section 11. Instead while calling the calculation of *prec* some additional data should be remembered for the path from the root to v (the descendant). To maintain this data, the calls to the calculation of *prec* should appear in some specific order. This technique was used in [2].

in time $O(|Q|^3)$, observing that $prec(u_i, v, Q_v)$ is equal to $prec(u_i, u_{i+1}, prec(u_{i+1}, v, Q_v))$ for any $0 \le i < n$.

Otherwise first we calculate sets $Q_i = prec(u_i, v, Q_v)$ for $n - |Q| \le i \le n$ in time $O(|Q|^3)$ (before that we have to find nodes u_i, but u_i is the least common ancestor of u and the rightmost t-child of u_{i+1}). We say that a state q *has a* $\overline{\texttt{parent}}$ *loop*, when there is a transition from q to q reading the letter $\overline{\texttt{parent}}$ (similarly for the other axes). Recall from Definition 1 that if a state has a $\overline{\texttt{parent}}$ loop, then it has a `prev` loop as well. We write $lev(u_i)$ to denote the level (distance from the root) of the node u_i; the levels are remembered in the node record. We calculate a set Q_0: a state p is in Q_0 if for some $n-|Q| \le i \le n$ and for some state $q \in Q_i$ with a $\overline{\texttt{parent}}$ loop there is $lev(first^{up}(u, p, q)) \ge lev(u_i)$ (which means that the state q may be reached in some node below u_i, while going up from the state p in the node u); in particular $first^{up}(u, p, q)$ should be nonempty pointer. Finally we hope that $Q_0 = prec(u, v, Q_v)$.

First observe that $Q_0 \subseteq prec(u, v, Q_v)$: We always have $(p, q) \in trans(u, first^{up}(u, p, q))$, from the definition of $first^{up}$. When $lev(first^{up}(u, p, q)) \ge lev(u_i)$ there is also $(p, q) \in trans(u, u_i)$, because the state q has $\overline{\texttt{parent}}$ and `prev` loops.

To see that $prec(u, v, Q_v) \subseteq Q_0$ take any state q_0 from $prec(u, v, Q_v)$. This means that on some string description of the simple path from u to v the automaton may be taken from state q_0 to some state $q_n \in Q_v$. Let $q_1, \dots, q_{n-1}$ be the states of the run after the nodes $u_1, \dots, u_{n-1}$. Because there are only $|Q|$ states and because an XPath automaton has only trivial cycles, there has to be $q_r = q_{r+1}$ for some $n - |Q| \le r < n$. In particular state q_r has a $\overline{\texttt{parent}}$ loop. Because the run exists, there has to be $q_r \in Q_r$ and $lev(first^{up}(u, q_0, q_r)) \ge lev(u_r)$. This means that $q_0 \in Q_0$.

The case when v is a left t-sibling of u is very similar, even simpler. We consider the sequence $u = u_0, u_1, \dots, u_n = v$ in which u_{i+1} is the previous t-sibling of u_i (so it is just its parent in $\widehat{t}$) and we consider states with a `prev` loop instead of these with $\overline{\texttt{parent}}$ loop.

Although the situation when v is a descendant of u is not completely symmetric, it is similar. Once again we divide the problem into two cases. Consider the case, when v is a t-descendant of u and take the sequence $u = u_0, u_1, \dots, u_n = v$ in which u_{i+1} is a t-child of u_i. First for $0 \le i \le |Q|$ we calculate sets $\widetilde{Q}_i$: state p is in $\widetilde{Q}_i$ if it has a $\overline{\texttt{child}}$ loop and for some $q \in Q_v$ there is $lev(trans^{down}(v, p, q)) \ge lev(u_i)$. Then we do $Q_i = \widetilde{Q}_i \cup prec(u_i, u_{i+1}, Q_{i+1})$ for $0 \le i < |Q|$ and $Q_{|Q|} = \widetilde{Q}_{|Q|}$. Argumentation that $Q_0 \subseteq prec(u, v, Q_v)$ is very similar to the previous one. □

10. The core problem

In this section, we identify the main difficulty in calculating node tests $\alpha = \beta$. The strategy will be as follows: first we define three kinds of sets. Then we show, that knowing them is enough to solve the node test $\alpha = \beta$. Finally we show how to calculate the easier two of these types of sets. Calculating of the third type is postponed to Section 11.

From Theorem 5.1 we know, that α and β may be recognized by XPath automata. By inspecting the proof of the theorem it is easy to see, that for both α and β we may use a common XPath automaton, denoted $\mathcal{A}$, with states Q, working in a binary tree $\widehat{t}$ (being just the union of the automata for α and β). The set of accepting states Q_F may 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 u, v is selected by α iff $(q_I^\alpha, q_F) \in trans(u, v)$ for some $q_I^\alpha \in Q_I^\alpha$ and $q_F \in Q_F$; similarly for β.

For any string value d and a node u in the class of d we calculate a set $class(u, d)$ of states p such that $(p, q_F) \in trans(u, v)$ for some $q_F \in Q_F$ and for some node v with string value d. Note that the requirement on u is weaker than that on v: u only need be in the class of d.

For any node u we define the set $core(u)$ of state pairs $(p_\uparrow, p_\downarrow)$ such that for some two nodes $v_\uparrow$,$v_\downarrow$ there is:

- $v_\uparrow$ is an ancestor of u and $v_\downarrow$ is a descendant of u (both ancestor and descendant need not to be proper);
- for some d both nodes $v_\uparrow$ and $v_\downarrow$ are in the class of d and no other node between them is the class of this d;
- for some $q_\uparrow \in class(v_\uparrow, d)$ and $q_\downarrow \in class(v_\downarrow, d)$ there is:

$$(p_\uparrow, q_\uparrow) \in trans(u, v_\uparrow) \qquad (p_\downarrow, q_\downarrow) \in trans(u, v_\downarrow)$$

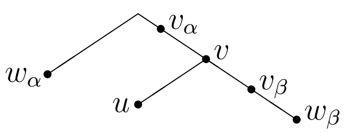
For any node u we define the set $double(u)$ of state pairs $(p_\uparrow, p_\downarrow)$ such that for some node v and some states $(q_\uparrow, q_\downarrow) \in core(v)$ there is:

$$(p_\uparrow, q_\uparrow) \in trans(u, v) \qquad (p_\downarrow, q_\downarrow) \in trans(u, v)$$

Lemma 10.1. *A node u satisfies the node test $\alpha = \beta$ if and only if $(q_I^\alpha, a_I^\beta) \in double(u)$ or $(q_I^\beta, a_I^\alpha) \in double(u)$ for some initial states $q_I^\alpha \in Q_I^\alpha$ and $q_I^\beta \in Q_I^\beta$.*

Proof $\Leftarrow$ Directly from the above definitions we see that for some two nodes w_α, w_β with the same string value d there is $(q_I^\alpha, q_F^\alpha) \in trans(u, w_\alpha)$ and $(q_I^\beta, q_F^\beta) \in trans(u, w_\beta)$ for some accepting states $q_F^\alpha, q_F^\beta \in Q_F$. This exactly means that u is selected by $\alpha = \beta$.

$\Rightarrow$ Take the two nodes w_α, w_β with the same data value d, such that α selects (u, w_α) and β selects (u, w_β). Consider the simple paths from u to w_α and to w_β. First the two paths go together to some node v, starting from which they are disjoint. Let v_α (and v_β) be the first node on the path from v to w_α (to w_β) in the class of d (where d is the common string value in w_α and w_β). One of the two paths from v, let say this to w_β, has to go only down. So v_β is a descendant of v. In such case v_α has to be an ancestor of v, because the least common ancestor of w_α and w_β is in the class of d (and it is an ancestor of v). Some of the mentioned nodes may coincide.



Let q_I^α, p_α and q_α (similarly q_I^β, p_β and q_β) be the states in u, in v and in v_α (in v_β) of the accepting run of $\mathcal{A}$ on the path from u to w_α (to w_β). Then we see that:

$$\begin{array}{ll} q_\alpha \in class(v_\alpha, d) & (p_\alpha, p_\beta) \in core(v) \\ q_\beta \in class(v_\beta, d) & (q_I^\alpha, q_I^\beta) \in double(u) \end{array} \qquad \square$$

Now we come to calculating the three types of sets. The following lemma follows from Lemma 9.2:

Lemma 10.2. *The set $class(u, d)$ can be calculated for every string value d and every node u in the class of d in time $O(|t||Q|^3)$.*

To remember the values of $class$ we use the skeleton representation. For each string value d, the set $class(u, d)$ will be stored in the d-skeleton, inside the node record that corresponds to the node u.

The main technical result is that the sets $core(u)$ may be efficiently calculated. The following theorem will be shown in Section 11:

Theorem 10.3. *The set $core(u)$ for every node u may be calculated in time $O(|t||Q|^3)$.*

Finally we calculate the last type of sets:

Lemma 10.4. *The set $double(u)$ for every node u may be calculated in time $O(|t||Q|^3)$.*

Proof Here we also do a bottom-up pass followed by a top-down pass. In the bottom-up pass we calculate the part $double^{down}(u)$ of $double(u)$ such that the node v from the definition is a descendant of u. See how $double^{down}(u)$ depends on this value in its two children u_1, u_2. It should contain (for $i = 1, 2$) all pairs $(p_\uparrow, p_\downarrow)$ such that for some states $(q_\uparrow, q_\downarrow) \in double^{down}(u_i)$ both pairs $(p_\uparrow, q_\uparrow)$ and $(p_\downarrow, q_\downarrow)$ are in $trans(u, u_i)$. We have to be a little careful to calculate them in $O(|Q|^3)$: In a first step we calculate the set of state pairs $(p_\uparrow, q_\downarrow)$ such that for some $q_\uparrow$ there is $(q_\uparrow, q_\downarrow) \in double^{down}(u_i)$ and $(p_\uparrow, q_\uparrow) \in trans(u, u_i)$. In a second step we calculate the required set. Straightforward implementation of both steps gives time $O(|Q|^3)$. To $double^{down}(u)$ we should also include all pairs from $core(u)$. The top-down pass is similar. $\square$

11. Solving the core problem

We now come to the last part of Theorem 1.1, where we prove Theorem 10.3. Recall, that we have to calculate the set $core(u)$ for every node u.

The main object used in this section is a *bracket*, which is a tuple $(v_\uparrow, v_\downarrow, Q_\uparrow, Q_\downarrow)$ where $v_\uparrow$ and $v_\downarrow$ are nodes such that $v_\uparrow$ is an ancestor of $v_\downarrow$ and $Q_\uparrow$ and $Q_\downarrow$ are sets of states. When both of the sets contain just one state, we simply write $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$. We say that the bracket *generates* a pair of states $(p_\uparrow, p_\downarrow)$ in a node u, when $v_\uparrow$ is an ancestor of u, $v_\downarrow$ is a descendant of u (they need not to be proper) and for some states $q_\uparrow \in Q_\uparrow$, $q_\downarrow \in Q_\downarrow$ there is $(p_\uparrow, q_\uparrow) \in trans(u, v_\uparrow)$ and $(p_\downarrow, q_\downarrow) \in trans(u, v_\downarrow)$. We say that a set of brackets is *correct* (respectively *complete*), when for every node u the set of pairs of states generated in u by all the brackets from the set is a subset (a superset) of $core(u)$.

The algorithm will keep at each moment some correct and complete set of brackets. Every bracket is remembered in its $v_\downarrow$ node. The general idea of the algorithm is to convert one brackets into other, simpler brackets. We say, that a bracket is *trivial*, when it is of the form $(v, v, q_\uparrow, q_\downarrow)$, i.e. when the two nodes are equal and both of the two sets of states are singletons. The goal is to calculate a correct and complete set of trivial brackets. Once we only have trivial brackets, we immediately have the function $core(u)$ in any node u: it contains all pairs $(q_\uparrow, q_\downarrow)$ for which we have a bracket $(u, u, q_\uparrow, q_\downarrow)$ in our set.

The initial set of brackets is the following: For any nodes $v_\uparrow$, $v_\downarrow$ such that $v_\uparrow$ is a parent of $v_\downarrow$ in some d-skeleton and for any node $v_\uparrow = v_\downarrow$ in some d-skeleton we have a bracket $(v_\uparrow, v_\downarrow, class(v_\uparrow, d), class(v_\downarrow, d))$. Directly from the definition of $core(u)$ follows, that such set of brackets is correct and complete. There are $O(|t|)$ brackets in the set.

Step 1. After this step we want to have only brackets, in which $v_\uparrow$ is a t-ancestor or a t-sibling of $v_\downarrow$ (i.e. ancestor or sibling in the original tree t).

Take any bracket $(v_\uparrow, v_\downarrow, Q_\uparrow, Q_\downarrow)$. Let v be the t-sibling of $v_\uparrow$, which is a t-ancestor of $v_\downarrow$ (we may find v as the least common ancestor of $v_\downarrow$ and the rightmost t-sibling of $v_\uparrow$). We replace the bracket by brackets $(v_\uparrow, v, Q_\uparrow, prec(v, v_\downarrow, Q_\downarrow))$ and $(v, v_\downarrow, prec(v, v_\uparrow, Q_\uparrow), Q_\downarrow)$, using Lemma 9.2 to calculate $prec$. Then all pairs of states, which were generated by the original bracket in any node between $v_\uparrow$ and v (including $v_\uparrow$ and v) are generated by the first new bracket and in any node between v and $v_\downarrow$ by the second new bracket. We still have $O(|t|)$ brackets.

Step 2. This is the most complex step. After it we should have only brackets $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ of one of four types: trivial brackets, brackets in which $v_\uparrow$ is the t-parent of $v_\downarrow$, brackets in which state $q_\uparrow$ has a $\overline{parent}$ loop and brackets in which $v_\uparrow$ is a t-sibling of $v_\downarrow$ and state $q_\uparrow$ has a `prev` loop.

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

Now we handle brackets $(v_\uparrow, v_\downarrow, Q_\uparrow, Q_\downarrow)$ where $v_\uparrow$ is a t-ancestor of $v_\downarrow$ (now only proper, since the case $v_\uparrow = v_\downarrow$ is already considered). Consider the sequence $v_\downarrow = v_0, v_1, \ldots, v_n = v_\uparrow$ where v_{i+1} is the t-parent of v_i. Let $k = \max(0, n - |Q|)$. We calculate the nodes v_i and sets $Q_i^\uparrow = prec(v_i, v_\uparrow, Q_\uparrow)$ for $k \le i \le n$. Each of them is calculated using the previous one in time $O(|Q|^2)$, as in the proof of Lemma 9.2. We also calculate $Q_k^\downarrow = prec(v_k, v_\downarrow, Q_\downarrow)$ using Lemma 9.2 and then step by step sets $Q_i^\downarrow = prec(v_i, v_\downarrow, Q_\downarrow)$ for $k < i \le n$. Then we add brackets $(v_{i+1}, v_i, q_{i+1}^\uparrow, q_i^\downarrow)$ for all $q_{i+1}^\uparrow \in Q_{i+1}^\uparrow, q_i^\downarrow \in Q_i^\downarrow, k \le i < n$.

We also add brackets $(v_i, v_\downarrow, q_i^\uparrow, q_\downarrow)$ for all states $q_i^\uparrow \in Q_i^\uparrow$ with a $\overline{parent}$ loop, $k \leq i \leq n$. There are $O(|Q|^3)$ new brackets. The first type of new brackets is allowed because v_{i+1} is the t-parent of v_i, the second type because $q_i^\uparrow$ has a $\overline{parent}$ loop.

Now see that the new set of brackets is complete. State pairs generated in all the nodes between v_k and $v_\uparrow$ by the original bracket are now generated by some of the new brackets of the first type. Consider any pair $(p^\uparrow, p^\downarrow)$ generated by the original bracket in some node u below the node v_k. This means that on some string description of the simple path from u to $v_\uparrow$ the automaton may be taken from state $p^\uparrow$ to some state $q_n \in Q_\uparrow$. Let $q_k, \ldots, q_{n-1}$ be the states of the run after the nodes $v_k, \ldots, v_{n-1}$. Because there are only $|Q|$ states, there has to be $q_r = q_{r+1}$ for some $k \leq r < n$. See that q_r has a $\overline{parent}$ loop and that $q_r \in Q_r^\uparrow$, so there is a new bracket $(v_r, v_\downarrow, q_r, q_\downarrow)$ by which the pair $(p^\uparrow, p^\downarrow)$ is also generated.

Brackets $(v_\uparrow, v_\downarrow, Q_\uparrow, Q_\downarrow)$ where $v_\uparrow$ is a left t-sibling of $v_\downarrow$ are handled in a very similar way. We consider the sequence $v_\downarrow = v_0, v_1, \ldots, v_n = v_\uparrow$ in which v_{i+1} is the previous t-sibling of v_i (so it is its parent in the binary tree $\widehat{t}$). In the part near $v_\uparrow$ we add trivial brackets (as there are no other nodes between v_i and v_{i+1}). In the second part we add brackets in which $q_i^\uparrow$ has a `prev` loop (they are allowed because all the nodes v_i are t-siblings).

Step 3. After this step we should have only trivial brackets and brackets $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ in which $v_\uparrow$ is the t-parent of $v_\downarrow$.

We want to eliminate brackets in which $q_\uparrow$ has a $\overline{parent}$ loop. The key observation is that when we have two such brackets $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ and $(v'_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ and $v_\uparrow$ is an ancestor of $v'_\uparrow$, then the second bracket may be removed, because $(q_\uparrow, q_\uparrow) \in trans(v'_\uparrow, v_\uparrow)$ (by Definition 1 state $q_\uparrow$ has also `prev` loop). So for each $v_\downarrow$ we always keep only at most $|Q|^2$ brackets, for every pair of states at most one, and we immediately remove the redundant ones. We consider every $v_\downarrow$ starting from the lowest nodes and ending in the root. Let v be the parent of $v_\downarrow$ in the binary tree $\widehat{t}$. We replace a bracket $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ by brackets $(v_\uparrow, v, q_\uparrow, q)$ for every q such that $(q, q_\downarrow) \in trans(v, v_\downarrow)$ (these brackets are processed again, when we are in the node v) and by trivial brackets $(v_\downarrow, v_\downarrow, q, q_\downarrow)$ for every q such that $lev(first^{up}(v, q, q_\uparrow)) \geq lev(v_\uparrow)$. This may be done in time $O(|Q|)$, so the whole processing takes time $O(|t||Q|^3)$. Completeness of the new set of brackets is clear. For correctness we use the fact that $q_\uparrow$ has $\overline{parent}$ and `prev` loops, thanks to that $(q_\uparrow, q_\uparrow) \in trans(first^{up}(v, q, q_\uparrow), v_\uparrow)$ (we may go up staying in the state $q_\uparrow$.

Brackets $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ where $q_\uparrow$ has a `prev` loop and $v_\uparrow$ is a t-sibling of $v_\downarrow$ are eliminated in exactly the same way. For correctness a $\overline{parent}$ loop in $q_\uparrow$ is not needed, because $v_\uparrow$ may be reached from $v_\downarrow$ using only `prev` axis.

Step 4. In the last step we want to eliminate brackets in which $v_\uparrow$ is the t-parent of $v_\downarrow$, leaving only trivial brackets. Once again for every $v_\downarrow$ we have at most $|Q|^2$ brackets, as $v_\uparrow$ (the t-parent of $v_\downarrow$) is the same in all brackets for fixed $v_\downarrow$. We consider every $v_\downarrow$ from the lowest nodes. Let v be the parent of $v_\downarrow$ in the binary tree $\widehat{t}$. Then we replace $(v_\uparrow, v_\downarrow, q_\uparrow, q_\downarrow)$ by brackets $(v_\uparrow, v, q_\uparrow, q)$ for every q such that $(q, q_\downarrow) \in trans(v, v_\downarrow)$ and by trivial brackets $(v_\downarrow, v_\downarrow, q, q_\downarrow)$ for every q such that $(q, q_\uparrow) \in trans(v_\downarrow, v_\uparrow)$. This is done in time $O(|Q|)$ (recall, that we remember in the tree values of $trans$ between a node and its t-parent), so the whole procedure takes time $O(|t||Q|^3)$.

12. Concluding remarks

Although the `position()` and `last()` functions are not handled here, in the most natural case they may be replaced by `count()` function. We mean the case, when they are used with `child` axis—then `position()` returns the number of left siblings, which satisfy some node test, and `last()` returns the number of all such siblings. This may be easily expressed by the `count()` function.

We leave as future work improvements of other fragments of XPath. It is very likely, that evaluation of full XPath does not require $O(|t|^4)$ data complexity. It would be also interesting to give an algorithm which evaluates a path expression in time linear in number of selected pairs.

13. References

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

[2] M. Bojańczyk and P. Parys. XPath evaluation in linear time. In *PODS*, pages 241–250, 2008.

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

[4] R. Pichler G. Gottlob, C. Koch. Xpath query evaluation: Improving time and space eficiency. In *ICDE'03*, pages 379–390, 2003.

[5] R. Pichler G. Gottlob, C. Koch. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

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

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

[8] C. Koch M. Benedikt. XPath leashed. *ACM Computing Surveys*.

[9] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

APPENDIX

Proof (of Lemma 5.2) This is a fairly standard construction. First, for each node u we calculate the subset $down(u)$ of state pairs in $loop(u)$ that correspond to paths that only visit descendants of u. The value of $down$ for u depends only on the values of $down$ in the two children of u, and the names in u. Assume for a moment that having this information we may calculate $down(u)$ effectively. Then the values $down(u)$ can be calculated in a single bottom-up pass through the tree. Second, we calculate for each node u the subset $up(u)$ of $loop(u)$ that correspond to paths that never visit descendants of u. The value of up in u depends only on the value of up in the parent of u and the value of $down$ in the sibling of u. In particular, the values $up(u)$ can be calculated in a single top-down pass through the tree. Once we have $down$ and up, the function $loop(u)$ can easily be calculated, as the transitive closure of union of $down(u)$ and $up(u)$.

The above algorithm would have the declared complexity, if we can calculate $down(u)$ basing on $down$ in the two children u_1, u_2 of u in time $O(|Q|^3)$. First note, that in time $O(|Q|^3)$ we may calculate 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. In $down(u)$ there should be pairs (p, q) such that from p to q there is a transition reading letter (j, a) such that j-th name in u is a. There should be also pairs corresponding to runs which read a letter $\overline{\texttt{child}}$, then do something from $loop(u_1)$ and then read a letter $\overline{\texttt{parent}}$. Let R_c be the set of pairs (p, q) such that from p to q there is a transition reading $\overline{\texttt{child}}$. Similarly R_p for $\overline{\texttt{parent}}$. Then to $down(u)$ we add the composition of R_c with $loop(u_1)$ and with R_p. Similarly for u_2 and axes `next` and `prev`. Then $down(u)$ is the transitive closure of all these pairs, since every string description of every path from u to u using only descendants of u may be divided into such fragments. Using similar technique we may calculate values of up in the two children of u basing on $up(u)$ and the values of $down$ in the children of u. $\square$

Proof (of Theorem 5.1) By $\widehat{t}'$ we denote the binary version of the tree t (as defined above) with the same node names as in t (in the final $\widehat{t}$ we will have different names). In a first step we modify α. We add the two new one-step axes $\overline{\texttt{child}}$ and $\overline{\texttt{parent}}$. We also add two new multistep axes $(\overline{\texttt{child}} + \texttt{next})^*$ and $(\overline{\texttt{parent}} + \texttt{prev})^*$. The axis $\overline{\texttt{child}}$ corresponds to going to the leftmost child and $(\overline{\texttt{child}} + \texttt{next})^*$ corresponds to going several times to the leftmost child or to the next sibling. The other two axes are their inverses. Then we may replace in the path expression α every use of `child`, `parent`, `child*` and `parent*` axes by some combination of the other axes (the other four original and the four new), so that the resulting α' selects the same pairs of nodes as α. For example every appearance of `child` should be replaced by $\overline{\texttt{child}} \cdot \texttt{next}^*$: we go to the leftmost child and then any number of times to its next sibling. The paths selected by such α' strictly correspond to structure of the binary tree $\widehat{t}'$: $\overline{\texttt{child}}$ goes to the left child in $\widehat{t}'$, `next` goes to the right child, etc.

In the natural way we compile α' to a nondeterministic automaton $\mathcal{A}'$ reading string descriptions of paths in $\widehat{t}'$. This means that a pair of nodes u, v is selected by α' (or by α) if and only if $(q_I, q_F) \in trans^{all}_{\mathcal{A}',\widehat{t}'}(u, v)$ for some initial state q_I and accepting state q_F. Such automaton $\mathcal{A}'$ satisfies conditions 1-3 from Definition 1 (recall that in path expressions we do not allow the Kleene star, only union and concatenation).

To get condition 4, we use Lemma 5.2 to calculate the values of the $loop$ function. We remember them in the tree $\widehat{t}'$, getting a tree $\widehat{t}$: in the name array for every node u we add elements corresponding to all pairs (q_i, q_j) for $i \leq j$ and we write there `true` or `false` depending on whether $(q_i, q_j) \in loop(u)$ or not. Because $\mathcal{A}'$ is an XPath automaton, only such pairs may be in the sets $loop$. We also modify $\mathcal{A}'$ getting $\mathcal{A}$: between every two states (q_i, q_j) for $i < j$ we add a transition which reads `true` in the name corresponding to (q_i, q_j).

If $\mathcal{A}$ accepts a string description of some path between some u and v, then also $\mathcal{A}'$ accepts a string description of some (maybe longer) path between them, because the new transition from q_i to q_j may be used only in a node for which $(q_i, q_j) \in loop(u)$. On the other hand see, that if $\mathcal{A}$ accepts a string description of some path between some u and v, then it accepts also a simple string description of the simple path between them.

The last part of the theorem is immediate: sets $loop(u) = trans_{\mathcal{A},\widehat{t}}(u, u)$ are already calculated, sets $trans_{\mathcal{A},\widehat{t}}(u, v)$ for v being a child or a parent of u are compositions of three known sets. $\square$

Proof (of Proposition 8.1) We use the algorithm, which in constant time (after linear preprocessing) finds LCA of any two nodes. Note, that using this algorithm we can also easily check if one node is an ancestor of another node.

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 the LCA algorithm allows to calculate skeletons in linear time. We consider each skeleton separately, all leaves in a 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 least common ancestor w of this new leaf v and the rightmost already processed leaf u. We need to add w in the appropriate place in the skeleton. We compare w with the nodes on the rightmost path of the skeleton, starting from u and going up. When w is between some node and its parent in the skeleton, we add it there, together with attached v. It is also possible that w is over the root of the current skeleton.

Why does it work in linear time? Possibly there are many nodes on the rightmost path of the current version of a skeleton. However always only one of the visited nodes is an ancestor of w. Other visited nodes, which are not ancestors of w no longer will be on the rightmost path, so every node can be visited only once in that role. $\square$

Proof (of Lemma 10.2) 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 u of a d-skeleton we calculate the part $class^{down}(u, d)$ of $class(u, d)$ such that the node v from the definition is a descendant of u (which includes $v = u$). The crucial observation is that the set $class^{down}(u, d)$ depends only on these sets for its two d-children u_1, u_2 and that it may be calculated in $O(|Q|^3)$: it is a union of $prec(u, u_i, class^{down}(u_i, d))$ for $i = 1, 2$ and if the string value of u is d, it is also a union with the set of accepting states Q_F.

In the top-down pass we calculate the part $class^{up}(u, d)$ of $class(u, d)$ such that the node v is not a descendant of u, this is very similar to the above. The expected set $class(u, d)$ is the sum of $class^{down}(u, d)$ and $class^{up}(u, d)$. $\square$