XPath evaluation in linear time

Mikołaj Bojańczyk *
Warsaw University, Poland
bojan@mimuw.edu.pl

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

ABSTRACT

We consider a fragment of XPath where attribute values can only be tested for equality. We show that for any fixed unary query in this fragment, the set of nodes that satisfy the query can be calculated in time linear in the document 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

Languages, Theory

1. INTRODUCTION

In this paper, we present an algorithm that, given an XPath node selecting query φ and XML document t, returns the set of nodes in t that satisfy φ. Previously proposed algorithms [6, 5, 8] would either have running time quadratic in $|t|$, or have a running time linear in $|t|$, but use a fragment of XPath without attribute values. The algorithm proposed in this paper can deal with attribute values and runs in time $O(2^{|\varphi|} \cdot |t|)$. In other words, the algorithm has linear time data complexity, since it is linear time once the query φ is fixed. The algorithm works for queries expressed in a fragment of XPath that can only check attribute values for equality.

XPath evaluation algorithms that are built into browsers are very inefficient, and can have running times that are exponential in the size of the queried XML document [6]. There have been a number of papers devoted to improving XPath evaluation, which can be grouped into two main approaches, see e.g. [7] for a survey.

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

One idea, as used in e.g. [6] and improved in [5], is to use dynamic programming. This allows evaluation algorithms that are polynomial 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.

Another idea is to compile queries into finite-state tree automata, see [8] for a survey. This approach only works if the node-test does not refer to attribute values (a fragment called CoreXPath), and therefore an XML document can be identified with a finitely labelled 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 [6].)

This paper can be seen as a generalization of the automata-theoretic framework to node tests that use attribute values. We consider a fragment of XPath where only equality over attribute values can be tested, as in e.g. [1]. In the terminology of [7], this is the fragment of FOXPath where only $=$ and $\neq$ can be used on attribute values (and not $\leq$, etc.). For instance, key constraints such as "select nodes whose attribute value in `attr` appears nowhere else in the document" use only equality, unlike the query "select nodes whose attribute value in `attr`, as a number, is maximal in the document". This restriction is reasonable, since it still allows to model keys with attribute values. The following is our main result:

Theorem 1.1
Let φ be a node test of XPath where attribute values are only tested for equality, and t an XML document. The set of nodes of t that satisfy φ can be computed in time $O(2^{|\varphi|} \cdot |t|)$.

The XML document t is represented as a text file, eg.

$$\langle \texttt{a} \rangle \langle \texttt{b attr} = \texttt{"val1"} \rangle \texttt{text} \langle / \texttt{b} \rangle \langle \texttt{b attr} = \texttt{"val2"} \rangle \langle / \texttt{b} \rangle \langle / \texttt{a} \rangle \ .$$

The length $|t|$ is the length of the text file.

In our linear time algorithm, we use "factorization forests", an important algebraic concept introduced by Simon [9]. Originally introduced in formal language theory to study tropical semirings and similar constructions, factorization forests have proved very useful in analyzing regular languages. In recent work [4, 2, 3], Colcombet has shown that factorization forests can be calculated by a deterministic transducer. Here is one corollary, crucial to our approach:

> Fix a regular word language $L \subseteq \Sigma^*$. There is an algorithm, which does a linear time precomputation on an input word $a_1 \dots a_n \in \Sigma^*$, and can then answer any query $a_i \cdots a_j \in L$? in constant time.

The algorithm in this paper builds on this idea, by applying factorization forests to accelerate evaluation of sub-queries in XPath.

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. In this section we also present a naive algorithm to illustrate why it is difficult to obtain linear time. The algorithm is then detailed in Sections 4, 5, 6 and 7. In general terms, Sections 4, 5 and 6 use finite automata, leaving the algebra—monoids and factorization forests—to Section 7. In particular, the part of the algorithm that is exponential in the query size is located entirely in Section 7, where the query is translated into a monoid, which incurs an exponential blowup. Finally, in Section 8, we discuss possible extensions of this work.

2. DATA MODEL AND XPATH

In this section we define the data model, as well as the fragment of XPath without arithmetic.

2.1 Data model

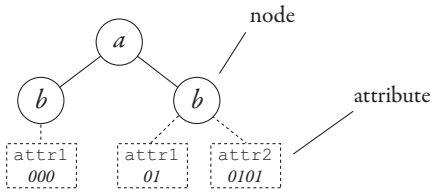
We represent an XML document as a tree, called a *data tree*. The tree is unranked, i.e. a node may have any number of children, and the children are ordered. Every node is assigned a *tag name*, which is taken from a finite alphabet Σ. Furthermore, a node may have a number of attributes, each of them with a string value (or possibly several string values, although duplicates are ignored). In other words, there is a finite set Γ of *attribute names*, and for each tree node u and attribute name a, there is a finite set $u.a \subseteq \{0,1\}^*$ of {attribute values}. We assume here that attribute values are strings over $\{0,1\}$, but any other alphabet could be used, e.g. ASCII characters. Since most of the time we will be dealing with data trees, we will sometimes write tree instead of data tree.

A *tree without data* is one where there are no attribute names (and therefore also no attribute values). In other words, this is a finite, unranked, ordered, finitely labeled tree.

Consider for instance the following XML document:

⟨a⟩
  ⟨b attr1 = "000"⟩1111⟨/b⟩
  ⟨b attr1 = "01" attr2 = "0101"⟩⟨/b⟩
⟨/a⟩ .

The data tree representing this document will use tag names $\Sigma = \{a, b\}$ and attribute names $\Gamma = \{\texttt{attr1}, \texttt{attr2}\}$. The data tree will look like this:



Note that in our representation, we ignore the text contained directly in the tags (e.g. the string 1111 in the document example above). This text could be modeled as another attribute.

Trees will be denoted by letters s, t. Nodes will be denoted by u, v, w. Edges will be denoted by x, y, z. Attribute values will be denoted by d. The set of nodes in a data tree t is written $\mathrm{dom}(t)$. The size of a data tree is the number of nodes plus the sum of lengths of its attribute values:

$$|t| = |\mathrm{dom}(t)| + \sum_{u \in \mathrm{dom}(T)} \sum_{a \in \Gamma} \sum_{d \in u.a} |d|$$

The size measure defined above is linear in the size for the text file representation, since the only difference is in the special characters like ⟨ or ".

2.2 XPath

In this section we define the fragment of XPath that is used in this paper. Basically, these are queries that can only compare attribute values via equality. For instance, testing the length of a string in an attribute value is not allowed, neither is parsing the string to extract its first letter, etc.

There are two types of expressions: programs and node tests. A *program* is a binary query. In each tree, a program will select a set of pairs (u, v) of nodes. Intuitively a program will describe the path from u to v, although the path might not be the shortest one. A typical program 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*, on the other hand, 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. In general, the two types of expression are mutually recursive, as defined below:

- Every tag name a is a node test, which holds in nodes with tag a.
- Node tests admit negation, conjunction and disjunction.
- There are two types of *atomic* program. Every axis

  $$\texttt{child} \quad \texttt{parent} \quad \texttt{next} - \texttt{sibling} \quad \texttt{prev} - \texttt{sibling}$$

  is an atomic program. Furthermore, a node test φ can be interpreted as an atomic program $[\varphi]$, which holds in pairs (u, u) such that φ holds in u.
- In general, a program is a regular expression over atomic programs. In other words, programs contain the atomic programs, and are closed under union, composition and Kleene star. For instance, the program $\texttt{child}^*$ selects (u, v) if v is a descendant of u.
- If α, β are programs and a, b are attribute names, then

  $$\alpha.a \ \texttt{eq} \ \beta.b$$

  is a node test. It selects a node u if there exist nodes v, w with a common attribute value in $v.a \cap w.b$ and such that (u, v) is selected by α and (u, w) is selected by β.
- Similar to the above, a node test

  $$\alpha.a \ \texttt{neq} \ \beta.b$$

  is also defined. Here, the requirement is that there are two different attribute values $d \in v.a$ and $d' \in w.b$.

Note that the operators `eq` and `neq` are not mutually exclusive. A node may satisfy none or one or both of $\alpha.a$ `eq` $\beta.b$ and $\alpha.a$ `neq` $\beta.b$.

Note that we allow the Kleene star in programs, while usually XPath does not. We do so because our techniques work even when the Kleene star is present. Also, the Kleene star allows us to use a smaller set of four axes.

When referring below to XPath, we mean the fragment above. This fragment of XPath is strictly more expressive than the two-variable logic introduced in [1]: every formula of the two-variable logic can be translated into XPath, but not the other way round. Note however, that this paper is about query evaluation, while [1] is about satisfiability, a more difficult decision problem.

3. PAPER OVERVIEW

In this section we overview the structure of the algorithm. Then, we show how a naive approach fails to give a linear time algorithm. Finally, we show that without loss of generality, only binary trees need be considered.

3.1 Proof strategy

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

Consider a node test φ defined in XPath. We want to present an algorithm that selects the nodes of a tree t satisfying φ. We want the algorithm to run in time linear in $|t|$; although the constant in the linear time will depend—exponentially—on the node test φ. The algorithm works by induction on the size of φ. The base case, when φ just tests the label, is immediate.

Consider now the induction step: a node test $\alpha.a$ `eq` $\beta.b$ (or similarly, but with `neq`). Let $\varphi_1, \ldots, \varphi_n$ be the node tests that appear in the programs α and β. Using the induction assumption, we run a linear time algorithm for each of these nodes tests, and label each node in the tree with the set of node tests from $\varphi_1, \ldots, \varphi_n$ that it satisfies. In other words, we may assume without loss of generality that the only node tests appearing in atomic programs in the regular expressions from α and β are tag names.

It remains therefore to show linear time algorithms for node tests of the following two forms

$$\alpha.a \texttt{ eq } \beta.b \qquad\qquad \alpha.a \texttt{ neq } \beta.b$$

where the only node tests appearing in atomic programs in the regular expressions from α and β are tag names.

The second type of node tests, with inequality, is easier, and will be dealt with in Section 4. The rest of the paper is then devoted to the first type of node tests, where equalities are used.

First, however, we present a naive algorithm for the equalities, to highlight the difficulties in obtaining linear time.

3.2 Naive algorithm

In this section, we present a naive algorithm—actually, two—for calculating the set of nodes satisfying

$$\alpha.a \texttt{ eq } \beta.b$$

For d an attribute value, let A_d be the set of nodes u, such that (u, v) is selected by α for some node v with $d \in v.a$. We will compute simultaneously all the sets A_d. We proceed in two steps:

- In the first step, we compute the set of pairs (u, v) selected by α. By using a dynamic algorithm, this can be done in time linear in the size of α and quadratic in the number of nodes in the tree.
- In the second step, we process each pair (u, v) computed above, and add it to a set A_d if $d \in v.a$.

In a similar way we calculate the sets B_d, corresponding to $\beta.b$. The nodes that satisfy $\alpha.a$ `eq` $\beta.b$ or $\alpha.a$ `neq` $\beta.b$ can be then calculated by using these sets.

Even if we assume all set operations to be done in constant time, the above procedure is quadratic in the number of nodes in the tree. A carefully designed program can actually achieve this quadratic bound.

A different naive algorithm would work as follows. For each possible attribute value d, it would find the set of nodes u such that $\alpha.a$ `eq` $\beta.b$ holds, with the witnessing data value being d. With some effort, each pass for d can be done in time linear in the size of the tree and the sizes of α, β. However, the number of attribute values may be linear in the size of the tree, hence also a quadratic complexity.

3.3 Binary data trees

A data tree is called *binary* if every node has at most two children, called the *left child* and the *right child* (it is possible that a node has no children, only the left child, only the right child, or both). Furthermore, whether or not a node is a left child can be read from its label. More formally, there is a partition of the tag names $\Sigma = \Sigma_L \cup \Sigma_R$ such that left children only use labels from Σ_L, and right children only use labels from Σ_R

Proposition 3.1 Without loss of generality, we may assume that the input is given as a binary data tree.

One of the advantages of binary trees is that we can reduce the axes to `parent` and `child`. For instance, the next-sibling axis comes down to going to the parent and then the right child, if the source node was a left child:

$$[\bigvee_{a \in \Sigma_L} a] \cdot \texttt{parent} \cdot \texttt{child} \cdot [\bigvee_{a \in \Sigma_R} a] \ .$$

From now on, all trees considered will be binary.

4. INEQUALITIES

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

$$\alpha.a \texttt{ neq } \beta.b$$

The basic idea is as follows. Let (u, v) be a node pair selected by the program α. Any attribute value $d \in v.a$ is called a *representative for $\alpha.a$ in u* . Likewise for $\beta.b$.

For each node u, we will calculate if there are zero, one, or at least two representatives for $\alpha.a$-values in u. If there is one, we will also remember which one. Likewise for $\beta.b$. This information is sufficient, since a node u satisfies $\alpha.a$ `neq` $\beta.b$ if and only if either: a) there is at least one representative for $\alpha.a$, and there are at least two representatives for $\beta.b$; or b) there is at least one representative for $\beta.b$, and there are at least two representatives for $\alpha.a$; or c) there are exactly one representative each for $\alpha.a$ and $\beta.b$, but these are different.

It remains to show that the information about the representatives can be calculated in linear time. We proceed in

two steps. First, in Section 4.1, we show how to efficiently evaluate word automata on loops in the tree. Then, in Section 4.2 we show how the representatives can be calculated.

4.1 Calculating loops

A *path* in a data tree is a sequence of nodes $u_1, \ldots, u_n$ where each two consecutive nodes are connected via either the `child` or `parent` axis. A *string description* of a path $u_1, \ldots, u_n$ is a word $a_1 m_1 a_2 a_2 m_2 a_3 \cdots a_{n-1} m_{n-1} a_n$, where a_i is the tag name of u_i and m_i is either `child` or `parent` depending on the relationship between u_i and u_{i+1}. Every edge from u_i to u_{i+1} gives a fragment $a_i m_i a_{i+1}$, so tag names $a_2, \ldots, a_{n-1}$ are repeated twice in the description. Thanks to that we have a desired composition property: a string description of a path $u_1, \ldots, u_n$ is a string description of the path $u_1, \ldots, u_k$ concatenated with a string description of the path $u_k, \ldots, u_n$ (for any $1 \leq k \leq n$).

Fix a nondeterministic automaton $\mathcal{A}$ that reads string descriptions. The focus of this section is to efficiently calculate runs of this automaton over string descriptions on various paths in a data tree.

Let Q be the states of $\mathcal{A}$. Let u, v be two nodes in a data tree t. We write $trans(u, v)$ for the set of state pairs (p, q) such that for some path from u to v, the string description of the path can take the automaton $\mathcal{A}$ from state p to state q. This set depends on both t and $\mathcal{A}$, but we hope the two will always be clear from the context. Note that two objects are quantified existentially here: the path from u to v, and the run of the nondeterministic automaton.

Lemma 4.1 Let t be a binary data tree, and let $\mathcal{A}$ be a nondeterministic automaton with states Q. The function

$$loop : \mathrm{dom}(t) \to P(Q^2) \qquad loop(u) = trans(u, u)$$

can be calculated in time $O(|t||Q|^2)$.

Note that the above result is not concerned with attributes in the tree, since the values $trans(u, u)$ only depend on the tree structure and the tag names.

Proof
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 label of u. Therefore, 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 *down* in the sibling of u (if such a sibling exists). 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. □

4.2 Calculating representatives

In order to calculate the representatives, we slightly generalize the problem, so that a dynamic algorithm can be applied. Let $\mathcal{A}$ be a nondeterministic automaton with states Q. For each pair of states p, q of this automaton, and an attribute a, a representative for (p, q, a) in a node u is an attribute value d that can be found in $v.a$, for some v with $(p, q) \in trans(u, v)$. The node v is called a witness for the representative (there may be several witnesses).

Finding representatives in this new sense is a generalization of the problem described at the beginning of Section 4, since any program α or β can be simulated by a nondeterministic automaton of the same size.

In order to find the representatives, we use the same type of two-step (first a bottom-up pass, then a top-down pass) approach as in the section on finding loops.

5. SKELETONS

We now proceed to find the nodes that satisfy a node test

$$\alpha.a \ \texttt{eq} \ \beta.b \ .$$

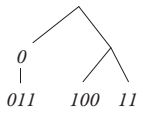
This part is much more involved than the one with inequalities, and takes up Sections 5, 6 and 7.

The first thing we need to do is show how a data tree is stored in memory by the algorithm; this is the subject of this section. We also show in Section 5.2 how this representation can be used to efficiently evaluate automata.

5.1 Skeleton representation

In this section, we describe how a tree is stored in memory by the algorithm.

A data tree is stored by having a record for each node, called the *node record*. This record contains the tag name, as well as pointers to the node records of the: parent, left child, right child. Some of these may be empty, if the appropriate nodes do not exist. Furthermore the node record for u contains for each attribute name a the set of attribute values $u.a$. To speed things up, this set is organized as a tree, where the nodes are attribute values and their longest common prefixes. For instance, if the set of attribute values is $\{0, 011, 11, 100\}$, then the following tree is stored:

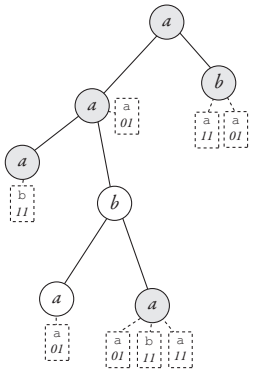


The *record representation* of a data tree consists of the node records for all the nodes of the data tree. The following simple result is given without proof:

Fact 5.1 The record representation of a data tree t can be computed in time linear in $|t|$.

Let u and v be two nodes in a data tree t. The *closest common ancestor* 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.

We say a node u has *attribute value* d if it has attribute value d under some name, i.e. $d \in u.a$ holds for some $a \in \Gamma$. Let t be a data tree, and let $d \in \{0, 1\}^*$ be an attribute value. The *class of* d is the set of nodes that are closest common ancestors of two nodes that have attribute d. Note that every node with attribute value d is in the class of d, since a node u is the closest common ancestor of u, u. For instance, in the picture below, the highlighted nodes describe the class of $d = 11$. Note that both the root and its left child are selected since they are closest common ancestors.



In the evaluation algorithm, it will be convenient to reason about classes. Therefore, for each attribute 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 an attribute value. The *d-skeleton* of t, is a tree obtained by only keeping the nodes of t from the class of d. The tag names are sets of attribute names from t, and there are no attribute names (i.e.the d-skeleton is a tree without data). The tag name of a node u in the d-skeleton is the set of attribute names $a \in \Gamma$ such that d belongs to $u.a$ in t. The tree structure in the d-skeleton is inherited from t. In particular, u is a child of v in the d-skeleton only if in the tree t, u is a descendant of v, and no node between u and v belongs to the class of d.

Note that every d-skeleton is binary. This is because the original tree t was binary, and the class of d includes closest common ancestors. However, it may happen that some nodes in the d-skeleton have only one child in the d-skeleton, even though they had two children in t.

The *skeleton representation* of a data tree t consists of the record representation of t and all of its d-skeletons, see Figure 1. Furthermore, for each d-skeleton, each node record contains a pointer to the corresponding node in t.

Note that the sum of sizes of all skeletons in t is linear in t, since each leaf in a skeleton corresponds to an attribute value. The following result shows that the skeleton representation can also be calculated in linear time:

Proposition 5.2 The skeleton representation of a data tree can be calculated in linear time (in the size of the XML text file representing the document).

Proof
During the computation, we will be using D-skeletons, for sets D of attribute values. A D-skeleton is defined the same way as a d-skeleton—the only difference is that we take nodes which have attribute values in the set D (and all their closest common ancestors). We only consider a special form of these sets—sets $d+$ consisting of attribute values that have the string d as a prefix, including d. Note that all the nodes of the original tree are in the $\epsilon+$-skeleton, because every attribute value has the empty word as a prefix.

At every step we take a set $d+$ and partition it into three sets $\{d\}$, $d0+$ and $d1+$, some of which may be empty. This partition can be obtained in a single bottom-up pass through the nodes in the $d+$-skeleton. It is convenient here that the attributes values in the tree t are stored in a tree. Therefore we only need constant time to process a single node of t and check whether it contains attribute values in one of the three sets $\{d\}$, $d0+$ or $d1+$.

The partition of $d+$ into three parts takes time linear in the number of nodes in the $d+$ skeleton, which is bounded by twice the number of occurrences of an attribute values from $d+$.

We apply this partitioning process to the $\epsilon+$ skeleton, until we are only left with singleton skeletons $\{d\}$. How much time does this take? An upper bound is (twice) the number of occurrences of attribute values from $d+$, over all prefixes d of attribute values occurring in the data tree. Rearranging the summing order, it is the same as summing, for each occurrence of an attribute value d in the tree, the number of prefixes of d. But this is the same as summing the lengths of the attribute values (plus one). Therefore, the processing time is bounded by $|t|$. □

5.2 Running automata on skeletons

In this section we show how skeletons can be used to accelerate evaluation of a regular expression (or automaton) on paths in the tree.

In the lemma below, for an attribute value d and a node u in the d-class, we describe the paths that begin in u and end in some node v with attribute value d (we also keep track of the attribute name a with $d \in v.a$). Note that the requirement on u is weaker than that on v: u only need be in the d-class.

Lemma 5.3 Let t be a binary data tree, let a be an attribute name, and let $\mathcal{A}$ be a nondeterministic automaton with states Q. The function $class_a$, mapping an attribute value d and a node u in the d-class to

$$class_a(u,d) = \bigcup_{v : d \in v.a} trans(u,v)$$

can be calculated in time $O(|t||Q|^2)$.

We first remark on how the function $class_a$ is represented. We will use the skeleton representation here. For each attribute value d, the value $class_a(u,d)$ will be stored in the d-skeleton, inside the node record that corresponds to the node u.

Before we prove Lemma 5.3, we need to present an intermediate result, Lemma 5.4.

Let d be an attribute value. Let u, v be nodes in a binary data tree t. We say a node v is a d-child of u if v is a child of u in the d-skeleton. Since t is binary, every node has at most two d-children, for every attribute value d.

Let u be a node in any d-skeleton. Let i be either L or R (standing for left or right child). We write $child(u,d,i)$ for the set of state pairs $(p,q) \in trans(u,v)$, for v the i-th d-child of u.

Lemma 5.4 Let t be a binary data tree and let $\mathcal{A}$ be a nondeterministic automaton with states Q. The function $child(u,d,i)$ can can be calculated in time $O(|t||Q|^2)$.

Proof
The proof is done by inspecting the proof of Proposition 5.2. We calculate $child(u,d,i)$ while creating d-skeletons. We extend the notation of $child$ into $d+$-skeletons, we write $child(u,d+,i)$ for $trans(u,v)$, where v is the i-th $d+$-child of u. Together with a $d+$-skeleton we calculate $child(u,d+,i)$ for both i and for every node u in the $d+$-skeleton.

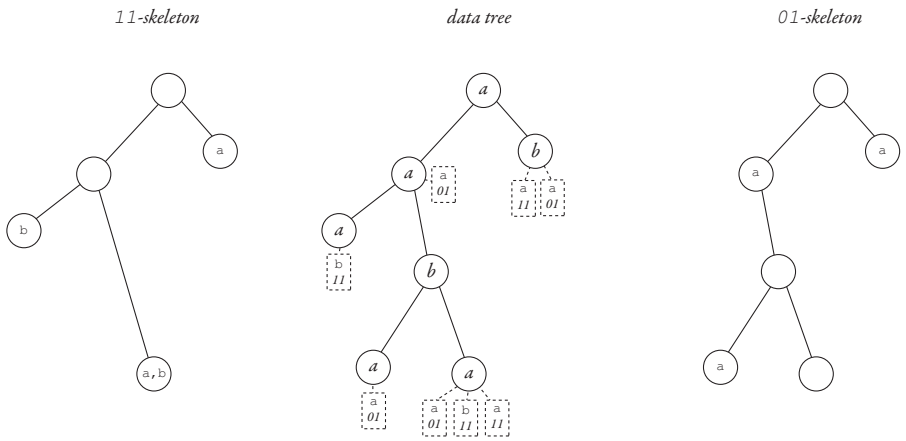


Figure 1: Skeleton representation of a data tree.

At the beginning we need $child(u, \epsilon+, i)$ for every node u. This is just a set of state pairs $(p, q) \in trans(u, v)$, where v is the i-th child of u. This value is easy to calculate using the set of loops $loop(u)$, as described in Lemma 4.1—first we may loop at u, then follow a transition from u to v, finally we may loop at v.

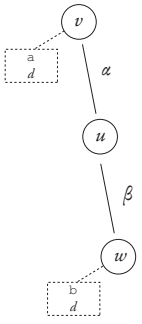
Later, when when partitioning the set $d+$ into the sets $\{d\}, d0+, d1+$, we need to calculate the function *child* for skeletons for these sets. We should do this in time linear in the size of the $d+$-skeleton. The calculation is done during the bottom-up processing described in Proposition 5.2. An edge of a new skeleton (for which we would like to calculate the *child* function) consist of some edges of the previous skeleton, and the value of *child* is obtained by composing the values of *child* for these edges. □

6. THE CORE PROBLEM

In this section, we identify a special case of node tests

$$\alpha.a \texttt{ eq } \beta.b \ ,$$

where already all the difficulty is contained. The special case is when α only moves up in the tree, and β only moves down in the tree. More formally, the *core problem* is a node test as above, where the only atomic programs that appear in α are `parent` and tag names, and the only atomic programs in β are `child` and tag names.



The main technical result in this paper is that the core problem can be solved in linear time:

Theorem 6.1
The set of nodes that satisfy an instance of the core problem can found in time $O(|t|)$.

It is only in the theorem above, i.e. in solving the core problem, that we will develop constants exponential in the size of the programs α, β. The constant will be polynomial in the size of a monoid recognizing the regular languages in the two programs, and a monoid can sometimes be exponentially large compared to a regular expression. The above theorem will be shown in Section 7. In this section, we show how solution to the core problem gives a solution to any node test of the form $\alpha.a$ `eq` $\beta.b$, without the restriction on α, β going in only one direction:

Proposition 6.2 Assume the core problem can be solved in time $O(|t|)$. The set of tree nodes that satisfy a node test of the form $\alpha.a$ `eq` $\beta.b$ can calculated in time $O(|t|)$.

The rest of this section is devoted to proving the above result.

Recall the notion of string description of a path that was used in Section 4.1. We assume that the programs α and β are given as nondeterministic word automata that read a string description of a path. A node pair (u, v) is selected by a program if there is some path from u to v whose description is accepted by the automaton. Without loss of generality, we may assume that the automaton for α and β is the same (denoted $\mathcal{A}$, with states Q) and only the accepting states are different, say $F_\alpha \subseteq Q$ for α, and $F_\beta \subseteq Q$ for β. We use a nondeterministic automaton, so Q is linear in the combined size of the expressions α and β.

In terms of the automaton $\mathcal{A}$, a node u satisfies the node test $\alpha.a \sim \beta.b$ if it satisfies the following property (*): for some nodes v, w with a common attribute value in both $v.a$ and $w.b$, the following hold:

$$trans(u, v) \cap \{q_I\} \times F_\alpha \neq \emptyset$$
$$trans(u, w) \cap \{q_I\} \times F_\beta \neq \emptyset$$

We will refer to the nodes v, w as *witnesses*.

Let p, q be states. Let $ande_{p,q}$ be set of nodes u' such that for some attribute value d, an ancestor v' of u', and a

descendant w' of u', the following hold for some states p', q':

$$(p, p') \in trans(u', v') \quad \{p'\} \times F_\alpha \cap class_a(v', d) \neq \emptyset \quad (V)$$
$$(q, q') \in trans(u', w') \quad \{q'\} \times F_\beta \cap class_b(w', d) \neq \emptyset \quad (W)$$

The set $dean_{p,q}$ is defined analogously, except that v is a descendant, and w an ancestor. In both cases, the descendants and ancestors need not be proper.

For instance, $u \in ande_{q_I,q_I}$ is a sufficient, but not necessary, condition for property (*). Proposition 6.2 will follow once we demonstrate the following three lemmas.

Lemma 6.3 For each two states p, q, the set $ande_{p,q}$ and $dean_{p,q}$ can be computed in linear time.

Lemma 6.4 A node u satisfies $\alpha.a$ `eq` $\beta.b$ if and only if it satisfies (**): there are states p, q and a node u' such that

$$(q_I, p), (q_I, q) \in trans(u, u') \qquad u' \in ande_{p,q} \cup dean_{p,q}$$

Lemma 6.5 The set of nodes that satisfy condition (**) can be computed in time $O(|Q||t|)$.

7. SOLVING THE CORE PROBLEM

We now come to the last part of Theorem 1.1, where we tackle the core problem. Recall that the core problem is to find the nodes that satisfy a node test $\alpha.a$ `eq` $\beta.b$, where the program α only goes up in the tree, and the program β only goes down in the tree.

The idea in our solution is to use a factorization theorem for finite monoids. The original result is a theorem by Simon, which talks about the existence of "factorization forests", see [9]. Recently, Colcombet [4, 2] discovered that the factorizations not only exist, but can be calculated by a deterministic transducer (more precisely, the transducer calculates a weaker version of the factorizations). One application of this result is a fast string-matching algorithm, as described below. Fix a regular word language $L \subseteq \Sigma^*$, recognized by a finite monoid M. For any word $a_1 \cdots a_n \in \Sigma^*$ one can do a preprocessing stage in time $O(|M|n)$, such that later on, any query $a_i \cdots a_j \in L$? can be answered in constant time.

In Section 7.1, we define monoids, which are used in the factorization results. Then, in Section 7.2, we introduce the factorizations. In Section 7.3 how the results can applied in a tree. Finally, in the remaining part of Section 7, we use these results to solve the core problem.

7.1 Monoids

Instead of automata, one can use monoids to recognize regular languages; this is the framework in which the Simon theorem is stated. Recall that a *monoid* is a set M together with an associative composition operation and an identity element, denoted 1. We use letters m, n for monoid elements. For composition, we use multiplicative notation, so the composition of two elements $m, n \in M$ is denoted $m \cdot n$. Associativity means that for any $m_1, m_2, m_3 \in M$, we have $(m_1 \cdot m_2) \cdot m_3 = m_1 \cdot (m_2 \cdot m_3)$, and therefore one can write $m_1 \cdot m_2 \cdot m_3$ without ambiguity. The definition of the identity element is that it satisfies $1 \cdot m = m \cdot 1 = m$, for any $m \in M$. For instance, Σ^* is a monoid, with the empty word being the identity element. A language $L \subseteq \Sigma^*$ is *recognized* by a monoid M if there is a morphism $\alpha : \Sigma^* \to M$ such that membership $w \in L$ depends only on the value $\alpha(w)$. In the above, a *morphism* is a function that preserves composition and identity elements.

Monoids and automata are equivalent descriptions for regular languages, in the following sense: a language is recognized by a finite automaton if and only if it is recognized by a finite monoid. The left-to-right implication is proved by associating to each automaton $\mathcal{A}$ a monoid $M \subseteq P(Q^2)$ obtained by composing the transition relations in the automaton. In particular, this monoid is exponential in the size of the automaton, which is optimal (e.g. the language "the n-th letter is a" is only recognized by monoids with 2^n elements).

7.2 Forward Ramseyan splits

In this section we describe Simon factorizations and cite a result which says that they can be computed in linear time by a deterministic transducer. To be more precise, we do not use factorizations as in the Simon theorem, only a weaker variant called forward Ramseyan splits, following [4, 2].

Consider a word $w = a_1 \cdots a_n$. An *edge in* w is any number $i = 0, \ldots, n$, which is identified with the space between position i and $i + 1$. We will say that i is the *source* position of a_i, and $i + 1$ is the *target* position of a_i. Fix some $K \in \mathbb{N}$. A *split of height* K *on* w is a function σ that assigns to each edge in w a number from $1, \ldots, K$. These numbers are used to split the word positions into a nested factorization. First, we look at the positions i with $\sigma(i) = K$; these positions split the word into a number of subwords. Then, these subwords are recursively split by positions with values $\sigma(i) = K - 1, K - 2, \ldots, 1$. The figure below shows a word, a split, and the corresponding nested factorization.

We say two edges x, y are *neighbors* if $\sigma(x) = \sigma(y)$ and all edges between x, y are mapped by σ to at most $\sigma(x)$. The neighborhood relation is an equivalence relation. We say x, y are *visible* from each other if all edges between x, y are mapped by σ to values strictly smaller than both $\sigma(x)$ and $\sigma(y)$. In particular, two consecutive neighbors are visible, and at most $2K$ edges are visible from any given edge.

Now assume that we have fixed an alphabet Σ and a finite monoid M, together with a morphism $\phi : \Sigma^* \to M$. Given a word $w = a_1 \cdots a_n$, and edges $x < y$ in this 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 both sources and targets of edges between x, y inclusively. In particular, the word from x to x is the empty word. By $val_w(x, y)$ we denote the value assigned by ϕ to the word from x to y in w. If $x \leq y \leq z$ then

$$val_w(x, z) = val_w(x, y) \cdot val_w(y, z) \qquad \text{for } x \leq y \leq z \ .$$

We say that a split σ is *forward Ramseyan for* w *under* ϕ if for every every four pairwise neighboring edges $x < y, x' < y'$, we have:

$$val_w(x, y) = val_w(x, y) \cdot val_w(x', y') \ .$$

Note that in the above, there is no restriction on the order between the edge pair $x < y$ and the edge pair $x' < y'$. In

particular, both $val_w(x,y)$ and $val_w(x',y')$ must be idempotents, since we could have chosen $x = x'$ and $y = y'$.

The difference between forward Ramseyan splits and the original factorization forests of Simon is that the latter require a stronger condition:

$$val_w(x,y) = val_w(x,y) \cdot val_w(x,y) = val_w(x',y') \ .$$

The following result of Colcombet shows that a forward Ramseyan split can be computed by a deterministic transducer (in particular, in linear time):

Theorem 7.1 ([2])
Fix a monoid morphism $\phi : \Sigma^* \to M$. There is a deterministic transducer that outputs, for every word $w \in \Sigma^*$, a forward Ramseyan split on w.

In the above, the deterministic transducer is a finite deterministic automaton where each transition additionally outputs a number from $1, \ldots, K$. We assume without loss of generality that the splits used assign K to the first position in the word.

7.3 Splits in a tree

Fix a morphism $\phi : \Sigma^* \to M$ and a tree with tag names Σ. In this section and the next, we will be using forward Ramseyan splits to find the value of ϕ on downward paths in t.

We extend the mapping *val* to trees in the following way. For two edges $x \leq y$ in the tree t, the *word from x to y* is obtained by reading the tags on the path from x to y, excluding the tags in the source of x and the target of y. Unlike the string description used in earlier sections, we do not include the relationship `child, parent` between successive positions; implicitly we keep going down in the tree. The mapping *val* is defined analogously to the word case; we omit the subscript t since the tree t is fixed.

For an edge x, let σ_x be the split that is calculated by the transducer from Theorem 7.1, when reading the word from the root to x. Take some edge $y \leq x$. Since the transducer is deterministic, the value $\sigma_x(y)$ does not depend on the choice of x, and can without ambiguity be denoted by $\sigma(y)$. The values of $\sigma(x)$ on all edges x can be calculated by doing a top-down pass through the tree, we will be using these heavily later on. For instance, when we say that two edges $x \leq y$ are neighbors, we mean that they have the same value k under σ, and all edges in between have values at most k.

In the algorithm, we will do a depth-first search traversal through the tree, with the current edge denoted by x_{cur}. At each point, we will also store the following additional information, called the *snapshot for x_{cur}*. The snapshot assigns the following information to every ancestor edge x of x_{cur}:

A. A pointer to each edge $y \leq x_{cur}$ visible from x.

B. A pointer to highest ancestor (i.e. closest to the root) edge $y \leq x$ that is a neighbor of x, possibly $y = x$.

Together with each pointer to y, the snapshot stores the result of *val* for the corresponding word. Information about an edge is stored in the node record for the target node of the edge.

Note that it is important that the snapshot is evaluated relative to an edge x_{cur}. Otherwise, there would be no constant bound on the number of descendants of x that are visible from x. As it is, the pointers in A involve at most K ancestors and at most K descendants of x.

Below we show two key properties of the snapshots. First, as stated in Lemma 7.2, a snapshot allows constant time evaluation of $val(x,y)$ for any two edges $x \leq y \leq x_{cur}$. Second, as stated in Lemma 7.3, snapshots can be updated in constant time when moving x_{cur} to its parent or child.

Lemma 7.2 Let $x \leq y \leq x_{cur}$. Using the snapshot in x_{cur}, the value $val(x,y)$ can be evaluated in time $2K$.

Proof
In a first step, we consider the special case where $x \leq y$ are mutually visible (whether this case holds can be determined by examining the pointers in part A of the snapshot). In this case we already have $val(x,y)$ remembered in the snapshot.

In a second step, we consider the special case where $x \leq y$ are neighbors (whether this case holds can be determined by examining the pointer in part B of the snapshot). Let $z \leq x_{cur}$ be the closest neighbor of x that is its descendant, this node is stored in the snapshot under A, together with $val(x,z)$. If $z = y$ then we are done, otherwise we use the assumption on σ being forward Ramseyan:

$$val(x,y) = val(x,z) \cdot val(z,y) = val(x,z) \ .$$

To solve the general case, we trace the split, moving from x in the direction of y. Formally, the value $val(x,y)$ is calculated by reverse induction on $\sigma(x) + \sigma(y)$. If x, y are neighbors or are mutually visible then the procedure above can be used. Note that this includes the induction base, since if $\sigma(x) + \sigma(y) = 2K$, then x and y have to be neighbors. Otherwise, assume that $\sigma(x) \leq \sigma(y)$ (the other case is similar). Let $z \leq y$ be the closest descendant of x with $\sigma(z) > \sigma(x)$. To read z from the snapshot, we can move to the farthest ancestor of x being its neighbor, then move to its closest visible ancestor with σ greater that $\sigma(x)$, and then move to its closest visible descendant with σ greater that $\sigma(x)$. Moreover, let $z' \leq z$ be farthest descendant of x being its neighbor. We have

$$val(x,y) = val(x,z') \cdot val(z',z) \cdot val(z,y) \ .$$

The first and second value is calculated from the above special cases (as x and z' are neighbors and z' and z are visible). The third value is obtained by induction assumption (as $\sigma(z) + \sigma(y) > \sigma(x) + \sigma(y)$). □

The following simple lemma is given without proof:

Lemma 7.3 Let y be either a child or the parent of x_{cur}. The snapshot for y can be computed in constant time based on the snapshot for x_{cur}.

7.4 Using splits to solve the core problem

In this section we use the splits to solve the core problem in linear time.

Recall that in the core problem we want to determine the set of nodes satisfying $\alpha.a$ `eq` $\beta.b$, where α is a program that only goes up in the tree by using the `parent` axis, and β is a program that only goes down in the tree by using the `child` axis.

By compiling the programs α and β into first a regular word language, and then a monoid morphism, we may assume that these two programs are represented by a single morphism $\phi : \Sigma^* \to M$ in the following manner: whether or not a node pair (v,w) is selected by α (resp. β) only depends

on the value assigned by ϕ to the sequence of tag names on the path from v to w. Note that the monoid M may be exponential in the size of the expressions in α, β. To have a common morphism for both programs, a cartesian product can be used.

Therefore, the core problem will be solved if for any node u, we can calculate the set of pairs $(m, n) \in M^2$ such that for some nodes v, w with $v \leq u \leq w$, there is a common attribute value in $v.a$ and $w.b$, and the path from v to u (resp. from u to w) is mapped to m (resp. n) by ϕ. The rest of this section is devoted to showing how this information can be calculated in linear time. We will be using splits, so it will be more convenient to talk about edges, which motivates the definition of brackets below.

Let $x \leq y$ be two edges, and let m, n be two monoid elements. We say (x, y, m, n) is *bracket* if there are two edges x', y' with $x' \leq x \leq y \leq y'$ such that

- The target nodes of x', y' contain a common attribute value, respectively under attribute names a, b.
- $val(x', x) = m$ and $val(y, y') = n$.

The edges x, y are called the high and low edges of the bracket, and the monoid elements m, n are called the high and low monoid elements of the bracket. We say a bracket (x', y', m', n') *witnesses* a bracket (x, y, m, n) if

$$x' \leq x \leq y \leq y' \quad m' \cdot val(x', x) = m \quad val(y, y') \cdot n' = n \ .$$

A *visible* (resp. *neighbor*, *trivial*) bracket is one where the high and low edges x, y are mutually visible (resp. neighbors, equal). In the algorithms below, a bracket will be represented as a labeled (by m, n) pointer to the edge x, which is stored in the node record of the target node of the low edge y. Slightly ahead of time, we remark that for any given low edge y, we will store at most constantly many brackets.

The careful reader will notice that we come upon a little technical issue: the values computed in brackets ignore the letter in the node that contains the attribute value d under attribute name b. This is because the edge y' in the definition of a bracket has attribute b in its target, while $val(y, y')$ does not read the label in the target of y'. A quick fix for this problem is to consider only instances $\alpha.a$ `eq` $\beta.b$ of the core problem where the issue is moot:

Lemma 7.4 Without loss of generality, we may assume that the program β ignores the letter in the target node.

As long as our query is of the form in the lemma above, the core problem is solved once all the trivial brackets are computed. In Section 7.5 we will give an algorithm that computes certain brackets that witness all trivial brackets. Then, in Section 7.6 we show how these witness brackets can be used to calculate the trivial brackets.

7.5 Computing witness brackets

This section is devoted to showing the following result:

Proposition 7.5 One can compute in linear time a set V of visible brackets such that every trivial bracket has a witness in V.

A very special type of bracket is a bracket $(x, y, 1, 1)$ where the attribute value d appears under attribute name a in the target node of x, and under attribute name b in the target node of y. This type of bracket is called a *d-atom*. The name is so chosen because the atoms generate all other brackets, i.e. each bracket is witnessed by some atom.

However, in the algorithm it will be convenient to consider a different type of generator bracket, where the high and low edges are close to each other in some d-skeleton. We say that two edges x, y are *d-consecutive* if both target nodes are in the d-skeleton, and either $x = y$ or in the d-skeleton the target node of x is the parent of the target node of y. A bracket is called a *d-axiom* if it is witnessed by a d-atom and the high and low edges are d-consecutive. Note that not every d-atom is a d-axiom, since the high and low edges might not be d-consecutive. An *axiom* is a d-axiom for some d.

Lemma 7.6 For every attribute value d, the set of d-axiom brackets can be computed in time linear in the size of the d-skeleton.

Proof
As a preprocessing stage, we decorate each two d-consecutive edges x, y in the d-skeleton with $val(x, y)$. This can be done in linear time by using the constant-time procedure in Lemma 7.2 (we do the decoration simultaneously for all d-skeletons). Using this decoration, the d-axiom brackets can be calculated by doing a top-down, and then bottom-up pass through the d-skeleton. □

We are now ready to present the algorithm announced in Proposition 7.5. We will process the tree by moving an edge x_{cur} in depth-first search fashion (during this processing, we keep track of the snapshot in x_{cur}). At each moment, we will have computed two sets of brackets: a set V of visible brackets, and a set N of neighbor brackets. The invariant will be:

(*) Every trivial bracket witnessed by an already processed axiom bracket has a witness in $V \cup N$. Here, an already processed axiom bracket is an axiom bracket whose low edge has been a previous value of x_{cur}.

The set N is used to temporarily store information about the path leading up to the edge x_{cur}. Each neighbor bracket (x, y, m, n) in N will also satisfy the following two properties:

1. The low edge y is an ancestor of x_{cur}.
2. If z is a neighbor of x, y between these edges, then
$$m \cdot val(x, z) = m \ .$$

After all edges have been processed, the set N is empty, and therefore the proposition follows thanks to the invariant (*).

In the algorithm, for a given edge y and (m, n), there will be at most one edge x such that the bracket (x, y, m, n) belongs to N. The idea is that this edge x is the highest edge x about which we know that conditions 1,2 are satisfied. This allows us to define the following update operation $updateN(x, y, m, n)$, which is only defined when conditions 1,2 above are satisfied. Let z be such that (z, y, m, n) belongs to N before the update. If z is either undefined (N contained no appropriate bracket) or a descendant of x, then the update replaces (z, y, m, n) by (x, y, m, n), otherwise nothing happens.

We now proceed to describe how the sets V, N of brackets are modified in the algorithm. There are two possible steps: when we move x_{cur} down into a new edge, and when we move x_{cur} up. These are described in the following two sections.

7.5.1 Entering an edge

Here we describe what the algorithm does when it enters a new edge x_{cur}, i.e. goes down into a child edge. We will examine every axiom bracket whose low edge is x_{cur}. For each such axiom bracket, we do a constant number of modifications to N, V.

Fix an axiom bracket (x, x_{cur}, m, n) that is examined. Some trivial brackets between x and x_{cur} may get new witnesses thanks to this axiom bracket, so we need to update the sets N, V to satisfy the invariant (*).

Using the split, we can find a sequence of edges

$$x = x_1 < x_2 < \cdots < x_k = x_{cur} \qquad k \leq 2K \ ,$$

such that every two consecutive edges are either visible or neighbors. This sequence can be calculated as in the proof of Lemma 7.2. For each two consecutive edges x_i, x_{i+1}, we use Lemma 7.2 to calculate in constant time the values

$$m_i = m \cdot val(x, x_i) \qquad n_i = val(x_{i+1}, x_{cur}) \cdot n \ .$$

For each i, we consider the path from x_i to x_{i+1} separately, updating N, V to take care of the invariant (*) for trivial brackets in this path.

If x_i and x_{i+1} are mutually visible, we can simply add the bracket (x_i, x_{i+1}, m_i, n_i) to V. Otherwise, x_i and x_{i+1} are neighbors, which requires updating brackets in N. Let y be the closest neighbor of x_i with $x_i < y \leq x_{i+1}$, this edge is stored in the snapshot. From the definition of a forward Ramseyan split (multiplied by m_i) we know that regardless of the choice of a neighbor z of y, x_{i+1} between these edges, we have

$$m_i \cdot val(x_i, y) \cdot val(y, z) = m_i \cdot \text{value}(x_i, y) \ .$$

Therefore, y is a good candidate for

$$N(y, x_{i+1}, m_i \cdot val(x_i, y), n_{i+1}) \ ,$$

so we call the *updateN* operation for the bracket above.

7.5.2 Leaving an edge

When visiting an edge x_{cur} for the last time (before exiting from its subtree), we remove all brackets in N that have low edge x_{cur}, and transfer this information to a constant number of modifications to N, V.

For every m, n, we test if N contains some bracket of the form (z, x_{cur}, m, n). If $z = x_{cur}$, then there is no nontrivial information, and nothing has to be done. Otherwise, let $y < x_{cur}$ be the closest neighbor of x_{cur}, as stored in the snapshot. We add $(m \cdot val(z, y), n)$ to $P(y, x_{cur})$, and call $updateN(z, y, m, val(y, x_{cur}) \cdot n)$.

7.6 Computing trivial brackets

In this section, we complete the solution the core problem, by showing:

Proposition 7.7 The set of trivial brackets can be computed in linear time.

Proof
We calculate the set of trivial brackets in a bottom-up pass. For each edge x, we calculate the set

$$\{(y_0, m, val(x, y_1) \cdot n) : (y_0, y_1, m, n) \in V, x \leq y_1\} \ .$$

This set in an edge x can be calculated using the sets in the two edges below x, and the set of brackets from the set V (as calculated in Proposition 7.5) that have x as a low edge. Note that there may be at most K nodes y_0 in this set, so the set has constant size. Using the set above, we can easily determine the trivial brackets that contain x. □

8. CONCLUDING REMARKS

We have presented an algorithm for evaluating XPath queries that has linear time data complexity. A price for the linear time is a multiplicative constant that is exponential in the query.

A possible solution could be to trade the exponent in the query size for an extra logarithmic factor in the data complexity. By using a standard divide and conquer approach instead of forward Ramseyan splits, one can modify our algorithm so that it avoids monoids but runs in time $O(|\varphi|^2 \cdot |t| \cdot \log |t|)$.

The reason for the exponential blowup is that we use monoids to represent word languages, so that we can use the Ramseyan splits. It is conceivable, however, that a similar construction would work without going through monoids, thus avoiding the exponential blowup. We leave this as an open question: is there an evaluation algorithm that is linear in the document size and polynomial in the query size?

It is also possible that the ideas in this paper can be extended to larger fragments of XPath, e.g. where attribute values can be compared with respect to the lexicographic order on strings. We leave this as future work.

9. REFERENCES

[1] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. In *Principles of Database Systems*, pages 10 – 19, 2006.

[2] T. Colcombet. A combinatorial theorem for trees. In *ICALP'07*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[3] T. Colcombet. Factorisation forests for infinite words. In *FCT'07*, 2007.

[4] T. Colcombet. On Factorization Forests. Technical Report hal-00125047, Irisa Rennes, 2007.

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

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

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

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

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