Fachrichtung 6.2 – Informatik
Naturwissenschaftlich-Technische Fakultät I
Mathematik und Informatik
Universität des Saarlandes

On the Complexity of Tree Pattern Matching

Bachelor's Thesis

Michaela Götz

Angefertigt unter der Leitung von
Prof. Dr. Christoph Koch
Saarbrücken, Juni 2007

Verfasser: Michaela Götz

Erstgutachter: Prof. Dr. Christoph Koch

Zweitgutachter: Prof. Dr. Markus Bläser

Eidesstattliche Erklärung Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Michaela Götz, Pittsburgh PA, USA, Juni 2007

Einverständniserklärung Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.

Michaela Götz, Pittsburgh PA, USA, Juni 2007

Abstract

Tree pattern matching is a fundamental problem that has a wide range of applications in Web data management, XML processing, and selective data dissemination. This work analyzes the complexity of an important special case of the tree pattern matching problem, i.e., the problem of matching a tree pattern with exclusively transitive (descendant) edges. We prove that deciding whether there is such a tree mapping is LOGSPACE-complete, improving on the current LOGCFL upper bound.

Acknowledgments

I want to thank my advisor Christoph Koch, who introduced the fundamental question of the complexity of tree pattern matchings to me. Conversations with him gave me an idea of what research is like. Furthermore, I want thank Wim Martens, who taught me a lot about how to formulate proofs clearly. Productive discussions with him influenced this work.

Contents

1 Introduction

Tree patterns are a simple query language for tree-structured data. They are at the heart of several widely-used Web languages such as XPath and XQuery [2]. As a consequence, they form part of a number of typing mechanisms such as XML Schema, and of Web Programming languages. They have also been used as query languages in their own right, for example for expressing subscriptions in publish-subscribe systems [1, 3, 4, 9].

The general tree pattern matching problem considered in the literature is the problem of finding a mapping between two node-labeled trees. In this thesis we consider an important special case of the tree pattern matching problem that we call the *tree descendants matching*. It is the problem of deciding, for two node-labeled rooted trees, whether there is a mapping θ from the nodes of the first tree, the *query tree*, to the nodes of the second, the *data tree*, such that if node u is a child of v in the first tree, then $\theta(u)$ is a *descendant* of $\theta(v)$ in the second.

While this problem is of immediate practical relevance and a substantial number of papers have studied the complexity of tree pattern matching and efficient algorithms for this problem, the precise complexity of both the general tree pattern matching and the tree descendants matching are open; they are both known to be in LOGCFL and LOGSPACE-hard [7]. LOGCFL is the set of languages, that are recognizable in polynomial time by a non-deterministic logarithmic space bounded auxiliary pushdown automaton, see [11]. The LOGSPACE-hardness is a direct consequence of the fact that reachability in trees is LOGSPACE-complete [6]. The membership in LOGCFL can be immediately concluded from earlier results on the complexity of the acyclic conjunctive queries [8] and the positive navigational fragment of XPath [7], both much stronger languages.

Overview. The rest of this thesis is organized as follows. Section 2 sketches the XML background of the tree pattern matching. In Sect. 3 we summerize our contributions. Preliminaries can be found in Sect. 4. In Sect. 5 we present a time-efficient algorithm for the tree descendants matching, and conclude that this problem is in LOGDCFL. In Sect. 6 we introduce a space-efficient algorithm for the tree descendants matching, and conclude that this problem is in LOGSPACE.

2 Background: XML

XML is a markup language, that provides a hierarchical structure to organize elements with respect to their content. An XML document nests elements in user-defined tags (consisting of an opening tag and a closing tag). As an example Fig. 1(a) depicts an XML document. It reflects the structure of a library classifying books in different genres. Every XML document can be represented as a tree. The tree representation can be extracted as follows:

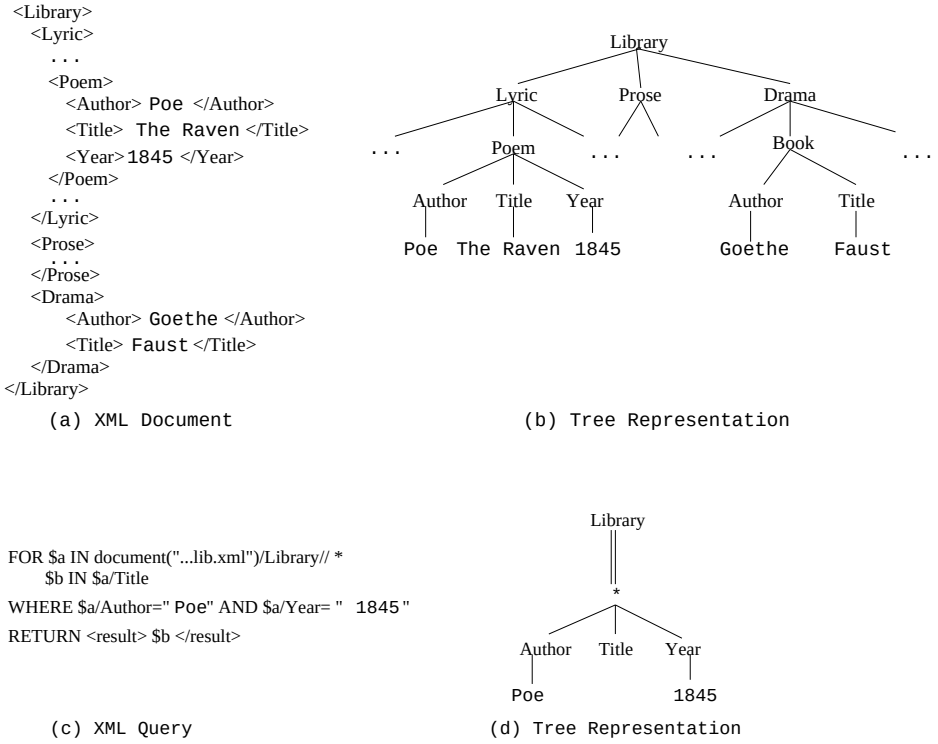


Figure 1: XML. An example XML document is depicted in (a) and its tree presentation is depicted in (b). To find out the opus of Edgar Allan Poe in 1845, we can formulate an XML query shown in (c), or we describe how to navigate in the tree representation of the XML document and formulate a tree pattern as shown in (d). First we need to go down from the "Library" node (we do not care how many levels) to some node (for instance "Book" or "Poem"), that has a child "Author" with a child "Poe" and another child "Year" with a child "1845". From that node we select the subtree rooted at "Title".

- Every appearance of a tag or an element x corresponds to a node in the tree.
- The number of tags x is contained in determines its depth d in the tree.
- The parent of the node corresponding to x is the unique node at depth $d-1$, which corresponds to a tag containing x.

The tree representation of the example XML document in Fig. 1(a) is shown in Fig. 1(b). The semi-structured format of XML enhances more freedom than rigid database tables. For example books can have variable attributes,

such as numerous authors, title, publisher, year, etc., but no book is forced to provide information to all attributes.

In order to address portions of the document, we make use of the structure and formulate tree patterns, see Fig. 1(d). A tree pattern tells us how to navigate from the top of the document tree to the desired portion of the document, for instance it tells us what nodes in the tree we need to pass and what kind of children they have to have and so on. In Fig. 1(c) the symbol x"/"y is used to indicate that y is the child of x, the symbol x"//"y is used to indicate that y is the descendant of x, and the symbol "$*$" is used as a wild card.

Not only the style of the queries but also the way to evaluate queries, is clearly different from traditional relational databases.

3 Contributions

This thesis deals with the evaluation of tree patterns over tree-shaped documents, e.g. with the problem of matching one tree into another. More specific, we will focus on the tree descendants matching: The problem of deciding, for two node-labeled rooted trees, whether there is a mapping θ from the nodes of the query tree to the nodes of the data tree such that if node u is a child of v in the first tree, then $\theta(u)$ is a *descendant* of $\theta(v)$ in the second. The input trees are given as nodes with pointers to their children.

We make the following contributions:

- We present a time-efficient LOGDCFL algorithm in Sec. 5, that solves the tree descendants matching problem on input trees D and Q in time $\mathcal{O}(|D| \cdot |Q|)$. LOGDCFL is the set of languages recognizable in polynomial time by a deterministic logarithmic space bounded auxiliary pushdown automaton, see [11].
- In Sec. 6 we present a space-efficient LOGSPACE algorithm for the tree descendants matching. Together with the fact that reachability is LOGSPACE complete, see [6], it follows that tree descendants matching is LOGSPACE complete.

4 Preliminaries

The first step in defining our problem is making the notion of a labeled tree formal. We introduce two equivalent definitions.

Definition 4.1 *A* tree *is defined over a finite alphabet* Σ *inductively as follows:*

(1) $a()$ *is a tree for some* $a \in \Sigma$*. We will call such a tree a* leaf*.*

(2) If $T_1, \ldots, T_n$ *are trees, then* $T = a(T_1, \ldots, T_n)$ *for* $n > 0$ *and* $a \in \Sigma$ *is a tree.*

Graphically, we think of $a(T_1 \cdots T_n)$ as a root node labeled a with the subtrees $T_1, \ldots, T_n$ attached to it. Before we make this more precise, we introduce some basic terms about graphs.

A *graph* G is a tuple (V, E), where V is a finite set and E is a set of unordered pairs of elements in V. We will refer to elements $v \in V$ as nodes and we will refer to elements $e \in E$ as edges. We might say $v \in G$ when meaning $G = (V, E)$ and $v \in V$.

A *path* from u to v in G is a sequence of nodes $v_1, \ldots, v_n$ such that $v_1 = u$, $v_n = v$ and consecutive nodes in the sequence are connected with an edge, e.g. $\{v_i, v_{i+1}\} \in E$ for $1 \leq i < n$. We call a graph *connected*, if there is a path between any two nodes in V. We call a graph *acyclic*, if there is no path in whose sequence a node occurs twice.

Now, we can describe a tree as a special graph.

Definition 4.2 *A* tree $T = (V, E)$ *over a finite alphabet* Σ *is an acyclic, connected graph, with a designated root node in* V *and a label function:* $V \to \Sigma$.

Clearly, there is a correspondence between the two definitions and we will use both definitions simultaneously in the following.

We say $u \in V$ is a *descendant* of $v \in V$, if all paths from u to root(T) go through v. We say $u \in V$ is a *child* of $v \in V$, if u is a descendant of v and there is an edge $\{u, v\} \in E$ connecting u and v.

We will use the notion *subtree*(v) to refer to a subtree of T with root v that consists of v and all its descendants and restricts E and the label function to this set of nodes. We talk about a *subtree of* T and mean a subtree(v) for some $v \in T$, $v \neq$ root(T).

We introduce the notion $\langle u \cdots v \rangle$ which denotes a path from u to v in T with a node sequence $v_1, \ldots, v_n$ of n distinct nodes, that are different from the root, e.g. $v_i \neq$ root(T) for $1 \leq i < n$. Note, that if such a path exists it is unique because a tree is acyclic.

There two natural ways to define the tree descendants matching. We will start with one based on mappings between the two trees and continue with a recursive definition. In both cases we are given a query tree and a data tree.

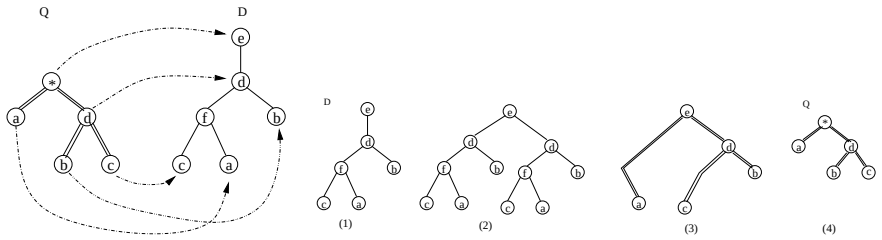
A *data tree* D is an ordinary tree over some alphabet Σ.

A *query tree* or *tree pattern* (with descendant edges) Q is a tree over the alphabet $\Sigma \uplus \{*\}$. The additional symbol "$*$" serves as a don't care symbol, which means that testing the equality of "$*$" and any other symbol yields true.

In the following definition we think of the input trees as acyclic graphs in the sense of Def. 4.2.

Definition 4.3 *The Tree Descendants Matching – A Functional View:*

Input *We are given a data tree* D *and a query tree* Q.



(a) Mapping θ. (b) Minor

Figure 2: The tree pattern matching problem with descendant edges. A label preserving mapping from a query tree into a data tree is shown in (a). It obeys the descendant requirement, e.g. the child relation in Q becomes a descendant relation under θ. A minor transformation is depicted in (b). It starts with the data tree D in (1) and duplicates a subtree in (2), then shrinks paths and omits nodes to obtain (3) and finally changes the order of siblings in (4) to obtain Q.

Question *Is there a mapping θ from the nodes of the tree pattern query Q to the nodes of the data tree such that*

Descendant Requirement *if node u is a child of v in Q, then $\theta(u)$ is a* descendant *of $\theta(v)$ in D and*

Label Agreement *for all $v \in D$: label(v) = label$(\theta(v))$*

If such a mapping exists, we say that D *matches* Q, denoted by $D \models_\theta Q$. See Fig. 2(a) for an example mapping.

Next we discuss a recursive definition, that builds upon the recursive Def. 4.1 of a tree.

Definition 4.4 *The Tree Descendants Matching – A Recursive View:*

Input *We are given a data tree D and query tree Q.*

Question *We say D matches Q, denoted by $D \models_r Q$, if and only if one of the following holds:*

(1) $Q = a()$ *or* $Q = *(), D = a(D_1 \cdots D_n)$, $n \in \mathbb{N}_0, a \in \Sigma$.

(2) $D = a(D_1 \cdots D_n)$, $n \in \mathbb{N}^+, a \in \Sigma$ *and* $\exists i \in \{1, \ldots, n\} : D_i \models_r Q$.

(3) $Q = a(Q_1 \cdots Q_m)$ *or* $Q = *(Q_1 \cdots Q_m), D = a(D_1 \cdots D_n)$, $n, m \in \mathbb{N}^+, a \in \Sigma$ *and for every* $k = 1, \ldots, m$ *there is an* $i_k \in \{1, \ldots, n\}$ *such that* $D_{i_k} \models_r Q_k$

Notice that in both definitions the ordering of children in our query trees does not matter. This corresponds to the well known semantics of XPath queries with descendant axes [5].

How do these definitions relate to each other? We will show that they are equivalent.

Proposition 4.5 *For all tree pattern queries Q and data trees D:*

$$D \models_r Q \text{ if and only if } D \models_\theta Q$$

.

Proof. We show both directions of this equivalence.
"⇒": Proof by induction on the size of D.

$|D| = 1$**:** $D \models_r Q$ implies that case (1) of the definition is true, because case (2) and case (3) require the data root to have children. Hence, $D = a()$, $Q = a()$ or $Q = *()$, for some $a \in \Sigma$. The only way to define a mapping in that case (matching the query root onto the data root) yields a label-preserving mapping. For the descendant requirement there is nothing to show.

$|D| > 1$**:** We consider three cases according to the recursive definition.

(1) If Q consists of one node whose label agrees with the label of D's root, we can define a label preserving mapping between the root nodes. For the descendant requirement there is nothing to show.

(2) If there is a subtree of D that matches Q with respect to our recursive definition, then the induction hypothesis yields the desired mapping.

(3) If the labels of the root nodes agree and every subtree of Q can be matched into a subtree of D, we combine the partial mappings that exist by induction hypothesis between the subtrees and furthermore map the query root onto the data root, to obtain θ. This mapping is clearly label preserving. Furthermore, any child u of the query root node is matched into data subtree attached to the data root - hence $\theta(u)$ is a descendant of θ(query root). For other parent-child nodes the descendant property is transferred from the partial mappings to θ.

"⇐": Proof by induction on the size of D.

$|D| = 1$**:** $D \models_\theta Q$ implies that the query root is matched onto the data node. The query tree cannot have any children due to the descendant requirement. Furthermore, the labels of the root nodes must agree. Hence, $D = a()$, $Q = a()$ or $Q = *()$, for some $a \in \Sigma$ and therefore $D \models_r Q$.

$|D| > 1$**:** We consider two cases.

- If the query root is matched onto some data node u, $u \neq \text{root}(D)$ the whole query tree is matched into the subtree(u) due to the descendant requirement. The claim follows from the induction hypothesis.
- If the query root node is matched onto root(D), then their labels agree. If the query only consists of one node, case (1) of the recursive definition is fulfilled and therefore $D \models_r Q$. Otherwise, we know that due to the descendant requirement the children $q_1 \dots q_n$ of the query root are matched onto data nodes different from the data root and that the subtree(q_i) is matched into subtree($\theta(q_i)$). In other words, the subtrees attached to the query root $Q_1 \cdots Q_m$ are mapped into subtrees attached to the data root, namely $D_{i_1} \cdots D_{i_m}$, and these mappings imply by induction hypothesis that $D_{i_k} \models_r Q_k$. Case (3) of the recursive definition is fulfilled and therefore $D \models_r Q$. ⊣

In the following we will use the notion $D \models Q$ instead of $D \models_r Q$ or $D \models_\theta Q$.

A third way to think about the tree descendants matching is in terms of minors. Loosely speaking, $D \models Q$ if we can modify D by duplicating subtrees, changing the order of siblings, shrinking paths to edges, and leaving out nodes, in order to obtain Q. See Fig. 2(b) for an example.

5 A Top-Down LOGDCFL Algorithm

The recursive Def. 4.4 of tree descendants matching gives rise to a recursive algorithm, that tests whether one of the cases (1), (2) or (3) are true. However, this procedure is not time efficient. In order to obtain an algorithm with a good runtime we make the following observation:

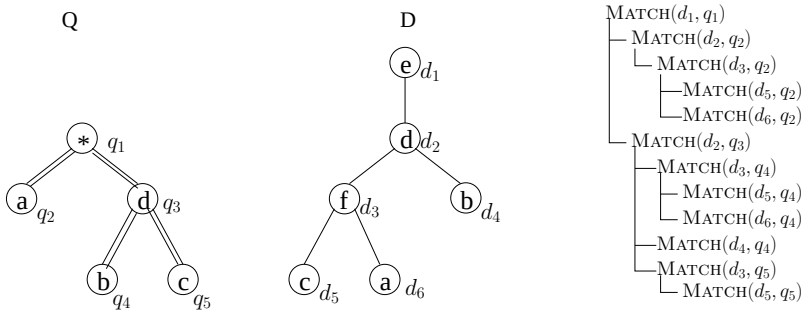
Lemma 5.1 *Let $D = a(T_1 \cdots T_n), n \in \mathbb{N}_0$ be a data tree and let Q be a query tree with a root labeled a or $*$.*
If case (2) of the recursive definition is true, it follows that case (1) or case (3) are also true, e.g.

$$
\begin{aligned}
(\exists i, 1 \leq i \leq n : D_i \models Q) \Rightarrow \; & Q = x(), x \in \{a, *\} \\
& \textit{or} \\
& Q = x(Q_1 \cdots Q_m), x \in \{a, *\}, n, m \in \mathbb{N}^+ \textit{ and} \\
& \forall k \in \{1, \dots, m\} \exists i_k \in \{1, \dots, n\} : D_{i_k} \models Q_k.
\end{aligned}
$$

Proof. Let $D = a(T_1 \cdots T_n), n \in \mathbb{N}_0$ be a data tree and let Q be a query tree with a root labeled a or $*$. Let i be a number in $\{1, \dots, m\}$ such that $D_i \models Q$. We consider two cases:

Algorithm 1 Tree pattern matching with descendant axes: Top-down algorithm MATCH

```
    MATCH (DNode d, QNode q)
 2: if d matches q then
        return ∀ child q_c of q ∃ child d_c of d: MATCH(d_c,q_c)
 4: else                                  ▷ q not matched yet, try d's children
        return ∃ child d_c of d: MATCH(d_c,q)
 6: end if
```



(a) Example. (b) Function Calls.

Figure 3: Illustrations of the tree descendants matching algorithm MATCH. In (a) the input trees are depicted. In (b) a possible tree of function calls is depicted.

- If $|Q| = 1$, then case (1) is a trivially true.
- If $|Q| > 1$, then $Q = a(Q_1 \cdots Q_m)$ or $Q = *(Q_1 \cdots Q_m)$, $m \in \mathbb{N}^+$. Theorem 4.5 becomes handy. With $D_i \models Q$ it follows that there is a label preserving mapping θ from Q to D_i, that obeys the descendant requirement (child relations in Q become descendants relations under θ). For each Q_k we restrict this mapping and obtain a mapping that is still label preserving and obeys the descendant requirement. Hence, $\forall k \in \{1, \ldots, m\} : D_i \models Q_k$, and case (3) is true. ⊣

Now, we are able to sketch the time efficient Algorithm 1. We expect the input trees to be given as pointer structures, e.g. nodes do not only contain their labels, but also pointers to their first child and to their next sibling. The two roots serve as input for MATCH.

5.1 Example

In Fig. 3(a) an example query tree and an example data tree are shown. These trees serve as input for MATCH. The resulting function calls are depicted in Fig. 3(b). Note that Algorithm 1 does not dictate how to evaluate

the logical expression in line 3 and 5. In Fig. 3(b) we follow a left-to-right ordering of the children and return the results in line 3 and 5 as soon as possible, e.g. we do not make function calls that cannot influence the return value.

First we call MATCH(d_1, q_1). Since the labels of these nodes agree, we have to test whether we can match both children of q_1 and their subtrees into the subtree(d_2). If this is the case, $D \models Q$, otherwise we know that there is no matching by Lemma 5.1.

So, we continue with calling MATCH(d_2, q_2). Their labels do not match, so we try to match q_2 further down in the subtree(d_2). Eventually, the call MATCH(d_6, q_2) successfully matches q_2 onto d_6.

It remains to match the second child of q_1 and its subtree into the subtree of d_2. So, we call MATCH(d_2, q_3). Both nodes are labeled with d, so we try to match q_3's children into subtrees attached to d_2. We start with q_4 and call MATCH(d_3, q_4). Their labels do not agree, so we try to match q_4 further down, but both function call L-MATCH(d_5, q_4) and L-MATCH(d_6, q_4) are not successful. However, we can still try to match q_4 onto the second child of d_2 and in fact d_4 matches q_4. We still have to match the second child of q_3 into a subtree attached to d_2. We start with L-MATCH(d_3, q_5) but their labels do not agree, so we try to match q_5 further down and call L-MATCH(d_5, q_3) and successfully match q_5.

So, we have matched all query nodes with respect to their labels and with respect to the descendant requirement. The corresponding mapping is shown in Fig 2(a).

5.2 Correctness of Match

Proposition 5.2 *Algorithm 1 is correct. That is, given the roots d and q of a data D and query tree Q,* MATCH(d, q) *decides whether $D \models Q$.*

Proof. Let d be a data node in D and let q be a query node in Q. By induction over the size of the subtree(d), denoted by n, we prove that MATCH(d, q) returns if and only if subtree$(d) \models$ subtree(q).

- $n = 1$: We have that subtree$(d) = a$ for some a. We return true, if and only if the query tree consists of one node and d matches this node. The correctness follows from the tree descendant matching definition, which says that for a data tree that consists only of one node labeled with some a we have subtree$(d) \models$ subtree(q) if and only if subtree$(q) = a$ or subtree$(q) = *$.

- $n > 1$: We consider two cases.

  - If d matches q, we return true if and only if, for every child q_c of q, there exists a child d_c of d such that MATCH(d_c, q_c) returns true. We consider two cases.

- If the query tree consists of only one node, we return true, which is correct, because in the definition of tree descendants matching case (1) is fulfilled, e.g. subtree$(q) = a()$ or subtree$(q) = *()$, subtree$(d) = a(D_1 \cdots D_n)$, $n \in \mathbb{N}_0, a \in \Sigma$, which implies that subtree$(d) \models$ subtree(q).
- If q has children, by the induction hypothesis, the recursive calls of MATCH(d_c, q_c) compute correctly whether subtree(d_c) matches subtree(q_c). We consider two cases.
  - If we return true, then case (3) in the tree descendants matching definition is fulfilled: subtree$(d) = a(D_1 \cdots D_n)$, subtree$(q) = x(Q_1 \cdots Q_m)$, $x \in \{a, *\}$, and for every $k = 1, \ldots, m$ there exists an $i_k \in \{1, \ldots, n\}$, such that $T_{i_k} \models Q_k$. Hence, subtree$(d) \models$ subtree(q).
  - If we return false, then there is a q_c such that for every d_c MATCH(d_c, q_c) is false, e.g. subtree$(d_c) \not\models$ subtree(q_c). We would also fail to match the whole query tree into a subtree of a child of d by Lemma 5.1. Again by the definition of tree pattern matchings none of the cases (1), (2), and (3) can be fulfilled and it is then correct to return false.

- If d does not match q, then by definition subtree$(d) \models$ subtree(q) if and only if case (3) is fulfilled, e.g. subtree$(d) = a(D_1 \cdots D_n)$ and there is a value i, $1 \leq i \leq n$ such that $D_i \models$ subtree(q). We check this. We return true, if and only if there is a child d_c of d such that subtree(q) can be matched into subtree(d_c). By the induction hypothesis, the recursive calls of MATCH(d_c, q) compute this correctly. So, our return value is correct. ⊣

5.3 Complexity of Match

5.3.1 Runtime

We assume that the input trees are given as pointer structures, e.g. each node contains its label and a pointer to the first child and a pointer to its next sibling. We further assume that given a pointer we can access the storage cell in constant time. We also assume that the comparison of two labels takes constant time.

Proposition 5.3 *Given a data tree D and a query tree Q as input of* MATCH. *The runtime of* MATCH *is in $O(|D| \cdot |Q|)$.*

Proof. To keep the analysis of the runtime of MATCH(d, q) concise, we introduce some notions. Let $n = |\text{subtree}(d)|$, $m = |\text{subtree}(q)|$, let $n_i = |subtree(i^{\text{th}}\text{child}(d))|$, $m_j = |subtree(j^{\text{th}}\text{child}(q))|$. Hence, $\sum_i n_i = n - 1$ and $\sum_j m_j = m - 1$. We denote the runtime of MATCH on input trees of

size n and m as function $T(n, m)$. We can observe from the algorithm that there is a $k \in \mathbb{N}$ such that:

$$T(n,m) \leq \begin{cases} k+m, & \text{if } n = 1 \\ k + \sum_j \sum_i T(n_i, m_j), & \text{if } n > 1 \text{ and } d \text{ matches } q \\ k + \sum_i T(n_i, m), & \text{o.w.} \end{cases}$$

We prove by induction on n that there is a constant $c \in \mathbb{N}$ such that $T(n, m) \leq c * n * m$.

$n = 1$ We choose c large enough.

$n > 1$ We consider two cases.

- d does not match q.

$$\begin{aligned} T(n,m) &\leq k + \sum_i T(n_i, m) \\ &\leq k + \sum_i cn_i m && \text{(I.H.)} \\ &= k + cm \sum_i n_i \\ &= k + cm(n-1) \\ &\leq cmn \end{aligned}$$

  The last inequality holds if we choose c large enough.

- d matches q.

$$\begin{aligned} T(n,m) &\leq k + \sum_j \sum_i T(n_i, m_j) \\ &\leq k + \sum_j \sum_i cn_i m_j && \text{(I.H.)} \\ &= k + c \sum_j m_j \cdot \sum_i n_i \\ &= k + c(m-1)(n-1) \\ &\leq cmn \end{aligned}$$

  Again, the last inequality holds if we choose c large enough. ⊣

5.3.2 Space Complexity.

Concerning the space complexity of MATCH, it is immediate from our implementation of the algorithm that it can be executed by a deterministic logarithmic space bounded auxiliary pushdown automaton, where the stack realizes the recursion. Moreover, the maximum recursion depth of Alg. 1 is $\mathcal{O}(\text{depth}(D))$.

By Prop. 5.3 the runtime is polynomial. We conclude:

Theorem 5.4 *The tree descendants matching is in LOGDCFL.*

While MATCH is quite time-efficient it is the question, whether we really need the recursion.

6 A LOGSPACE Algorithm

We argue how to transform Alg. 1 into a LOGSPACE algorithm that decides whether $D \models Q$. Intuitively, our LOGSPACE algorithm processes the data and query trees in a top-down left-to-right manner, just like Alg. 1, but the essential difference lies in a backtracking procedure. When, for example, Alg. 1 matches a query leaf q onto some data node d, then it uses the recursion stack to discover the data node onto which q's parent was matched in the data tree and tries to match q's next sibling in some subtree of that data node. Instead of using this recursion stack, the LOGSPACE algorithm enters a sub-procedure BACKTRACK(d, q) that calculates the highest ancestor d' of the data node d such that the path $\langle d' \cdots \text{root}(D)\rangle$ matches the path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$. As we will see later, d' is in fact the node onto which q's parent was matched. We present the LOGSPACE algorithm in Alg. 2.

For the ease of presentation, we assume the *left-to-right pre-order* ordering on nodes in trees. For a node u, we denote by $u + 1$ the next node in the depth first, left-to-right traversal.

6.1 Correctness of L-Match

We want to show that L-MATCH returns true on input D and Q if and only if $D \models Q$.

Proposition 6.1 *Alg. 2 is correct. That is, given the roots d and q of a data D and query tree Q,* L-MATCH(d, q) *decides whether $D \models Q$.*

We break the proof down into three parts; the soundness (Prop. 6.3), the completeness (Prop. 6.5), and the termination (Prop.6.7) and prove these parts in the following three subsections.

The fact that L-MATCH relies on the algorithm MATCH whose correctness is proven in Prop. 5.2 gives us a hint, why L-MATCH might also be correct. However, we will rigorously prove the correctness of L-MATCH.

To simplify the analysis, we imaginarily extend the algorithm by defining a mapping θ: Whenever the algorithm is executed we pull out paper and pen to write down θ. If the algorithm compares the labels of d and q in the function call L-MATCH(d, q) and they agree (in line 2, 6), we set $\theta(q) = d$ (and may overwrite older assignments).

If we ask questions about the algorithm, we can make use of our mapping θ and prove properties about it. These properties can refer to snapshots during the execution.

Algorithm 2 LOGSPACE decision procedure: Top-down algorithm L-MATCH.

```
    L-MATCH (DNode d, QNode q)
 2: if d matches q, and both d and q have children then        ▷ θ(q) = d
        return L-MATCH (d + 1, q + 1)
 4: else if d does not match q and d has a child then
        return L-MATCH (d + 1, q)
 6: else if d matches q and q is a leaf then                  ▷ θ(q) = d
        if q is final then
 8:         return true
        else
10:         d' ← BACKTRACK(d, q + 1)  ▷ node onto which q + 1.parent was
    matched
            return L-MATCH (d' + 1, q + 1)
12:     end if
    else            ▷ d is a leaf and (d does not match q or q is not a leaf)
14:     if d is final then
            return false
16:     end if
        q' ← q
18:     while q' has a parent do
            d' ← BACKTRACK(d, q')          ▷ node onto which q'.parent was
    matched
20:         if d' is an ancestor of d + 1 then
                return L-MATCH (d + 1, q')
22:         else q' ← q'.parent
            end if
24:     end while
        return L-MATCH (d + 1, q')
26: end if
```

6.1.1 Soundness

In this section we want to prove that whenever L-MATCH returns true on input D and Q, then $D \models Q$. In fact we prove a stronger claim: If L-MATCH returns true, then our mapping θ is a label preserving matching that obeys the descendant requirement. Hence, $D \models Q$.

In order to prove the soundness of L-MATCH, we first show the following lemma:

Lemma 6.2 *Let D be a data tree and Q be a query tree. Further, let* L-MATCH(d, q) *be a function call resulting from the initial procedure call* L-MATCH(*root*(D), *root*(Q)).

Then at the time when L-MATCH(d, q) *is called*

(1) θ restricted to query nodes less that q is a label preserving matching that obeys the descendant requirement,

(2) θ matches the path $\langle q.parent \cdots root(Q)\rangle$ into the path $\langle d.parent \cdots root(D)\rangle$ as high as possible,

(3) the path $\langle q \cdots root(Q)\rangle$ cannot be matched into the path $\langle d.parent \cdots root(D)\rangle$.

Proof. Proof by induction on the position k of L-MATCH(d, q) in the series of function calls resulting from the initial procedure call L-MATCH(root(D), root(Q)).

$k = 1$: For L-MATCH(root(D), root(Q)) there is nothing to show.

$k > 1$: Let the claim be true for the first k function calls. Let L-MATCH(d, q) be the k^{th} function call. We prove that it is also true for the $k + 1^{\text{th}}$ function call (if there is one). We consider four cases according to Alg. 2.

- If the labels of d and q agree and both nodes have children (line 2), the following function call is L-MATCH($d + 1$, $q + 1$), where $d + 1$ and $q + 1$ are the leftmost children of d and q, respectively. We know by induction hypothesis that θ restricted to query nodes less than q is a label preserving matching that obeys the descendant requirement. We enlarge this mapping by $\theta(q) = d$. This mapping is clearly label-preserving. We need to show that $\theta(q)$ is a descendant of $\theta(q.\text{parent})$. But this clear, since by induction hypothesis $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ is matched as high as possible into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$, which proves (1). Combining this with the fact that $\langle q \cdots \text{root}(Q)\rangle$ cannot be matched into $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ we conclude that $\langle q \cdots \text{root}(Q)\rangle$ is matched as high as possible into $\langle d \cdots \text{root}(D)\rangle$, which proves (2). With the descendant requirement it then follows that the path $\langle d \cdots \text{root}(D)\rangle$ cannot match the path $\langle q + 1 \cdots \text{root}(Q)\rangle$, which proves (3).
- If the labels of d and q do not agree and d has children (line 4), the following function call is L-MATCH($d + 1$, q), where $d + 1$ is the leftmost child of d. We do not enlarge θ in that case and all requirements (1), (2), and (3) follow from the induction hypothesis.
- If the labels of d and q agree and q is a leaf (line 6) and q is not final, we enlarge the mapping θ by $\theta(q) = d$. As in the first case of this proof we know by induction hypothesis that θ restricted to query nodes less than q is a label preserving matching that obeys the descendant requirement. The enlarged θ is still label

preserving and still obeys the descendant requirement, because due to the induction hypothesis $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ is matched into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$. Hence, (1) is true.

BACKTRACK$(d, q+1)$ calculates the highest ancestor d' of the data node d such that $\langle d' \cdots \text{root}(D)\rangle$ matches $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$. Why does d' exist? First, note that $q+1.\text{parent}$ is an ancestor of q due to the left-to-right pre-order ordering. Second, by induction hypothesis $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ can be matched into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$. Putting both facts together, the sub-path $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$ can still be matched into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$. Hence, d' exists and $d'+1$ is its the leftmost child.

The following function call is L-MATCH$(d'+1, q+1)$. By induction hypothesis the mapping θ matches the path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ as high as possible and therefore, θ also matches the sub-path $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$ as *high* as possible into that data path. Due to BACKTRACK, d' is an ancestor of d, such that $\langle d' \cdots \text{root}(D)\rangle$, matches $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$. Putting both facts together, we have (2): θ matches the sub-path $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$ as high as possible into the path $\langle d' \cdots \text{root}(D)\rangle$.

The fact that d' is the *highest* ancestor of d such that $\langle d' \cdots \text{root}(D)\rangle$ matches $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$ implies that $\langle d' \cdots \text{root}(D)\rangle$ does not match $\langle q+1 \cdots \text{root}(Q)\rangle$. Hence, (3) is proven.

- If d is a leaf and (the labels of d and q do not agree or q has children) (line 13) and d is not final, we know that we have to try to match q somewhere else. We do not enlarge θ (but we might restrict θ), so we still have a label-preserving mapping that obeys the descendant requirement and (1) is true. We consider two cases.
  - First, assume that the following function call is L-MATCH$(d+1, q')$ in line 25. Then q' has no parent, $q' = \text{root}(Q)$, and (2) is trivially true. To prove (3), e.g. to prove that root(Q) cannot be matched into the path $\langle d+1.\text{parent} \cdots \text{root}(D)\rangle$, we consider two cases.
    - If $q = q' = \text{root}(Q)$, by induction hypothesis root(Q) cannot be matched into $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ and therefore also not into the sub-path $\langle d+1.\text{parent} \cdots \text{root}(D)\rangle$.
    - If $q \neq q' = \text{root}(Q)$, then root$(Q)$ is an ancestor of q. By induction hypothesis the mapping θ matches root(Q) onto some ancestor of d and in fact root(Q) cannot be matched higher, e.g. the path $\langle \theta(\text{root}(Q)).\text{parent} \cdots \text{root}(D)\rangle$ does not match root(Q). Furthermore, the fact that we did not return a function call in line 21 implies

that then $\theta(\text{root}(Q))$ is a descendant of $d+1$.parent. Putting both facts together, we conclude that root(Q) cannot be matched into the path $\langle d{+}1.\text{parent} \cdots \text{root}(D)\rangle$.

– Now, assume that following function call is L-MATCH($d+1$, q') in line 21. BACKTRACK(d, q') has calculated the highest ancestor d' of the data node d such that $\langle d' \cdots \text{root}(D)\rangle$ matches $\langle q'.\text{parent} \cdots \text{root}(Q)\rangle$. Why does d' exist? First, note that q' has a parent (line 20) and that q' is an ancestor or self of q. Hence, q'.parent is an ancestor of q.parent. Further note, that by induction hypothesis the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ matches the path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ and therefore it also matches the sub-path $\langle q'.\text{parent} \cdots \text{root}(Q)\rangle$. It follows that d' exists and that $d'+1$ is its leftmost child.
We know that q' is the lowest ancestor or self of q such that d' is an ancestor of $d+1$ (observe the while loop). In fact, q' and its parent exist, because otherwise we would not make the next function call in line 21, but in line 25. Next, we will prove (2). By induction hypothesis the mapping θ matches the query path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$ and therefore also the sub-path $\langle q'.\text{parent} \cdots \text{root}(Q)\rangle$ as high as possible into the data path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$. It follows that the mapping θ matches the path $\langle q'.\text{parent} \cdots \text{root}(Q)\rangle$ as high as possible into the sub-path $\langle d' \cdots \text{root}(D)\rangle$ due to BACKTRACK. With the fact that d' is an ancestor of $d+1$ (line 21) it then follows that the mapping θ matches the path $\langle q'.\text{parent} \cdots \text{root}(Q)\rangle$ as high as possible into the path $\langle d+1.\text{parent} \cdots \text{root}(D)\rangle$, which proves (2).
In order to prove (3), e.g. to prove that the path $\langle d{+}1.\text{parent} \cdots \text{root}(D)\rangle$ cannot match the path $\langle q' \cdots \text{root}(Q)\rangle$, we consider two cases:
  * If $q = q'$, by induction hypothesis the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ cannot match the path $\langle q \cdots \text{root}(Q)\rangle$. Due to the left-to-right pre-order the path $\langle d+1.\text{parent} \cdots \text{root}(D)\rangle$ is a sub-path of $\langle d.\text{parent} \cdots \text{root}(D)\rangle$. The claim follows.
  * If $q \neq q'$, recall that q' is the lowest ancestor of q such that $\theta(q'.\text{parent})$ is an ancestor of $d+1$ (observe the while loop and recall that by induction hypothesis $d' = \theta(q'.\text{parent})$). It follows, that q' is matched somewhere on the path from d.parent to (but not including) $d{+}1$.parent. By Lem. 6.2 we cannot match the path $\langle q' \cdots \text{root}(Q)\rangle$ higher. Hence, the path $\langle d+1.\text{parent} \cdots \text{root}(D)\rangle$ does not match the path $\langle q' \cdots \text{root}(Q)\rangle$.

∘ Otherwise there does not follow a function call. ⊣

Proposition 6.3 *Alg. 2 is sound. That is, given a data D and query tree Q, if Alg.2 returns true, then $D \models Q$.*

Proof. If L-MATCH(d, q) returns true in line 8, we know that q is final (line 7) and that the labels of q and d agree (line 6). By Lem. 6.2 θ is a label-preserving mapping, which matches every node in $Q\backslash\{q\}$ with respect to the descendant requirement into D, such that q's parent is matched onto some ancestor of d. We enlarge the mapping by $\theta(q) = d$, and conclude that $D \models Q$. ⊣

6.1.2 Completeness

In this section we want to prove that whenever L-MATCH returns false on input D and Q, then $D \not\models Q$. In order to prove the completeness, we first show the following Lemma:

Lemma 6.4 *Let D be a data tree and let Q be a query tree. Further, let* L-MATCH(d, q) *be a function call resulting from the initial procedure call* L-MATCH(*root*(D), *root*(Q)).

Then, it holds for all left siblings $\hat{d}$ on the path from d to (but not including) θ(q.parent) or, in case q has no parent, the path from d to root(D) that

$$\textit{subtree}(\hat{d}) \not\models \textit{subtree}(q).$$

Proof. Note that by Lem. 6.2 we can talk about the image under θ of query nodes less than q. The proof is by induction on the position k of L-MATCH(d, q) in the series of function calls resulting from the initial procedure call L-MATCH(root(D), root(Q)).

$k = 1$: For L-MATCH(root(D), root(Q)) there is nothing to show, because there are no left siblings on the path $\langle \text{root}(D) \rangle$.

$k > 1$: Let the claim be true for the first k function calls. Let L-MATCH(d, q) be the k^{th} function call. We prove that it is also true for the $k+1^{\text{th}}$ function call (if there is one). We consider four cases according to Alg. 2.

- If the labels of d and q agree and both nodes have children (line 2), $\theta(q)$ is said to be d. The following function call is L-MATCH($d+1$, $q+1$), where $d+1$ and $q+1$ are the leftmost children of d and q, respectively.
  The path from $d+1$ to (but not including) $\theta(q+1.\text{parent})$ is the path from $d+1$ to (but not including) its parent d. Now, $d+1$ has no left sibling, so there is nothing to show.

- If the labels of d and q do not agree and d has children (line 4), the following function call is L-MATCH$(d+1,\ q)$, where $d+1$ is the leftmost child of d.

  By induction hypothesis we know that no there is no left sibling $\hat{d}$ on the path from d to (but not including) $\theta(q.\text{parent})$ or the path from d to root(D) in case q has no parent, such that subtree$(\hat{d})$ matches subtree(q). If we enlarge the path by d's leftmost child, the left siblings on the larger path are still the same. So, there is nothing left to show.

- If the labels of d and q agree and q is a leaf (line 6) and q is not final, $\theta(q)$ is said to be d.

  BACKTRACK$(d, q+1)$ calculates the highest ancestor d' of the data node d such that $\langle d' \cdots \text{root}(D)\rangle$ matches $\langle q+1.\text{parent} \cdots \text{root}(Q)\rangle$. By Lem. 6.2 we have that $\theta(q+1.\text{parent}) = d'$.

  The following function call is L-MATCH$(d'+1,\ q+1)$. The path from $d'+1$ to (but not including) $\theta(q+1.\text{parent})$ is the path from $d'+1$ to (but not including) its parent d'. Now, $d'+1$ has no left sibling, so there is nothing to show.

- If d is a leaf and (the labels of d and q do not agree or q has children) (line 13) and d is not final, then subtree(d) does not match subtree(q). We first show the following invariant we will need later:

  Invariant *Whenever the body of the while loop in line 18 is executed without returning a function call in line 21, it follows for the current q' that the subtree$(\theta(q'.\text{parent}))$ does not match the subtree$(q'.\text{parent})$.*

  Proof. We prove the claim by induction over the number of executions of the while body, denoted by l.

  $l = 1$**:** Here $q' = q$, q has a parent (line 18), and we know that:
  - the subtree(q) cannot be matched into the subtree(d) (line 13),
  - q cannot be matched into the path from d.parent to (but not including) $\theta(q.\text{parent})$ by Lem. 6.2,
  - there are no right siblings on the path from d to (but not including) $\theta(q.\text{parent})$ (otherwise we would have returned a function call in line 21),
  - the subtree(q) cannot be matched into the subtree$(\hat{d})$ for every left sibling $\hat{d}$ of the path from d to (but not including) $\theta(q.\text{parent})$ by the main induction hypothesis.

  Hence, no subtree of $\theta(q.\text{parent})$ matches subtree(q), which implies that the subtree$(\theta(q.\text{parent}))$ does not match the subtree$(q.\text{parent})$.

$l > 1$**:** Let the claim be true for the first l while loop executions. We prove, that it is also true for the $l + 1^{\text{th}}$ execution.
Let q' be the current query node of the $l + 1^{\text{th}}$ while loop execution. Here, $q' \neq q$ and q' has a parent (line 18). There must have been a function call L-MATCH$(q', \theta(q'))$. Note, that $\theta(q'.\text{parent})$ has not changed since then. Furthermore, there must have been a while loop execution with the child of q' on the path from q to q' as current node. We know that:

- the subtree(q') cannot be matched into the subtree$(\theta(q'))$, by the induction hypothesis,
- q' cannot be matched into the path from $\theta(q').\text{parent}$ to (but not including) $\theta(q'.\text{parent})$ by Lem. 6.2.
- there are no right siblings on the path from $\theta(q')$ to (but not including) $\theta(q'.\text{parent})$ (otherwise we would have returned a function call in line 21),
- the subtree(q') cannot be matched into the subtree$(\hat{d})$ for every left sibling $\hat{d}$ of the path from $\theta(q')$ to (but not including) $\theta(q'.\text{parent})$ by the main induction hypothesis.

Hence, no subtree of $\theta(q'.\text{parent})$ matches subtree(q'), which implies that the subtree$(\theta(q'.\text{parent}))$ does not match the subtree$(q'.\text{parent})$. ⊣

Now, we come back to the proof of the main induction. We consider two cases.

- First, assume that following function call is L-MATCH$(d+1, q')$ in line 25. Here q' is the query root. We need to show that there is no left sibling $\hat{d}$ on the path $\langle d+1 \cdots \text{root}(D)\rangle$, such that subtree$(\hat{d}) \models$ subtree(q'). We consider two cases:
  - If $q = q' = \text{root}(Q)$, by induction hypothesis the subtree(q) cannot be matched into the subtree$(\hat{d})$ of some left sibling $\hat{d}$ of the path $\langle d \cdots \text{root}(D)\rangle$. Since $d+1.\text{parent}$ is an ancestor of d, it is enough to show that the subtree(previous sibling$(d+1)$), which is the left sibling of $d+1$ that includes d, does not match the subtree(q). We know that
    - the subtree(q) cannot be matched into the subtree(d),
    - q cannot be matched into the path $\langle d.\text{parent} \cdots \text{root}(D)\rangle$ by Lem. 6.2,
    - there are no right siblings on the path from d to (but not including) the previous sibling$(d+1)$ due to the left-to-right pre-order,
    - the subtree(q) cannot be matched into the subtree$(\hat{d})$ for every left sibling $\hat{d}$ of the path from d to root(D) by induction hypothesis.

It follows, that we cannot match the subtree(q) into the subtree(previous sibling($d+1$)) at all.

* If $q \neq q' = \text{root}(Q)$, then there must have been a while loop execution with q''s child on the path from q to q' as current query node. Hence, $d+1$.parent is an ancestor of $\theta(q')$ (otherwise we would have returned a function call in line 21)
  Furthermore, by Lem. 6.2 there must have been a function call L-MATCH($q', \theta(q')$). It follows by induction hypothesis that the subtree(q') cannot be matched into the subtree($\hat{d}$) of some left sibling $\hat{d}$ of the path $\langle \theta(q') \cdots \text{root}(D) \rangle$.
  Putting both facts together, it is enough to show that the subtree(previous sibling($d+1$)), which is the left sibling of $d+1$ that includes d, does not match the subtree(q'). We know that:
  - the subtree(q') cannot be matched into the subtree($\theta(q')$) by the invariant,
  - q' cannot be matched into the path from $\theta(q')$.parent to root(D) by Lem. 6.2,
  - there are no right siblings on the path from $\theta(q')$ to (but not including) the previous sibling($d+1$), because $d+1$.parent is an ancestor of $\theta(q')$, which is an ancestor of d by Lem. 6.2,
  - the subtree(q') cannot be matched into the subtree($\hat{d}$) for every left sibling $\hat{d}$ of the path from $\theta(q')$ to root(D) by the induction hypothesis.

  It follows, that we cannot match the subtree(q') into the subtree(previous sibling($d+1$)) at all.

– Now, assume that the following function call is L-MATCH($d+1, q'$) in line 21. BACKTRACK(d, q') has calculated the highest ancestor d' of the data node d such that $\langle d' \cdots \text{root}(D) \rangle$ matches $\langle q'.\text{parent} \cdots \text{root}(Q) \rangle$. By Lem. 6.2 d' is equal to $\theta(q'.\text{parent})$.
  We know that q' is the lowest ancestor or self of q such that $\theta(q'.\text{parent})$ is an ancestor of $d+1$ (observe the while loop). It follows, that q' is matched somewhere on the path from d to (but not including) $d+1$.parent (for the case $q' \neq q$). No matter whether $q' = q$ or not, there was a function call L-MATCH(q', $\bar{d}$) for some $\bar{d}$ on the path from d to (but not including) $d+1$.parent. By induction hypothesis and Lem. 6.2 there is no left sibling $\hat{d}$ on the path from $\bar{d}$ to (but not including) $\theta(q'.\text{parent})$ such that subtree($\hat{d}$) matches subtree(q').
  Since $\bar{d}$ is in the subtree(previous sibling($d+1$)), we now only

need to show that subtree(previous sibling$(d+1)$) does not match the subtree(q').
We consider two cases:

- If $q = q'$, then we know that:
  - the subtree(q) cannot be matched into the subtree(d) see line 13,
  - q cannot be matched into the path from d.parent to (but not including) $\theta(q.\text{parent})$ by Lem. 6.2,
  - there are no right siblings on the path from d to (but not including) the previous sibling$(d+1)$ due to the left-to-right pre-order,
  - the subtree(q) cannot be matched into the subtree($\hat{d}$) for every left sibling $\hat{d}$ of the path from d to (but not including) $\theta(q.\text{parent})$ by induction hypothesis .

  It follows, that subtree(previous sibling$(d+1)$) does not match the subtree(q').
- If $q \neq q'$, there must have been a while loop execution with q''s child on the path from q to q' as current query node and there must have been a function call L-Match($\theta(q')$, q'). Note, that $\theta(q.\text{parent})$ has not changed since then. We know that:
  - the subtree(q') cannot be matched into the subtree($\theta(q')$) by the invariant,
  - q' cannot be matched into the path from $\theta(q')$.parent to (but not including) $\theta(q.\text{parent})$ by Lem. 6.2,
  - there are no right siblings on the path from $\theta(q')$ to (but not including) the previous sibling$(d+1)$, because $d+1$.parent is an ancestor of $\theta(q')$, which is an ancestor of d by Lem. 6.2,
  - the subtree(q') cannot be matched into the subtree($\hat{d}$) for every left sibling $\hat{d}$ of the path from $\theta(q')$ to (but not including) $\theta(q.\text{parent})$ by the induction hypothesis.

  It follows, that we cannot match the subtree(q') into the subtree(previous sibling$(d+1)$) at all.

- Otherwise there does not follow a function call. ⊣

Proposition 6.5 *Alg. 2 is complete. That is, given a data D and query tree Q, if Alg. 2 returns false, then $D \not\models Q$.*

Proof. We consider two cases.

- Let $|D| = 1$. L-Match(root(D), root(Q)) returns true, if root(Q) is a leaf with an appropriate label in line 8 and false otherwise in line 16, which proves the completeness for that case.

- Now, let $|D| > 1$. Assume L-MATCH returns false in line 16. Let d and q be the nodes, such that in the execution of L-MATCH(d, q) false was returned. Due to line 13, d is a leaf and (q has children or the labels of q and d do not agree). Due to line 14, d is the final node, which means that there are no right siblings on the path from d to the root.

  Consider a slight modification of the data tree: We attach to the root of the data tree a child on the right. Its value in the left-to-right pre-order is now $d + 1$, the highest value of nodes in the data tree. Call this tree D'. Observe from the algorithm, that replacing D by D' does not make any difference in the function calls before L-MATCH(d, q), because the algorithm traverses the data tree due to the left-to-right pre-order. However, in the function call L-MATCH(d, q) the algorithm would not return false anymore, instead it would call L-MATCH($d+1$, q') for some query node q'. By Lem. 6.4 we know that for every child d' of the data root in D, the subtree(d') cannot match the subtree(q'). We consider two cases.

  - Assume that q' has a parent. It is clear that if there was a matching from Q into D, we would be able to match the subtree(q') into some subtree of the data root. But we are not able to do this, so $D \not\models Q$.
  - Assume that q' is the query root. By Lem. 6.2 we know that we cannot match the query root into the path $\langle d + 1.\text{parent} \cdots \text{root}(D)\rangle$. Hence, the labels of the query root and the data root do not agree and if there was a matching from Q into D, we would be able to match the subtree(q') into some subtree of the data root. But we are not able to do this, so $D \not\models Q$. ⊣

6.1.3 Termination

Before we can conclude that L-MATCH is correct, we need to prove that the function call L-MATCH(root(D), root(Q)) terminates on every input D and Q. First, note that the while loop in line 18 terminates, because in every execution q' gets q'.parent and our input trees are of finite depth.

We now only need to argue that whenever we call L-MATCH(d, q) for some $d \in D$ and $q \in Q$, we have not called L-MATCH(d, q) before. We prove this by the following lemma, where we use Lem. 6.2 and refer to θ.

Lemma 6.6 *Let D be a data tree and Q be a query tree. Further, let* L-MATCH(d, q) *be a function call resulting from the initial procedure call* L-MATCH(*root*(D), *root*(Q)).

Then at the time when L-MATCH(d, q) *is called*

(1) for all query nodes $\bar{q} \geq q$, for all data nodes $\bar{d} \geq d$: we have not called L-MATCH($\bar{d}$, $\bar{q}$) *before.*

(2) for all right siblings $\hat{q}$ of nodes on the path $\langle q \cdots root(Q)\rangle$, for all nodes $\bar{q} \in subtree(\hat{q})$, for all data nodes $\bar{d} \in subtree(\theta(\hat{q}.parent))$: we have not called L-MATCH($\bar{d}$, $\bar{q}$) *before.*

(3) for all query nodes $\bar{q}$ on the path $\langle q.parent \cdots root(Q)\rangle$, for all data nodes $\bar{d} > \theta(\bar{q})$: we have not called L-MATCH($\bar{d}$, $\bar{q}$) *before.*

Proof. Proof by induction on the position k of L-MATCH(d, q) in the series of function calls resulting from the initial procedure call L-MATCH(root(D), root(Q)).

$k = 1$**:** For L-MATCH(root(D), root(Q)) there is nothing to show.

$k > 1$**:** Let the claim be true for the first k function calls. Let L-MATCH(d, q) be the k^{th} function call. We prove that it is also true for the $k + 1^{\text{th}}$ function call (if there is one). We consider four cases according to Alg. 2.

- If the following function call is L-MATCH($d+1$, $q+1$) (see line 3), then $\theta(q)$ gets d. Here, $d + 1$ is the leftmost child of d and $q + 1$ is the leftmost child of q. The induction hypothesis (case(1)) implies that for all $\bar{q} \geq q$, for all data nodes $\bar{d} \geq d$ we had not called L-MATCH($\bar{d}$, $\bar{q}$) before we called L-MATCH(d, q). In the meantime, we executed L-MATCH(d, q), so for all $\bar{q} \geq q + 1$, for all data nodes $\bar{d} \geq d + 1$: we have not called L-MATCH($\bar{d}$, $\bar{q}$), which proves (1).
  The induction hypothesis (case (2)) implies further that for all right siblings $\hat{q}$ of nodes on the path $\langle q \cdots \text{root}(Q)\rangle$, for all nodes $\bar{q} \in \text{subtree}(\hat{q})$, for all data nodes $\bar{d} \in \text{subtree}(\theta(\hat{q}.\text{parent}))$ we had not called L-MATCH($\bar{d}$, $\bar{q}$) before we called L-MATCH(d, q). Since $q + 1$ is the left-most child of q, we only need to show in order to prove (2), that for all right siblings $\hat{q}$ of $q + 1$, for all nodes $\bar{q} \in \text{subtree}(\hat{q})$, for all data nodes $\bar{d} \in \text{subtree}(\theta(\hat{q}.\text{parent}))$, which is equal to the subtree(d), we have not called L-MATCH($\bar{d}$, $\bar{q}$) before. But this follows from the induction hypothesis case (1), because data nodes in subtree(d) are greater or equal d and query nodes in subtrees of $q + 1$'s right siblings are greater than q.
  The induction hypothesis (case (3)) implies further that for all $\bar{q}$ on the path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$, for all data nodes $\bar{d} > \theta(\bar{q})$ we had not called L-MATCH($\bar{d}$, $\bar{q}$) before we called L-MATCH(d, q). In order to prove (3), we only need to show that for data nodes $\bar{d} > \theta(q) = d$ we have not called L-MATCH($\bar{d}$, q). But this is a consequence of the induction hypothesis (case (1)).
- If the following function call is L-MATCH($d + 1$, q) (see line 5), then $d + 1$ is the leftmost child of d. The induction hypothesis (case(1)) implies that (1) is true (as above). Since we only

called L-MATCH(d, q) in the meantime and did not change the mapping θ at all, (2) is a direct consequence of the induction hypothesis (case(2)) and (3) is a direct consequence of the induction hypothesis (case(3)).

- If the following function call is L-MATCH($d'+1$, $q+1$) (see line 11), then $\theta(q)$ gets d. Here, $q+1$ is a right sibling of a node on the path $\langle q \cdots \text{root}(Q)\rangle$ (due to the left-to-right pre-order and the fact that q is a leaf, see line 6) and $d'+1$ is the leftmost child of some ancestor of d. The induction hypothesis (case(1)) implies that for all $\bar{q} \geq q$, for all data nodes $\bar{d} \geq d$ we had not called L-MATCH($\bar{d}$, $\bar{q}$) before we called L-MATCH(d, q). In the meantime, we executed L-MATCH(d, q), so for all $\bar{q} \geq q+1$, for all data nodes $\bar{d} \geq d$ we have not called L-MATCH($\bar{d}$, $\bar{q}$). In order to prove (1) we still need to show that this is also true for all $\bar{q} \geq q+1$ and for all data nodes $\bar{d}$ with $d'+1 \leq \bar{d} < d$. We consider two cases:
  - If $\bar{q} \in \text{subtree}(q+1)$ we can make use of the induction hypothesis (case (2)). The query node $q+1$ is a right sibling of a node on the path $\langle q \cdots \text{root}(Q)\rangle$ and hence for all nodes $\bar{d} \in \text{subtree}(\theta(q+1.\text{parent}))$ we have not called L-MATCH($\bar{d}$, $\bar{q}$) before. This proves our case, because by Lem. 6.2 $\theta(q+1.\text{parent})$ is equal to d' and d' is an ancestor of d. Clearly, subtree(d') includes all nodes $\bar{d}$ with $d'+1 \leq \bar{d} < d$. Hence, for all $\bar{q} \in \text{subtree}(q+1)$ and for all $\bar{d}$ with $d'+1 \leq \bar{d} < d$ we have not called L-MATCH($\bar{d}$, $\bar{q}$) before.
  - If $\bar{q} \notin \text{subtree}(q+1)$ we can make use of the induction hypothesis (case (2)) again. Due to the left-to-right pre-order, $\bar{q}$ is then in a subtree of some right sibling $\hat{q}$ of a node on the path $\langle q+1 \cdots \text{root}(Q)\rangle$. This $\hat{q}$ is also a right sibling of a node on the path $\langle q \cdots \text{root}(Q)\rangle$. By induction hypothesis (case (2)) it follows that for all data nodes $\bar{d} \in \text{subtree}(\theta(\hat{q}.\text{parent}))$ we have not called L-MATCH($\bar{d}$, $\bar{q}$). This proves our case, because $\hat{q}$.parent is an ancestor or self of $q+1$.parent and by Lem. 6.2 $\theta(\hat{q}.\text{parent})$ is an ancestor or self of $\theta(q+1.\text{parent})$, which is equal to d'. Clearly, subtree(d') includes all nodes $\bar{d}$ with $d'+1 \leq \bar{d} < d$ and so does subtree($\theta(\hat{q}.\text{parent})$). Hence, for all $\bar{q} \notin \text{subtree}(q+1)$ and for all $\bar{d}$ with $d'+1 \leq \bar{d} < d$ we have not called L-MATCH($\bar{d}$, $\bar{q}$) before.

  As mentioned above right siblings of a node on the path $\langle q+1 \cdots \text{root}(Q)\rangle$ are also a right siblings of a node on the path $\langle q \cdots \text{root}(Q)\rangle$. Hence, (2) immediately follows from the induction hypothesis (case(2)).

  Since $q+1$.parent is an ancestor of q, the path $\langle q+1.\text{parent} \cdots$

root(Q)$\rangle$ is a sub-path of the path $\langle q$.parent $\cdots$ root(Q)$\rangle$. Hence, (3) immediately follows from the induction hypothesis (case(3)).

- If the following function call is L-MATCH($d+1$, q') (see line 21 or line 25), then $d+1$ is a right sibling of a node on the path $\langle d$ $\cdots$ root(D)$\rangle$ (due to the left-to-right pre-order and the fact that d is a leaf, see line 13) and q' is an ancestor or self of q.

  The induction hypothesis (case(1)) implies that for all $\bar{q} \geq q$, for all data nodes $\bar{d} \geq d$ we had not called L-MATCH($\bar{d}$, $\bar{q}$) before we called L-MATCH(d, q). In the meantime, we only executed L-MATCH(d, q), so for all $\bar{q} \geq q$, for all data nodes $\bar{d} \geq d+1$: we have not called L-MATCH($\bar{d}$, $\bar{q}$) before.

  In order to prove (1) we still need to show this is also true for all query nodes $\bar{q}$ with $q' \leq \bar{q} < q$ and for all data nodes $\bar{d} \geq d+1$. Note that such a node $\bar{q}$ is a node on the path $\langle q$.parent $\cdots$ $q'\rangle$ (due to the left-to-right pre-order and the fact that q' is an ancestor or self of q). The induction hypothesis (case (3)) implies then that for all data nodes $\bar{d} > \theta(\bar{q})$ we have not called L-MATCH($\bar{d}$, $\bar{q}$). This proves our case, because by Lem. 6.2 $\theta(\bar{q})$ is an ancestor of d and hence less than d. We conclude that for all nodes $\bar{q}$ with $q' \leq \bar{q} < q$, for all nodes $\bar{d} \geq d+1$ we have not called L-MATCH($\bar{d}$, $\bar{q}$) before.

  Since q' is an ancestor or self of q, the path $\langle q' \cdots$ root(Q)$\rangle$ is a sub-path of the path $\langle q \cdots$ root(Q)$\rangle$. Hence, (2) immediately follows from the induction hypothesis case(2) and (3) follows from the induction hypothesis case(3). ⊣

As a consequence of Lem. 6.6 case (3) and the fact that the while loop line 18 always terminates we can state the following proposition.

Proposition 6.7 *Alg. 2 terminates. Let d and q be the roots of a data D and query tree Q,* L-MATCH(d, q) *terminates.*

6.2 Space Complexity of L-Match

The analysis of the complexity of L-MATCH is done with respect to the Turing Machine model. We assume that the input trees are coded in a reasonable way on the input tape for instance as a list of edges.

To see that this L-MATCH only needs logarithmic space, observe from the algorithm that its recursion is linear and that the function calls are in return statements. So, we do not need the recursion stack. Furthermore, we only use a constant number of variables for data nodes and query nodes that only need logarithmic space. It remains to argue, why BACKTRACK only needs logarithmic space. BACKTRACK(d, q) calculates the highest ancestor d' of the data node d such that the path $\langle d' \cdots$ root(D)$\rangle$ matches the path $\langle q$.parent $\cdots$ root(Q)$\rangle$. The only difficulty lies in the fact that we cannot

store both paths. So, we start at the root nodes and compare their labels. If they match, we determine their children that lie on the paths (this can be done by scanning the input tape at most depth(D) times). Otherwise, we determine only the child of the data node. We proceed recursively. Once we matched the whole path $\langle q.\text{parent} \cdots \text{root}(Q)\rangle$, we return the data node, onto which we matched q.parent.

Note, that the left-to-right pre-order of the nodes is not a requirement on the input trees. We used this ordering to ease the representation of the algorithm. Given a node its successor in this ordering can clearly be computed in logarithmic space.

6.3 The Complexity of tree descendants matching

As mentioned above, L-MATCH is in LOGSPACE. Putting this together with the fact that reachability in trees is LOGSPACE-complete [6] given the tree as a pointer structure, we obtain the main result of this thesis:

Theorem 6.8 *The problem of tree descendants matching is LOGSPACE-complete.*

The representation of the trees is of importance, because reachability in trees that are represented as a list of symbols and brackets following the Def. 4.1 of a tree is in TC^0 [10].

References

[1] Mehmet Altinel and Mike Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'2000)*, pages 53–64, Cairo, Egypt, 2000.

[2] Nicolas Bruno, Divesh Srivastava, and Nick Koudas. "Holistic Twig Joins: Optimal XML Pattern Matching". In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, Madison, Wisconsin, June 2002.

[3] Chee Yong Chan, Wenfei Fan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. "Tree Pattern Aggregation for Scalable XML Data Dissemination". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[4] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, San Jose, California, USA, February 26-March 1, 2002, 2000.

[5] J. Clark and S. DeRose. XML Path Language (XPath). Technical report, World Wide Web Consortium, November 1999. http://www.w3.org/TR/xpath.

[6] Stephen A. Cook and P. McKenzie. "Problems Complete for Deterministic Logarithmic Space". *J. Algorithms*, **8**:385–394, 1987.

[7] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the ACM*, 52(2):284–335, 2005.

[8] Georg Gottlob, Nicola Leone, and Francesco Scarcello. "The Complexity of Acyclic Conjunctive Queries". *Journal of the ACM*, **48**(1):431–498, 2001.

[9] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT)*, 2003.

[10] B. Jenner, K.-J. Lange, and P. McKenzie. Tree isomorphism and some other complete problems for deterministic logspace. Technical Report 1059, 1997.

[11] I.H. Sudborough. "Time and Tape Bounded Auxiliary Pushdown Automata". In *Mathematical Foundations of Computer Science (MFCS'77)*, pages 493–503. Springer Verlag, LNCS 53, 1977.