Information Systems 34 (2009) 602–623

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Efficient algorithms for descendant-only tree pattern queries ☆

Michaela Götz [a], Christoph Koch [a], Wim Martens [b,*]

[a] Cornell University, Ithaca, NY 14853, United States
[b] Technical University of Dortmund, Germany

ARTICLE INFO

Keywords:
XML
XPath
Query processing
Tree pattern queries
Complexity

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. In this paper we develop efficient algorithms for the tree homeomorphism problem, i.e., the problem of matching a tree pattern with exclusively transitive (descendant) edges. We first prove that deciding whether there is a tree homeomorphism is LOGSPACE-complete, improving on the current LOGCFL upper bound. Furthermore, we develop a practical algorithm for the tree homeomorphism decision problem that is both space- and time-efficient. The algorithm is in LOGDCFL and space consumption is strongly bounded, while the running time is linear in the size of the data tree. This algorithm immediately generalizes to the problem of matching the tree pattern against all subtrees of the data tree, preserving the mentioned efficiency properties.

© 2009 Elsevier B.V. All rights reserved.

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 [4]. 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 [1,5,6,14].

The general tree pattern matching problem considered in the literature is the problem of finding a mapping between two node-labeled trees which is, in a sense, a cross of a subtree homomorphism and a homeomorphism. In this article we consider a clean and important special case of the tree pattern embedding problem that we call the *tree homeomorphism problem*. The question we consider is whether there is a mapping θ from the nodes of the first tree, the *tree pattern* or *query*, to the nodes of the second tree, the *data tree*, such that if node y is a child of x in the first tree, then $\theta(y)$ is a *descendant* of $\theta(x)$ in the second tree. We also consider the *tree homeomorphism matching problem*: finding *all* nodes v of the data tree such that there is such a tree homeomorphism with v the image of the root node of the pattern tree. This problem of selecting all nodes whose subtrees match the tree pattern has frequent application in XML and Web query processing [1,10].

While this problem is of immediate practical relevance and a substantial number of papers have studied complexity and efficient algorithms for tree pattern matching, the precise complexity of both the general tree pattern matching problem and the tree homeomorphism problem are open; they are both known to be in LOGCFL and LOGSPACE-hard [11]. The former can be immediately concluded from earlier results on the complexity of the acyclic conjunctive queries [12] and the positive navigational fragment of XPath [11], both much stronger

☆ The present paper is the full version of Ref. [13], which appeared in the Symposium on Data Base Programming Languages 2007.

* Corresponding author.
E-mail addresses: goetz@cs.cornell.edu (M. Götz), koch@cs.cornell.edu (C. Koch), wim.martens@udo.edu (W. Martens).

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.03.010

Table 1
Time and space consumption for algorithms solving the tree homeomorphism matching problem.

	Time	Space	Streaming
Yannakakis (1981) [20]	$O(\|Q\| \cdot \|D\| \cdot depth(D))$	$O(depth(Q) \cdot \|D\|)$	No
Gottlob et al. (2002) [10]	$O(\|Q\| \cdot \|D\|)$	$O(\|Q\| \cdot \|D\|)$	No
Olteanu et al. 2004 [17]	$O(\|Q\| \cdot \|D\| \cdot depth(D))$	$O(\|Q\| \cdot depth(D) + \|D\|)$	Yes
Bar-Yossef et al. (2005) [3]	$O(\|Q\| \cdot \|D\|)$	$O(\|Q\| \log \|D\| + cand_D)$	Yes
Ramanan (2005) [18]	$O((\|Q\| + depth(D)) \cdot \|D\|)$	$O(\|Q\| \cdot depth(D) + cand_D)$	Yes
Our bottom-up algorithm	$O(\|Q\| \cdot \|D\| \cdot depth(\|Q\|))$	$O(depth(D) \cdot branch(D))$	No
Our LOGSPACE algorithm	$poly(\|Q\| + \|D\|)$	$O(\log(\|Q\| + \|D\|))$	No

Here $depth(\cdot)$ and $branch(\cdot)$ denote the depth and maximal branching factor of a tree, respectively.

languages. The latter is a direct consequence of the fact that reachability in trees is LOGSPACE-complete [8].

Much work has been dedicated to developing efficient algorithms for finding matches of tree patterns and tree homeomorphisms. Certain algorithms aim at processing the data tree as a stream (i.e., in a single scan) [2,3,5,6,9,14,16–18]. For this case a number of lower bound results have been obtained using mechanisms from communication complexity [2,3,15]. It is basically known that streaming algorithms for even simple tree patterns consume space proportional to the size of the data tree in the worst case. Table 1 lists algorithms for the tree homeomorphism matching problem together with bounds on their running time and space consumption. Here D is the data tree and Q is the tree pattern. We assume a random-access machine model with unit cost for reading and writing integers. Some of the algorithms presented support generalizations of the tree homeomorphism problem but where a better bound is known for the tree homeomorphism problem, it is shown. Some of the streaming algorithms [3,18] use a notion of candidate node sets $cand_D$ which depends on the algorithm and which can be of size close to $|D|$ in the worst case. The algorithm of [3] makes the assumption of so-called non-recursive data trees, in which no two nodes such that one is a descendant of the other may have the same label. Finally, streaming algorithms such as [16] focus on being able to process SAX-events in constant time, at the cost of an exponential preprocessing step.

In this article we study the tree homeomorphism (matching) problem. We establish a tight complexity characterization and develop an algorithm for the node-selection problem (shown at the bottom of Table 1) that is both time- and space-efficient. In detail, the technical contributions of this article are as follows:

- We first develop a top-down algorithm for the tree homeomorphism problem that is in LOGDCFL.[1]
- From this we develop a proof that the problem is LOGSPACE-complete, improving on the LOGCFL upper bound from [11].
- As our main result we develop a bottom-up LOGDCFL algorithm for computing all solutions of the tree homeomorphism problem which is both time- and space-efficient. This is a rather difficult algorithm and the correctness proof is involved. The algorithm runs in time $O(|D| \cdot |Q| \cdot depth(Q))$ and employs a stack of depth bounded by $\mathcal{O}(depth(D)branch(D))$.
  The algorithm may be of relevance in practical implementations. Indeed, in most Web or XML applications, the data tree is *much* larger than the tree pattern yet its depth is rather small. It can be observed that ours is the only algorithm in Table 1—and to the best of our knowledge, in existence—that can guarantee a space bound that does not contain the size, but only depth and branching factor, of the data tree as a term. At the same time the algorithm admits a good time bound.
  Furthermore, the algorithm is of relevance in theory as well. It is a first step in classifying the complexity of positive Core XPath with only child and descendant axes, which is probably the most widely used XPath fragment in practice. Its precise complexity, however, is unknown.
- In some applications (e.g., for certain XML data trees), a few nodes can have a very large number of children. Our algorithm can be made to run in space $O(depth(D)\log(branch(D)))$ with the same time bound if we assume the data tree to be in a ranked form that can be obtained by a LOGSPACE linear-time preprocessing algorithm. Given that ours is an offline algorithm it means little loss of generality to assume that data trees are kept in a database in this preprocessed form.

The article presents these result basically in the order given here.

2. Definitions

By $\mathbb{N}$ we denote the set of strictly positive integers. By Σ we always denote a fixed but infinite set of labels. The trees we consider are rooted, ordered, finite, labeled, unranked trees, which are directed from the root downwards. That is, we consider trees with a finite number of nodes and in which nodes can have arbitrarily many children. A *Σ-tree t*

[1] For our purposes, it is enough to know that LOGDCFL is characterized by deterministic LOGSPACE bounded pushdown automata which run in polynomial time [19].

(or tree t) is a relational structure over a finite number of unary labeling relations $a(\cdot)$, where each $a \in \Sigma$, and binary relations Child $(\cdot,\cdot)$ and NextSibling $(\cdot,\cdot)$. Here, $a(u)$ expresses that u is a node with label a, and Child (u,v) (respectively, NextSibling (u,v)) expresses that v is a child (respectively, next sibling) of u. We assume that each node in a tree bears precisely one label, i.e., for each u, there is precisely one $a \in \Sigma$ such that $a(u)$ holds in t.

By ε we denote the empty tree. By $a(T_1 \cdots T_n)$ we denote the tree in which the root bears the label a and has n non-empty subtrees $T_1 \cdots T_n$, from left to right. If the a-labeled root has no children, we write a rather than $a()$. By root (t) we denote the root node of t.

By $<_{\text{pre}}$ and $<_{\text{post}}$ we denote the depth-first left-to-right pre-ordering, respectively, left-to-right post-ordering in trees. That is, if u is a node with children $u_1,\ldots,u_n$ from left to right, then we have that $u <_{\text{pre}} u_1 <_{\text{pre}} \cdots <_{\text{pre}} u_n$ and $u_1 <_{\text{post}} \cdots <_{\text{post}} u_n <_{\text{post}} u$. Furthermore, u_1 is the successor of u in $<_{\text{pre}}$, i.e., there does not exists a v such that $u <_{\text{pre}} v <_{\text{pre}} u_1$. Similarly, u is the successor of u_n in the post-ordering. In Section 3, we will assume the $<_{\text{pre}}$ ordering on nodes, and in Section 4, we will assume the $<_{\text{post}}$ ordering.

A *Σ-hedge* H (or hedge H) is a finite ordered sequence $T_1 \cdots T_n$ of trees. When we write a hedge as $T_1 \cdots T_n$, we tacitly assume that every T_i is a non-empty tree. In the hedge $T_1 \cdots T_n$, we assume that $u_i <_{\text{pre}} u_{i+1}$ and $u_i <_{\text{post}} u_{i+1}$ holds for each $i = 1,\ldots,n-1$, where u_i and u_{i+1} are the roots of T_i and T_{i+1}, respectively. Notice that we do not necessarily assume a sibling relation between the roots of T_i and T_{i+1}.

In the sequel, we will slightly abuse terminology and use the term "tree" to also refer to a hedge consisting of one tree, and we use the term "hedge" to also refer to the union of trees and hedges. We assume familiarity with terms such as *child*, *parent*, *descendant*, *ancestor*, *leaf*, *root*, *first child*, *last child*, *first sibling*, *previous sibling*, *last sibling*, and *next sibling*.

For a hedge H, we denote by Nodes (H) the *set of nodes* of H. By $|H|$, we denote the number of nodes of H. Let $H = T_1 \cdots T_n$ with $n \geqslant 1$. The *label* of node u in the tree or hedge H is sometimes also denoted by $\text{lab}^H(u)$. The *depth* of a node u in H, denoted by $\text{depth}^H(u)$, is 1 when u is the root of some T_i and $1 + \text{depth}(v)$ when u is a child of v. The *height* of a node u in hedge H, denoted by $\text{height}^H(u)$, is 1 when u is a leaf and $\max(\text{height}^H(u_1),\ldots,\text{height}^H(u_k)) + 1$ when u has $k > 0$ children $u_1,\ldots,u_k$. By $\text{subtree}^H(u)$, we denote the subtree of H rooted at node u. By $\text{parent}^H(u)$, we denote the parent of u in H, if it exists. In the remainder of the article, we usually leave H implicit when H is clear from the context.

2.1. The tree homeomorphism problem

A *tree pattern query* (with descendant edges) Q is an (unranked) tree over the alphabet $\Sigma \uplus \{*\}$. That is, we assume that the special label $*$ does not appear in Σ. In the following, we use the terms *data tree* or *data hedge* to refer to ordinary Σ-trees and Σ-hedges.

Definition 1 (*Tree pattern matching*). Given a data hedge H, a node $u \in \text{Nodes}(H)$, and a tree pattern query Q, we say that *H matches Q at node u*, denoted by $H \vDash^u Q$, if there exists a mapping $h : \text{Nodes}(Q) \to \text{Nodes}(H)$ such that,

- if $\text{lab}^Q(v) = a$ for some $a \in \Sigma$, then $\text{lab}^H(h(v)) = a$;
- if Child (v_1, v_2) holds in Q, then $h(v_1)$ is an ancestor of $h(v_2)$ in H; and
- $u = h(\text{root}(Q))$.

If the above mapping h exists, we call h a *tree pattern matching*.

Notice that the ordering of children in our tree pattern queries does not matter, and that the label $*$ is a wildcard label for the query. This corresponds to the well known semantics of XPath queries with descendant axis [7]. In the following, we abbreviate by $H \vDash Q$ that $H \vDash^u Q$ for some $u \in \text{Nodes}(H)$. Alternatively, we say that H *matches* Q.

In this article, we are interested in the following problems.

Definition 2 (*Tree homeomorphism (matching) problem*). Given a data tree T and a tree pattern query Q, the *tree homeomorphism problem* consists of deciding whether $T \vDash Q$. Furthermore, we are interested in *computing all answers* for the tree homeomorphism problem, that is, computing all nodes $u \in \text{Nodes}(T)$ such that $T \vDash^u Q$. We refer to the latter problem as *tree homeomorphism matching problem*.

We assume that trees are stored on tape as a set of records; one for each node. Each record contains a pointer to its first child, last child, parent, previous sibling, and next sibling.

In the remainder of the article, we assume a fixed data tree D and a fixed query tree Q for ease of presentation. We will refer to nodes of D and Q as *data nodes* and *query nodes*, respectively.

3. A top-down algorithm

This section provides a simple top-down algorithm for the tree homeomorphism matching problem. The core of this top-down algorithm lies in a simple procedure that decides, given a data node d and a query node q, whether $\text{subtree}(d) \vDash \text{subtree}(q)$.

3.1. A top-down LOGDCFL algorithm

The procedure MATCH, illustrated in Algorithm 1 tests whether $\text{subtree}(d) \vDash \text{subtree}(q)$. The intuition of this procedure is the following. Essentially, we immediately follow the semantics of the tree patterns. We test whether d matches q. If d matches q, it only remains to (recursively) test whether all subpatterns rooted at children of q can be matched somewhere in subtrees rooted at children d_c of the data tree d. If d does not match q, then we need to search whether $\text{subtree}(q)$ matches in some subtree rooted at some child d_c of d.

Algorithm 1. 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
```

Lemma 1. MATCH *is correct. That is, given a data node* d *and a query node* q, MATCH *returns true if and only if* $\text{subtree}(d) \vDash \text{subtree}(q)$.

Proof. By induction over the size of the data tree, denoted by n.

$n = 1$: We have that $\text{subtree}(d) = a$ for some $a \in \Sigma$. MATCH returns true if and only if the query tree consists of one node and d matches this node. The correctness follows from the tree pattern matching definition, which says that if $\text{subtree}(d) = a$, $\text{subtree}(q) = a$ or $\text{subtree}(q) = *$, $\text{subtree}(d) \vDash \text{subtree}(q)$.

$n > 1$: We consider two cases:

- If d matches q, we return true if, for every child q_c of q, there exists a child d_c of d such that $\text{MATCH}(d_c, q_c)$ returns true. If the query tree consists of only one node, this is obviously correct. If q has children, the correctness follows from the induction hypothesis and the definition of tree pattern matchings: if $\text{subtree}(d) = a(T_1 \cdots T_n)$, $\text{subtree}(q) = x(Q_1 \cdots Q_m)$, $x \in \Sigma \uplus \{*\}$, $a \vDash x$, and, for every $k = 1, \ldots, m$, there exists an $i_k \in \{1, \ldots, n\}$, such that $T_{i_k} \vDash Q_k$, then $\text{subtree}(d) \vDash \text{subtree}(q)$. If there exists a q_c such that $\text{MATCH}(d_c, q_c)$ is false for every d_c, we would also fail to match the whole query tree into a subtree of a child of d. Again by the definition of tree pattern matchings it is then correct to return false.
- If d does not match q, we test whether there is a child d_c of d such that $\text{subtree}(q)$ can be matched into $\text{subtree}(d_c)$. By the induction hypothesis, the recursive calls of $\text{MATCH}(d_c, q)$ compute this correctly. If there is such a matching, it is correct to return true by the definition of tree pattern matchings: if $\text{subtree}(d) = a(T_1 \cdots T_n)$ and $T_i \vDash \text{subtree}(q)$, then $\text{subtree}(d)$ *els*$\text{subtree}(q)$. Furthermore, if $\text{subtree}(d) = a(T_1 \cdots T_n)$, d does not match q, and there does not exist a T_i such that $T_i \vDash \text{subtree}(q)$, then, by definition, $\text{subtree}(d) \nvDash \text{subtree}(q)$. Hence, it is correct to return false. □

Hence, MATCH is a correct algorithm for the tree homeomorphism problem. By slightly adapting MATCH, we can even turn it into an algorithm TOP-DOWN-MATCH for the tree homeomorphism *matching* problem too. First, we need a procedure EXACT-MATCH that, given a data node d and query node q, decides whether $\text{subtree}(d)$ matches $\text{subtree}(q)$ at node d. This is easy: EXACT-MATCH only differs from MATCH in line 5, where it just returns false. Given a data node d and the root q_{root} of the query tree, TOP-DOWN-MATCH now simply iterates over all the data nodes and returns every data node d for which $\text{EXACT-MATCH}(d, q_{\text{root}})$ returns true. From this construction and from the correctness of MATCH, it is now immediate that TOP-DOWN-MATCH is correct as well.

3.1.1. Time and space complexity

We start with an analysis of the time complexity of MATCH and then we describe how an upper bound of the runtime of EXACT-MATCH can be derived from that.

Observation 1. $\text{MATCH}(d, q)$ *compares each node in* $\text{subtree}(d)$ *at most once with each node in* $\text{subtree}(q)$. *The running time of* $\text{MATCH}(d, q)$ *is* $|\text{subtree}(d)| \cdot |\text{subtree}(q)|$.

Proof. This is an easy induction on $|\text{subtree}(d)|$. If $|\text{subtree}(d)| = 1$, then MATCH tests whether d matches q and discovers that there are no children of d to iterate over. Hence, the running time is in $\mathcal{O}(|\text{subtree}(q)|)$.

If $|\text{subtree}(d)| > 1$, then MATCH tests whether d matches q and it either calls itself recursively for every child d_c of d and every child q_c of q; or it calls itself recursively for every child d_c of d and q. In both cases, we can apply the induction hypothesis. In the first case, the time complexity becomes $\mathcal{O}(\sum_{q_c}(\sum_{d_c}(|\text{subtree}(d_c)| \cdot |\text{subtree}(q_c)|)))$, and in the second case, the time complexity becomes $\mathcal{O}(\sum_{d_c}(|\text{subtree}(d_c)| \cdot |\text{subtree}(q)|))$. Hence, both cases are in $\mathcal{O}(|\text{subtree}(d)| \cdot |\text{subtree}(q)|)$. □

It is easy to see that Observation 1 implies that the time complexity of $\text{EXACT-MATCH}(d, q)$ is also in $\mathcal{O}(|\text{subtree}(d)| \cdot |\text{subtree}(q)|)$. As TOP-DOWN-MATCH simply calls EXACT-MATCH for every data node, we immediately have the following result.

Proposition 1. *The running time of* TOP-DOWN-MATCH *is in* $\mathcal{O}(|D|^2 \cdot |Q|)$. *Moreover,* TOP-DOWN-MATCH *makes* $\mathcal{O}(|D|^2 \cdot |Q|)$ *comparisons between a data node and a query node.*

It is immediate from our implementation that the algorithm can be executed by a deterministic logarithmic space bounded auxiliary pushdown automaton (see, e.g., [19]). Moreover, by Proposition 1, this auxiliary pushdown automaton runs in polynomial time. It follows from [19] that the tree homeomorphism matching problem is in LOGDCFL. As the maximum recursion depth of Algorithm 1 is $\mathcal{O}(\text{depth}(D))$, this renders the algorithm quite space-efficient, but the running time being quadratic in the size of the data tree, and the many unnecessary comparisons between query and data nodes are quite unsatisfactory. In Section 4, we show how these issues can be resolved by turning to a bottom-up approach.

3.2. A LOGSPACE *procedure*

While the top-down algorithm does not seem to be well-suited for efficiently computing *all* nodes u for which $D \vDash^u Q$, it is quite useful for *deciding* whether $D \vDash Q$, from a complexity theory point of view. Indeed, as we will exhibit, a modified version of MATCH can decide in LOGSPACE whether $D \vDash Q$.

For ease of presentation of the algorithm, we assume the *depth-first left-to-right pre-order* ordering on nodes in trees and hedges in the remainder of this section. For a node u, we denote by $u + 1$ the successor node of u in the left-to-right pre-order $<_{\text{pre}}$. We note that this assumption does not restrict our algorithm as one can compute this successor in LOGSPACE.

Algorithm 2. Top-down algorithm L-MATCH. Here, $+1$ denotes the successor in the depth-first left-to-right pre-ordering.

```
    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 maximal then
8:          return true                                     ▷ none of q's ancestors has a next sibling
        else
10:         d' ← BACKTRACK(d, q + 1)                        ▷ node to which parent(q + 1) 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 maximal then
            return false
16:     end if
        q' ← q
18:     while q' has a parent do
            d' ← BACKTRACK(d, q')                           ▷ node to which parent(q') was matched
20:         if d' is an ancestor of d + 1 then
                return L-MATCH (d + 1, q')
22:         else q' ← parent(q')
            end if
24:     end while
        return L-MATCH (d + 1, q')
26: end if
```

We argue how to transform Algorithm 1 into a LOGSPACE algorithm that decides whether $D \models Q$. We will first give an intuition of the transformation. Then we will discuss some implementation details that will allows us to analyze the space consumption. A formal proof of the correctness follows.

Intuitively, the LOGSPACE algorithm processes the data and query trees in a top-down manner, just like Algorithm 1, and it processes the children of a node from left to right. Whenever Algorithm 1 uses the recursion stack to determine which function call to issue next or which final value to return, the LOGSPACE algorithm *recomputes* the information necessary to make these decisions.

Therefore, the essential difference between Algorithm 1 and the LOGSPACE algorithm lies in a backtracking procedure. When, for example, Algorithm 1 matches a leaf q of the query tree 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 subprocedure BACKTRACK(d, q) that *recomputes* the data node onto which q's parent was matched. In particular, BACKTRACK(d, q) computes the highest possible node d' on the path from D's root to d, such that the path from D's root to d' matches the path from Q's root to q's parent. The crux of the algorithm is that this is *correct*, i.e., d' is equal to the data node onto which q's parent was matched; and that BACKTRACK(d, q) can be performed using only logarithmic space on a Turing Machine. BACKTRACK(d, q) stores d and q on tape and goes to the roots of the query and data tree. It then matches the path to d with the path to q in a greedy manner. The crux of executing BACKTRACK(d, q) using logarithmic space lies in the following. If we arrive at a node u in D (resp., Q), we have to be able to determine the child of u that lies on the path to d (resp., q). To this end, we first store d (resp., q) in a temporary variable v. We continue following the parent relation in this fashion until we find u, at which point we return the value of v, which is a child of u.[2]

In more detail, for given input nodes d and q the LOGSPACE procedure tests whether d matches q and based on the result of this test it computes the next function call. This is a rather extensive case study. In case d matches q and both nodes have children the next function call has the leftmost child of d and the leftmost child of q as its input. In case d does not match q but has children the next function call has the leftmost child of d and q as its input. In other cases, computing the next function call can be more complicated. When, for example, Algorithm 1 matches a leaf q of the query tree onto some data node d it will try to match $q + 1$ next, which is the lowest right sibling we encounter on the path from q to the root. If no such sibling exists, all query nodes are matched and the algorithm returns true. Otherwise, Algorithm 1 uses the recursion stack to compute the data node onto which $q + 1$'s parent was matched in the data tree and tries to match $q + 1$ in some proper subtree of that data node. Instead of using this recursion stack, the LOGSPACE algorithm enters the subprocedure BACKTRACK$(d, q + 1)$ that recomputes the data node onto which $q + 1$'s parent was matched. The next function call in that case has the leftmost child of BACKTRACK$(d, q + 1)$ and $q + 1$ as input. There is one more case: if d is a leaf and either d does not match q or q has children, then Algorithm 1 tries to match q to d's right

[2] Notice that the parent pointer is not mandatory for this argument. One can also determine v's parent in LOGSPACE by scanning the input tape and searching for a node with a child pointer to v.

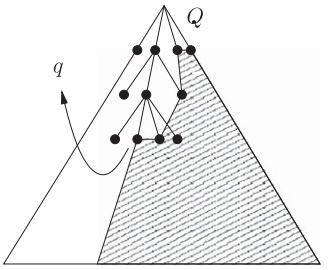

Fig. 1. Illustration of the remainder of q in Q.

sibling if it has one. In general, Algorithm 1 will try to move a query node onto $d+1$ next if such a node exists, otherwise it returns false. If $d+1$ exists, it uses the recursion stack to find the ancestor-or-self of q that is closest to q and whose parent was matched to an ancestor of $d+1$. Algorithm 1 tries to match this ancestor in subtree($d+1$). If no such parent exists then Algorithm 1 tries to match the root in the subtree($d+1$). Analogously as before, the LOGSPACE algorithm uses BACKTRACK to test for an ancestor of q whether its parent was matched to an ancestor of $d+1$.

We present the LOGSPACE procedure in Algorithm 2. For ease of presentation, we have written the algorithm as a recursive procedure, but it can be implemented to only use logarithmic space. This can be seen by observing that every recursive call to L-MATCH in Algorithm 2 is a return-statement, so the algorithm does not change when the recursion stack is not used at all. The input of the algorithm is, just as before, the root nodes d and q of the data tree D and query tree Q, respectively. In particular, we can rewrite the LOGSPACE procedure into a non-recursive algorithm: we wrap a while loop (with condition true) around the function body. In the function body we replace each function call by an update of d and q (according to the input of the function call) followed by a break statement. Thus we start an execution of the while loop for each function call.

For the sake of understanding the general idea behind Algorithm 2, let, for a query node q, the *remainder of q in Q* be the subhedge of Q consisting of the nodes $\{q' \mid q \leqslant_{\text{pre}} q' \leqslant_{\text{pre}} q_{\max}\}$, where $q_{\max}$ is the maximal query node w.r.t. the depth-first left-to-right ordering. We illustrate the remainder of q in Q in Fig. 1. Given a data node d and a query node q, the algorithm first tries to match the remainder of q in Q consistently with what has already been matched in D (lines 2–12). If this fails, it either returns false (line 15), or enters the backtracking procedure (lines 18–25).

We argued above that we can implement BACKTRACK in LOGSPACE. Algorithm 2 does not require a recursion stack and only uses logarithmic space. Thus we have the following proposition.

Proposition 2. *Algorithm* 2 *runs in* LOGSPACE.

3.2.1. Correctness of L-MATCH

We want to show that L-MATCH returns true on input D and Q if and only if $D \vDash Q$. To simplify the analysis, we imaginarily extend the algorithm by defining a matching θ. If the algorithm compares the labels of d and q in the function call L-MATCH(d,q) and they agree (in lines 2 and 6), we set $\theta(q) = d$ (and may overwrite older assignments). This mapping θ is merely used to simplify the reasoning about the algorithm.

Soundness. We will prove that whenever L-MATCH returns true on input D and Q, then $D \vDash Q$. In fact we prove a stronger claim: if L-MATCH returns true, then our mapping θ is a tree pattern matching (cfr. Definition 1). Hence θ witnesses that if L-MATCH returns true, then $D \vDash Q$.

In order to prove the soundness of L-MATCH, we first show the following Lemma, that also implies that BACKTRACK is indeed correct. That is, given q and d, the node onto which q's parent was matched can be computed by calculating the highest possible node d' on the path from D's root to d, such that the path from D's root to d' matches the path from Q's root to q's parent.

Lemma 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) *the restriction of θ to query nodes smaller than q in the ordering $<_{\text{pre}}$ is a tree pattern matching*;
(2) *θ matches the path* $\langle \text{parent}(q) \cdots \text{root } (Q)\rangle$ *into the path* $\langle \text{parent}(d) \cdots \text{root } (D)\rangle$ *as high as possible*; *and*
(3) *the path* $\langle q \cdots \text{root } (Q)\rangle$ *cannot be matched into the path* $\langle \text{parent}(d) \cdots \text{root } (D)\rangle$.

Proof. We prove the Lemma by induction on the position k of L-MATCH(d,q) in the sequence of function calls resulting from the initial procedure call L-MATCH(root (D), root (Q)). If $k = 1$ then we have L-MATCH(root (D), root (Q)), in which case there is nothing to show.

So, from now on, we assume that Lemma 2 is true for the first k function calls and we let L-MATCH(d,q) be the kth function call. We prove that it is also true for the $k+1$th function call (if there is one). We consider four cases according to Algorithm 2.

- If the labels of d and q agree and both nodes have children (line 2), the next 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 that θ, restricted to query nodes smaller than q, is a tree pattern matching. We extend this mapping by $\theta(q) = d$. This mapping clearly preserves labels. Hence, we only need to show that $\theta(q)$ is a descendant of $\theta(\text{parent}(q))$. But this clear, since by induction $\langle \text{parent}(q) \cdots \text{root } (Q)\rangle$ is matched as high as possible into the path $\langle \text{parent}(d) \cdots \text{root } (D)\rangle$, which proves (1). Combining this with the fact that $\langle q \cdots \text{root } (Q)\rangle$ cannot be matched into $\langle \text{parent}(d) \cdots \text{root } (D)\rangle$ 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). As we must match $q+1$ onto a descendant of $\theta(q)$, 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 next function call is L-MATCH$(d+1, q)$, where $d+1$ is the leftmost child of d. We do not extend θ in that case and all requirements (1)–(3) follow from the induction hypothesis.
- If the labels of d and q agree, q is a leaf, and q is not maximal (line 6), we extend the mapping θ by $\theta(q) = d$. As in the first case of this proof we know by induction that θ, restricted to query nodes less than q, is a tree pattern matching. The extended θ is still a tree pattern matching, because, due to the induction hypothesis, $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$ is matched into the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$. 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 $\langle \text{parent}(q+1) \cdots \text{root}(Q)\rangle$. Why does d' exist? First, note that parent$(q+1)$ is an ancestor of q due to the left-to-right pre-order ordering. Second, by induction, $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$ can be matched into the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$. Putting both facts together, the sub-path $\langle \text{parent}(q+1) \cdots \text{root}(Q)\rangle$ can still be matched into the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$. Hence, d' exists and $d'+1$ is its leftmost child.
  The next function call is L-MATCH$(d'+1, q+1)$. By induction, the mapping θ matches the path $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$ into the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ as high as possible and therefore, θ also matches the sub-path $\langle \text{parent}(q+1) \cdots \text{root}(Q)\rangle$ as *high* as possible into $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$. It also follows that BACKTRACK in fact calculated the node onto which parent$(q+1)$ was matched, e.g. $d' = \theta(\text{parent}(q+1))$. Combining the last two facts with the descendant requirement that is fulfilled by θ yields (2) and (3): θ matches the sub-path $\langle \text{parent}(q+1) \cdots \text{root}(Q)\rangle$ as high as possible into the path $\langle d' \cdots \text{root}(D)\rangle$ and therefore $\langle q+1 \cdots \text{root}(Q)\rangle$ cannot be matched into $\langle d' \cdots \text{root}(D)\rangle$.
- If d is a leaf and (d does not match q or q is not a leaf) and d is not maximal (line 13), we have to try to match q somewhere else. We do not extend θ, so θ restricted to query nodes smaller than q is still a tree pattern matching, which proves (1). To prove the other items, we consider two cases.
  Case 1: Assume that the next 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), i.e., to prove that root (Q) cannot be matched into the path $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$, we consider two cases.
  - If $q = q' = \text{root}(Q)$, by induction, root (Q) cannot be matched into $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ and therefore also not into $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$, which is a sub-path of $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$, which proves (3).
  - If $q \neq q' = \text{root}(Q)$, then root (Q) is an ancestor of q. By the induction hypothesis on (1) we have that $\langle \text{parent}(\theta(\text{root}(Q))) \cdots \text{root}(D)\rangle$ is a sub-path of $\langle d \cdots \text{root}(D)\rangle$. Also, $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$ is a sub-path of $\langle d \cdots \text{root}(D)\rangle$. As L-MATCH did not return a function call in line 21, $\theta(\text{root}(Q))$ is not an ancestor of parent$(d+1)$. Hence, $\langle \text{parent}(\theta(\text{root}(Q))) \cdots \text{root}(D)\rangle$ includes $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$. By induction, $\langle \text{parent}(\theta(\text{root}(Q))) \cdots \text{root}(D)\rangle$ does not match root (Q) and this property carries over to $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$, which proves (3).

  Case 2: Otherwise, the next 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 $\langle \text{parent}(q') \cdots \text{root}(Q)\rangle$. Why does d' exist? First, note that q' lies on the path $\langle q \cdots \text{root}(Q)\rangle$ and has a parent (line 20). Further, note that, by induction, the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ matches the path $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$ and therefore it also matches the sub-path $\langle \text{parent}(q') \cdots \text{root}(Q)\rangle$. It follows that d' exists and that $d'+1$ is its leftmost child. We know that q' is the lowest node on $\langle q \cdots \text{root}(Q)\rangle$ such that BACKTRACK$(d, q') = d'$ is an ancestor of $d+1$, by the condition in the while loop. Next, we will prove (2). By induction, the mapping θ matches the query path $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$ and therefore also the sub-path $\langle \text{parent}(q') \cdots \text{root}(Q)\rangle$ as high as possible into the data path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$. It follows that the mapping θ also matches the path $\langle \text{parent}(q') \cdots \text{root}(Q)\rangle$ as high as possible into the sub-path $\langle d' \cdots \text{root}(D)\rangle$. As d' is an ancestor of $d+1$ (line 21) we now have that the mapping θ matches the path $\langle \text{parent}(q') \cdots \text{root}(Q)\rangle$ as high as possible into the path $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$, which proves (2).
  In order to prove (3), i.e., to prove that the path $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$ cannot match the path $\langle q' \cdots \text{root}(Q)\rangle$, we consider two cases:
  - If $q = q'$, by the induction hypothesis, the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ cannot match the path $\langle q \cdots \text{root}(Q)\rangle$. We have that $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$ is a sub-path of $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ because d is a leaf. The claim follows.
  - If $q \neq q'$, recall that q' is the lowest ancestor of q such that $\theta(\text{parent}(q'))$ is an ancestor of $d+1$ (observe the while loop and recall that, by induction, $d' = \theta(\text{parent}(q'))$). It follows, that q' is matched somewhere on the path from parent(d) to (but not including) parent$(d+1)$. By the induction hypothesis, we cannot match the path $\langle q' \cdots \text{root}(Q)\rangle$ any higher. Hence, the path $\langle \text{parent}(d+1) \cdots \text{root}(D)\rangle$ does not match the path $\langle q' \cdots \text{root}(Q)\rangle$.
- Otherwise there does not follow a function call. □

Proposition 3. *Algorithm 2 is sound. That is, given a data D and query tree Q, if Algorithm 2 returns true, then* $D \models Q$.

Proof. If L-MATCH(d, q) returns true in line 8, then q is maximal (line 7) and the label of d matches the one of q (line 6). By Lemma 2 the mapping θ is a tree pattern matching of $Q \backslash \{q\}$ on D, such that q's parent is matched onto some ancestor of d. We extend the mapping by $\theta(q) = d$, and conclude that $D \models Q$. □

Completeness. In this section we want to prove that whenever L-MATCH returns false on input D and Q, then $D \not\models Q$. For two nodes x and y in a tree, we denote by $\langle x \cdots \overline{y}\rangle$ the path from x to y that excludes y itself. In order to prove the completeness, we first show the following Lemma.

Recall that the previous sibling of a node is its sibling to the left.

Lemma 3. *Let D be a data tree and let Q be a query tree. 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 previous siblings $\hat{d}$ of nodes on the path $\langle d \cdots \overline{\theta(\text{parent}(q))}\rangle$ or, in case q has no parent, on the path $\langle d \cdots \text{root}(D)\rangle$ that*

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

Proof. Note that, by Lemma 2, we can refer to the restriction of θ to query nodes smaller than q as a tree pattern matching. The proof is by induction on the position k of L-MATCH(d,q) in the sequence of function calls resulting from the initial procedure call L-MATCH (root (D), root (Q)). If $k=1$ then we have L-MATCH (root (D), root (Q)) in which case there is nothing to show because there are no left siblings on the path $\langle \text{root}(D)\rangle$.

So, from now on, we assume that $k \geqslant 1$ and the Lemma is true for the first k function calls. Let L-MATCH(d,q) be the kth function call. We prove that it is also true for the $k+1$th function call (if there is one). We consider four cases according to Algorithm 2.

- If the labels of d and q agree and both nodes have children (line 2), $\theta(q)$ is defined to be d. The next 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 $\langle d+1 \cdots \overline{\theta(\text{parent}(q+1))}\rangle$ is the path $\langle d+1 \cdots \overline{d}\rangle$. Since $d+1$ has no left sibling there is nothing to show.
- If the labels of d and q do not agree and d has children (line 4), the next function call is L-MATCH$(d+1,q)$, where $d+1$ is the leftmost child of d. Since $d+1$ has no left siblings, the claim follows from the induction hypothesis.
- If the labels of d and q agree, q is a leaf (line 6), and q is not maximal, $\theta(q)$ is defined to be d. BACKTRACK$(d, q+1)$ calculates the highest ancestor d' of d such that $\langle d' \cdots \text{root}(D)\rangle$ matches $\langle \text{parent}(q+1) \cdots \text{root}(Q)\rangle$. By Lemma 2 we have that $\theta(\text{parent}(q+1)) = d'$. The next function call is L-MATCH $(d'+1, q+1)$. The path $\langle d'+1 \cdots \overline{\theta(\text{parent}(q+1))}\rangle$ is the path $\langle d'+1 \cdots \overline{d'}\rangle$, where d' is $d'+1$'s parent. As $d'+1$ has no left sibling, 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 maximal, then subtree(d) does not match subtree(q). We first show the following invariant which we will need later:

  Invariant 2. *For every call of* L-MATCH *until the kth call, 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 subtree$(\theta(\text{parent}(q')))$ does not match subtree$(\text{parent}(q'))$.*

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

  $\ell = 1$: Here $q' = q$, q has a parent (line 18), and we know that (i) subtree(q) cannot be matched into subtree(d) (line 13), (ii) q cannot be matched into the path $\langle \text{parent}(d) \cdots \overline{\theta(\text{parent}(q))}\rangle$ by Lemma 2, (iii) there are no right siblings on the path $\langle d \cdots \overline{\theta(\text{parent}(q))}\rangle$, since otherwise we would have returned a function call in line 21, and (iv) subtree(q) cannot be matched into subtree$(\hat{d})$ for every left sibling $\hat{d}$ of the path $\langle d \cdots \overline{\theta(\text{parent}(q))}\rangle$, by the induction hypothesis of Lemma 3. From (i–iv) we can conclude that no proper subtree of $\theta(\text{parent}(q))$ matches subtree(q), which implies that subtree$(\theta(\text{parent}(q)))$ does not match subtree $(\text{parent}(q))$.

  $\ell > 1$: Let the claim be true for the first ℓ while loop executions. We prove that it is also true for the $\ell+1$th execution. Let q' be the query node of the $\ell+1$th while loop execution. Here, $q' \neq q$ and q' has a parent (line 18). There must have been a function call L-MATCH$(q', \theta(q'))$ and 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 (i) subtree(q') cannot be matched into subtree$(\theta(q'))$ by the induction hypothesis, (ii) q' cannot be matched into the path $\langle \text{parent}(\theta(q')) \cdots \overline{\theta(\text{parent}(q'))}\rangle$ by Lemma 2, (iii) there are no right siblings on the path $\langle \theta(q') \cdots \overline{\theta(\text{parent}(q'))}\rangle$, since otherwise we would have returned a function call in line 21, and (iv) subtree(q') cannot be matched into subtree$(\hat{d})$, for every left sibling $\hat{d}$ of the path $\langle \theta(q') \cdots \overline{\theta(\text{parent}(q'))}\rangle$ by the induction hypothesis of Lemma 3. From (i–iv), we can conclude that no proper subtree of $\theta(\text{parent}(q'))$ matches subtree (q'), which implies that the subtree$(\theta(\text{parent}(q')))$ does not match the subtree$(\text{parent}(q'))$. □

We return to the proof of the main induction. We denote the left sibling of $d+1$ by prevSib$(d+1)$. We consider two cases.

Case 1: Assume that next 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 the induction hypothesis, subtree(q) cannot be matched into subtree$(\hat{d})$ for any left sibling $\hat{d}$ of the path $\langle d \cdots \text{root}(D)\rangle$. Since parent$(d+1)$ is an ancestor of d, it is enough to show that the subtree rooted at prevSib$(d+1)$, which is a subtree that includes d, does not match subtree(q). We know that (i) subtree(q) cannot be matched into subtree(d), (ii) q cannot be matched into the path $\langle \text{parent}(d) \cdots \text{root}(D)\rangle$ by Lemma 2, (iii) there are no right siblings on the path $\langle d \cdots \overline{\text{prevSib}(d+1)}\rangle$due to the left-to-right pre-order successor, and (iv) 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 the induction hypothesis. From (i–iv) it follows that we cannot match subtree(q) into the subtree rooted at prevSib$(d+1)$.
- If $q \neq q' = \text{root}(Q)$, then by Lemma 2 there must have been a function call L-MATCH$(q', \theta(q'))$. By the

induction hypothesis, subtree(q') cannot be matched into subtree($\hat{d}$) for any of the left siblings $\hat{d}$ of the path $\langle\theta(q')\cdots$ root $(D)\rangle$. Furthermore, there must have been a while loop execution with q''s child on the path from q to q' as current query node. Since parent($d+1$) is an ancestor of $\theta(q')$ (otherwise we would have returned a function call in line 21), it is enough to show that the subtree rooted at prevSib($d+1$) does not match the subtree(q'). We know that (i) subtree(q') cannot be matched into subtree($\theta(q')$) by Invariant 2, (ii) q' cannot be matched into the path from parent($\theta(q')$) to root (D) by Lemma 2, (iii) there are no right siblings on the path $\langle\theta(q')\cdots\overline{\text{prevSib}(d+1)}\rangle$, because parent($d+1$) is an ancestor of $\theta(q')$, which is an ancestor of d, and (iv) subtree(q') cannot be matched into subtree($\hat{d}$) for every left sibling $\hat{d}$ of the path from $\theta(q')$ to root (D) by the induction hypothesis. From (i–iv) it follows that we cannot match subtree(q') into the subtree rooted at prevSib($d+1$).

Case 2: Otherwise, the next 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 $\langle d'\cdots$ root $(D)\rangle$ matches $\langle$parent$(q')\cdots$ root $(Q)\rangle$. By Lemma 2, d' equals θ(parent(q')).

We know that q' is the lowest node on $\langle q\cdots$ root $(Q)\rangle$ such that θ(parent(q')) is an ancestor of $d+1$, because of the condition in the while loop. It follows that q' is matched somewhere on the path $\langle d\cdots\overline{\text{parent}(d+1)}\rangle$ (for the case $q'\neq q$). No matter whether $q'=q$ or not, there was a function call L-MATCH(q',d_0) for some d_0 on the path $\langle d\cdots\overline{\text{parent}(d+1)}\rangle$. By the induction hypothesis and Lemma 2 there is no left sibling $\hat{d}$ on the path $\langle d_0\cdots\overline{\theta(\text{parent}(q'))}\rangle$ such that subtree($\hat{d}$) matches subtree(q').

Since d_0 is in the subtree rooted at prevSib($d+1$), we now only need to show that subtree(prevSib($d+1$)) does not match subtree(q'). We consider two cases:

- If $q=q'$, then we know that (i) subtree(q) cannot be matched into subtree(d), (ii) q cannot be matched into the path $\langle$parent$(d)\cdots\overline{\theta(\text{parent}(q))}\rangle$ by Lemma 2, (iii) there are no right siblings on the path $\langle d\cdots\overline{\text{prevSib}(d+1)}\rangle$ due to the definition of the left-to-right pre-order successor, and (iv) subtree(q) cannot be matched into subtree($\hat{d}$) for every left sibling $\hat{d}$ of the path $\langle d\cdots\overline{\theta(\text{parent}(q))}\rangle$ by the induction hypothesis. From (i–iv) it follows that the subtree rooted at prevSib($d+1$) does not match 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($\theta(q'),q'$). We know that (i) subtree(q') cannot be matched into subtree($\theta(q')$)by Invariant 2, (ii) q' cannot be matched into the path $\langle$parent$(\theta(q'))\cdots\overline{\theta(\text{parent}(q))}\rangle$ by Lemma 2, (iii) there are no right siblings on the path $\langle\theta(q')\cdots\overline{\text{prevSib}(d+1)}\rangle$, because there are no right siblings on the path $\langle d\cdots\overline{\text{prevSib}(d+1)}\rangle$, and the path $\langle\theta(q')\cdots\overline{\text{prevSib}(d+1)}\rangle$ is a sub-path of that path (otherwise we would have returned a function call earlier, when the child of q' was the current query node), and (iv) subtree(q') cannot be matched into subtree($\hat{d}$) for every previous sibling $\hat{d}$ of the path $\langle\theta(q')\cdots\overline{\theta(\text{parent}(q))}\rangle$ by the induction hypothesis. From (i–iv) it follows that we cannot matchsubtree(q') into the subtree rooted at prevSib($d+1$).
- Otherwise there does not follow a function call. □

Proposition 4. *Algorithm 2 is complete. That is, given a data D and query tree Q, if Algorithm 2 returns false, then $D\not\models Q$.*

Proof. We prove the proposition by induction on the number of nodes in the data tree D. If $|D|=1$ then 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 15, which proves the completeness for that case.

Now suppose that $|D|>1$. Assume L-MATCH returns false in line 15. 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 either q has children or the labels of q and d do not agree. Due to line 14 , d is the maximal node w.r.t. $<_{\text{pre}}$, 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 an extra rightmost child to the root. 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 according 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 Lemma 3 we know that for every child d' of the data root in D, subtree(d') cannot match 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 Lemma 2 we know that we cannot match the query root into the path $\langle$parent$(d+1)\cdots$ 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 subtree(q') into some subtree of the data root. But we are not able to do this, so $D\not\models Q$. □

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' is overwritten with parent(q') 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 in the following

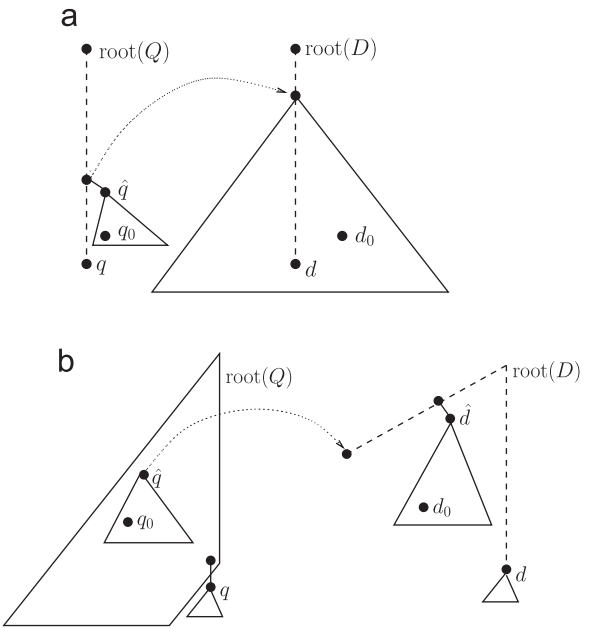

Fig. 2. Illustrations of the induction hypotheses in the proof of Lemma 4. (a) Induction hypothesis (I2) and (b) induction hypothesis (I3), respectively.

lemma (it is an immediate consequence of Lemma 4 letting $q_0 = q$ and $d_0 = d$).

Lemma 4. *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*

$\forall q_0 \geqslant q, \quad \forall d_0 \geqslant d$, *we have not yet called* L-MATCH(d_0, q_0) *before.*

Proof. We prove the lemma by induction on the position k of L-MATCH(d,q) in the sequence of function calls resulting from the initial procedure call.

More specifically, our induction hypothesis will be: at the time when L-MATCH(d,q) is called

(I1): $\forall q_0 \geqslant q, \forall d_0 \geqslant d$, we have not yet called L-MATCH (d_0, q_0) before;

(I2): for all right siblings $\hat{q}$ of nodes on the path $\langle q \cdots \text{root}\ (Q)\rangle$, for all nodes $q_0 \in \text{subtree}(\hat{q})$, and for all data nodes $d_0 \in \text{subtree}(\theta(\text{parent}(\hat{q})))$, we have not yet called L-MATCH(d_0, q_0).

(I3): for all nodes $\hat{q} < q$, for all nodes $q_0 \in \text{subtree}(\hat{q})$, for all right siblings $\hat{d}$ on the path $\langle \theta(\hat{q}) \cdots \text{root}\ (D)\rangle$, and for all data nodes $d_0 \in \text{subtree}(\hat{d})$, we have not yet called L-MATCH(d_0, q_0).

We illustrate the hypotheses (I2) and (I3) in Fig. 2.

If $k = 1$ then we have L-MATCH(root (D), root (Q)) in which case there is nothing to show.

So, from now on we assume that the Lemma holds for the first k function calls. Let L-MATCH(d,q) be the kth function call. We prove that the Lemma also holds for the $k+1$th function call (if there is one).

Let us start with a simple observation concerning (I3). The induction hypothesis for (I3) implies that, for all query nodes $\hat{q} < q$, for all query nodes $q_0 \in \text{subtree}(\hat{q})$, for all right siblings $\hat{d}$ on the path $\langle \theta(\hat{q}) \cdots \text{root}\ (D)\rangle$, and for all data nodes $d_0 \in \text{subtree}(\hat{d})$ we have not called L-MATCH(d_0, q_0) before we called L-MATCH(d,q). We argue why this remains true even after calling L-MATCH(d,q), but before the next function call is made. Towards a contradiction, assume that this was not the case. In that case there would be a query node $\hat{q} < q$ such that $q \in \text{subtree}(\hat{q})$, and a right sibling $\hat{d}$ on the path $\langle \theta(\hat{q}) \cdots \text{root}\ (D)\rangle$, such that $d \in \text{subtree}(\hat{d})$. But this cannot be because, due to Lemma 2, the ancestors of q are matched on the path $\langle d \cdots \text{root}\ (D)\rangle$ and hence d cannot be in a subtree of a right sibling on the path $\langle \theta(\hat{q}) \cdots \text{root}\ (D)\rangle$. Hence, (I3) is also still true right after calling L-MATCH(d,q). (†)

Next we will consider the four possible function calls following the kth function call L-MATCH(d,q) and we will show that (I1)–(I3) still hold for the next function call.

- If the next function call is L-MATCH$(d+1, q+1)$ (line 3), then $\theta(q)$ is defined to be d. Here, $d+1$ is the leftmost child of d and $q+1$ is the leftmost child of q. The induction hypothesis for (I1) implies that for all $q_0 \geqslant q$, for all data nodes $d_0 \geqslant d$, we have not called L-MATCH(d_0, q_0) before we called L-MATCH(d,q). In the meantime, we only executed L-MATCH(d,q), so for all $q_0 \geqslant q+1$, for all data nodes $d_0 \geqslant d+1$, we have not called L-MATCH(d_0, q_0), which proves (I1).
  Item (I2) of the induction hypothesis implies that, for all right siblings $\hat{q}$ of nodes on the path $\langle q \cdots \text{root}\ (Q)\rangle$, for all nodes $q_0 \in \text{subtree}(\hat{q})$, for all data nodes $d_0 \in \text{subtree}(\theta(\text{parent}(\hat{q})))$, we have not called L-MATCH(d_0, q_0) before we called L-MATCH(d,q). Since $q+1$ is the leftmost child of q, we only need to show that, for all right siblings $\hat{q}$ of $q+1$, for all nodes $q_0 \in \text{subtree}(\hat{q})$, for all data nodes $d_0 \in \text{subtree}(\theta(\text{parent}(\hat{q})))$ (which is the subtree(d)), we have not called L-MATCH(d_0, q_0) before. But this follows from the induction on (I1), because data nodes in subtree(d) are greater or equal to d and query nodes in subtrees of $q+1$'s right siblings are greater than q. This shows (I2).
  In order to prove (I3) we need to show that for $\hat{q} < q+1$, for all query nodes $q_0 \in \text{subtree}(\hat{q})$, for all right siblings $\hat{d}$ on the path $\langle \theta(\hat{q}) \cdots \text{root}\ (D)\rangle$, and for all data nodes $d_0 \in \text{subtree}(\hat{d})$ we have not called L-MATCH(d_0, q_0) before calling L-MATCH$(d+1, q+1)$.
  By the observation (†) above this is true for $\hat{q} < q$. So, let us consider $\hat{q} = q$. The fact that $\theta(q) = d$ now implies that $d_0 > d$ and $q_0 \geqslant q$. The claim follows from the induction on item (1) and the fact that we only called L-MATCH(d,q) in the meantime.
- If the next function call is L-MATCH$(d+1, q)$ (line 5), then $d+1$ is the leftmost child of d. The induction on (I1) implies that (I1) is true (as above). Since we only called L-MATCH(d,q) in the meantime and did not change the mapping θ at all, (I2) is a direct consequence

of the induction hypothesis on (I2). Since we did not change the mapping θ and the query node serving as argument of the $k+1$th function call is the same as the argument of the kth function call, (I3) follows from the observation (†) made above.

- If the next function call is L-MATCH$(d'+1, q+1)$ (line 11), then $\theta(q)$ is defined to be 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 on (I1) assures that, for all $q_0 \geqslant q$, for all data nodes $d_0 \geqslant d$, we have not called L-MATCH(d_0, q_0) before we called L-MATCH(d, q). In the meantime, we executed L-MATCH(d, q), so for all $q_0 \geqslant q+1$, for all data nodes $d_0 \geqslant d$, we have not called L-MATCH(d_0, q_0). In order to prove (I1), we still need to show that this is also true for all $q_0 \geqslant q+1$ and for all data nodes d_0 with $d'+1 \leqslant d_0 < d$. We consider two cases:
  - If $q_0 \in \text{subtree}(q+1)$ we can make use of the induction hypothesis on (I2). The query node $q+1$ is a right sibling of a node on the path $\langle q \cdots \text{root}(Q)\rangle$ and hence we have not called L-MATCH(d_0, q_0) before for all nodes $d_0 \in \text{subtree}(\theta(\text{parent}(q+1)))$. This proves our case because, by Lemma 2, $\theta(\text{parent}(q+1))$ is equal to d' and d' is an ancestor of d. Clearly, subtree(d') includes all nodes d_0 with $d'+1 \leqslant d_0 < d$. Hence, for all $q_0 \in \text{subtree}(q+1)$ and for all d_0 with $d'+1 \leqslant d_0 < d$ we have not called L-MATCH(d_0, q_0) before.
  - If $q_0 \notin \text{subtree}(q+1)$ we can make use of the induction hypothesis on (I2) again. By definition of the left-to-right pre-order, q_0 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 on (I2) it follows that, for all data nodes $d_0 \in \text{subtree}(\theta(\text{parent}(\hat{q})))$, we have not called L-MATCH(d_0, q_0). This proves our case, because parent$(\hat{q})$ is an ancestor of or equal to parent$(q+1)$ and, by Lemma 2, $\theta(\text{parent}(\hat{q}))$ is an ancestor of or equal to $\theta(\text{parent}(q+1))$, which is equal to d'. Clearly, subtree(d') includes all nodes d_0 with $d'+1 \leqslant d_0 < d$ and so does subtree$(\theta(\text{parent}(\hat{q})))$. Hence, for all $q_0 \notin \text{subtree}(q+1)$ and for all d_0 with $d'+1 \leqslant d_0 < d$ we have not called L-MATCH(d_0, q_0) 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, (I2) immediately follows from the induction hypothesis on (I2).
  In order to prove (I3) we need to show that, for $\hat{q} < q+1$, for all query nodes $q_0 \in \text{subtree}(\hat{q})$, for all right siblings $\hat{d}$ on the path $\langle \theta(\hat{q}) \cdots \text{root}(D)\rangle$, and for all data nodes $d_0 \in \text{subtree}(\hat{d})$, we have not called L-MATCH(d_0, q_0) before calling L-MATCH$(d'+1, q+1)$.
  By the observation (†) above this is true for $\hat{q} < q$. So, let us consider $\hat{q} = q$. The left-to-right pre-order and the fact that $\theta(q) = d$, implies that $d_0 > d$ and $q_0 \geqslant q$. The claim follows from the induction on (I1) and the fact that we only called L-MATCH(d, q) in the meantime.

- If the next function call is L-MATCH$(d+1, q')$ (lines 21 or 25), then $d+1$ is a right sibling of a node on the path $\langle d \cdots \text{root}(D)\rangle$ (due to the left-to-right pre-order and the fact that d is a leaf, see line 13) and q' is an ancestor of or is equal to q.
  The induction on (I1) implies that, for all $q_0 \geqslant q$, for all data nodes $d_0 \geqslant d$, we have not called L-MATCH(d_0, q_0) before we called L-MATCH(d, q). In the meantime, we only executed L-MATCH(d, q), so, for all $q_0 \geqslant q$, for all data nodes $d_0 \geqslant d+1$, we have not called L-MATCH(d_0, q_0) before.
  In order to prove (I1) we still need to show that this is also true for all query nodes q_0 with $q' \leqslant q_0 < q$ and for all data nodes $d_0 \geqslant d+1$. So, take a query node q_0 such that $q' \leqslant q_0 < q$. Note that such a query node q_0 is in subtree(q'). Furthermore, each node d_0 that is greater or equal to $d+1$ is in the subtree of some right sibling $\hat{d}$ on the path $\langle \text{prevSib}(d+1) \cdots \text{root}(D)\rangle$ because d is a leaf. This path is a sub-path of $\langle \theta(q') \cdots \text{root}(D)\rangle$, because q' is the lowest ancestor of q whose parent is an ancestor of $d+1$, which means that q' is mapped onto a node on $\langle \text{parent}(d) \cdots \text{prevSib}(d+1)\rangle$ by Lemma 2 and the fact that $q' < q$. By induction on (I3) it follows that we have not called L-MATCH(d_0, q_0) before calling L-MATCH(d, q).
  Since q' is an ancestor of or equal to q, the path $\langle q' \cdots \text{root}(Q)\rangle$ is a sub-path of the path $\langle q \cdots \text{root}(Q)\rangle$. Hence, (I2) immediately follows from the induction on (I2).
  Since we did not change the mapping θ and the query node serving as argument of the $k+1$th function call is smaller or equal to the argument of the kth function call, (I3) follows from the observation (†) made above. □

Propositions 3, 4, and Lemma 4 imply the correctness of L-MATCH.

Proposition 5. *Algorithm* 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* $D \models Q$.

3.2.2. Space complexity of L-MATCH

We already argued in the main body of the paper that the recursion stack has no influence on the operation of L-MATCH. 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 $\langle d' \cdots \text{root}(D)\rangle$ matches the path $\langle \text{parent}(q) \cdots \text{root}(Q)\rangle$. The difficulty lies in the fact that we cannot store both paths. Instead, we store d and q. We also store two help variables d_0 and q_0, which are initialized to be root (D) and root (Q), respectively. We now iterate over the following. We compare the labels of d_0 and q_0. If they match, we overwrite d_0 and q_0 with the children of d_0 and q_0 that lie on the paths to d and q, respectively. This is performed as explained in the beginning of this section. We can start at d (resp., q), scan the input tape for the unique node that has a child pointer to d (resp., q), and continue upwards in this manner until we find a child of d_0, resp., q_0. If the labels of d_0 and q_0 do not match, we only overwrite d_0 with its child

on the path to d. We continue until we matched the whole path $\langle \text{parent}(q) \cdots \text{root}(Q) \rangle$. Finally we return the data node onto which we matched parent(q).

3.2.3. The complexity of the tree homeomorphism problem

As argued above, L-MATCH can be performed in LOGSPACE. Putting this together with the fact that reachability in trees is LOGSPACE-complete, given the tree as a pointer structure [8], we obtain the following Theorem.

Theorem 3. *The tree homeomorphism problem is LOGSPACE-complete.*

4. The bottom-up algorithm

Although the previously presented top-down algorithms for tree homeomorphism matching are quite space-efficient, their time complexity is quite high and they involve quite a lot of recomputing of already obtained matchings, which is unsatisfactory. We therefore turn to a bottom-up matching approach which has the property that *no* obtained matchings between the data and query tree need to be recomputed, which leads to a better time complexity of the overall algorithm.

Before presenting the bottom-up algorithm for the tree homeomorphism matching problem in detail, we need to introduce several formal notions. As in the previous section, we first present an algorithm for the tree homeomorphism problem and then show how to change it into an algorithm for the tree homeomorphism matching problem.

In the present section, we assume the *left-to-right post-order* ordering $<_{\text{post}}$ on nodes in trees and hedges. For a node u, we denote by $u+1$ and $u-1$ the successor and predecessor of u in the left-to-right post-order ordering, respectively. Moreover, when we, e.g., use terminology such as "largest" and "smallest", we always assume the left-to-right post-ordering. In this section, we also assume that XML documents are stored on tape in left-to-right post-order (or, alternatively, together with a left-to-right post-order index), which allows a random-access machine model to verify the left-to-right post-order ordering in constant time. To simplify the presentation of our algorithm, we also assume two dummy nodes in every tree and hedge: nil and ∞. The node nil is such that nil $+1$ is the smallest node in the hedge, and the node ∞ is defined as the successor of the largest node of the hedge. Given two nodes $h_{\text{from}} \leqslant h_{\text{until}}$ in a hedge H, we denote by the interval $[h_{\text{from}}, h_{\text{until}}]$ the subhedge of H consisting only of the nodes $\{v \mid h_{\text{from}} \leqslant v \leqslant h_{\text{until}}\}$.[3] The notion of such an interval in a tree is illustrated in Fig. 3(a). Here, the interval $[h_{\text{from}}, h_{\text{until}}]$ is the striped area in the tree. Given a hedge H and a node $h \in \text{Nodes}(H)$, we denote by $\text{subhedge}^H(h)$ the subhedge $[h_{\text{from}}, h]$, where h_{from} is the smallest descendant of h's leftmost sibling according to the left-to-right post-order ordering. We illustrate this notion in Fig. 3(b).

[3] Notice that our definition of a hedge did not assume all root nodes of the individual trees to be siblings of one another.

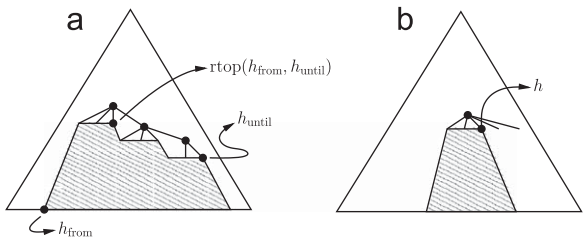

Fig. 3. Illustration of a hedge interval and RTOP (a) and of $\text{subhedge}^H(h)$ (b).

When H is a data hedge or a tree pattern query, we refer to $[h_{\text{from}}, h_{\text{until}}]$ as a data or query hedge interval, respectively. We extend the semantics of tree pattern matching to hedges as follows. Let $Q_1 \cdots Q_n$ be a query hedge interval $[q_{\text{from}}, q_{\text{until}}]$ and $D_1 \cdots D_m$ be a data hedge interval $[d_{\text{from}}, d_{\text{until}}]$. We say that $[d_{\text{from}}, d_{\text{until}}]$ matches $[q_{\text{from}}, q_{\text{until}}]$, denoted by $[d_{\text{from}}, d_{\text{until}}] \vDash [q_{\text{from}}, q_{\text{until}}]$, if, for every Q_i, $i = 1, \ldots, n$, there exists a D_j, $j = 1, \ldots, m$, such that $D_j \vDash Q_i$.

Before presenting the intuition about the bottom-up tree homeomorphism algorithm, we describe an auxiliary procedure RTOP, which, given two nodes h_{from} and h_{until}, returns the rightmost node among the topmost nodes in the interval $[h_{\text{from}}, h_{\text{until}}]$. More formally, $\text{RTOP}(h_{\text{from}}, h_{\text{until}})$ is the node u such that depth(u) is minimal and u is larger than every other node v in $[h_{\text{from}}, h_{\text{until}}]$ with depth(u) = depth(v). This notion is illustrated in Fig. 3(a). Furthermore, in order to simplify the presentation of the algorithm, we define $\text{RTOP}(h_{\text{from}}, h_{\text{until}}) = \infty$ if $h_{\text{from}} > h_{\text{until}}$. Notice that RTOP can easily be computed in time linear in the depth of the tree and in logarithmic space by traversing the path from h_{until} to the query root and comparing the previous siblings of nodes on the path with h_{from} w.r.t. the left-to-right post-ordering. Indeed, assume that $h_{\text{from}} \leqslant h_{\text{until}}$. Let u be the highest ancestor of h_{until} that has a previous sibling s such that $s \geqslant h_{\text{from}}$. If no such u exists, then $\text{rtop}(h_{\text{from}}, h_{\text{until}})$ is h_{until}. Otherwise, $\text{rtop}(h_{\text{from}}, h_{\text{until}})$ is s.

We first present an algorithm for *deciding* whether $D \vDash Q$ and show later how it can be extended to an algorithm for the tree homeomorphism matching problem. The main procedure of our algorithm is called TMATCH. Given a data node d and query nodes q_{from} and q_{until}, TMATCH returns the largest query node q in the interval $[q_{\text{from}}, q_{\text{until}}]$ such that $\text{subtree}^D(d)$ matches $[q_{\text{from}}, q]$ if q exists; and $q_{\text{from}} - 1$ otherwise. Hence, if d is the root of D, and q_{from} and q_{until} are the leftmost leaf and the root of Q, respectively, then $D \vDash Q$ if and only if TMATCH returns q_{until}.

TMATCH uses an auxiliary procedure called HMATCH, which, given a data node d and query nodes q_{from} and q_{until}, returns the largest node q in the interval $[q_{\text{from}}, q_{\text{until}}]$ such that $\text{subhedge}^D(d)$ matches $[q_{\text{from}}, q]$ if q exists; and $q_{\text{from}} - 1$ otherwise.

We start by explaining the operation of TMATCH, which is presented in Algorithm 3. Given a data node d and query nodes q_{from} and q_{until}, TMATCH first starts by recursively calling HMATCH with the same query nodes for the subhedge D' of D defined by d's last child, yielding result q_{best} (see Fig. 4(a)). In the remainder of TMATCH, we essentially want to test how q_{best} can be improved when we also consider the node d in addition to D'. One particular interesting case

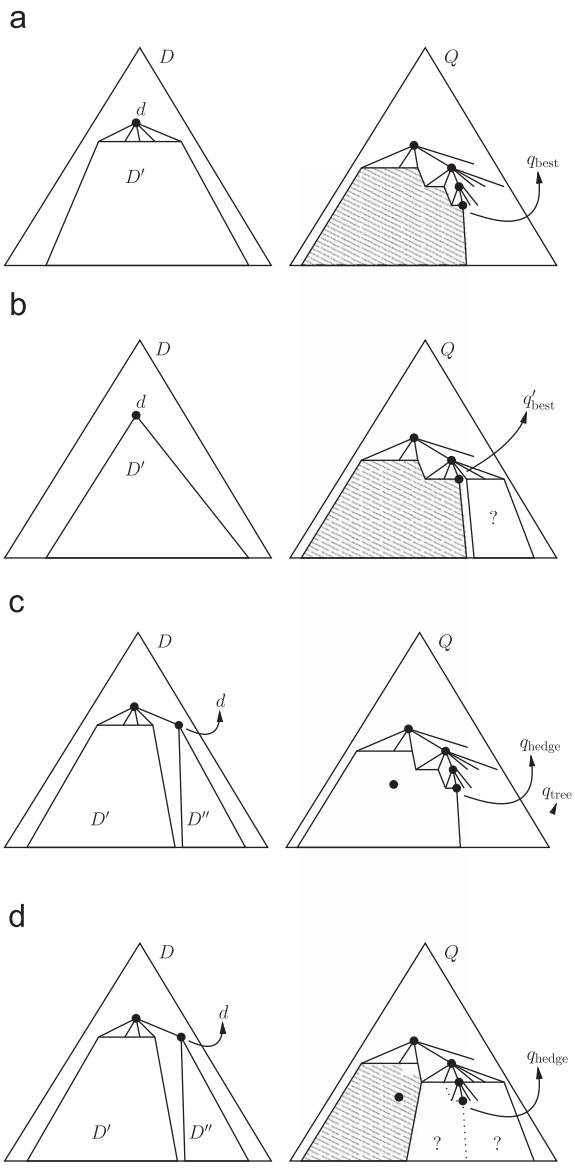

Fig. 4. Illustrations of the tree homeomorphism algorithm. (a) Operation of TMatch: recursive call of HMatch. (b) Operation of TMatch: recursive call of TMatch. (c) Operation of HMatch: first recursive calls of TMatch and HMatch. (d) Operation of HMatch: a subsequent recursive call of TMatch, trying to improve q_{tree}.

is when q_{best} is a last sibling and its parent has the same label as d. In this case, we can at least improve our best query node to q_{best}'s parent which we call here q'_{best}. Furthermore, it is possible that q'_{best} is not yet the best query node we can obtain. In particular, we still need to test which part of the hedge defined by $[q'_{best}+1, \text{lastSib}(q'_{best})]$ can be matched in the subtree below d (see Fig. 4(b)). The largest node that is obtained in this manner is the node that TMatch should return.

Algorithm 3. Function TMatch. Here, $+1$ and -1 denote the successor and predecessor in the depth-first left-to-right post-ordering, respectively.

```
    TMatch (DNode d, QNode q_from, QNode q_until)
2:  if d is a leaf then q_best ← q_from − 1
    else q_best ← HMatch(lastChild(d), q_from, q_until)
4:  end if
    if q_best + 1 ⩽post q_until and d matches q_best + 1 then
6:    q_best ← q_best + 1
      if q_best + 1 ⩽post lastSib(q_best) then
8:      return TMatch(d, q_best + 1, lastSib(q_best))
      else return q_best
10:     end if
    else return q_best
12: end if
```

We now explain the operation of HMatch, which is presented in Algorithm 4. Essentially, given d, q_{from}, and q_{until}, HMatch starts by recursively calling itself with the same query nodes on the hedge defined by the previous sibling of d (i.e., D' in Fig. 4(c)), yielding q_{hedge}, and by calling TMatch with the same query nodes on the subtree under d itself (D'' in Fig. 4(c)), yielding q_{tree}. The remainder of HMatch consists of iteratively improving q_{tree} and q_{hedge}. That is, while it is possible that D' and D'' yield small values of q_{tree} and q_{hedge}, their concatenation can give rise to a much larger part of the query that can be matched. Essentially, this is due to the fact that the matching of tree pattern queries is *unordered*. For example, it can occur that we need to match a certain first sibling in D', a second one in D'', a third one again in D' and so on. Hence, the procedure HMatch alternates between finding best matches in D' and D'' until it reaches a fixpoint.

Algorithm 4. Function HMatch. Here, $+1$ and -1 denote the successor and predecessor in the depth-first left-to-right post-ordering, respectively.

```
    HMatch (DNode d, QNode q_from, QNode q_until)
2:  if d is a first sibling then return TMatch(d, q_from, q_until)
    else
4:    q_hedge ← HMatch(prevSib(d), q_from, q_until)
      q_tree ← TMatch(d, q_from, q_until)
6:    loop
        if q_hedge = q_tree then return q_hedge
8:      else if q_tree <post q_hedge then
          rtop ← RTop (q_tree + 1, q_hedge)
10:       while rtop <post ∞ and q_hedge <post lastSib(rtop) do
            q_tree ← TMatch(d, rtop + 1, lastSib(rtop))
12:         rtop ← RTop (q_tree + 1, q_hedge)
          end while
14:       if q_tree ⩽post q_hedge then return q_hedge
          end if
16:     else
          rtop ← RTop (q_hedge + 1, q_tree)
18:       while rtop <post ∞ and q_tree <post lastSib(rtop) do
            q_hedge ← HMatch(prevSib(d), rtop + 1, lastSib(rtop))
20:         rtop ← RTop (q_hedge + 1, q_tree)
          end while
22:       if q_hedge ⩽post q_tree then return q_tree
          end if
24:     end if
      end loop
26: end if
```

However, we need to take care in how this fixpoint is computed. One possible case is illustrated in Fig. 4(d). This particular case builds further on the situation in

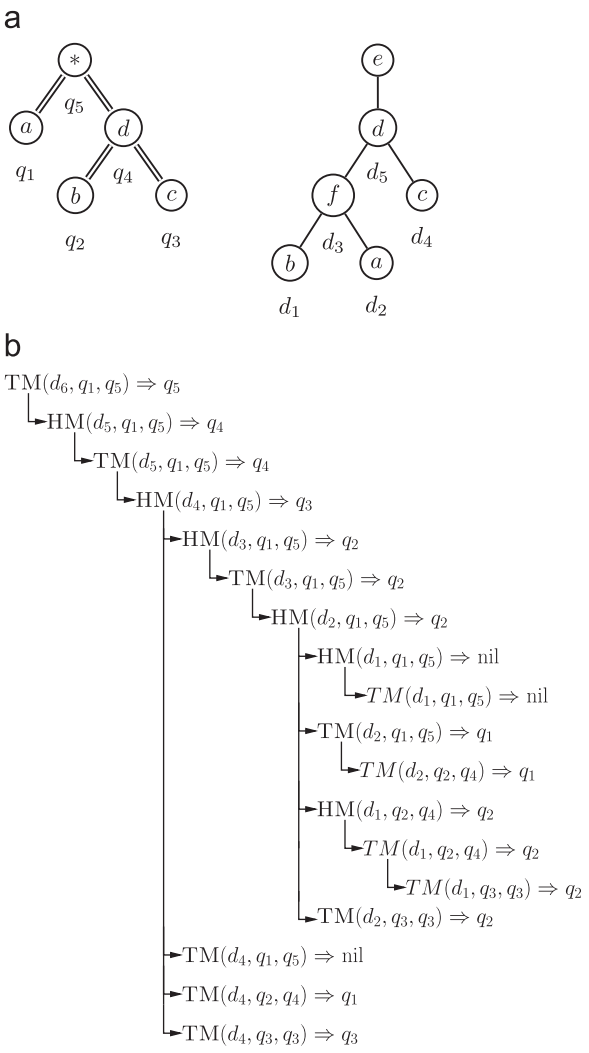

Fig. 5. Illustrations for Example 1. (a) Query tree (left) and data tree (right) of Example 1. (b) Function calls of HMatch (HM) and TMatch (TM) of Example 1.

Fig. 4(c). Here, we try to improve q_{tree} by starting the TMatch procedure again for the node d, but now only with the part of the query marked with question marks. The case where q_{tree} is larger than q_{hedge} is dual and not illustrated here.

Example 1. Figs. 5(a) and (b) illustrate an example for the bottom up algorithm. For brevity, we denote TMatch and HMatch with TM and HM, respectively. The first calls of TM and HM demonstrate the basic recursive structure of our algorithm: TM on a node d calls HM on the rightmost child of d. HM on a node d returns TM of d if that node is a first sibling; or performs a divide-and-conquer technique by calling HM on the left sibling of d and TM on d itself (as in the function call $\mathrm{HM}(d_4, q_1, q_5)$). Further recursive calls to TM or HM are then needed to maximize the part of the query that can be matched.

The simplest function call in the example that performs such further recursive calls is the call $\mathrm{HM}(d_2, q_1, q_5)$, which starts by computing $q_{\text{hedge}} = \mathrm{HM}(d_1, q_1, q_5)$ and $q_{\text{tree}} = \mathrm{TM}(d_2, q_1, q_5)$. As can be seen in Fig. 5(b), $q_{\text{hedge}} = \text{nil}$. The call $\mathrm{TM}(d_2, q_1, q_5)$ is more successful, because d_2 and q_1 are both labeled with a. In general, it might be possible that q_2 and further nodes can be matched in subtree(d_2). The function call $\mathrm{TM}(d_2, q_2, q_4)$ checks that possibility. (For sure, q_1 and q_5 cannot both be matched on d_2, which is why we restrict the query tree interval by q_4.) But q_2 is not labeled with a so the return value of the two TM calls is q_1. After this initial phase, $\mathrm{HM}(d_2, q_1, q_5)$ tries to improve q_{tree} and q_{hedge} iteratively. It calls $\mathrm{HM}(d_1, q_2, q_4)$ and improves q_{hedge} to be q_2, because q_2 and d_1 are both labeled with b. Further improvements fail as there is no c-labeled node in the subhedge of d_2.

A similar iterative improvement is illustrated by $\mathrm{HM}(d_3, q_1, q_5)$. Observe that we try to improve q_{tree} here and call $\mathrm{TM}(d_4, q_2, q_4)$ and $\mathrm{TM}(d_4, q_3, q_3)$. Only the latter

call yields an improvement. But we cannot omit the former one: if subtree(d_4) would match subtree(q_4), then the former call would yield q_4 and the latter call would yield q_3. As we want our algorithm to return the largest query node such that the interval ending with it can be matched the result of the former call would have been the relevant one in that case.

4.1. Correctness

The main technical difficulty of this section is proving that TMatch is correct.

Lemma 5. *Let D be a data tree and let Q be a query tree.* TMatch *is correct, that is, given the root node d of D, the smallest and largest node q_{from} and q_{until} of Q, respectively,* TMatch *returns q_{until} iff $D \models Q$.*

For the proof of Lemma 5, we start with a few simple observations.

Observation 4. *A node u is not a last sibling $\Leftrightarrow u+1$ is a leaf.*

Proof. Left to right: if u is not a last sibling, then $u+1$ is the leftmost descendant leaf of the right sibling of u, or the right sibling of u itself if it is a leaf. Right to left: if u is the last node in a left-to-right post-order traversal, then u is a last sibling for which $u+1$ does not exist. For all other last siblings u, $u+1$ is u's parent, which is not a leaf. □

We call a hedge interval *complete* when if it contains a certain node, it also contains its children.

Observation 5. *In Algorithms* 3 *and* 4, *the following properties hold*:

(1) q_{until} *is always a last sibling.*
(2) q_{from} *is always a leaf.*
(3) $[q_{\text{from}}, q_{\text{until}}]$ *is always a complete interval.*

Proof. (1) In our initial call of TMatch, q_{until} is the root node of the tree, which is always a last sibling. The property for the deeper recursive calls follows immediately from a straightforward inspection of the recursive function calls in the algorithm.

(2) In our initial call of TMatch, q_{from} is the smallest node of Q, which is always a leaf. Furthermore, in TMatch we only call HMatch with q_{from} as a second parameter and TMatch with $q_{\text{best}}+1$ as a second parameter if q_{best} is not a last sibling (which is a leaf due to Observation 4). In HMatch all recursive calls have either q_{from} or rtop $+1$ as second parameter. We show that, in this case, rtop is never a last sibling. Hence, according to Observation 4, rtop $+1$ is always a leaf. In the calls of TMatch on line 11, we have that rtop $<\infty$ and $q_{\text{hedge}}<$lastSib(rtop), due to the while condition. As rtop $<\infty$, we have that rtop $\leqslant q_{\text{hedge}}$ due to the calls of RTop on lines 9 and 12. Hence, rtop $<$ lastSib(rtop). The proof is analogous for the calls of HMatch on line 19.

(3) In the initial call of TMatch, the claim obviously holds. In TMatch we call HMatch with q_{from} and q_{until}, for which the claim then trivially also holds; and TMatch with $q_{\text{best}}+1$ and lastSib(q_{best}) if q_{best} is not a last sibling. Hence, $[q_{\text{best}}+1, \text{lastSib}(q_{\text{best}})]$ is equal to the hedge subtree(nextSib(q_{best})) $\cdots$ subtree(lastSib(q_{best})), which is complete. The proof for the recursive calls in HMatch is analogous. □

Observation 6. *Let d_1 and d_2 be data nodes and q be a query node. If $[d_1, d_2]$ does not match* subtree(q), *then $[d_1, d_2]$ does not match any query tree interval containing* subtree(q).

Proof. Let q_{from} and q_{until} be such that $[q_{\text{from}}, q_{\text{until}}] =$ subtree(q). For $q'_{\text{from}} \leqslant q_{\text{from}}$ and $q'_{\text{until}} \geqslant q_{\text{until}}$, it can be shown by a simple structural induction on the hedge $[q'_{\text{from}}, q'_{\text{until}}]$ that $[d_1, d_2]$ does not match $[q'_{\text{from}}, q'_{\text{until}}]$. □

Observation 7. *Let H be a data hedge and $[q_{\text{from}}, q_{\text{until}}]$ be a complete query tree interval. We have that q is the largest node in $[q_{\text{from}}, q_{\text{until}}]$ such that $H \models [q_{\text{from}}, q]$ if and only if*

- *H matches $[q_{\text{from}}, q]$; and*
- *either $q = q_{\text{until}}$ or H does not match* subtree($q+1$).

Proof. Left to right: let H be a data hedge and let $[q_{\text{from}}, q_{\text{until}}]$ be a query tree interval. Let q be the largest node in $[q_{\text{from}}, q_{\text{until}}]$ such that $H \models [q_{\text{from}}, q]$. If $q = q_{\text{until}}$ we are done. Otherwise, if, towards a contradiction, H matches subtree($q+1$), then we also immediately have that H matches $[q_{\text{from}}, q+1]$, which contradicts the maximality of q.

Right to left: let q be a query node in $[q_{\text{from}}, q_{\text{until}}]$ such that H matches $[q_{\text{from}}, q]$. If $q = q_{\text{until}}$ then we are done. Otherwise, notice that, as $q+1$ is in the complete interval $[q_{\text{from}}, q_{\text{until}}]$, we have that subtree($q+1$) is entirely contained in $[q_{\text{from}}, q_{\text{until}}]$. Hence, if H does not match subtree($q+1$), then H also cannot match $[q_{\text{from}}, q+1]$. The latter can be shown by a simple structural induction on $[q_{\text{from}}, q+1]$. □

4.1.1. Correctness of TMatch

For readability, we split the correctness proof into several lemmas. Essentially, the proof is by induction on the height of the data node d in D.

Lemma 6. *Let d be a leaf data node and q_{from} and q_{until} be query nodes. Given d, q_{from}, and q_{until},* TMatch *is correct, that is,* TMatch *returns the largest node q in $[q_{\text{from}}, q_{\text{until}}]$ such that* subtree(d) $\models [q_{\text{from}}, q]$ *if it exists; and $q_{\text{from}} - 1$ otherwise.*

Proof. By induction on the number of nodes of $[q_{\text{from}}, q_{\text{until}}]$.

$q_{\text{from}} = q_{\text{until}}$: We initialize q_{best} with $q_{\text{from}} - 1$ on line 2. If d does not match q_{from} on line 5, we immediately return $q_{\text{best}} = q_{\text{from}} - 1$ on line 11. If d matches $q_{\text{from}} = q_{\text{until}}$ on line 5, q_{best} gets the value q_{from} on line 6. As $q_{\text{from}} = q_{\text{until}}$ is a last sibling (Observation 5), we do not execute the recursive call on line 8 and return q_{from} in line 9. Both cases are easily seen to be correct.

$q_{\text{from}} < q_{\text{until}}$: We initialize q_{best} with $q_{\text{from}} - 1$ on line 2. If d does not match q_{from} on line 5, we return

$q_{best} = q_{from} - 1$ in line 11, which is correct. If d matches q_{from} in line 5, then q_{best} gets the value q_{from} and we enter the if-test on line 7. We need to consider two cases:
(1) q_{from} *is a last sibling*: In this case, we return q_{from} on line 9. This is correct, as $q_{from}+1$ is q_{from}'s parent, which cannot be matched onto d due to the semantics of the descendant axis.
(2) q_{from} *is not a last sibling*: if q_{from} has a right sibling, we execute TMatch recursively on d, $q_{from}+1$, and lastSib(q_{from}), yielding q. By induction, q is computed correctly. That is, if $q = (q_{from}+1)-1$, which implies that d does not match $q_{from}+1$, we return q_{from}, which is correct. Otherwise, we argue that subtree(d) $= d$ matches $[q_{from}, q]$ but not subtree($q+1$). By Observation 7, this would complete the proof. By induction, we immediately have that d matches $[q_{from}, q]$. If $q <$ lastSib(q_{from}), we also have by induction that d does not match subtree($q+1$). If $q =$ lastSib(q_{from}), then $q+1$ is q_{from}'s parent. Hence, d does not match subtree($q+1$), as $q+1$ has a child and d has not. □

Lemma 7. *Let d be a data node with height $n>1$ and q_{from} and q_{until} be query nodes. If* HMatch *is correct for all data nodes of height up to $n-1$, then* TMatch *is correct for all data nodes of height up to n. That is, given d, q_{from}, and q_{until},* TMatch *returns the largest node q in $[q_{from}, q_{until}]$ such that* subtree(d) $\vDash [q_{from}, q]$ *if it exists; and $q_{from}-1$ otherwise.*

Proof. Assume that HMatch is correct for all data nodes of height up to $n-1$. As d is not a leaf, we start by calling HMatch on lastChild(d), q_{from}, and q_{until} on line 3 (see also Fig. 4(a)), yielding q_{best}. By our assumption, q_{best} is computed correctly. We now prove the lemma by induction on the number of nodes of $[q_{from}, q_{until}]$.

$q_{from} = q_{until}$: We consider two cases.
(1) If subhedge (lastChild(d)) does not match q_{from}, then q_{best} is $q_{from}-1$. Consequently, we test whether d matches q_{from} on line 5. If d does not match q_{from}, we return $q_{from}-1$ on line 11. If d matches q_{from}, then q_{best} gets the value q_{from}. As $q_{from} = q_{until}$ is a last sibling (Observation 5), we do not execute the recursive call on line 8 and return q_{from} in line 9. Both cases are easily seen to be correct.
(2) Otherwise, $q_{best} = q_{from} = q_{until}$. In this case we return q_{best}, which is correct.

$q_{from} < q_{until}$: (1) If both subhedge (lastChild(d)) and d do not match q_{from}, then we return $q_{from}-1$ on line 11, which is correct.
(2) If subhedge (lastChild(d)) matches q_{from} and $q_{best} = q_{until}$ on line 5, then we return q_{until}. Due to the correctness of HMatch, this means that subhedge (lastChild(d)) already matches $[q_{from}, q_{until}]$, hence, subtree(d) matches $[q_{from}, q_{until}]$ by our tree pattern matching semantics.
(3) If subhedge (lastChild(d)) matches q_{from}, $q_{best}+1 \leqslant q_{until}$, and d does not match $q_{best}+1$ on line 5, then we return q_{best} in line 11. We consider two cases.

- q_{best} *is not a last sibling*: Hence, $q_{best}+1$ is a leaf (Observation 4). Due to the correctness of HMatch for subhedge (lastChild(d)), we know that subhedge (lastChild(d)) does not match subtree $(q_{best}+1) = q_{best}+1$. Hence, returning q_{best} is correct.
- q_{best} *is a last sibling*: Hence, $q_{best}+1$ is q_{best}'s parent. Due to the correctness of HMatch, we have that subhedge (lastChild(d)) $\vDash [q_{from}, q_{best}]$. Towards a contradiction, assume that subhedge (d) $\vDash$ subtree($q_{best}+1$). As d does not match $q_{best}+1$, this implies that subhedge (lastChild(d)) $\vDash$ subtree ($q_{best}+1$). However, this contradicts that HMatch is correct. Hence, it is correct to return q_{best} due to Observation 7.

(4) Otherwise, denote by q^0_{best} the value of the variable q_{best} after the assignment on line 3. We have that q^0_{best} is correctly computed on line 3 and that d matches $q^0_{best}+1$, after which q_{best} gets the value $q^0_{best}+1$. Notice that $q^0_{best}+1 \geqslant q_{from}$. We need to consider two cases:

- $q^0_{best}+1$ *is a last sibling*: We return $q^0_{best}+1$ in line 9. If $q^0_{best}+1 = q_{until}$, this is correct. If $q^0_{best}+1 < q_{until}$, towards a contradiction, assume that subtree(d) matches subtree($q^0_{best}+2$). As $q^0_{best}+2$ is the parent of $q^0_{best}+1$, this would mean that subhedge (lastChild(d)) $\vDash$ subtree ($q^0_{best}+1$), which is a contradiction.
- $q^0_{best}+1$ *is not a last sibling*: if $q^0_{best}+1$ has a right sibling, we execute TMatch on d, $q^0_{best}+2$, and lastSib($q^0_{best}+1$) on line 8, yielding q. By induction, q is computed correctly. If q is $(q^0_{best}+2)-1$, which implies that subtree(d) does not match $q^0_{best}+2$, we return $q^0_{best}+1$, which is correct. Otherwise, according to Observation 7, we need to show that subtree(d) matches $[q_{from}, q]$ but not subtree($q+1$). By induction, we have that subtree(d) matches $[q_{from}, q]$. If $q <$ lastSib($q^0_{best}+1$), we also have by induction that subtree(d) does not match subtree($q+1$). If $q =$ lastSib($q^0_{best}+1$), we have that subtree(d) doesnot match

subtree($q+1$), because there does not exist an $u \neq 1$ s.t. subtree(d) $\models$ subtree ($q^0_{\text{best}}+1$), and $q+1$ is $q^0_{\text{best}}+1$'s parent. □

4.1.2. Correctness of HMatch

Lemma 8. *Let* rtop = RTop (q_1, q_2) *and* $q_1 \leqslant q_2$. *If* $q_1 \in$ subhedge (q_2), *then* rtop = q_2 *and* $q_2 \leqslant$ lastSib(rtop). *If* $q_1 \notin$ subhedge (q_2), *then* rtop $< q_2$ *and* $q_2 <$ lastSib(rtop).

Proof. Recall that, by definition, subhedge (q_2) is the interval $[q_{\text{small}}, q_2]$, where q_{small} is the smallest descendant of q_2's leftmost sibling.

$q_1 \in$ subhedge (q_2): As both q_1 and q_2 are in subhedge (q_2), we have that $[q_1, q_2]$ is entirely contained in subhedge (q_2).

By definition, rtop is the largest node in $[q_1, q_2]$ among the nodes with minimal depth. As q_2 has minimal depth in subhedge (q_2) and q_2 is the largest node in $[q_1, q_2]$, we have that rtop = q_2.

$q_1 \notin$ subhedge (q_2): Notice that this can only occur when q_2 has a parent. As $q_1 \leqslant q_2$, we have that $q_1 < q_{\text{small}}$. By definition of the left-to-right post-ordering, we have that q_1 is either a left sibling of an ancestor of q_2 (not including the ancestors themselves), or a descendant-or-self thereof. Let u_1 and u_2 be the two unique siblings such that $u_1 \neq u_2$, q_1 is in subtree(u_1), and q_2 is in subtree(u_2). Notice that $q_1 \leqslant u_1 < q_2 < u_2$. Hence, u_1 is in $[q_1, q_2]$ and depth (u_1) $<$ depth (q_2). As q_2 has minimal depth in subhedge (q_2), we have that rtop is not in subhedge (q_2). By definition of RTop, this immediately implies that rtop $< q_2$. Furthermore, as depth (lastSib(rtop)) = depth (rtop) $\leqslant$ depth (u_1) = depth (u_2) and as lastSib(rtop) is also rtop's largest sibling, we have that lastSib (rtop) $\geqslant u_2 > q_2$. □

Corollary 1. *If rtop = RTop*(q_1, q_2) *then* $q_1 \in$ *subhedge*(*rtop*).

Proof. As rtop is in $[q_1, q_2]$, rtop is also the rightmost node among the topmost nodes in $[q_1,$ rtop]. If we assume that $q_1 \notin$ subhedge (rtop), then Lemma 8 implies that rtop $<$ rtop which is a contradiction. □

Lemma 9. *All function calls of* TMatch(d, q_1, q_2) *in the loop of* HMatch *have the property that* $[q_1, q_2]$ *is an interval which includes* subtree($q_{\text{hedge}}+1$). *All function calls of* HMatch(d, q_1, q_2) *in the loop of* HMatch *have the property that* $[q_1, q_2]$ *is an interval which includes* subtree($q_{\text{tree}}+1$).

Proof. For the first statement, we have to show that (i) $q_1 \leqslant q_{\text{small}}$, where q_{small} is the smallest node in subtree($q_{\text{hedge}}+1$) and (ii) $q_2 \geqslant q_{\text{hedge}}+1$.

First, observe that the function calls of RTop on lines 9 and 12 results in a value of rtop that is at most q_{hedge}. If rtop $< q_{\text{hedge}}$ then rtop $< q_{\text{small}}$ as, by Lemma 8, rtop = q_{hedge} when rtop is in $[q_{\text{small}}, q_{\text{hedge}}]$. Hence, rtop $+ 1 \leqslant q_{\text{small}}$. If rtop = q_{hedge}, we know that rtop is not a last sibling due to the condition of the while loop on line 10. Hence, $q_{\text{small}} = q_{\text{hedge}} + 1 =$ rtop $+ 1$ is a leaf (Observation 4). This proves property (i).

Property (ii) is immediate as the condition of the while-loop on line 10 requires that $q_2 =$ lastSib(rtop) $\geqslant q_{\text{hedge}}+1$.

The proof of the second statement is analogous to the proof of the first statement. □

Lemma 10. *The loop on line* 6, *and the while loops on lines* 10 *and* 18 *perform at most a linear number of iterations.*

Proof. Notice that we exit the loop on line 6 if max($q_{\text{tree}}, q_{\text{hedge}}$) *does not* increase. However, this value cannot keep increasing indefinitely as it is bounded from above by q_{until} in the algorithm. Hence, the loop performs at most a linear number of iterations.

The while loop on line 10 terminates after a linear number of iterations, as the value of rtop increases with each execution and the while loop only continues as long as rtop is smaller than q_{hedge}, a value which remains unchanged. The argument for the while loop on line 18 is analogous. □

Lemma 11. *Let d be a data node and* q_{from} *and* q_{until} *be query nodes. If* TMatch *is correct for all data nodes of height up to n, then* HMatch *is correct for all data nodes of height up to n. That is, given d,* q_{from}, *and* q_{until}, HMatch *returns the largest node q in* $[q_{\text{from}}, q_{\text{until}}]$ *such that* subhedge (d) *matches* $[q_{\text{from}}, q]$ *if it exists*; *and nil otherwise.*

Proof. Let k be such that d has k left siblings (including d itself). We prove the lemma by induction on k.

If $k = 1$ then the Lemma is immediate from the function call on line 2 and the assumption that TMatch is correct for all data nodes of height up to n.

So, from now on, we assume that $k > 1$. We need to show that the algorithm returns $q_{\text{from}} - 1$ if subhedge (d) does not match q_{from}. Otherwise, we show that we return a q in $[q_{\text{from}}, q_{\text{until}}]$ if subhedge (d) matches $[q_{\text{from}}, q]$ and either

- $q = q_{\text{until}}$, or
- neither subtree(d), nor subhedge (prevSib(d)) matches subtree($q+1$).

In the remainder of the proof, we refer to the above property with the label (†). The correctness of property (†) follows directly from our tree pattern query semantics if we return $q_{\text{from}} - 1$ and from Observation 7 otherwise. Indeed, from Observation 5 we know that $[q_{\text{from}}, q_{\text{until}}]$ is complete. Furthermore, subhedge (d) does not match subtree($q+1$) if and only if neither subtree(d) nor subhedge (prevSib(d)) match subtree($q+1$).

Notice that the loop on line 6 terminates by Lemma 10. We now proceed with an induction over the number ℓ of loop executions proving that the following invariants hold:

- (I1): if q_{tree} is not $q_{\text{from}} - 1$ then subhedge (d) matches $[q_{\text{from}}, q_{\text{tree}}]$;
- (I2): if q_{hedge} is not $q_{\text{from}} - 1$ then subhedge (d) matches $[q_{\text{from}}, q_{\text{hedge}}]$;
- (I3): $q_{\text{tree}} = q_{\text{until}}$ or subtree(d) does not match subtree($q_{\text{tree}}+1$); and,
- (I4): $q_{\text{hedge}} = q_{\text{until}}$ or subhedge (prevSib(d)) does not match subtree($q_{\text{hedge}}+1$).

At the same time, we show that, if the algorithm returns a certain value q, the property (†) holds for q.

$\ell = 0$ (*before the first loop execution*): We computed q_{hedge}, which results from executing HMATCH on prevSib(d), q_{from}, and q_{until}; and we computed q_{tree}, which results from executing TMATCH on d, q_{from}, and q_{until} (see also Fig. 4(c)). By induction on k, we have that q_{hedge} is computed correctly. Moreover, as we assume that TMATCH is correct for all data nodes of height up to n, we also have that q_{tree} is computed correctly. Properties (I1) and (I2) immediately follow from the correctness of the recursive calls of TMATCH and HMATCH. Moreover, Observation 7 implies that (I3) and (I4) also hold. As the algorithm does not return anything up to here, we do not have to show yet that (†) holds.

$\ell \geqslant 1$ (*subsequent loop executions*): We consider three cases.

(1) If $q_{\text{hedge}} = q_{\text{tree}}$, we return q_{hedge}. This is correct, as in this case, properties (I1)–(I4) immediately imply property (†).

(2) If $q_{\text{tree}} < q_{\text{hedge}}$, notice that we do not change the value of q_{hedge} in this iteration of the loop. Hence, for the induction, we only need to show that properties (I1) and (I3) are preserved. We consider two cases.
If $q_{\text{hedge}} = q_{\text{until}}$ the while loop in line 10 is not executed and we return q_{until} in line 14. Here, it follows immediately from (I2) that (†) holds.
If $q_{\text{hedge}} < q_{\text{until}}$ we consider two cases.
- If subtree(d) does not match subtree($q_{\text{hedge}} + 1$), none of the function calls TMATCH(d, q_1, q_2) in the while loop yield a value greater than q_{hedge}. This follows from the correctness of TMATCH for data nodes up to height n, and from Lemma 9, stating that $[q_1, q_2]$ always includes subtree($q_{\text{hedge}} + 1$). Indeed, should such a function call TMATCH (d, q_1, q_2) yield a greater value than q_{hedge}, then we would have that subtree(d) matches subtree $(q_{\text{hedge}} + 1)$, which contradicts that we are investigating the case that subtree(d) does not match subtree($q_{\text{hedge}} + 1$). Hence, we return q_{hedge} in line 14. Correctness of the property(†) for q_{hedge} now follows from the following facts:
  - $q_{\text{hedge}} \geqslant q_{\text{from}}$, as $q_{\text{tree}} < q_{\text{hedge}}$;
  - $q_{\text{hedge}} < q_{\text{until}}$;
  - subhedge (d) matches $[q_{\text{from}}, q_{\text{hedge}}]$, by (I2);
  - subtree(d) does not match subtree $(q_{\text{hedge}} + 1)$; and,
  - subhedge (prevSib(d)) does not match subtree $(q_{\text{hedge}} + 1)$ by (I4).
- If subtree(d) matches subtree($q_{\text{hedge}} + 1$) the proof is more complicated. First, observe that the while loop on line 10 terminates by Lemma 10.
  For the remainder of this case, we will show that $q_{\text{tree}} > q_{\text{hedge}}$ after exiting the while loop in the $i+1$th execution of the test on line 10. In particular, this implies that the algorithm will not return any value in iteration ℓ of the loop. So we only need to show that, at the end of the current iteration, properties (I1) and (I3) hold.

To show (I3), we will show that, if in the jth execution of the while loop we obtain a value q for the variable q_{tree} for which it holds that $q > q_{\text{hedge}}$ then we either have that $q = q_{\text{until}}$ or that subtree(d) does not match subtree($q + 1$). Afterwards, we show (I1).
We start by showing that $q_{\text{tree}} > q_{\text{hedge}}$ after exiting the while loop:
Goal 1: $q_{\text{tree}} > q_{\text{hedge}}$ *after exiting the while loop in the* $i+1$th *execution of the test on line* 10. *So we execute the while body i times and then exit the loop.*
Let q^i_{tree} denote the value of q_{tree} at the end of the ith execution (i.e., after the assignment on line 11) and let q^0_{tree} be the value of q_{tree} before entering the while loop. Furthermore, let rtop^i denote the value of rtop at the end of the ith execution (i.e., after the assignment on line 12). Let rtop^0 be the value of rtop before entering the while loop.
($i = 0$): We will show that this case does not occur. That is, the body of the while loop is always executed at least once. Towards a contradiction, assume that we do not execute the body of the while loop. We consider two cases. If we exit the while loop one of them must hold.

- *Case* 1: rtop$^0 < \infty$ and $q_{\text{hedge}} \geqslant$ lastSib(rtop^0). Recall that rtop^0 = RTOP $(q^0_{\text{tree}} + 1, q_{\text{hedge}})$. Due to Lemma 8, $q_{\text{hedge}} \geqslant$ lastSib(rtop^0) implies that (i) rtop^0 = lastSib(rtop^0) = q_{hedge} and that (ii) $q^0_{\text{tree}} + 1$ is in subhedge (q_{hedge}). As $q_{\text{hedge}} < q_{\text{until}}$ and q_{hedge} is a last sibling this means that $q^0_{\text{tree}} + 1$ is in subtree($q_{\text{hedge}} + 1$). Moreover, as we are in the case that $q_{\text{tree}} < q_{\text{hedge}}$, we know by induction on ℓ (statement (I3) in particular) that subtree(d) does not match subtree($q^0_{\text{tree}} + 1$). However, as we have shown above that $q^0_{\text{tree}} + 1$ is in subtree($q_{\text{hedge}} + 1$), this contradicts the fact that we are in the case that subtree(d) matches subtree($q_{\text{hedge}} + 1$).
- *Case* 2: rtop$^0 = \infty$. By definition of RTOP, this means that $q^0_{\text{tree}} + 1 > q_{\text{hedge}}$. But we are currently investigating in the case that $q^0_{\text{tree}} < q_{\text{hedge}}$. Contradiction.

Hence, we showed that the while loop on line 10 is executed at least once.
($i > 0$): Again, we consider the two possible settings in which we exit the while loop. We show again that the first of the two does not occur here.
- *Case* 1: rtop$^i < \infty$ and $q_{\text{hedge}} \geqslant$ lastSib(rtop^i). Recall that rtop^i = RTOP $(q^i_{\text{tree}} + 1, q_{\text{hedge}})$. Due to Lemma 8, $q_{\text{hedge}} \geqslant$ lastSib(rtop^i) implies that (i) rtop^i = lastSib(rtop^i) = q_{hedge} and that (ii) $q^i_{\text{tree}} + 1$ is in subhedge (q_{hedge}), implying that $q^i_{\text{tree}} + 1 \leqslant q_{\text{hedge}}$. As $q_{\text{hedge}} < q_{\text{until}}$ and q_{hedge} is a last sibling this means that $q^i_{\text{tree}} + 1$ is in subtree($q_{\text{hedge}} + 1$). Since we did not exit the while loop in the ith test, we have that $q_{\text{hedge}} <$ lastSib(rtop^{i-1}). Hence, we have that $q^i_{\text{tree}} + 1 \leqslant q_{\text{hedge}} <$ lastSib(rtop^{i-1}). Recall that q^i_{tree} = HMATCH(prevSib(d), rtop$^{i-1} + 1$,

lastSib(rtop^{i-1})). By the correctness of TMATCH, Observation 7, and the fact that [rtop^{i-1} + 1, lastSib(rtop^{i-1})] is a complete interval (Observation 5) we can conclude thatsubtree(d) does not match subtree($q^i_{\text{tree}}+1$) which, we argued above, is a subtree of subtree($q_{\text{hedge}}+1$). Hence, subtree(d) does not match subtree($q_{\text{hedge}}+1$), which contradicts the fact that we are in the case that subtree(d) matches subtree($q_{\text{hedge}}+1$).

- *Case* 2: rtop$^i=\infty$. Hence, $q^i_{\text{tree}}+1>q_{\text{hedge}}$. We prove that it cannot be the case that $q^i_{\text{tree}}=q_{\text{hedge}}$. Hence, $q^i_{\text{tree}}>q_{\text{hedge}}$ and Goal 1 follows. To this end, assume, towards a contradiction, that $q^i_{\text{tree}}=q_{\text{hedge}}$. Recall that $q^i_{\text{tree}}=\text{TMATCH}(d,\text{rtop}^{i-1}+1,\text{lastSib}(\text{rtop}))$. Moreover, lastSib(rtop^{i-1})$>q_{\text{hedge}}$ since otherwise we would have exited the while loop right after test i. We conclude that $q_{\text{hedge}}+1$ is a node in [rtop$^{i-1}+1$, lastSib(rtop^{i-1})]. However, as subtree(d) matches subtree($q_{\text{hedge}}+1$), this would imply that subtree(d) also matches $[\text{rtop}^{i-1}+1,q_{\text{hedge}}+1]=[\text{rtop}^{i-1}+1,q^i_{\text{tree}}+1]$ which is in contradiction with the correctness of TMATCH.

This concludes the proof of Goal 1.

Goal 2. *If in the* jth *execution of the while loop we obtain a value* q *for the variable* q_{tree} *for which it holds that* $q>q_{\text{hedge}}$ *and* $q+1\leqslant q_{\text{until}}$, *then we have that* subtree(d) *does not match* subtree($q+1$).

Observe that we need at least one execution of the body of the while, since before the first execution we have that $q_{\text{tree}}<q_{\text{hedge}}$. Let q^j_{tree} denote the value of q_{tree} at the end of the jth execution (i.e., after the assignment on line 11) and let q^0_{tree} be the value of q_{tree} before entering the while loop. Furthermore let rtop^j denote the value of rtop at the end of the jth execution (i.e., after the assignment on line 12). Let rtop^0 be the value of rtop before entering the while loop.

Hence, for every $j\geqslant 1$, q^j_{tree} is the result of a function call $\text{TMATCH}(d,\text{rtop}^{j-1}+1,\text{lastSib}(\text{rtop}^{j-1}))$. If $q^j_{\text{tree}}>q_{\text{hedge}}$ we will exit the while loop right after the current iteration. We consider three cases.

- If $q^j_{\text{tree}}<\text{lastSib}(\text{rtop}^{j-1})$ we have that subtree(d) does not match the subtree of $q^j_{\text{tree}}+1$ due to the correctness of TMATCH for data nodes up to height n and Observation 7.
- If $q^j_{\text{tree}}=q_{\text{until}}$ the claim is trivial.
- The remaining case is that $q^j_{\text{tree}}=\text{lastSib}(\text{rtop}^{j-1})<q_{\text{until}}$. In this case, $q^j_{\text{tree}}+1$ is the parent of q^j_{tree} due to Observation 4. We consider two cases.

  $j=1$: We want to prove that subtree(d) does not match subtree($q^0_{\text{tree}}+1$) and that subtree($q^0_{\text{tree}}+1$) is a subtree of subtree($q^1_{\text{tree}}+1$). Then we can conclude that subtree(d) does not match subtree($q^1_{\text{tree}}+1$).

  We start by proving that subtree(d) does not match subtree($q^0_{\text{tree}}+1$). By induction on ℓ (and, in particular, by (I3)) we know that $q^0_{\text{tree}}=q_{\text{until}}$ or subtree(d) does not match subtree($q^0_{\text{tree}}+1$). If $q^0_{\text{tree}}=q_{\text{until}}$ we wouldn't be in the case that $q^0_{\text{tree}}<q_{\text{hedge}}$. We can conclude that subtree(d) does not match subtree($q^0_{\text{tree}}+1$).

  It remains to be shown that subtree($q^0_{\text{tree}}+1$) is a subtree of subtree($q^1_{\text{tree}}+1$). Line 9 states that $\text{rtop}^0=\text{RTOP}(q^0_{\text{tree}}+1,q_{\text{hedge}})$. Corollary 1 implies that then $q^0_{\text{tree}}+1$ is a node in subhedge (rtop^0). Now we take into consideration that we are investigating in the case that $q^1_{\text{tree}}=\text{lastSib}(\text{rtop}^0)$ which implies that subhedge (rtop^0) $\subseteq$ subhedge (q^1_{tree}). Combining this with the consequence of the Corollary it follows that $q^0_{\text{tree}}+1$ is a node in subhedge (q^1_{tree}). Recall that $q^1_{\text{tree}}+1$ is q^1_{tree}'s parent. Hence, $q^0_{\text{tree}}+1$ is a node in subtree($q^1_{\text{tree}}+1$) and subtree($q^0_{\text{tree}}+1$) is a subtree of subtree($q^1_{\text{tree}}+1$).

  $j>1$: Analogously as in the $j=1$ case, we prove that subtree(d) does not match subtree($q^{j-1}_{\text{tree}}+1$) and that subtree($q^{j-1}_{\text{tree}}+1$) is a subtree of subtree($q^j_{\text{tree}}+1$). Then we can conclude that subtree(d) does not match subtree($q^j_{\text{tree}}+1$).

  We start by proving that subtree(d) does not match subtree($q^{j-1}_{\text{tree}}+1$). We have that $q^{j-1}_{\text{tree}}=\text{TMATCH}(d,\text{rtop}^{j-2}+1,\text{lastSib}(\text{rtop}^{j-2}))$. Notice that, if $q^{j-1}_{\text{tree}}<\text{lastSib}(\text{rtop}^{j-2})$, we immediately have by the correctness of TMATCH and Observation 7 that subtree(d) does not match subtree($q^{j-1}_{\text{tree}}+1$). So, towards a contradiction, let us assume that $q^{j-1}_{\text{tree}}\geqslant\text{lastSib}(\text{rtop}^{j-2})$.

  Notice that $q_{\text{hedge}}<\text{lastSib}(\text{rtop}^{j-2})$ and that $\text{rtop}^{j-1}\leqslant q_{\text{hedge}}$, otherwise we wouldn't have arrived in the jth iteration. Moreover, $\text{rtop}^{j-2}<\text{rtop}^{j-1}$. As $\text{rtop}^{j-2}<\text{rtop}^{j-1}\leqslant$ lastSib (rtop^{j-2}), we also have that lastSib(rtop^{j-1})$\leqslant$ lastSib(rtop^{j-2}). This implies that lastSib(rtop^{j-1}) $\leqslant$lastSib(rtop^{j-2})$<\ q^{j-1}_{\text{tree}}+1\leqslant q^j_{\text{tree}}$, which is in contradiction with $q^j_{\text{tree}}=\text{lastSib}$ (rtop^{j-1}), which is the case we are investigating.

  It remains to be shown that subtree($q^{j-1}_{\text{tree}}+1$) is a subtree of subtree($q^j_{\text{tree}}+1$). Line 12 states that $\text{rtop}^{j-1}=\text{RTOP}(q^{j-1}_{\text{tree}}+1,q_{\text{hedge}})$. Corollary 1 implies that then $q^{j-1}_{\text{tree}}+1$ is a node in subhedge (rtop^{j-1}). Now we take into consideration that we are investigating the case that $q^j_{\text{tree}}=\text{lastSib}(\text{rtop}^{j-1})$ which implies that subhedge (rtop^{j-1}) $\subseteq$ subhedge (q^j_{tree}). Combining this with the consequence of the Corollary it follows that $q^{j-1}_{\text{tree}}+1$ is a node in subhedge (q^j_{tree}). Recall that $q^j_{\text{tree}}+1$ is q^j_{tree}'s parent. Hence, $q^{j-1}_{\text{tree}}+1$ is a node in subtree($q^j_{\text{tree}}+1$) and subtree($q^{j-1}_{\text{tree}}+1$) is a subtree of subtree($q^j_{\text{tree}}+1$).

  This concludes the proof of Goal 2.

It remains to show that (I1) holds at the end of the ℓth iteration of the loop, that is, that subhedge (d) matches $[q_{\text{from}},q_{\text{tree}}]$. Due to (I2) we

have that subhedge (d) matches $[q_{\text{from}}, q_{\text{hedge}}]$. Recall that the number of while loop executions is at least one. Hence, we have that $q_{\text{tree}} = \text{TMatch}(d, \text{rtop}+1, \text{lastSib}(\text{rtop}))$, where $\text{rtop} \leqslant q_{\text{hedge}} < q_{\text{tree}} \leqslant \text{lastSib}(\text{rtop})$. The first inequality follows from the fact that $\text{rtop} < \infty$ and the definition of RTop, the second one follows from Goal 1, and the third one from the correctness of TMatch. Hence, we have that

- subhedge $(d) \vDash [q_{\text{from}}, \text{rtop}]$ and
- subtree$(d) \vDash [\text{rtop}+1, q_{\text{tree}}]$.

Moreover, the facts that $\text{rtop}+1$ is a leaf (Observation 4) and $q_{\text{tree}} \leqslant \text{lastSib}(\text{rtop})$ imply that subhedge $(d) \vDash [q_{\text{from}}, q_{\text{tree}}]$.

This concludes the proof the case where subtree(d) matches subtree$(q_{\text{hedge}}+1)$.

This concludes the proof of the case where $q_{\text{hedge}} < q_{\text{until}}$, and also the proof of the case where $q_{\text{tree}} < q_{\text{hedge}}$.

(3) If $q_{\text{hedge}} < q_{\text{tree}}$ the proof is dual to the proof of case (2). □

The correctness of Lemma 5 now follows from Lemmas 6, 7, and 11.

We now argue how TMatch can be modified to a procedure TMatch-All, that computes *all* data nodes u such that $D \vDash^u Q$. In order to compute *all* the matches, we add a test to line 9 of TMatch. That is, before returning q_{best}, we test whether q_{best} is the root of Q, and we output d if it is. Now we return $q_{\text{best}} - 1$, as if the query root was not matched. Furthermore, TMatch-All recursively calls TMatch-All and HMatch-All instead of TMatch and HMatch. Here HMatch-All is the same as HMatch, except that it recursively calls TMatch-All and HMatch-All instead of HMatch and TMatch.

The following theorem can now be proved:

Theorem 8. *Let d be the root node of D and let q_{from} be the smallest and q_{root} be the largest node of Q, respectively.* TMatch-All *is correct, that is,* $\text{TMatch} - \text{All}(d, q_{\text{from}}, q_{\text{until}})$ *outputs the data nodes u such that $D \vDash^u Q$.*

Proof. It follows directly from our additional test and the correctness of TMatch that $D \vDash^u Q$ for all the nodes u that $\text{TMatch} - \text{All}$ outputs.

It remains to prove that, if $D \vDash^u Q$, then TMatch-All outputs u. Towards a contradiction, assume that there is an u such that $D \vDash^u Q$, but u was not reported by $\text{TMatch} - \text{All}$. By an easy induction it can be shown that for every data node d_0 in D there is a call $\text{TMatch} - \text{All}$ for d_0's subtree and Q. In particular, there was a call $\text{TMatch} - \text{All}(u, q_{\text{from}}, q_{\text{root}})$. Since this call did not output u, it follows that u must have children and that $\text{HMatch-All}(\text{lastChild}(u), q_{\text{from}}, q_{\text{root}}) < q_{\text{root}} - 1$, (because otherwise q_{root} and u would have been compared and u would have been written to the output). In general, we have that $\text{HMatch-All}(d, q_1, q_2) = \min((\text{HMatch}(d, q_1, q_2), q_{\text{root}} - 1))$. It then follows that $\text{HMatch-All}(\text{lastChild}(u), q_{\text{from}}, q_{\text{root}}) = \text{HMatch}(\text{lastChild}(u), q_{\text{from}}, q_{\text{root}})$.

If we now call $\text{TMatch}(u, q_{\text{from}}, q_{\text{root}})$, it calls $\text{HMatch}(\text{lastChild}(u), q_{\text{from}}, q_{\text{root}})$, which yields again a value less than $q_{\text{root}} - 1$. Therefore, the return value of $\text{TMatch}(u, q_{\text{from}}, q_{\text{root}})$ is less than q_{root}. But we assumed that subtree$(u) \vDash Q$, which contradicts the correctness of TMatch proved in Lemma 5. □

4.2. Time and space complexity

First, we need to show that our algorithm determines in PTIME whether $D \vDash Q$. Notice that the naïve manner of computing the running time of TMatch gives rise to only an exponential upper bound. Indeed, define (i) $T(N)$ as the running time of TMatch on d, q_{from}, and q_{until}, where subtree(d) and $[q_{\text{from}}, q_{\text{until}}]$ have N nodes in total, and (ii) $H(N)$ as the running time of HMatch on d, q_{from}, and q_{until}, where subhedge (d) and $[q_{\text{from}}, q_{\text{until}}]$ have N nodes in total. Then, we have that $T(2) \leqslant p(N)$ for a polynomial p, $T(N) \leqslant p(N) + H(N-1) + T(N-1)$, and $H(N) \leqslant T(N) + X(N)$, where $X(N) \geqslant 0$. Hence, $T(N) \leqslant 2^{N-1}$, which is obviously not sufficient.

We therefore employ a slightly more sophisticated approach in the following Lemma.

Lemma 12. *Given the root node of a data tree D, and the smallest and largest query nodes and of a query tree Q, respectively,* TMatch *runs in time $\mathcal{O}(|D| \cdot |Q| \cdot \text{depth}(Q))$. Moreover,* TMatch *makes $\mathcal{O}(|D| \cdot |Q|)$ comparisons between a data node and a query node.*

Proof. Let $|D|$ and $|Q|$ be the number of nodes in the data and query tree, respectively. We first show by induction on the height n of the data node d that the number of calls to the function TMatch in the computation tree is at most $|D||Q|$. To this end, we prove three intermediate goals.

Goal 1: *Let d be a leaf data node. A computation of* $\text{TMatch}(d, q_{\text{from}}, q_{\text{until}})$ *yielding result q makes at most* $|[q_{\text{from}}, q+1]|$ *calls to* TMatch.

By induction on the size of the query tree interval $[q_{\text{from}}, q_{\text{until}}]$. If d is a leaf and $q_{\text{from}} = q_{\text{until}}$, then TMatch does not call HMatch recursively and the test on line 7 fails. Therefore, there is only 1 call to TMatch and the induction hypothesis holds. If $q_{\text{from}} < q_{\text{until}}$, and TMatch is not called recursively, then the minimal value we return is $q_{\text{from}} - 1$. Again, there is only 1 call to TMatch and the induction hypothesis holds. Otherwise, we call TMatch on line 8, yielding result q. By induction, the total number of calls to TMatch is at most $1 + |[q_{\text{from}}+1, q+1]|$. As $|[q_{\text{from}}, q+1]| = 1 + |[q_{\text{from}}+1, q+1]|$, the induction holds. This concludes the proof of Goal 1.

Goal 2: *Let d be a data node with height $n > 1$. If the computation of* $\text{HMatch}(\text{lastChild}(d), q_{\text{from}}, q_{\text{until}})$, *yielding the result q^0_{best}, performs at most* $|\text{subhedge}(\text{lastChild}(d))| \cdot |[q_{\text{from}}, q^0_{\text{best}}+1]|$ *calls to* TMatch, *then the computation of* $\text{TMatch}(d, q_{\text{from}}, q_{\text{until}})$, *yielding result q, makes at most* $|\text{subtree}(d)| \cdot |[q_{\text{from}}, q+1]|$ *calls to* TMatch.

We prove Goal 2 by induction on the size of the query tree interval $[q_{\text{from}}, q_{\text{until}}]$. TMatch starts by calling HMatch(lastChild(d), $q_{\text{from}}, q_{\text{until}}$) yielding q^0_{best}. Hence, $|\text{subhedge (lastChild}(d))| \cdot |[q_{\text{from}}, q^0_{\text{best}} + 1]|$ calls to TMatch are performed by this subroutine.

If $q_{\text{from}} = q_{\text{until}}$, then we either return q^0_{best} on line 11 or $q^0_{\text{best}} + 1$ on line 9. In both cases, the number of calls to TMatch is at most $|\text{subhedge (lastChild}(d))| \cdot |[q_{\text{from}}, q^0_{\text{best}} + 1]| + 1$ which is at most $|\text{subtree}(d)| \cdot |[q_{\text{from}}, q^0_{\text{best}} + 1]|$.

If $q_{\text{from}} < q_{\text{until}}$, and TMatch is not called recursively, then the minimal value we return is q^0_{best}. Again, the number of calls to TMatch is at most $1 + |\text{subhedge (lastChild}(d))| \cdot |[q_{\text{from}}, q^0_{\text{best}} + 1]|$ and the induction hypothesis holds. Otherwise, we call TMatch on line 8, yielding result q. By induction, the total number of calls to TMatch is at most $1 + |\text{subhedge (lastChild}(d))| \cdot |[q_{\text{from}}, q^0_{\text{best}} + 1]| + |\text{subtree}(d)| \cdot |[q^0_{\text{best}} + 2, q + 1]|$ which is at most $|\text{subtree}(d)| \cdot |[q_{\text{from}}, q + 1]|$. This concludes the proof of Goal 2.

Goal 3: *Let d be a data node. If the computation of* TMatch(d, q_1, q_2), *yielding* q_{tree} *makes at most* $|\text{subtree}(d)| \cdot |[q_1, q_{\text{tree}} + 1]|$ *calls to* TMatch, *then the computation of* HMatch($d, q_{\text{from}}, q_{\text{until}}$), *yielding* q *makes at most* $|\text{subhedge }(d)| \cdot |[q_{\text{from}}, q + 1]|$ *calls to* TMatch.

Let k be such that d has k left siblings (including d itself). We prove the lemma by induction on k. If $k = 1$, Goal 3 is an immediate consequence from the assumption of Goal 3 and the recursive call of TMatch on line 2. If $k > 1$, then we start by calling HMatch(prevSib(d), $q_{\text{from}}, q_{\text{until}}$), yielding $q^{1,0}_{\text{hedge}}$, and calling TMatch($d, q_{\text{from}}, q_{\text{until}}$), yielding $q^{1,0}_{\text{tree}}$. By induction on k, we have that the call of HMatch induces $|\text{subhedge (prevSib}(d))| \cdot |[q_{\text{from}}, q^{1,0}_{\text{hedge}} + 1]|$ calls to TMatch. Moreover, by the statement of Goal 3, we have that the recursive call of TMatch induces $|\text{subtree}(d)| \cdot |[q_{\text{from}}, q^{1,0}_{\text{tree}} + 1]|$ calls to TMatch in total.

According to Lemma 10, the loops on lines 6, 10, and 18 perform at most a linear number of iterations. Hence, TMatch and HMatch are called (directly) at most a quadratic number of times in the loop.

By $q^{i,j}_{\text{tree}}$, we denote the value of the variable q_{tree} in the ith iteration of the loop and at the end of the jth iteration of the while loop in line 10. Moreover, let ℓ denote the number of loop executions and let $\max_i$ denote the number of executions of the while loop on line 10 in the ith loop execution. Then, we have that every computation of TMatch(d, q_1, q_2) in the while loop performs at most $|\text{subtree}(d)| \cdot |[q^{i,j-1}_{\text{tree}} + 2, q^{i,j}_{\text{tree}} + 1]|$ calls to TMatch when $j > 1$ and at most $|\text{subtree}(d)| \cdot |[q^{i-1,\max_{i-2}(j)}_{\text{tree}} + 2, q^{i,1}_{\text{tree}} + 1]|$ calls otherwise. Notice that $q^{1,0}_{\text{tree}} < q^{1,1}_{\text{tree}} < \cdots < q^{1,\max_1}_{\text{tree}} < q^{2,1}_{\text{tree}} < \cdots < q^{\ell,\max_\ell}_{\text{tree}} \leqslant q$, where q is the value we return. Hence, the sum of the calls to TMatch made by the computations of TMatch on line 11 is at most $|\text{subtree}(d)| \cdot |[q^{1,0}_{\text{tree}} + 2, q + 1]|$.

Analogously, we obtain that the sum of the calls to TMatch by the computations of HMatch on line 19 is at most $|\text{subhedge (prevSib}(d))| \cdot |[q^{1,0}_{\text{tree}} + 2, q + 1]|$.

In total, this means that the number of calls to TMatch is at most

$$\begin{aligned}
&|\text{subhedge (prevSib}(d))| \cdot |[q_{\text{from}}, q^{1,0}_{\text{hedge}} + 1]| \\
&\quad + |\text{subhedge (prevSib}(d))| \cdot |[q^{1,0}_{\text{hedge}} + 2, q + 1]| \\
&\quad + |\text{subtree}(d)| \cdot |[q_{\text{from}}, q^{1,0}_{\text{tree}} + 1]| \\
&\quad + |\text{subtree}(d)| \cdot |[q^{1,0}_{\text{tree}} + 2, q + 1]|
\end{aligned}$$

which is at most $|\text{subhedge }(d)| \cdot |[q_{\text{from}}, q + 1]|$. Hence, Goal 3 follows.

As a consequence of Goals 1, 2, and 3, the total number of calls to TMatch performed by the algorithm is $|D||Q|$. As the only data versus query node comparison in the algorithm occurs in line 5 of TMatch, and as each call of TMatch performs at most one data versus query node comparison (excluding comparisons in recursive calls), the total algorithm also performs at most $|D||Q|$ data versus query node comparisons.

We now argue how this leads us to showing that the overall algorithm has polynomial running time. Consider the entire tree of the calls to TMatch and HMatch in the algorithm, where the children of a node are the functions it calls directly. This computation tree contains at most $|D||Q|$ calls of TMatch. Moreover, every call of HMatch performs at least one direct recursive call to TMatch, so the computation tree also contains at most $|D||Q|$ calls of HMatch. Analogously, the entire computation tree contains at most $|D||Q|$ calls to rtop. As rtop can be implemented to run in time $\mathcal{O}(\text{depth }(Q))$, the total algorithm runs in time $\mathcal{O}(|D||Q|\text{depth }(Q))$. □

The depth (Q) factor in the complexity of TMatch is due to the calls to rtop in HMatch, and the computation of the successors of query nodes.

From the complexity of TMatch and the definition of TMatch-All, we can immediately infer the following complexity results about TMatch-All.

Theorem 9. TMatch − All(D, Q) *runs in time* $\mathcal{O}(|D| \cdot |Q| \cdot \text{depth }(Q))$. *Moreover,* TMatch-All *makes* $\mathcal{O}(|D| \cdot |Q|)$ *comparisons between a data node and a query node.*

Currently, the maximum recursion depth of TMatch − All is $\mathcal{O}(\text{depth }(D) \times \text{branch}(D))$, where branch$(D)$ is the maximum number of children a node in D has. We have the branch(D) factor because HMatch($d, q_{\text{from}}, q_{\text{until}}$) calls HMatch(prevSib($d$), $q_{\text{from}}, q_{\text{until}}$). However, this bound can be improved using a simple preprocessing step: we can turn D into a binary tree D_{bin} by inserting intermediate levels of special nodes between each data node and its children. By doing so, D only grows linearly in size and the depth only grows by a factor of log(branch(D)).

As Q only uses descendant axes, we have that $D \vDash^u Q$ if and only if $D_{\text{bin}} \vDash^u Q$.[4] When this preprocessing step is carried out, our algorithm still has $\mathcal{O}(|D||Q|\text{depth }(Q))$ time

[4] Under the assumption that the new dummy nodes do not match $*$, which can be trivially incorporated in the algorithm.

complexity, but the recursion/stack depth is improved to $\mathcal{O}(\text{depth}(D)\log(\text{branch}(D)))$.

5. Conclusions and final thoughts

As our main results we have exhibited a complexity result, showing that tree pattern matching with only descendant axes is LOGSPACE-complete; and a time- and space-efficient bottom-up algorithm for computing all possible exact matches of such a tree pattern in a tree.

From a theory point of view, this is still only a small step in finding the exact complexity of positive conjunctive Core XPath with only child and descendant axes (or, alternatively, tree pattern queries with child and descendant axes), which is probably the most widely used fragment of XPath in practice. Hence, it is quite surprising that the exact complexity of this fragment is still unknown.

From a practical point of view, our bottom-up algorithm gives a good space and time bound on the processing of such descendant-only tree pattern queries. A minor annoyance we still feel for the algorithm is the depth (Q) factor in the time complexity. However, we need to stress that, in practical applications, depth (Q) will indeed be very small. In our algorithm, this depth (Q) factor arises from computing the $\text{RTop}\,(q_{\text{tree}}, q_{\text{hedge}})$-values in each call of HMatch in the algorithm. It may be possible that this factor can be avoided when integrating the computation of these values in the recursion of the algorithm. For a practical application, one can also avoid the depth (Q) factor in runtime evaluation by a pre-processing step that computes all the values of $\text{RTop}\,(q_{\text{tree}}, q_{\text{hedge}})$ in advance on the query.

Acknowledgment

The third author wants to express his gratitude towards the FWO-Vlaanderen for a scholarship that permitted him to visit Christoph Koch in the Technical University of Vienna in January–February, 2005.

References

[1] M. Altinel, M. Franklin, Efficient filtering of XML documents for selective dissemination of information, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), 2000, pp. 53–64.

[2] Z. Bar-Yossef, M. Fontoura, V. Josifovski, On the memory requirements of XPath evaluation over XML streams, in: Proceedings of the International Symposium on Principles of Database Systems (PODS), 2004, pp. 177–188.

[3] Z. Bar-Yossef, M. Fontoura, V. Josifovski, Buffering in query evaluation over XML streams, in: Proceedings of the International Symposium on Principles of Database Systems (PODS), 2005.

[4] N. Bruno, D. Srivastava, N. Koudas, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2002, pp. 310–321.

[5] C.Y. Chan, W. Fan, P. Felber, M.N. Garofalakis, R. Rastogi, Tree pattern aggregation for scalable XML data dissemination, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), 2002, pp. 826–837.

[6] C.Y. Chan, P. Felber, M.N. Garofalakis, R. Rastogi, Efficient filtering of XML documents with XPath expressions, in: Proceedings of the International Conference on Data Engineering (ICDE), 2000, pp. 235–244.

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

[8] S.A. Cook, P. McKenzie, Problems complete for deterministic logarithmic space, Journal of Algorithms 8 (1987) 385–394.

[9] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, P. Fischer, Path sharing and predicate evaluation for high-performance XML filtering, ACM Transactions on Database Systems 28 (4) (2003) 467–516.

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

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

[12] G. Gottlob, N. Leone, F. Scarcello, The complexity of acyclic conjunctive queries, Journal of the ACM 48 (1) (2001) 431–498.

[13] M. Götz, C. Koch, W. Martens, Efficient algorithms for the tree homeomorphism problem, in: Proceedings of the International Symposium on Database Programming Languages (DBPL), 2007, pp. 17–31.

[14] T.J. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata and stream indexes, ACM Transactions on Database Systems 29 (4) (2004) 752–788.

[15] M. Grohe, C. Koch, N. Schweikardt, Tight lower bounds for query processing on streaming and external memory data, in: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), 2005.

[16] A. Gupta, D. Suciu, Stream processing of XPath queries with predicates, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003, pp. 419–430.

[17] D. Olteanu, T. Furche, F. Bry, An evaluation of regular path expressions with qualifiers against XML streams, in: Proceedings of the British National Conference on Databases (BNCD), 2004, pp. 31–44.

[18] P. Ramanan, Evaluating an XPath query on a streaming XML document, in: Proceedings of the International Conference on Management of Data (COMAD), 2005, pp. 41–52.

[19] I.H. Sudborough, Time and tape bounded auxiliary pushdown automata, in: Proceedings of the Mathematical Foundations of Computer Science (MFCS), Springer, Berlin, 1977, pp. 493–503.

[20] M. Yannakakis, Algorithms for acyclic database schemes, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), 1981, pp. 82–94.