Incremental XPath Evaluation

HENRIK BJÖRKLUND
Umeå University, Department of Computing Science
and
WOUTER GELADE
Hasselt University and Transnational University of Limburg, School for Information Technology, Belgium
and
WIM MARTENS
Technical University of Dortmund, Department of Computer Science

Incremental view maintenance for XPath queries asks to maintain a materialized XPath view over an XML database. It assumes an underlying XML database D and a query Q. One is given a sequence of updates U to D and the problem is to compute the result of $Q(U(D))$, i.e., the result of evaluating query Q on the database D after having applied the updates U. This paper initiates a systematic study of the boolean version of this problem. In the boolean version, one only wants to know wheter $Q(U(D))$ is empty or not.

In order to quickly answer this question, we are allowed to maintain an auxiliary data structure, and the complexity of the maintenance algorithms is measured in (i) the size of the auxiliary data structure, (ii) the worst-case time per update needed to compute $Q(U(D))$ and (iii) the worst-case time per update needed to bring the auxiliary data structure up to date. We allow three kinds of updates: node insertion, node deletion, and node relabeling. Our main results are that downward XPath queries can be incrementally maintained in time $\mathcal{O}(\text{depth}(D) \cdot \text{poly}(|Q|))$ per update and conjunctive forward XPath queries in time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(|Q|))$ per update, where $|Q|$ is the size of the query, and $\text{depth}(D)$ and $\text{width}(D)$ are the nesting depth and maximum number of siblings in the database D, respectively. The auxiliary data structures for maintenance are linear in $|D|$ and polynomial in $|Q|$ in all these cases.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; F.4.3 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms: Algorithms, Design, Languages, Standardization, Theory

Additional Key Words and Phrases: XML, XPath, view maintenance

The present paper is the full version of [Björklund et al. 2009], which was presented at the 12th International Conference on Database Theory (ICDT) in St. Petersburg, Russia, 2009.

1. INTRODUCTION

The XPath language, proposed by the World Wide Web Consortium (W3C), is essentially a query language for selecting nodes in an XML document. Node-selection is one of the most basic operations on XML documents and therefore XPath lies at the core of most of today's data processing languages for XML. For example, it forms an essential component of languages such as XQuery, XSLT, XML Schema (which uses XPath for defining keys), etc.

The most fundamental algorithmic question concerning XPath is *query evaluation*. That is, given an XPath query Q and an XML document D, return all nodes that are selected by Q in D. The query evaluation problem for various fragments of XPath has been researched quite intensely over the last decade (see [Benedikt and Koch 2008] for an overview). In this paper, we are interested in the *incremental XPath evaluation* problem. We first explain the general idea behind incremental XPath evaluation and discuss the variant that we study later. In incremental XPath evaluation one is given an XPath query Q and XML data D. Furthermore, we assume that the answer for Q on D is already known. However, when D is updated to D', we want to be able to infer the updated answer for Q on D' as quickly as possible. The idea is, of course, to maintain extra information such that the updated answer to Q on D' can be computed without having to re-evaluate Q from scratch.

We provide two motivating scenarios for incremental XPath evaluation in general:

(A). Trigger conditions. A database system has a trigger condition in which the precondition is stated by means of an XPath query. Here, the system may be interested in knowing very quickly after an update whether the event of the trigger needs to be carried out or not.

(B). View maintenance. A database system has a certain view definition, formulated as an XPath query, and we simply want to maintain its results after updates.

Notice that a scenario similar to (B) can also be relevant when exchanging data on the Web. When a community exchanges data, it is often the case that a certain user Y is interested in the result of a fixed query on the data of another user X. To make the example more concrete, it may be useful to think of X as a huge bioinformatics database and of Y as a research group that is only interested in a particular set of sequences. We also assume that X knows Y's view definition and wants to keep Y up-to-date (i.e., Y has a query subscription). The data of X may change often, while the interests of Y remain the same. Of course, it could be beneficial for both parties if Y's view definition does not have to be completely recomputed and the entire result sent over the Internet every time X's data changes. In such a setting, Y may herself have a representation of the current result of her query in order to do local research and analysis. So, after an update of X's data, it would be relevant for X to be able to quickly determine the *changes* that Y has to make to her old result, rather than sending her the complete new result, which may be much larger than the update. More concretely, say that D_{old} (resp. D_{new}) is the database of X before (resp. after) an update and V_{old} and V_{new} are the results of Y's view on D_{old} and D_{new}, respectively. Since Y has V_{old} locally, which needs to be updated,

Table I. Overview of our results. The second column indicates the XPath fragment under condideration and the third column reports the obtained upper bounds for incremental evaluation. The time complexities are measured in terms of the XML document D and query Q and are worst-case times *per update*. AuxSize is the size of the auxiliary datastructure that has to be maintained. The last column refers to the theorems in which the results are proved. All results are exclusively on *Boolean* maintenance, except for row (3).

	Fragment	**Complexity**	**Theorem**
(1)	NavXPath	Time: $\mathcal{O}(\log^2(\|D\|)) \cdot 2^{\mathcal{O}(\|Q\|)}$ AuxSize: $\mathcal{O}(\|D\|) \cdot 2^{\mathcal{O}(\|Q\|)}$	Theorem 3.2
(2)	NavXPath	Time: $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D))) \cdot 2^{\mathcal{O}(\|Q\|)}$ AuxSize: $\mathcal{O}(\|D\|) \cdot 2^{\mathcal{O}(\|Q\|)}$	Theorem 3.2
(3)	$\downarrow, \Downarrow, \wedge, \vee, \neg, [\cdot], *$	Time: $\mathcal{O}(\text{depth}(D) \cdot \|Q\|)$ AuxSize: $\mathcal{O}(\|D\| \cdot \|Q\|)$	Theorem 4.1
(4)	$\rightarrow, \Rightarrow, \wedge, [\cdot], *$	Time: $\mathcal{O}(\log(\|D\|) \cdot \|Q\|^6)$ AuxSize: $\mathcal{O}(\|D\| \cdot \|Q\|^3)$	Theorem 5.1
(5)	$\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge, [\cdot], *$	Time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \|Q\|^7)$ AuxSize: $\mathcal{O}(\|D\| \cdot \|Q\|^3)$	Theorem 6.1

it may be much more beneficial for X to send a list of update(...) / delete(...) instructions to Y that update V_{old} to V_{new}, rather than sending over the possibly huge V_{new} to Y.

1.1 Two Versions of the Problem

Incrementally evaluating queries on a relational database is an intensively researched topic in database theory. In the literature, it is also known as *incremental view maintenance* (see, e.g., [Shmueli and Itai 1984; Gupta et al. 1993]).

From our two motivating scenarios above, we can immediately infer two versions of the incremental XPath evaluation problem that we believe to be important in practice. We will refer to the first as the *Boolean maintenance* version, and to the second as the *(materialized) view maintenance* version. In brief, in the Boolean maintenance version, which corresponds to scenario *(A)*, we are simply interested in whether an XPath expression is satisfied or not after performing an update on the database. In the view maintenance version, which corresponds to scenario *(B)*, the output of the XPath query is maintained and, after an update of the database, we want to update this set of outputs.

In this paper we focus almost exclusively on the Boolean maintenance version of the problem. View maintenance is clearly more general, but it turns out that the Boolean maintenance version is algorithmically already quite challenging. We believe that the Boolean maintenance version is interesting in its own right and that it already contains an important core of the view maintenance version, in the sense that the difficulties of the Boolean maintenance version need to be understood before view maintenance can be properly tackled.

1.2 Our Contributions

We allow the algorithms for incremental XPath evaluation to maintain an auxiliary data structure to help re-evaluation. The cost of the algorithms in this article are then measured in terms of *(a)* the time needed to recompute the result of XPath query, *(b)* the time needed to update the auxiliary data structure, and *(c)* the size

of the auxiliary data structure.[1] The time measures are worst-case bounds for *a single update* to the XML document, and the updates we consider are of the form: relabel node x, delete the subtree rooted at x, or insert a node x in position y. As our model of computation, we assume a RAM-model that can store, e.g., counter values of size $\mathcal{O}(|D| + |Q|)$ in constant space (i.e., one register).

Our main results are summarized in Table I. Here, $|D|$ and $|Q|$ are the sizes of the XML document and the XPath query, respectively. We consider XPath queries without operations on data values. In the terminology of [Benedikt and Koch 2008], we consider fragments of *Navigational XPath (NavXPath)*.[2] In particular, our study focuses on features in XPath 1.0 rather than XPath 2.0. The results in the table concern Boolean incremental evaluation. The time complexities hold for recomputing the result of the XPath query and updating the auxiliary data structure, and hold for all the updates we consider. Some of the time complexities contain a factor $\text{depth}(D)$ or $\text{width}(D)$ which denote the depth of the tree representation of D and the maximum number of siblings in D, respectively. We believe the $\text{depth}(D)$ factor in row (3) to be very relevant for practical purposes, since the depth of D is extremely small in practice, especially when compared to the full size of D. The other rows denote the fragments of NavXPath that only allow the listed operators and axes. Here, $\downarrow$, $\Downarrow$, $\rightarrow$, $\Rightarrow$ denote the axes child, descendant, nextsibling,[3] and following-sibling, respectively. Predicate is denoted by $[\cdot]$, wildcard by $*$, and the Boolean operators by $\wedge$, $\vee$, and $\neg$, respectively.

In case (3) in the table, we can also do view maintenance for the query in a restricted form. Essentially, we can maintain the materialized view consisting of the set of the nodes where the root of the query matches D. The intuitive reason why this restricted form of materialized view maintenance can be handled is that all positions where the truth value of the root of the query changes lie on the path from the change in D to its root.

Finally, we want to bring the incremental XPath evaluation problem to the attention of the database community. To the best of our knowledge, this is the first paper that provides (sub-linear) worst-case upper bounds on incremental XPath evaluation.

1.3 Related work

The (non-incremental) XPath evaluation problem has been studied quite extensively in the literature [Bojańczyk and Parys 2008; Gottlob et al. 2005; Gottlob et al. 2005; Götz et al. 2009; Parys 2009]. We refer to [Benedikt and Koch 2008] for a detailed overview. There is a large amount of work on indexing on XML documents (e.g, [O'Neil et al. 2004]), but indexing schemes are usually aimed towards answering a large class of XPath queries and require time at least linear in the document for complicated queries.

There are already several papers dealing with incremental XPath evaluation [Matsumura and Tajima 2005; Onizuka et al. 2005; Sawires et al. 2006; Sawires et al.

[1] For a more formal treatment, see Section 2.2.

[2] The XPath fragment we consider is formally defined in Section 2.1.

[3] Next-sibling is strictly speaking not a primitive axis in XPath, but can be expressed using following-sibling::$*$[position() = 1].

2005], but none of these papers give any worst-case complexity bounds. The papers [Matsumura and Tajima 2005; Sawires et al. 2005] only consider leaf deletion and insertion, and [Onizuka et al. 2005] considers deletion and insertion of entire subtrees.

An early paper on pattern matching in trees is by Hoffmann and O'Donnell [Hoffmann and O'Donnell 1982]. They studied pattern matching in trees and also treat incremental maintenance, but their setting is rather different from ours. Hoffmann and O'Donnel's work is motivated by problems related to program interpreters, code optimization in compilers, and automated theorem proving. In particular, this means that they only consider queries with the child-relation, which should be injectively mapped onto ranked trees. Our queries are more general, do not have injective semantics, and apply to unranked trees. The results on incremental maintenance in [Hoffmann and O'Donnell 1982] is essentially a by-product of their preprocessing of the patterns, which is similar to the Knuth-Morris-Pratt string matching algorithm. Their complexity results are not comparable to ours, since they spend more time on preprocessing and have a higher complexity with respect to the tree size but a lower complexity with respect to the query size.

The incremental validation of XML schemas [Balmin et al. 2004; Barbosa et al. 2004] is closely related to our problems. In incremental schema evaluation, one is asked to maintain satisfaction of an XML document by an XML schema, where the document can be updated. Balmin et al. describe algorithms for incremental validation of DTDs, XML Schemas and tree automata [Balmin et al. 2004]. We use their incremental maintenance result for tree automata to infer some upper bounds in this paper.

Incremental view maintenance for *Active XML* has been studied by [Abiteboul et al. 2007; 2009]. Abiteboul et al. investigate the complexity of deciding whether there exists a possible update sequence to an active XML document that satisfies a query and note that incremental maintenance is possible by directly using known techniques for Datalog. They also provide an implementation of the maintenance algorithm.

Incremental XPath evaluation can be seen as a generalization of the XPath evaluation problem on XML streams (see, e.g., [Bar-Yossef et al. 2007; Grohe et al. 2007; Schweikardt 2007]). In streaming XPath evaluation, one reads the XML document as a sequence of SAX-events, i.e., the sequence of opening and closing tags in the ordering in which they occur in the XML file. When viewing an XML document as a tree, this ordering corresponds to the depth-first left-to-right ordering of the tree. Streaming XPath evaluation can then be seen as incremental XPath evaluation in which the only update operation is that nodes can be added at the last position in the depth-first left-to-right ordering.

Our paper investigates upper time complexity bounds for incremental XPath evaluation. We do not know any non-trivial lower bound on the time complexity of this problem. However, since incremental XPath evaluation generalizes XPath evaluation on streams, any time lower bound on XPath evaluation on streams *for reading a single tag* would also be a time lower bound per update for incremental XPath evaluation. A related problem is the inherent dynamic complexity of formal languages, which was investigated in, e.g., [Gelade et al. 2009].

Incremental view maintenance is a deeply investigated topic in relational and object-relational databases. However, the focus of this work was different than ours. Rather than focusing on dedicated algorithms for updating a view in sublinear time, this research has mostly investigated questions such as *is it possible to maintain views expressed in language $\mathcal{L}_1$ by an algorithm expressible as a query in language $\mathcal{L}_2$* [Dong and Su 2000]. The goal is of course to obtain a language $\mathcal{L}_2$ which can be evaluated very quickly and which is well supported by the database system. In this sense, Dong and Topor investigate incremental evaluation of nonrecursive datalog queries over relational data [Dong et al. 1995]. Griffin and Libkin studied maintaining relational views with duplicates [Griffin and Libkin 1995]. Incremental maintenance of *nested* relational views is investigated in, e.g., [Libkin and Wong 1997; Liu et al. 1999]. Gupta et al. developed a method for maintaining nonrecursive materialized views defined in SQL or Datalog [Gupta et al. 1993]. The algorithm is based on keeping track of the number of possible derivations of tuples. More recently, Dong et al. investigated properties of query languages that imply *un*maintainability of recursive relational views [Dong et al. 2003]. Unmaintainability refers to being unable to formulate the updates in relational calculus. The algebraic perspective of relational incremental view maintenance has been studied by Koch [Koch 2010].

Maintaining transitive closure of graphs using SQL has been investigated by Dong et al. [Dong et al. 1999]. Pang et al. continued this work and provide algorithms expressible in FO(+,<) for maintaining transitive closure and all-pairs shortest-distance on weighted graphs [Pang et al. 2005].

2. PRELIMINARIES

By Σ we always denote an infinite set of labels. Our abstraction of an XML document is a rooted, ordered, finite, labeled, unranked tree, which is 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. We view an XML document D as 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 next-sibling$(\cdot,\cdot)$. Here, $a(u)$ expresses that u is a node with label a, and child(u,v) (respectively, next-sibling(u,v)) expresses that v is a child (respectively, next sibling) of u. We also use the notations descendant(u,v) and following-sibling(u,v) for the respective transitive closures of the above relations. The label of a node u in D must be unique and is denoted by $\text{lab}^D(u)$. We write Nodes(D) and Edges(D) for the sets of nodes and edges of a tree (document) D. As usual, Edges(D) is the set of pairs (u,v) such that child(u,v) holds in D. The root of D is denoted by root(D). We define the *size of* D, denoted by $|D|$, to be the number of nodes of D.

Notice that we have an infinite set of labels from which our (finite) trees can choose. This reflects how trees occur in an XML-context: an XML tree is a finite structure, but there is no restriction on how it should be labeled (if no schema is provided).

2.1 XPath Patterns

We assume that the reader is familiar with XPath [Clark and DeRose 1999]. As an abstraction of XPath, we will usually use *XPath Patterns*, which we formally

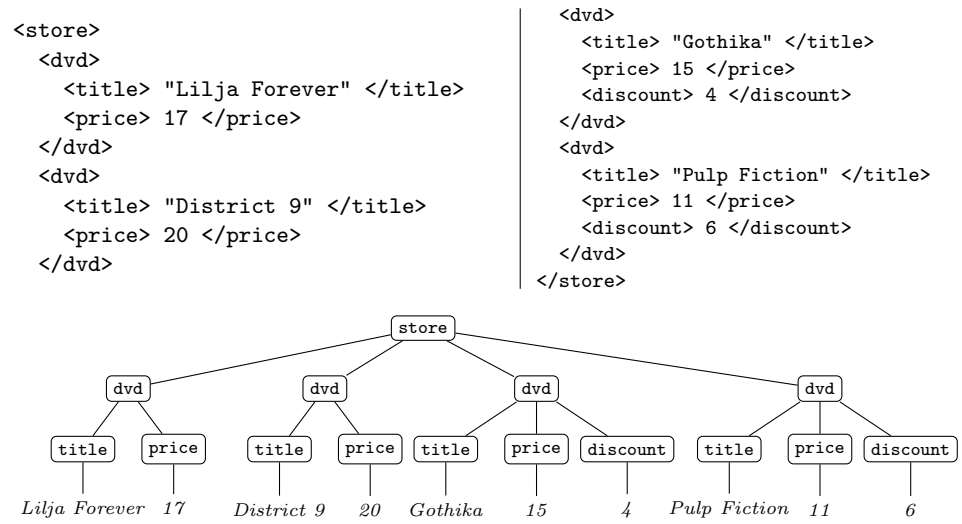


Fig. 1. An example of an XML document and its tree representation.

define in this section. They can be seen as an extension of *tree pattern queries* (see, e.g., [Miklau and Suciu 2004; Götz et al. 2009]) with the full expressive power of *navigational XPath* [Benedikt and Koch 2008]. Essentially, XPath Patterns are parse trees of XPath 1.0 queries. They are a convenient technical tool for our algorithms since they allow us to reason about *nodes* and *edges* in the pattern. Following XPath, our XPath Patterns will make use of *axes*. The axes in this article are fairly standard: self, child ($\downarrow$), descendant ($\Downarrow$), descendant-or-self ($\downarrow^*$), parent ($\uparrow$), ancestor ($\Uparrow$), ancestor-or-self ($\uparrow^*$), next-sibling ($\rightarrow$), following-sibling ($\Rightarrow$), previous-sibling ($\leftarrow$), and preceding-sibling ($\Leftarrow$). We note that the remaining XPath axes (i.e., following and preceding) can be expressed by these axes using only a linear blow-up.

XPath Patterns are parse trees of XPath 1.0 queries. We formally define them as follows.

Definition 2.1. An *XPath Pattern* is a rooted, unranked, unordered, finite, labeled tree in which the nodes and edges bear *types*. The type of a node u, denoted by $\text{type}(u)$ can either be *label* or *syntax*. When the type of a node is *label* then the label must be in $\Sigma \uplus \{*\}$. When the type of a node is *syntax*, the label must be one of $\wedge, \vee, \neg$. The type of an edge e, denoted by $\text{type}(e)$, can be *syntax*, or any XPath axis.

Figure 2(a) contains an illustration of an XPath pattern. We only use two types of axes in Figure 2(a): the child axis (depicted by single arrows) and the descendant axis (depicted by double arrows).

Remark 2.2. Throughout the paper, we will use the letter D to denote the XML document, and Q to denote the XPath Pattern. We will refer to nodes of Q as *query nodes* and to nodes of D as *document nodes*. When the type of $u \in \text{Nodes}(Q)$ is *label* (resp. *syntax*), we refer to u as a *label node* (resp. *syntax node*). Similarly for edges, we also use the terminology X-edge for an edge e with $\text{type}(e) = X$, e.g.,

child-edge, self-edge,... For a set X of axes and boolean operators, we denote by XPath(X) the set of XPath Patterns using only the axes and operators from X.

We assume that XPath Patterns are *well-formed*, that is, *(i)* all incoming edges to syntax nodes must be syntax edges; *(ii)* no other edges are syntax edges; and *(iii)* a syntax node labeled $\neg$ has only one child.

We define the semantics of XPath Patterns inductively on the structure of the pattern. Given a document D, a document node $u \in \text{Nodes}(D)$ and an XPath Pattern Q, we will, for each query node $i \in \text{Nodes}(Q)$ and each edge $e \in \text{Edges}(Q)$, define the two notions $D \models^u Q[i]$ and $D \models^u Q[e]$. Loosely speaking these notions express that the subpattern (i.e., subtree) of Q that is rooted at node i (edge e, resp.) is satisfied in D at node u. We say that a query node $i \in \text{Nodes}(Q)$ *matches* a document node $u \in \text{Nodes}(D)$ if i is a label node and either $\text{lab}^Q(i) = *$ or $\text{lab}^Q(i) = \text{lab}^D(u)$.

We say that $D \models^u Q[i]$ when one of the following conditions holds:

—query node i matches document node u and, for all edges $e = (i, j)$, it holds that $D \models^u Q[e]$;
—$\text{lab}^Q(i) = \wedge$ and, for all edges $e = (i, j)$, it holds that $D \models^u Q[e]$;
—$\text{lab}^Q(i) = \vee$ and there exists an edge $e = (i, j)$ such that $D \models^u Q[e]$; or
—$\text{lab}^Q(i) = \neg$ and, for the unique edge $e = (i, j)$, we have that $D \models^u Q[e]$ does not hold.

Moreover, for each edge $e = (i, j)$ in Q, we say that $D \models^u Q[e]$ when one of the following holds:

—e is a syntax edge or $\text{type}(e) = \text{self}$, and $D \models^u Q[j]$; or
—$\text{type}(e) = \downarrow$ (resp., $\Downarrow$, $\downarrow^*$, $\uparrow$, $\Uparrow$, $\uparrow^*$, $\rightarrow$, $\Rightarrow$, $\leftarrow$, $\Leftarrow$) and there exists a child v of u (resp., descendant, descendant-or-self, parent, ancestor, ancestor-or-self, next sibling, following sibling, previous sibling, preceding sibling v of u) such that $D \models^v Q[j]$.

Finally we say that the document D models the XPath Pattern Q ($D \models Q$) iff $D \models^{\text{root}(D)} Q[\text{root}(Q)]$. We also abbreviate $D \models^u Q[\text{root}(Q)]$ with $D \models^u Q$. If $D \models Q$ holds, we also sometimes write that D *satisfies* Q.

REMARK 2.3. In the above, we define $D \models Q$ to mean that there is a matching of the query pattern that matches the root of the pattern to the root of the document. In some situations, however, matchings that map the root of the pattern to other nodes of the document may be of interest. This can be achieved by modifying the query pattern slightly. If we add a label-node with label $*$, make it the new root of the query pattern, and add a single descendant-edge from the new node to the previous root of the pattern, we obtain a query pattern Q' such that $D \models Q'$ if and only if $D \models^u Q$ for some document node u other than root(D).

Figure 2(a) illustrates an example of an XPath Pattern that is satisfied in the same set of trees as the XPath query $/a[.//b \text{ and } ./e \text{ and}(\text{not}(./c))]/*//d$. Figure 2(b) shows a document tree satisfying the XPath Pattern from Figure 2(a).

REMARK 2.4. Navigational XPath [Benedikt and Koch 2008] also allows the use of the (binary) union operator, but with respect to the questions in this paper, this

(a) XPath Pattern for the query $/a[.//b \text{ and } ./e \text{ and } (\text{not}(./c))]/*//d$.

(b) A document modeling the XPath Pattern of Figure 2(a).

Fig. 2. An XPath Pattern and a document.

union operator is equivalent to the disjunction in XPath patterns. We chose not to add union explicitly as, in our setting, this restricted union operator can simply be simulated by the $\vee$ operator without blow-up.

2.2 The Incremental Evaluation Problem

Our formal treatment of the incremental evaluation problem for XPath Patterns is similar to the one of Balmin et al. for incremental validation of XML schemas [Balmin et al. 2004].

We first formally define the *Boolean incremental XPath evaluation problem*. That is, given an XPath Pattern Q, an XML document D, the knowledge whether $D \models Q$ or not, and an update to D yielding another XML document D', we wish to efficiently check if $D' \models Q$. In particular, the cost should be less than evaluating Q on D' from scratch. The individual updates are the following:

(a) replace the current label of a specified node by another label,

(b) insert a new leaf node as the next sibling of a specified node,

(c) insert a new leaf node as the first child of a specified node, and

(d) delete a specified node; if the node is an internal one, the subtree of D rooted at the node is also deleted.

It should be noted that in other work it is sometimes also allowed to insert entire subtrees into the document, instead of single nodes. However, as the above updates allow to insert nodes at any position in the document tree, this can be accomodated in our framework by inserting the nodes of the subtree one by one.

We allow some cost-free one-time pre-processing, such as computing an automaton representation of a pattern. We will also initialize and then maintain an auxiliary structure $\text{aux}(Q, D)$ to help in the evaluation. The cost of the incremental evaluation algorithm is given w.r.t.:

(a) the time needed to test whether $D' \models Q$ using D and $\text{aux}(Q, D)$, as a function of $|D|$ and $|Q|$,

(b) the time needed to compute $\text{aux}(D')$ from D and $\text{aux}(Q, D)$, as a function of $|D|$ and $|Q|$,

(c) the size of the auxiliary structure $\text{aux}(Q, D)$ as a function of $|D|$ and $|Q|$.

As mentioned in the Introduction, the time complexities in Table I always hold for both (a) and (b).

3. FULL NAVIGATIONAL XPATH

We start with an approach to Boolean incremental evaluation for full XPath Patterns. It builds heavily on well-known techniques for translating XPath into finite-state tree automata (see, e.g., [Schwentick 2004; ten Cate and Lutz 2009]).

The robust notion of regular string and ranked tree languages can easily be generalized to unranked tree languages. The latter class is usually defined in terms of non-deterministic unranked tree automata and posseses similar closure properties. The class of tree languages accepted by unranked tree automata is called the *unranked regular tree languages*. We refer the unfamiliar reader to [Neven 2002] for a gentle introduction. We formally introduce non-deterministic tree automata (Definition A.4) and prove the following Lemma in Appendix A. For an automaton A let $L(A)$ denote the set of trees accepted by A.

THEOREM 3.1. *Let Q be an XPath Pattern. A non-deterministic unranked tree automaton A with $L(A) = \{D \mid D \models Q\}$ can be constructed in time $2^{\mathcal{O}(|Q|)}$.*

We noticed that Theorem 3.1 was independently discovered by Libkin and Sirangelo (Theorem 2 in [Libkin and Sirangelo 2008]).[4] It was already known that standard constructions allowed to construct A in time $2^{\mathcal{O}(\text{poly}(|Q|))}$. The emphasis of Theorem 3.1 is that $2^{\mathcal{O}(|Q|)}$ suffices.

Balmin et al. [Balmin et al. 2004] have shown that given an unranked tree automaton A one can incrementally decide membership of an XML document D in $L(A)$ in time[5] either $\mathcal{O}(\log^2(|D|)\cdot\text{poly}(|A|)$ or $\mathcal{O}(\text{depth}(D)\cdot\log(\text{width}(D))\cdot\text{poly}(A))$, with respect to the same update operations we consider. The size of the auxiliary data structure is $\mathcal{O}(|D|)\cdot\text{poly}(|A|)$. The basic building block of the technique used in [Balmin et al. 2004] is sketched in Section 5.1. This immediately implies the following.

THEOREM 3.2. *Boolean incremental evaluation for an XPath Pattern Q and an XML document D can be performed in*

(1) time $\mathcal{O}(\log^2(|D|)\cdot 2^{\mathcal{O}(|Q|)})$ per update; or
(2) time $\mathcal{O}(\mathit{depth}(D)\cdot\log(\mathit{width}(D))\cdot 2^{\mathcal{O}(|Q|)})$ per update;

both with an auxiliary data structure of size $|D|\cdot 2^{\mathcal{O}(|Q|)}$.

4. DOWNWARD XPATH

As seen in the previous section, an automata-theoretic approach combined with the results from [Balmin et al. 2004] yields a maintenance algorithm that is polylogarithmic in the document and exponential in the query size, with auxiliary data which is linear in the document and exponential in the query. This may work as

[4]The paper [Libkin and Sirangelo 2008] appeared when we prepared the camera-ready version of [Björklund et al. 2009].
[5]Actually, they state a complexity bound of $\mathcal{O}(\text{depth}(D)\cdot\log(|D|)\cdot\text{poly}(|A|))$, but a slightly more detailed analysis shows that the complexity is also $\mathcal{O}(\text{depth}(D)\cdot\log(\text{width}(D))\cdot\text{poly}(|A|))$.

long as the query is very small, but for larger queries, the complexity becomes prohibitive.

4.1 Boolean Incremental Evaluation

In this section, we present a first maintenance algorithm that is also polynomial in the query. In particular, we provide an algorithm for incrementally maintaining a downward XPath pattern, i.e., an XPath$(\downarrow, \Downarrow, \wedge, \vee, \neg)$-pattern. We show the following result:

THEOREM 4.1. *Boolean incremental evaluation for an XPath$(\downarrow, \Downarrow, \wedge, \vee, \neg)$ Pattern Q and XML document D can be performed in time $\mathcal{O}(depth(D) \cdot |Q|)$ per update. The size of the auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|)$.*

The algorithm works as follows. For each node u in D, with children $u_1, \ldots, u_k$, we store a record R_u consisting of

—the set of query nodes $\text{Match}(u) := \{q \in \text{Nodes}(Q) \mid D \models^u Q[q]\}$,
—for every query node q, the number of children of u in which $Q[q]$ is satisfied, i.e., the cardinality $\text{numChild}^q(u)$ of the set $\{u_i \mid q \in \text{Match}(u_i)\}$,
—the set of query nodes that are satisfied in some descendant of u, i.e., the set $\text{MatchDesc}(u) := \{q \in \text{Nodes}(Q) \mid \exists u' \in \text{Nodes}(D)$ such that $\text{descendant}(u, u') \wedge D \models^{u'} Q[q]\}$, and
—for every query node q, the number of children of u that have a descendant satisfying $Q[q]$, i.e, the cardinality $\text{numDesc}^q(u)$ of the set $\{u_i \mid q \in \text{MatchDesc}(u_i)\}$.

Hence, D satisfies Q if and only if $\text{root}(Q)$ is in $\text{Match}(\text{root}(D))$. So, once the auxiliary data structure is computed, testing whether $D \models Q$ is trivial. The size of each record R_u is $\mathcal{O}(|Q|)$, so the size of the entire auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|)$.

It now suffices to show that we can incrementally update the auxiliary data structure in time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$. Notice that, for each of the updates of a node u in D (label change, node insertion, and node deletion), only the data records R_v for nodes v on the path from u to the root of D change. We recompute these data records in a bottom-up fashion.

Suppose that we performed an update at some node in D and let u be the next node on the path to the root for which the record must be updated, let v be its parent node, and $u_1, \ldots, u_k$ its children. When visiting u we assume that the updated values $\text{numChild}^q_{\text{new}}(u)$ and $\text{numDesc}^q_{\text{new}}(u)$, and $\text{Match}_{\text{new}}(u_i)$ and $\text{MatchDesc}_{\text{new}}(u_i)$, for all $q \in \text{Nodes}(Q)$ and $i \in \{1, \ldots, k\}$ are known, and we compute $\text{Match}_{\text{new}}(u)$, $\text{MatchDesc}_{\text{new}}(u)$, $\text{numChild}^q_{\text{new}}(v)$, and $\text{numDesc}^q_{\text{new}}(v)$. If u is a leaf, we have $\text{numChild}^q_{\text{new}}(u) = \text{numDesc}^q_{\text{new}}(u) = 0$ for all q. The recomputation works as follows:

(1) $\text{Match}_{\text{new}}(u)$: We compute the set $\text{Match}_{\text{new}}(u)$ by inspecting Q in a bottom-up fashion.
In Q, we have *syntax* and *label* nodes, and for the latter we distinguish *child* and *descendant* nodes depending on whether the incoming edge is a child or descendant edge. We will call a node q of Q *satisfied (w.r.t. node u in D)* if (a) q is a syntax node and $q \in \text{Match}_{\text{new}}(u)$, (b) q is a child node and

$\text{numChild}^q_{\text{new}}(u) > 0$, or (c) q is a descendant node and $\text{numChild}^q_{\text{new}}(u) > 0$ or $\text{numDesc}^q_{\text{new}}(u) > 0$.

Now, let q be a query node with children $q_1, \ldots, q_\ell$. Then, $q \in \text{Match}_{\text{new}}(u)$ if (a) q is a label node, the label of q matches the label of u, and all children $q_1, \ldots, q_\ell$ are satisfied w.r.t. u; (b) q is a syntax node labeled "$\wedge$" and all $q_1, \ldots, q_\ell$ are satisfied w.r.t. u; (c) q is a syntax node labeled "$\vee$" and at least one of $q_1, \ldots, q_\ell$ is satisfied w.r.t. u; or (d) q is a syntax node labeled "$\neg$" and its (unique) child is not satisfied w.r.t. u.

From these conditions, we can easily compute $\text{Match}_{\text{new}}(u)$ in time $\mathcal{O}(|Q|)$ in one bottom-up pass over Q.

(2) $\text{numChild}^q_{\text{new}}(v)$: For every $q \in \text{Nodes}(Q)$,

$$\text{numChild}^q_{\text{new}}(v) = \begin{cases} \text{numChild}^q(v) + 1 & \text{if } q \in \text{Match}_{\text{new}}(u) \text{ and } q \notin \text{Match}(u) \\ \text{numChild}^q(v) - 1 & \text{if } q \notin \text{Match}_{\text{new}}(u) \text{ and } q \in \text{Match}(u) \\ \text{numChild}^q(v) & \text{otherwise.} \end{cases}$$

(3) $\text{MatchDesc}_{\text{new}}(u)$:

$$\text{MatchDesc}_{\text{new}}(u) = \{q \mid \text{numChild}^q_{\text{new}}(u) > 0\} \cup \{q \mid \text{numDesc}^q_{\text{new}}(u) > 0\}$$

(4) $\text{numDesc}^q_{\text{new}}(v)$: For every $q \in \text{Nodes}(Q)$,

$$\text{numDesc}^q_{\text{new}}(v) = \begin{cases} \text{numDesc}^q(v) + 1 & \text{if } q \in \text{MatchDesc}_{\text{new}}(u) \\ & \qquad \text{and } q \notin \text{MatchDesc}(u) \\ \text{numDesc}^q(v) - 1 & \text{if } q \notin \text{MatchDesc}_{\text{new}}(u) \\ & \qquad \text{and } q \in \text{MatchDesc}(u) \\ \text{numDesc}^q(v) & \text{otherwise.} \end{cases}$$

This algorithm requires time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$. Indeed, we only have to update $\text{depth}(D)$ records, each of which can be done in time $\mathcal{O}(|Q|)$. This proves Theorem 4.1.

4.2 Beyond Boolean Incremental Evaluation

For an XPath$(\downarrow, \Downarrow, \wedge, \vee, \neg)$-pattern Q, the membership of nodes in the set $V := \{u \in \text{Nodes}(D) \mid D \models^u Q\}$ only changes for nodes on the path from the update to D's root. If we are interested in maintaining the materialized view V, the algoritm for Theorem 4.1 can also output the changes to V, i.e., output a set of nodes to be inserted, resp., removed from V, in time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$.

5. XPATH$(\rightarrow, \Rightarrow, \wedge)$ ON STRINGS

In the previous section, we presented an algorithm to efficiently maintain downward navigational queries. Our goal in the remainder of the article will be to partially extend this fragment by adding the next-sibling and following-sibling axes. This will, however, prove to be non-trivial and will come at the cost of removing negation and disjunction in the queries.

In this section, we present an algorithm for incrementally evaluating XPath$(\rightarrow, \Rightarrow, \wedge)$ on sequences of siblings or, in other words, strings. This algorithm will then be used in Section 6 to extend the algorithm of the previous section to also handle

the next- and following-sibling axes. More specifically, this section is devoted to proving the following result.

THEOREM 5.1. *Boolean incremental evaluation for an XPath*$(\rightarrow, \Rightarrow, \wedge)$ *Pattern* Q *and a string* D *can be performed in time* $\mathcal{O}(\log(|D|) \cdot \mathit{poly}(|Q|))$ *per update with an auxiliary data structure of size* $\mathcal{O}(|D| \cdot |Q|^3)$.

5.1 Evaluating an NFA on Strings

First, we explain the intuition behind incrementally evaluating a non-deterministic finite automaton (NFA) on strings, and the challenges that arise when trying to adapt this algorithm for incrementally evaluating XPath$(\rightarrow, \Rightarrow, \wedge)$ on strings. The following technique was first described by Patnaik and Immerman [Patnaik and Immerman 1997] and worked out in more detail by Balmin et al. [Balmin et al. 2004].

A *non-deterministic finite automaton (NFA)* is a tuple $N = (\text{States}(N), \text{Alph}(N), \text{Rules}(N), \text{init}(N), \text{Final}(N))$, where $\text{States}(N)$ denotes its set of states, $\text{Alph}(N)$ its alphabet, $\text{init}(N)$ its set of initial states, and $\text{Final}(N)$ its set of final or accepting states. The transition rules $\text{Rules}(N)$ are of the form $q_1 \xrightarrow{a} q_2$, indicating that reading an $a \in \text{Alph}(N)$ in state q_1 can bring the automaton in state q_2. Acceptance is defined in the standard manner. We denote by $L(N)$ the set of strings accepted by N.

Assume that we have a string $w = a_1 \cdots a_n \in \Sigma^*$ for which we incrementally want to maintain whether $w \in L(N)$. We first describe the auxiliary data structure we will maintain to do this efficiently. For each i, j, $1 \leq i < j \leq n$, let T_{ij} be the transition relation $\{(p, q) \mid p, q \in \text{States}(N), p \xrightarrow{a_i \cdots a_j} q\}$, where $p \xrightarrow{a_i \cdots a_j} q$ denotes that N can reach state q when it starts in state p and reads $a_i \cdots a_j$. Note that $T_{ij} = T_{ik} \circ T_{(k+1)j}$, $i < k < j$, where $\circ$ denotes composition of binary relations.

For simplicity, assume first that n is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the T_{ij} for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the transition relation tree $\mathcal{T}_n$ whose nodes are sets T_{ij}, defined inductively as follows:

—the root is T_{1n};
—each node T_{ij} for which $j - i > 0$ has children T_{ik} and $T_{(k+1)j}$ where $k = i - 1 + \frac{j-i+1}{2}$; and
—the T_{ii} are the leaves, for all $1 \leq i \leq n$.

Note that $\mathcal{T}_n$ has $n + (n/2) + \cdots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $\mathcal{O}(n \cdot |\text{States}(N)|^2)$.

First, notice that given $\mathcal{T}_n$ it is easy to decide whether $w \in L(N)$. Indeed, $w \in L(N)$ if and only if $(q, f) \in T_{1n}$ for some $q \in \text{init}(N)$ and $f \in \text{Final}(N)$. Therefore, we only have to show that this auxiliary data structure can be updated efficiently.

For simplicity, consider the case when one update occurs, changing the label of the symbol at position k of w to b. That is, the new string is $w = a_1 \cdots a_{k-1} b a_{k+1} \cdots a_n$. Note that the relations $T_{ij} \in \mathcal{T}_n$ that are affected by the updates are those lying on the path from the leaf T_{kk} to the root of $\mathcal{T}_n$. Denote the set of these relations by I

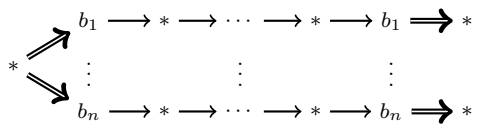


Fig. 3. Query illustrating the challenges when translating XPath$(\rightarrow, \Rightarrow, \wedge)$ into an NFA of polynomial size. Single and double arrows denote $\rightarrow$-axes and $\Rightarrow$-axes, respectively. For ease of presentation, the query trees are oriented horizontally instead of vertically.

and notice that it contains at most $\log n$ relations. The tree $\mathcal{T}_n$ can now be updated by recomputing the T_{ij}'s in I bottom-up as follows: First, the leaf relation T_{kk} is set according to Rules(N) and b. Then each $T_{ij} \in I$ with children T' and T'', of which one has been recomputed, is replaced by $T' \circ T''$. Thus, at most $\log n$ relations have been recomputed, each in time $\mathcal{O}(|\text{States}(N)|^2 \cdot \log |\text{States}(N)|)$, yielding a total time of $\mathcal{O}(|\text{States}(N)|^2 \cdot \log |\text{States}(N)| \cdot \log n)$.

The above approach can easily be adapted to strings whose length is not a power of 2. Further, the auxiliary data structure has size $\mathcal{O}(n \cdot |\text{States}(N)|^2)$. Finally, handling updates in which elements are inserted or deleted is also done in [Balmin et al. 2004], but then some precautions have to be taken in order to make sure that the tree $\mathcal{T}_n$ remains properly balanced.

There is a close connection between the tree $\mathcal{T}_n$ defined here and the tree obtained by the Simon decomposition theorem [Simon 1990]. The Simon decomposition tree is of linear size in n and, in addition, the depth of this tree is constant in n and exponential in N. Such a tree would allow to determine whether $a_i \cdots a_j \in L(N)$ and for substrings $a_i \cdots a_j$ of w even in constant time w.r.t. n, rather than logarithmic time in n. However, when performing updates to w, the Simon decomposition tree sometimes needs to be completely recomputed and therefore does not lead to worst-case logaritmic time complexity in n w.r.t. incremental maintenance.

5.2 Challenges for XPath$(\rightarrow, \Rightarrow, \wedge)$

Our approach for incremental XPath$(\rightarrow, \Rightarrow, \wedge)$ evaluation is based on the incremental algorithm for NFAs explained in Section 5.1. However, adapting the approach for NFAs poses the following serious challenges:

(1) It is not possible to translate an XPath$(\rightarrow, \Rightarrow, \wedge)$ query (or its complement) into an NFA of polynomial size. For instance, consider the XPath$(\rightarrow, \Rightarrow, \wedge)$ query illustrated in Figure 3, where $\rightarrow * \rightarrow \cdots \rightarrow *$ denotes an n-fold concatenation of $\rightarrow *$. Although this query is of size polynomial in n, any NFA defining this language, or its complement, must be of size exponential in n, as can be shown using standard techniques [Glaister and Shallit 1996].

The reason is that XPath$(\rightarrow, \Rightarrow, \wedge)$ queries possess a form of *alternation*. That is, when reading a string from left to right while testing whether it matches an XPath$(\rightarrow, \Rightarrow, \wedge)$ query, one needs both existential and universal quantification. Universal quantification is needed for handling the $\wedge$-operations and the branching in the query ("*all* the following subqueries must match"), and existential quantification is needed for handling the $\rightarrow$ and $\Rightarrow$-axes ("*there exists* a position in the future such that the following subquery matches"). Translat-

ing an XPath$(\rightarrow, \Rightarrow, \wedge)$ query into a polynomial size alternating automaton is possible, but this poses challenge (2).

(2) It is not clear how to extend the maintenance algorithm from Section 5.1 to alternating automata. The problem is that maintaining binary transition relations T_{ij} is not enough to ensure correctness. One would need to maintain all *pairs of sets of states* instead, which makes the data structure exponential in the size of the query.

In other words, we will describe how to extend the approach of Section 5.1 to a limited form of alternating automata. The reason why we are still able to adapt this approach to XPath$(\rightarrow, \Rightarrow, \wedge)$ queries is because the queries do not use the full power of alternating automata.

5.3 From NFAs to XPath$(\rightarrow, \Rightarrow, \wedge)$

Since we are only concerned with matching queries on strings for the moment, we can simplify our queries in a pre-processing step. If a query node u has two children v_1 and v_2, and both (u, v_1) and (u, v_2) have type $\rightarrow$, then any matching of the pattern on a string must match v_1 and v_2 to the same string position, namely to the one directly to the right of the position where u is matched. This means that we might as well merge v_1 and v_2 into a single query node v, if v_1 and v_2 have compatible labels. If they have conflicting labels, we simply conclude that there is no string onto which the pattern can be matched.

Also, since we consider only XPath Patterns Q without negation or disjunction, $D \models^u Q$ if and only if there exists a homomorphic mapping $\phi : \text{Nodes}(Q) \rightarrow \text{Nodes}(D)$ (we say Q *can be matched onto* D) such that $\phi(\text{root}(Q)) = u$, for every node $u \in \text{Nodes}(Q)$, either $\text{lab}^Q(u) = *$ or $\text{lab}^Q(u) = \text{lab}^D(\phi(u))$, and, for all edges $e = (i, j) \in Q$, we have

—type(i) = syntax implies $\phi(i) = \phi(j)$,

—type$(e) = \rightarrow$ (resp., $\Rightarrow$) implies that $\phi(j)$ is a next sibling (resp., following sibling) of $\phi(i)$.

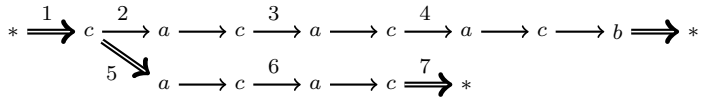
In the remainder of this section, we refer to ϕ as a *matching of* Q *(on* D*).*

This means that we don't have to differentiate between syntax nodes and label nodes. We simply regard every node as a label node that also, implicitly, acts as an and-node. These considerations make the following assumption possible:

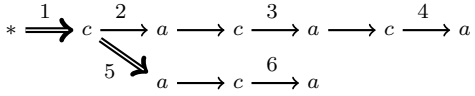
Proviso 5.2. In the rest of this section, no query node has two outgoing edges with type $\rightarrow$, all query nodes are of type label, and all edges have type $\rightarrow$ or $\Rightarrow$. Every query node is treated as an implicit and-node.

The incremental algorithm for an NFA remembers in each relation T_{ij}, the pairs (p, q) such that, reading the string from position i to position j can bring the automaton from state p to state q. We will remember something similar for XPath$(\rightarrow, \Rightarrow, \wedge)$, which we first illustrate by means of an example. Intuitively, instead of automaton states, we will now store *edges in* Q. For an edge $e = (x, y)$ we refer to x as the *source* and y as the *target* of e.

Example 5.3. Consider the query Q in Figure 4(a) and the string $D = cacac$. Intuitively, D should be seen as a substring of a much larger string, for which we



(a) The query Q of Example 5.3.



(b) The query Q' of Example 5.3.

Fig. 4. Queries for Example 5.3. Single and double arrows denote $\rightarrow$-axes and $\Rightarrow$-axes, respectively.

want to compute the information for T_{ij}. Intuitively, we will remember all pairs $(e_1, e_2) \in \text{Edges}(Q) \times \text{Edges}(Q)$ such that the part of the query from the target of e_1 to the source of e_2 can be matched inside D, much like in the NFA case. We talk about edges here because when combining a matching of a part of a query on a part of the string with another matching for a consecutive part of the string, what really interests us is which query *edges* lead from one string part to the next.

For $D = cacac$, we remember the pairs $(1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (5,7)$. The intuition is that, if we read D from left to right and we start matching in, e.g., edge[6] 1, then pair $(1,4)$ tells us that we can match until edge 4 at the end of D, *ignoring all paths in Q that branch away* from the path from 1 to 4. So we essentially treat each path in Q as an NFA, where the edges are its states. The difference from the NFA approach is that single pairs do not tell us the whole story. For instance, the pair $(1,4)$ does not tell us what happens with the path that branches away, the one from edge 5 to edge 7. Thus we have to combine pairs in order to get matchings that span more than one path in Q. For example, $(1,4)$ and $(1,6)$ can be combined to form a partial matching of Q in the following way. Let Q' be obtained from Q by cutting off everything left of 1 and everything right of 4 and 6 (see Figure 4(b)). We can now match Q' into D such that the target of 1 is matched precisely onto the leftmost symbol and the sources of 4 and 6 onto the rightmost symbol, so the matching could continue to the right of D. Notice that we do not care (yet) about the target labels of 4 and 6.

Note, however, that we cannot combine pairs arbitrarily: $(1,3)$ and $(1,7)$ cannot be combined into a correct partial matching. Naively combining them would lead to incorrect partial matchings. Any matching for $(1,3)$ would have to match the source node of 3 to the last position of the string $D = cacac$. This is because 3 has type $\rightarrow$, and we want to be able to continue the matching of Q to the right of D. This, in turn, means that the source of edges 2 and 5 must be matched against the middle position of D. But if we do this, there are only two positions left for the four query nodes between edges 5 and 7 to match against. Thus $(1,3)$ and $(1,7)$ cannot be combined.

To solve this problem, we have to be careful about which information to store. A naive generalization of the NFA approach would store pairs (p, P) such that p is

[6]For the sake of the argument, the reader can assume that the source node of edge 1 is already matched one position before the first position of D.

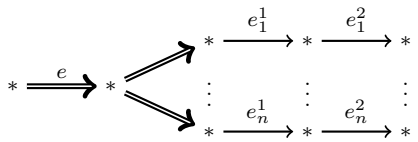


Fig. 5. For any tuple $(i_1, i_2, \ldots, i_n) \in \{1,2\}^n$, the pair $(e, \{e_1^{i_1}, e_2^{i_2}, \ldots, e_n^{i_n}\})$ is a good partial matching for the above query pattern on any string of length at least 3. There are 2^n such pairs, which makes naively storing them all infeasible.

a start edge and P is a set of edges such that all the pairs $\{(p, p') \mid p' \in P\}$ can be combined into a correct partial matching. In the example, $(1, \{4, 6\}))$ would be one such pair. Unfortunately, as illustrated by Figure 5, storing such pairs would again lead to storing an exponential amount of information in the query Q.

Therefore, we have to adopt a smarter approach, which we describe next.

5.4 Towards the Incremental Algorithm

The rough outline of the algorithm for incrementally evaluating an XPath($\rightarrow, \Rightarrow, \wedge$) pattern can now be described. It is similar to the algorithm for NFAs described in Section 5.1, with the three crucial differences that

—the relations T_{ij} store different information;

—the algorithm for joining two relations T_{ik} and $T_{(k+1)j}$ into T_{ij} is completely different; and

—the test for acceptance that needs to be performed at T_{1n} is different.

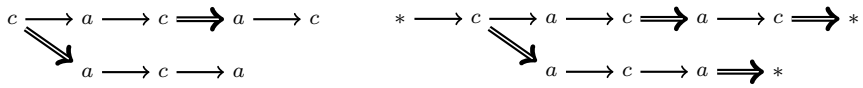
Here, we describe what information will be stored in T_{ij}. Section 5.6 treats the problem of joining two relations relations T_{ik} and $T_{(k+1)j}$.

First, we need to introduce some new notation and terminology. The relations T_{ij} store information about the query Q and its subqueries. We introduce the next proviso to enable a uniform formal treatment of Q and its subqueries (see Remark 5.6).

Proviso 5.4. From now on, we assume that our query Q has a single outgoing edge $\top := (\text{root}(Q), u)$ from the root, which we will refer to as the *root edge*. The type of the root edge can be $\Rightarrow$ or $\rightarrow$. Likewise, for every parent p of a leaf node v, we assume that p has a single outgoing *leaf edge* $\bot_v = (p, v)$, with $\text{type}(\bot_v) = \Rightarrow$.

If one is interested in matching a query P against a string D, then a new root r and new leaves labeled $*$ should be added to P, thereby obtaining P' with root edge $(r, \text{root}(P))$ and leaf edges conforming to Proviso 5.4; see Figure 6. For technical reasons (i.e., the uniform treatment of Q and its subqueries), our algorithm will only start matching at node $\text{root}(P)$ (see Remark 5.8). If the root edge of P' has type $\rightarrow$, then we will match $\text{root}(P)$ at the first position of D. Otherwise, $\text{root}(P)$ can be matched at an arbitrary position of D. Our treatment for the leaf edges is similar. The reason why leaf edges are typed $\Rightarrow$ is because we don't require all the leaves of P to be matched at the last position of D.

Proviso 5.5. We extend some standard terminology of relations between nodes in trees to edges. We say that edge $e_1 = (x_1, y_1)$ is a descendant edge of $e_2 = (x_2, y_2)$



(a) A query P without root or leaf edges.

(b) In order to satisfy Proviso 5.4, we add a dummy root and two dummy leaves to P, together with root and leaf edges.

Fig. 6. Example of how nodes and edges can be added to a query in order to satisfy Proviso 5.4.

if y_1 is a descendant of y_2.

A *path* in a pattern Q is a sequence $\rho = (x_1, y_1) \cdots (x_k, y_k)$ of edges of Q such that $y_i = x_{i+1}$ for all $i \in \{1, \ldots, k-1\}$. If x_1 is the root and y_k is a leaf, we say that ρ is a *maximal path*. A *cut* of a pattern Q is a subset C of Edges(Q) such that every maximal path in Q has exactly one edge in C.

Let C be a cut and $e = (x, y)$ be an edge of Q such that e is not a descendant of any edge in C. The *induced subquery of Q w.r.t. e and C*, denoted subQ(Q, e, C), is the pattern obtained from Q by considering the subtree of Q rooted at e, and removing everything below C. More formally, subQ(Q, e, C) is the query Q' where

—Nodes(Q') is $\{x, y\} \uplus \{z \in \text{Nodes}(Q) \mid \text{descendant}(y, z)$ holds in Q and $\nexists (u, v) \in C$ such that descendant(v, z) holds in $Q\}$;

—the edges in Q' are the same as in Q, i.e., Edges(Q') $= \{(x, y) \mid (x, y) \in \text{Edges}(Q)$ and $x, y \in \text{Nodes}(Q')\}$;

—the root edge of Q' is $\top = (x, y)$; and

—all edges and nodes in Q' inherit their types from Q.

To simplify notation further on, we use subQ$(Q, e, \bot)$ to denote subQ(Q, e, C) where C is the cut consisting of all leaf edges of Q. Hence, subQ$(Q, \top, \bot) = Q$.

Notice that induced subqueries always have a unique edge leaving their root. Similar to Proviso 5.4, we will also refer to this edge as the *root edge* of the subquery. Therefore, we have the following property for all queries and subqueries:

Remark 5.6. In this section, all queries and induced subqueries of Q have a unique root edge.

We now define the notion of a partial matching of a subquery into a string D. Our terminology will be slightly more refined — we use *full* and *top* matchings. Intuitively, a full matching ϕ of a query Q will map all the nodes of Q into D, except for its root node and its leaves. A top matching of Q will be a partial matching that only matches some upper part of Q into D, also excluding the root node.

Definition 5.7 (Top matching, full matching). Let P be an induced subquery of Q and let $e = (x, y)$ be the root edge of P. Let C be a cut of P and $C_{\text{Low}} = \{c_1, \ldots, c_n\} = \{v \mid \exists u.(u, v) \in C\}$. Let inner$(P, C)$ be all nodes of P that are descendants of x and have a descendant in C_{Low}.

Then $\phi : \text{inner}(P, C) \to \text{Nodes}(D)$ is a *top matching* of P if the following hold:

—ϕ is a matching from inner(P, C) to D;

—if e has type "$\to$", then $\phi(y)$ is the first position in D; and

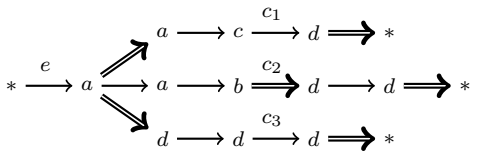


Fig. 7. Example of some of the concepts from Definition 5.7. Call the query depicted above P. Then e is the *root edge* of P and the set $C = \{c_1, c_2, c_3\}$ represents a possible *cut* of P. Consider the subset $C' = \{c_1, c_2\}$ of C. The set $\text{inner}(P, C')$ contains all nodes between e and C', i.e., the five nodes labeled a, b, or c. A *top matching* of P on a string D with respect to C' is a mapping from $\text{inner}(P, C')$ to the nodes of D such that the following conditions are satisfied. (1) Since the root edge e has type $\rightarrow$, the target node of e must be mapped to the first position of D. (2) Since edge c_1 has type $\rightarrow$, it's source node (the unique node with the label c) has to be matched to the last position of D. The source node of c_2 does not have to be matched to the last position of D since c_2 has type $\Rightarrow$.

—for each $j = 1, \ldots, n$, if there is a u such that $(u, c_j) \in C$ is a "$\rightarrow$"-edge, then $\phi(u)$ is the last position in D.

We say that C is a *witness for* ϕ and, for any $C' \subseteq C$, we also say that ϕ is a *top matching w.r.t.* C'. A top matching ϕ is also a *full matching of* P *on* D, if C_{Low} is the set of leaves of P.

Figure 7 illustrates some of the concepts from the above definition.

Remark 5.8. Note that the notions of full and top matching do not require the root nor leaf nodes to be matched. This is consistent with our discussion following Proviso 5.4. It follows that, for every query P, $D \models P$ if and only there is a full matching from the extended version P' of P (with a root edge of type $\rightarrow$) to D.

5.5 The Incremental Algorithm

Recall that, in the incremental evaluation algorithm, T_{ij} denotes the auxiliary data record for the string $a_i \cdots a_j$. Example 5.3 shows that a naive generalization of the algorithm for NFAs that would store all pairs (e, C) in T_{ij}, such that there is a full matching of $\text{subQ}(Q, e, C)$ on $a_i \cdots a_j$ would need to store exponentially many cuts C for an edge e in the worst case. Intuitively, our improvement to this naive approach is to store a ternary relation over edges in each T_{ij}. This relation is a combination of the binary relation shown in Example 5.3, which contains pairs of edges $(e_{\text{top}}, e_{\text{bot}})$ such that there exists a top matching of $\text{subQ}(Q, e_{\text{top}}, \bot)$ w.r.t. $\{e_{\text{bot}}\}$, and a *co-matchability* constraint, which allows us to infer which such pairs can be combined to form a consistent matching. More precisely, we store triples $(e_{\text{top}}, e_{\text{bot}}, e)$ such that there is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ for $\text{subQ}(Q, e_{\text{top}}, \bot)$ on D. We formalize this in terms of *matching triples*, for which we first introduce some notation.

Definition 5.9. An edge $e = (x, y)$ in Q is *direct* if either

—e is a $\rightarrow$-edge; or

—e is a $\Rightarrow$-edge and all other edges (x, z) in Q are also $\Rightarrow$-edges.

All other edges in Q are called *bridge edges*. A path in Q that consists only of direct edges is a *direct path*. Let $e_1 e_2 \cdots e_k$ be a path in Q. Then the *bridge distance* of

e_k from e_1, denoted $\|e_1, e_k\|$ is the number of bridge edges in $\{e_2, \ldots, e_k\}$ (i.e., we count e_k, but not e_1). If $\|e_1, e_k\| = 0$, we also say that e_k is a *direct descendant* of e_1. The set of all bridge edges on the path from e_1 to e_k, again not including e_1, will be denoted Bridges(e_1, e_k).

Notice that each bridge edge is a $\Rightarrow$-edge and that $\|e_1, e_2\|$ is only defined if e_2 is a descendant of e_1 in Q.

Consider the example query depicted in Figure 10. The double arrows denote edges of type $\Rightarrow$ and the lines denote (sequences of) edges of type $\rightarrow$. Call the topmost edge e_{top}. The edges e_1, e_2, e_3, e_4 are *bridge edges*, because they have type $\Rightarrow$ and there are also edges of type $\rightarrow$ leaving their source nodes. The edge of type $\Rightarrow$ on the path from e_{top} to c is, however, not a bridge edge, since all edges leaving its source node have type $\Rightarrow$. The *bridge distance* from e_{top} to another edge is the number of bridge edges on the path between them. Thus $\|e_{\text{top}}, c\| = 0$, $\|e_{\text{top}}, c_1\| = 1$, $\|e_{\text{top}}, c_2\| = 2$ and so on. Notice that, e.g., edges c_3 and d have the same bridge distance from e_{top}, in this case 3.

Definition 5.10 (Matching triple). Let $e_{\text{top}}, e_{\text{bot}}, e$ be query edges of Q such that e is a descendant and e_{bot} a direct descendant of e_{top}. Then $(e_{\text{top}}, e_{\text{bot}}, e)$ is a *matching triple for* D if there is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ for subQ$(Q, e_{\text{top}}, \bot)$ on D. We denote the set of matching triples of D by Triples(D).

For incrementally maintaining an XPath$(\rightarrow, \Rightarrow, \wedge)$ query, we can now maintain a tree $\mathcal{T}_n$ for the string $D = a_1 \cdots a_n$ as in Section 5.1, but containing matching triples instead of pairs of states. This tree can be computed as follows:

(A) For each position i in D, T_{ii} is the set of all matching triples for a_i and can be computed directly.

(B) Each T_{ij} in the data structure can be computed from T_{ik} and $T_{(k+1)j}$, where $k = i - 1 + \frac{j-i+1}{2}$, by adding, for each pair of edges $e_{\text{top}}, e_{\text{bot}}$ in Q such that e_{bot} is a direct descendant of e_{top}, the triples $(e_{\text{top}}, e_{\text{bot}}, e)$ that are computed by Join(Q,T_{ik},$T_{(k+1)j}$,e_{top},e_{bot}). This Join-procedure is the main technical difficulty in this article and is explained in Section 5.6. For the time being, it is only important to know that it runs in time polynomial in $|Q|$ and that it computes the matching triples for T_{ij} correctly.

At the root $T_{1,n}$ of the data structure we have that $D \models Q$ if and only if there exists a direct descendant e_{bot} of e_{top} such that, for all leaf edges e of Q, we have $(e_{\text{top}}, e_{\text{bot}}, e) \in T_{1,n}$, where e_{top} is the unique root edge of Q (This is formally stated in Lemma 5.11 below). Clearly, this can be tested in polynomial time. The size of the auxiliary data structure $\mathcal{T}_n$ is $\mathcal{O}(n \cdot |Q|^3) = \mathcal{O}(|D| \cdot |Q|^3)$.

Lemma 5.11. *Let Leaf(Q) be the set of leaf edges Q. Let e_{top} be the root edge of Q, and $e_{bot} \in$ Leaf(Q) be a direct descendant of e_{top}. Then, there is a full matching of Q on D, if and only if for all $e \in$ Leaf(Q), $(e_{top}, e_{bot}, e) \in$ Triples(D).*

When a position of D is updated, the incremental update mechanism is exactly the same as in Section 5.1, with the only difference that the updates in $\mathcal{T}_n$ follow the rules (A) and (B) above for recomputing the T_{ij}'s on the path from a leaf to the root. Such an update takes poly(Q) time for (A), and $\mathcal{O}(\log(D) \cdot \text{poly}(Q))$ time

for the iteration in (B). Finally, testing if the root condition is fulfilled again takes time polynomial in Q.

Node insertions and node deletions complicate matters slightly, as we can no longer rely on a fixed division of D into intervals to build $\mathcal{T}_n$. The solution is to use balanced search trees, that can easily be re-balanced after an insertion or deletion adds or removes an interval. One option is to use 2-3-trees (a variant of B-trees). This approach is taken by Balmin et al. and can be used in exactly the same way in our setting. We refer to. [Balmin et al. 2004] for the details.

Hence, in order to prove the main theorem of this section (Theorem 5.1), it remains to prove Lemma 5.11 and the correctness and the feasibility of the joins in (B) in time poly(Q). The next section presents this join procedure, and Lemma 5.14 in Section 5.7 proves it to be correct and running in time poly(Q). The proof of Lemma 5.11 is given at the end of Section 5.7

5.6 Joining the Data for Two Substrings

In this section we will present the join algorithm. Before we can state the algorithm, we first have to give some additional definitions.

Definition 5.12 (Bridge width). Let r be the root edge of Q. For an induced subquery P of Q, the *bridge width* of P, denoted $\|P\|_{\text{bw}}$, is the maximal bridge distance in P, i.e., $\|P\|_{\text{bw}} = \max\{\|e_1, e_2\| \mid e_1, e_2 \in \text{Edges}(P)\}$. The subquery of P with *bridge width* i, denoted $\text{bw}_i(P)$, is the query obtained from P by removing all edges e such that $\|\text{root}(P), e\| > i$, and removing all nodes thus disconnected from $\text{root}(P)$.

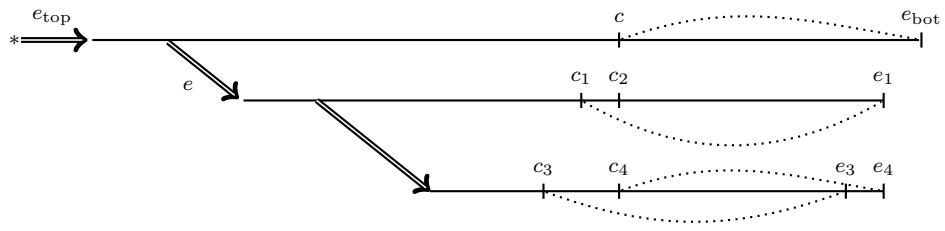
We also write $\|e\|_{\text{bw}}$ instead of $\|r, e\|$, when $r = \text{root}(P)$.

By $D_1 \cdot D_2$ we denote the concatenation of strings D_1 and D_2. If $D = D_1 \cdot D_2$ and ϕ a top matching for Q on D, we write $\text{Border}(D_1, D_2, \phi)$ for the set of edges $e = (x, y)$ of Q such that $\phi(x) \in D_1$ and either $\phi(y) \in D_2$ or $\phi(y)$ is undefined. Finally, for $C \subseteq \text{Edges}(Q)$, let $\text{Low}(C)$ be the set obtained from C by removing all edges which have a descendant in C.

The core problem for the join algorithm is the following: Given strings D_1 and D_2, the sets of matching triples $\text{Triples}(D_1)$ and $\text{Triples}(D_2)$, the query Q, and edges e_{top} and e_{bot} from Q such that e_{bot} is a direct descendant of e_{top}, compute all triples $(e_{\text{top}}, e_{\text{bot}}, e)$ that belong to $\text{Triples}(D_1 \cdot D_2)$. We can then compute the set of all possible matching triples by iterating over all choices of e_{top} and e_{bot}. An algorithm for the core problem is given as Algorithm 1. To get a feel for what the algorithm must do, we consider an example.

Example 5.13. Consider the query pattern Q in Figure 8. Each double line denotes a $\Rightarrow$-edge, and a single line denotes a sequence of $\rightarrow$-edges. Notice that we depicted Q sideways, so all edges are directed from left to right. We now assume that we already have the matching triples $\text{Triples}(D_1)$ and $\text{Triples}(D_2)$ for strings D_1 and D_2 and we want to compute $\text{Triples}(D)$, where $D = D_1 \cdot D_2$. For our example, the matching triples for D_1 and D_2 are the ones given in Figure 8(b).

Strictly speaking, $\text{Triples}(D_1)$ and $\text{Triples}(D_2)$ would contain many more matching triples, such as $(e_{\text{top}}, e_{\text{top}}, e_{\text{top}})$, but we limit ourselves to an interesting subset. We cross out a few triples because we want to explicitly assume that they are *not*



(a) Abstract query.

Triples(D_1)	(e_{top}, c, c)	(e_{top}, c, c_1)	(e_{top}, c, c_2)	(e_{top}, c, c_3)	(e_{top}, c, c_4)
	(e, c_1, c_3)	(e, c_2, c_3)	(e, c_2, c_4)	~~(e, c_1, c_4)~~	...
Triples(D_2)	$(c, e_{\text{bot}}, e_{\text{bot}})$	(c_1, e_1, e_1)	~~(c_2, e_1, e_1)~~	(c_3, e_3, e_3)	(c_4, e_4, e_4) ...

(b) Matching triples for D_1 and D_2.

Fig. 8. Abstract query and matching triples for Example 5.13.

matching triples. (Notice that it is possible that (e, c_1, c_3) is a matching triple, while (e, c_1, c_4) is not since the latter could violate the lowest $\Rightarrow$-edge in Figure 8.) We depict the triples in Triples(D_2) with dotted lines in Figure 8.

The algorithm takes the two sets of matching triples as input, together with the edges e_{top} and e_{bot}. It will infer every edge e such that $(e_{\text{top}}, e_{\text{bot}}, e)$ is a matching triple. The algorithm iterates over edges with increasing bridge distance from e_{top}. In the figure, the edges on the path from e_{top} to e_{bot} have bridge distance zero, the e-edge, together with the edges on the middle line have bridge distance one, and all other edges have bridge distance two.

Initially, on lines 3–8 of the algorithm, we join (e_{top}, c, c) with $(c, e_{\text{bot}}, e_{\text{bot}})$ into $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})$, which is our first triple in Triples(D). (Notice that the value of c here is the same as in line 5 of the algorithm.) During the computation we iteratively construct a set C of edges (x, y) in Q that contains edges for which we matched the source x in D_1 and the target y in D_2, and which forms a cut through Q. Intuitively, C should be a "greedy cut", i.e., we remember the edges that are as close to the leaves of Q as possible. (In the algorithm, we use the term "lowermost" to say that an edge should be as close to the leaves as possible.) Therefore, on line 5, we set $C = \{c\}$ which forms a cut through the part of Q only containing the edges at distance zero.

We then proceed to the loop that begins on line 12. In the first iteration, when $j = 0$, we should consider all edges e with distance zero from e_{top}. However, in our example, all such edges lie on the path from e_{top} to e_{bot}. We can never have a matching triple $(e_{\text{top}}, e_{\text{bot}}, e)$ where $e \neq e_{\text{bot}}$ lies on the same path as e_{bot}, since only one of them can be matched at the end of D. Thus we already covered this case by adding $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})$ to Triples(D).

Therefore, we proceed to the second iteration of the while loop in which all edges at distance one are considered. Here, the if-statement on lines 19–22 applies, and we can combine

$$(e_{\text{top}}, c, c_1) \in \text{Triples}(D_1) \text{ with } (c_1, e_1, e_1) \in \text{Triples}(D_2) \text{ into } (e_{\text{top}}, e_{\text{bot}}, e_1).$$

By doing so, we add c_1 to C' (in which we temporarily store candidate edges for the

Algorithm 1 Join algorithm. The algorithm takes a query Q, two sets of matching triples $\text{Triples}(D_1)$ and $\text{Triples}(D_2)$ and two edges $e_{\text{top}}, e_{\text{bot}}$ as input. It is assumed that e_{bot} is a direct descendant of e_{top}. The algorithm computes all matching triples $(e_{\text{top}}, e_{\text{bot}}, e)$ of $D = D_1 \cdot D_2$, where e is a descendant of e_{top}.

Join(Query Q, Triples(D_1), Triples(D_2), Edge e_{top}, Edge e_{bot})
2: $P \leftarrow \text{subQ}(Q, e_{\text{top}}, \bot)$
if $\exists c : (e_{\text{top}}, c, c) \in \text{Triples}(D_1) \wedge (c, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D_2)$ **then**
4: $c \leftarrow$ the lowermost such edge
$C \leftarrow \{c\}$
6: $T_0(D) \leftarrow \{(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})\}$
else
8: **return** $\emptyset$
end if
10: $j \leftarrow 0$
$T_1(D) \leftarrow \cdots \leftarrow T_{\|P\|_{\text{bw}}}(D) \leftarrow \emptyset$
12: **while** $j \leq \|P\|_{\text{bw}}$ **do**
$C' \leftarrow \emptyset$
14: **for all** $e \in \text{Edges}(P)$ s.t. $\|e_{\text{top}}, e\| = j$ **do**
if $(c, e_{\text{bot}}, e) \in \text{Triples}(D_2)$ **then**
16: $T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
else if $\exists e' \in C \setminus \{c\}\ \exists e'' : (e', e'', e) \in \text{Triples}(D_2)$ **then**
18: $T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
else if $\exists e' : (e_{\text{top}}, c, e') \in \text{Triples}(D_1) \wedge (e', e, e) \in \text{Triples}(D_2)$
20: $\wedge \forall e_k \in \text{Bridges}(e_{\text{top}}, e') \cup \{e_{\text{top}}\}$ with $\|e_{\text{top}}, e_k\| = k < j$,
$\forall c_k \in C$ with $\|e_{\text{top}}, c_k\| = k < j$:
22: c_k descendant of $e_k \Rightarrow (e_k, c_k, e') \in \text{Triples}(D_1)$ **then**
$e' \leftarrow$ the lowermost such edge
24: $C' \leftarrow C' \cup \{e'\}$
$T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
26: **end if**
end for
28: $C \leftarrow C \cup \text{Low}(C')$
$j \leftarrow j + 1$
30: **end while**
return $\cup_{j=0}^{\|P\|_{\text{bw}}} T_j(D)$

greedy cut C) on line 24. As at the end of the while loop, we still have $C' = \{c_1\}$, we obtain $C = \{c, c_1\}$ after the second iteration. Notice that $\{c, c_1\}$ indeed forms a cut of $\text{bw}_1(Q)$, the part of Q only containing the edges with bridge width at most one.

The third iteration of the while loop is the first one that becomes interesting, since we have to consider a combination of three paths in the query, while our triples only store information about pairs of paths. First, we discuss how naive joins may go wrong. One may be tempted to conclude from $(e_{\text{top}}, c, c_4) \in \text{Triples}(D_1)$, $(c, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D_2)$, and $(c_4, e_4, e_4) \in \text{Triples}(D_2)$ that $(e_{\text{top}}, e_{\text{bot}}, e_4)$ is a matching triple for D. However, it is not! The reasons are that *(i)* we could match

up to c_2 in D_1 (since $(e_{\text{top}}, c, c_2) \in \text{Triples}(D_1)$), but we could not continue this matching in D_2 (since $(c_2, e_1, e_1) \notin \text{Triples}(D_2)$) and *(ii)* $(e, c_1, c_4) \notin \text{Triples}(D_1)$. In other words, we cannot achieve a matching that is consistent with the lowest possible cut $C = \{c, c_1\}$ that we have computed thus far. This is the point where we need to make use of C to make the correct combinations, and decide which matching can be combined and which cannot. Therefore, by applying lines 19–22 we combine, among others,

$$(e_{\text{top}}, c, c_3) \text{ and } (e, c_1, c_3) \text{ from Triples}(D_1) \text{ with } (c_3, e_3, e_3) \text{ from Triples}(D_2) \text{ into } (e_{\text{top}}, e_{\text{bot}}, e_3).$$

After iteration three, we will, for this query, have computed all the matching triples and a cut $C = \{c, c_1, c_3\}$. Should the query be larger, C would be used to witness further matchings. This concludes Example 5.13.

We now present some more general ideas behind Algorithm 1. We already explained in Example 5.13 that the algorithm investigates triples $(e_{\text{top}}, e_{\text{bot}}, e)$ in order of increasing bridge distance between e_{top} and e, and that C is a greedy cut containing edges (x, y) for which x is matched in D_1 and y in D_2. We assume that the algorithm has computed all the triples with $\|e_{\text{top}}, e\| < j$ and that C contains the edges closest to the leaves of Q with bridge distance smaller than j such that there is a top matching of $P = \text{subQ}(Q, e_{\text{top}}, \bot)$ with respect to C on D_1. Now, consider what the algorithm does for an edge e with $\|e_{\text{top}}, e\| = j$. This edge is submitted to three tests: the if-statements on lines 15, 17, and 19–22. The situations the tests look for are depicted in Figure 9. Each double line denotes a $\Rightarrow$-edge, and a single line denotes a sequence of $\rightarrow$-edges. All edges are directed from left to right.

The first test, on line 15, looks for the situation depicted in Figure 9(a), i.e., when the path from e_{top} to e branches of from the path from e_{top} to e_{bot} after the edge c. If this is the case, the algorithm only has to check whether $(c, e_{\text{bot}}, e) \in \text{Triples}(D_2)$.

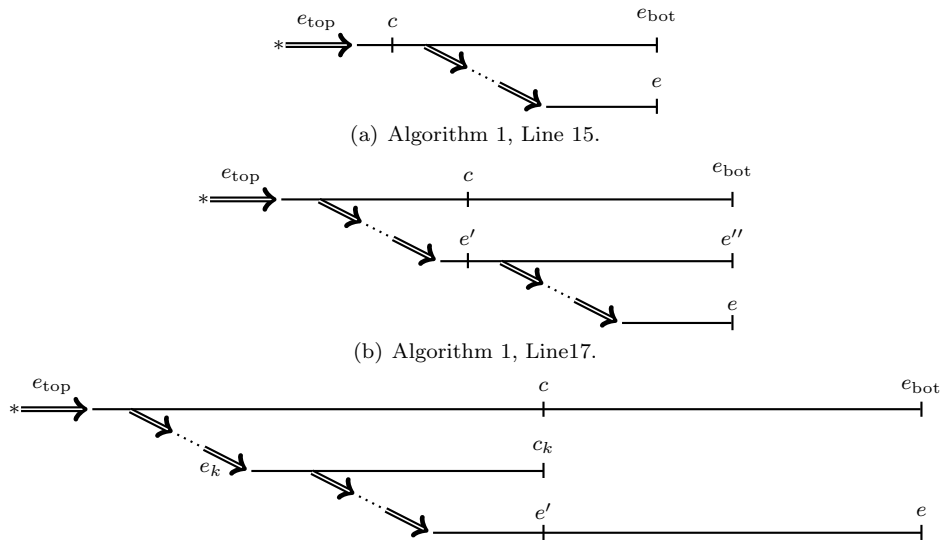
The second test, on line 17, looks for the situation in Figure 9(b). Here, there is an edge e', different from c, that lies on the path from e_{top} to e and already belongs to C (which implies $\|e_{\text{top}}, e'\| < j$). The algorithm now only needs to test whether, for some e'', the triple (e', e'', e) belongs to $\text{Triples}(D_2)$.

The third test, on lines 19–22, is the most complicated. It looks for the situation in Figure 9(c). Here, no edge on the path from e_{top} to e yet belongs to C. This means that the algorithm has to verify that the edge e' that is on the path from e_{top} to e and goes from D_1 into D_2 in the intended matching, is also consistent with the cut C constructed thus far.

By applying Algorithm 1 for all possible edges e_{bot} and e_{top} we obtain the following.

LEMMA 5.14. *Given the query Q and the sets Triples(D_1) and Triples(D_2) of matching triples for Q on strings D_1 and D_2, the set Triples(D) of matching triples for Q on $D = D_1 \cdot D_2$ can be computed in time polynomial in $|Q|$.*

PROOF. We first analyze the running time of Algorithm 1. Its first part (lines 3-9) takes time linear in $|Q|$. (We can assume constant time lookup in the sets $\text{Triples}(D_1)$ and $\text{Triples}(D_2)$, which can be implemented, e.g., as tree-dimensional matrices.) In the second part (lines 10-30) all edges that are descendants of e_{top},



(a) Algorithm 1, Line 15.

(b) Algorithm 1, Line17.

(c) Algorithm 1, Lines 19–22.

Fig. 9. Illustrations of the situations identified by the if-statements in the for-loop of Algorithm 1.

in order of increasing bridge distance, are considered. The for-loop that starts on line 14 makes one iteration per such edge. Within the loop, three cases are distinguished by the if-statements on lines 15, 17 and 19–22. The most complicated of these is the third, lines 19–22, which looks at cubically many triples, and thus runs in cubic time. All in all, Algorithm 1 runs in time $\mathcal{O}(|Q|^4)$.

To compute all matching triples in Triples(D) we have to call the join algorithm for all pairs $(e_{\text{top}}, e_{\text{bot}})$, i.e., a quadratic number of times, so the total running time is $\mathcal{O}(|Q|^6)$. □

The auxiliary data structure needed for incremental evaluation over a string D has a number of tree nodes that is linear in $|D|$. Each tree node contains a set of matching triples, and thus needs space $\mathcal{O}(|Q|^3)$. In total, the size of the auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|^3)$. Together with Lemma 5.14, this gives us Theorem 5.1.

5.7 Correctness of the Join Algorithm

This section is devoted to proving the correctness of the join algorithm (Lemma 5.25). We start by proving a number of lemmas concerning matchings and related matters.

For our approach, some top matchings of Q will be "better" than others. Intuitively, a top matching ψ is better than ϕ if either ψ can match a larger subquery of Q or if ψ can match some nodes of Q more to the left in the string D. We formalize this by defining the following partial order on top matchings. Here, "larger" according to the ordering should correspond to the intuition of a better top matching.

Proviso 5.15. *We will make use of the standard orderings $<$ of positions in strings, i.e., if $D = a_1 \cdots a_n$ with positions $\{1, \ldots, n\}$, we write $x < y$ for positions x and y if x occurs to the left of y. Similarly for $\leq, >, \geq$.*

DEFINITION 5.16. *Let Ψ be the set of all top matchings of Q on D. Then $\preceq$ is the order on Ψ defined by $\phi \preceq \psi$ if, for every $x \in Nodes(Q)$, either $\phi(x) \geq \psi(x)$, or $\phi(x)$ is undefined. If $\phi \preceq \psi$ and $\phi \neq \psi$ we write $\phi \prec \psi$.*

Since we are investigating matchings on strings, two top matchings can be combined to form another top matching as follows. By $\mathrm{Dom}(\phi)$ we denote the domain of a matching ϕ.

DEFINITION 5.17 (MERGE). *Let Q be a pattern, and D a string. Let P_ϕ and P_ψ be subqueries of Q and ϕ and ψ top matchings of P_ϕ and P_ψ on D, respectively. Then,*

$$merge(\phi,\psi)(x) := \begin{cases} \phi(x) & : \quad x \in Dom(\phi) \setminus Dom(\psi) \text{ or } \phi(x) \leq \psi(x) \\ \psi(x) & : \quad \text{otherwise.} \end{cases}$$

When ϕ and ψ are top matchings of the same pattern, the new matching $\mathrm{merge}(\phi, \psi)$ is at least as good as ϕ and ψ. We formalize this in the following lemma. Notice that the correctness of the lemma crucially depends on the fact that D is a string.

LEMMA 5.18. *Let ϕ and ψ be top matchings for Q on D. Then, merge(ϕ,ψ) is also a top matching for Q on D. Furthermore, $\phi \preceq$ merge(ϕ,ψ) and $\psi \preceq$ merge(ϕ,ψ).*

PROOF. We first prove that $\mathrm{merge}(\phi, \psi)$ is a top matching on D by showing that all edge conditions in the domain of $\mathrm{merge}(\phi, \psi)$ are satisfied. Let C_ϕ and C_ψ be the witness cuts for ϕ and ψ, respectively. We prove that the cut C, obtained by taking the lowest edge of $C_\phi \cup C_\psi$ on each maximal path ρ of Q, is a witness for $\mathrm{merge}(\phi, \psi)$.

Let $P = \mathrm{subQ}(Q, \top, C)$. Let $e = (x, y)$ be an edge of P. If e is a $\rightarrow$-edge, then, by definition of $\mathrm{merge}(\phi, \psi)$, and since D is a string, either *(i)* $\mathrm{merge}(\phi, \psi)(x) = \psi(x)$ and $\mathrm{merge}(\phi, \psi)(y) = \psi(y)$, *(ii)* $\mathrm{merge}(\phi, \psi)(x) = \phi(x)$ and $\mathrm{merge}(\phi, \psi)(y) = \phi(y)$, or *(iii)* e is a leaf edge of P and $\mathrm{merge}(\phi, \psi)(x)$ is the last position of D. Thus the edge condition required by e is satisfied.

Assume, on the other hand, that e is a $\Rightarrow$-edge. If e is a leaf edge of P then any value for $\mathrm{merge}(\phi, \psi)(x)$ satisfies e. Otherwise, assume w.l.o.g. that $\mathrm{merge}(\phi, \psi)(y) = \phi(y)$ (the other case is analogous). Towards a contradiction, assume that e is violated by $\mathrm{merge}(\phi, \psi)$, i.e., $\mathrm{merge}(\phi, \psi)(x) \geq \mathrm{merge}(\phi, \psi)(y)$. Since ϕ is a top matching we have that $\phi(x) < \phi(y) = \mathrm{merge}(\phi, \psi)(y)$. Therefore, $\mathrm{merge}(\phi, \psi)(x) = \psi(x)$ and $\mathrm{merge}(\phi, \psi)(x) \neq \phi(x)$. But this would mean that $\mathrm{merge}(\phi, \psi)(x) = \psi(x)$ and $\psi(x) > \phi(x)$, which contradicts the definition of $\mathrm{merge}(\phi, \psi)(x)$. Therefore we can conclude that $\mathrm{merge}(\phi, \psi)(x)$ is a top matching for Q on D.

Finally, notice that it immediately follows from the definitions that $\phi \preceq \mathrm{merge}(\phi, \psi)$ and $\psi \preceq \mathrm{merge}(\phi, \psi)$. □

It now follows that the set of all top matchings has a lattice structure.

LEMMA 5.19. *Let Ψ be the set of all top matchings of Q on D. Then $(\Psi, \preceq)$ is a lattice.*

PROOF. Consider two top matchings $\phi, \psi \in \Psi$. If $\phi \preceq \psi$, then ϕ is the greatest lower bound for (ϕ, ψ) and ψ is the least upper bound. Assume that neither $\psi \preceq \phi$,

nor $\phi \preceq \psi$. Now consider the matching $\text{merge}(\phi, \psi)$[7] from Definition 5.17. By Lemma 5.18, $\text{merge}(\phi, \psi)$ is a top matching for Q on D and $\phi \preceq \text{merge}(\phi, \psi)$ and $\psi \preceq \text{merge}(\phi, \psi)$ hold. We show that $\text{merge}(\phi, \psi)$ is the least upper bound for (ϕ, ψ).

To this end, let χ be a top matching such that $\phi \preceq \chi$, $\psi \preceq \chi$. Then, for any $x \in \text{Dom}(\chi)$, either $\text{merge}(\phi, \psi)(x)$ is undefined or $\text{merge}(\phi, \psi)(x) \geq \chi(x)$. Thus, $\text{merge}(\phi, \psi) \preceq \chi$ and we can conclude that $\text{merge}(\phi, \psi)$ is indeed the least upper bound for ϕ and ψ.

The proof that every pair in Ψ has a unique greatest lower bound is symmetrical. □

We say that a top matching is *maximal* for Q on D if it is maximal w.r.t. $\preceq$. Notice that Lemma 5.19 allows us to talk about *the* maximal top matching for Q on D.

COROLLARY 5.20. *Let $S \subseteq Edges(Q)$, and Ψ_S be the set of all top matchings w.r.t. S for Q on D. Then $(\Psi_S, \preceq)$ is a lattice.*

PROOF. If S is not a subset of some cut of Q, $(\Psi_S, \preceq)$ is empty, in which case we are done. Assume now that S is a subset of some cut of Q. Given two top matchings ϕ and ψ in Ψ_S, it is enough to notice that the top matching $\text{merge}(\phi, \psi)$ defined in the proof of Lemma 5.19 belongs to Ψ_S. The corollary then follows from that proof. □

We define a top matching to be *maximal* w.r.t. a set $S \subseteq \text{Edges}(Q)$ if it is the maximal top matching w.r.t. $\preceq$, *such that S is a subset of its witness cut.* Again, we can now talk about *the* maximal top matching w.r.t. S of Q on D.

We now prove two more detailed lemmas which will allow us to combine matchings or infer the existence of particular matchings. Here, for a pattern Q, a top matching ϕ for Q and a subquery P of Q, we write $\phi_{|P}$ for ϕ restricted to P, i.e., $\phi_{|P}(x) = \phi(x)$ if $x \in \text{Nodes}(P)$ and $\phi_{|P}(x)$ is undefined otherwise.

LEMMA 5.21. *Let $D = D_1 \cdot D_2$. Let ϕ be a top matching for Q on D, and let $Border(D_1, D_2, \phi) = C$. For $e \in C$, let ψ be a top matching for $subQ(Q, e, \bot)$ on D_2. Then, $merge(\phi, \psi)$ is a top matching for Q on D.*

PROOF. Let P denote $\text{subQ}(Q, e, \bot)$. First, observe that, for every node u not occurring in P, $\text{merge}(\phi, \psi)(u) = \phi(u)$, and therefore every edge not occurring in P is satisfied. Furthermore, it follows from Lemma 5.18 that $\text{merge}(\phi, \psi)_{|P}$ is a top matching for P on D_2. Hence all edges in P, except for e, are also satisfied by $\text{merge}(\phi, \psi)$.

So, it only remains to show that also e is satisfied by $\text{merge}(\phi, \psi)$. If ψ is a top matching on D_2 w.r.t. e, we have nothing to prove for e. Otherwise, there are two cases. If $e = (x, y)$ is a $\Rightarrow$-edge, then $\text{merge}(\phi, \psi)(x) = \phi(x) \in \text{Nodes}(D_1)$ and $\text{merge}(\phi, \psi)(y) \in \text{Nodes}(D_2)$, which satisfies e. If $e = (x, y)$ is a $\rightarrow$-edge, $\text{merge}(\phi, \psi)(x) = \phi(x)$ must map to the last position of D_1 (by the fact that ϕ is a valid top matching) and $\text{merge}(\phi, \psi)(y)$ must map to the first position of D_2 (as both ϕ and ψ are valid top matchings), and hence e is satisfied. □

[7] In lattice theory, the common term would be $join(\phi, \psi)$, but we use $\text{merge}(\phi, \psi)$ here to prevent confusion with the terminology of the join algorithm we present later in this section.

LEMMA 5.22. *Let $C, S \subseteq Edges(Q)$ such that C is a cut of Q containing only $\Rightarrow$-edges and no edge in S is a descendant of an edge in C. Let $P = subQ(Q, \top, C)$. Then there exists a top matching w.r.t. S of Q on D if and only if there exists a top matching w.r.t. S of P on D. In particular,*

(1) if ϕ is a top matching w.r.t. S of P on D, then ϕ is also a top matching w.r.t. S of Q on D, and

(2) if ϕ is the maximal top matching w.r.t. S of Q on D, then $\phi_{|P}$ is the maximal top matching w.r.t. S of P on D.

PROOF. Notice that the complete lemma follows from (1) and (2). Claim (1) follows immediately from the fact that C only contains $\Rightarrow$-edges.

It remains to prove Claim (2). Notice that, since no edge in S is a descendant of an edge in C, $\phi_{|P}$ is still a top matching w.r.t. S. Furthermore, since ϕ is a top matching of Q on D, and P is a subquery of Q, $\phi_{|P}$ is a valid top matching of P on D. It therefore only remains to show that $\phi_{|P}$ is the maximal top matching w.r.t. S of P on D.

Towards a contradiction, assume that there is a top matching ψ w.r.t. S of P on D such that $\phi_{|P} \prec \psi$. Then, define the top matching ϕ' as follows

$$\phi'(x) = \begin{cases} \psi(x) & \text{if } \psi(x) \text{ is defined} \\ \phi(x) & \text{otherwise} \end{cases}$$

We show that ϕ' is a top matching w.r.t. S of Q on D such that $\phi \prec \phi'$. Since ϕ was the maximal such top matching this would give the desired contradiction. To this end, we first argue that, since ϕ and ψ are top matchings w.r.t. S, ϕ' is also a top matching w.r.t. S. Since ϕ and ψ are both valid top matchings, the only edges that are potentially not satisfied by ϕ' are edges (x, y) such that $\psi(x)$ is defined, but $\psi(y)$ is not. By definition of P, all such edges belong to C.

Therefore, it only remains to show that for every $e = (x, y) \in C$, e is satisfied by ϕ'. As $e \in C$, e is a $\Rightarrow$-edge. Also, e can only be unsatisfied by ϕ' if both $\phi'(x)$ and $\phi'(y)$ are defined. Assume this is the case. Since y is not a node in P, this means that $\phi(y)$, and thus also $\phi(x)$, is defined. Further, $\phi'(x) = \psi(x) \geq \phi(x)$, as $\phi_{|P} \prec \psi$, and $\phi'(y) = \phi(y)$. Hence, e is satisfied, and thus ϕ' is a top matching w.r.t. S of Q on D. Finally, by definition of ϕ' and as $\phi_{|P} \prec \psi$, $\phi \prec \phi'$ which yields the desired contradiction. □

In order to prove the correctness theorem, we have to state one more lemma. The following lemma, however, is fairly technical and tailored to one specific situation in the correctness proof — it contains the core of the correctness of the test in lines 19–22. We start with defining the concept of *bound edges.*

DEFINITION 5.23 (BOUND EDGES). Let $C \subseteq \text{Edges}(Q)$ be a (partial) cut and $e, e' \in \text{Edges}(Q)$ such that e' is a descendant of e. Then, we denote by $\text{Bound}(e, e', C)$ the set of all triples (e_1, e_2, c_1) such that

(1) e_1 and e_2 are both bridge edges which lie on the path from e to e',

(2) $\|e_1\|_{\text{bw}} + 1 = \|e_2\|_{\text{bw}}$,

(3) $c_1 \in C$ is a direct descendant of e_1, and

(4) all edges on the path from x_2, the source node of e_2, to c_1 are $\rightarrow$-edges.

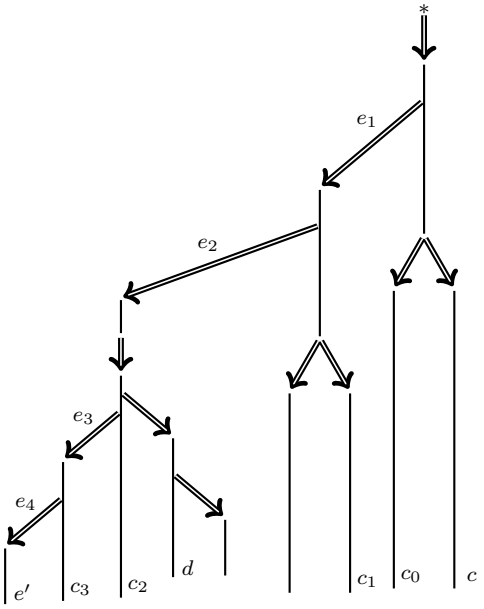


Fig. 10. Illustration of a possible situation with the preconditions of Lemma 5.24. We have that $c, c_0, \ldots, c_3$ are in C.

In case (4), we say that e_2 is *bound* w.r.t. c_1. We illustrate the definition of bound edges in Figure 10. In the figure, e_3 is bound w.r.t. c_2 and e_4 is bound w.r.t. c_3. However, e_2 is not bound w.r.t. c_1 because there is a $\Rightarrow$-edge on the path rom the source node of e_2 to c_1.

Before stating the lemma, we have to set the scene. Let P be a query and e_{top} be the root edge of P. Let D be a string and $C \subseteq \text{Edges}(P)$ a cut of $\text{bw}_{i-1}(P)$, for some i. Let ϕ be the maximal matching w.r.t. C of $\text{bw}_{i-1}(P)$ on D. Let $c \in C$ with $\|c\|_{\text{bw}} = 0$ and $e' \in \text{Edges}(P)$ with $\|e'\|_{\text{bw}} = i$. Let $e_1 = (x_1, y_1), \ldots, e_i = (x_i, y_i)$ denote the bridge edges on the path from e_{top} to e' such that $\|e_k\|_{\text{bw}} = k$, for $1 \leq k \leq i$. Let $c_0 \in C$ be a direct descendant of e_{top} and for each k, with $1 \leq k < i$, we choose an arbitrary but fixed $c_k \in C$ such that

—e_{k+1} is bound w.r.t. c_k, if such a c_k exists, and
—c_k is a direct descendant of e_k otherwise.

Notice that c_k is always a direct descendant of e_k. Figure 10 has an illustration of how these preconditions could look like for a query of width four.

Lemma 5.24. *With the above notation, assume that the following top matchings exist:*

—*a top matching ω w.r.t. $\{c_0, e'\}$ of P and,*
—*for each $(e_j, e_{j+1}, c_j) \in Bound(e_{top}, e', C)$ a top matching ω_j w.r.t. $\{e', c_j\}$ for $subQ(P, e_j, \bot)$.*

Then, there exists a top matching ϕ' w.r.t. $C \cup \{e'\}$ for $bw_i(P)$.

PROOF. We prove the lemma by constructing the desired matching ϕ'. Let $P_{e'}$ be the query obtained from $\mathrm{bw}_i(P)$ by deleting all edges f which have $\|e_{\mathrm{top}}, f\| = i$, and do not belong to the path from e_{top} to e'. Due to Lemma 5.22, there exists a top matching w.r.t. $C \cup \{e'\}$ of $\mathrm{bw}_i(P)$ if and only if there exists a top matching w.r.t. $C \cup \{e'\}$ on $P_{e'}$. Hence, we can restrict attention to $P_{e'}$.

Recall from the discussion preceding the lemma statement that ϕ is the maximal matching of w.r.t. C of $\mathrm{bw}_{i-1}(P)$ on D. Now, first assume that, for every j with $1 \leq j \leq i$, $\omega(x_j) \geq \phi(x_j)$. Then, let ϕ' be defined for all $x \in \mathrm{Nodes}(P_{e'})$ as follows: $\phi'(x) = \phi(x)$ if $\phi(x)$ is defined, and $\phi'(x) = \omega(x)$, on the path from y_i to e'. Then, ϕ' is a top matching w.r.t. $C \cup \{e'\}$ of $P_{e'}$ and we are done.

The remaining case is therefore that there exists a node x_j such that $\omega(x_j) < \phi(x_j)$. We first argue that $\omega(x_1) \geq \phi(x_1)$ must hold. Towards a contradiction, assume that $\omega(x_1) < \phi(x_1)$. Then, the top matching ψ obtained from ω and ϕ by taking

—ω on all nodes y on the path from e_{top} to c_0 for which $\omega(y) < \phi(y)$, and

—ϕ everywhere else,

is a top matching w.r.t. C of $\mathrm{bw}_{i-1}(P)$. Notice that x_1 lies on the path from e_{top} to c_0. Therefore, since $\psi(x_1) < \phi(x_1)$, we would have that $\psi \not\preceq \phi$ which contradicts the maximality of ϕ. Hence, $\omega(x_1) \geq \phi(x_1)$.

So, we have now shown that $\omega(x_1) \geq \phi(x_1)$ and that there exists a $k > 1$ such that $\omega(x_k) < \phi(x_k)$. It follows that there must be a $1 < j \leq i$ such that $\omega(x_j) < \phi(x_j)$ and $\omega(x_{j-1}) \geq \phi(x_{j-1})$. We now show that e_j is bound w.r.t. c_{j-1}. Towards a contradiction, suppose that e_j is not bound. By definition of c_{j-1}, this means that there is a $\Rightarrow$-edge on the path from x_j to c_{j-1}. Let $e_{\mathrm{desc}} = (x_{\mathrm{desc}}, y_{\mathrm{desc}})$ be the closest such edge to x_j. We will now construct a top matching χ w.r.t. C of $\mathrm{bw}_{i-1}(P)$ such that $\chi \not\preceq \phi$, hence contradicting the maximality of ϕ. For any $x \in \mathrm{Nodes}(\mathrm{bw}_{i-1}(P))$, let $\chi(x) = \omega(x)$ for all nodes on the path from y_{j-1} to x_{desc} and $\chi(x) = \phi(x)$, otherwise. We show that χ is a top matching of $\mathrm{bw}_{i-1}(P)$. To this end, notice that all edges except e_j, e_{j-1}, and e_{desc} are satisfied by χ because ϕ and ω are valid top matchings. Now,

—e_j is satisfied because $\chi(x_j) = \omega(x_j) < \phi(x_j) < \phi(y_j) = \chi(y_j)$;

—e_{j-1} is satisfied because $\chi(x_{j-1}) = \phi(x_{j-1}) \leq \omega(x_{j-1}) < \omega(y_{j-1}) = \chi(y_{j-1})$; and

—e_{desc} is satisfied because $\chi(x_{\mathrm{desc}}) = \omega(x_{\mathrm{desc}}) < \phi(x_{\mathrm{desc}}) < \phi(y_{\mathrm{desc}}) = \chi(y_{\mathrm{desc}})$, where $\omega(x_{\mathrm{desc}}) < \phi(x_{\mathrm{desc}})$ follows from the facts that $\omega(x_j) < \phi(x_j)$ and the path from x_j to x_{desc} only consists of $\rightarrow$-edges.

Furthermore, by definition of χ, χ is a top matching w.r.t. C (it only differs from ϕ on the path from e_{j-1} to e_{desc}). Since $\chi(x_j) = \omega(x_j) < \phi(x_j)$, it holds that $\chi \not\preceq \phi$. Hence, we have our contradiction and we can conclude that e_j is a bound edge.

We have now shown that at least one edge in $\{e_2, \ldots, e_i\}$ must be a bound edge. Let j be the biggest number such that e_{j+1} is a bound edge. By the statement of the lemma there then exists a top matching ω_j w.r.t. $\{e', c_j\}$ for $\mathrm{subQ}(P, e_j, \bot)$.

We now show that for all k with $j < k \leq i$, we have $\omega_j(x_k) \geq \phi(x_k)$. Towards a contradiction, assume that $\omega_j(x_k) < \phi(x_k)$ for some $j < k \leq i$. As e_{j+1} is a bound

edge, and ω_j and ϕ both are top matchings w.r.t. $\{c_j\}$, we know that $\omega_j(x_{j+1}) = \phi(x_{j+1})$. As before, there then has to be a $\ell > j+1$ such that $\omega_j(x_\ell) < \phi(x_\ell)$ but $\omega_j(x_{\ell-1}) \geq \phi(x_{\ell-1})$. Using the same argument as above, one can conclude that the bridge e_ℓ is bound. However, as $\ell > j+1$ this contradicts the fact that we have chosen the largest j such that e_{j+1} is bound.

Using the fact that for all k with $j < k \leq i$, we have $\omega_j(x_k) \geq \phi(x_k)$, and in particular that $\omega_j(x_i) \geq \phi(x_i)$, we can now define ϕ', the top matching w.r.t. $C \cup \{e'\}$ on $\mathrm{bw}_i(P)$ as follows. For any $x \in \mathrm{Edges}(\mathrm{bw}_i(P))$, let $\phi'(x) = \phi(x)$ if $\phi(x)$ is defined, and $\phi'(x) = \omega_j(x)$, if x is on the path from y_i to e'. We first argue that ϕ' is a top matching of $P_{e'}$. To this end, notice that all edges except for e_i are satisfied by ϕ' due to the fact that ϕ and ω_j are valid top matchings. For e_i, we know that it is a $\Rightarrow$-edge because e_i is a bridge and, furthermore, $\phi'(x_i) = \phi(x_i) \leq \omega_j(x_i) < \omega_j(y_i) = \phi'(y_i)$. Hence, e_i is also satisfied. Finally, since ϕ is a top matching w.r.t. C, and ω_j is a top matching w.r.t. $\{e'\}$ we obtain by definition that ϕ' is a top matching w.r.t. $C \cup \{e'\}$. Hence, ϕ' is a top matching w.r.t. $C \cup \{e'\}$ of $P_{e'}$. This concludes the proof of Lemma 5.24. □

We are now finally ready to prove the correctness of Algorithm 1.

LEMMA 5.25 JOIN CORRECTNESS. *Let $D = D_1 \cdot D_2$ and $e_1, e_2 \in Edges(Q)$. Let Q, $Triples(D_1)$, $Triples(D_2)$, e_{top}, and e_{bot} be given as input to Algorithm 1. The algorithm includes (e_{top}, e_{bot}, e) in its output if and only if there is a top matching w.r.t. $\{e_{bot}, e\}$ of $subQ(Q, e_{top}, \bot)$ on D.*

PROOF. Notice that Algorithm 1 is correct when there does not exist a top matching w.r.t. $\{e_{\mathrm{bot}}\}$ of $\mathrm{subQ}(Q, e_{\mathrm{top}}, \bot)$ on D. The program then does not find an appropriate value for c in line 3 and returns in line 8.

For the remainder of the proof, assume that there exists a top matching w.r.t. $\{e_{\mathrm{bot}}\}$ and let $P = \mathrm{subQ}(Q, e_{\mathrm{top}}, \bot)$. Let $Q_{e_{\mathrm{top}}, e_{\mathrm{bot}}}$ be the subquery of Q consisting only of the edges and nodes on the direct path from e_{top} to e_{bot}. We will prove the correctness of the algorithm by induction on the number of iterations of the while-loop, but in order to state the invariants in a uniform manner, we have to do a slight abuse of notation. We will define the edges in $Q_{e_{\mathrm{top}}, e_{\mathrm{bot}}}$, i.e., all edges on the path from e_{top} to e_{bot} to have bridge width -1. Hence, $\mathrm{bw}_{-1}(P) := Q_{e_{\mathrm{top}}, e_{\mathrm{bot}}}$ and $\|e_{\mathrm{top}}, e\| := -1$ for all edges in $Q_{e_{\mathrm{top}}, e_{\mathrm{bot}}}$. We also use T_{-1} to refer to the contents of T_0 before the first iteration of the while-loop. Under this notation, we will inductively prove the following statement:

Every time the algorithm reaches the while statement on line 12, the following invariants hold. Here, j and c are the variables used in the algorithm. (Recall that c on lines 4 and 22 of the algorithm refer to the same edge in Q.)

(I1) If $j \geq 0$, then for all e such that $\|e_{\mathrm{top}}, e\| < j$,

$(e_{\mathrm{top}}, e_{\mathrm{bot}}, e) \in \mathrm{Triples}(D)$ if and only if $(e_{\mathrm{top}}, e_{\mathrm{bot}}, e) \in T_{-1}(D) \cup \cdots \cup T_{j-1}(D)$.

(I2) If $j \geq 0$, then there is a top matching w.r.t. $\{e_{\mathrm{bot}}\}$ of $\mathrm{bw}_{j-1}(P)$ on D. The maximal such matching ϕ_{j-1} has $\mathrm{Border}(D_1, D_2, \phi_{j-1}) = C$, with $c \in C$.

In other words, when $j = 0$, then (I1) states that $(e_{\mathrm{top}}, e_{\mathrm{bot}}, e_{\mathrm{bot}}) \in \mathrm{Triples}(D)$ if and only if $(e_{\mathrm{top}}, e_{\mathrm{bot}}, e_{\mathrm{bot}}) \in T_{-1}(D)$ and (I2) states that if $c \in C$, then there

is a top matching w.r.t. $\{e_{\text{bot}}\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D and the maximal such matching $\phi_{e_{\text{top}},e_{\text{bot}}}$ has $\text{Border}(D_1, D_2, \phi_{e_{\text{top}},e_{\text{bot}}}) = \{c\} = C$. Notice that the lemma follows from these invariants.

We prove the invariants by induction on j. The base case is $j = 0$.

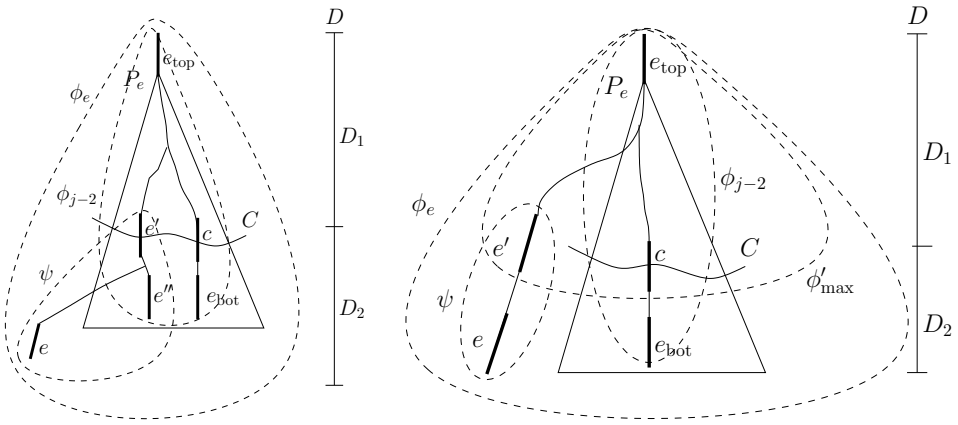
We first prove invariant (I1). Suppose $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}}) \in T_{-1}(D)$ (i.e., $T_0(D)$ on line 6 in the algorithm). Then, there is a c such that $(e_{\text{top}}, c, c) \in \text{Triples}(D_1)$ and $(c, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D_2)$ such that c is as close to the leaves as possible (i.e., a "lowermost" c). By definition of matching triples, this means that there is a top matching w.r.t. $\{c\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D_1 and a top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{subQ}(Q_{e_{\text{top}},e_{\text{bot}}}, c, \bot)$ on D_2. By combining these top matchings, we get a top matching ϕ w.r.t. $\{e_{\text{bot}}\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D. Since every edge leaving the path from e_{top} to e_{bot} in Q is a $\Rightarrow$-edge, ϕ is by Lemma 5.22 also a top matching w.r.t. $\{e_{\text{bot}}\}$ of P on D, and hence $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D)$. Conversely, suppose that $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D)$. By definition of matching triples and Lemma 5.22, there then is a top matching w.r.t. $\{e_{\text{bot}}\}$ for $Q_{e_{\text{top}},e_{\text{bot}}}$ on D. Let $\phi_{e_{\text{top}},e_{\text{bot}}}$ be the maximal such top matching and let c be the unique edge in $\text{Border}(D_1, D_2, Q_{e_{\text{top}},e_{\text{bot}}})$. It follows that there is a top matching w.r.t. $\{c\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D_1 and a top matching w.r.t. $\{e_{\text{bot}}\}$ for $\text{subQ}(Q_{e_{\text{top}},e_{\text{bot}}}, c, \bot)$ on D_2. This, in turn, means by definition of matching triples that $(e_{\text{top}}, c, c) \in \text{Triples}(D_1)$ and $(c, e_{\text{bot}}, e_{\text{bot}}) \in \text{Triples}(D_2)$. Thus, the condition on line 3 is satisfied, and $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})$ is included in the set $T_{-1}(D)$ (i.e., $T_0(D)$ on line 6 in the algorithm). This concludes the proof of invariant (I1) for $j = 0$.

For invariant (I2), we can immediately conclude from $c \in C$ that there is a top matching w.r.t. $\{e_{\text{bot}}\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D. Indeed, the top matching ϕ we constructed above when proving (I1) is such a matching with $\text{Border}(D_1, D_2, \phi) = \{c\}$. Hence, there also exists a maximal top matching w.r.t. $\{e_{\text{bot}}\}$ of $Q_{e_{\text{top}},e_{\text{bot}}}$ on D due to Corollary 5.20. Denote this maximal top matching by $\phi_{e_{\text{top}},e_{\text{bot}}}$. For proving (I2), it now suffices to show that $\phi_{e_{\text{top}},e_{\text{bot}}}$ has $\text{Border}(D_1, D_2, \phi_{e_{\text{top}},e_{\text{bot}}}) = \{c\}$. Assuming this is not the case, there are two possibilities. First, if the single edge in $\text{Border}(D_1, D_2, \phi_{e_{\text{top}},e_{\text{bot}}})$ is lower than c, we immediately get a contradiction as c is chosen to be the lowermost edge satisfying the condition in line 3 of Algorithm 1. Otherwise, if the single edge in $\text{Border}(D_1, D_2, \phi_{e_{\text{top}},e_{\text{bot}}})$ is higher than c, then $\phi \not\preceq \phi_{e_{\text{top}},e_{\text{bot}}}$. This contradicts the maximality of $\phi_{e_{\text{top}},e_{\text{bot}}}$, and concludes the proof of invariant (I2) for $j = 0$ and the induction base case.

For the remainder of the proof, we fix a $j > 0$ and assume that (I1) and (I2) hold up to $j-1$. Let ϕ_{j-2} denote the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-2}(P)$ on D. By the induction hypothesis for (I2) ϕ_{j-2} exists and $\text{Border}(D_1, D_2, \phi_{j-2}) = C$.

For the induction on (I1), it suffices to consider triples $(e_{\text{top}}, e_{\text{bot}}, e)$ with $\|e_{\text{top}}, e\| = j - 1$. For $e \in \text{Edges}(P)$ with $\|e_{\text{top}}, e\| = j - 1$, let P_e be the query obtained from $\text{bw}_{j-1}(P)$ by deleting all edges f which either are (1) descendants of e_{bot} or (2) have $\|e_{\text{top}}, f\| = j - 1$, and do not belong to the path from e_{top} to e or to the path from e_{top} to e_{bot}. Notice that, according to Lemma 5.22, there exists a top matching w.r.t. $\{e_{\text{bot}}, e\}$ of P_e on D if and only if there exists a top matching w.r.t. $\{e_{\text{bot}}, e\}$ of P on D (all edges leaving P_e are $\Rightarrow$-edges). Therefore, we will work with P_e instead of P in the following.

Assume $(e_{\text{top}}, e_{\text{bot}}, e)$ is included in $T_{j-1}(D)$. Then, there are three possibilities.



(a) Case 2. (b) Case 3.

Fig. 11. Two cases in the proof of Lemma 5.25.

Cases 2 and 3 are graphically represented in Figure 11.

Case 1: The first case (line 15) is that $(c, e_{\text{bot}}, e) \in \text{Triples}(D_2)$. Then, there is a top matching ψ w.r.t. $\{e_{\text{bot}}, e\}$ for $\text{subQ}(P_e, c, \bot)$ on D_2. Let ϕ_e be $\text{merge}(\psi, \phi_{j-2})$ (see Definition 5.17). We show that ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ for P_e on D, from which it then follows that $(e_{\text{top}}, e_{\text{bot}}, e)$ belongs to $\text{Triples}(D)$. Since $c \in C$, it follows immediately from Lemma 5.21 that ϕ_e is a top matching for P_e on D. Further, as ψ is a top matching w.r.t. $\{e_{\text{bot}}, e\}$, ϕ_{j-2} is a top matching w.r.t. $\{e_{\text{bot}}\}$, and e is not matched by ϕ_{j-2} (since $\|e\|_{\text{bw}} = j-1$), it follows from Definition 5.17 that ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$.

Case 2: The second case (line 17) is similar to the first. Now, there is an $e' \in C$, such that $e' \neq c$, and an e'' such that $(e', e'', e) \in \text{Triples}(D_2)$. Then, there is a top matching ψ w.r.t. $\{e\}$ for $\text{subQ}(P_e, e', \bot)$ on D_2. Let ϕ_e be $\text{merge}(\psi, \phi_{j-2})$. We show that ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ for P_e on D, from which it then follows that $(e_{\text{top}}, e_{\text{bot}}, e)$ belongs to $\text{Triples}(D)$. It follows immediately from Lemma 5.21 that ϕ_e is a top matching for P_e on D.

To see that ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ notice that (1) ψ is a top matching w.r.t. $\{e\}$ and e is not matched by ϕ_{j-2} as $\|e\|_{\text{bw}} = j-1$ and (2) ϕ_{j-2} is a top matching w.r.t. $\{e_{\text{bot}}\}$ and e_{bot} is not matched by ψ. Indeed, as $c \neq e'$, e_{bot} is not a descendant of e' and hence e_{bot} does not occur in $\text{subQ}(P_e, e', \bot)$ for which ψ is a top matching. From Definition 5.17 it now follows that ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$.

Case 3: If the third case (lines 19–22) applies, it holds that the first two cases did not apply and there is an e' s.t. $(e_{\text{top}}, c, e') \in \text{Triples}(D_1)$ and $(e', e, e) \in \text{Triples}(D_2)$. Since the first case did not apply, we know that e' does not belong to C. Since $(e', e, e) \in \text{Triples}(D_2)$ there is a top matching w.r.t. $\{e\}$ for $\text{subQ}(P_e, e', \bot)$ on D_2. Let ψ be the maximal such matching. By induction we know that the maximal top matching ϕ_{j-2} of $\text{bw}_{j-2}(P)$ w.r.t. $\{e_{\text{bot}}\}$ exists and that $\text{Border}(D_1, D_2, \phi_{j-2}) = C$, with $c \in C$ (c is the unique edge in C on the path from e_{top} to e_{bot}).

For $e_k = e_{\text{top}}$, on lines 19–22, there must be some $c_0 \in C$ which is a direct descen-

dant of e_{top}, such that $(e_{\text{top}}, c_0, e') \in \text{Triples}(D_1)$. Hence, there is a top matching ω w.r.t. $\{c_0, e'\}$ of P_e on D_1. Further, when $e_k \in \text{Bridges}(e_{\text{top}}, e')$, it follows from lines 19–22 that, for all bound triples (e_k, e_{k+1}, c_k) in $\text{Bound}(e_{\text{top}}, e', C)$, there is a triple (e_k, c_k, e') in $\text{Triples}(D_1)$. According to Lemma 5.24, this means that there is a top matching ϕ' for P_e on D_1 w.r.t. $C \cup \{e'\}$. Let ϕ'_{max} be the maximal such matching.

Now, we can combine ϕ'_{max}, ϕ_{j-2}, and ψ to construct a top matching ϕ_e w.r.t. $\{e_{\text{bot}}, e\}$ for P_e on D. For every $x \in \text{Nodes}(P_e)$, let $\phi_e(x) := \phi'_{\text{max}}(x)$ if $\phi'_{\text{max}}(x)$ is defined, $\phi_e(x) := \psi(x)$, if x is on the path from e' to e, and $\phi_e(x) := \phi_{j-2}(x)$, otherwise. Notice that $\phi_e(x) = \phi_{j-2}(x)$ for all x which are target nodes of edges in C, or descendants thereof. Now, notice that since ψ is a top matching w.r.t. $\{e\}$ and ϕ_{j-2} is a top matching w.r.t. $\{e_{\text{bot}}\}$, ϕ_e is a top matching w.r.t. $\{e_{\text{bot}}, e\}$. Furthermore, it follows from the facts that ϕ'_{max} is a top matching w.r.t. $C \cup \{e'\}$ on D_1, ψ is a top matching for $\text{subQ}(P_e, e', \bot)$ on D_2 and $\text{Border}(D_1, D_2, \phi_{j-2}) = C$, that ϕ_e is a top matching of P_e on D. Hence, ϕ_e is the desired top matching w.r.t. $\{e_{\text{bot}}, e\}$ for P_e on D. This concludes the proof of Case 3.

For future reference, we prove the following claim.

Claim 5.26. *The maximal matching ϕ'_e of P_e w.r.t. $\{e_{bot}, e\}$ has Border$(D_1, D_2, \phi'_e) = C \cup \{e'\}$.*

Proof of Claim 5.26. Let $C'_e = \text{Border}(D_1, D_2, \phi'_e)$. Notice first that, by Lemma 5.22, it holds that ϕ'_e, restricted to $\text{bw}_{j-2}(P)$, equals ϕ_{j-2}. Hence, $C \subseteq \text{Border}(D_1, D_2, \phi'_e)$. Furthermore, since P_e consists of $\text{bw}_{j-2}(P)$ extended with a single path of edges to e, C'_e can contain at most one additional edge in addition to C. Assuming first that C'_e does not contain e', nor a descendant of e', it follows that $\phi_e \not\preceq \phi'_e$, which contradicts the maximality of ϕ'_e. Now, suppose that C'_e contains a descendant e_{max} of e'. Then, setting $e' = e_{\text{max}}$ on line 22 will satisfy all conditions in this test. Indeed, ϕ'_e witnesses the existence of all matching triples in the test. But then we again obtain a contradiction, since e' was chosen to be the lowermost edge satisfying this test in line 23. We can therefore conclude that $e' \in C'_e$ and thus $C'_e = C \cup \{e'\}$. This concludes the proof of Claim 5.26. □

Together, the above three cases prove that, for edges e with $\|e_{\text{top}}, e\| = j - 1$, if $(e_{\text{top}}, e_{\text{bot}}, e) \in T_{j-1}(D)$, then $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$.

It now remains to show the converse, i.e., that for edges e with $\|e_{\text{top}}, e\| = j - 1$, $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$ implies $(e_{\text{top}}, e_{\text{bot}}, e) \in T_{j-1}(D)$. Therefore, assume that $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$ and hence there exists a top matching w.r.t. $\{e_{\text{bot}}, e\}$ of P_e on D. Denote the maximal such matching by ψ_e. Let e' be the unique edge on the path from e_{top} to e which is included in $\text{Border}(D_1, D_2, \psi_e)$. We distinguish three possibilities.

First, suppose $e' = c$. Then, the restriction of ψ_e to $\text{subQ}(P_e, c, \bot)$ is a top matching w.r.t. $\{e, e_{\text{bot}}\}$ of $\text{subQ}(P_e, c, \bot)$ on D_2, which means that $(c, e_{\text{bot}}, e) \in \text{Triples}(D_2)$. Thus the condition on line 15 is satisfied, and $(e_{\text{top}}, e_{\text{bot}}, e)$ is included in $T_{j-1}(D)$.

Second, suppose $e' \neq c$, but $e' \in \text{Edges}(\text{bw}_{j-2}(P_e))$. We first show that then $e' \in C$. To this end, notice that, by construction of P_e, e is a leaf edge of P_e. Therefore, ψ_e is also the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ for P_e on D. From Lemma 5.22 it then follows that ψ_e, restricted to $\text{bw}_{j-2}(P_e)$, is the maximal top

matching w.r.t. $\{e_{\text{bot}}\}$ for $\text{bw}_{j-2}(P)$ on D, i.e. ϕ_{j-2}. Hence, $\text{Border}(D_1, D_2, \psi_e) \cap \text{Edges}(\text{bw}_{j-2}(P_e)) = \text{Border}(D_1, D_2, \phi_{j-2}) = C$, and as $e' \in \text{Border}(D_1, D_2, \psi_e)$ and $e' \in \text{Edges}(\text{bw}_{j-2}(P_e))$ it thus follows that $e' \in C$. Furthermore, the restriction of ψ_e to $\text{subQ}(P_e, e', \bot)$ is a top matching of $\text{subQ}(P_e, e', \bot)$ w.r.t. $\{e\}$ on D_2, which means that $(e', e'', e) \in \text{Triples}(D_2)$, for some e'' which is a direct descendant of e'. Thus the condition on line 17 is satisfied, and $(e_{\text{top}}, e_{\text{bot}}, e)$ is included in $T_{j-1}(D)$.

Third, suppose that $e' \notin \text{Edges}(\text{bw}_{j-2}(P))$, and hence $\|e'\|_{\text{bw}} = j - 1$. Notice that $C \subseteq \text{Edges}(\text{bw}_{j-2}(P))$, as the new edges in the inner for loop are added into a new set C'. Hence, $e' \notin C$. As $e' \in \text{Border}(D_1, D_2, \psi_e)$, it holds that ψ_e, restricted to $\text{subQ}(P_e, e', \bot)$, is a top matching w.r.t. $\{e\}$ for $\text{subQ}(P_e, e', \bot)$ on D_2. As $\|e'\|_{\text{bw}} = \|e\|_{\text{bw}} = j - 1$, e is a direct descendant of e', and hence $(e', e, e) \in \text{Triples}(D_2)$.

Furthermore, as noted above, $C = \text{Border}(D_1, D_2, \phi_{j-2}) \subseteq \text{Border}(D_1, D_2, \psi_e)$. Therefore, ψ_e, restricted to $\text{subQ}(P_e, \top, \text{Border}(D_1, D_2, \psi_e))$ is a top matching w.r.t. $C \cup \{e'\}$ of P_e on D_1. Denote the latter top matching by ϕ'. From ϕ' it immediately follows that $(e_{\text{top}}, c', e') \in \text{Triples}(D_1)$, for any $c' \in C$. Furthermore, for any $e_k \in \text{Bridges}(e_{\text{top}}, e')$ with $\|e_{\text{top}}, e_k\| = k < j - 1$, and any $c_k \in C$ such that $\|e_{\text{top}}, c_k\| = k < j - 1$ and c_k is a descendant of e_k, notice that the restriction of ϕ' to $\text{subQ}(P_e, e_k, \bot)$ is a top matching w.r.t. $\{c_k, e'\}$ of $\text{subQ}(P_e, e_k, \bot)$ on D_1, and hence $(e_k, c_k, e') \in \text{Triples}(D_1)$. In conclusion, all conditions on line 22 are satisfied, and $(e_{\text{top}}, e_{\text{bot}}, e)$ is included in $T_{j-1}(D)$. This concludes the induction on (I1).

We now prove that condition (I2) also holds for $j > 0$. As noted before, we know by induction that ϕ_{j-2}, the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-2}(P)$ on D, exists and $\text{Border}(D_1, D_2, \phi_{j-2}) = C$. We must now show that ϕ_{j-1}, the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-1}(P)$ on D, exists and has $\text{Border}(D_1, D_2, \phi_{j-1}) = C \cup \text{Low}(C')$. Indeed, on line 28, the new value of C is set to $C \cup \text{Low}(C')$. It immediately follows from the existence of ϕ_{j-2} and Lemma 5.22 that ϕ_{j-1} exists. The rest of the proof is devoted to proving that $\text{Border}(D_1, D_2, \phi_{j-1}) = C \cup \text{Low}(C')$. We start by giving an alternative definition of ϕ_{j-1}.

To this end, let $S = \{e \mid \|e\|_{\text{bw}} = j - 1 \wedge (e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D))\}$. That is, S contains the edges which are included in $T_{j-1}(D)$ by the algorithm. For any $e \in S$, let ϕ'_e be the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ of P_e on D. Observe that for any $e \in S$, by Lemma 5.22 and the maximality of ϕ'_e, it holds that ϕ'_e, restricted to $\text{bw}_{j-2}(P)$, is ϕ_{j-2}, i.e. the maximal top matching w.r.t. $\{e_{\text{bot}}\}$ for $\text{bw}_{j-2}(P)$ on D. Notice also that as e is a leaf edge of P_e and $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$, ϕ'_e is also the maximal top matching w.r.t. $\{e_{\text{bot}}, e\}$ of P_e on D.

Let ϕ' be the matching obtained by merging ϕ_{j-2} with all ϕ'_e matchings for $e \in S$. We show that ϕ' equals the maximal matching ϕ_{j-1}. To this end, we first show that ϕ' is a top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-1}(P)$ on D. From Lemma 5.22 it follows that ϕ_{j-2} as well as ϕ'_e, for all $e \in S$, are top matchings w.r.t. $\{e_{\text{bot}}\}$ for $\text{bw}_{j-1}(P)$ on D. From Corollary 5.20 it then follows that ϕ', the least upper bound of all these top matchings, is also a top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-1}(P)$ on D. Second, we show that $\text{Dom}(\phi') = \text{Dom}(\phi_{j-1})$. To this end, first notice that, since ϕ_{j-1} is maximal and ϕ' is also a top matching w.r.t. $\{e_{\text{bot}}\}$ of $\text{bw}_{j-1}(P)$ on D, $\text{Dom}(\phi') \subseteq \text{Dom}(\phi_{j-1})$. Conversely, suppose that $\text{Dom}(\phi_{j-1}) \not\subseteq \text{Dom}(\phi')$. Let x be a lowermost node in $\text{Dom}(\phi_{j-1}) \setminus \text{Dom}(\phi')$, and let e be an edge in P of which

x is the source. Then, ϕ_{j-1} is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ of P on D, and hence $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$. It then follows that $e \in S$, and hence, by definition of ϕ', $x \in \text{Dom}(\phi')$. Contradiction. Hence, $\text{Dom}(\phi') = \text{Dom}(\phi_{j-1})$. We conclude by showing that $\phi'(x) = \phi_{j-1}(x)$ for all $x \in \text{Dom}(\phi_{j-1})$. To this end, first observe that $\phi_{j-1}(x) \leq \phi'(x)$ holds for any $x \in \text{Dom}(\phi_{j-1})$ as ϕ_{j-1} is maximal. Towards a contradiction, suppose that there is a $y \in \text{Dom}(\phi_{j-1})$ such that $\phi_{j-1}(y) < \phi'(y)$. Let c be the edge of which y is the target. As by Lemma 5.22 both ϕ_{j-1} and ϕ' restricted to $\text{bw}_{j-2}(P)$ equal ϕ_{j-2}, $\|c\|_{\text{bw}} = j-1$ must hold. Now, let e be a lowermost edge which is a descendant of c and whose source node $x \in \text{Dom}(\phi_{j-1})$. Then, $e \in S$, but ϕ_{j-1}, restricted to P_e, is a greater matching than ϕ'_e. This contradicts the maximality of ϕ'_e. We can therefore conclude that ϕ' equals ϕ_{j-1}.

We are now ready to show that $\text{Border}(D_1, D_2, \phi_{j-1})$ equals $C \cup \text{Low}(C')$. Notice first that by Lemma 5.22 it holds that ϕ_{j-1}, restricted to $\text{bw}_{j-2}(P)$, equals ϕ_{j-2}. It follows that $\text{Border}(D_1, D_2, \phi_{j-1}) \cap \text{Edges}(\text{bw}_{j-2}(P)) = \text{Border}(D_1, D_2, \phi_{j-2}) = C$. It hence suffices to show that $\text{Border}(D_1, D_2, \phi_{j-1}) \cap (\text{Edges}(\text{bw}_{j-1}(P)) \setminus \text{Edges}(\text{bw}_{j-2}(P))) = \text{Low}(C')$. To this end, we use the fact that ϕ_{j-1} equals ϕ'.

We investigate which edges of $\text{Border}(D_1, D_2, \phi')$ have bridge width $j-1$. For every $e \in S$, let $\text{Border}(e)$ denote the unique edge in P_e which is both on the path from e_{top} to e, and in $\text{Border}(D_1, D_2, \phi'_e)$. Then, let $S' = \{\text{Border}(e) \mid e \in S \wedge \|\text{Border}(e)\|_{\text{bw}} = j-1\}$. By definition of ϕ', $\text{Border}(D_1, D_2, \phi') \cap (\text{Edges}(\text{bw}_{j-1}(P)) \setminus \text{Edges}(\text{bw}_{j-2}(P))) = \text{Low}(S')$. It hences suffices to show that $\text{Low}(S') = \text{Low}(C')$ to conclude the proof. To this end, note that C' is formed by adding, for every $e \in S$ such that $\|\text{Border}(e)\|_{\text{bw}} = j-1$, an edge e' to C' on line 23 (the if-test on line 19, corresponding to case 3 in the argument above). By Claim 5.26, $e' = \text{Border}(e)$. Hence, $C' = S'$, and thus $\text{Low}(C') = \text{Low}(S')$. □

We conclude the section by proving Lemma 5.11.

LEMMA 5.11. *Let Leaf(Q) be the set of leaf edges Q. Let e_{top} be the root edge of Q, and $e_{bot} \in$ Leaf(Q) be a direct descendant of e_{top}. Then, there is a full matching of Q on D, if and only if for all $e \in$ Leaf(Q), $(e_{top}, e_{bot}, e) \in$ Triples(D).*

PROOF. First, suppose there exists a full matching ϕ for Q on D. Then for any $e \in \text{Leaf}(Q)$, as $e_{\text{bot}} \in \text{Leaf}(Q)$, ϕ is a top matching w.r.t. $\{e_{\text{bot}}, e\}$ for Q on D, and hence $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$.

Conversely, suppose that for all $e \in \text{Leaf}(Q)$, $(e_{\text{top}}, e_{\text{bot}}, e) \in \text{Triples}(D)$ and let ϕ_e be a corresponding top matching w.r.t. $\{e_{\text{bot}}, e\}$ of Q on D. Let ϕ be obtained by merging all such ϕ_e matchings. Then, by Lemma 5.18, ϕ is a top matching for Q on D. Furthermore, by the definition of merge (Definition 5.17), the witness cut of ϕ equals $\text{Leaf}(Q)$. Hence, ϕ is a full matching of Q on D. □

6. CONJUNCTIVE FORWARD XPATH

We will adapt the algorithm of Section 4 to handle the next-sibling ($\rightarrow$) and following-sibling ($\Rightarrow$) axes. However, in order to do this we need to disallow disjunction ($\vee$) and negation ($\neg$) in the pattern, leaving us with the fragment $\text{XPath}(\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge)$, which we refer to as conjunctive forward XPath. All such queries can be thought of as tree pattern queries with only label nodes, so we do not need to consider syntax nodes. Every branching in the pattern implicitly denotes a conjunction. We will show the following.

THEOREM 6.1. *Boolean incremental evaluation for an XPath*$(\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge)$ *Pattern* Q *and an XML document* D *can be performed in time* $\mathcal{O}(depth(D) \cdot \log(width(D)) \cdot poly(|Q|))$ *per update. The size of the auxiliary data structure is* $\mathcal{O}(|D| \cdot |Q|^3)$.

For a query node q of Q, let the *subpattern without siblings of* q, denoted by subtreeNoSibling(q), be the subtree of Q rooted at q from which all sibling edges leaving q (and the corresponding subtrees) are removed. Here, by *sibling edge*, we mean both "$\rightarrow$"- and "$\Rightarrow$"-edges. Notice that we only remove sibling edges that are directly attached to q, so subtreeNoSibling(q) can still contain sibling edges deeper in the pattern.

Further, for a query node q of Q, the *subpattern with only siblings of* q, denoted subtreeOnlySibling(q), is the subtree of Q rooted at q and containing all nodes reachable from q by following only sibling edges. Notice that subtreeOnlySibling(q) only contains sibling edges, and is thus a query in the fragment XPath$(\rightarrow, \Rightarrow, \wedge)$ treated in the previous section.

Let downNodes(Q) be the subset of Nodes(Q) such that for each query node $q \in$ downNodes(Q), the unique incoming edge to q in Q has type $\downarrow$ or $\Downarrow$.

The algorithm works as follows. For each node u in D we store a record R_u consisting of:

—the set of query nodes
MatchNoSibling(u) = $\{q \in \text{Nodes}(Q) \mid D \models^u \text{subtreeNoSibling}(q)\}$,

—the set of query nodes in downNodes(Q) that are satisfied in some child of u, i.e., the set MatchChild(u) = $\{q \in \text{downNodes}(Q) \mid \exists u'.\text{child}(u, u') \wedge D \models^{u'} Q[q]\}$,

—the set of query nodes that are satisfied in some descendant of u, i.e., the set MatchDesc(u) = $\{q \in \text{Nodes}(Q) \mid \exists u'.\text{descendant}(u, u') \wedge D \models^{u'} Q[q]\}$,

—for each query node q, the number of children of u that have a descendant satisfying $Q[q]$, i.e., the cardinality $\text{numDesc}^q(u)$ of the set $\{u' \mid \text{child}(u, u') \wedge \exists u'' \text{ descendant}(u', u'') \wedge D \models^{u''} Q[q]\}$.

Without loss of generality, we assume that the root node of Q does not have outgoing sibling edges. Indeed, if it has, it can never be mapped to the root of D. Therefore, it suffices to check whether MatchNoSibling(root(D)) contains root(Q) to decide whether $D \models Q$.

However, to maintain the above records, we still need to store some additional information. For each node u of D and each query node $q \in$ downNodes(Q) we also store the data structures needed to incrementally verify whether a string w satisfies subtreeOnlySibling(q). Here, w is a string formed by relabeled versions of the children of u in D, and the data structures are those maintained by the algorithm in Section 5. The concrete details about the string w are given below.

We now show how the records R_u can be updated. As in Section 4, it suffices to recompute this information for all nodes on the path of the updated node to the root. Let u be the next node to be updated, v be its parent, and $u_1, \ldots, u_k$ its children. We use MatchNoSibling(u)$_{\text{new}}$, MatchChild(u)$_{\text{new}}$, etc. to refer to the record values after the update. We assume that MatchChild(u)$_{\text{new}}$, MatchDesc(u)$_{\text{new}}$, and numDesc(u)$_{\text{new}}$ are given and show how to compute MatchNoSibling(u)$_{\text{new}}$, MatchChild(v)$_{\text{new}}$, MatchDesc(v)$_{\text{new}}$, and numDesc(v)$_{\text{new}}$. If u is a leaf node, we

have $\text{MatchChild}(u) = \text{MatchDesc}(u) = \emptyset$ and $\text{numDesc}^q(u) = 0$, for all query nodes q.

—$\text{MatchNoSibling}(u)_{\text{new}}$: For a child q' of q, we say that q' is a $\downarrow$-child (respectively, $\Downarrow$-child), if $\text{type}((q, q')) = \downarrow$ (respectively, $\Downarrow$). A $\downarrow$-child q' is *satisfied* if $q' \in \text{MatchChild}(u)_{\text{new}}$, and a $\Downarrow$-child q' if $q' \in \text{MatchChild}(u)_{\text{new}} \cup \text{MatchDesc}(u)_{\text{new}}$. Then, $q \in \text{MatchNoSibling}(u)_{\text{new}}$ if q's label matches the label of u and all its $\downarrow$- and $\Downarrow$-children are satisfied.

—$\text{MatchChild}(v)_{\text{new}}$: For any $q \in \text{downNodes}(Q)$, to know whether $D \models^{u'} Q[q]$, for some child u' of v, we have to consider all query nodes which are reachable from q by following edges typed with sibling axes, i.e., all nodes in $\text{subtreeOnlySibling}(q)$. Indeed, $q \in \text{MatchChild}(v)$ should hold if these reachable query nodes can be matched to children of v in such a manner that the matching is (1) consistent with the sibling edges of the query and (2) every query node q' that is reachable from q by sibling edges is matched to such a child node u' such that $D \models^{u'} \text{subtreeNoSibling}(q')$, i.e., $q' \in \text{MatchNoSibling}(u')$.
The existence of such a matching can efficiently be decided (and maintained) as follows. First, consider the string $w = \overline{v_1} \cdots \overline{u} \cdots \overline{v_n}$, corresponding to the sequence $v_1 \cdots u \cdots v_n$ of children of v where the $\overline{v_i} = \text{MatchNoSibling}(v_i)$ (and $\overline{u} = \text{MatchNoSibling}(u)_{\text{new}}$), i.e., the label is formed by the set of query nodes whose subpattern without siblings can be matched here. Second, consider the query $Q_{\text{sib}} = \text{subtreeOnlySibling}(q)$. Then, we say that a query node q' of Q_{sib} *matches* a string symbol $\overline{v_i} = \text{MatchNoSibling}(v_i)$ if $q' \in \text{MatchNoSibling}(v_i)$. Now, $q \in \text{MatchChild}(v)$ if and only if there exists a matching of Q_{sib} on w. This matching does not need to be a root matching, but can be any matching. Furthermore, notice that at most one label, namely the one for u, in w changes when an update occurs. Therefore, we can use the algorithm presented in Section 5 to incrementally maintain tree pattern queries over strings, in this slightly altered semantics, to efficiently decide whether $q \in \text{MatchChild}(v)_{\text{new}}$.

—$\text{numDesc}(v)_{\text{new}}$: For all q, $\text{numDesc}^q_{\text{new}}(v)$

$$= \begin{cases} \text{numDesc}^q(v) + 1 & \text{if } q \in \text{MatchDesc}(u)_{\text{new}} \text{ and } q \notin \text{MatchDesc}(u)_{\text{old}} \\ \text{numDesc}^q(v) - 1 & \text{if } q \notin \text{MatchDesc}(u)_{\text{new}} \text{ and } q \in \text{MatchDesc}(u)_{\text{old}} \\ \text{numDesc}^q(v) & \text{otherwise} \end{cases}$$

—$\text{MatchDesc}(v)_{\text{new}} : \text{MatchDesc}(v)_{\text{new}} =$
$\{q \mid \exists q' \in \text{MatchChild}(v)_{\text{new}} : q \in \text{Nodes}(Q[q'])\} \cup \{q \mid \text{numDesc}^q_{\text{new}}(v) > 0\}$.

We argue that the algorithm works in time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(|Q|))$. We have to update at most $\text{depth}(D)$ nodes, so it suffices to argue that handling one node can be done in time $\mathcal{O}(\log(\text{width}(D)) \cdot \text{poly}(|Q|))$. All sets except $\text{MatchChild}(v)$ can easily be updated in time $\mathcal{O}(|Q|)$. Further, for updating $\text{MatchChild}(v)$, we have to apply the incremental maintenance algorithm for strings, which has complexity $\mathcal{O}(\log(w) \cdot |P|^6)$, where $w \leq \text{width}(D)$ (see Theorem 5.1) and P is the query to be maintained. This algorithm has to be run for all query nodes $q \in \text{downNodes}(Q)$ and corresponding queries $\text{subtreeOnlySibling}(Q)$. These subpatterns are all disjoint, and therefore the sum of their sizes is at most $|Q|$. The slightly altered semantics of the matching relation (that a query node matches a document node if the label of the query node is in the set defined by

the document node) can add a linear factor $|Q|$ to the algorithm of Section 5. This all together means that the update of MatchChild(v) can be done in time $\mathcal{O}(\log(\text{width}(D)) \cdot |Q|^7)$. Thus the total time complexity of the algorithm presented in this section is $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot |Q|^7)$.

Finally, we show that the data structure can be stored in space $\mathcal{O}(|D| \cdot |Q|^3)$. As in Section 4, the node records can be stored in space $\mathcal{O}(|D| \cdot |Q|)$. However, this is dominated by the space needed to store the auxiliary data structures for the incremental maintenance algorithm for strings. According to Theorem 5.1 these data structures can be stored in space $\mathcal{O}(|w| \cdot |P|^3)$, where w is the string and P the query to be evaluated. Then, as all subpatterns we use for incremental string maintenance are disjoint, and every document node has only one parent, it follows that all information can be stored in space $\mathcal{O}(|D| \cdot |Q|^3)$.

7. CONCLUSIONS AND OUTLOOK

We have shown that incremental evaluation of XPath queries can be performed significantly more efficiently than re-evaluation, for several practically interesting fragments of XPath.

Of course, our study is far from complete and this work should be seen as an initial theoretical step in this line of work. Indeed, with the exception of Section 4.2, we have exclusively investigated Boolean maintenance, so the array of possible further research is still wide open. We hope that we were able to show that, incremental evaluation for some seemingly very innocent fragments of XPath (essentially the tree pattern fragment) is already quite non-trivial, even if the XML data is structured as a string instead of a tree (Section 5). The overall question that needs attention for future research is, *Which XPath queries can we incrementally evaluate in time polylogarithmic in the data and polynomial in the query?* In pursuit of this overall question, we list the following interesting questions:

—Can the algorithm/approach from Section 5 be extended from strings to trees?

—Can we strengthen the view maintenance approach? Ideally, we would like to be able to maintain a set of designated *output nodes* in XPath Patterns, that produce a relation as output of the query.

Finally, we remark that some of the results presented in this paper may be of interest in automated verification as well. The XPath fragments we study are highly similar to some temporal logics, since the X (next) and F (future) operators in temporal logics correspond to the $\rightarrow$ (resp., $\downarrow$) and $\Rightarrow$ (resp., $\Downarrow$) axes in XPath on strings (resp., trees). Therefore, our results imply that, incremental maintenance of the truth of a temporal logic can sometimes be maintained efficiently over strings and trees as well.

ACKNOWLEDGMENTS

We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599. Wouter Gelade is a Research Assistant of the Fund for Scientific Research — Flanders (Belgium). Wim Martens is supported by a by a grant of the

North-Rhine Westfalian Academy of Sciences and Arts, and the Stiftung Mercator Essen.

Furthermore, we are grateful to the anonymous reviewers of ICDT 2009 and of ACM Transactions on Database Systems, whose valuable suggestions helped improve the presentation of this paper.

REFERENCES

Abiteboul, S., Bourhis, P., and Marinoiu, B. 2007. Incremental view maintenance for active documents. In *Journées Bases de Données Avancées (BDA)*.

Abiteboul, S., Bourhis, P., and Marinoiu, B. 2009. Efficient maintenance techniques for views over active documents. In *International Conference on Extending Database Technology (EDBT)*. ACM Press, New York, 1076–1087.

Balmin, A., Papakonstantinou, Y., and Vianu, V. 2004. Incremental validation of XML documents. *ACM Transactions on Database Systems 29,* 4, 710–751.

Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2007. On the memory requirements of XPath evaluation over XML streams. *Journal of Computer and System Sciences 73,* 3, 391–441.

Barbosa, D., Mendelzon, A., Libkin, L., Mignet, L., and Arenas, M. 2004. Efficient incremental validation of XML documents. In *International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Washington DC, 671–682.

Benedikt, M., Fan, W., and Geerts, F. 2007. XPath satisfiability in the presence of DTDs. *Journal of the ACM 55,* 2.

Benedikt, M. and Koch, C. 2008. XPath leashed. *ACM Computing Surveys 41,* 1.

Björklund, H., Gelade, W., Marquardt, M., and Martens, W. 2009. Incremental XPath evaluation. In *International Conference on Database Theory (ICDT)*. ACM Press, New York, 162–173.

Bojańczyk, M. and Parys, P. 2008. XPath evaluation in linear time. In *International Symposium on Principles of Database Systems (PODS)*. ACM Press, New York, 241–250.

Clark, J. and DeRose, S. 1999. XML Path Language (XPath) version 1.0. Tech. rep., World Wide Web Consortium. http://www.w3.org/TR/xpath/.

Dong, G., Libkin, L., Su, J., and Wong, L. 1999. Maintaining transitive closure of graphs in SQL. *International Journal of Information Technology 5,* 1, 46–78.

Dong, G., Libkin, L., and Wong, L. 2003. Incremental recomputation in local languages. *Information and Computation 181,* 2, 88–98.

Dong, G. and Su, J. 2000. Incremental maintenance of recursive views using relational calculus/SQL*. *Sigmod RECORD 29,* 1, 44–51.

Dong, G., Su, J., and Topor, R. 1995. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence 14,* 2–4, 187–223.

Gelade, W., Marquardt, M., and Schwentick, T. 2009. The dynamic complexity of formal languages. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, Germany, 481–492.

Glaister, I. and Shallit, J. 1996. A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters 59,* 2, 75–77.

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

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

Götz, M., Koch, C., and Martens, W. 2009. Efficient algorithms for descendant-only tree pattern queries. *Information Systems 34,* 7, 602–623.

Griffin, T. and Libkin, L. 1995. Incremental maintenance of views with duplicates. In *International Symposium on Management of Data (SIGMOD)*. ACM Press, New York, 328–339.

Grohe, M., Koch, C., and Schweikardt, N. 2007. Tight lower bounds for query processing on streaming and external memory data. *Theoretical Computer Science 380,* 1–2, 199–217.

Gupta, A., Mumick, I. S., and Subrahmanian, V. 1993. Maintaining views incrementally. In *International Symposium on Management of Data (SIGMOD)*. ACM Press, New York, 157–166.

Hoffmann, C. and O'Donnell, M. 1982. Pattern matching in trees. *Journal of the ACM 29,* 1, 68–95.

Koch, C. 2010. Incremental query evaluation in a ring of databases. In *International Symposium on Principles of Database Systems (PODS)*. ACM Press, New York.

Libkin, L. and Sirangelo, C. 2008. Reasoning about XML with temporal logics and automata. In *International Symposium on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer, Germany, 97–112.

Libkin, L. and Wong, L. 1997. Incremental recomputation of recursive queries with nested sets and aggregate functions. In *International Symposium on Database Programming Languages (DBPL)*. Springer, Germany, 222–238.

Liu, J., Vincent, M., and Mohania, M. 1999. Incremental maintenance of nested relational views. In *International Database Engineering and Applications Symposium (IDEAS)*. 197–205.

Martens, W. and Neven, F. 2005. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science 336,* 1, 153–180.

Matsumura, H. and Tajima, K. 2005. Incremental evaluation of a monotone XPath fragment. In *ACM Conference on Information and Knowledge Management (CIKM)*. ACM Press, New York, 245–246.

Miklau, G. and Suciu, D. 2004. Containment and equivalence for a fragment of XPath. *Journal of the ACM 51,* 1, 2–45.

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

Neven, F. and Schwentick, T. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science 2,* 3.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. 2004. ORDPATHs: Insert-friendly XML node labels. In *International Symposium on Management of Data (SIGMOD)*. ACM Press, New York, 903–908.

Onizuka, M., Chan, F. Y., Michigami, R., and Honishi, T. 2005. Incremental maintenance for materialized XPath/XSLT views. In *World Wide Web Conference (WWW)*. ACM Press, New York, 671–681.

Pang, C., Dong, G., and Ramamohanarao, K. 2005. Incremental maintenance of shortest distance and transitive closure in first-order logic and SQL. *ACM Transactions on Database Systems 30,* 3, 698–721.

Parys, P. 2009. XPath evaluation in linear time with polynomial combined complexity. In *International Symposium on Principles of Database Systems (PODS)*. ACM Press, New York, 55–64.

Patnaik, S. and Immerman, N. 1997. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences 55,* 2, 199–209.

Sawires, A., Tatemura, J., Po, O., Agrawal, D., Abbadi, A. E., and Candan, K. S. 2006. Maintaining XPath views in loosely coupled systems. In *International Conference on Very Large Data Bases (VLDB)*. ACM Press, New York, 583–594.

Sawires, A., Tatemura, J., Po, O., Agrawal, D., and Candan, K. S. 2005. Incremental maintenance of path-expression views. In *International Symposium on Management of Data (SIGMOD)*. ACM Press, New York, 443–454.

Schweikardt, N. 2007. Machine models and lower bounds for query processing. In *International Symposium on Principles of Database Systems (PODS)*. ACM Press, New York, 41–52.

Schwentick, T. 2004. XPath query containment. *Sigmod RECORD 33,* 1, 101–109.

Shmueli, O. and Itai, A. 1984. Incremental view maintenance. In *International Symposium on Management of Data (SIGMOD)*. ACM Press, New York, 240–255.

Simon, I. 1990. Factorization forests of finite height. *Theoretical Computer Science 72,* 1, 65–94.

TEN CATE, B. AND LUTZ, C. 2009. The complexity of query containment in expressive fragments of XPath 2.0. *Journal of the ACM 56,* 6.

VARDI, M. Y. 1998. Reasoning about the past with two-way automata. In *International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, Germany, 628–641.

This document is the online-only appendix to:

Incremental XPath Evaluation

HENRIK BJÖRKLUND
Umeå University, Department of Computing Science
and
WOUTER GELADE
Hasselt University and Transnational University of Limburg, School for Information Technology, Belgium
and
WIM MARTENS
Technical University of Dortmund, Department of Computer Science

ACM Transactions on Database Systems, Vol. V, No. N, M 20YY, Pages 1–0??.

A. TRANSLATING XPATH TO TREE AUTOMATA

The goal of this Appendix is to translate an XPath Pattern into an equivalent unranked tree automaton as in the statement of Theorem 3.1.

Although XPath operates directly on unranked trees, we will intermediately work with binary trees encoding these unranked trees. Following [Neven 2002], for an (unranked) tree T, let $\text{enc}(T)$ be its binary encoding, obtained as follows: The nodes of $\text{enc}(T)$ are the nodes of T plus a set of leaf nodes marked #. Further, the root node of $\text{enc}(T)$ is the root node of T and for any node, its left child in $\text{enc}(T)$ is its first child in T (or # if its a leaf), and its right child in $\text{enc}(T)$ is its next sibling in T (or # if it has none). Figure 12 shows an example.

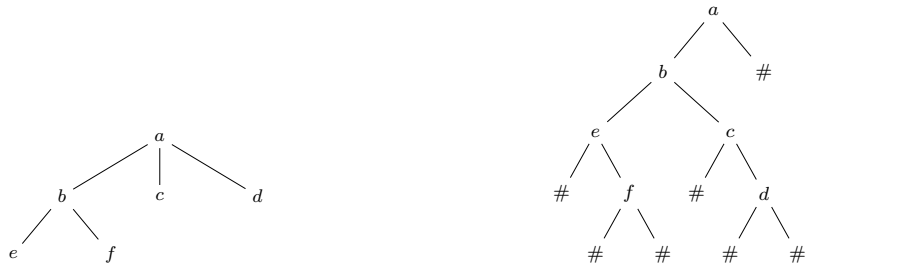
We first show how to translate an XPath Patterns into a *loop-free 2-way alternating tree automaton* in linear time (Lemma A.3) and then show how to transform it into a non-deterministic tree automaton (Lemma A.5).

We start by defining these 2-way alternating tree automata, which operate on binary trees:

Definition. *Let $B^+(P)$ be the set of positive Boolean formulas over propositions P (i.e., formulas without negation), but including* true *and* false*. A* two way alternating tree automaton *(2ATA) over binary trees is defined as a tuple $A = (States(A), Alph(A), Rules(A), init(A))$, where*

—*$States(A)$ is a finite set of states,*

—*$Alph(A)$ is a finite set of alphabet symbols,*



(a) Unranked tree T. (b) Its binary encoding enc(T).

Fig. 12. An unranked tree and its binary encoding.

—*Rules*(A) *is a set of transition rules on the form* $(q, a) \to \theta$*, where* $q \in$ *States*$(A), a \in$ *Alph*(A)*, and* θ *is a formula from* $B^+(\{\nwarrow, \nearrow, -, \swarrow, \searrow\} \times$ *States*$(A))$*, and*

—*init*(A) *is the inital state.*

Rules(A) should be such that for each pair $(q, a) \in$ *States*$(A) \times$ *Alph*(A) *there is at most one rule in Rules*(A) *with* (q, a) *as its left hand side. If* $(q, a) \to \theta \in$ *Rules*(A)*, we also write* $rhs_A(q, a) = \theta$*. Elements in* $\{\nwarrow, \nearrow, -, \swarrow, \searrow\}$ *denote directions in the tree. For a node* u *of* T*,* $u\cdot\nearrow$ *(respectively,* $u\cdot\nwarrow$*) denotes the parent* v *of* u *if* u *is the left child (respectively, right child) of* v *and is undefined otherwise. Further,* $u\cdot-$ *is* u *itself (stay transition), and* $u\cdot\swarrow$ *(respectively,* $u\cdot\searrow$*) denotes the left child (respectively, right child) of* u *if it exists and is undefined otherwise.*

Given a binary tree T*, a* run tree *of* A *on* T *is an unranked tree* R *in which each node is labeled by an element of Nodes*$(T) \times$ *States*(A) *such that the following holds.*

—*The label of the root of* R *is* $(root(T), init(A))$ *and*

—*for every node* x *of* R *labeled* (v, q_v)*, with* $(q_v, lab^T(v)) \to \theta \in$ *Rules*(A)*, there is a set* $S \subseteq \{\nwarrow, \nearrow, -, \swarrow, \searrow\} \times$ *States*(A) *such that,*

  —*for every* $(i, q') \in S$*,* $v \cdot i$ *is defined and there exists a child* y *of* x *in* R *labeled* $(v \cdot i, q')$*, and*

  —*the truth assignment that assigns true to all elements of* S*, and false to all other elements of* $\{\nwarrow, \nearrow, -, \swarrow, \searrow\} \times$ *States*(A)*, satisfies* θ*.*

A run tree R *is* accepting *if, for every leaf node of* R *labeled* (u, q)*,* $rhs_A(lab(u), q)$ *= true. A binary tree* T *is* accepted *by a 2ATA* A *if there exists a finite accepting run tree of* A *on* T*, i.e., an accepting run tree with no infinite paths. By* $L(A)$ *we denote the set of trees accepted by* A*.*

We say that a 2ATA A *is* loop-free *if, for all possible trees* T*, every run tree of* A *on* T *is finite, i.e., contains no infinite paths. Intuitively, this means that* A *never rejects a tree by entering an infinite loop.*

We now show that, given an XPath Pattern, we can efficiently construct an equivalent two-way alternating tree automaton. We note, however, that it is well known that there is a connection between XPath expressions and two-way alternating automata. Benedikt, Fan and Geerts [Benedikt et al. 2007] have shown that it is possible to construct, in linear time, a two-way alternating word automaton, accepting string encodings of trees defined by an XPath query. This construction,

however, only works when the considered trees have a fixed depth. Further, ten Cate and Lutz [ten Cate and Lutz 2009] have shown that it is possible to construct, in quadratic time, a two-way alternating tree automaton equivalent to a given XPath query. For our purposes, however, we need to construct a two-way alternating *tree* automaton of *linear* size, equivalent to a given query. Therefore, and for reasons of completeness, we give the full construction below.

REMARK A.2 (FINITE VERSUS INFINITE ALPHABETS). *As stated in Section 2, we consider the alphabet Σ, over which XML document labels range, to be infinite, while a 2-way alternating tree automaton A has a finite alphabet $Alph(A)$. A given query Q, however, is finite, and only uses labels from a finite subset Γ of Σ plus the wildcard symbol $*$ that matches any label. When constructing an automaton for Q, we consider the automaton alphabet to be $\Gamma \cup \{\$\}$, where $\$ \notin \Gamma$. When the automaton is run on a tree T, we treat every symbol $a \notin \Gamma$ occurring in T as the symbol $\$$.*

Formally, this means that we extend the definition of $L(A)$ as follows. Let T be a binary tree using labels from Σ and let $\$$ be a symbol not occurring in Σ. Let $T_\$$ be the tree obtained from T by replacing every label $a \notin Alph(A)$ by $\$$. In the following, we assume $Alph(A)$ to contain the symbols $\$$ and $\#$ and define $L(A)$ to be $\{T \mid$ there is a finite accepting run tree of A on $T_\$\}$.

LEMMA A.3. *Let Q be an XPath Pattern. A loop-free two-way alternating tree automaton A with $L(A) = \{enc(T) \mid T \models Q\}$ can be constructed in time $\mathcal{O}(|Q|)$.*

PROOF. Let Q be an XPath Pattern. We construct $A = (\mathrm{States}(A), \mathrm{Alph}(A), \mathrm{Rules}(A), \mathrm{init}(A))$, such that $L(A) = \{\mathrm{enc}(T) \mid T \models Q\}$, as follows. The set $\mathrm{States}(A)$, contains, for each node u of Q, two states q_u and $\overline{q_u}$. Similarly, $\mathrm{States}(A)$ contains, for each edge e of Q, the states q_e and $\overline{q_e}$, and additionally, for the edges of type $\downarrow, \Downarrow, \downarrow^*, \Uparrow, \uparrow^*$, the states q'_e and $\overline{q_e}'$. A state q_u or q_e can be seen as a pointer to a node u or edge e in the pattern Q for which the automaton guesses that the subtree below u or e, respectively, can be matched at its current position in the tree T. The states $\overline{q_u}$ and $\overline{q_e}$ denote that the automaton guesses that the subtrees rooted at u and e, respectively, can*not* be matched here. The starting state $\mathrm{init}(A)$ is the state corresponding to the root of Q, i.e., $\mathrm{init}(A) := q_{root(Q)}$.

Finally, the transition relation $\mathrm{Rules}(A)$ is given in Table II. (Recall that $\mathrm{rhs}_A(q, a) = \theta$ if and only if $(q, a) \to \theta$ is the unique rule in Rules(A) with (q, a) as its left hand side.) For readability, we use the four predicates firstSibling, lastSibling, isRoot and isLeaf to indicate that the node the automaton is currently visiting is a first or a last sibling, the root, or a leaf in T, respectively. These can be easily tested by the automaton using a constant number of transitions, because a node is a first sibling in T if and only if it is a left child in $\mathrm{enc}(T)$ and not labeled $\#$, a last sibling in T if and only if its right child in $\mathrm{enc}(T)$ is labeled with $\#$, and a leaf in T if and only if its left child in $\mathrm{enc}(T)$ is labeled with $\#$. We can also assume w.l.o.g. that we can test whether a node is the root, e.g., by assuming that there is a special end-marker above the original root of the input tree.[8] Then a node is the

[8]That is, one could re-define $\mathrm{enc}(T)$ to incorporate this marker. We omitted this in the definition of $\mathrm{enc}(T)$ to keep the overall argument simple.

For every $a \in \mathrm{Alph}(A)$, we have the following transitions.	
If $\mathrm{type}(u) = \mathrm{label}$	
and $\mathit{Edges}(u) \neq \emptyset$:	$\mathrm{rhs}_A(q_u, a) = \text{false if } \mathrm{lab}^Q(u) \notin \{a, *\}, \text{ otherwise } \bigwedge_{e \in \mathit{Edges}(u)}(-, q_e)$
	$\mathrm{rhs}_A(\overline{q_u}, a) = \text{true if } \mathrm{lab}^Q(u) \notin \{a, *\}, \text{ otherwise } \bigvee_{e \in \mathit{Edges}(u)}(-, \overline{q_e})$
and $\mathit{Edges}(u) = \emptyset$:	$\mathrm{rhs}_A(q_u, a) = \text{false if } \mathrm{lab}^Q(u) \notin \{a, *\}, \text{ otherwise true}$
	$\mathrm{rhs}_A(\overline{q_u}, a) = \text{true if } \mathrm{lab}^Q(u) \notin \{a, *\}, \text{ otherwise false}$
If $\mathrm{type}(u) = \mathrm{syntax}$	
and $\mathrm{lab}^Q(u) = \wedge/\vee$:	$\mathrm{rhs}_A(q_u, a) = \bigwedge_{e \in \mathit{Edges}(u)}(-, q_e) \;/\; \bigvee_{e \in \mathit{Edges}(u)}(-, q_e)$
	$\mathrm{rhs}_A(\overline{q_u}, a) = \bigvee_{e \in \mathit{Edges}(u)}(-, \overline{q_e}) \;/\; \bigwedge_{e \in \mathit{Edges}(u)}(-, \overline{q_e})$
and $\mathrm{lab}^Q(u) = \neg$:	$\mathrm{rhs}_A(q_u, a) = (-, \overline{q_e})$, $e \in \mathit{Edges}(u)$ unique
	$\mathrm{rhs}_A(\overline{q_u}, a) = (-, q_e)$, $e \in \mathit{Edges}(u)$ unique
With $e = (u, v)$,	
if $\mathrm{type}(e) = \mathrm{syntax}$:	$\mathrm{rhs}_A(q_e, a) = (-, q_v)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (-, \overline{q_v})$
if $\mathrm{type}(e) = \mathrm{self}$:	$\mathrm{rhs}_A(q_e, a) = (-, q_v)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (-, \overline{q_v})$
if $\mathrm{type}(e) = \downarrow$:	$\mathrm{rhs}_A(q_e, a) = (\swarrow, q'_e)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (\swarrow, \overline{q'_e})$
	$\mathrm{rhs}_A(q'_e, a) = (-, q_v) \vee (\searrow, q'_e)$
	$\mathrm{rhs}_A(\overline{q'_e}, a) = (-, \overline{q_v}) \wedge [(\searrow, \overline{q'_e}) \vee \mathrm{lastSibling})]$
if $\mathrm{type}(e) = \Downarrow$:	$\mathrm{rhs}_A(q_e, a) = (\swarrow, q'_e)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (\swarrow, \overline{q'_e})$
	$\mathrm{rhs}_A(q'_e, a) = (-, q_v) \vee (\swarrow, q'_e) \vee (\searrow, q'_e)$
	$\mathrm{rhs}_A(\overline{q'_e}, a) = (-, \overline{q_v}) \wedge [(\swarrow, \overline{q'_e}) \vee \mathrm{isLeaf}] \wedge [(\searrow, \overline{q'_e}) \vee \mathrm{lastSibling}]$
if $\mathrm{type}(e) = \downarrow^*$:	$\mathrm{rhs}_A(q_e, a) = (-, q_v) \vee (\swarrow, q'_e)$
	$\mathrm{rhs}_A(q'_e, a) = (-, q_v) \vee (\swarrow, q'_e) \vee (\searrow, q'_e)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = (-, \overline{q_v}) \wedge (\swarrow, \overline{q'_e})$
	$\mathrm{rhs}_A(\overline{q'_e}, a) = (-, \overline{q_v}) \wedge [(\swarrow, \overline{q'_e}) \vee \mathrm{isLeaf}] \wedge [(\searrow, \overline{q'_e}) \vee \mathrm{lastSibling}]$
if $\mathrm{type}(e) = \uparrow$:	$\mathrm{rhs}_A(q_e, a) = (\nwarrow, q_e) \vee (\nearrow, q_v)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = (\nwarrow, \overline{q_e}) \vee (\nearrow, \overline{q_v})$
if $\mathrm{type}(e) = \Uparrow$:	$\mathrm{rhs}_A(q_e, a) = (\nwarrow, q_e) \vee (\nearrow, q'_e)$
	$\mathrm{rhs}_A(q'_e, a) = (-, q_v) \vee (\nwarrow, q_e) \vee (\nearrow, q'_e)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = (\nwarrow, \overline{q_e}) \vee (\nearrow, \overline{q'_e})$
	$\mathrm{rhs}_A(\overline{q'_e}, a) = (-, \overline{q_v}) \wedge [(\nwarrow, \overline{q_e}) \vee \mathrm{firstSibling}] \wedge [(\nearrow, \overline{q'_e}) \vee \mathrm{isRoot}]$
if $\mathrm{type}(e) = \uparrow^*$:	$\mathrm{rhs}_A(q_e, a) = (-, q_v) \vee (\nwarrow, q'_e) \vee (\nearrow, q_e)$
	$\mathrm{rhs}_A(q'_e, a) = (\nwarrow, q'_e) \vee (\nearrow, q_e)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = (-, \overline{q_v}) \wedge [(\nwarrow, \overline{q'_e}) \vee \mathrm{firstSibling}] \wedge [(\nearrow, \overline{q_e}) \vee \mathrm{isRoot}]$
	$\mathrm{rhs}_A(\overline{q'_e}, a) = [(\nwarrow, \overline{q'_e}) \vee \mathrm{firstSibling}] \wedge [(\nwarrow, \overline{q_e}) \vee \mathrm{isRoot}]$
if $\mathrm{type}(e) = \rightarrow$:	$\mathrm{rhs}_A(q_e, a) = (\searrow, q_v)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (\searrow, \overline{q_v}) \vee \mathrm{lastSibling}$
if $\mathrm{type}(e) = \Rightarrow$:	$\mathrm{rhs}_A(q_e, a) = (\searrow, q_v) \vee (\searrow, q_e)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = [(\searrow, \overline{q_v}) \wedge (\searrow, \overline{q_e})] \vee \mathrm{lastSibling}$
if $\mathrm{type}(e) = \leftarrow$:	$\mathrm{rhs}_A(q_e, a) = (\nwarrow, q_v)$ and $\mathrm{rhs}_A(\overline{q_e}, a) = (\nwarrow, \overline{q_v}) \vee \mathrm{firstSibling}$
if $\mathrm{type}(e) = \Leftarrow$:	$\mathrm{rhs}_A(q_e, a) = (\nwarrow, q_v) \vee (\nwarrow, q_e)$
	$\mathrm{rhs}_A(\overline{q_e}, a) = [(\nwarrow, \overline{q_v}) \wedge (\nwarrow, \overline{q_e})] \vee \mathrm{firstSibling}$

Table II. Transitions of the 2ATA from the Proof of Lemma A.3.

root of the original tree if and only if its parent bears the special end-marker. For $u \in \mathrm{Nodes}(Q)$ let $\mathit{Edges}(u)$ denote the set of outgoing edges of u, i.e., edges of the form (u, v) for some node v of Q. Then, for each node u and each edge $e = (u, v)$ the transitions given in Table II apply.

As an example let us look at the transitions from a state corresponding to the descendant-edge ($\Downarrow$). At first there is one step to the first child of the current node. From then on, the additional state q' is used to search for a proper sibling or descendant node.

It now follows by induction that, for a tree T, $u \in \mathrm{Nodes}(T)$ and $v \in \mathrm{Nodes}(Q)$

(respectively, $e \in \text{Edges}(Q)$), there is an accepting run tree of A on $\text{enc}(T)$ from (u, q_v) (respectively, (u, q_e)) if and only if $T \models^u Q[v]$ (respectively, $T \models^u Q[e]$). It hence follows that $\text{enc}(T) \in L(A)$ if and only if $T \models Q$.

Finally, it is easy to see that the size of the resulting automaton is linear in the size of Q and that it can be computed in linear time. Furthermore, A is loop-free, because when following a transition, it either stays in the same state but consistently moves only up- or only downwards in the tree; or, it changes its state to a new state that corresponds to a node or edge one level deeper in the XPath Pattern. This guarantees that each run tree of A is finite. □

We now show that, given a loop-free two-way alternating tree automaton, it is possible to construct an equivalent non-deterministic unranked tree automaton (NTA) in exponential time. First, we formally introduce unranked tree automata.

DEFINITION A.4. A *non-deterministic unranked tree automaton (NTA)* is a tuple $A = (\text{States}(A), \text{Alph}(A), \text{Rules}(A), \text{Final}(A))$, where $\text{States}(A)$ is a finite set of states, $\text{Final}(A) \subseteq \text{States}(A)$ is the set of final states, and $\text{Rules}(A)$ is a set of rules of the form $q \xrightarrow{a} L$, where L is a regular string language over $\text{States}(A)$ for every $a \in \text{Alph}(A)$ and $q \in \text{States}(A)$. The regular langage L is represented by a non-deterministic finite string automaton.

A *run* of A on a tree T is a labeling $\lambda : \text{Nodes}(T) \to \text{States}(A)$ that respects the rules of the automaton. In other words, for every $v \in \text{Nodes}(T)$ with children $v_1, \ldots, v_n$, there is a rule $\lambda(v) \xrightarrow{\text{lab}^T(v)} L$ with $\lambda(v_1) \cdots \lambda(v_n) \in L$. Note that when v has no children, then this criterion reduces to $\varepsilon \in L$. A run is *accepting* if and only if the root is labeled with an accepting state, that is, $\lambda(\varepsilon) \in \text{Final}(A)$. A tree is accepted if there is an accepting run. The set of all accepted trees is denoted by $L(A)$. We can extend the definition of $L(A)$ to trees using the infinite alphabet Σ similarly as we did for 2ATAs.

The proof of the following lemma goes along the lines of [Martens and Neven 2005], where the equivalence of two-way alternating finite automata and nondeteministic finite automata on (finite) strings was proved, using techniques from Vardi [Vardi 1998].

LEMMA A.5. *Given a loop-free 2ATA A one can construct a non-deterministic tree automaton B such that $L(A) = L(B)$ in time $2^{O(|A|)}$.*

PROOF. Let $A = (\text{States}(A), \text{Alph}(A), \text{Rules}(A), \text{init}(A))$ be a 2ATA. The NTA $B = (\text{States}(B), \text{Alph}(A), \text{Rules}(B), \text{init}(B), \text{Final}(B))$ will simulate A in a top-down manner. The states of B are tuples of subsets of $\text{States}(A)$.

Let T be a tree in $L(A)$ with accepting run tree R. Then B will read T in a top-down manner, and for each node u in T, it will guess the states q of A such that the label (u, q) appears somewhere in R. That is, B guesses, for each node u in T, all states in which A visited u. Notice that, as R is an accepting run, each such (u, q) "leads to acceptance" of A on T. Additionally, it also remembers these states for the parent of u (for verifying the correctness of the transitions of A) and the information whether u is a left or right child of its parent.

More formally $\text{States}(B) := 2^{\text{States}(A)} \times 2^{\text{States}(A)} \times \{\swarrow, \searrow\}$. A tuple $(U, V, i) \in \text{States}(B)$ contains the set V of states leading to acceptance from the current node

and the set U of states leading to acceptance from the parent of the current node.

Further, the root node of any document does not have a parent, but $\text{init}(A)$ should lead to acceptance here. Hence, $\text{init}(B) = \{(\emptyset, V, \searrow) \mid \text{init}(A) \in V\}$. Notice that we consider all sets V such that $\text{init}(A) \in V$ as valid start states. We need more than just $\{\text{init}(A)\}$ to also take the other states into account in which A visited the root. Here, the $\searrow$ in the sets in $\text{init}(B)$ is unimportant and chosen arbitrarily.

Now, the transition relation of B should be locally consistent with A. That is, let $i \in \{\swarrow, \searrow\}$ and define $\bar{i}$ to be the converse direction, i.e., $\bar{i} = \nearrow$ if $i = \swarrow$ and $\bar{i} = \nwarrow$ if $i = \searrow$. For $a \in \text{Alph}(A)$, and $U, V, W_1, W_2 \in 2^{\text{States}(A)}$, we have that

$$(U, V, i) \xrightarrow{a} (V, W_1, \swarrow)(V, W_2, \searrow) \in \text{Rules}(B)$$

if and only if the following holds. Let $S := (\{\bar{i}\} \times U) \cup (\{-\} \times V) \cup (\{\swarrow\} \times W_1) \cup (\{\searrow\} \times W_2)$. Then the truth assignment that assigns true to all elements of S, and false to all other elements of $\{\nwarrow, \nearrow, -, \swarrow, \searrow\} \times \text{States}(A)$, satisfies $\theta := \text{rhs}_A(q, a)$ for every $q \in V$. The transition relation therefore checks that the set V is "correct", provided that all other sets are correct.

Finally, $((U, V, i), a) \in F$ if and only if, for S defined as $(\{\bar{i}\} \times U) \cup (\{-\} \times V)$, S satisfies $\theta := \text{rhs}_A(q, a)$ for all $q \in V$, in the same manner as above. The condition on the final states hence guarantees that the guessed states at the leaf nodes are correct and consistent with previous guesses.

It can now be shown that, for any tree T, there is an accepting run tree of A on T if and only if there is an accepting run of B on T. Indeed, given a run tree of A on T we can construct a consistent accepting run of B on T, and vice versa. It should be observed, however, that we need the loop-freeness of A to ensure that an accepting run of B leads to an accepting run of A. □

The previous results can now be combined to prove Theorem 3.1. We only need to transform the automaton constructed above into a non-deterministic *unranked* tree automaton (see Definition A.4).

Proof of Theorem 3.1: *Let Q be an XPath Pattern. A non-deterministic unranked tree automaton A, with $L(A) = \{D \mid D \models Q\}$, can be constructed in time $2^{\mathcal{O}(|Q|)}$.*

Proof. Let Q be an XPath Pattern. By combining Lemma A.3 and Lemma A.5, we can construct an NTA B, with $L(B) = \{\text{enc}(D) \mid D \models Q\}$ in time $2^{O(|Q|)}$. Denote the backward translation from (encoded) binary trees to unranked trees, by dec, i.e. $\text{dec}(\text{enc}(D)) = D$. Then, again in [Neven 2002], it is noted that given a non-deterministic tree automaton accepting encoded binary trees, one can construct, in polynomial time, an equivalent unranked tree automaton accepting their decoded unranked versions. (Actually, this non-deterministic unranked tree automaton for the decoding can be constructed in linear time.) In other words, we can construct, in time linear in the size of NTA B, (or, in time $2^{O(|Q|)}$), the unranked tree automaton A accepting $\{\text{dec}(D) \mid D \in L(B)\} = \{D \mid D \models Q\}$. This is the desired automaton. □

Remark A.6 (Correspondence with XPath Containment). *Notice that this Appendix, together with Proposition 1 from [Miklau and Suciu 2004] provides an*

alternative proof for Theorem 2 in [Libkin and Sirangelo 2008]. Libkin and Sirangelo show that, given an unranked tree automaton A and a Boolean combination $\mathcal{C}$ of inclusions $Q_1 \subseteq Q_2$ between XPath queries Q_1 and Q_2, one can construct an unranked tree automaton of size $|A| \cdot 2^{O(|\mathcal{C}|)}$ that accepts the empty language if and only if $\mathcal{C}$ is true w.r.t. $L(A)$. (Here, an inclusion $Q_1 \subseteq Q_2$ is true w.r.t. $L(A)$ when, for each tree $T \in L(A)$, the result of evaluating Q_1 on T is a subset of the result of evaluating Q_2 on T.)

Indeed, Proposition 1 from [Miklau and Suciu 2004] and observations from [Neven and Schwentick 2006] show that the containment problem for two XPath queries Q_1 and Q_2 can be reduced to the language inclusion problem $\{T \mid T \models Q_1\} \subseteq \{T \mid T \models Q_2\}$ and that the containment problem w.r.t. a tree automaton A can be reduced to $(\{T \mid T \models Q_1\} \cap L(A)) \subseteq \{T \mid T \models Q_2\}$.

Given queries Q_1 and Q_2 we can therefore construct a loop-free 2ATA in time $O(|Q_1| + |Q_2|)$ for the set $\{enc(T) \mid T \models Q_1 \wedge T \not\models Q_2\}$, which accepts the empty language if and only if $\{T \mid T \models Q_1\} \subseteq \{T \mid T \models Q_2\}$. As a consequence, given a Boolean combination $\mathcal{C}$ of inclusions between XPath queries, we can also construct a loop-free 2ATA $A_{\mathcal{C}}$ in time $O(|\mathcal{C}|)$ which accepts the empty language if and only if $\mathcal{C}$ is true w.r.t. the set of all trees. Converting this 2ATA to an NTA gives an equivalent NTA $N_{\mathcal{C}}$ of size $2^{O(|\mathcal{C}|)}$. Finally, given a tree automaton A, the NTA for $L(N_{\mathcal{C}}) \cap L(A)$ has size $|A| \cdot 2^{O(|\mathcal{C}|)}$ and accepts the empty language if and only if $\mathcal{C}$ is true w.r.t. A.

This remark immediately implies the following corollary.

Corollary A.7 (See also Theorem 2 from [Libkin and Sirangelo 2008]). *Given an unranked tree automaton A and a Boolean combination $\mathcal{C}$ of XPath inclusions, we can construct an unranked tree automaton of size $|A| \cdot 2^{O(|\mathcal{C}|)}$ whose language is empty if and only if $\mathcal{C}$ is true w.r.t. A.*

Since emptiness of an NTA can be tested in linear time, this also gives an algorithm to test whether $\mathcal{C}$ is true w.r.t. A in time $|A| \cdot 2^{O(|\mathcal{C}|)}$.