# Learning Twig and Path Queries

Slawomir Staworko, Piotr Wieczorek

# Learning Twig and Path Queries

Sławek Staworko
Mostrare, INRIA & LIFL (CNRS UMR8022)
University of Lille, France
slawomir.staworko@inria.fr

Piotr Wieczorek
Institute of Computer Science
University of Wrocław
piotr.wieczorek@cs.uni.wroc.pl

## ABSTRACT

We investigate the problem of learning XML queries, *path* queries and *twig* queries, from examples given by the user. A learning algorithm takes on the input a set of XML documents with nodes annotated by the user and returns a query that selects the nodes in a manner consistent with the annotation. We study two learning settings that differ with the types of annotations. In the first setting the user may only indicate *required nodes* that the query must select (i.e., *positive examples*). In the second, more general, setting, the user may also indicate *forbidden nodes* that the query must not select (i.e., *negative examples*). The query may or may not select any node with no annotation.

We formalize what it means for a class of queries to be *learnable*. One requirement is the existence of a learning algorithm that is *sound* i.e., always returning a query consistent with the examples given by the user. Furthermore, the learning algorithm should be *complete* i.e., able to produce every query with sufficiently rich examples. Other requirements involve tractability of the learning algorithm and its robustness to nonessential examples. We identify practical classes of Boolean and unary, path and twig queries that are learnable from positive examples. We also show that adding negative examples to the picture renders learning unfeasible.

## Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval—*Query formulation*; I.2.6 [**Artificial Intelligence**]: Learning

## General Terms

Algorithms, Languages, Theory.

## Keywords

XML, XPath, twigs, learning, query containment, query inference, minimality, consistency.


*ICDT 2012*, March 26–30, 2012, Berlin, Germany.


## 1. INTRODUCTION

XML has become a de facto standard for representation and exchange of data in web applications. An XML document is basically a labeled tree whose leaves store textual data and the standard XML format is text based to allow users an easy and direct access to the contents of the document [42]. However, to satisfy even modest information needs, the user is often required to formulate her queries using one of existing query languages whose common core is XPath [43, 44]. XPath queries allow to access the contents of the desired nodes with a syntax similar to directory paths used to navigate in the UNIX file system. Unfortunately, even the XPath query language, and any language with formal syntax, might be too difficult to be accessible to every user, and in general, there is a lack of frameworks allowing the user to formulate the query without the knowledge of a specialized query language.

In this paper, we propose to address this gap with the help of algorithms that infer the query from examples given by the user. We remark, however, that the need for general inference of XML queries is justified by other novel database applications. For instance, in the setting of XML data exchange [6] the pattern queries used to define data mappings need to be specified by the user. A learning algorithm could be a base for real *ad-hoc* data exchange solutions, where the pattern queries defining mappings are inferred as new sources are discovered. Another example of potential application is wrapper induction [20, 39].

The problem of XML query learning is defined as follows: given an XML document with nodes annotated by the user construct a query that selects the nodes accordingly to the annotations. Clearly, this problem has two parameters: the class of queries within which the algorithm should produce its result and the type of annotations the user may use. In the current work we focus on two well-known subclasses of XPath: *twig* and *path queries* [1]. We identify two types of annotations: *required* nodes i.e., nodes that need to be selected by the query, and *forbidden* nodes i.e., those that the query must not select. Because we do not require all nodes to be annotated, every unannotated node is implicitly annotated as *neutral*, which means that the query may or may not select it. In terms of computational learning theory [22], a required node is called a *positive example* and a forbidden node is a *negative example*. In this paper, we consider two settings: one, where the user provides only positive examples, and a more general one, where both positive and negative examples are present.

**Example 1** Take for instance the XML document in Figure 1 with a library listing. Some of its elements are annotated as required (+) and some as forbidden (−).

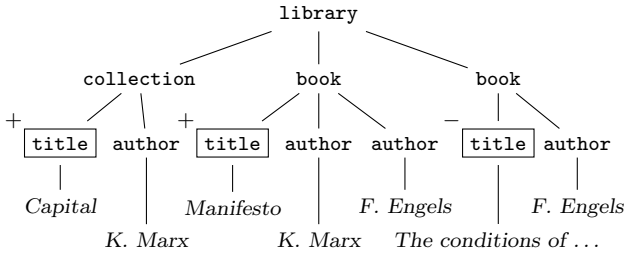



**Figure 1: Annotation of a library database**

The query that the user might want to receive is one that selects the titles of works by K. Marx:

$$q_0 = /\texttt{library}/\star[\texttt{author}=\text{"K. Marx"}]/\texttt{title}.$$

The query /library/⋆[author="K. Marx"]/⋆ is also consistent with the annotation but it properly contains $q_0$. This makes $q_0$ more specific w.r.t. the user annotations, and therefore, may be better fitted for the results of learning. The query selecting titles of all works, /library/⋆/title is not consistent because it selects the forbidden title node. The query /library/book[author="K. Marx"]/title is also not consistent with the annotation because it does not select the required title node of *Capital*. □

Our study requires us to define precisely what it means for a class of queries $\mathcal{Q}$ to be *learnable*. We propose a definition influenced by computational learning theory [22], and inference of languages in particular [21, 32, 13]. First of all, for $\mathcal{Q}$ to be learnable there must exist a learning algorithm learner which on the input takes a sample $S$ i.e., a set of examples, and returns a query $q \in \mathcal{Q}$. Naturally, learner should be *sound*, that is the query $q$ must be consistent with the sample $S$. Because the soundness condition is not enough to filter out trivial learning algorithms (cf. discussion following Definition 2), we furthermore require learner to be *complete*, that is able to learn every query with sufficiently informative examples. More precisely, learner is *complete* if for every $q \in \mathcal{Q}$ there exists a so called *characteristic sample* $CS_q$ of $q$ (w.r.t. learner) such that learner($CS_q$) returns $q$. Note that an unsavy user in the role of a teacher may not know exactly what is the characteristic sample, but rather attempt to approach it by adding more and more examples until the algorithm returns a satisfactory query. Consequently, it is commonly required for the characteristic sample to be *robust* under inclusion i.e., learner($S$) should return $q$ for any sample $S$ that extends $CS_q$ while being consistent with $q$. Finally, polynomial restrictions are imposed on learner and the size of the characteristic sample to ensure tractability of the framework.

The primary goal of this paper is learning unary queries, but on the way there we also investigate the learnability of Boolean queries. Unary queries select a set of nodes in a document and are typically used for information extraction tasks. On the other hand, Boolean queries test whether or not a given document satisfies certain property, and their typical use case is the classification of documents e.g., for filtering purposes. When learning a Boolean query, an example is a tree with a marker indicating whether it is a positive or a negative example.

**Example 2** Consider a simple XML feed with offers from a consumer-to-consumer web site (Figure 2) annotated by the user as either required (+) or forbidden (−).

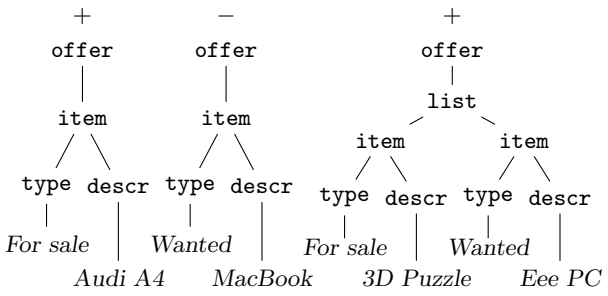



**Figure 2: An annotated XML stream.**

A Boolean query satisfying the user annotations selects all sale offers i.e., $q_1$ = .[offer//item/type="For sale"]. □

We investigate the learnability for Boolean and unary, path and twig queries in the presence of positive examples only and in the presence of both positive and negative examples. For learning in the presence of positive examples only, we identify practical subclasses of *anchored* path queries and *path-subsumption-free* twig queries that are learnable. The main idea behind our learning algorithms is to attempt to construct an (inclusion-)minimal query consistent with the examples. Intuitively this means that our algorithms try to construct a query that is as specific as possible with respect to the user input (cf. $q_0$ in Example 1). This approach is common to a host of algorithms learning concepts from positive examples [3] including reversible regular languages [4], $k$-testable regular languages [18], and single occurrence regular expressions [8]. While our learning algorithms for path queries return minimal queries consistent with the input sample, we show that this approach cannot be fully adopted for twig queries because there are input samples for which the consistent minimal twig query is of exponential size. Here, our learning algorithms return queries that can be seen as polynomially-sized approximations.

The learnability of the full classes of path and twig queries remains an open question. However, we identify the essential properties of the query classes that enable our learning techniques, and observe that these properties do not hold for the full classes of path and twig queries. This indicates that new approaches may need to be explored if learning of the full classes is feasible at all.

In the setting where both positive and negative examples are allowed, we study the *consistency problem*: given a document with a set of positive and negative annotations is there a query that satisfies the annotations? This problem is trivial if only positive examples are given because the

universal query, that selects all nodes in a tree, is consistent with any set of positive examples. However, as we show, adding even one negative example renders the consistency problem intractable. This result holds for all considered classes of queries, including anchored path queries and path-subsumption-free twig queries, and in fact, it holds for so simple classes of queries that it is hard to envision some reasonable restrictions that would admit learnability in the presence of positive and negative examples.

The main contribution of this paper is defining and establishing theoretical boundaries for learning path and twig queries from examples. To the best of our knowledge this is the first work addressing this particular problem. Additionally, we investigate two problems that might be of independent interest: constructing a minimal query consistent with a set of positive examples and checking the consistency of a set of positive and negative examples. The characterization of the properties of the learnable classes of queries and the algorithm for learning unary path queries are based on existing techniques, tree pattern homomorphisms [27, 26] and pattern learning [2, 37], but we employ them in new, nontrivial ways. The remaining results, including the remaining learning algorithms and intractability of the consistency problem, are new and nontrivial.

The paper is organized as follows. In Section 2 we introduce basic notions and define formally the learning framework. In Section 3 we define the learnable subclasses of queries and identify their essential properties that enable our learning algorithms. In Sections 4 through 7 we present the corresponding learning algorithms. In Section 8 we discuss the impact of negative examples on learning. We discuss the related work in Section 9. Finally, we summarize our results and outline further directions in Section 10. Because of space restriction we present only sketches of the most important proofs; complete proofs will be given in the full version of the paper (currently in preparation for journal submission).

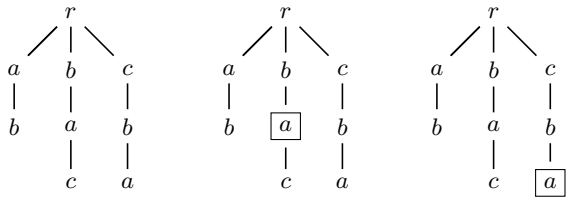
**Acknowledgments.** We would like to thank our fellow colleagues and anonymous reviewers for their helpful comments. We also would like to thank Radu Ciucanu and Ioana Adam who implemented the algorithms and shared their insights allowing to improve theoretical properties of the algorithms. This research has been partially supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region (CPER) 2007-2013, Codex project ANR-08-DEFIS-004, and Polish Ministry of Science and Higher Education research project N N206 371339.

## 2. BASIC NOTIONS

Throughout this paper we assume an infinite set of node labels $\Sigma$ which allows us to model documents with textual values. We also assume that $\Sigma$ has a total order, that can be tested in constant time, and has a minimal element that can be obtained in constant time as well. We extend the order on $\Sigma$ to the standard lexicographical order $\leq_{lex}$ on words over $\Sigma$ and define a well-founded canonical order on words: $w \leq_{can} u$ iff $|w| < |u|$ or $|w| = |u|$ and $w \leq_{lex} u$.

**Trees.** We model XML documents with unranked labeled trees. Formally, a *tree* $t$ is a tuple $(N_t, root_t, lab_t, child_t)$, where $N_t$ is a finite set of nodes, $root_t \in N_t$ is a distinguished root node, $lab_t : N_t \to \Sigma$ is a labeling function, and $child_t \subseteq N_t \times N_t$ is the parent-child relation. We assume that the relation $child_t$ is acyclic and require every non-root node to have exactly one predecessor in this relation. By $Tree_0$ we denote the set of all trees.

The *size* of a tree is the cardinality of its node set. The *depth* of a node is the length of the path from the root to the node and the *height* of the tree is the depth of its deepest leaf. For a tree $t$ by $Paths(t)$ we denote the set of paths from the root node to the leaf nodes of $t$. We view a path both as a tree, in particular it has nodes, and as a word. Often, we use unranked terms over $\Sigma$ to represent trees. For instance, the term $r(a(b), b(a(c)), c(b(a)))$ corresponds to the tree $t_0$ in Figure 3(a).

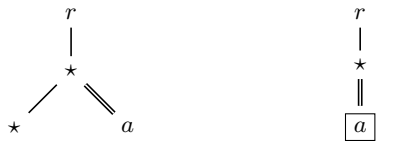


(a) Tree $t_0$. (b) Decorated trees $t_1$ and $t_2$.

**Figure 3: Trees.**

To represent examples and answers to queries, we use trees with one distinguished *selected* node. Formally, a *decorated tree* is a pair $(t, sel_t)$, where $t$ is a tree and $sel_t \in N_t$ is a distinguished *selected node*. We denote the set of all decorated trees by $Tree_1$. Figure 3(b) contains two decorated versions of $t_0$: the selected node is indicated with a square box. In the sequel, we rarely make the distinction between standard trees and decorated ones, and when it does not lead to ambiguity, we refer to both structures as simply trees.

**Queries.** We work with the class of twig queries, also know as *tree pattern queries* [1]. Twig queries are essentially unordered trees whose nodes may be additionally labeled with a distinguished wildcard symbol $\star$ and that use two types of edges, child and descendant, corresponding to the standard XPath axes. To model unary queries we also add a distinguished selecting node.



(a) Boolean twig query $q_0$. (b) Unary path query $p_0$.

**Figure 4: Twig queries.**

A *Boolean twig query* $q$ is a tuple $(N_q, root_q, lab_q, child_q, desc_q)$, where $N_q$ is a finite set of nodes, $root_q \in N_q$ is the root node, $lab_q : N_q \to \Sigma \cup \{\star\}$ is a labeling function, $child_q \subseteq N_q \times N_q$ is a set of child edges, and $desc_q \subseteq N_q \times N_q$ is a set of descendant edges. We assume that $child_q \cap desc_q = \emptyset$ and

that the relation $child_q \cup desc_q$ is acyclic and require every non-root node to have exactly one predecessor in this relation. By $Twig_0$ we denote the set of all Boolean twig queries. A *unary twig query* is a pair $(q, sel_q)$, where $q$ is a Boolean twig query and $sel_q \in N_q$ is a distinguished *selecting* node. We denote the set of all twig queries by $Twig_1$. Figure 4 contains examples of twig queries: child edges are drawn with a single line, descendant edges with a double line, and the selecting node is indicated with a square box.

Additionally, we use restricted classes of Boolean and unary path queries, $Path_0$ and $Path_1$ respectively. Formally, $Path_i$ contains those elements of $Twig_i$ whose nodes have at most one child. Furthermore, the selecting node of a unary path query is always its only leaf (cf. Figure 4(b)). We note that $Twig_1$ captures exactly the class of descending positive disjunction-free XPath queries, and in the sequel, we use elements of the abbreviated XPath syntax [43, 44] to present both elements of $Twig_1$ and $Twig_0$. For instance, the query in Figure 4(a) can be written as $r/\star[\star]//a$, and the query in Figure 4(b) as $r/\star//a$.

Because no unary twig query can select at the same time the root node and another node of a tree, we disallow the root to be an answer, and from now on, we consider only unary queries and decorated trees whose selected node is other than root. Note that this restriction can be easily bypassed by adding a virtual root node to every tree in the input sample. Also, this way the *universal query* is $\star//\star$.

**Embeddings.** We define the semantics of twig queries using the notion of embedding which is essentially a mapping of nodes of a query to the nodes of a tree (or another query) that respects the semantics of the edges of the query. In the sequel, for two $x, y \in \Sigma \cup \{\star\}$ we say that $x$ *matches* $y$ if $y \neq \star$ implies $x = y$. Note that this relation is not symmetric: $a$ matches $\star$ but $\star$ does not match $a$.

Formally, for $i \in \{0, 1\}$, a query $q \in Twig_i$ and a tree $t \in Tree_i$, an *embedding* of $q$ in $t$ is a function $\lambda : N_q \to N_t$ such that:

1. $\lambda(root_q) = root_t$,
2. for every $(n, n') \in child_q$, $(\lambda(n), \lambda(n')) \in child_t$,
3. for every $(n, n') \in desc_q$, $(\lambda(n), \lambda(n')) \in (child_t)^+$,
4. for every $n \in N_q$, $lab_t(\lambda(n))$ matches $lab_q(n)$,
5. if $i = 1$, then $\lambda(sel_q) = sel_t$.

Then, we write $\lambda : q \hookrightarrow t$ or simply $t \preccurlyeq q$.

Figure 5 presents all embeddings of the query $q_0$ in the tree $t_0$ (Figure 3(a)).

Note that we do not require the embedding to be injective i.e., two nodes of the query may be mapped to the same node of the tree. Embeddings of path queries are, however, always injective. Also, note that the semantics of //-edge is that of a proper descendant (and not that of descendant-or-self).

Typically, the semantics of a unary query is defined in terms of the set of nodes it selects in a tree [25, 26]: a node $n$ of a tree $t$ is an *answer* to a unary twig query $q$ in $t$ if there is an embedding $\lambda : q \hookrightarrow t$ such that $\lambda(sel_q) = n$ (then $n$ is also said to be *reachable* by $q$ in $t$). However, we use an alternative way of defining the semantics of a query. Formally, the *language* of a query $q \in Twig_i$ for $i \in \{0, 1\}$ is the set

$$\mathcal{L}_i(q) = \{t \in Tree_i \mid t \preccurlyeq q\}.$$

Naturally, the two notions are very closely related e.g., the decorated trees $t_1$ and $t_2$ (Figure 3) belong to $\mathcal{L}_1(p_0)$ (Figure 4) and the nodes selected in $t_1$ and $t_2$ are exactly the answers to $p_0$ in tree $t_0$.

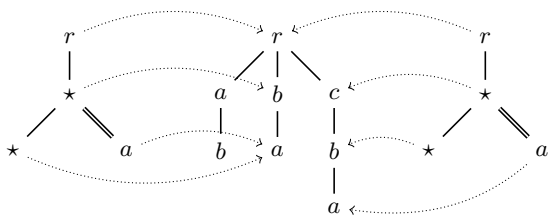



**Figure 5: Embeddings of $q_0$ in $t_0$.**

The notion of an embedding extends in a natural fashion to a pair of queries $q, p \in Twig_i$ for some $i \in \{0, 1\}$: an *embedding* of $q$ in $p$ is a function $\lambda : N_q \to N_p$ that satisfies the conditions 1, 2, 4, 5 above (with $t$ being replaced by $p$) and the following condition:

3′. for all $(n, n') \in desc_q$, $(\lambda(n), \lambda(n')) \in (child_p \cup desc_p)^+$.

Then, we write $\lambda : p \hookrightarrow q$ or simply $q \preccurlyeq p$ and say that $p$ *subsumes* $q$.

The *containment* (or *inclusion*) $q \subseteq p$ of two queries $q, p \in Twig_i$ for $i \in \{0, 1\}$ is simply $\mathcal{L}_i(q) \subseteq \mathcal{L}_i(p)$, and we say that $q$ and $p$ are *equivalent*, denoted $q \equiv p$, if $q \subseteq p$ and $p \subseteq q$. Note that for twigs, subsumption implies containment i.e., if $q \preccurlyeq p$, then $q \subseteq p$. The converse does not hold in general. For instance, we have $a[.//b] \subseteq \star[\star]$ but $a[.//b] \not\preccurlyeq \star[\star]$. There are also significant computational differences: the containment of twigs is coNP-complete [36, 30] whereas their subsumption is in PTIME.

**Query minimality.** In this paper we identify queries that are minimal for a given set of trees (as examples). It is important to emphasise that we always mean minimality in terms of query inclusion. Formally, for $i \in \{0, 1\}$, a class of queries $\mathcal{Q} \subseteq Twig_i$, a query $q \in \mathcal{Q}$, and a set of trees $S \subseteq Tree_i$, we say that $q$ is *minimal query in $\mathcal{Q}$ consistent with $S$* if $S \subseteq \mathcal{L}_i(q)$ and there is no $q' \in \mathcal{Q}$ such that $q' \subseteq q$, $q' \not\equiv q$, and $S \subseteq \mathcal{L}_i(q')$.

**Learning framework.** We use a variant of the standard language inference framework [22, 21, 32, 13] adapted to learning queries. A learning setting comprises of the set of concepts that are to be learnt, in our case queries, and the set of instances of the concepts that are to serve as examples in learning, in our case trees (possibly decorated). These two sets are bound together by the semantics which maps every concept to its set of instances.

**Definition 1** A *learning setting* is a tuple $(\mathcal{D}, \mathcal{Q}, \mathcal{L})$, where $\mathcal{D}$ is a set of examples, $\mathcal{Q}$ is a class of queries, and $\mathcal{L}$ is a function that maps every query in $\mathcal{Q}$ to the set of all its examples (a subset of $\mathcal{D}$). □

As an example, a setting for learning unary Twig queries from positive examples is the tuple $(Tree_1, Twig_1, \mathcal{L}_1)$. This general formulation allows also to easily define settings for learning from both positive and negative examples, which we present in Section 8.

To define formally what learnability for queries means we fix a learning setting $\mathcal{K} = (\mathcal{D}, \mathcal{Q}, \mathcal{L})$ and introduce some auxiliary notions. A *sample* is a finite nonempty subset $S$ of $\mathcal{D}$ i.e., a set of examples. The *size* of a sample is the sum of the sizes of the examples it contains. A sample $S$ is *consistent* with a query $q \in \mathcal{Q}$ if $S \subseteq \mathcal{L}(q)$. A *learning algorithm* is an algorithm that takes a sample and returns a query in $\mathcal{Q}$ or a special value NULL.

**Definition 2** A query class $\mathcal{Q}$ is *learnable in polynomial time and data* in the setting $\mathcal{K} = (\mathcal{D}, \mathcal{Q}, \mathcal{L})$ iff there exits a polynomial learning algorithm learner and a polynomial *poly* such that the following two conditions are satisfied:

1. **Soundness.** For any sample $S$ the algorithm learner$(S)$ returns a query consistent with $S$ or a special NULL value if no such query exists.
2. **Completeness.** For any query $q \in \mathcal{Q}$ there exists a sample $CS_q$ such that for every sample $S$ that extends $CS_q$ consistently with $q$ i.e., $CS_q \subseteq S \subseteq \mathcal{L}(q)$, the algorithm learner$(S)$ returns a query equivalent to $q$. Furthermore, the size of $CS_q$ is bounded by $poly(|q|)$. □

The sample $CS_q$ is often called the *characteristic sample* for $q$ w.r.t. learner and $\mathcal{K}$ but we point out that for a learning algorithm there may exist many samples fitting the role and the definition of learnability requires merely that one such sample exists. The soundness condition is a natural requirement but alone it is insufficient to eliminate trivial learning algorithms. For instance, for the setting where only positive examples are used, an algorithm returning the universal query $\star//\star$ is sound. Consequently, we require the algorithm to be complete analogously to how it is done for grammatical language inference [21, 32, 13]. An alternative and natural way to ban trivial learning algorithms would be to require the algorithm to return some minimal query consistent with the input sample. Our approach follows this direction but as we show later on, it is not possible to fully adhere to it because there exist samples for which the minimal consistent twig query is of exponential size.

# 3. LEARNABLE QUERY CLASSES

In this section we define the classes of queries, that in the following sections we prove learnable from positive examples, and identify two essential properties of these classes that enable our learning algorithms. Both properties follow from the importance of logical implication in learning: learning can often be seen as a search of the correct hypothesis obtained by an iterative refinement of some initial hypothesis and at every iteration the current hypothesis is often a logical consequence of the previous one. The first property requires the containment to be equivalent to subsumption, which allows to capture containment with a simple structural characterization. The second property is the existence of polynomially sized *match sets* [26], which were originally introduced as an easy way of testing query inclusion. The match sets that we construct will serve us as the characteristic samples. We emphasise that the full classes of twig and path queries do not have these properties but this does not imply that they are not learnable but it merely precludes the direct adaptation of our learning techniques. Whether the full classes of queries are learnable remains an open question.

To formally define the two properties, we fix a class of queries $\mathcal{Q}$ with their semantics defined by $\mathcal{L}$. The properties are:

($\mathbf{P}_1$) for every two $q_1, q_2 \in \mathcal{Q}$, $q_1 \subseteq q_2$ if and only if $q_1 \preccurlyeq q_2$.

($\mathbf{P}_2$) every $q \in \mathcal{Q}$ has a polynomial *match set* i.e., a set $CS_q$ of (positive) examples such that the size of $CS_q$ is polynomial in the size of $q$ and for every $q' \in \mathcal{Q}$ we have $q \subseteq q'$ if and only if $CS_q \subseteq \mathcal{L}(q')$.

We next present the construction of match sets in a generic form and then we introduce the learnable classes of queries and state the properties $\mathbf{P}_1$ and $\mathbf{P}_2$ for them.

## 3.1 Match sets as characteristic samples

We now present the construction of match sets that will be later on used as characteristic samples. Because the constructions of the match sets for all the subclasses of queries are very similar, we present it in a generic form. Take a twig query $q$, let $N$ be the size of $q$, $a_0$ be the minimal element of $\Sigma$, and $a_1$ and $a_2$ be two fresh symbols not used in $q$ and different from $a_0$. The constructed match set $CS_q$ contains exactly two trees: $t_0$ is obtained from $q$ by replacing every $\star$ with $a_0$ and every descendant edge by a child edge; $t_1$ is obtained from $q$ by replacing every $\star$ with $a_1$ and every descendant edge with a path of length $N$ whose all nodes are labeled with $a_2$. Figure 6 contains the characteristic sample for the unary twig query $q_1 = r/b[a//b]//c[d]/\star/c$. We point

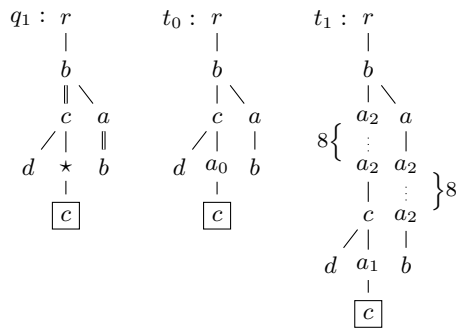



**Figure 6: The characteristic sample for $q_1$.**

out that for a query and a learning algorithm there might be more than just one characteristic sample. This is also the case with our learning algorithms. While the construction we present above might seem quite artificial, we use it due to its properties that might be of independent interest (match sets). Simpler, and easier to compose by a unskilled user, characteristic samples are often possible.

## 3.2 Anchored path queries

We begin with a base subclass of path queries, called anchored path queries. Essentially, a path query is anchored when no inner $\star$ node is incident to a //-edge. The main reason for introducing this class of queries is that when working with their embeddings the restriction on the use of // allows us to limit the "jumps" that the embedding may perform in between two nodes connected by a descendant edge. An additional restriction on the leaf node of Boolean path queries is imposed for technical reasons (cf. proof of Lemma 3.1 for more details).

Formally, the class of *unary anchored path queries* imposes one restriction: a //-edge cannot be incident to a $\star$-node unless it is the root node or the leaf node (which is also selecting). For instance, the unary queries $r//a//b/\star/c$, $\star//a//b/\star$, and $\star//\star$ are anchored but the query $r//a/\star//b$ is not. An additional restriction is imposed on the *Boolean anchored path queries*: if the leaf node is $\star$, then the edge incident to it is //. For instance, the Boolean queries $a//b/\star/c//\star$ and $a//b/\star/c//a/\star/b$ are anchored but the Boolean query $\star//a//b/\star$ is not anchored. We denote by $AnchPath_1$ and $AnchPath_0$ the sets of unary and Boolean anchored path queries respectively.

Clearly, the subclasses of anchored path queries are properly included in the full classes of path queries, however, we believe that the restrictions are not very limiting and the classes of anchored queries remain practical. Basically, anchored path queries cannot discriminate the descendants of a node based on their depth alone. We also point out that the additional restriction imposed on Boolean queries is quite minor: the Boolean query $r//a/\star$ is not anchored but it is equivalent to $r//a//\star$ which is anchored. Note, however, that the Boolean query $r//a/\star/\star$ does not have an equivalent Boolean anchored query.

While $\mathbf{P}_1$ for anchored path queries follows from the results in [27, 26], below we present a proof using a technique that allows to show $\mathbf{P}_1$ and $\mathbf{P}_2$ for all the query classes we introduce later on (and these results are new and cannot be derived from the results in [27, 26]).

**Lemma 3.1** *Unary and Boolean anchored path queries have the properties* $\mathbf{P}_1$ *and* $\mathbf{P}_2$.

To prove this lemma it suffices to show the following claim.

Claim 1 *For any* $i \in \{0,1\}$ *and any two* $q, q' \in AnchPath_i$, *if* $CS_q \subseteq \mathcal{L}_i(q')$, *then there exists an embedding* $\lambda : q' \hookrightarrow q$.

Proof We first give an equivalent yet more structured definition of anchored path queries. A *block* is a path query fragment $B$ of the form $\sigma_0/\ldots/\sigma_n$, where $n \geq 0$, $\sigma_0, \sigma_n \in \Sigma$, and $\sigma_1, \ldots, \sigma_{n-1} \in \Sigma \cup \{\star\}$. An *anchored path query* $q$ is a path query of the form $B_0//B_1//\ldots//B_k$, where $k \geq 0$, $B_i$ is a block for $1 \leq i \leq k-1$, and $B_0$ is either a block that can start with $\star$ or a single occurrence of $\star$. Also, in case of Boolean anchored path queries $B_k$ is either a block or a single occurrence of $\star$ and in case of unary anchored path queries $B_k$ is either a block that can end with $\star$ or a single occurrence of $\star$.

We first prove the claim for unary queries (i.e., $i = 1$). Let $N = |q|$ and $CS_q = \{t_0, t_1\}$ be constructed as described in Section 3.1. For every node $n$ of $t_1$ whose label is not $a_2$ by $origin(n)$ we denote the node of $q$ corresponding to $n$. Also, fix $\lambda_1 : q' \hookrightarrow t_1$.

We make several observations. First, $|q'| \leq N$, or otherwise there would be no embedding of $q'$ into $t_0$. For the same reason, $q'$ does not use the labels $a_1$ and $a_2$. Therefore, if a node $n$ of $q'$ is mapped by $\lambda_1$ to a node with label $a_1$ or $a_2$, then $lab_{q'}(n) = \star$.

Next, we show that $\lambda_1$ maps nodes of $q'$ only to those nodes of $t_1$ that are not labeled with $a_2$. This is clearly the case for the root node and the selecting node of $q'$, that are mapped to the root node and the selecting node of $t_1$, and from the construction of $t_1$, they have labels different from $a_2$. In the following we show the this is the case with other nodes.

Let $q'$ be of the form $B_0//B_1//\ldots//B_k$. Note that if a node $n$ is on the border of $B_j$ (for $0 \leq j \leq k$) then from the definition of a block $n$ cannot be mapped to $a_2$. This is because $n$ is either a root node, or a selecting node or its label is not $\star$.

Suppose, that some node of $q'$ belonging to $B_i$ for $0 \leq i \leq k$, is mapped to a node with label $a_2$ and let $n_1$ and $n_2$ be the nodes that are on the borders of $B$. Because $|B| \leq |q'| \leq N$ and in $t_1$ nodes labeled with $a_2$ come in sequences of length $N$, one of the nodes $n_1$ and $n_2$ needs to be mapped to a node labeled with $a_2$. This implies that one of $n_1$ and $n_2$ is labeled with $\star$; a contradiction.

This shows that $\lambda = \lambda_1 \circ origin$ is a properly defined function mapping $N_{q'}$ to $N_q$. We now show that $\lambda$ is an embedding of $q'$ into $q$. The condition 2 holds because $\lambda_1$ preserves the child relation and if nodes $(n_1, n_2)$ are in $child_{t_1}$ and both are not labelled with $a_2$ then $(origin(n_1), origin(n_2)) \in child_q$. The conditions 1, 3, and 5 follow from the definition of $\lambda$. For the condition 4, take any $n \in N_{q'}$ such that $\sigma = lab_{q'}(n) \neq \star$ and note that then, $\lambda_1(n)$ in $t_1$ has the same label $\sigma$ which is different from $a_1$ (because $q$ does not use $a_1$) and $a_2$ (as shown above). Therefore the node of $q$ that corresponds to $\lambda_1(n)$ is labeled with $\sigma$ as well.

The proof for Boolean anchored path queries is analogous and it suffices to consider the case when $B_k$ is a single occurrence of $\star$. Then indeed an embedding $\lambda_1 : q' \hookrightarrow t_1$ may map the $\star$-leaf to a node labeled with $a_2$. We note, however, that $\lambda_1$ can be easily altered to map the $\star$-leaf to a non $a_2$-node because the $\star$-leaf is connected to with descendant edge and every $a_2$ node in $t_1$ has a descendant that is not labeled with $a_2$. □

To show $\mathbf{P}_1$ it is enough to show the implication from left to right. Assume $q \subseteq q'$ and note that $CS_q \subseteq \mathcal{L}(q)$. Therefore, $CS_q \subseteq \mathcal{L}(q')$, which by Claim 1, gives us $q \preccurlyeq q'$. $\mathbf{P}_2$ follows directly from Claim 1.

## 3.3 Conjunctions of anchored path queries

In our approach to learn twig queries we use path learning algorithms to infer a set of path queries satisfied in the input sample and then we combine these path queries into a twig query. Therefore, the midpoint between learning path queries

and twig queries is learning conjunctions of path queries. We apply this technique only to learn Boolean twig queries and so we focus only on learning Boolean conjunctions of path queries. For convenience we use sets of Boolean path queries to represent conjunctions but a conjunction can also be seen as a Boolean twig query consisting of path queries meeting at the root node. The second representation is used to define the semantics of conjunctions and their characteristic samples.

Because our path learning algorithms infer anchored queries, we consider only conjunctions of Boolean anchored path queries. Also, if we have inferred two Boolean path queries $p_1$ and $p_2$, and $p_1$ subsumes $p_2$, then from the point of learning there is no point in keeping $p_2$ because $p_1$ contains more specific information and makes $p_2$ redundant. Consequently, we consider only *reduced* conjunctions i.e., having no two different $p_1, p_2$ such that $p_1 \subseteq p_2$. Naturally the conjunctions must be also *head-consistent* i.e., any two paths queries in a conjunction much have the same root label or otherwise we would not be able to represent it as a twig query. By $ConjPath_0$ we denote the class of conjunctions of Boolean anchored path queries satisfying the restrictions described above. The use of anchored path queries allows to prove the following lemma in a manner analogous to the proof of Lemma 3.1.

**Lemma 3.2** *Conjunctions of Boolean anchored path queries have the properties* $\mathbf{P}_1$ *and* $\mathbf{P}_2$.

## 3.4 Path-subsumption-free twig queries

As mentioned previously, our learning algorithms for twig queries attempt to construct the query $q$ by combining the path queries from a conjunction inferred beforehand. Because we infer a reduced set of path queries, the constructed Boolean twig query $q$ has no two $p_1, p_2 \in Paths(q)$ such that $p_1 \subseteq p_2$, where $Paths(q)$ is the set of Boolean path queries on paths from the root to all leaves of $q$. Naturally, all path queries in $Paths(q)$ need to be anchored. Formally, a Boolean twig query $q$ is *path-subsumption-free* iff $Paths(q)$ is a reduced set of Boolean anchored path queries and by $PsfTwig_0$ we denote the class of Boolean path-subsumption-free twig queries.

The restrictions are relaxed slightly for unary twig queries and reflect our learning algorithm that first infers a unary anchored path, and next, decorates it with elements of $PsfTwig_0$ used as filter expressions. Recall that the selecting path in a unary twig query is the path query on the path from the root node to the selecting node. Formally, a unary twig query $q$ is *path-subsumption-free* iff the unary path query from the root node to the selecting node of $q$ is anchored and every Boolean path query on the path ending at a (non-selecting) leaf node and beginning at the closest node on the selecting path is anchored. By $PsfTwig_1$ we denote the class of unary path-subsumption-free twig queries.

The classes of path-subsumption-free twig queries may seem at first very limited. We note, however, that a twig query belongs to our class if every leaf label is different or every pair of leaves with the same label cannot be compared with $\preccurlyeq$ (and all paths are anchored). This simple sufficient condition yields a rather large class of twig queries used in practice, especially if we consider the following remark. One of the advantages of considering an infinite set of labels $\Sigma$ is the ability to capture textual values (stored in the leaves of a tree). Then, non-selecting leaves of tree patterns are used for equality tests of text values, and rarely the same value is used to make an equality test (on similar paths).

**Lemma 3.3** *Path-subsumption-free twig queries have the properties* $\mathbf{P}_1$ *and* $\mathbf{P}_2$.

We point out that in the proof of Lemma 3.3 we only use the fact that path-subsumption-free twig queries are constructed from anchored paths. The other restriction, namely that the path queries in $Paths(q)$ cannot subsume one another, is not used in this proof and it is essential only for the proper work of our learning algorithms. Our recent results show that learning is possible without this restriction and we intend to present these findings in the journal version of the paper.

# 4. LEARNING UNARY PATH QUERIES

In Figure 7 we present a learning algorithm for the learning setting $\mathsf{AnchPath}_1 = (Tree_1, AnchPath_1, \mathcal{L}_1)$ inspired by and extending several learning algorithms for regular string patterns [2, 37] (cf. Section 9 for more details). Recall that $SelPath(t)$ stands for the path from the root node to the selecting node of $t$ and extend it to samples $SelPath(S) = \{SelPath(t) \mid t \in S\}$. The algorithm begins with a universal path query $\star//\star$ and considers only the paths from the root to the selected nodes in the input sample. It constructs the path query in three stages.

**algorithm** $\mathsf{learnerAnchPath}_1(S)$
**Input:** a sample $S \subseteq Tree_1$ of decorated trees
**Output:** a minimal $p \in AnchPath_1$ such that $S \subseteq \mathcal{L}_1(p)$
1: $w := \min_{\leq_{can}}(SelPath(S))$
2: **let** $w$ be of the form $a_0/a_1/\cdots/a_n$
3: $p := \star//\star$
4: **foreach** subpath $u$ of $a_1/a_2/\cdots/a_{n-1}$ in the order of decreasing lengths **do**
5: replace in $p$ any //-edge by $//u//$ as long as $S \subseteq \mathcal{L}_1(p)$
6: **let** $p$ be of the form $b_0//p_0//b_1$
7: **if** $S \subseteq \mathcal{L}_1(p\{b_0 \leftarrow a_0\})$ **then**
8: $p := p\{b_0 \leftarrow a_0\}$
9: **if** $S \subseteq \mathcal{L}_1(p\{b_1 \leftarrow a_n\})$ **then**
10: $p := p\{b_1 \leftarrow a_n\}$
11: **foreach** descendant edge $\alpha$ in $p$ **do**
12: find maximal $\ell$ s.t. $S \subseteq \mathcal{L}_1(p\{\alpha \leftarrow //(\star/)^\ell\})$
13: **if** $S \subseteq \mathcal{L}_1(p\{\alpha \leftarrow /(\star/)^\ell\})$ **then**
14: $p := p\{\alpha \leftarrow /(\star/)^\ell\}$
15: **return** $p$

**Figure 7: Learning algorithm for $\mathsf{AnchPath}_1$.**

In the first stage (lines 4 through 6) the algorithm attempts to identify a collection of factors, essentially path fragments, that are mutually common to every path in $SelPath(S)$. Note that if a factor is present in every path, then it is also present in the $\leq_{can}$-minimal path $w$. The candidate query $p$ is gradually refined with the factors and the invariant is that these factors are mutually present on every path in $SelPath(S)$ and in the specified order. For every new candidate $w'$, $\mathsf{learnerAnchPath}_1$ attempts to find a place where $w'$ can be inserted and yield a path query $p'$ consistent with $S$.

In the second stage (lines 8 through 11), the algorithm takes the query $p$ and attempts to specialise the first and the last occurrences of wildcard i.e., replace them with the corresponding symbol taken from $w$. Here, $p\{x \leftarrow e\}$ creates a copy of $p$ and replaces in it the reference $x$ by expression $e$ (the original $p$ remains unchanged). In the third stage (lines 12 through 16) the algorithm attempts to specialize every //-edge in $p$ i.e., replace it with a maximally long sequence $/\star/\star/\ldots/\star$.

**Example 3** In this example we show the execution of the algorithm $\mathsf{learnerAnchPath}_1$ on the sample $\{t_1, t_2, t_3\}$ presented in Figure 8 together with path queries constructed during the execution.

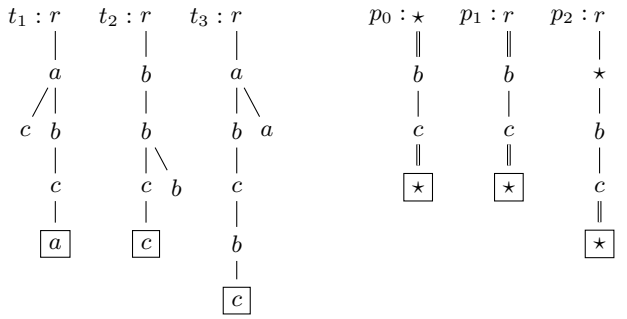



**Figure 8: A sample and the constructed queries.**

In the first stage the algorithm identifies a factor $b/c$ present in every selecting path and the resulting path query is $p_0 = \star//b/c//\star$. There is no other common factor and the algorithm moves to the second stage where it specializes the root node of $p_0$ obtaining this way $p_1 = r//b/c//\star$; the selecting node cannot be specialized because the selected nodes of $t_1$ and $t_2$ have two different labels, $a$ and $c$ resp. Finally, the algorithm attempts to specialize the descending edges. Only the top one can be replaced by a $\star$-path of length 1, yielding $p_2 = r/\star/b/c//\star$, which is also the final result of the learning algorithm. □

There are aspects of the algorithm that are not fully specified e.g., from two different subpaths of the same length which one should be chosen first in the loop in line 4. We do not enforce any particular choice because it is inessential from the theoretical point (soundness and completeness) and in practical implementations the choice could be made with the help of heuristics.

**Example 4** Consider the sample consisting of the two trees: $r(a(b(c(d))))$ and $r(b(c(a(b(d)))))$. In the first stage the algorithm may identify either a factor $a/b$ or $b/c$ but not both of them. As a consequence the algorithm may return one of two possible queries $p_1 = r//ab//d$ or $p_2 = r//bc//d$. In order to make the algorithm deterministic we may enforce some order of processing among candidate factors of the same length e.g. from left to right. □

We observe that $S \subseteq \mathcal{L}_1(p)$ is an invariant maintained throughout $\mathsf{learnerAnchPath}_1$ and with a simple analysis one can show that $\mathsf{learnerAnchPath}_1$ is sound for $\mathsf{AnchPath}_1$. But what makes this algorithm particularly interesting is the following.

**Lemma 4.1** *The algorithm* $\mathsf{learnerAnchPath}_1$ *returns a minimal anchored path query consistent with the input sample.*

We prove the claim below, which by $\mathbf{P}_1$ for $AnchPath_1$ is equivalent to the lemma above.

CLAIM 2 *If* $\mathsf{learnerAnchPath}_1(S)$ *returns* $p$*, then there is no unary anchored path query* $q \neq p$ *such that* $q \preccurlyeq p$ *and* $S \subseteq \mathcal{L}_1(q)$.

PROOF Suppose otherwise and take a unary anchored path query $q \neq p$ having an embedding $\lambda : p \hookrightarrow q$. We note that $q$ can be viewed as result of applying a substitution $\theta$ i.e., $q = p\theta$, which substitutes in $p$ some the labels of $\star$-nodes with labels in $\Sigma$ and replaces some of the //-edges with path queries. This substitution can be decomposed into a composition $\theta = \theta_1 \circ \theta_2 \circ \ldots \circ \theta_k$ of atomic operations, which for brevity we present here as rewriting rules: 1) $// \mapsto /$ replacing a //-edge with a child edge, 2) $// \mapsto //B//$ replacing a //-edge with a block $B$ (cf. proof of Claim 1), 3) $\star \mapsto a$ changing the $\star$ label of a node to some $a \in \Sigma$, 4) $// \mapsto /\star/\star/\ldots/\star/$ replacing a //-edge with a $\star$-path. Now, if we take the path query $q' = p\theta_1$, then $q \preccurlyeq q' \preccurlyeq p$ and $q' \neq p$. Consequently, it suffices to assume that $p$ is obtained from $q$ by applying just one atomic substitution $\theta^\circ$.

Essentially, the first three types of atomic operations allow to identify a new factor or a longer factor that would have been discovered and properly incorporated into the resulting query during the execution of $\mathsf{learnerAnchPath}_1(S)$ in lines 4-6. The last type, $// \mapsto /\star/\ldots/\star$ allows to identify a descending edge that would have been converted to a $\star$-path in lines 13-17. These arguments show that $p$ could not have been the result of $\mathsf{learnerAnchPath}_1(S)$; a contradiction. □

We argue that Lemmas 3.1 and 4.1 imply completeness of $\mathsf{learnerAnchPath}_1$ w.r.t. $\mathsf{AnchPath}_1$. Indeed, if $CS_q \subseteq S$ and $\mathsf{learnerAnchPath}_1(S)$ returns $p$, then $q \subseteq p$ because $CS_q \subseteq \mathcal{L}_1(p)$ but there is no query $q' \subset p$ that $S \subseteq \mathcal{L}(q')$. Hence, $q$ and $p$ are equivalent.

**Theorem 4.2** *Anchored path queries are learnable in polynomial time and data from positive examples* (*i.e., in the setting* $\mathsf{AnchPath}_1$).

# 5. LEARNING BOOLEAN PATH QUERIES

Learning Boolean path queries is more challenging than learning unary path queries. In decorated trees, which are examples for learning unary queries, the selected nodes unambiguously indicate the path to be matched by the query. The examples for learning Boolean path query are trees with no indication of the path the constructed query should match. To address this problem we devise an algorithm that infers a conjunction of Boolean anchored path queries that are satisfied in the given sample. Recall that $AnchPath_0$ is the class of Boolean anchored path queries and $ConjPath_0$ is the class of reduced and head-consistent conjunctions of Boolean anchored path queries (represented as sets of Boolean path queries). The corresponding learning settings are $\mathsf{AnchPath}_0 = (Tree_0, AnchPath_0, \mathcal{L}_0)$ and

$\mathsf{ConjPath}_0 = (\mathit{Tree}_0, \mathit{ConjPath}_0, \mathcal{L}_0)$, where $\mathcal{L}_0$ interprets a set of path queries $P$ as a the twig query obtained by gluing the root nodes together.

Figure 9 contains the learning algorithms for $\mathsf{ConjPath}_0$ and $\mathsf{AnchPath}_0$. First, we introduce $\mathsf{learnerAnchPath}_0^*$, a helper learner derived from $\mathsf{learnerAnchPath}_1$, which infers a minimal Boolean anchored path query that is satisfied by the given path $u$ and every tree in the input sample. Note that to ensure that the output is a Boolean anchored query $\mathsf{learnerAnchPath}_0^*$ skips the specialization of the last //-edge if doing so would yield a query that is not anchored (i.e., ending with $\star$ not preceded immediately by //). The purpose of taking the initial path $u$ from the input is the ability to consider every path in $S$ as the word in which to search for common factors.

**algorithm** $\mathsf{learnerAnchPath}_0^*(u, S)$
**Input:** a path $u$ and a sample $S \subseteq \mathit{Tree}_0$ of trees
**Output:** a minimal $p \in \mathit{AnchPath}_0$ s.t. $S \cup \{u\} \subseteq \mathcal{L}_0(p)$
This algorithm is obtained from $\mathsf{learnerAnchPath}_1$ by:

- initializing $w$ to $u$ (line 1)
- replacing every $S \subseteq \mathcal{L}_1(p)$ by $S \cup \{w\} \subseteq \mathcal{L}_0(p)$
- skipping the execution of loop 13–17 for the last //-edge if $b_1 = \star$.

**algorithm** $\mathsf{learnerConjPath}_0(S)$
**Input:** a sample $S \subseteq \mathit{Tree}_0$ of trees
**Output:** a set of minimal queries $P \subseteq \mathit{AnchPath}_0$ such that $S \subseteq \mathcal{L}_0(P)$

1: $P := \emptyset$
2: **for** $u \in \mathit{Paths}(S)$ **do**
3: $\quad p := \mathsf{learnerAnchPath}_0^*(u, S)$
4: $\quad$ **if** $\nexists q \in P.\ q \preccurlyeq p$ **then**
5: $\quad\quad P := P \setminus \{q \in P \mid p \preccurlyeq q\}$
6: $\quad\quad P := P \cup \{p\}$
7: **return** $P$

**algorithm** $\mathsf{learnerAnchPath}_0(S)$
**Input:** a sample $S \subseteq \mathit{Tree}_0$ of trees
**Output:** a minimal $p \in \mathit{AnchPath}_0$ such that $S \subseteq \mathcal{L}_0(p)$

1: $P := \mathsf{learnerConjPath}_0(S)$
2: choose any $p$ from $P$
3: **return** $p$

**Figure 9: Learning $\mathsf{ConjPath}_0$ and $\mathsf{AnchPath}_0$.**

Essentially, $\mathsf{learnerConjPath}_0$ considers every path $u$ in tree of $S$ and uses $\mathsf{learnerAnchPath}_0^*$ to find a most specific (i.e., minimal) Boolean path query $p$ satisfied by $u$ and every other element of $S$. The set $P$ aggregates all minimal results of running $\mathsf{learnerAnchPath}_0^*$ over all paths in the input sample. The learning algorithm $\mathsf{learnerAnchPath}_0$ simply takes the result of $\mathsf{learnerConjPath}_0$ and chooses one element. The choice is arbitrary, but later, we show that in the presence of the characteristic sample $\mathsf{learnerConjPath}$ returns a singleton and there is no ambiguity.

**Example 5** We run $\mathsf{learnerConjPath}_0$ on the sample $S_0$ (Figure 10) corresponding to the positive examples from Example 2 simplified for clarity of presentation.

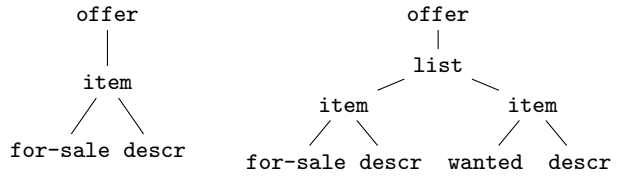



**Figure 10: Input sample from Example 5.**

The set of paths $\mathit{Paths}(S_0)$ in the sample consists of:

$$u_1 = \texttt{offer/item/for-sale},$$
$$u_2 = \texttt{offer/item/descr},$$
$$u_3 = \texttt{offer/list/item/for-sale},$$
$$u_4 = \texttt{offer/list/item/descr},$$
$$u_5 = \texttt{offer/list/item/wanted}.$$

Running $\mathsf{learnerAnchPath}_0^*$ on those paths yields:

$$\mathsf{learnerAnchPath}_0^*(u_1, S_0) = \texttt{offer//item/for-sale},$$
$$\mathsf{learnerAnchPath}_0^*(u_2, S_0) = \texttt{offer//item/descr},$$
$$\mathsf{learnerAnchPath}_0^*(u_3, S_0) = \texttt{offer//item/for-sale},$$
$$\mathsf{learnerAnchPath}_0^*(u_4, S_0) = \texttt{offer//item/descr},$$
$$\mathsf{learnerAnchPath}_0^*(u_5, S_0) = \texttt{offer//item//}\star.$$

Note that the result of $\mathsf{learnerAnchPath}_0^*$ on $u_5$ is the Boolean anchored query $\texttt{offer//item//}\star$ and not the more specific $\texttt{offer//item/}\star$ because it is not anchored; $\mathsf{learnerAnchPath}_0^*$ skips the attempt to specialize the last //-edge because it is followed by $\star$. The query $\texttt{offer//item//}\star$ is, however, subsumed by all the previous queries, and therefore, $\mathsf{learnerConjPath}_0(S_0)$ returns a set containing only the queries $\texttt{offer//item/for-sale}$ and $\texttt{offer//item/descr}$. The run of $\mathsf{learnerAnchPath}_0$ on $S_0$ returns one of those queries e.g., the one whose string representation is lexicographically minimal $\texttt{offer//item/descr}$. While this is not best choice for Example 2, the negative examples can be used in a heuristic to select a query rejecting the most negative examples, in this case $\texttt{offer//item/for-sale}$. □

Because $\mathsf{learnerConjPath}_0(S)$ returns a set $P$ of Boolean path queries that are satisfied in every tree in $S$, this algorithm is sound. Naturally, $\mathsf{learnerAnchPath}_0$ is also sound because it returns one element of $P$. To show completeness of both learning algorithms, we point out an important property of $\mathsf{learnerAnchPath}_0^*$. The construction of characteristic samples $CS_P$ and $CS_p$ is in Section 3.1.

**Lemma 5.1** *Take a conjunctive query $P \in \mathit{ConjPath}_0$, let $CS_P = \{t_0, t_1\}$ be the characteristic sample for $P$, and take any sample $S \subseteq \mathcal{L}_0(P)$ containing two examples $t_0', t_1'$ such that $\mathit{Paths}(t_i) = \mathit{Paths}(t_i')$ for $i \in \{0, 1\}$. Then,*

1. *for every $u \in \mathit{Paths}(S)$, $\mathsf{learnerAnchPath}_0^*(u, S)$ returns a path query equal to or subsumed by some $p \in P$.*
2. *for every $p \in P$ there exists $u \in \mathit{Paths}(S)$ such that $\mathsf{learnerAnchPath}_0^*(u, S)$ returns $p$.*

The above result shows completeness of $\mathsf{learnerConjPath}_0$. As for $\mathsf{learnerAnchPath}_0$, if we take a Boolean anchored path

query $p$ and apply the previous lemma to $P = \{p\}$, we get that for any sample $S$ consistent with $p$ and containing $CS_p$ the algorithm $\mathsf{learnerConjPath}_0(S)$ returns the singleton $\{p\}$, and thus, $\mathsf{learnerAnchPath}_0(S)$ returns $p$. This result allows to prove learnability of both classes of queries.

**Theorem 5.2** *The query classes $ConjPath_0$ and $AnchPath_0$ are learnable in polynomial time and data from positive examples (i.e., in the settings $\mathsf{ConjPath}_0$ and $\mathsf{AnchPath}_0$ resp.)*

We also show minimality of $\mathsf{learnerAnchPath}_0$.

**Lemma 5.3** *For any finite $S \subseteq Tree_0$, $\mathsf{learnerConjPath}_0(S)$ returns a set of minimal Boolean anchored path queries consistent with $S$ and $\mathsf{learnerAnchPath}_0(S)$ returns a minimal Boolean anchored path query consistent with $S$.*

We point out that while the result of $\mathsf{learnerConjPath}_0(S)$ is a set of minimal queries, it is not necessarily a minimal conjunctive query i.e., it is not a maximal set of minimal queries. In the example below we show that a set of positive examples may have an exponential number of minimal Boolean path queries, and therefore, constructing their conjunction cannot be done in polynomial time.

**Example 6** Fix $n > 0$ and take the set of positive examples $S_{\text{exp}}$ of containing exactly two trees

$$t_0 = r(a_1(b_1(\dots a_n(b_n(c))\dots))),$$
$$t_1 = r(b_1(a_1(\dots b_n(a_n(c))\dots))).$$

Any query of the form $r//\beta_1// \dots //\beta_n//c$, with $\beta_i \in \{a_i, b_i\}$ and $i \in \{1, \dots, n\}$, is a minimal Boolean path query consistent with $S_{\text{exp}}$. □

# 6. LEARNING BOOLEAN TWIG QUERIES

It this section we investigate learning path-subsumption-free twig queries from positive examples i.e., the learning setting $\mathsf{PsfTwig}_0 = (Tree_0, PsfTwig_0, \mathcal{L}_0)$. Recall that $q \in PsfTwig_0$ is query such that the set of root-to-leaf paths $Paths(q)$ consists of Boolean anchored path queries and does not contain two path queries such that one subsumes another. Our approach is based on the algorithm $\mathsf{learnerConjPath}_0$, which infers a set $P$ of minimal Boolean path queries and a method that allows to reconstruct a twig query from path queries in $P$. Intuitively speaking, we shall interleave the path queries from $P$ to obtain the twig query. Below, we describe formally this technique.

Given a path query $p$ and a node $n \in N_p$, the *split* of $p$ at $n$ is a pair of path queries $p_1$ and $p_2$ such that $p_1$ is the path from $root_p$ to $n$ and $p_2$ is the path from $n$ to the only leaf of $p$. Note that $n$ becomes the root node of $p_2$. A *fusion* of $p$ into a twig query $q$ is a twig query $q'$ such that the pair $p_1$ and $p_2$ is a split of $p$ at $n$, there exists an embedding $\lambda : p_1 \hookrightarrow q$, and $q'$ is obtained from $q$ by attaching $p_2$ at node $\lambda(n)$ (the node $\lambda(n)$ and the root node $n$ of $p_2$ become the same node, the label of $n$ in $p_2$ is ignored). By $Fusions(p, q)$ we denote the set of all fusions of $p$ into $q$. Figure 11 presents all fusions of $r//a/b$ into $r[\star/a]//a/c$.

We point out that if $q$ is path-subsumption-free and $p$ is anchored, then all elements of $Fusions(p, q)$ are path-subsumption-free. We note that $Fusions(p, q)$ may be empty

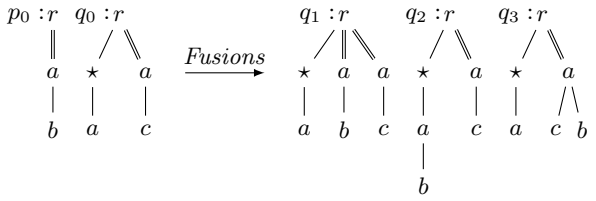



**Figure 11: Fusions of $p_0$ into $q_0$.**

e.g., there is no fusion of $a/a$ into $b[a]/b$, but as we argue next, this is never the case in the learning algorithm $\mathsf{learnerPsfTwig}_0$ which we present in Figure 12. We slightly extend the notation: $\varnothing$ denotes a *phantom* empty twig query and $Fusions(\varnothing, p) = \{p\}$.

**algorithm** $\mathsf{learnerPsfTwig}_0(S)$
**Input:** a sample $S \subseteq Tree_0$ of trees
**Output:** a query $p \in PsfTwig_0$ such that $S \subseteq \mathcal{L}_0(p)$
1: $q := \varnothing$
2: $P := \mathsf{learnerConjPath}_0(S)$
3: **for** $p \in P$ **do**
4: $\quad C := \{q' \in Fusions(p, q) \mid S \subseteq \mathcal{L}_0(q')\}$
5: $\quad q :=$ choose any $\preccurlyeq$-minimal element of $C$
6: **return** $q$

**Figure 12: Learning algorithm for $\mathsf{PsfTwig}_0$.**

Basically, $\mathsf{learnerPsfTwig}_0$ uses $\mathsf{learnerConjPath}_0$ to construct a set $P$ of Boolean path queries satisfied in all trees of $S$ and then fusions all the paths into one twig query. Note that $C$ is never empty because $q$ is build up from path queries in $P$ that are satisfied in $S$ and have the same label in their root nodes. Consequently, $\mathsf{learnerPsfTwig}_0$ executes without errors and is sound. The order in which $\mathsf{learnerPsfTwig}_0$ performs fusions is arbitrary, but later on, we show that in the presence of the characteristic sample, the set $C$ has exactly one element at all times, and the final result is the goal query. First, we illustrate the work of $\mathsf{learnerPsfTwig}_0$ on an example.

**Example 7** Consider a sample $S_1$ containing two DBLP listings in Figure 13: one with a collection of articles and the other with a collection of books.

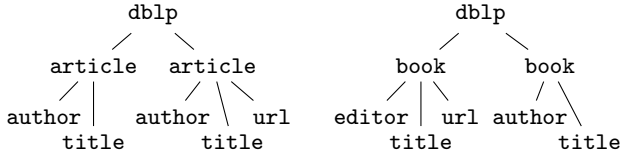



**Figure 13: Input sample**

$\mathsf{learnerConjPath}_0(S_1)$ returns the following path queries:

$$p_1 = \texttt{dblp}/\star/\texttt{author}, \quad p_2 = \texttt{dblp}/\star/\texttt{title}, \quad p_3 = \texttt{dblp}/\star/\texttt{url}.$$

We perform fusions in the order $p_1$, $p_2$, and $p_3$. Fusing $p_1$ and $p_2$ yields the query $\texttt{dblp}/\star[\texttt{title}]/\texttt{author}$ and fusing $p_3$ into it gives $q' = \texttt{dblp}[\star/\texttt{url}]/\star[\texttt{title}]/\texttt{author}$. Note that in the last step, $\texttt{dblp}/\star[\texttt{title}][\texttt{url}]/\texttt{author}$ is one of the fusions but it is not consistent with the input sample $S_1$. On the

other hand, if the order of fusions is $p_2$, $p_3$, and $p_1$, then the end result is $q'' = \texttt{dblp}[\star/\texttt{author}]/\star[\texttt{title}]/\texttt{url}$. □

While in the previous example the queries $q'$ and $q''$ are minimal path-subsumption-free twig queries consistent with $S_1$, in general $\mathsf{learnerPsfTwig}_0$ does not need to produce such minimal queries. In fact, we show that for certain samples, such a minimal query may be of exponential size and thus impossible to construct by a polynomial algorithm.

**Example 8 (cont'd Example 6)** Recall the sample $S_{\text{exp}}$ and observe that the minimal twig query consistent with $S_{\text{exp}}$ has the shape of a perfect binary tree of height $n+1$ where every node at depth $i \in \{0, \ldots, n-1\}$ has two children labeled with $a_{i+1}$ and $b_{i+1}$ (connected with their parent with a //-edge). Naturally, this minimal query is path-subsumption-free. □

Now, we move to completeness of $\mathsf{learnerPsfTwig}_0$ and we fix a query $q \in \mathit{PsfTwig}_0$ and a sample $S \subseteq \mathcal{L}_0(q)$. Recall the construction of the characteristic sample $CS_q$ for $q$ from Section 3.1. First, we observe that for $q \in \mathit{PsfTwig}_0$ every $p \in \mathit{Paths}(q)$ is a $\preccurlyeq$-minimal element of $\mathit{Paths}(q)$. As a simple consequence of Lemma 5.1 we get the following.

**Lemma 6.1** *If $S$ contains $CS_q$, then* $\mathsf{learnerConjPath}_0(S)$ *returns $\mathit{Paths}(q)$.*

To state that the algorithm approaches the goal query $q$ with every fusion, we need to define formally the search space of subqueries of $q$ and show that when moving with the fusion operator we never leave the space and finally reach $q$. A Boolean twig query $q'$ is a *subquery* of $q$ if there exists a subset $N$ of leaves of $q$ such that $q'$ is a subgraph induced by the set of paths from the root of $q$ to the leaves in $N$. The main claim follows.

**Lemma 6.2** *Assume that $CS_q \subseteq S$. For any subquery $q'$ of $q$, and any path query $p \in \mathit{Paths}(q) \setminus \mathit{Paths}(q')$ the set of elements of $\mathit{Fusions}(p, q')$ consistent with $S$ has exactly one $\preccurlyeq$-minimal element $q''$. Furthermore, $q''$ is a subquery of $q$.*

If $CS_q \subseteq S$, then by Lemma 6.1 $P = \mathit{Paths}(q)$, and therefore, whatever is the order of choosing paths from $P$ in line 3, the algorithm $\mathsf{learnerPsfTwig}_0$ approaches $q$ and when all paths in $P$ are fused, we obtain $q$.

**Theorem 6.3** *Path-subsumption-free Boolean twig queries are learnable in polynomial time and data from positive examples (i.e., in the setting* $\mathsf{PsfTwig}_0$*).*

# 7. LEARNING UNARY TWIG QUERIES

In this section, we present an algorithm $\mathsf{learnerPsfTwig}_1$ (Figure 14) for learning unary path-subsumption-free twig queries from positive examples i.e., in the learning setting $\mathsf{PsfTwig}_1 = (\mathit{Tree}_1, \mathit{PsfTwig}_1, \mathcal{L}_1)$.

Essentially, the learning algorithm uses $\mathsf{learnerAnchPath}_1$ to construct a path query $p$ and then it uses $\mathsf{learnerPsfTwig}_1^*$, a helper learner derived from $\mathsf{learnerPsfTwig}_0$, to decorate the nodes of the path query with filter expressions (Boolean twig queries). Here, we use the non-abbreviated syntax of XPath

**algorithm** $\mathsf{learnerPsfTwig}_1^*(S, q')$
**Input:** a sample $S \subseteq \mathit{Tree}_1$ of decorated trees and a query $q' \in \mathit{PsfTwig}_1$ such that $S \subseteq \mathcal{L}_1(q')$
**Output:** a query $q \in \mathit{PsfTwig}_1$ s.t. $q \preccurlyeq q'$ and $S \subseteq \mathcal{L}_1(q)$
This algorithm is obtained from $\mathsf{learnerPsfTwig}_0$ by:
- initializing $q$ to $q'$ (line 1)
- replacing every $\mathcal{L}_0$ by $\mathcal{L}_1$

**algorithm** $\mathsf{learnerPsfTwig}_1(S)$
**Input:** a sample $S$ of decorated trees
**Output:** a query $q \in \mathit{PsfTwig}_1$ such that $S \subseteq \mathcal{L}_1(q)$
1: $p := \mathsf{learnerAnchPath}_1(S)$
2: let $p$ be of the form $\ell_0/\alpha_1{::}\ell_1/\ldots/\alpha_k{::}\ell_k$
3: $q'_k := \ell_k$
4: **for** $i = k, \ldots, 0$ **do**
5: $\quad S_i := \emptyset$
6: $\quad$ **for** $t \in S$ **do**
7: $\quad\quad$ let $n$ be the deepest node on the path from the root node $root_t$ to the selected node $sel_t$, such that $n$ is reachable from $root_t$ with $\ell_0/\alpha_1{::}\ell_1/\ldots/\alpha_i{::}\ell_i$ and $sel_t$ is reachable from $n$ with $q'_i$
8: $\quad\quad$ add the subtree of $t$ rooted at $n$ to $S_i$
9: $\quad q_i := \mathsf{learnerPsfTwig}_1^*(S_i, q'_i)$
10: $\quad$ **if** $i > 0$ **then**
11: $\quad\quad q'_{i-1} := \ell_{i-1}/\alpha_i{::}q_i$
12: **return** $q_0$

**Figure 14: Learning algorithm for $\mathsf{PsfTwig}_1$.**

to represent the path query $p$ as $\ell_0/\alpha_1{::}\ell_1/\ldots/\alpha_k{::}\ell_k$, where $\ell_i \in \Sigma \cup \{\star\}$ and $\alpha_i$ is either *child* or *descendant*.

When decorating the $i$-th step of $p$ i.e., the fragment $\alpha_i{::}\ell_i$, with a filter expression, the algorithm first constructs a sample $S_i$ of subtrees that serve as positive examples for learning the corresponding filter expression. From every decorated tree in the input sample $S$ one subtree is extracted. Each subtree is rooted at a node $n$ on the path from the root node to the selected node of the decorated tree $t$. The choice of $n$ is done so that it can be reached with the unprocessed part of the path query $\ell_0/\alpha_1{::}\ell_1/\ldots/\alpha_i{::}\ell_i$ and at the same time the decorated part of the path query $q'_i$ selects the selected node $sel_t$ when evaluated from $n$. An important invariant of the outer for loop (lines 4-12) is that there is at least one such $n$ for every $t \in S$. If there is more than one possible choice, the deepest node is chosen.

**Example 9** Consider a sample $S_2$ (Figure 15) that contains the positive examples corresponding to (a simplified version of) the document from Example 1. $\mathsf{learnerAnchPath}_1(S_2)$

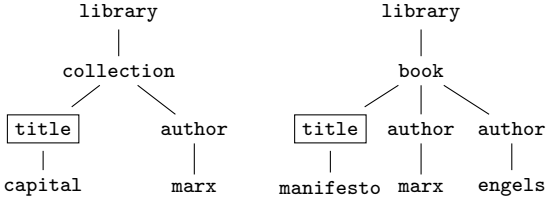



**Figure 15: Examples from a library database**

returns the query $p = /\texttt{library}/\star/\texttt{title}$. The algorithm attempts to specialize the bottom fragment $q_2' = \texttt{title}$ using the two subtrees $\texttt{title}(\texttt{capital})$ and $\texttt{title}(\texttt{manifesto})$. The only Boolean anchored path query these subtrees do have in common is $\texttt{title}//\star$, which is fused into the query yielding $q_2 = \texttt{title}[.//\star]$. Next, the algorithm moves to $q_1' = \star/\texttt{title}[.//\star]$ and calls $\mathsf{learnerPsfTwig}_1^*$ with two subtrees: one at the node `collection` and one rooted at the node `book`. $\mathsf{learnerConjPath}_0$ called with these two trees on input returns two path queries $\star/\texttt{title}//\star$ and $\star/\texttt{author}/\texttt{marx}$. The first path query is subsumed by $q_1'$, and therefore, it is absorbed by $q_1'$ when fusing. Fusing the second path query into $q_1'$ yields the query $q_1 = \star[\texttt{author}/\texttt{marx}]/\texttt{title}[.//\star]$. Finally, the algorithm moves level up to the query $q_0' = q_0 = \texttt{library}/\star[\texttt{author}/\texttt{marx}]/\texttt{title}[.//\star]$, which is also the end result of $\mathsf{learnerPsfTwig}_1$. □

We observe that $q_0$ can be considered as overspecialized: it contains the filter expression $[.//\star]$ which tests that the selected `title` nodes have contents, a test trivially true in the presence of a reasonable schema information. Currently, however, our algorithms do not take advantage of schema information.

The soundness of $\mathsf{learnerPsfTwig}_1$ follows from the invariant of the main loop (lines 4–12): for every $t \in S$ in line 7 there is at least one node with the desired property. Completeness of $\mathsf{learnerPsfTwig}_1$ follows essentially from completeness of the algorithms $\mathsf{learnerAnchPath}_1$ and $\mathsf{learnerPsfTwig}_0$, and from the fact that in line 7 we chose the deepest node.

**Theorem 7.1** *Path-subsumption-free unary twig queries are learnable in polynomial time and data from positive examples (i.e., in the setting* $\mathsf{PsfTwig}_1$*).*

# 8. IMPACT OF NEGATIVE EXAMPLES

In the previous sections, we considered the setting where the user provides positive examples only. In this section, we allow the user to additionally specify negative examples. We use two symbols + and − to mark whether an example $t$ of some query is a positive one $(t, +)$ or a negative one $(t, -)$. Formally, for $i \in \{0, 1\}$ we consider the following learning settings: $\mathsf{Path}_i^{\pm} = (\mathit{Tree}_i^{\pm}, \mathit{Path}_i, \mathcal{L}_i^{\pm})$ and $\mathsf{Twig}_i^{\pm} = (\mathit{Tree}_i^{\pm}, \mathit{Twig}_i, \mathcal{L}_i^{\pm})$, where $\mathit{Tree}_i^{\pm} = \mathit{Tree}_i \times \{+, -\}$ and $\mathcal{L}_i^{\pm}(q) = \mathcal{L}_i(q) \times \{+\} \cup (\mathit{Tree}_i \setminus \mathcal{L}_i(q)) \times \{-\}$.

We study the problem of checking whether there even exists a query consistent with the input sample because any sound learning algorithm needs to return NULL if and only if there is no such query. Formally, given a learning setting $\mathcal{K} = (\mathcal{D}, \mathcal{C}, \mathcal{L})$, the *$\mathcal{K}$-consistency* is the following decision problem

$$\mathsf{CONS}_{\mathcal{K}} = \{S \subseteq \mathcal{D} \mid \exists q \in \mathcal{C}.\ S \subseteq \mathcal{L}(q)\}.$$

Note that in the presence of positive examples the consistency problem is trivial as long as the query class contains the universal query $\star//\star$. In the presence of negative examples this problem becomes quite complex.

**Theorem 8.1** $\mathsf{Twig}_i^{\pm}$*-consistency is NP-complete for any* $i \in \{0, 1\}$ *(even in the presence of one negative example).*

PROOF We only outline the proof of NP-hardness of $\mathsf{Twig}_0^{\pm}$-consistency with a reduction from SAT. Showing the membership to NP is more difficult, uses a nontrivial minimal-witness argument, and is omitted.

We illustrate the reduction on an example of a CNF formula $\varphi_0 = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2)$ for which the corresponding sample is presented in Figure 16 (positive and negative examples are indicated with the symbols + and − respectively).

The building block of the reduction is a *brush tree* which is used to encode Boolean valuations and constraints on them. For instance, for the set of variables $\{x_1, x_2, x_3\}$ the full brush tree is $d(x_1(0,1), x_2(0,1), x_3(0,1))$ but typically we remove some of the leaves. For instance, the valuation $V_0 = \{(x_1, \mathit{false}), (x_2, \mathit{false}), (x_2, \mathit{true})\}$ is represented by the tree $t_0 = d(x_1(0), x_2(0), x_3(1))$. Note that the tree pattern $c(t_0)$ separates the positive examples from the negative ones in Figure 16 because $V_0$ satisfies $\varphi_0$.

The constructed set of examples consists of several $c$-trees. The positive $c$-trees specify the satisfying valuations of the input CNF formula; there is one $c$-tree per clause of the input formula. Each $c$-tree contains one brush tree per literal of the clause, every brush tree encoding the valuations that satisfy the corresponding literal (one leaf removed). The negative $c$-tree ensures that a brush filter that separates the positive examples from negative is well-formed and encodes a valuation. This $c$-tree contains one brush tree per variable of the input formula, every brush tree has both leaves of the corresponding variable $x_i$ removed. We claim that this set of examples is consistent if and only if the input CNF formula is satisfiable. The *if* part is trivial and the proof of the *only if* part is technical and uses the observation that the depth of any twig query separating the positive examples from negative ones is bounded by 4. □

The result holds even for very limited query classes that do not use //-edges and $\star$, and in particular the result hold for path-subsumption-free twig queries.

The problem of consistency of the input sample in the presence of positive and negative examples has also been consideed for string patterns and found to be NP-complete [28]. The proof can be easily adapted to show the following.

**Theorem 8.2** $\mathsf{Path}_i^{\pm}$*-consistency is NP-complete for any* $i \in \{0, 1\}$*.*

We remark, however, that the proof cannot be extended to twig queries because these are much more expressive even when interpreted over linear trees (words).

Overall, the negative results for checking consistency give us

**Corollary 8.3** *Unless* $P = NP$*, none of the classes* $\mathit{Path}_i$ *and* $\mathit{Twig}_i$ *for* $i \in \{0, 1\}$ *is learnable in polynomial time and data in the presence of positive and negative examples.*

# 9. RELATED WORK

Our research adheres to computational learning theory [22], a branch of machine learning, and in particular, to the area

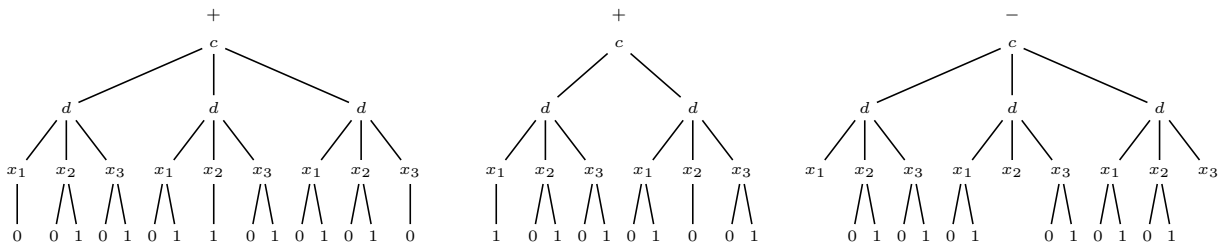



**Figure 16: Reduction of SAT to $\mathsf{Twig}_0^{\pm}$-consistency for $\varphi_0 = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2)$.**

of language inference [21]. Our learning framework is inspired by the one generally used for inference of languages of word and trees [31, 33] (see also [14] for survey of the area). Analogous frameworks have been employed in the context of XML for learning of DTDs and XML Schemas [9, 8], XML transformations [23], and $n$-ary automata queries [11].

Because the positive examples are generally believed to be easier to obtain, learning from positive examples only is desirable. However, many classes of languages are learnable only in the presence of both positive and negative examples e.g., regular languages [21], deterministic regular languages [8] are not learnable from positive examples only, in fact any *superfinite* language class, a class containing all finite languages and at least one infinite, cannot be learned from positive examples even if we consider algorithms that do not work in polynomial time. To enable learning from only positive examples various restrictions have been considered e.g., reversible languages [4], $k$-testable languages [18], languages of $k$-occurrence regular expressions [8], and $(k, l)$-contextual tree languages [34]. What is important to point out here is that the ability to learn subclasses of path and twig queries from positive examples comes from the fact that the expressive power of path and twig queries is relatively weak. Paradoxically, the very same fact is also responsible for the unfeasability to learn path and twig queries when both positive and negative examples are present.

Our basic learning algorithm for unary embeddable path queries is inspired and can be seen as an extension of algorithms for inference of word patterns [2, 37] (see [38] for a survey of the area). A word pattern is a word using extra wildcard characters. For instance, *regular patterns* use a wildcard $\oplus$ matching any nonempty string e.g., $a{\oplus}b{\oplus}c$ matches *aabbc* and *abbbc* but not *abbc*, *abc*, and *cbc*. *Extended regular patterns* use a wildcard $\circledast$ that matches any (possibly empty) string e.g., $a{\circledast}b$ matches *ab* and *acbcb*. To capture unary path queries we need to use the wildcard $\oplus$ and another wildcard $\odot$ that matches a single letter, and then for instance the pattern $a{\oplus}b{\odot}c$ corresponds to the path query $/a//b/\star/c$ when interpreted over paths of the input tree. We observe that $\oplus$ is equivalent to $\circledast\odot$ and engineer our learning algorithm using the ideas behind the algorithms for inference of regular patters [2] and extended regular patters [37].

Learning of unary XML queries has been pursued with the use of node selecting tree automata [11], with extensions allowing to infer $n$-ary queries [24], take advantage of schema information [12], and use pruning techniques to handle incompletely annotated documents [11]. The main advantage of using node selecting tree automata is their expressive power. Node selecting tree automata capture exactly the class of $n$-ary MSO tree queries [40, 24], which properly includes twig and path queries. However, tree automata have several drawbacks which may render them unsuitable for learning in certain scenarios: this is a heavy querying formalism with little support from the existing infrastructure and it does not allow an easy visualization of the inferred query.

Although, the class of twig queries is properly included by the class of MSO queries and path queries are captured by regular languages, using automata-based techniques to infer the query and then convert it to twigs is unlikely to be successful because automata translation is a notoriously difficult task and typically leads to significant blowup [16] and it is generally considered beneficial to avoid it [17]. An alternative approach, along the lines of [9], would be to define a set of structural restrictions on the automaton that would ensure an easy translation to twig queries and enforce those conditions during inference. However, such restrictions would need to be very strong, at least for twig queries, and this approach would require significant modification of the inference algorithm, to the point where it would constitute a new algorithm.

Methods used for inference of languages represented by automata differ from the methods used in our learning algorithms. An automata-based inference typically begins by constructing an automaton recognizing exactly the set of positive examples, which is then generalized by a series of generalization operation e.g., fusions of pairs of states. To avoid overgeneralization of the automata, negative examples are used to filter only consistent generalizations operations [32], and if negative examples are not available, structural properties of the automata class can be used to pilot the generalization process [4, 18, 8]. Our algorithms, similarly to word pattern inference algorithms [2, 37], begin with the universal query and iteratively specialize the query by incorporating subfragments common to all positive examples.

XLearner [29] is a practical system that infers XQuery programs. It uses Angluin's DFA inference algorithm [5] to construct the XPath components of the XQuery program. The system uses direct user interaction, essentially equivalence and membership queries, to refine the inferred query. Because of that the learning framework, called the *minimally adequate teacher* [5], is different from ours and allows to infer more powerful queries. We also point out that learning twigs is not feasible with equivalence queries only [10].

Raeymaekers et al. propose learning of $(k, l)$-contextual tree languages to infer queries for web wrappers [34]. $(k, l)$-contextual tree languages form a subclass of regular tree languages that allows to specify conditions on the nodes of the tree at depth up to $l$ and each condition involves exactly $k$ subsequent children of a node. Because only nodes at bounded depth can be inspected and the relative order among children is used, $(k, l)$-contextual tree languages are incomparable with twig queries which can inspect nodes at arbitrary depths but ignore the relative order of nodes.

Finally, we point out that the problem of query inference has been studied in the setting of relational setting [35, 41, 19]. Relational databases and their query languages offer a set of opportunities and challenges radically different from those encountered in semi-structured databases. For instance, the query inference involves constructing a desired selection condition that yields the required tuples from a table, a task that easily becomes intractable.

# 10. CONCLUSIONS AND FUTURE WORK

We have studied the problem of inferring an XML query from examples given by the user. We have investigated several classes of Boolean and unary, path and twig queries and considered two settings for the problem: one allowing positive examples only and one that allows both positive and negative examples. For the setting with positive examples only, we have presented sound and complete learning algorithms for practical subclasses of queries: anchored path queries and path-subsumption-free twig queries. On the other hand, inclusion of negative examples to the input sample renders learning unfeasible.

We believe that negative examples have an important informative quality and we intend to investigate approaches that take advantage of it. Two directions are possible: relaxing the definition of learnability and extending the query class. A notion allowing the query to select some negative examples and omit some positive examples is a natural direction of making our learning algorithms capable of producing queries of better quality (cf. Example 5) and able to handle noisy samples. For the second direction, our preliminary results show that adding union to the query languages renders consistency quite simple to decide but the satisfaction of $\mathbf{P}_1$ and $\mathbf{P}_2$ is not clear, and therefore, new learning techniques need to be developed. We are also interested in extending the query language with other operators (e.g., negation) and see their impact on learnability.

We observe that the main reason for restricting our attention to anchored path queries are the properties $\mathbf{P}_1$ and $\mathbf{P}_2$ defined in Section 3 that allow to use embeddings to equate the semantics of the query with its structure and enforce the existence of match sets of polynomial size. [27, 26] introduced adorned path queries, allowing to represent $//\star/\star$ as $//^{\geq 2}$, and extended embeddings to homomorphisms of adorned queries. Homomorphisms are shown to connect tightly the structure of path queries and their semantics ($\mathbf{P}_1$). It would be interesting to see to what extent the notion of homomorphism could be used to improve learnability results. We point out that for path queries the only know construction of match sets produces exponential sets. Moreover, the homomorphism technique does not work for twig queries.

Finally, we would like to enable our algorithms to take advantage of schema information (cf. Example 9). The schema may be given explicitly e.g., as a DTD, or implicitly as a result of a learning algorithm. Because testing the containment of XPath queries in the presence of DTDs is know to be intractable in general [15, 30, 7] and in fact most of the reductions showing hardness use (or can be modified to use) anchored queries, the use of DTDs in this context may be quite limited and we intend to investigate alternative schema formalisms tailored for query learning.

# 11. REFERENCES


[1] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB Journal*, 11(4):315–331, 2002.

[2] D. Angluin. Finding patterns common to a set of strings. In *ACM Symposium on Theory of Computing (STOC)*, pages 130–141. ACM, 1979.

[3] D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980.

[4] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, November 1987.

[6] M. Arenas and L. Libkin. XML data exchange: Consistency and query answering. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 13–24, 2005.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *ACM Symposium on Principles of Database Systems (PODS)*, 2005.

[8] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web*, 4(4), 2010.

[9] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems (TODS)*, 35(2), 2010.

[10] J. Carme, M. Ceresna, and M. Goebel. Query-based learning of xpath expressions. In *International Colloquium on Grammatical Inference (ICGI)*, pages 342–343, 2006.

[11] J. Carme, R. Gilleron, A. Lemay, and J. Niehren. Interactive learning of node selecting tree transducers. *Machine Learning*, 66(1):33–67, 2007.

[12] J. Champavère, R. Gilleron, A. Lemay, and J Niehren. Schema-guided induction of monadic queries. In *International Colloquium on Grammatical Inference (ICGI)*, pages 15–28, 2008.

[13] C. de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, 1997.

[14] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38:1332–1348, September 2005.

[15] A. Deutsch and A. Tannen. Containment and integrity constraints for XPath. In *KRDB*, 2001.

[16] A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[17] Henning Fernau. Extracting minimum length document type definitions is NP-hard. In *International Colloquium on Grammatical Inference (ICGI)*, pages 277–278, 2004.

[18] P. Garcia and E. Vidal. Inference of $k$-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, 1990.

[19] J. Gillis and J. Van den Bussche. Induction of relational algebra expressions. In *Inductive Logic Programming (ILP)*, pages 25–33, 2009.

[20] M. Goebel and M. Ceresna. Wrapper induction. In *Encyclopedia of Database Systems*, pages 3560–3565. Springer US, 2009.

[21] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[22] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, 1994.

[23] A. Lemay, S. Maneth, and J. Niehren. A learning algorithm for top-down XML transformations. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 285–296, 2010.

[24] A. Lemay, J. Niehren, and R. Gilleron. Learning $n$-ary node selecting tree transducers from completely annotated examples. In *International Colloquium on Grammatical Inference (ICGI)*, pages 253–267, 2006.

[25] L. Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

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

[27] T. Milo and D. Suciu. Index structures for path expressions. In *International Conference on Database Theory (ICDT)*, pages 277–295, 1999.

[28] S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18(3):217–242, 2000.

[29] A. Morishima, H. Kitagawa, and A. Matsumoto. A machine learning approach to rapid development of XML mapping queries. In *International Conference on Data Engineering (ICDE)*, 2004.

[30] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *International Conference on Database Theory (ICDT)*, pages 315–329. Springer-Verlag, 2003.

[31] J. Oncina and P. Garcia. Inference of rational tree sets. Technical Report DSIC-ii-1994-23, Departamento de Sistemas Informaticos y Computaction, Universidad Politecnica de Valencia, 1994.

[32] J. Oncina and P. Gracia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, 1991.

[33] J. Oncina and P. Gracia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992.

[34] S. Raeymaekers, M. Bruynooghe, and J. Van den Bussche. Learning $(k, l)$-contextual tree languages for information extraction from web pages. *Machine Learning*, 71(2–3):155–183, 2008.

[35] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J Widom. Synthesizing view definitions from data. In *International Conference on Database Theory (ICDT)*, pages 89–103, 2010.

[36] T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[37] T. Shinohara. Polynomial time inference of extended regular pattern languages. In *RIMS Symposium on Software Science and Engineering*, pages 115–127. Springer-Verlag, LNCS 147, 1982.

[38] T. Shinohara and S. Arikawa. Pattern inference. In *Algorithmic Learning for Knowledge-Based Systems*, volume 961, pages 259–291. Springer LNCS 961, 1995.

[39] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1–3):233–272, 1999.

[40] J. W. Thatcher and Wright J. B. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

[41] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *ACM SIGMOD International Conference on Management of Data*, pages 535–548, 2009.

[42] W3C. Extensible markup language (XML) 1.0, 1999. `http://www.w3.org/TR/xml/`.

[43] W3C. XML path language (XPath) 1.0, 1999. `http://www.w3.org/TR/xpath`.

[44] W3C. XML path language (XPath) 2.0, 2007. `http://www.w3.org/TR/xpath20`.