XPath Query Containment[‡]

Thomas Schwentick[§]

1 Introduction

Consider an XML publish-subscribe scenario with hundreds of subscribers and tens of thousands of XML documents to be delivered per day. Subscribers specify the documents in which they are interested in by means of XPath [8] expressions. If an expression matches a (part of a) document it is delivered to the subscriber. Naturally, it is desired that the decision to which subscriber a document must be sent should be taken quickly. Although the test whether a single XPath expression matches can be done in polynomial time, it is not efficient to test every such expression for every document. Fortunately, there is a partial order on expressions, i.e., for some expressions p, q it might hold that whenever a document matches p it also matches q (denoted $p \subseteq_0 q$). If we already know that a document matches p, we do not need to test q anymore, as it matches automatically. Correspondingly, if we know that q does not match then p will not match either. Hence, the inclusion structure of the XPath expressions should be computed in advance to decrease online computation time. This leads to the algorithmic problem of *XPath Query Containment*, i.e., checking whether $p \subseteq_0 q$ (for a different, index-based approach see, e.g., [6]).

The main idea of this article is to describe some of the main algorithmic techniques that have been proposed for XPath Query Containment. These techniques are described in Section 5. Before that, in Sections 2 and 3 the basic definitions on XPath and the Query Containment Problem are given and in Section 4 there is an overview of complexity results for the problem. Finally, in Section 6 a couple of related questions are discussed. Because of space limitations, the examples in this survey are neither practical nor entertaining but as small as possible, using only tags like `<a>, <b>`, etc.

There are more reasons to study the XPath containment problem than the scenario mentioned above. As XPath occurs as a sublanguage in other XML languages (XQuery, XSLT, XLink, XPointer, XML Schema,...) the problem of XPath query containment and the closely related questions of equivalence and minimization are fundamental for query optimization.

Further, as XPath is used to define keys in XML Schema and other constraints can be specified by using fragments of XPath, understanding the key and constraint implication problem requires an understanding of the XPath query containment problem [4].

Compared to the classical containment problem for relational conjunctive queries, the problem is on the one hand easier, as the structures are trees rather than arbitrary relational structures, but on the other hand much harder, as the queries might involve recursion (e.g., by navigation along the `descendant`-axis).

2 XPath

2.1 Data model

We model XML documents as rooted trees with labels from an infinite (unranked) alphabet Σ. The symbols from Σ correspond to XML tags. Every node corresponds to an element The root of the tree corresponds to the root element of the document and is

[‡] Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

[§] Philipps-Universität Marburg, FB Mathematik und Informatik, 35032 Marburg, Germany, tick@informatik.uni-marburg.de denoted by root. Subelements are modelled by children. We refer to such trees as *XML trees.* Figure 1 displays a simple example document and its corresponding XML tree t_0.

```
<a> <c></c>
    <c> <b> </b>
    </c>
</a>
<a> <b> </b>
    <b>  <d> </d>
         <c> <c> </c>
         </c>
    </b>
</a>
```

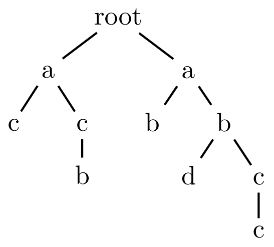


Figure 1: XML document and corresponding tree.

2.2 Syntax of XPath

In this article we only consider a fragment of XPath. For our purposes, a *step expression* s is of the form `axis` '::' `expr pred`*, where `axis` is one of the XPath axes `self`, `child`, `descendant`, `descendant-or-self`, `parent`, `ancestor`, `ancestor-or-self`, `following-sibling`, `preceding-sibling`. Further, `expr` is a *node test*, i.e., either a tag name or $*$. Finally `pred`* is a possibly empty sequence of items of the form $[p]$, where p is an expression as defined below.

The syntax for *expressions* p is given by $p ::= s \mid s$'/'$p \mid p$'|'$p \mid$ '/'p, where s is a step expression. An expression of the form $/p$ is called *absolute.* Other expressions are called *relative.*

An example expression p_0 is
`child`::a[`descendant`::d]/`child`::$*$/`descendant`::c.

2.3 Semantics of XPath

For each axis x and each XML tree t, we denote by $A_x(t)$ the set of all pairs (u, v) of nodes from t such that u and v are *in x-relation.* E.g., $A_{\texttt{child}}$ contains all pairs (u, v), for which v is the child of u.

Each expression p defines, on each XML tree t, a binary relation $R_p(t)$ as follows. For a step expression $s = x :: e\, p_1 \cdots p_k$, the relation $R_s(t)$ is defined as the set of pairs (u, v) of nodes, for which

- $(u, v) \in A_x(t)$,
- v matches e, i.e., $e = *$ or the label of v is e, and
- each set $R_{p_i}(t)$ contains at least one pair (v, w).

For an expression $p = s/q$, where s is a step expression and q is an expression, the relation $R_p(t)$ is $\{(u, v) \mid u, v, w \in t, (u, w) \in R_s(t), (w, v) \in R_q(t)\}$. Finally, $R_{p|q} = R_p(t) \cup R_q(t)$ and $R_{/p}$ is the set of all pairs (root, v) from $R_p(t)$.

Hence, the expression p_0 above defines the set of all pairs (u, v), where v is an element labelled c, which is a descendant of an element with arbitrary label, which in turn is a child of a child of u with label a. Furthermore, this child has to possess a descendant with label d. Evaluated on the tree t_0 we get the relation $R_{p_0} = \{(v_1, v_2), (v_1, v_3)\}$, where v_1 is the right child of the root node, v_3 is the rightmost leaf, labelled c, and v_2 is its parent.

2.4 Abbreviated syntax

For the most frequently used kinds of steps along the forward axes there exists an abbreviated syntax. We write

- $p/e/q$ instead of $p/$`child` $:: e/q$,
- $p//e/q$ instead of $p/$`descendant` $:: e/q$,
- $./p$ instead of `self` $:: */p$.

Hence, expression p_0 can also be written as a[.//d]/$*$//c.

2.5 Pattern trees

Expressions p which only use the `child` and `descendant` axes can be conveniently represented by their *pattern tree* $T(p)$. Each step of an expression corresponds to a node, which is a child of the node of the previous step. Steps along the `child` axis are indicated by single lines, steps along the `descendant` axis by double lines. We refer to edges of the first kind as *child edges* and to the others as *descendant* *edges.* A predicate expression of a step s gives rise to a subtree of the node corresponding to s. The node which corresponds to the last step of an expression (the *selection node*) is underlined to distinguish it from the leaves that are obtained from predicates. Hence, the tree depicted in Figure 2 is the pattern tree $T(p_0)$.

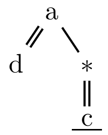


Figure 2: Pattern tree for p_0

It should be stressed that the order in which the children of a node are depicted does not carry any meaning. In particular, this order does not need to be matched in the document.

2.6 XPath fragments

Work on XPath query containment has mainly focussed on the two most important axes, `child` and `descendant`. It even considered fragments, where disjunction, predicates $[q]$ and/or the wildcard $*$ are not allowed. We refer to such fragments by writing $\mathrm{XP}(L)$, where L is a list of the allowed components in abbreviated notation. E.g., the fragment, where only `child`, predicates[1] and wildcard are allowed is denoted by $\mathrm{XP}(/, [\,], *)$.

3 Containment

3.1 Simple containment

In this section we define the basic notions about XPath query containment.

As explained in Section 2 an XPath expression p defines a binary relation $R_p(t)$, for every XML tree t. The most general notion of containment to consider is therefore based on binary relations. We write $p \subseteq_2 q$ if $R_p(t) \subseteq R_q(t)$, for every XML tree t.

An alternative notion of containment only considers whether nodes match relative to the root of the tree. Here, we interpret an expression p as absolute expression, defining the set $R_p^{\mathrm{root}}(t)$ of nodes v, for which $(\mathrm{root}, v) \in R_p(t)$. We write $p \subseteq_1 q$ if $R_p^{\mathrm{root}}(t) \subseteq R_q^{\mathrm{root}}(t)$, for every XML tree t.

Finally, we define *Boolean containment* which only asks whether p and q match at all, relative to the root. We write $t \models p$, if $R_p^{\mathrm{root}}(t) \neq \emptyset$. If $t \models p$ implies $t \models q$, for every XML tree t, then we write $p \subseteq_0 q$.

It turns out that all three containment notions are strongly related. If only the `child` and `descendant` axes are allowed $\subseteq_2$ and $\subseteq_1$ are actually equivalent. If predicates are allowed then it is even sufficient to consider Boolean queries. Figure 3 shows how the tree pattern $T(p_0)$ can be modified into a new tree pattern $T(p_0')$ by adding a child to its selection node. It holds that $p \subseteq_1 q$ if and only $p' \subseteq_0 q'$ [19]. In the remainder of this article, we will only consider Boolean containment $\subseteq_0$. Therefore, in pattern trees we no longer distinguish a selection node.

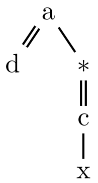


Figure 3: Pattern tree for p_0

It is easy to verify that $p_1 \subseteq_0 q_1$ holds for the expressions p_1 and q_1 underlying the tree patterns in Figure 4. We will encounter several ways to prove this fact in the next section.

3.2 Containment under constraints

In general, $p_2 \subseteq_0 q_2$ does not hold for the expressions $p_2 = /a/b//d$ and $q_2 = /a//c$. Nevertheless, it holds for documents like the example document above which conform to the following DTD d_2.

$$\begin{aligned} &\mathrm{root} \rightarrow a^* \\ &a \rightarrow b^* \mid c^* \\ &b \rightarrow d^+ c^+ \\ &c \rightarrow b?c? \end{aligned}$$

[1] If predicates are allowed the `self` axis can always be used in predicate expressions.

We say that $p \subseteq_0 q$ under DTD d, if $t \models p$ implies $t \models q$, for all trees t that are valid w.r.t. d.

The containment problem has been studied in the presence of DTDs and of several other types of constraints [26, 2, 22]. A very general class of constraints, *simple XPath Integrity Constraints (SXICs)* were introduced in [9]. They are reminiscent of embedded dependencies in relational databases (cf. [1]).

4 Complexity results

There are many complexity results for XPath containment, most of them with matching upper and lower bounds. Some upper bound techniques will be discussed in the next section. For space reasons we cannot touch techniques for lower bounds here. In Table 1 we list some of the main results, grouped by complexity. All complexities for coNP and the higher classes are tight, i.e., the problems are complete for the respective class.

In [19] the borderline between tractable and intractable fragments inside $XP(/, //, [\,], *)$ is studied. In particular, it is shown that the containment problem becomes tractable if the number of //-edges in the pattern q is bounded, but it remains coNP-complete if only the number of wildcards or predicate occurences is bounded.

Other axes. As already mentioned, most work concentrated on the forward axes. Some results from [9] concerning backward axes are mentioned in the table. In [24] it is shown that each XPath expression has an equivalent expression without backward axes. However, this expression might have exponential size.

5 Some algorithmic techniques

In this section a couple of techniques will be presented that were used to obtain upper bounds for various fragments of XPath. These techniques are based on canonical models, homomorphisms, the chase procedure, and on tree automata, respectively.

All these techniques use the simple but fundamental fact that $p \not\subseteq_0 q$ if and only if there is a *counter-*

PTIME	$XP(/, //, *)$ [21] $XP(/, [\,], *)$ (see [19]) $XP(/, //, [\,])$ [2], with fixed bounded SXICs [9] $XP(/, //)$ + DTDs [22] $XP[/, [\,]]$ + DTDs [22]
coNP	$XP(/, //, [\,], *)$ [19] $XP(/, //, [\,], *, \mid)$, $XP(/, \mid)$, $XP(//, \mid)$ [22] $XP(/, [\,])$ + DTDs [22] $XP(//, [\,])$ + DTDs [22]
Π_2^p	$XP(/, //, [\,], \mid)$ + existential variables + path equality + `ancestor-or-self` axis + fixed bounded SXICs [9] $XP(/, //, [\,], *, \mid)$ + existential variables + all backward axes + fixed bounded SXICs [9] $XP(/, //, [\,], \mid)$ + existential variables with inequality [22]
PSPACE	$XP(/, //, [\,], *, \mid)$ and $XP(/, //, \mid)$ if the alphabet is finite [22] $XP(/, //, [\,], *, \mid)$ + variables with XPath semantics [22]
EXPTIME	$XP(/, //, [\,], \mid)$ + existential variables + bounded SXICs [9] $XP(/, //, [\,], *, \mid)$ + DTDs [22] $XP(/, //, \mid)$ + DTDs [22] $XP(/, //, [\,], *)$ + DTDs [22]
Undecidable	$XP(/, //, [\,], \mid)$ + existential variables + unbounded SXICs [9] $XP(/, //, [\,], \mid)$ + existential variables + bounded SXICs + DTDs [9] $XP(/, //, [\,], *, \mid)$ + nodeset equality + simple DTDs [22] $XP(/, //, [\,], *, \mid)$ + existential variables with inequality[22]

Table 1: Complexity results for XPath containment.

example, i.e., a tree t such that $t \models p$ but $t \not\models q$.

5.1 The canonical model technique

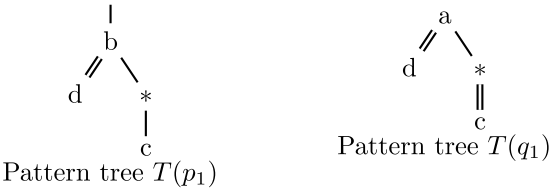
Unfortunately, the fundamental equivalence does not directly provide an algorithm for testing containment, as the set of possible trees t is infinite. Nevertheless, if for a fragment X it holds that $p \not\subseteq_0 q$ if and only if there is a counter-example t of polynomial size in p and q, then the containment test for X is in coNP (as the test for the complement of containment is then in NP). Accordingly, an exponential size bound for counter-example trees gives rise to a coNEXPTIME algorithm and so on.

The method of *canonical models*, introduced in [19, 20], tries to prune the search space by showing that there are always counter-examples (if any) with a similar shape as the pattern p. As an illustration of this technique we consider the following result [19].

Theorem 5.1 *Containment of* XP$(/, //, [\,], *)$ *expressions can be tested in* coNP.

The proof shows that $p \not\subseteq_0 q$ only holds if there is a counter-example t obtained from p as follows. Let z be a new symbol not occuring in p and q. Every $*$ in the pattern tree $T(p)$ is replaced by z. Every descendant edge is replaced by a chain of at most $m(q) + 1$ child edges with interior nodes labelled by z. Here, $m(q)$ is the maximum length of a chain in $T(q)$ consisting solely of child edges and $*$-nodes. It is clear that all these trees match p. It should be noted that the proof relies on the existence of a new symbol z.

As an example we consider the patterns p_1 and q_1 of Figure 4.



Pattern tree $T(p_1)$

Pattern tree $T(q_1)$

Figure 4: Patterns p_1 and q_1 with $p_1 \subseteq_0 q_1$.

Note that $m(q_1) = 1$. Therefore in order to verify $p_1 \subseteq_0 q_1$ it is sufficient to check that the trees listed in Figure 5 match q. To get an idea why the way $m(q)$ was defined is suitable, consider the patterns p_3 and q_3 in Figure 6. Replacing the descendant edges of p_3 by $*$-chains of length $3 = m(q_3) + 1$ results in a counter-example. Replacing them uniformly with shorter $*$-chains does not. In general though, it might be necessary to replace some edges by shorter paths.

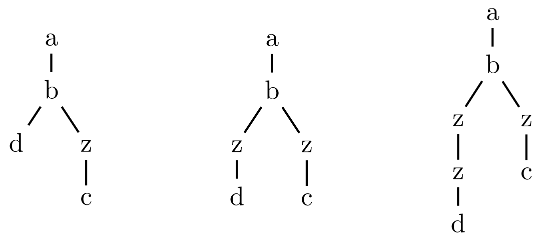
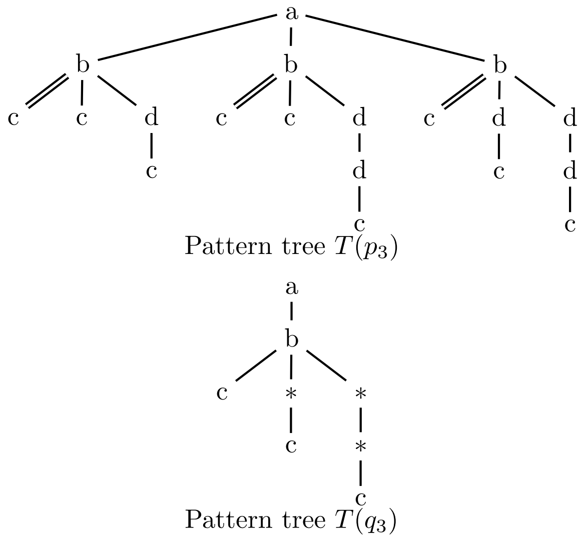


Figure 5: Trees to be tested to ensure $p_1 \subseteq_0 q_1$



Pattern tree $T(p_3)$

Pattern tree $T(q_3)$

Figure 6: Patterns p_3 and q_3 with $p_3 \not\subseteq_0 q_3$.

5.2 The homomorphism technique

There is a classical characterization result for conjunctive queries against relational databases. A query p is contained in a query q if and only if there is a homomorphism from q to p [7]. Similar characterizations can also be given for some XPath fragments. For simplicity we define homomorphisms only via pattern trees although they can also be directly defined for expressions. A homomorphism h from q to p maps each node of $T(q)$ to a node of $T(p)$ such that the following conditions hold.

(i) The root of $T(q)$ must be mapped to the root of $T(p)$.

(ii) If (u, v) is a child-edge of $T(q)$ then $(h(u), h(v))$ is a child-edge of $T(p)$.

(iii) If (u, v) is a descendant-edge of $T(q)$ then $h(v)$ has to be below $h(u)$ in $T(p)$.

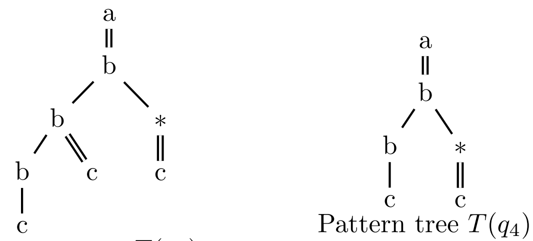
(iv) If u is labelled with $e \neq *$ then $h(u)$ also has to carry label e.

E.g., the mapping which maps the a-node of $T(q_1)$ to the a-node of $T(p_1)$, the d-node to the d-node, the c-node to the c-node and the $*$-node to the b-node is a homomorphism from $T(q_1)$ to $T(p_1)$. Note that, in general, a homomorphism does not need to be injective. If there exists a homomorphism from $T(q)$ to $T(p)$ then $p \subseteq_0 q$. For some fragments also 'only if' holds as the following result from [26, 2, 19] shows.

Theorem 5.2 *Let p, q be expressions from* XP$(/, //, [\,])$ *(or from* XP$(/, [\,], *)$*). Then $p \subseteq_0 q$ if and only if there is a homomorphism from $T(q)$ to $T(p)$.*

Unfortunately, even for XP$(/, //, [\,], *)$ the existence of a homomorphism is not *necessary* for containment, as exemplified by the patterns p_4 and q_4 in Figure 7. Although there seem to be two possible targets for the upper b-node of q_4, none of them really makes a homomorphism. Nevertheless, by reasoning on the possible lengths of a path in a tree t that is matched with the left descendent edge of p_4, it is easy to show that indeed $p_4 \subseteq_0 q_4$ holds.

A closer inspection shows that, at least for XP$(/, //, [\,], *)$ and its fragments, the homomorphism technique is essentially a special case of the canonical model technique, in which only one tree has to be tested. Let p and q be two expressions. We call a chain with two edges and an intermediate node labelled with a new symbol y a *special chain.* Let the tree $t(p)$ be obtained from $T(p)$ by replacing every `child`-edge with a special chain and each `descendant`-edge with a (normal) edge. Let q' be the expression corresponding to the pattern tree obtained from $T(q)$ by replacing every `child`-edge with a special chain. Then there is a homomorphism from $T(q)$ to $T(p)$ if and only if $t(p) \models q'$.



Pattern tree $T(p_4)$

Pattern tree $T(q_4)$

Figure 7: Patterns p_4 and q_4: $p_4 \subseteq_0 q_4$, but there is no corresponding homomorphism.

5.3 The automata technique

The basic idea of the next approach is very simple: compute the set C of *all* counter-examples and check whether C is empty. This looks impossible at first sight, as C might be infinite. But it turns out that C can often be represented by a finite device, a tree automaton. The containment question can then be solved by suitably combining the tree automata corresponding to the involved expressions (and constraints) and checking whether the resulting automaton accepts a non-empty set. For a gentle introduction to tree automata in the XML context see [23].

As a simple example for this technique, we consider XP$(/, //)$ in the presence of DTDs. We first describe the construction of a top-down automaton $\mathcal{A}_{p_2}$ for the expression $p_2 = a/b//d$. $\mathcal{A}_{p_2}$ shall accept a tree t if and only if $t \models p_2$. It traverses t from the root to the leaves. While doing so it nondeterministically selects one path from the root. On this path, it computes, for each node v, the furthest position in p_2 which is matched by the path from the root to v. Hence, on this path the states of $\mathcal{A}_{p_2}$ are basically positions of p_2, i.e., s_0, s_a, s_b, s_d. For nodes not on the path the automaton enters a dummy state s_*. The accepting states of $\mathcal{A}_{p_2}$ are s_c and s_*. The automaton accepts a tree t if there is a *run*, which has only accepting states at the leaves of t, hence, if there is a path matching p_2. Figure 8 repesents an accepting run of $\mathcal{A}_{p_2}$ on the tree t_0 of Figure 1. It is also easy to construct an automaton $\mathcal{A}_{\bar{q_2}}$ which accepts all trees which do *not* match $q_2 = /a//c$. This automaton uses only states s_0, s_a, s_c and accepts, if no leave gets the state s_c, i.e., if no path matches q_2. The general construction for expressions from XP$(/, //)$ is slightly more involved, but for each expression p one can get polynomial-size non-deterministic top-down automata $\mathcal{A}_p$ and $\mathcal{A}_{\bar{p}}$. Finally, from a DTD d, a non-deterministic top-down automaton $\mathcal{A}_d$ (of polynomial size in d) which accepts exactly the trees conforming to d can easily be constructed. By taking the product of $\mathcal{A}_p$, $\mathcal{A}_{\bar{q}}$ and $\mathcal{A}_d$ we get an automaton which accepts all counter-example trees t conforming to d. Whether this automaton accepts any tree can then be tested in polynomial time. Hence we get the following theorem from [22].

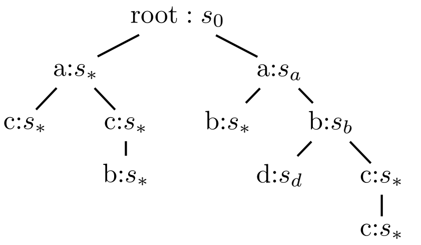


Figure 8: Accepting run of $\mathcal{A}_{p_2}$ on t_0.

Theorem 5.3 *Containment of* XP$(/, //)$ *expressions in the presence of DTDs can be tested in* PTIME.

It should be noted that, as the example p_2, q_2 and d_2 shows, the expressions p and q might be matched to different paths in a tree.

The automata technique can also be used for more expressive XPath fragments, involving, e.g., predicates and disjunction, but the corresponding automata can become larger. The basic idea is to associate with an expression p a bottom-up automaton which computes, for each node v of a tree t, the set of subexpressions of p that match the subtree of t rooted at v. As the states are now *sets of subpatterns* as opposed to single subpatterns (or positions in patterns), the automata have exponential size in worst case. Therefore this approach only gives an EXPTIME algorithm for XPath containment in the presence of DTDs. But, as was shown in [22], this is optimal.

Theorem 5.4 *Containment test of* XP$(/, //, [\,], *, |)$ *expressions in the presence of DTDs is complete for* EXPTIME.

Without DTD constraints the complexity is considerably smaller and, surprisingly, depends on whether the alphabet Σ is finite or infinite [22].

Theorem 5.5 *Containment test of* XP$(/, //, [\,], *, |)$ *is complete for* coNP. *It becomes complete for* PSPACE*, if the alphabet Σ is finite.*

Note that the proof idea of Theorem 5.1 above does not work in the case of a finite alphabet as there might not exist an unused symbol z.

5.4 The chase technique

In the relational case the homomorphism technique can be extended by the chase [17] to check query containment in the presence of integrity constraints. This approach can also be used for XML. In [9] the queries p and q are translated into relational queries p' and q'. The relational chase is then applied to p' with the relational translation of the given XML constraints together with additional general constraints that are needed to recover some of the information lost by the translation.

In other work the chase is directly applied to pattern trees [26, 2, 28]. We illustrate the basic idea using the simple example from above. The DTD d_2 implies that each b element has a d-child as well as a c-child. We can write this as the two constraints $b \to d$ and $b \to c$. Applying the chase procedure with these two constraints to the pattern tree of p_2 will add a d- and a c-child to the node b resulting in the

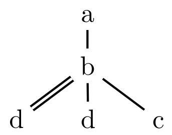


Figure 9: Pattern tree after chasing p_2

pattern shown in Figure 9. As there is an obvious homomorphism from $T(q_2)$ we get (again) that $p_2 \subseteq_0 q_2$ in the presence of d_2.

6 Related work

XPath equivalence. Of course, equivalence of XPath expressions can be reduced to containment. In [19] it is shown that, for forward axes and in the presence of predicates, these two problems are actually equivalent. Essentially, $p \subseteq_0 q$ if and only if p and $p[/q]$ are equivalent. A different approach to XPath equivalence via Datalog has been taken in [27].

XPath minimization. A related problem is the minimization of XPath queries, i.e., given an expression p to find a minimal equivalent expression p'. As pointed out in [10], minimization is possible in polynomial time for an XP-fragment, if containment for this fragment can be decided via homomorphisms and it always holds that p' is essentially a subpattern of p. In this way, PTIME-minimization was proved for XP$(/, [\,], *)$ [26] and for XP$(/, //, [\,])$ [2]. In [10] it is shown that XP$(/, //, [\,], *)$ also has the subpattern property. We already saw that it does not have the homomorphism property though, therefore the minimization problem is coNP-hard. Nevertheless, a PTIME-algorithm can be obtained for expressions from XP$(/, //, [\,], *)$ in which, for each node, all but one subtrees are linear. A general framework for optimization of XPath expressions has been studied in [16].

XPath evaluation. In [11] it was shown that XPath expressions can be evaluated in polynomial time (combined complexity), for a much larger fragment of XPath than the one considered here. This result was improved both in theory (precise complexity results) [12] and pratice [13]. A quick introduction to this topic can be found in [14].

Characterizing XPath. In [3] XPath fragments are characterized in terms of existential first-order logic. Furthermore closure properties and axiomatizability of many fragments are studied. An elegant characterization and an extension of XPath by so-called conditional axes can be found in [18]. The containment problem is also studied.

Containment for path queries on graphs. A lot of work has been done on containment for regular path queries in the more general framework of semistructured data, see e.g. [5, 15] and citations therein.

Tree pattern matching. An overview of algorithms for pattern matching in trees and graphs can be found in [25].

Acknowledgement The author thanks Frank Neven for many useful comments on an early draft of this article.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *The VLDB Journal*, 11(4):315–331, 2002.

[3] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *ICDT 2003*, page ?, 2003.

[4] Peter Buneman, Wenfei Fan, and Scott Weinstein. Interaction between path and type constraints. *ACM Trans. Comput. Logic*, 4(4):530–577, 2003.

[5] D. Calvanese, G. DeGiacomo, and M. Vardi. Decidable containment of recursive queries. In *Proc. Database Theory - ICDT '03, 9th International Conference*, pages 330–345, 2003.

[6] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. 28th International Conference on Very Large Data Bases (VLDB), Hongkong*, pages 235–244, 2002.

[7] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC 1977*, pages 77–90.

[8] World Wide Web Consortium. XML Path Language (XPath), Version 1.0. W3C Recommendation, 16 November 1999. `http://www.w3.org/TR/xpath`.

[9] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001).

[10] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *Proc. 29th International Conference on Very Large Data Bases (VLDB), Berlin*, pages 153–164, 2003.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of 28th Conf. on VLDB*, 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. 22nd Symposium on Principles of Database Systems (PODS), San Diego*, 2003.

[13] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *19th International Conference on Data Engineering (ICDE), Bangalore*, 2003.

[14] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath processing in a nutshell. *ACM SIGMOD Record*, 32(2):21–27, 2003.

[15] G. Grahne and Alex Thomo. Query containment and rewriting using views for regular path queries under constraints. In *Proc. 22nd Symposium on Principles of Database Systems (PODS), San Diego*, pages 111–122, 2003.

[16] April Kwong and Michael Gertz. Schema-based optimization of XPath expressions. 2002.

[17] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[18] Maarten Marx. XPath with conditional axes. To appear in EDBT 2004.

[19] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. 21th Symposium on Principles of Database Systems (PODS 2002)*, pages 65–76, 2002.

[20] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004. Full version of [19].

[21] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proc. Database Theory - ICDT '99, 7th International Conference*, pages 277–295, 1999.

[22] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proc. 9th Int. Conf. on Database Theory (ICDT), Siena*, pages 315–329, 2003.

[23] Frank Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[24] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking forward. In *EDBT Workshops 2002*, pages 109–127, 2002.

[25] Dennis Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proc. 21st ACM Symp. on Principles of Database Systems*, pages 39–52, 2002.

[26] P. T. Wood. Minimising simple XPath expressions. WebDB informal proceedings, 2001.

[27] P. T. Wood. On the equivalence of XML patterns. In Lloyd et al., editor, *Computational Logic – CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1152–1166. Springer, 2000.

[28] Peter Wood. Containment for XPath fragments under DTD constraints. In *Proc. Database Theory - ICDT '03, 9th International Conference*, pages 300–314, 2003.