Containment and Equivalence for an XPath Fragment

[Extended Abstract]

Gerome Miklau
Dept of Computer Science
University of Washington
Box 352350
Seattle, WA 98195
gerome@cs.washington.edu

Dan Suciu
Dept of Computer Science
University of Washington
Box 352350
Seattle, WA 98195
suciu@cs.washington.edu

ABSTRACT

XPath is a simple language for navigating an XML document and selecting a set of element nodes. XPath expressions are used to query XML data, describe key constraints, express transformations, and reference elements in remote documents. This paper studies the containment and equivalence problems for a fragment of the XPath query language, with applications in all these contexts.

In particular, we study a class of XPath queries that contain branching, label wildcards and can express descendant relationships between nodes. Prior work has shown that languages which combine any two of these three features have efficient containment algorithms. However, we show that for the combination of features, containment is coNP-complete. We provide a sound and complete EXPTIME algorithm for containment, and study parameterized PTIME special cases. While we identify two parameterized classes of queries for which containment can be decided efficiently, we also show that even with some bounded parameters, containment is coNP-complete. In response to these negative results, we describe a sound algorithm which is efficient for all queries, but may return false negatives in some cases.

1. INTRODUCTION

XPath is a simple language for navigating an XML tree and returning a set of answer nodes. XPath expressions are ubiquitous in XML applications. They are used in XQuery [4] to bind variables; in XML Schema [27] to define keys; in XLink [9] and XPointer [8] to reference elements in external documents; in XSLT as match expressions, and in content-based packet routing [21].

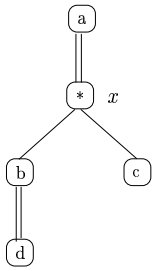


Figure 1: A simple tree pattern

Instances of the containment problem for XPath expressions occur in each these applications, and others. For example, inference of keys described by XPath expressions reduces to containment, and optimization of XPath expressions can be accomplished using an algorithm for containment.

The focus of this paper is the complexity of the containment problem for a simple fragment of XPath which is used frequently in practice. This fragment consists of: node tests, the child axis (/), the descendant axis (//), wildcards (*), and qualifiers (or branches, denoted [...]). Isolating the three most important features, we call this class of queries $XP^{\{[],*,//\}}$. It is a rather robust subset of XPath: many applications use only expressions in this fragment. Further restrictions, on the other hand, seem impractical since each of the constructs mentioned occur often. An expression in $XP^{\{[],*,//\}}$ is best represented as a *tree pattern*. For example, the expression a//*[b//d][c] is represented by the tree pattern pictured in Figure 1 where double-lines represent *descendant* edges, * is a label wildcard, and x marks the return node. Starting at the root, this pattern first checks if the root node is labeled a. If not it returns the empty set; otherwise it returns all its descendants that have both a b-child with a d-descendant, and a c-child.

For a given expression p and input tree t, we denote by $p(t)$ the set of nodes in t returned by the evaluation of p. Two expressions p, p' are contained (denoted $p \subseteq p'$) if $\forall t.p(t) \subseteq p'(t)$. Two expressions are equivalent if $p \subseteq p'$ and $p' \subseteq p$. We show in Section 2 that these two problems are mutually reducible, and focus our attention on the containment problem.

Our first result is that the containment problem for $\mathrm{XP}^{\{[],*,//\}}$ expressions is co-NP complete. This is rather surprising in light of prior results on the complexity of XPath containment, which have shown that for any combination of two of the constructs `*`, `//` and `[...]` the containment problem is in PTIME. In the absence of descendant edges ($\mathrm{XP}^{\{[],*\}}$), a PTIME containment algorithm follows from classic results on acyclic conjunctive queries [28]. Without label wildcards, the fragment $\mathrm{XP}^{\{[],//\}}$ was recently found to have a polynomial time containment algorithm [1]. And for $\mathrm{XP}^{\{*,//\}}$, patterns do not have branching, and are therefore closely related to a fragment of regular string expressions. A result in [17] shows that the containment problem for this fragment is also in PTIME. We show that containment is coNP-complete if branching, label wildcards, and descendant edges are considered together.

The high complexity of containment creates a new challenge: find practical algorithms for checking containment. We pursue two goals: (i) to find an efficient, sound algorithm, and show that it is complete in particular cases of practical interest; and (ii) to find a sound and complete algorithm and show that it is efficient in particular cases of practical interest. We answer (i) by describing a simple algorithm which always runs in PTIME and proving that it is complete when the containing query has no branching.

Our second class of results deal with problem (ii), which is more difficult than (i). It is not hard to describe a sound and complete algorithm that runs in exponential time, but the challenge consists in improving it to run in PTIME in non-trivial special cases. In particular we considered special cases inspired by the known PTIME results: (a) bound the number of `//`'s by a constant, (b) bound the number of `*`'s by a constant, and (c) bound the number of branches by a constant. We give a positive answer to (a): our exponential time algorithm runs in PTIME whenever the number of `//`'s in p is bounded by some number k (k will be the degree of the polynomial describing the running time). We give negative answers to (b) and (c). For (b), we show that the containment of $\mathrm{XP}^{\{[],*,//\}}$ expressions is coNP-complete even when p has no $*$'s and p' contains only two $*$'s. For (c), we show that the containment of $\mathrm{XP}^{\{[],*,//\}}$ expressions is coNP-complete even when p has five branches and p' has three branches. As our answer to problem (ii), we describe a containment algorithm based on alternating tree automata which runs in exponential time in general, but runs in PTIME in some special cases of practical interest, including when the number of `//`'s in p is restricted.

The organization of the paper is as follows. The next section contains the definition of tree patterns, their semantics and evaluation, and the relationship between tree patterns and XPath expressions. Section 3 is an overview of the main results of the paper. Sections 4 and 5 discuss ways of reasoning about containment: canonical models and pattern homomorphisms, respectively. Section 6 presents the proof of coNP-hardness for containment of $\mathrm{XP}^{\{[],*,//\}}$ patterns. Section 7 gives an overview of a sound and complete algorithm using tree automata. In Section 8 we provide a brief discussion of assorted issues including disjunction in patterns, connections to computation tree logic, and the special case when the alphabet is bounded. Section 9 discusses related work, and Section 10 concludes.

2. DEFINITIONS AND BACKGROUND

We model XML documents as trees over an infinite alphabet. A tree is an unordered, unranked finite structure with nodes labeled by symbols from Σ. (Although XML documents form ordered trees, our fragment of XPath ignores order, so we disregard it in our definition of trees.) The set of all trees is T_Σ. We study a fragment of XPath, denoted $\mathrm{XP}^{\{[],*,//\}}$, consisting of expressions given by the following grammar where n is an element name and . means the current node:

$$q \rightarrow q\,/\,q \quad | \quad q\,//\,q \quad | \quad q\,[\,q\,] \quad | \quad n \quad | \quad * \quad | \quad . \qquad (1)$$

Attributes and text values are handled similarly to elements and are omitted from the discussion. The "current node" is needed in contexts like $a/b/[.//c]$, and can otherwise be eliminated, e.g. $a/./b$ is equivalent to a/b.

Given an expression $q \in \mathrm{XP}^{\{[],*,//\}}$ and tree $t \in T_\Sigma$, $q(t)$ denotes a set of nodes in t. We adopt the formal semantics (omitted from this abstract) given in [24], fixing the root as context node. Two $\mathrm{XP}^{\{[],*,//\}}$ expressions are contained if their result sets are contained for every tree. Two expressions are equivalent if their result sets are equal.

Tree Patterns We also use an alternative, and slightly more general representation of expressions in $\mathrm{XP}^{\{[],*,//\}}$ as tree *patterns*. A pattern p is an unordered tree over alphabet $\Sigma \cup \{*\}$ with a distinguished subset of edges called *descendant edges*, and a k-tuple of nodes called the *result tuple*, for some $k \geq 0$. The arity of the result tuple, k, is called the *arity* of p, and the pattern is called *boolean* if k is zero. Descendant edges are shown in diagrams with double lines, and other edges are called *child edges*. As usual, for a node x, $\textsc{degree}(x)$ is the number of its children, and $\textsc{label}(x)$ is its label. For a pattern p, the nodes of p are $\textsc{nodes}(p)$, and its root is $\textsc{root}(p)$. The set of all tree patterns is denoted $\mathrm{P}^{\{[],*,//\}}$.

We define the subclasses $\mathrm{XP}^{\{[],*\}}$, $\mathrm{XP}^{\{[],//\}}$, $\mathrm{XP}^{\{*,//\}}$,

and $P^{\{[],*\}}$, $P^{\{[],//\}}$, $P^{\{*,//\}}$ by restricting to only two of the three features, label wildcard ($*$), descendant edge ($//$), and branching ($[]$). For example $P^{\{[],*\}}$ denotes the set of tree patterns including branching and label wildcards without descendant edges.

Given a tree pattern p with arity k and a tree $t \in T_\Sigma$, $p(t)$ denotes a k-ary relation on the nodes of t defined as follows. First, we define an *embedding* from p to t to be a function $e : \text{NODES}(p) \rightarrow \text{NODES}(t)$ which is root preserving, respects node labels, and respects edge relationships. Formally this means that (1) $e(\text{ROOT}(p)) = \text{ROOT}(t)$, (2) for each $x \in \text{NODES}(p)$, $\text{LABEL}(x) = *$ or $\text{LABEL}(x) = \text{LABEL}(e(x))$, and (3) for each $x, y \in \text{NODES}(p)$, if (x, y) is child edge in p then $(e(x), e(y))$ is an edge in t, and if (x, y) is a descendant edge in p then $e(y)$ is a proper descendant of $e(x)$.

An example of an embedding is pictured in Figure 2(a,b). Denoting $(x_1, x_2, \ldots\ x_k)$ the return nodes in p, we define:

$$p(t) = \{(e(x_1), \ldots, e(x_k)) \mid e \text{ is an embedding from } p \text{ to } t\}$$

An embedding need not be an injective function – there may be two nodes in the pattern mapped to the same node of the input tree. Even two distinct return nodes may be mapped to the same node under an embedding. Furthermore, sibling nodes in a pattern are not ordered so there are no order-based restrictions to embeddings.

Evaluation of a pattern p on a tree t is efficient. For each $\bar{v} \in (\text{NODES}(t))^k$ one can check in $O(|p||t|^2)$ time whether there exists an embedding that maps the result tuple to $\bar{v}$. The algorithm proceeds bottom-up on the nodes p and t building a table with one entry for each pair of nodes (x, y), $x \in t$ and $y \in p$: entry (x, y) contains true iff there exists an embedding from the subpattern rooted at y to the subtree rooted at x that is compatible with mapping the result tuple to $\bar{v}$. There are $|p||t|$ entries in the table and we need $O(|t|)$ steps to compute each entry, because of the presence of descendant edges in p.

Other Notions of Pattern Matching The study of tree pattern matching problems has a long history which has focused primarily on the problem of evaluation of patterns, not containment. Nevertheless, it is illuminating to consider the differences between the semantics of our patterns and other matching problems.

Two pattern matching problems are especially related to ours. The first, sometimes called classical tree pattern matching, involves a more restrictive embedding [13]. This pattern matching problem is equivalent to a boolean pattern without descendant edges if we require that an embedding respect the order of siblings in the pattern nodes. (It follows that the matching function would then be injective.) There is an obvious algorithm which solves this problem in $O(mn)$ time. Improving this bound was a long-time open problem, first solved in [15] to attain a bound of $O(nm^{0.75}polylog(m))$. To our knowledge, the best algorithm is $O(nlog^3m)$[7].

The second related problem was defined in [14] as *unordered tree inclusion*. The simplest statement of the problem is: given a pattern and input tree, can the pattern tree be obtained from the input tree by node deletions. It turns out that this problem is equivalent to evaluating a pattern in our formalism where all edges are descendant edges, but with a different definition of embedding. The embedding v is required to be injective, and in addition, x and y are ancestors in p if and only if $v(x)$ and $v(y)$ are ancestors in t. This is a stronger requirement than ours and prevents, for instance, two siblings from mapping over a path in t that shares an edge. These subtle differences result in an increased evaluation complexity and it is shown in [14] that unordered tree inclusion is NP-complete.

From XPath to Tree Patterns Every expression in $XP^{\{[],*,//\}}$ can be translated into a tree pattern of arity one with the same semantics, and, conversely, each pattern of arity one can be translated into an $XP^{\{[],*,//\}}$ expression. Figure 2(b) illustrates a pattern in $P^{\{[],*,//\}}$, with arity one, which is equivalent to the $XP^{\{[],*,//\}}$ expression `a[a]//*[b]//c`. The containment problems for $XP^{\{[],*,//\}}$ and for $P^{\{[],*,//\}}$ are thus equivalent. While not present in XPath, patterns of higher arity are of great interest to us because they capture multiple variable bindings, which occur for example in the `FOR` clauses of XQuery [4]. For the study of containment, however, arity is not an important consideration, as we explain next.

Boolean Patterns In the case of a pattern with arity zero, $p(t)$ evaluates to the empty-tuple if there exists an embedding from p to t. Otherwise, $p(t)$ is the empty-set. We therefore view such patterns as *boolean*, and say $p(t)$ is true if an embedding exists and false otherwise. For boolean patterns, containment reduces to implication: $p \subseteq p'$ if and only if $\forall t.p(t) \rightarrow p'(t)$.

For the purpose of the containment problem, it suffices to limit our discussion to boolean patterns, as the next proposition claims:

Proposition 1. *There is a translation of k-ary patterns over alphabet Σ, to boolean patterns over alphabet $\Sigma \cup \{x_1, x_2, \ldots\ x_k\}$ such that for any k-ary patterns p, p', and their translations $\bar{p}, \bar{p}'$, $p \subseteq p'$ if and only if $\bar{p} \subseteq \bar{p}'$.*

Figure 3 shows a pattern p of arity 3, and the boolean pattern $\bar{p}$ which is its translation. The full version [16] of this abstract includes the formal proof of Proposition 1.

Containment and Equivalence The containment and equivalence problems are mutually reducible in polynomial time. Equivalence is simply two-way con-

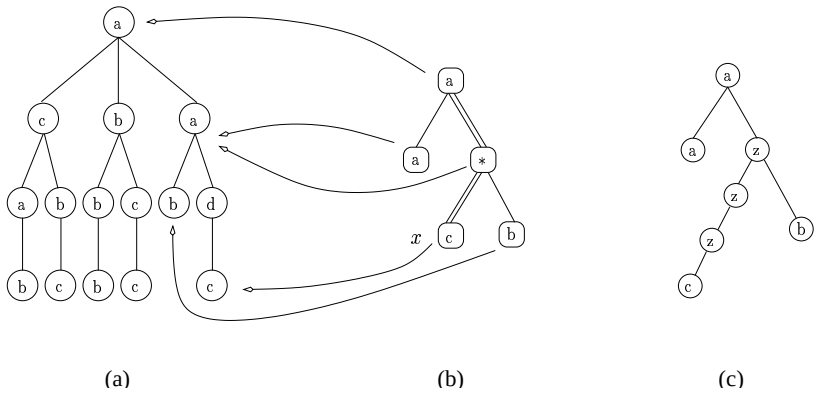


Figure 2: (a) tree instance t; (b) pattern p and an embedding from p to t; (c) canonical model in $m^z(p)$ (described in Section 4).

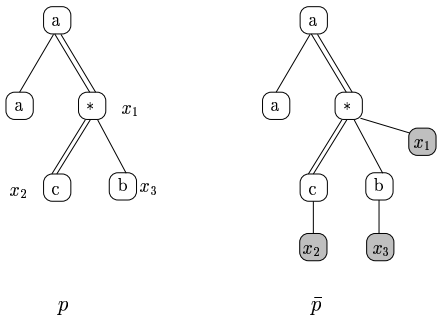


Figure 3: A tree pattern p of arity 3, with result nodes marked, and its translation to a boolean pattern $\bar{p}$, used in Proposition 1.

tainment. In addition, given two boolean patterns p and p', and an algorithm for equivalence, we can decide containment. First form a new tree pattern P from p and p' by fusing their roots. If containment is to hold, either $\text{LABEL}(\text{ROOT}(p)) = \text{LABEL}(\text{ROOT}(p'))$ or for some $a \in \Sigma$, $\text{LABEL}(\text{ROOT}(p)) = a$ while $\text{LABEL}(\text{ROOT}(p')) = *$. In the former case, $\text{LABEL}(\text{ROOT}(P))$ is their common label; in the latter $\text{LABEL}(\text{ROOT}(P)) = a$. Pattern P is a boolean pattern such that $P(t)$ is true if and only if $p(t) \wedge p'(t)$ is true, for any input tree t. Then it follows that $p \subseteq p'$ if and only if p is equivalent to P. We discuss only containment in the remainder of the paper.

3. MAIN RESULTS

This section summarizes the main results of the paper. Theorem 1 states that the containment problem for $\text{P}^{\{[],*,//\}}$ is coNP-complete. Although we present other coNP-completeness results below that imply this result, the proof of Theorem 1 is provided in Section 6 because it offers the best intuition into the complexity of containment.

Theorem 1 (coNP completeness). *Given two patterns $p \in P^{\{[],//\}}$ and $p' \in P^{\{[],*,//\}}$, deciding $p \subseteq p'$ is coNP-complete.*

Searching for practical solutions, we investigate two directions: (1) find an efficient containment algorithm that is always sound, but not necessarily complete, and (2) find a sound and complete algorithm that is in EXPTIME in general, but is provably efficient in special cases. We will give an algorithm answering (1) in Section 5, and summarize its properties here:

Proposition 2 (Incomplete algorithm). *There exists a sound but incomplete algorithm that, given patterns $p, p' \in P^{\{[],*,//\}}$ decides $p \subseteq p'$ in $O(|p|^2|p'|)$ time. If $p' \in P^{\{*,//\}}$ then the algorithm is complete.*

The answer to (2) is more complex. Recall that for the restricted classes $\text{P}^{\{[],//\}}, \text{P}^{\{[],*\}}, \text{P}^{\{*,//\}}$ there are PTIME algorithms. Ideally we would like to extend these to run in polynomial time in these cases, to degrade gracefully as we introduce a few instances of the missing feature, and to become exponential for $\text{P}^{\{[],*,//\}}$. For example, the algorithm should run in PTIME when we impose some bound on, say, the number of $*$'s. Unfortunately, for two of the three features in $\text{P}^{\{[],*,//\}}$ this is not possible, as shown in the next two theorems. Technically, these are the most difficult results in the paper; their proofs are included in [16].

Theorem 2 (coNP - bounded wildcard). *Given two patterns $p \in P^{\{[],//\}}$ and $p' \in P^{\{[],*,//\}}$, where p' has at most 2 label wildcards, deciding $p \subseteq p'$ is coNP-complete.*

Theorem 3 (coNP - bounded branching). *Given two patterns $p \in P^{\{[],*,//\}}$ and $p' \in P^{\{[],*,//\}}$, where p has at most 5 branches and p' has at most 3 branches, deciding $p \subseteq p'$ is coNP-complete.*

Given these additional lower bounds, we give the best possible answer to (2) by providing a sound and complete EXPTIME algorithm in Section 7. The main properties of that algorithm are summarized in the next two theorems. Let d be the number of descendant edges

in p, and let w' be one plus the longest number of $*$-nodes in p' forming a path consisting exclusively of child edges. (For example if p' =a/*/*//*/*/b/* then $w' = 2 + 1 = 3$.)

Theorem 4 (Parameterized PTIME). *For patterns $p, p' \in P^{\{[],*,//\}}$, $p \subseteq p'$ can be decided in $O(\,|p||p'|\,(w'+1)^{(d+1)})$ time.*

Further, let d' and s' be the number of descendant edges and label wildcards in p', respectively, and for $a \in \Sigma$, define $\text{degree}_a(p')$ to be the sum of the degrees of all a-labeled nodes in p'. Also, let r be the maximum degree of a node in p.

Theorem 5 (Parameterized PTIME). *For patterns $p, p' \in P^{\{[],*,//\}}$, and $c' = max_{a\in\Sigma}\{\text{degree}_a(p')\}$, then $p \subseteq p'$ can be decided in $O(\,|p|\,2^{(s'+d'+c')r})$ time.*

Importantly, the time bounds of Theorems 4 and 5 are both achieved by the *same* algorithm, described in Section 7. Theorem 4 says that this algorithm runs in PTIME for $P^{\{[],*\}}$ and degrades gracefully when descendant edges are introduced. We know from Theorem 2 that something similar is not possible for $*$'s, but Theorem 5 comes close by imposing extra limitations in addition to those on the number of $*$'s.

4. CANONICAL MODELS

One way to reason about containment is by way of canonical models. To begin with, the *models* of a boolean pattern p are the trees of T_Σ on which p evaluates to true: $Mod(p) = \{t \in T_\Sigma \mid p(t) \text{ is true}\}$. Containment of boolean patterns can be re-stated in terms of models:

Proposition 3. *For any (boolean) tree patterns p and p', $p \subseteq p'$ if and only if $Mod(p) \subseteq Mod(p')$.*

The set of *canonical models* of p, denoted $m(p)$, is a subset of $Mod(p)$ consisting of trees with the same shape as p. We first define a canonical model for a pattern p without descendant edges. In this case, a canonical model is obtained from p by replacing each $*$ with some symbol from Σ. The set of all such canonical models, denoted $m(p)$, is an infinite set since Σ is infinite. Suppose now that p has d descendant edges, $r_1, r_2, \ldots\ r_d$, and let $u_1 \geq 0, \ldots, u_d \geq 0$ be d integers. We define $p[u_1, u_2, \ldots\ u_d]$ to be the pattern (without descendant edges) obtained by replacing each edge r_i with a chain of u_i new $*$-labeled nodes. Then, define:

$$m(p) = \bigcup\{m(p[u_1, \ldots, u_d]) \mid u_1 \geq 0, \ldots, u_d \geq 0\} \quad (2)$$

Figure 2(c) shows a canonical model of p in Figure 2(b) (which is a canonical model of $p[0, 2]$). We prove the following criterion for containment in the full version of this paper:

Proposition 4. *For any tree patterns p and p', $p \subseteq p'$ if and only if $m(p) \subseteq Mod(p')$*

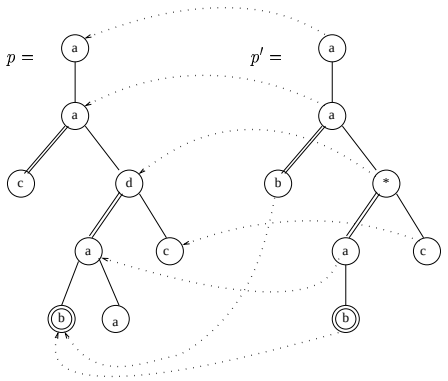


Figure 4: Two tree patterns p, p' and a homomorphism from p' to p, proving $p \subseteq p'$.

This does not, however, imply a decision procedure for containment because $m(p)$ is infinite: we have infinitely many choices for each u_i, and we have infinitely many choices from Σ for each $*$ replacement. To address the latter we fix a symbol $z \in \Sigma$ that does not occur[1] in p or p', and, for any tree pattern q without descendant edges denote $m^z(q)$ the canonical model obtained by replacing each $*$ is replaced with z. To address the former, we impose some upper bound n on the u_i's in Equation (2), defining:

$$m^z_n(p) = \bigcup\{m^z(p[u_1, \ldots, u_d]) \mid u_1 \leq n, \ldots, u_d \leq n\}$$

Recall from Section 3 that we defined w' to be one plus the length of longest sequence of $*$'s connected by child edges in p'.

Proposition 5. *For any tree patterns p and p', the following are equivalent: (1) $p \subseteq p'$, (2) $m_{w'}(p) \subseteq Mod(p')$, (3) $m^z_{w'}(p) \subseteq Mod(p')$.*

The proof is provided in [16]. The proposition implies that containment for $P^{\{[],*,//\}}$ is in coNP, as required for Theorems 1,2, and 3: indeed, in order to check $p \not\subseteq p'$ it suffices to guess d numbers $u_1, \ldots, u_d$, construct the tree $t = m^z(p[u_1, \ldots, u_z])$, then check that $p'(t)$ is false. The proposition also gives an upper bound on the containment problem slightly worse than that stated in Theorem 4. It suffices to enumerate all canonical models $t \in m^z_{w'}(p)$ and check $p'(t)$ in time $O(|t|^2\,|p'|)$. There are $\leq (w'+1)^d$ canonical models in $m^z_{w'}(p)$, and each has a size bounded by $|t| \leq (w'+1)|p|$, resulting in an evaluation complexity of $O(|p|^2\,|p'|(w'+1)^{(d+2)})$. The algorithm in Sec. 7 improves this upper bound.

5. PATTERN HOMOMORPHISMS

We can also reason about containment using homomorphisms between patterns, which are the basis of algorithms for the PTIME fragments

[1]See Section 8 for comments on the case when Σ is finite.

$P^{\{*,//\}}$, $P^{\{[],*\}}$, $P^{\{[],//\}}$, and for the incomplete algorithm in Proposition 2. When, for a given class of queries, the existence of a homomorphism is a necessary and sufficient condition of containment, there is generally a polynomial time decision procedure. Therefore, we also explain in this section where the homomorphism technique breaks down for containment of $P^{\{[],*,//\}}$ patterns.

A homomorphism is a function $h : \text{NODES}(p') \rightarrow \text{NODES}(p)$ between two patterns p' and p. The conditions on a homomorphism are extensions of those for an embedding (Section 2). A homomorphism h must be root-preserving, respect node labels, and obey edge constraints. More precisely: (1) $h(\text{ROOT}(p')) = \text{ROOT}(p)$, (2) for each $x \in \text{NODES}(p')$, $\text{LABEL}(x) = *$ or $\text{LABEL}(x) = \text{LABEL}(h(x))$, and (3) for each $x, y \in \text{NODES}(p)$, if (x, y) is a child edge in p' then $(h(x), h(y))$ must be a child edge in p; if (x, y) is a descendant edge in p' then $(h(x), h(y))$ must be a path in p of length ≥ 0, which may include child edges and/or descendant edges. We define the length of a path to be the number of intermediate nodes: that is, for a path of length 0, $h(x)$ is the parent of $h(y)$.

The existence of a homomorphism is always a sufficient condition for containment because if e is an embedding from p to some tree t, then $e \circ h$ is also an embedding from p' to t (so that $Mod(p) \subseteq Mod(p')$). Moreover one can check in $O(|p|^2|p'|)$ whether there exists a homomorphism from p' to p,[2] which gives a sufficient condition for $p \subseteq p'$ verifiable in polynomial time. Figure 4 contains two tree patterns p, p' and a homomorphism that implies containment.

In the case of $P^{\{[],*\}}$ and $P^{\{[],//\}}$ the existence of a homomorphism is also a necessary condition for containment. This follows from the fact that for any pattern p in either $P^{\{[],*\}}$ or $P^{\{[],//\}}$ there is a single canonical model t such that, for any p', $p \subseteq p'$ iff $p'(t)$ is true. For patterns in $P^{\{[],*\}}$, this is immediate since $m^z(p)$, contains a single tree. In the case of $P^{\{[],//\}}$, it is sufficient to consider the unique canonical model of $p[u_1, u_2, \ldots\ u_d]$ for each $u_i = 1$. In either case, if $p \subseteq p'$, then there exists an embedding $e : p' \rightarrow t$, and this implies a homomorphism $p' \rightarrow p$. This technique forms the basis of the PTIME algorithms for $P^{\{[],*\}}$ and $P^{\{[],//\}}$ in [28, 26, 1].

However, when both * and // are allowed in tree patterns, the existence of a homomorphism is no longer a necessary condition. The tree patterns in Figure 5 (a) corresponding to p=a/*//b, p'=a//*/b illustrate this. Although p, p' are equivalent, there is no homomorphism from p' to p because there is no destination for the wildcard in p'.

In the case of linear queries in $P^{\{*,//\}}$, [17] shows that containment can still be decided in PTIME by combining adjacent //'s and *'s in p' into single units, then searching for a homomorphism. In our example this removes the problematic middle node, as shown in Figure 5 (b), making a homomorphism possible. In general, we replace each // in p' with $//^0$, then apply the following rules repeatedly:

$$\begin{aligned} //^m * / &\rightarrow //^{m+1} \\ / * //^n &\rightarrow //^{n+1} \\ //^m * //^n &\rightarrow //^{m+n+1} \end{aligned} \quad (3)$$

For example, p' = a//*/*/b/*/c//d rewrites to a$//^2$b/*/c$//^0$d. Given this new representation, the definition of a homomorphism $h : p' \rightarrow p$ is modified to require that an edge $//^n$ be mapped to a path of length $\geq n$. The result stated in [17] is that $p \subseteq p'$ iff there exists a homomorphism from p' to p. The full version of this abstract contains and expands the proof of that result.

It is easy to extend this notion of a homomorphism to $P^{\{[],*,//\}}$, which results in the algorithm in Figure 6(a). The algorithm is clearly sound, since whenever there exists a homomorphism from p' to p, then $p \subseteq p'$. Its running time is $O(|p|^2|p'|)$ (since Step 1 can be done in linear time), thus proving half of Prop 2. The second half says that this algorithm is complete when p' is linear: we prove this in [16].

But, in general, the algorithm in Fig. 6 (a) is incomplete. Consider the two patterns p, p' in Figure 6 (b); here there exists no homomorphism from p to p', but $p \subseteq p'$. To show this we have to reason by cases. Let t be a canonical model of p and consider the middle branch in t: the $c - d$ paths can be either of length 0, or of length ≥ 0. In the first case we define the embedding $p' \rightarrow t$ as shown by the top choice in the figure: to the 2nd and 3rd branch in t. In the second case we define the embedding $p' \rightarrow t$ as shown by the bottom choice: to the 1st and 2nd branch. Thus, to check containment we need to "reason by cases".

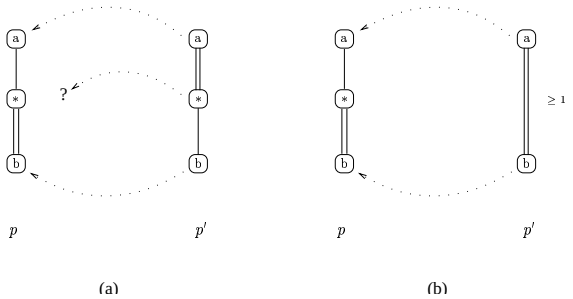
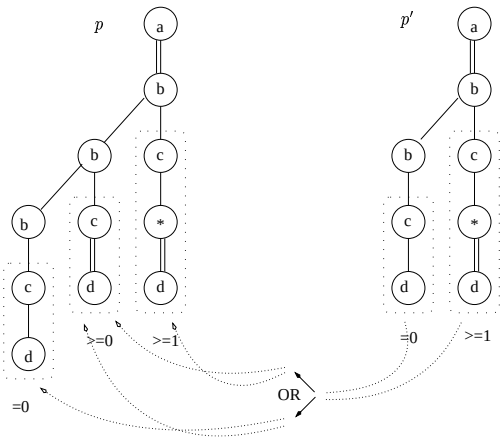


Figure 5: (a) Two equivalent queries p, p' with no homomorphism from p' to p; (b) same queries represented differently, and a homomorphism between them.

[2] This time bound generalizes the time bound for evaluation of a boolean pattern on a tree (Section 2).

Algorithm CheckContainment I
Input: p, p'
Output: true if $p \subseteq p'$
Method
Step 1 Apply re-writings (3) to p'
Step 2 Find homomorphism from p' to p
if found **then** return true
else return false

(a)



(b)

Figure 6: A sound, but incomplete algorithm (a), and an example where it is incomplete (b).

6. PROOF OF CONP-HARDNESS

In this section we complete the proof of Theorem 1, that containment is coNP-complete for tree patterns in $P^{\{[],*,//\}}$, by reducing the complement of satisfiability to the containment problem. Recall that we argued in Section 4 that containment of $P^{\{[],*,//\}}$ patterns is in coNP.

To simplify the later construction, we prove a preliminary result. Define containment of a boolean pattern p in a union of patterns as follows: $p \subseteq p_1 \cup \cdots \cup p_k$ holds if, for all trees t, $p(t) \rightarrow p_1(t) \vee p_2(t) \vee \cdots \vee p_k(t)$.

Lemma 1. *Given patterns p and $p_1, p_2, \ldots\ p_k$ in $P^{\{[],*,//\}}$, there exist patterns q, q' in $P^{\{[],*,//\}}$ such that $p \subseteq p_1 \cup \cdots \cup p_k$ if and only if $q \subseteq q'$. Furthermore, q and q' are polynomial in the sizes of $p, p_1, p_2, \ldots\ p_k$, and q and q' have no more label wildcards than those present in $p, p_1, p_2, \ldots\ p_k$.*

Proof We assume w.l.o.g. that all patterns $p, p_1, \ldots, p_k$ have the roots labeled with the same symbol $a \in \Sigma$: if not, we transform the patterns into $p', p_1', \ldots, p_k'$ by adding another root node labeled a to each pattern, and we have $p \subseteq p_1 \cup \ldots \cup p_k$ iff $p' \subseteq p_1' \cup \ldots \cup p_k'$.

The construction of q and q' is shown in Figure 7. Pattern q' consists of a spine of the k subtrees $p_1, p_2, \ldots\ p_k$ connected to a root node by a descendant edge. Pattern p consists of a longer spine, at the center of which sits a subtree equal to pattern p. The pattern subtree V, which is repeated in q, has no wildcards and no descendant edges, and is chosen so that for any j, $V \subseteq p_j$. This can be achieved by fusing the (common) roots of the p_i subtrees (this is possible because their roots have the same label), and replacing all label wildcards in the p_i with an arbitrary letter, and all descendant edges with child edges.

With this construction, the canonical models of q are completely determined by a choice of canonical model for q's subtree p: for each $t \in m(q)$ we denote $t_p \in m(p)$ the subtree corresponding to p (see Fig. 7).

We assume first that $p \subseteq p_1 \cup \cdots \cup p_k$, and show that for every $t \in m(q)$, we have $q'(t)$ is true, which proves $q \subseteq q'$. Given $t \in m(q)$, clearly $p(t_p)$ is true, hence $p_i(t_p)$ is true, for some $i = 1, \ldots, k$. We prove that $q'(t)$ is true by constructing an embedding $e : q' \rightarrow t$ with the following properties. First, e maps the subpattern p_i to t_p: this is possible since $p_i(t_p)$ is true. Second, e maps every other p_j to a corresponding V: this is possible since $V \subseteq p_j$, and there enough V's both above t_p and below t_p (namely $k-1$ both above and below). Finally, it maps the root of q' to the root of t.

Conversely, we assume $q \subseteq q'$ and show that $\forall t_p \in m(p)$, $p_1(t_p) \vee \cdots \vee p_k(t_p)$: it is easy to check that the latter implies $p \subseteq p_1 \cup \cdots \cup p_k$. Let $t_p \in m(p)$, and denote with t its extension to a tree $t \in m(q)$, by adding the spine and $k-1$ copies of V above and below t_p in an obvious way. We have $q'(t)$ is true, hence there exists an embedding $e : q' \rightarrow t$. This embedding must map the spine in q' to the spine in t. Let x be the spine node in t that is right above t_p. At least one spine node in q' must be mapped to x: this is because there are only $k-1$ spine nodes above x, only $k-1$ spine nodes below, and the spine in q' has k nodes and no descendant edges: hence e cannot avoid mapping some node y into x. Let p_i be the pattern below y: it follows that $p_i(t_p)$ is true. □

We now prove that containment is coNP hard. Let ψ be a 3-CNF formula with n propositional variables $y_1, y_2, \ldots\ y_n$, and k clauses $c_1, c_2, \ldots\ c_k$. We construct patterns $A, C_1, \ldots, C_k$, pictured in Figure 8, such that ψ is not satisfiable iff $A \subseteq C_1 \cup \cdots \cup C_k$. Tree pattern A is constructed so that its canonical models, $m(A)$, encode truth assignments to the n variables of ψ. Tree

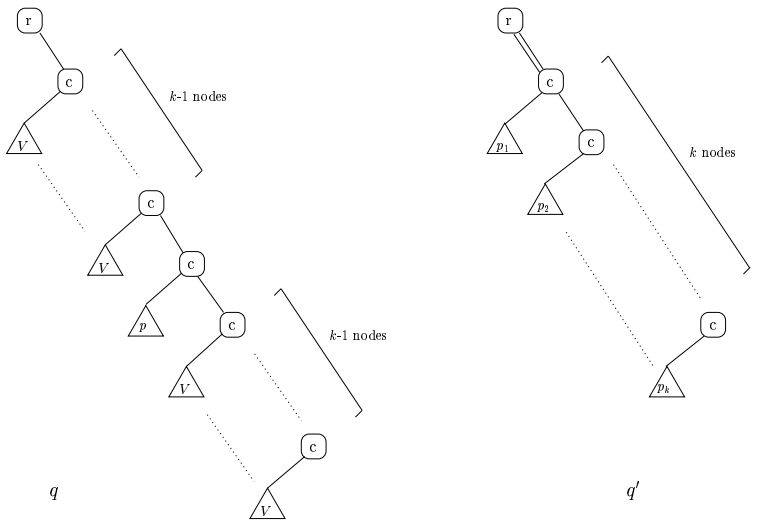


Figure 7: Patterns q and q' from Lemma 1, constructed from $p, p_1, p_2, \ldots\ p_k$ so that $q \subseteq q'$ if and only if $p \subseteq p_1 \cup \cdots \cup p_k$

pattern C_i is constructed so that the following property holds:

(*) For every $t \in m(A)$, $C_i(t)$ is true iff the truth assignment encoded by t makes the clause c_i false.

Property (*) is sufficient to prove coNP hardness because of the following equivalences and of Lemma 1: $(A \subseteq C_1 \cup \cdots \cup C_k) \iff$ (for every $t \in m(A)$ there exists i s.t. $C_i(t)$ is true) $\iff$ (for every truth assignment there exists i s.t. c_i is false under that assignment) $\iff$ (ψ is not satisfiable). In the remainder of the proof we show how to construct $A, C_1, \ldots, C_k$ such as to satisfy property (*).

Pattern A has one branch for each variable y_i in ψ, and we use a unique element of the alphabet a_i for each variable. Consider a canonical model $t \in m(Y_i)$ (see Fig. 8). If t has one b-child under its a_i-labeled root, then we consider this canonical model true with respect to y_i. If t contains one or more added nodes, then we say t makes y_i false. Under this interpretation of true and false, each canonical model of A corresponds to a truth assignment to the variables of ψ, and all truth assignments are represented by some canonical model.

Next we define a tree pattern C_i for each clause of ψ. We only illustrate on an example: the general case follows immediately. Suppose clause $c_i = (\neg y_j \vee y_k \vee \neg y_l)$. Pattern tree C_i is pictured in Figure 8, and consists of a root node with three subtrees, one for each term appearing in c_i. A variable like y_j that appears negated in c_i results in a branch consisting of subtree $T(y_j)$. Variable y_k, which does not appear negated in c_i, results in a branch containing $F(y_k)$. This construction enforces property (*).

7. ALGORITHM FOR CONTAINMENT

We describe here a sound and complete algorithm for checking containment of two tree patterns, whose running time has upper bounds given by both Theorem 4 and Theorem 5. The algorithm reduces the tree pattern containment problem to the containment problem for regular tree languages. Tree patterns are defined in terms of unranked, unordered trees, while tree automata compute on ranked, ordered trees: part of the difficulty in designing the algorithm consists of translating between these two formalisms, a process we call *ranking*. Also, several non-obvious techniques are needed in order to achieve the upper bounds of Theorems 4 and 5.

Background on Tree Automata Tree automata are defined on *ranked* trees. An alphabet of rank r is a finite set Ω partitioned into $\Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_r$; a symbol $a \in \Omega_k$ is said to have rank $k = 0, \ldots, r$. A *ranked tree* is an ordered tree s.t. for every node x, if $\text{LABEL}(x) \in \Omega_k$ then x has exactly k children. We denote with $T_\Omega^<$ the set of ranked trees. A *finite tree automaton*, FTA, A, has a set of states, $\text{STATES}(A)$, and transitions of the form $(q_1, \ldots, q_k; a) \rightarrow q$, where $q_1, \ldots, q_k, q \in \text{STATES}(A)$ and $a \in \Omega_k$. The FTA is deterministic if whenever $(q_1, \ldots, q_k; a) \rightarrow q$ and $(q_1, \ldots, q_k; a) \rightarrow q'$ then $q = q'$: DFTA abbreviates a *deterministic finite tree automaton*. Given an FTA A and an input tree t, a computation of A on t proceeds bottom up, assigning to each node x in t a set of states such that, if $\text{LABEL}(x) = a$, x's children are assigned the states $q_1, \ldots, q_k$ respectively, and there exists a transition $(q_1, \ldots, q_k; a) \rightarrow q$, then x is assigned the state q: if A is deterministic, then each node is assigned at most one state. The automaton *accepts* t if the root is assigned some state from a given set of terminal states; $lang(A)$ denotes the set of trees accepted by A, and is called a *regular tree language*.

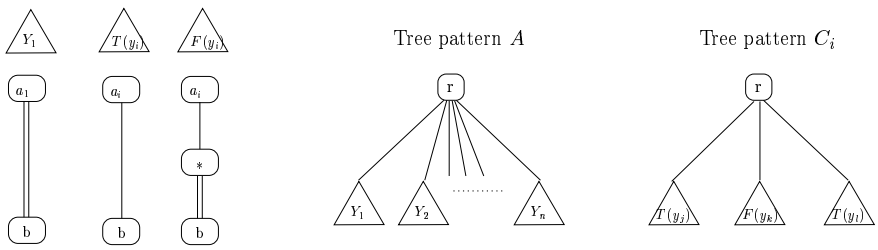


Figure 8: The canonical models of A encode truth assignments to the literals $y_1, y_2, \ldots y_n$ of ψ based on the lengths of the branches. Tree pattern C_i is constructed from clause $c_i = (\neg y_j \vee y_k \vee \neg y_l)$.

Given an FTA A with n states, one can construct an equivalent DFTA $det(A)$ with at most 2^n states. $A \times A'$ denotes the standard product automaton of two FTA's (definition omitted from this abstract): $lang(A \times A') = lang(A) \cap lang(A')$. If A' is deterministic, then one can check $lang(A) \subseteq lang(A')$ in time $O(|A||A'|)$, where $|A|$ (and similarly $|A'|$) denotes the number of states in A plus the number of transitions in A (the latter may be exponential in the number of states)[19].

An *alternating finite tree automaton*, AFTA, A, has transitions of the form $(q_1, \ldots, q_k; a) \rightarrow q$, $k = 0, \ldots, r$, and transitions of the form $\bigwedge(q_1, \ldots, q_m) \rightarrow q$, $m \geq 0$: we call the latter an AND-transition. Given an input tree, t, a computation of A on t proceeds by assigning states to tree nodes, like in an FTA, with the following addition: if a node x is assigned the states $q_1, \ldots, q_m$ and there exists a transition $\bigwedge(q_1, \ldots, q_m) \rightarrow q$, then x is also assigned the state q. A accepts t if t's root contains a terminal state. The definition of an AFTA is adapted from word automata [5].

Given an AFTA with n states there exists an equivalent DFTA $det(A)$ with at most 2^n states. Thus, the additional AND-transitions do not increase the cost of determinizing the automaton.

Ranking Trees Recall that Σ is an infinite, unranked alphabet. Given a pattern p, recall from Sec. 4 that we chose a symbol $z \in \Sigma$ s.t. z does not occur in p, in order to define the set of canonical models $m^z(p)$. We now construct a finite, ranked alphabet Ω_p, an *unranking function* $U_p : T^<_{\Omega_p} \rightarrow T_\Sigma$, and a regular tree language $Reg_{\Omega_p} \subseteq T^<_{\Omega_p}$ s.t. $U_p(Reg_{\Omega_p}) = m^z(p)$.

Let $\text{NODES}(p) = \{x_1, \ldots, x_n\}$, and let $\text{EDGES}_{//}(p) = \{e_1, \ldots, e_d\}$ be all the descendant edges. Define $\Omega_p = \text{NODES}(p) \cup \text{EDGES}_{//}(p)$. The rank is defined as follows: for every $x \in \text{NODES}(p)$, its rank in Ω_p is $\text{DEGREE}(x)$; for every $e \in \text{EDGES}_{//}(p)$, its rank in Ω_p is 1. We now define an *unranking* function $u_p : \Omega_p \rightarrow \Sigma$. For $x \in \text{NODES}(p)$, if $\text{LABEL}(x) = a \in \Sigma$, then $u_p(x) = a$; else (when $\text{LABEL}(x) = *$) $u_p(x) = z$. For $e \in \text{EDGES}_{//}(p)$, $u_p(e) = z$. We extend u_p to a function $u_p : T^<_{\Omega_p} \rightarrow T^<_\Sigma$, where $T^<_\Sigma$ denotes the set of ordered, unranked trees labeled with Σ. Finally, we define $U_p : T^<_{\Omega_p} \rightarrow T_\Sigma$ as follows: $U_p(t)$ is $u_p(t)$ without the order. Now we can define Reg_{Ω_p} as follows. We fix an order on the nodes in p, and define Reg^z_Σ to be the set $m^z(p)$ in which every canonical model is ordered according to the order in p. Then, define $Reg_{\Omega_p} = u_p^{-1}(Reg^z_\Sigma)$.

We illustrate with the tree pattern in Fig. 9 (a). The label symbols in p are r, a, b and the wildcard $*$, and its nodes are denoted $r_1, a_1, a_2, b_1, b_2, b_3, z_1$. Then $\Omega_p = \{r_1, a_1, a_2, b_1, b_2, b_3, z_1, z_2, z_3\}$, with b_1, b_2 of arity 0, a_2 of arity 2, and the rest of arity 1.

The Algorithm We can now describe the algorithm:

Algorithm: CheckContainment II
Input: tree patterns p, p'
Output: *true* if $p \subseteq p'$; *false* otherwise.
Method:
Step 1: construct the DFTA A accepting Reg_{Ω_p}
Step 2: construct the AFTA A' accepting $U_p^{-1}(Mod(p'))$
Step 3: compute the AFTA $B = A \times A'$ (the product automaton).
Step 4: compute the DFTA $C = det(B)$.
Step 5: **if** $lang(A) \subseteq lang(C)$ **then** return *true*, **else** return *false*.

Before discussing the details, we prove that the algorithm is sound and complete. This follows from the proposition below, which is an immediate consequence of Proposition 5 and the fact that $U_p(Reg_{\Omega_p}) = m^z(p)$.

Proposition 6. *For any patterns p, p', $p \subseteq p'$ iff $Reg_{\Omega_p} \subseteq U_p^{-1}(Mod(p'))$.*

Algorithm Details The automaton in **Step 1** is deterministic and defined as follows: $\text{STATES}(A) = \text{NODES}(p)$, for each node $x \in \text{NODES}(p)$ with children $x_1, \ldots, x_k$, A has a transition $(x_1, \ldots, x_k; x) \rightarrow x$, and for every descendant edge e from node x to node y, A has a transition $(y; e) \rightarrow y$. The set of terminal states is $\{\text{ROOT}(p)\}$. Thus A has n states and $n+d$ transitions. It is so simple because we constructed the ranked alphabet Ω_p based on p's structure, and we ordered the trees in Reg_{Ω_p} according to the fixed order in p. For an illustration, the

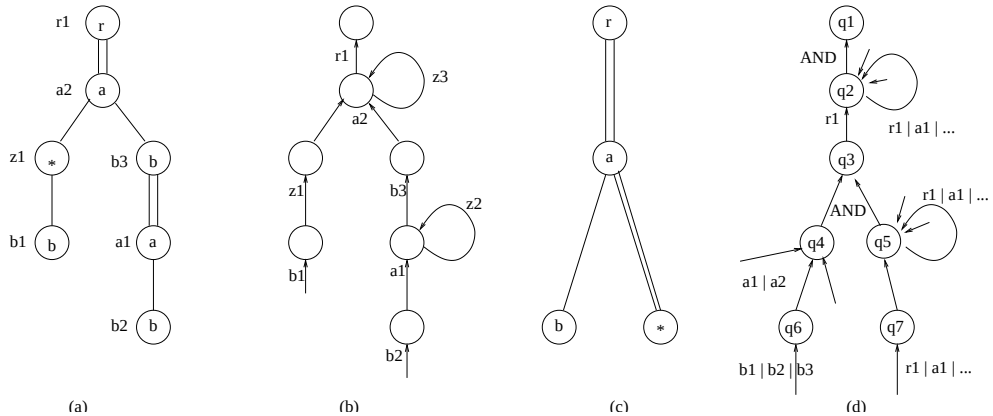


Figure 9: (a) a tree pattern p, (b) the DFTA A, (c) a pattern p', and (d) a fragment of its AFTA A'.

automaton A corresponding to the tree pattern in Fig. 9 (a) is shown in (b). It has 7 states and $7+2$ transitions.

The automaton in **Step 2**, A', is more complex, since it needs to accept trees matched by p', but ranked and ordered according to p. We also need A' to have polynomial size in p', and this is not possible with FTA's when p' has branches. The problem is illustrated by the following example: $p' = a[b_1][b_2]\dots[b_n]$, where the symbol a has arity n in p. In an input tree t a node labeled a may have the children $b_1, \dots, b_n$ in any order, and A' must enumerate all $n!$ orderings of $b_1, \dots, b_n$, resulting in $n!$ transitions.

Our solution is to make A' an AFTA. For a node in p' with several subpatterns, A' will check that each subpattern matches independently, then use an AND-transition to verify that all subpatterns matched. More precisely, denoting $\text{NODES}(p'), \text{EDGES}(p')$ the set of nodes, and edges in p', we define $\text{STATES}(A') = \text{NODES}(p') \cup \text{EDGES}(p') \cup \{q_0\}$, and the transitions are the following:

- $(q_0, \dots, q_0; a) \to q_0$, for every $a \in \Omega_p$.
- $(q_0, \dots, q_0; a) \to x$, for every leaf node $x \in \text{NODES}(p')$ and every symbol $a \in \Omega_p$ s.t. $u_p(a) = \text{LABEL}(x)$ or $\text{LABEL}(x) = *$.
- $(q_0, \dots, q_0, y, q_0, \dots, q_0; a) \to e$, for every node y with incoming edge e, and for every $a \in \Omega_p$ s.t. $u_p(a) = \text{LABEL}(x)$ or $\text{LABEL}(x) = *$. If a has arity k, then there are k such transitions, one for each position of y in the sequence of k states.
- $(q_0, \dots, q_0, e, q_0, \dots, q_0; a) \to e$ for every descendant edge e, and every $a \in \Omega_p$. If a has arity k, then there are k such transitions, one for each position of e.
- $\bigwedge(e_1, \dots, e_k) \to x$, for every node x with outgoing edges $e_1, \dots, e_k$, $k \geq 1$.

The set of terminal states is $\{\text{ROOT}(p')\}$. Hence, A' has $2n'$ states, and $2n' \times |\Omega_p| \leq 2n' \times 2n$ transitions.

Fig. 9 (c) shows a pattern p' and (d) shows a fragment of A'. Here A' has 8 states (q_0 is not shown), and the figure sketches some transitions. For example the transitions into q_4 are: $(q_6; a_1) \to q_4$, $(q_6, q_0; a_2) \to q_4$, and $(q_0, q_6; a_2) \to q_4$, because a_1 is unary and a_2 is binary.

Step 3 is an optimization. We could have skipped this step and computed $C = det(A')$ directly, then checked $lang(A) \subseteq lang(C)$. But $det(A \times A')$ has a subset of the states of $A \times det(A')$, since A is deterministic, and, in some cases, $det(A \times A')$ has only polynomially many states while $det(A')$ has exponentially many. For an illustration, if $p = a/b_1/b_2/\dots/b_n$ and $p' = a[.//b_1][.//b_2]\dots[.//b_n]$, then $det(A')$ has exponential size (it needs to allow for all orderings of $b_1, \dots, b_n$), while $det(A \times A')$ has polynomially many states, since only 1 such ordering is present. This optimization is essential for both Theorem 4 and 5.

Finally, **Step 4** and **Step 5** are self-explanatory.

Theorems 4 and 5 follow from a careful analysis of the number of states in $C = det(A \times A')$. Their proofs are omitted from this abstract.

8. DISCUSSION

This section briefly covers additional topics of interest.

Disjunction It is easy to extend our discussion to patterns with disjunction. It turns out, however, that with disjunction, P and XP behave differently. We extend P to $\text{P}^{\{\text{or}\}}$ allowing pattern trees with *or* nodes of degree two. A tree t is accepted by p if (1) there exists a choice of "left" or "right" for each *or*-node in p, which transforms p into a pattern q without *or*-nodes, and (2) $q(t)$ is true. If no branches other than OR nodes are allowed, then containment for two patterns in $\text{P}^{\{\text{or}\}}$ can be reduced in PTIME to the containment of two patterns

in P, by a simple application of Lemma 1. However, when branches are allowed, a minor modification to the proof of Theorem 1 shows that containment for $P^{\{[],or\}}$ patterns is coNP-complete. On the other hand we can extend the grammar for XPath, Eq.(1), with $q ::= q \mid q$, and denote with $XP^{\{|\}}$ this extended language. $XP^{\{|\}}$ is exponentially more concise than $P^{\{or\}}$, because for example $(a_1 \mid b_1)/(a_2 \mid b_2)/\ldots/(a_n \mid b_n)$ requires a tree pattern of exponential size in $P^{\{or\}}$. For $XP^{\{|\}}$ containment is coNP-complete even if we disallow all three features $[\,], *, //$, which follows from the complexity of containment for regular languages restricted to union and concatentation [12].

Evaluation on graphs In addition to the tree structure, an XML document has a graph structure defined by node ids and references. XPath can traverse this graph structure. This is captured in our formalism by interpreting tree patterns on graphs rather than trees. All results in this paper apply directly to an extension of boolean patterns evaluated on graphs. Namely, for a graph g, $p(g)$ is true if there exists an embedding $e : p \rightarrow g$, and we have $\forall g.p(g) \Rightarrow p'(g)$ iff $\forall t.p(t) \Rightarrow p'(t)$. In other words the containment problem on graphs is the same as that on trees. To see this, let unfold(g) be the (possibly infinite) tree unfolding of some graph g. Then, for any pattern p, the following are equivalent: (1) $p(g)$ is true, (2) p(unfold(g)) is true, (3) $\exists t \subseteq$ unfold(g), t finite and $p(t)$ is true.

Computation Tree Logic (CTL) Tree patterns can be expressed in a certain fragment of CTL [23] consisting of *true*, $x = a$, conjunction, "eventually true" formulas EFϕ, and "successively true" formulas EXϕ. Defined properly, this fragment is completely equivalent to tree patterns in $P^{\{[],*,//\}}$, and the containment problem becomes the implication problem for this fragment. All coNP completeness results in this paper apply to this fragment of CTL as well, showing that in this fragment the implication problem is coNP-complete.

Finite Alphabet Throughout the paper we assumed that our alphabet Σ is infinite. While this is the only scenario of interest in practice (since the alphabet denotes XML tags), the case when Σ is finite is interesting from a theoretical standpoint. All co-NP completeness results in this paper hold if $\mid \Sigma \mid = 2$ (the idea is that symbols from a larger alphabet can be encoded with chains, if we have at least two symbols). But most decision procedures, including those that preceded our work, fail. For instance, when p=`a/a//b/b` and p'=`a//a/b//b`, then $p \subseteq p'$ if $\Sigma = \{a, b\}$ but $p \not\subseteq p'$ when $\Sigma = \{a, b, c\}$. Thus the homomorphism criterion in [1] no longer holds for a finite alphabet.

9. RELATED WORK

The classes of patterns that include descendant edges ($P^{\{[],//\}}$ and $P^{\{[],*,//\}}$) can be expressed in datalog with recursion, for which containment is undecidable in general [20]. In [25] the author showed, using chase techniques, that the datalog fragment needed for $P^{\{[],*,//\}}$ has a decidable containment problem. Containment for $P^{\{[],//\}}$ was shown to be in PTIME in [1]. Queries in $P^{\{[],*\}}$ can be viewed as conjunctive queries over tree structures. In general, containment for conjunctive queries is NP-complete [6], however for acyclic conjunctive queries containment is in PTIME [28], from which it follows that $P^{\{[],*\}}$ containment is solvable in PTIME. This bound for $P^{\{[],*\}}$ was also noted in [26].

Linear queries in $P^{\{*,//\}}$ are a special case of regular expressions on strings, for which there is a PSPACE-complete containment algorithm in general [22]. For the fragment of regular string expressions in $P^{\{*,//\}}$, a linear-time containment algorithm is claimed in [17], although the proof was not published. A PTIME algorithm for linear patterns in $P^{\{//\}}$ was provided in [2].

On a graph-based data model, the authors of [11] showed that for a restricted language without wildcard, similar to $P^{\{[],//\}}$, containment is NP-complete. In [3] the authors study tree two-way regular path queries on a graph model. In addition to a more general data model, these queries are more expressive than ours because they allow general regular path expressions and inverse. A PSPACE upper bound for containment is shown for this class of queries. Finally, the authors of [10] prove containment results for a host of XPath-related languages. One closely-related result applies to an extension of $P^{\{[],*,//\}}$ which includes binding of variables and equality testing, for which containment is shown to be Π_2^p-hard.

10. CONCLUSION

We have studied the complexity of containment and equivalence for an important core fragment of XPath. Many XML applications could benefit from a practical decision procedure for containment of such expressions. We show this fragment of XPath has an intractable containment problem in general, and our results provide intuition into the factors that contribute to its high complexity. Nevertheless, we show that in some significant special cases, containment can be decided efficiently, and we provide an algorithm that does so.

One direction for future work is to extend this fragment of XPath with additional features, although it is clear that it will be even more challenging to prove efficient special cases of the problem. Another direction is to study containment of XPath expressions over sets of documents conforming to constraints or schema restrictions. Preliminary work shows that sufficiently expressive constraints make this problem intractable for XPath fragments that otherwise have efficient containment problems.

11. ACKNOWLEDGMENTS

We would like to thank Igor Tatarinov, Paul Beame, and Moshe Vardi for their comments. Suciu was partially supported by the NSF CAREER Grant 0092955,

a gift from Microsoft, and an Alfred P. Sloan Research Fellowship.

12. REFERENCES

[1] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. *SIGMOD*, 2001.

[2] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for xml, 2000.

[3] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query answering and query containment over semistructured data. Submitted for publication, 2001.

[4] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 07 June 2001. W3C working draft.

[5] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. In *Journal of the ACM*, pages 115–133, January 1981.

[6] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of 9th ACM Symposium on Theory of Computing*, pages 77–90, May 1977.

[7] R. Cole, R. Harihan, and P. Indyk. Tree pattern matching and subset matching in deterministic $o(nlog3n)$ time. *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 245–254, 1999.

[8] S. DeRose, R. D. Jr., and E. Maler. XML pointer language (XPointer) working draft. http://www.w3.org/TR/1999/WD-xptr-19991206, December 1999.

[9] S. DeRose, E. Maler, and D. Orchard. Xml linking language (Xlink). http://www.w3.org/TR/2000/REC-xlink-20010627, June 2001.

[10] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *KRDB 2001*, 2001.

[11] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Principles of Database Systems (PODS)*, pages 139–148, 1998.

[12] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness*. W. H. Freeman, San Francisco, 1979.

[13] C. Hoffmann and M. O'Donnell. Pattern matching in trees. *Journal of the Association for Computing Machinery*, 29(1):68–95, 1982.

[14] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, pages 340–356, 1995.

[15] Kosaraju. Efficient tree pattern matching. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989.

[16] G. Miklau and D. Suciu. Containment and equivalence of tree patterns. University of Washington Technical Report (TR 02-02-03), February 2002. http://www.cs.washington.edu/homes/gerome.

[17] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[18] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages, Volume 1*. Springer Verlag, Berlin, Heidelberg, New York., 1997.

[19] H. Seidl. Personal communication, August 2001.

[20] O. Shmueli. Equivalence of datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231–242, February 1993.

[21] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[22] L. J. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *5th STOC*, pages 1–9. ACM, 1973.

[23] M. Vardi. Why is modal logic so robustly decidable, 1997.

[24] P. Wadler. A formal semantics of patterns in xslt. *Markup Technologies*, pages 183–202, 1999.

[25] P. T. Wood. On the equivalence of xml patterns. *International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 1152–1166, 2000.

[26] P. T. Wood. Minimizing simple xpath expressions. *Fourth International Workshop on the Web and Databases (WebDB'2001)*, 2001.

[27] XML Schema part 1: Structures. http://www.w3.org/TR/1999/WD-xmlschema-1-19991217/, 17 December 1999. W3C Working Draft.

[28] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delobel(eds)*, 1981.