Containment and Equivalence for a Fragment of XPath

GEROME MIKLAU and DAN SUCIU
University of Washington

XPath is a 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 algorithm for containment that runs in exponential time, 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.

Categories and Subject Descriptors: []:

General Terms:

Additional Key Words and Phrases:

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 [Chamberlin et al. 2001] to bind variables; in XML Schema [XSch 1999] to define keys; in XLink [DeRose et al. 2001] and XPointer [DeRose et al. 1999] to reference elements in external documents; in XSLT as match expressions, and in content-based packet routing [Snoeren et al. 2001] as filter expressions. Instances of the containment problem for XPath expressions occur in each these applications, and others. For example, inference of keys described by XPath expressions requires a test for containment, and similarly certain optimization methods for XQuery require an XPath containment test.

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 predicates (or filters, 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

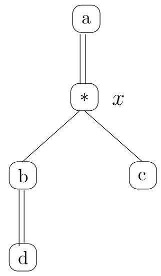


Fig. 1. A simple tree pattern with return node x marked. It corresponds to the XPath expression a// * [b//d][c].

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: the b and c children may occur in any order.

For a given XPath 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 $XP^{\{[],*,//\}}$ expressions is coNP 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, a PTIME containment algorithm for the fragment $XP^{\{[],*\}}$ follows from classic results on acyclic conjunctive queries [Yannakakis 1981]. Without label wildcards, the fragment $XP^{\{[],//\}}$ was recently found to have a polynomial time containment algorithm [Amer-Yahia et al. 2001]. And for $XP^{\{*,//\}}$, patterns do not have branching, and are therefore closely related to a fragment of regular string expressions. A result in [Milo and Suciu 1999] 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 (Algorithm 3 and Fig. 9), 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 that generalize those where containment was known to be in PTIME: (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): containment can be checked in PTIME whenever the number of //'s in p is bounded by some number d (d will be the degree of the polynomial describing the running time). However, (b) and (c) have negative answers. More precisely, the containment of $XP^{\{[],*,//\}}$ expressions is coNP-complete even when p has no $*$'s and p' contains only two $*$'s, which answers (b) negatively. For (c), the containment of $XP^{\{[],*,//\}}$ 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 that runs in exponential time in general, but runs in PTIME in some special cases of practical interest (Algorithm 2).

In summary the results in this paper characterize almost completely the boundary between the PTIME cases and the co-NP complete cases for the XPath containment problem. The paper also describes two algorithm for containment, a PTIME imcomplete algorithm, and an exponential time complete algorithm, and proves formal properties justifying their usage in practice.

A note to the practitioner The reader interested in implementing a containment algorithm may first try the algorithm in Fig. 9 (Sec. 3.2), and read the associated homomorphism Algorithm 3. It checks containment of two XPath expressions[1] p and p' in time $O(|p||p'|)$, which is more efficient than an ad-hoc approach, which runs in time $O(|p|^2|p'|)$. The only drawback of this (and of the naive) algorithm is that it may return false negatives, i.e. may fail to detect containment for certain XPath expressions p and p'. But cases when the algorithm fails are very unlikely in practice: the "simplest" such case is illustrated in Fig. 10, and the reader may inspect this figure to determine whether such cases are indeed expected in the specific application. If yes, and a complete algorithm is really needed, then the reader is advised to consider Algorithm 2, in Sec. 3.1.

Paper Organization The organization of the paper is as follows. Section 2 contains the definition of tree patterns, their semantics and evaluation, and the relationship between tree patterns and XPath expressions. Sections 3 discusses two ways of reasoning about containment: canonical models and pattern homomorphisms, respectively. Canonical models result in a complete algorithm for checking containment whose worst case running time is exponential (Sec. 3.1), while homomorphisms lead to a polynomial time algorithm for checking containment that is incomplete (Sec. 3.2). We state and prove the co-NP hardness results in Section 4. Section 5 discuss a number of assorted issues: disjunction in patterns, connections to computation tree logic, and the special case when the alphabet is finite. Section 6 discusses related work, and Section 7 concludes.

[1]The algorithm is expressed in terms of tree patterns. Sec. 2.1 describes the correspondence between tree patterns and XPath expressions.

2. DEFINITIONS AND BACKGROUND

We review here the basic definitions of XML trees, XPath queries, and their semantics. Then we introduce an alternative query formalism, called tree pattern, which is equivalent to XPath queries, and prove that for the purpose of the containment and equivalence of XPath queries it is sufficient to consider only the containment problem for *boolean* tree patterns. Finally, we describe an algorithm for the evaluation of a boolean tree pattern on an XML tree.

2.1 Trees and Patterns

XML Trees We model an XML document as a tree with nodes labeled from an infinite alphabet Σ. The symbols in Σ represent the element labels, attribute labels, and text values that can occur in XML documents. By requiring this set to be infinite we ensure that no XPath expression contains all possible labels: all our results on containment of XPath expressions depend critically on this assumption, and we will briefly discuss the case when Σ is finite in Sec. 5. Notice that the XML trees we consider are unordered and unranked. We denote the set of all trees with T_Σ. For a tree $t \in T_\Sigma$ we denote $\text{NODES}(t)$ and $\text{EDGES}(t)$ the sets of nodes and edges, respectively, by $\text{ROOT}(t)$ its root node, and write $\text{LABEL}(x)$ for the label on node x, $\text{LABEL}(x) \in \Sigma$. We also denote $\text{EDGES}^+(t)$ the transitive closure of $\text{EDGES}(t)$: $\text{EDGES}^+(t) = \text{EDGES}(t) \cup \text{EDGES}(t) \circ \text{EDGES}^+(t)$, and denote $\text{EDGES}^*(t)$ the reflexive and transitive closure of $\text{EDGES}(t)$: $\text{EDGES}^*(t) = \text{NODES}^2(t) \cup \text{EDGES}^+(t)$. Define the distance between two nodes $(x, y) \in \text{EDGES}^*(t)$ to be: $d(x, x) = 0$, $d(x, y) = 1$ when $(x, y) \in \text{EDGES}(t)$ and $d(x, y) = 1 + d(z, y)$ when $\exists z, (x, z) \in \text{EDGES}(t), (z, y) \in \text{EDGES}^+(t)$. Finally, we define the size of a tree t, in notation $|t|$, to be the number of edges in t.

XPath Queries We study a fragment of XPath, denoted $\text{XP}^{\{[\,],*,//\}}$, consisting of expressions given by the following grammar:

$$q \rightarrow n \;\;|\;\; * \;\;|\;\; . \;\;|\;\; q\,/\,q \;\;|\;\; q\,//\,q \;\;|\;\; q\,[\,q\,] \tag{1}$$

Here $n \in \Sigma$ is any label, $*$ denotes a label wildcard, and . denotes the "current node". The constructions / and // mean child and descendant navigation respectively, while [ ] denotes a predicate. Our discussion will focus on the three constructs $[\,], //, *$ and the notation $\text{XP}^{\{[\,],*,//\}}$ signifies that all three are allowed. We will denote $\text{XP}^{\{[\,],*\}}$, $\text{XP}^{\{[\,],//\}}$, $\text{XP}^{\{*,//\}}$ the subsets of XPath expressions restricted to only two of the three constructs.

The meaning of an expression $q \in \text{XP}^{\{[\,],*,//\}}$ on a tree $t \in T_\Sigma$, in notation $q(t)$, is a set of nodes in t. We adapt the formal semantics from [Wadler 1999], fixing the root as context node: that is, $q(t) = q(\text{ROOT}(t))$, where $q(x)$ for $x \in \text{NODES}(t)$ is defined below, by induction on the structure of q:

$$\begin{aligned}
n(x) &= \{y \mid (x, y) \in \text{EDGES}(t), \text{LABEL}(y) = n\} \\
*(x) &= \{y \mid (x, y) \in \text{EDGES}(t)\} \\
.(x) &= \{x\} \\
(q_1/q_2)(x) &= \{z \mid y \in q_1(x), z \in q_2(y)\} \\
(q_1//q_2)(x) &= \{z \mid y \in q_1(x), (y, u) \in \text{EDGES}^*(t), z \in q_2(u)\} \\
q_1[q_2](x) &= \{y \mid y \in q_1(x), q_2(y) \neq \emptyset\}
\end{aligned}$$

Notice that the definition of $q(t)$ never inspects the label of the root node, $\text{ROOT}(t)$. For example if $a \in \Sigma$, then $a(t)$ returns the children of $\text{ROOT}(t)$ that are labeled a, and ignores the root label. This follows standard XML semantics, where $\text{ROOT}(t)$ corresponds to the *document node* and is unlabeled.

We need to include the current node construct in our fragment in order to use it contexts like $a/b[.//c]$. In other cases it can sometimes be eliminated, e.g. $a/./b$ is equivalent to a/b. However, in other case it cannot be eliminated. For example the semantics of $a//.$ is that of the union between a and $a//*$. While most of the results we discuss in this paper extend to a language which has explicit union, we prefer to keep the discussion simple and restrict the usage of . to a context immediately inside a predicate [ ]. Thus, a construction like $a[.//b]$ is allowed, while $a//.$ is not.

Two expressions q, q' in $\text{XP}^{\{[],*,//\}}$ are contained, in notation $q \subseteq q'$, if their result sets are contained for every tree: $q(t) \subseteq q(t'), \forall t \in T_\Sigma$. Two expressions are equivalent if their result sets are equal.

Tree Patterns We use an alternative, and more general representation of queries as tree patterns. A *tree pattern* of arity k, $k \geq 0$ is a tree p whose nodes are labeled with symbols from $\Sigma \cup \{*\}$, with a distinguished subset of edges called *descendant edges*, and a k-tuple of nodes called the *distinguished nodes*. We use the same notations $\text{NODES}(p), \text{EDGES}(p), \text{ROOT}(p)$ and $\text{LABEL}(x)$ for $x \in \text{NODES}(p)$ as before. The set of descendant edges is denoted $\text{EDGES}_{//}(p)$, while the other edges are called *child edges* and their set is denoted $\text{EDGES}_{/}(p)$; thus $\text{EDGES}(p) = \text{EDGES}_{//}(p) \cup \text{EDGES}_{/}(p)$. In diagrams we represent descendant edges with double lines and child edges with single lines. Figure 1 illustrates a tree pattern of arity 1; another tree pattern of arity 1 is shown in Figure 2 (b). In both figures the distinguished node is indicated with an x.

The set of all tree patterns is denoted $\text{P}^{\{[],*,//\}}$. We define the following three subclasses: $\text{P}^{\{[],*\}}$ denotes all patterns without descendant edges, $\text{P}^{\{[],//\}}$ denotes patters without $*$ labels, and $\text{P}^{\{*,//\}}$ denotes linear patterns, i.e. where every node has at most one child. As before, the size of a tree pattern, $|p|$, is defined to be the number of edges in p.

Every tree $t \in T_\Sigma$ is automatically a tree pattern of arity 0: just define $\text{EDGES}_{/}(t) = \text{EDGES}(t)$ and $\text{EDGES}_{//}(t) = \emptyset$.

Given a tree pattern p and a tree t, define an *embedding* from p to t to be a function $e : \text{NODES}(p) \rightarrow \text{NODES}(t)$ which satisfies the following conditions:

Root-preserving. $e(\text{ROOT}(p)) = \text{ROOT}(t)$,
Label-preserving. For each $x \in \text{NODES}(p)$, $\text{LABEL}(x) = *$ or $\text{LABEL}(x) = \text{LABEL}(e(x))$,
Child-edge-preserving. For each $(x, y) \in \text{EDGES}_{/}(p)$, $(e(x), e(y)) \in \text{EDGES}(t)$, and
Descendant-edge-preserving. For each $(x, y) \in \text{EDGES}_{//}(p)$, $(e(x), e(y)) \in \text{EDGES}^{+}(t)$.

An embedding does not need to be an injective function. An example of an embedding is pictured in Figure 2(a) and (b).

Finally, denoting $\bar{x} = (x_1, x_2, \ldots x_k)$ the k-tuple of distinguished nodes in p, we define the meaning of a tree pattern p on a tree t to be the following subset of $\text{NODES}^k(t)$:

$$p(t) = \{e(\bar{x}) \mid e \text{ is an embedding from } p \text{ to } t\}$$

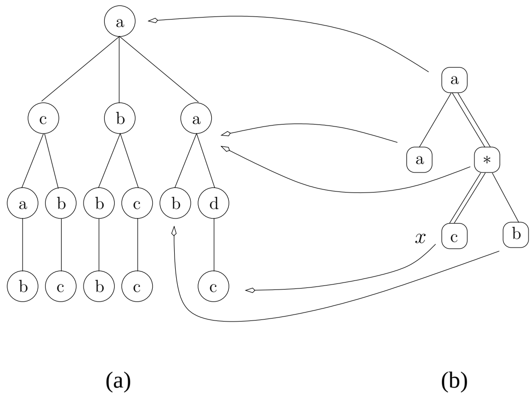


Fig. 2. (a) tree instance t, (b) pattern p and an embedding from p to t.

2.2 From XPath to Tree Patterns

Every XPath expression can be translated into a tree pattern of arity 1, and vice versa, while preserving semantics. The only subtlety is that XPath expressions ignore the label of the root node, while tree patterns do not. To account for that, given some tree $t \in T_\Sigma$ and a label $r \in \Sigma$, we denote r/t a tree whose root node is labeled r and has a single subtree, t. Then every XPath expression q can be translated into a tree pattern $\bar{q}$ of arity one, such that $\forall t \in T_\Sigma$, $q(r/t) = \bar{q}(t)$; and, conversely, every tree pattern p of arity one can be translated into an XPath expression $\bar{p}$ s.t. $\forall t \in T_\Sigma$, we have $p(t) = \bar{p}(r/t)$. We omit the tedious, but straightforward translation, and only illustrate with two examples: Figure 1 shows the tree pattern for `a//*[b//d][c]`, and Figure 2(b) shows the tree pattern for the XPath expression `a[a]//*[b]//c`. The containment problems for XPath expressions and for unary tree patterns are thus equivalent. Moreover, the translation also preserves the fragments of interest to us; $XP^{\{[],*\}}$ corresponds to $P^{\{[],*\}}$, $XP^{\{[],//\}}$ corresponds to $P^{\{[],//\}}$ and $XP^{\{*,//\}}$ corresponds to $P^{\{*,//\}}$ respectively. Thus, from now on, we shall consider tree patterns only, i.e. $P^{\{[],*,//\}}$ and its fragments, but it should be clear that all results apply also to XPath expressions.

Tree patterns are actually more general, since they can have arities other than 1. This makes them applicable, for example, to the optimization of the `FOR` clause of an XQuery expression [Chamberlin et al. 2001]: there, multiple XPath expressions are used to bind multiple variables, and they can be combined into a single tree pattern whose distinguished nodes correspond to those variables. For the study of containment, however, arity is not an important consideration, as we explain next.

2.3 Boolean Patterns

For the purpose of the containment problem, it suffices to limit our discussion to tree patterns with arity zero, which we call *boolean* patterns. When p is boolean then $p(t)$ is either $\emptyset$ or $\{()\}$: in the first case we say that $p(t)$ is *false*, in the latter we say it is *true*. For boolean patterns, containment means implication: $p \subseteq p'$ if and

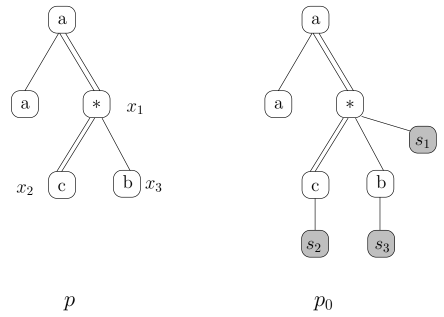


Fig. 3. A tree pattern p of arity 3, with the distinguished nodes x_1, x_2, x_3, and its translation to a boolean pattern p_0, used in Proposition 1: p_0 has three extra nodes labeled s_1, s_2, s_3.

only if $\forall t.p(t) \rightarrow p'(t)$. The next proposition shows how a solution to containment of boolean patterns can be used to solve containment for k-ary patterns.

PROPOSITION 1. *Let $s_1, \ldots, s_k$ be k labels that are not in Σ. There is a translation of k-ary patterns over the alphabet Σ, to boolean patterns over the alphabet $\Sigma \cup \{s_1, s_2, \ldots\ s_k\}$, such that for any k-ary patterns p, p', and their translations p_0, p'_0, we have $p \subseteq p'$ if and only if $p_0 \subseteq p'_0$.*

The translation of p into p_0 consists of adding k extra nodes, labeled $s_1, \ldots, s_k$ and making them children of the k distinguished nodes in p. Figure 3 illustrates this construction. Intuitively, $p \subseteq p' \iff p_0 \subseteq p'_0$ because the extra nodes in p_0 and p'_0 have to match exactly, implying that the distinguished nodes in p and p' match too. A formal proof based on this argument is given in the Appendix.

Thus, all results about containment of boolean patterns immediately apply to k-ary patterns, for any $k \geq 0$. Notice that the translation from p to p_0 in Proposition 1 preserves both fragments $\mathrm{P}^{\{[],//\}}$ and $\mathrm{P}^{\{[],*\}}$, hence all results for boolean patterns in these two fragments also hold for k-ary patterns in the same fragments. However, the translation does not preserve $\mathrm{P}^{\{*,//\}}$, since p_0 may have extra branches. For the positive positive results to carry from boolean patterns in $\mathrm{P}^{\{*,//\}}$ to k-ary patterns, we need a different encoding, which we describe here briefly. Assume first that p and p' have no output nodes labeled with $*$'s. Then we construct the boolean tree patterns p_0 and p'_0 by replacing each label a on some node x with a new label consisting of a and followed by those symbols s_i that correspond to the positions in the output tuple where x occurs. For example, assuming the output tuple to be (x, y, x, x, z, y), then node x will have its label a replaced with (a, s_1, s_3, s_4), node y will have its label b replaced with (b, s_2, s_6), and node z will have its label c replaced with (c, s_5). All other nodes have their labels unchanged. Doing this in both p and p' results in the boolean patterns p_0, p'_0, over the alphabet $\Sigma \times \mathcal{P}(\{s_1, \ldots, s_k\})$. The new labels ensure that output nodes in p'_0 can only be mapped to output nodes in p_0, but in general this technique fails, becuase it prevents us to map a non-output node in p'_0 to an output node in p_0. Still, the technique work for the case when p' is a linear pattern, since then any embedding from p'_0 is injective, and we never need to map a non-output node to an output node. It remains now to consider the case when p and p' have $*$'s. Here the observation is that an output node must be mapped to the correpsondingly labeled output node: hence if it is labeled with $*$, we may as well relabel it, and it's unique target in the image with a new symbol in Σ. This eliminates all $*$'s from output nodes, and we apply the construction above. We leave the details to the reader. As a consequence, all results discussed in the paper for boolean patterns also apply to k-ary patterns, for arbitrary k.

In the rest of the paper we will assume all tree patterns to be boolean tree patterns, unless otherwise stated.

2.4 Mutual Reducibility of Containment and Equivalence

The containment and equivalence problems are mutually reducible in polynomial time. Equivalence is simply two-way containment. 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_0 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, $\text{LABEL}(\text{ROOT}(p_0))$ is their common label; in the latter $\text{LABEL}(\text{ROOT}(p_0)) = a$. Pattern p_0 is a boolean pattern such that, for any input tree t, $p_0(t)$ is true if and only if $p(t) \wedge p'(t)$ is true. Then it follows that $p \subseteq p'$ if and only if p is equivalent to p_0. We discuss only containment in the remainder of the paper.

2.5 Tree Pattern Evaluation

We give below an algorithm that, given a boolean pattern p and tree t, checks whether $p(t)$ is true. The algorithm deploys a standard dynamic programming method, computing a boolean matrix $\mathcal{C}(x, y)$ for $x \in \text{NODES}(t), y \in \text{NODES}(p)$ s.t. $\mathcal{C}(x, y)$ is true if there exists an embedding from the subpattern rooted at y to the subtree rooted at x. An improvement, suggested by [Dalvi and Sanghai 2002], allows the algorithm to run in time $O(|p||t|)$, by computing a second matrix $\mathcal{D}$, whose entry $\mathcal{D}(x, y)$ is true if there exists an embedding from the subpattern rooted at y to some subtree of t whose root is either x or a descendant of x. Recall that $|t|$ denotes the number of edges in t, and $|p|$ the number of edges in p.

Algorithm 1 Find embedding $p \to t$

1: **for** x in $\text{NODES}(t)$ **do** {The iteration proceeds bottom up on nodes of t}
2: **for** y in $\text{NODES}(p)$ **do** {The iteration proceeds bottom up on nodes of p}
3: **compute** $\mathcal{C}(x, y) = (\text{LABEL}(y) = * \vee \text{LABEL}(y) = \text{LABEL}(x)) \wedge$
4: $\bigwedge_{(y,y') \in \text{EDGES}_{/}(p)} (\bigvee_{(x,x') \in \text{EDGES}(t)} \mathcal{C}(x', y')) \wedge$
5: $\bigwedge_{(y,y') \in \text{EDGES}_{//}(p)} (\bigvee_{(x,x') \in \text{EDGES}(t)} \mathcal{D}(x', y'))$
6: **compute** $\mathcal{D}(x, y) = \mathcal{C}(x, y) \vee \bigvee_{(x,x') \in \text{EDGES}(t)} \mathcal{D}(x', y)$
7: **return** $\mathcal{C}(\text{ROOT}(t), \text{ROOT}(p))$

PROPOSITION 2. *Algorithm 1 decides for any tree pattern p, and input tree t whether $p(t)$ is true, and runs in time $O(|p||t|)$.*

PROOF. The inner loop consists of lines 4, 5, and 6. The condition $\mathcal{C}(x', y')$ in line 4 is checked once for every pair of edges $(y, y') \in \text{EDGES}_{/}(p)$ and $(x, x') \in \text{EDGES}(t)$. The condition $\mathcal{D}(x, y')$ is checked once for every pair of edges $(y, y') \in \text{EDGES}_{//}(p)$

and $(x, x') \in \text{EDGES}(t)$. The total number of times that these two conditions are checked is thus no more than $|\text{EDGES}(p)||\text{EDGES}(t)| = |p||t|$. The condition $\mathcal{C}(x', y)$ in line 6 is checked once for every node $y \in \text{NODES}(p)$ and every edge in $\text{EDGES}(t)$. The total running time is thus $O(|p||t|)$. $\square$

2.6 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 [Hoffmann and O'Donnell 1982]. Here both the patterns and the trees are ordered, and the patterns are boolean, and without descendant edges. An embedding is required to be order preserving, but not necessarily root preserving. A simple extension of the Algorithm 1 to account for the node order runs in time $O(mn)$ for a pattern with m nodes and a tree with n nodes. Improving this bound was a long-time open problem, first solved in [Kosaraju 1989] to attain a bound of $O(nm^{0.75} polylog(m))$. The best algorithm to date is $O(nlog^3 m)$[Cole et al. 1999].

The second related problem was defined in [Kilpelainen and Mannila 1995] 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, where the embedding e is required to map two distinct children of a node $x \in \text{NODES}(p)$ into two different subtrees of $e(x)$. In particular, e is injective. This subtle difference results in an increased evaluation complexity and it is shown in [Kilpelainen and Mannila 1995] that unordered tree inclusion is NP-complete.

3. CHECKING CONTAINMENT

How does one check containment $p \subseteq p'$ for two boolean patterns ? Based on it's definition, is not even clear that this is decidable, since we need to check that $p(t) \Rightarrow p'(t)$ holds for all trees t, and there are infinitely many trees. One approach is to show that it suffices to check only finitely many trees: in fact one can restrict the search to "canonical" trees t, which "look like" p and are "no bigger" than p'. We pursue this idea in Sec. 3.1, and arraive at an exponential time algorithm for checking containment, and a proof that containment is in co-NP.

The second approach we discuss here uses a different concept to reason about containment: find a *pattern homomorphism* from p' to p. This leads us in Sec. 3.2 to a polynomial time containment algorithms that is sound, but not always complete. We show however that this procedure is also complete in certain cases.

3.1 Checking Containment with Canonical Models

A *model* of a boolean pattern p is a tree $t \in T_\Sigma$ on which p evaluates to true. We denote with $Mod(p)$ the set of models:

$$Mod(p) = \{t \in T_\Sigma \mid p(t) \text{ is true}\}$$

Containment of boolean patterns can be re-stated in terms of models: $p \subseteq p'$ if and only if $Mod(p) \subseteq Mod(p')$.

Consider the containment problem: given p, p', check if $p \subseteq p'$. One way to approach the problem is to search for a tree t s.t. $p(t)$ is true and $p'(t)$ is false, because this implies non-containment, $p \not\subseteq p'$. If such a tree t exists, we call it a *witness*. Thus, to solve the containment problem it suffices to search for a witness tree, or to establish that none exists. Since there are infinitely many trees, we need to reduce the space of witnesses. Clearly, it suffices to search the witness t in $Mod(p)$, but the latter is still an infinite set. To further restrict this set we introduce canonical models next.

Let p be a tree pattern, $p \in \mathrm{P}^{\{[\,],*,//\}}$, and suppose it has d descendant edges, $\textsc{edges}_{//}(p) = \{r_1, \ldots, r_d\}$. Given d numbers $\bar{u} = (u_1, \ldots, u_d)$, $u_1 \geq 0, \ldots, u_d \geq 0$, define the *$\bar{u}$-extension* of p, in notation $p[\bar{u}]$, to be the pattern obtained from p by replacing every descendant edge r_i with a chain of u_i new nodes, labeled $*$ and connected with child edges. Thus, $p[\bar{u}]$ is in $\mathrm{P}^{\{[\,],*\}}$, $\textsc{nodes}(p) \subseteq \textsc{nodes}(p[\bar{u}])$ and, if $r_i = (x, y)$ is the i'th descendant edge in p, then the distance in $p[\bar{u}]$ from x to y is $d(x, y) = u_i + 1$ (see the definition of the distance function in Sec. 2.1). We call the new nodes in $\textsc{nodes}(p[\bar{u}])$ *extension nodes*. For an illustration, consider the pattern in Fig. 4 (a) and the extension $p[0, 2]$ in (b); there are two extension nodes. We have the following:

LEMMA 1. *Let $e : p \to t$ be an embedding from the tree pattern p to the tree t. There exists a unique extension $p[\bar{u}]$ and a unique embedding $e' : p[\bar{u}] \to t$ s.t. $\forall x \in \textsc{nodes}(p)$, $e(x) = e'(x)$.*

PROOF. For each $i = 1, \ldots, d$, the embedding e maps the descendant edge $r_i = (x_i, y_i) \in \textsc{edges}_{//}(p)$ into a pair of nodes $(e(x_i), e(y_i)) \in \textsc{edges}^+(t)$. Define $u_i = d(e(x_i), e(y_i)) - 1$ where d is the distance function in t, and let $\bar{u} = (u_1, \ldots, u_d)$. Extend e to $e' : p[\bar{u}] \to t$ by mapping the extension nodes between x_i and y_i to the nodes connecting $e(x_i)$ to $e(y_i)$. □

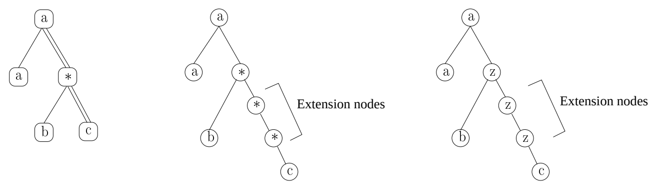
Given a pattern p and a symbol $z \in \Sigma$, denote $s^z(p)$ the tree pattern obtained by substituting each occurrence of $*$ in p with z. Hence $s^z(p) \in \mathrm{P}^{\{[\,],//\}}$. We can now define canonical models formally:

DEFINITION 1. *Let $z \in \Sigma$ be some symbol, and $p \in P^{\{[\,],*,//\}}$ be a boolean tree pattern with d descendant edges. A canonical model for p is a tree of the form $s^z(p[\bar{u}])$, for some $\bar{u} = (u_1, \ldots, u_d)$, $u_1 \geq 0, \ldots, u_d \geq 0$. We denote $mod^z(p)$ the set of canonical models:*

$$mod^z(p) \;=\; \{s^z(p[\bar{u}]) \mid \bar{u} = (u_1, \ldots, u_d), 0 \leq u_1, \ldots, 0 \leq u_d\} \tag{2}$$

Also, given a number $n \geq 0$, we define the set of bounded canonical models:

$$mod^z_n(p) \;=\; \{s^z(p[\bar{u}]) \mid \bar{u} = (u_1, \ldots, u_d), 0 \leq u_1 \leq n, \ldots, 0 \leq u_d \leq n\}$$



(a) tree pattern p (b) tree pattern $p[0,2]$ (c) canonical model $s^z(p[0,2])$

Fig. 4. A pattern p, the extension pattern $p[0,2]$, and the canonical model $t = s^z(p[0,2])$.

One should think of a canonical model as some tree t that "looks like" p. The symbol z is just some arbitrary symbol to subsitute $*$, since a tree is not supposed to contain $*$. For an illustration, Figure 4(c) shows the canonical model $s^z(p[0,2])$ for the pattern p in Figure 4(a).

For every canonical model $t = s^z(p[\bar{u}])$, the function $e_t : \text{NODES}(p) \to \text{NODES}(t)$ defined as $e_t(x) = x, \forall x \in \text{NODES}(p)$ is an embedding, which we call the *canonical embedding*. Hence, a canonical model is indeed a model and $mod^z_n(p) \subseteq mod^z(p) \subseteq Mod(p)$. When p has at least one descendant edge, then $mod^z(p)$ is infinite. The set $mod^z_n(p)$ is always finite, for any tree pattern p and any $n \geq 0$. The main property of a canonical model is that, in order to find a witness t for $p \not\subseteq p'$, it suffices to restrict the search to $t \in mod^z_n(p)$, where $z \in \Sigma$ is any symbol that does not occur in p' and n depends only on p'. We show now how to construct this number n from p'.

Define the *star length* of a pattern q to be the largest number w s.t. there exists a sequence of w nodes, $x_1, \ldots, x_w$, labeled with $*$'s and connected by child edges: i.e. $(x_{i-1}, x_i) \in \text{EDGES}_/(q)$, $\forall i = 2, \ldots, w$, and $\text{LABEL}(x_i) = *$, for $i = 1, \ldots, w$.

PROPOSITION 3. *Let p and p' be two boolean tree patterns, $z \in \Sigma$ be a symbol that does not appear in p', and w' be the star length of p'. Then the following are equivalent: (1) $p \subseteq p'$, (2) $mod^z(p) \subseteq Mod(p')$, (3) $mod^z_n(p) \subseteq Mod(p')$, where $n = w' + 1$.*

PROOF. Statement (1) is equivalent to $Mod(p) \subseteq Mod(p')$, hence the implications (1) $\to$ (2) and (2) $\to$ (3), are obvious. We prove now (3) $\to$ (1). Suppose $p \not\subseteq p'$, and let $t \in T_\Sigma$ be a witness, i.e. $p(t)$ is true and $p'(t)$ is false. Since $p(t)$ is true, there exists an embedding $e : p \to t$. It follows from Lemma 1 that there exists an embedding from some $p[\bar{u}]$ to t, $e' : p[\bar{u}] \to t$, which agrees with e on the nodes of p. Consider the canonical model $t_1 = s^z(p[\bar{u}])$; we show that t_1 is still a witness, i.e. $p'(t_1)$ is false. Indeed, suppose $p'(t_1)$ were true. Then there exists an embedding $e_1 : p' \to t_1$, and we define the function $f : \text{NODES}(p') \to \text{NODES}(t)$ by composing $e_1 : p' \to t_1$ with $e' : p[\bar{u}] \to t$: this is possible since $\text{NODES}(t_1) = \text{NODES}(p[\bar{u}])$. We show that f is an embedding, contradicting the fact that $p'(t)$ is false. Clearly f preserves the structure (root, edges) since both e_1 and e' do so, and the structure of t_1 is identical to that of $p[\bar{u}]$. We show that f also preserves the labels. The only case where the labels in $t_1 = s^z(p[\bar{u}])$ and $p[\bar{u}]$ differ is at nodes y that are labeled z in t_1. So let $x \in \text{NODES}(p')$ s.t. the label of $y = e_1(x)$ in t_1 is z. In that case $\text{LABEL}(x) = *$, because z does not occur in p', which implies that f is label preserving at node x. This ends the proof of the fact that $t_1 = s^z(p[\bar{u}]) \in mod^z(p)$ is a witness, i.e. $p(t_1)$ is true while $p'(t_1)$ is false.

We now construct some canonical model $t_2 \in mod^z_n(p)$ that is still a witness. This follows directly from the next lemma.

LEMMA 2. *Let p and p' be two boolean tree patterns, $z \in \Sigma$ be a symbol that does not appear in p', and w' be the star length of p'. Let $t_1 = s^z(p[\bar{u}])$ be a canonical model s.t. $p'(t_1)$ is false. Define $\bar{v} = (v_1, \ldots, v_d)$ to be $v_i = \min(u_i, n)$, for $i = 1, \ldots, d$, where $n = w' + 1$, and $t_2 = s^z(p[\bar{v}])$. Then $p'(t_2)$ is false.*

The intuition for the lemma is that, if $p'(t_2)$ were true, then we can stretch the chains of extra nodes in t_2 to obtain t_1, and we still have $p'(t_1)$ true. This is because the chains we need to stretch from t_2 to t_1 are too long for any chain child-connected of $*$'s in p' to cover them completely, hence p' maps descendant edges to those chains, allowing us to stretch them. A formal proof is given in the appendix. To conclude the proof of Proposition 3 we notice that t_2 is still a witness for $p \not\subseteq p'$ and that $t_2 \in mod^z_n(p)$.

□

PROPOSITION 4. *The following problem is in coNP: given two tree patterns $p, p' \in P^{\{[], *, //\}}$, decide whether $p \subseteq p'$.*

PROOF. This is a consequence of Proposition 3. In order to check $p \not\subseteq p'$ it suffices to guess d numbers $u_1, \ldots, u_d$, each $u_i \leq w' + 1$, where w' is the star length of p', and construct canonical model $t = s^z(p[u_1, \ldots, u_d])$, then check in polynomial time that $p'(t)$ is false. □

Proposition 3 also gives a naive algorithm for checking containment: simply iterate over all $t \in mod^z_{w'+1}(p)$ and check $p'(t)$, which requires $O(|t|\,|p'|)$ steps using Algorithm 1 (Sec. 2.5). Recall that $|p|$ denotes the number of edges in p. We compute now the total running time of this naive algorithm. Given a d-tuple $\bar{u} = (u_1, \ldots, u_d)$, the size of the canonical model $s^z(p[\bar{u}])$ is:

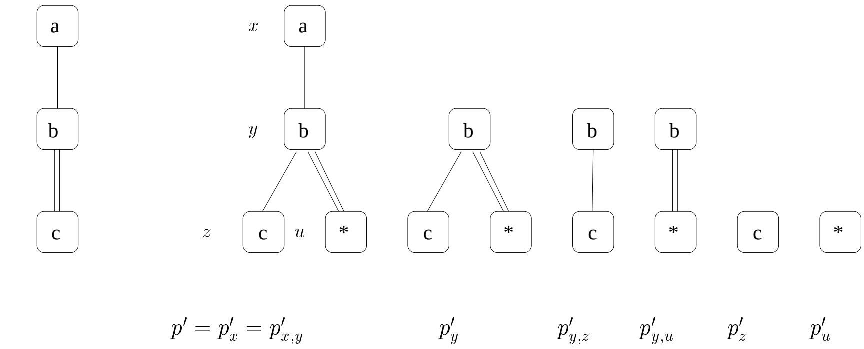
$$|s^z(p[\bar{u}])| \;=\; |p| + u_1 + \ldots + u_d$$

Checking $p'(s^z(p[\bar{u}]))$ thus takes $O(|s^z(p[\bar{u}])| \times |p'|)$ steps, and the total running time of the naive algorithm is:

$$\begin{aligned}
&\sum_{0 \leq u_1 \leq w'+1, \ldots, 0 \leq u_d \leq w'+1} (|p| + u_1 + \ldots + u_d) \times |p'| = \\
&= \; (|p|(w'+2)^d + d(w'+2)^{d-1}\frac{(w'+1)(w'+2)}{2}) \times |p'| \\
&\leq \; |p|(w'+2)^{d+1}|p'|
\end{aligned}$$

We used here the fact that $d \leq |p|$ and $1 + \frac{w'+1}{2} < w' + 2$. Thus, one can decide $p \in p'$ in time $O(|p|\,|p'|(w'+2)^{(d+1)})$.

This naive algorithm is not practical, however, since much of the work in evaluating $p'(t)$ is repeated for various canonical models t.



(a) tree pattern p (b) tree pattern p' and its subpatterns

Fig. 5. A pattern p (a), another tree pattern p' and its seven subpatterns (b).

3.1.1 *An algorithm for checking containment.* We present now an improvement of the naive algorithm for checking containment of two patterns, $p \subseteq p'$, which is complete, and which avoids repeated computations. The algorithm may seem complex, at a first look: the need for this complexity is best appreciated after reading Section 3.2, where a much simpler and efficient algorithm turns out to be incomplete. This section can be skipped at a first reading.

We start with some definitions and notations.

Match Sets Let t be a tree. Each node and each edge in t defines a subtree, as follows. A node $x \in \text{NODES}(t)$ defines the subtree t_x consisting of the node x and all descendants of x; in particular, $\text{ROOT}(t_x) = x$, and $t_{\text{ROOT}(t)} = t$. An edge $(x, y) \in \text{EDGES}(t)$ defines the subtree $t_{x,y}$ consisting of t_y plus the node x and the edge (x, y). We denote $\mathcal{S}(t)$ the set of all subtreess associated to nodes and edges. If $|t|$ denotes the number of edges, then $\mathcal{S}(t)$ has $2|t| + 1$ subtrees. We apply the same definition to patterns p, and denote $\mathcal{S}(p)$ the set of subpatterns. Figure 5 (b) shows a pattern p' with three edges, where the set $\mathcal{S}(p')$ has seven subpatterns.

For a pattern q, denote q^* the pattern obtained by relabeling q's root node with $*$.

We now introduce the notion of a match set, which is adapted from [Hoffmann and O'Donnell 1982]. We fix a tree pattern p' for the remainder of this section. Let $t \in T_\Sigma$ a tree. The *match set*, $ms(t) \in \mathcal{P}(\mathcal{S}(p'))$, is the set defined by:

$$\begin{aligned} ms(t) \;=\; & \{p'_x \mid x \in \text{NODES}(p'), p'_x \in \mathcal{S}(p'), p'_x(t) = \mathit{true}\} \cup \\ & \{p'_{x,y} \mid (x, y) \in \text{EDGES}_{/}(p'), p'_{x,y} \in \mathcal{S}(p'), p'_{x,y}(t) = \mathit{true}\} \\ & \{p'_{x,y} \mid (x, y) \in \text{EDGES}_{//}(p'), p'_{x,y} \in \mathcal{S}(p'), (p'_{x,y})^*(t) = \mathit{true}\} \end{aligned}$$

The definition treats descedant-edge subpatterns slightly differently in that it ignores the label of their root node, for reasons that will become clear below.

For the pattern p' in Fig. 5 (b), consider the linear tree $t_1 = /a/b/c$: then $ms(t_1) = \{p'_x, p'_{x,y}, p'_{y,u}, p'_u\}$. Consider now the linear tree $t_2 = /a/b/z/c$ (here z is another symbol). Then $ms(t_2) = \{p'_{y,u}, p'_u\}$.

$$
\begin{array}{lcl}
\mathit{closeMS}(ms, a) & = & ms \cup \{p'_x \mid (\text{LABEL}(x) = a \vee \text{LABEL}(x) = *), \\
& & \quad \forall (x,y) \in \text{EDGES}(p') \Rightarrow p'_{x,y} \in ms\} \\
\mathit{nodeMS}(ms_1, \ldots, ms_k, a) & = & \mathit{closeMS}(ms_1 \cup \ldots \cup ms_k, a) \\
\mathit{edgeMS}(ms, a) & = & \mathit{closeMS}(\{p'_{x,y} \mid (x,y) \in \text{EDGES}_{/}(p'), p'_y \in ms, \text{LABEL}(x) = a\} \cup \\
& & \quad \{p'_{x,y} \mid (x,y) \in \text{EDGES}_{//}(p'), p'_y \in ms\} \cup \\
& & \quad \{p'_{x,y} \mid (x,y) \in \text{EDGES}_{//}(p'), p'_{x,y} \in ms\}, a) \\
\mathit{inflateMS}(ms) & = & \{ms' \mid ms' = \underbrace{\mathit{edgeMS}(\ldots \mathit{edgeMS}(ms, *), \ldots, *)}_{k \text{ times}}, 0 \leq k \leq w' + 1\}
\end{array}
$$

Fig. 6. Auxiliary functions used in Algorithm 2.

The idea in Algorithm 2 is that $p \not\subseteq p'$ iff there exists a canonical tree $t \in mod^z(p)$ such that $p' = p'_{\text{ROOT}(p')} \notin ms(t)$. So, our goal is to compute the set of all match sets, $\mathcal{MS}[p] = \{ms(t) \mid t \in mod^z(p)\}$, then check the condition $\forall ms \in \mathcal{MS}[p], p'_{\text{ROOT}(p')} \in ms$: this is done in lines 13-16 of Algorithm 2.

For an illustration, consider the patterns p and p' in Fig. 5. We have $\mathcal{MS}[p] = \{\{p'_x, p'_{x,y}, p'_{y,u}, p'_u\}, \{p'_{y,u}, p'_u\}\}$. Indeed, t_1 and t_2 illustrated earlier are canonical trees for p, hence both $ms(t_1)$ and $ms(t_2)$ are in $\mathcal{MS}[p]$. Moreover, any other canonical tree is of the form $t = a/b/z/\ldots/z/c$, i.e. has at least 2 z's, and $m(t) = m(t_2)$.

Notice that, while a match set ms is an element of $\mathcal{P}(\mathcal{S}(p'))$, the set of matchsets $\mathcal{MS}[p]$ is an element of $\mathcal{P}(\mathcal{P}(\mathcal{S}(p')))$. $\mathcal{MS}[p]$ has at most as many elements as canonical trees in $mod^z_{w'+1}(p)$, where w' is the star length of p'. This follows from Prop. 3. But $\mathcal{MS}[p]$ may be much smaller than $mod^z_{w'+1}(p)$, because many canonical trees t may produce the same match set.

By focusing on match sets rather than canonical trees we avoid the repeated computations in the naive algorithm. So all we need is to explain how $\mathcal{MS}[p]$ is computed in the first part of Algorithm 2. In order to do that we need to examine how to compute simple match sets first.

Computing Match Sets To compute $ms(t)$, one can proceed inductively, by computing $ms(t_u)$ and $ms(t_{u,v})$ for all node- and edge-subtrees of t. The functions we need for that are shown in Fig. 6. We explain them here.

First, suppose we have computed the set $ms \subseteq \mathcal{S}(p')$ consisting of the edge subpatterns matching t (i.e. the second line in the definition of $ms(t)$). Then we can find the node subpatterns matching t by computing:

$$ms(t) \;=\; \mathit{closeMS}(ms, \text{LABEL}(\text{ROOT}(t)))$$

where *closeMS* is in Fig. 6. That is, it suffices to add all node subpatterns p'_x which match the root label in t and for which all outgoing child-subpatterns $p'_{x,y}$ are in ms. Notice that, if $ms = \emptyset$ then $\mathit{closeMS}(ms, a)$ returns the set of node-subpatterns p'_x for which x is a leaf and $\text{LABEL}(x)$ matches a.

Now consider a node u of t, and assume we want to compute $ms(t_u)$. Let $v_1, \ldots, v_k$ be all children of u. Then:

$$ms(t_u) \;=\; \mathit{nodeMS}(ms(t_{u,v_1}), \ldots, ms(t_{u,v_k}), \text{LABEL}(u))$$

where *nodeMS* is shown in Fig. 6.

Similarly, lets compute $ms(t_{u,v})$ from $ms(t_v)$:

$$ms(t_{u,v}) \;=\; \mathit{edgeMS}(ms(t_v), \text{LABEL}(u))$$

where *edgeMS* is shown in Fig. 6. This function requires some discussion. The first line should be clear: we just move one edge up, in all subpatterns in ms. The second line is justified as follows. If (x, y) is a descendant edge and $p'_{x,y}$ is in $ms(t_v)$, then it should also be in $ms(t_u)$. To enable this simple inductive definition we had to ignore the label on the root node of $p'_{x,y}$: that label will be checked later, in *closeMS*.

Computing Sets of Match Sets We can now describe how to compute $\mathcal{MS}[p]$. To the previous inductive computations we only need to add an inductive computation for a descendant edge in p. Recall that such an edge is replaced by a sequence of up to $w' + 1$ symbols z. This justifies the function *inflateMS*(ms) in Fig. 6. The function applies *edgeMS*$(-, *)$ repeatedly, thus simulating the effect of an edge labeled z (z and $*$ are interchangeable here). Given one match set ms, *inflateMS*(ms) returns a set of $w' + 2$ match sets. Notice that $ms \in$ *inflateMS*(ms).

To compute $\mathcal{MS}[p]$ we will compute inductively $\mathcal{MS}[q]$ for all node- and edge-subpatterns q of p. For a node u in p, we denote $v_1, \ldots, v_k$ its children.

$$\begin{aligned}
\mathcal{MS}[p_u] &= \{\mathit{nodeMS}(ms_1, \ldots, ms_k, \text{LABEL}(u)) \mid ms_1 \in \mathcal{MS}[p_{u,v_1}], \ldots, ms_k \in \mathcal{MS}[p_{u,v_k}]\} \\
\mathcal{MS}[p_{u,v}] &= \{\mathit{edgeMS}(ms, \text{LABEL}(u)) \mid ms \in \mathcal{MS}[p_v]\} \\
&\qquad \text{when } (u,v) \in \text{EDGES}_{/}(p) \\
\mathcal{MS}[p_{u,v}] &= \{\mathit{edgeMS}(ms, \text{LABEL}(u)) \mid ms_0 \in \mathcal{MS}[p_v], ms \in \mathit{inflateMS}(ms_0)\} \\
&\qquad \text{when } (u,v) \in \text{EDGES}_{//}(p)
\end{aligned}$$

The Containment Algorithm This justifies now the Containment Algorithm 2.

Algorithm 2 Check if $p \subseteq p'$

---

for u in NODES(p), (u, v) in EDGES(p) **do** {The iteration proceeds bottom up on nodes u and edges (u, v) of p}

    To process a node u:
    let $a = \text{LABEL}(u)$, $v_1, \ldots, v_k = \text{children}(v)$
    compute $\mathcal{MS}[p_u] = \{\mathit{nodeMS}(ms_1, \ldots, ms_k, a) \mid ms_1 \in \mathcal{MS}[p_{u,v_1}], \ldots, ms_k \in \mathcal{MS}[p_{u,v_k}]\}$

    To process an edge (u, v):
    let $a = \text{LABEL}(u)$, and $MS = \mathcal{MS}[p_v]$
    if $(u, v) \in \text{EDGES}_{//}(p)$ **then**
        let $MS = \bigcup\{\mathit{inflateMS}(ms) \mid ms \in MS\}$
    compute $\mathcal{MS}[p_{u,v}] = \{\mathit{edgeMS}(ms, a) \mid ms \in MS\}$

for $ms \in \mathcal{MS}[p_{\text{ROOT}(p)}]$ **do**
    if $p'_{\text{ROOT}(p')} \notin ms$ **then**
        return false
return true

We compute now the running time of Algorithm 2. A subpattern in $\mathcal{S}(p')$ can be represented as either a node or an edge. We assume that nodes and edges in p' have unique identifiers, e.g. an integer in the range $0, 1, \ldots, 2|p'|$. A match set, $ms \in \mathcal{P}(\mathcal{S}(p'))$ can be represented as bitmap of length $2|p'| + 1$. A union, $ms_1 \cup ms_2$, or an equality test, $ms_1 = ms_2$ can thus be performed in $O(|p'|)$ time. A set of matchsets, $MS \in \mathcal{P}(\mathcal{P}(\mathcal{S}(p')))$, is represented as a trie. Insertions, and membership tests also take $O(|p'|)$. To compute the running time of the algorithm it is important to notice that there exists a many-to-one correspondence between canonical databases in $mod^z_{w'+1}(q)$ and match sets in $\mathcal{MS}[q]$, for any supattern q of p. In particular, $|\mathcal{MS}[q]| \leq |mod^z_{w'+1}(q)| \leq |mod^z_{w'+1}(p)| = (w' + 2)^d$, where d is the number of descendant edges in p. It follows that the two computation of entries in $\mathcal{MS}$, i.e. lines 5 and 11 in the algorithm, take $O(|p'|(w' + 2)^d)$ time. For example, in line 5 we notice that every match set $ms_1, \ldots, ms_k$ corresponds to some canonical tree $t_1 \in mod^z_{w'+1}(p_{u,v_1}), \ldots, t_k \in mod^z_{w'+1}(p_{u,v_k})$. In turn, the k-tuple $(t_1, \ldots, t_k)$ corresponds to a canonical tree $t \in mod^z_{w'+1}(p_u)$, hence the total number of steps done in line 5 is no more than $|mod^z_{w'+1}(p_u)|$; moreover, each takes $O(|p'|)$ time.

Thus we have:

THEOREM 1. *Algorithm 2 checks containment of two tree patterns p, p'. It runs in time $O(|p||p'|(w' + 2)^d)$.*

Special cases While this is only marginally better than the naive algorithm discussed earlier, it can be much better in practice, because the number of matchsets in $\mathcal{MS}[q]$ is often much smaller than the number of canonical trees for q. We discuss here some special cases when the running time is better.

First, consider the case when p has no descendant edges, i.e. $d = 0$. While Theorem 1 already gives us a running time of $O(|p||p'|)$, it is instructive to see how this happens exactly. In this case all entries $\mathcal{MS}[p_u]$ and $\mathcal{MS}[p_{u,v}]$ contain a single match set. This is because the function *inflateMS* is never called, and this (line 10) is the only place in the algorithm where we may generate more than one match set in $\mathcal{MS}[-]$. Thus, $\mathcal{MS}[-]$ can be viewed as a relation from the nodes and edges in p to the nodes and edges in p'. It is interesting to see the anaology to Algorithm 1, which computes an embedding from a pattern to a tree, by computing two relations $\mathcal{C}[-,-]$ and $\mathcal{D}[-,-]$. For two nodes u, x we have $p'_x \in ms \in \mathcal{MS}[p_u]$ iff $C[u, x]$ is true, and for a descendant edge $(x, y) \in \text{EDGES}_{//}(p')$ we have $p'_{x,y} \in ms \in \mathcal{MS}[p_u]$ iff u has some child v s.t. $\mathcal{D}[v, y]$ is true. Thus, in this special case Algorithm 2 degenerates to Algorithm 1.

Next, consider another simple case, when every symbol in p' occurs only once. That is, if x, y are two nodes in p', then $\text{LABEL}(x) \neq \text{LABEL}(y)$; we assume for the moment that p' does not contain $*$. Let u be a node in p and $a = \text{LABEL}(u)$. Then any match set $ms \in \mathcal{MS}[p_u]$ may contain at most one node-subpattern p'_x, namely that one for which $\text{LABEL}(x) = a$, and at most d' edge-subpatterns $p'_{x,y}$, namely the descendant edges in p', where $d' = |\text{EDGES}_{//}(p')|$. Thus, there are at most $2^{d'+1}$ different matchsets that can be included in $\mathcal{MS}[p_u]$. The running time is then $O(|p||p'|2^{k_{max}(d'+1)})$, where k_{max} is the maximum outdegree of a node in p. This can be seen by examining line 5 of Algorithm 2, which iterates over $\leq k_{max}$ sets, each with $\leq 2^{d'+1}$ elements.

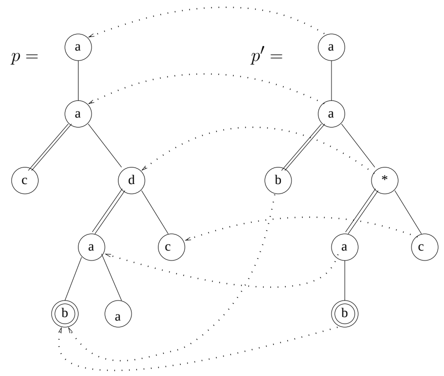


Fig. 7. Two tree patterns p, p' and a homomorphism from p' to p, proving $p \subseteq p'$.

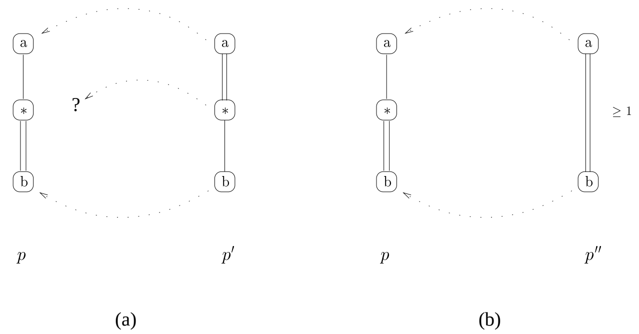


Fig. 8. (a) Two equivalent queries p, p' with no homomorphism from p' to p; (b) same queries represented differently, and a homomorphism between them.

3.2 Checking Containment with Pattern Homomorphisms

The second technique that we use to reason about containment is a homomorphism between patterns. As a first attempt, let us define a homomorphism $h : p' \to p$ to be a function from $\text{NODES}(p')$ to $\text{NODES}(p)$ that satisfies the definition of an embedding (given in Sec. 2.1), with the following strengthening of the child-edge preservation condition: if the edge (x, y) is in $\text{EDGES}_/(p')$ then $(h(x), h(y))$ must be in $\text{EDGES}_/(p)$. Figure 7 illustrates such a homomorphism. We will show that, given two patterns p, p', one can determine in time $O(|p||p'|)$ whether a homomorphism $p' \to p$ exists. Moreover, if it exists, then we can show that $p \subseteq p'$. Thus, an efficient practical algorithm for checking containment is to search for a homomorphism. However, this algorithm is not always complete. The challenge is to define the homomorphism to be powerful enough to guarantee that the algorithm is complete in non-trivial special cases.

Motivation for adornment The problem with the naive definition for a homomorphism above is that it fails to be a necessary criterion for patterns in $P^{\{*,//\}}$ (i.e. no branches). This was already observed in [Milo and Suciu 1999], and is illustrated here in Figure 8 (a). The two tree patterns here correspond to the

XPath expressions p=`a/*//b`, p'=`a//*/b`. Although p, p' are equivalent, there is no homomorphism from p' to p because there is no destination for the wildcard in p': for example we cannot map $*$ to $*$ because then the child edge $(*, b)$ would be mapped into a descendant edge. The solution is to eliminate the $*$ node and *adorn* the descendant edge with "≥ 1", meaning that there is at least one intermediate node on this paths. This is shown in Fig. 8 (b), which also illustrates a homomorphism from the adorned tree pattern. When introducing the adornments, every descendant edge is initially adorned with ≥ 0, then adjacent edges sharing a $*$ node are combined into descendant edge with a higher adornment. Only $*$ nodes that have a unique child may be eliminated this way, i.e. if a $*$ node has two or more outgoing edges then we cannot eliminate it. This process can be described as a a set of rewrite rules, using XPath-like syntax:

$$\begin{aligned} // &\rightarrow //^{\geq 0} \\ //^{\geq m} * / &\rightarrow //^{\geq m+1} \\ / * //^{\geq n} &\rightarrow //^{\geq n+1} \\ //^{\geq m} * //^{\geq n} &\rightarrow //^{\geq m+n+1} \end{aligned} \tag{3}$$

For example, $p' = \texttt{a//*/*/b/*/c//d}$ is rewritten to $p'' = \texttt{a//}^{\geq 2}\ \texttt{b/*/c//}^{\geq 0}\texttt{d}$. Now the homomorphism is defined from the adorned pattern p'' to p, and its existence is shown in [Milo and Suciu 1999] to be a necessary and sufficient condition for containment when both patterns are linear. To illustrate on our example in Figure 8, the pattern p' reduces to the adorned pattern p'' in (b), making a homomorphism possible.

Adorned patterns and homomorphisms We now define formally adorned tree patterns and homomorphisms between adorned tree patterns. An *adorned tree pattern* is a tree pattern p with an adornment function $\alpha : \text{EDGES}_{//}(p) \rightarrow N$. Using the notations in Sec. 2, given two nodes $(x, y) \in \text{EDGES}^*(p)$ we define their distance, $d(x, y)$, as follows:

$$\begin{aligned} d(x,x) &= 0 & \\ d(x,y) &= 1 & \text{if } (x,y) \in \text{EDGES}_{/}(p) \\ d(x,y) &= 1 + \alpha(x,y) & \text{if } (x,y) \in \text{EDGES}_{//}(p) \\ d(x,y) &= d(x,z) + d(z,y) & \text{if } (x,z) \in \text{EDGES}(p), (z,y) \in \text{EDGES}^{+}(p) \end{aligned}$$

DEFINITION 2. *A* homomorphism $h : p' \rightarrow p$ *is a function* $h : \text{NODES}(p') \rightarrow \text{NODES}(p)$ *satisfying the following four conditions:*

(1) $h(\text{ROOT}(p')) = \text{ROOT}(p)$,
(2) *if* $x \in \text{NODES}(p')$, *then* $\text{LABEL}(x) = *$ *or* $\text{LABEL}(x) = \text{LABEL}(h(x))$,
(3) *if* $(x, y) \in \text{EDGES}_{/}(p')$, *then* $(h(x), h(y)) \in \text{EDGES}_{/}(p)$, *and*
(4) *if* $(x, y) \in \text{EDGES}_{//}(p')$ *then* $(h(x), h(y)) \in \text{EDGES}^{+}(p)$ *and* $1 + \alpha(x, y) = d(x, y) \leq d(h(x), h(y))$.

Fig. 7 and Fig. 8 (b) show two examples of homomorphisms.

Every (unadorned) tree pattern p admits a trivial adornment by setting $\alpha(x, y) = 0$, for every $(x, y) \in \text{EDGES}_{//}(p)$, hence our discussion in the remainder of this section also applies to unadorned patterns.

Given a tree t and an adorned pattern p, an embedding $e : p \to t$ is defined to be a homomorphism from p to t, where t is viewed as a pattern. Obviously, this coincides with the definition of an embedding in Sec. 2.1 when p is unadorned.

Computing a homomorphism Algorithm 3 takes two adorned tree patterns p, p' and checks if there exists a homomorphism from p' to p. The algorithm runs in time $O(|\, p \,||\, p' \,|)$, and generalizes the algorithm for finding an embedding (presented in Sec. 2). It proceeds bottom up in both p and p', and computes two tables $\mathcal{C}(x, y)$ and $\mathcal{D}'(x, y)$ with $x \in \text{NODES}(p), y \in \text{NODES}(p')$. The meaning of these tables is the following: $\mathcal{C}(x, y)$ is a boolean value denoting whether there exists a homomorphism from the subpattern rooted at y to the subpattern rooted at x. $\mathcal{D}'(x, y)$ is defined to be $\max\{d(x, x') \mid (x, x') \in \text{EDGES}^*(p) \wedge \mathcal{C}(x', y) = \mathit{true}\}$. We take maximum of an empty set to be $-\infty$, hence $\mathcal{D}'(x, y)$ is either ≥ 0, or $-\infty$. The algorithm computes both tables bottom up in an obvious way.

Algorithm 3 Find homomorphism $p' \to p$

1: **for** x in $\text{NODES}(p)$ **do** {The iteration proceeds bottom up on nodes of p}
2:   **for** y in $\text{NODES}(p')$ **do** {The iteration proceeds bottom up on nodes of p'}
3:     **compute** $\mathcal{C}(x, y) = (\text{LABEL}(y) = \text{“}*\text{''} \vee \text{LABEL}(x) = \text{LABEL}(y)) \wedge$
4: $\qquad\qquad \bigwedge_{(y,y') \in \text{EDGES}_{/}(p')} (\bigvee_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{C}(x', y')) \wedge$
5: $\qquad\qquad \bigwedge_{(y,y') \in \text{EDGES}_{//}(p')} (\mathcal{D}'(x, y') \geq 1 + \alpha(y, y'))$
6:     **if** $\mathcal{C}(x, y)$ **then**
7:       $d = 0$;
8:     **else**
9:       $d = -\infty$
10:     **compute** $\mathcal{D}'(x, y) = \max(d, 1 + \max_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{D}'(x', y),$
11: $\qquad\qquad 1 + \max_{(x,x') \in \text{EDGES}_{//}(p)} (\alpha(x, x') + \mathcal{D}'(x', y)))$
12: **return** $\mathcal{C}(\text{ROOT}(p), \text{ROOT}(p'))$

PROPOSITION 5. *Algorithm 3 decides whether there exists a homomorphism from p' to p and runs in time $O(|p||p'|)$.*

PROOF. The proof for the running time is identical to that for the Algorithm 1. The inner loops are the test $\mathcal{C}(x', y')$ in line 4, the test $\mathcal{D}'(x, y') \geq \ldots$ in line 5, and the computation of $\mathcal{D}'(x', y)$ in lines 10 and 11. These are executed at most once for every pair of edges, in p and in p' respectively, hence the total number of steps is $O(|p||p'|)$. $\square$

Checking containment We now turn to our main question: checking containment of two patterns, by using homomorphisms. The main result is that a homomorphism is a sufficient condition, and that in certain cases it is also a necessary condition for containment.

The algorithm for checking containment in shown in Fig. 9.

Step 1 modifies p and p' by adding a shadow leaf to each leaf and labeling it with some symbol $a \in \Sigma$: more precisely, for each leaf node x in p (and similarly in p') create a shadow leaf x', label it with the symbol $a \in \Sigma$ and insert a new edge (x, x') into $\text{EDGES}_{//}(p)$ (or $\text{EDGES}_{//}(p')$ respectively). There are no restrictions on

Algorithm CheckContainment I
 Input: p, p'
 Output: true if $p \subseteq p'$
 Method
  Step 1 Add shadow leaf symbols to p and p'
   connect them with descendant edges to the old leaves
   and label all of them with the same symbol $a \in \Sigma$
  Step 2 Apply the re-writings (3) to p' repeatedly, until it reduces to p''
  Step 3 Find homomorphism from p'' to p
  if found **then** return **true**
   else return **false**

Fig. 9. A sound algorithm for checking containment $p \subseteq p'$.

the choice of the symbol a except that all shadow leaves must be labeled with the same symbol. This step at most doubles the size of p and p', and preserves the containment relationship between p and p'. The purpose of this step is to remove dangling tails in p'. For example, consider $p = b/*//c$ and $p' = b//*$. Here $p \subseteq p'$ (recall that they are boolean patterns), but no homomorphism exists from p' to p: the re-writings in Step 2 are useless here because the dangling $*$ in p' has no outgoing edge, hence cannot be eliminated. However, by transforming both p and p' as in Step 1 we get $p = b/*//c//a$ $p' = b//*//a$, and now p' reduces to $p'' = b//^{\geq 1}a$, and there exists an obvious homomorphism from p'' to p.

Step 2 applies repeatedly the reductions (3) to p'. In effect this step replaces a chain of nodes $x_1, x_2, \ldots, x_n$ in p' with a single descendant edge (x_1, x_n) and adorns it such that the distance $d(x_1, x_n)$ is preserved, provided that all nodes $x_2, \ldots, x_{n-1}$ are labeled with $*$, each node x_{i-1} has only one outgoing edge, namely to x_i, for $i = 2, \ldots, n$, and there exists at least one descendant edge among $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)$. We have seen in Figure 8 that, without this step, the algorithm would be incomplete on linear patterns.

Step 3 checks for a homomorphism from p'' to p and reports the result.

We prove now the main properties of this algorithm. The soundness and the running time follow immediately from our discussions above:

THEOREM 2. *Algorithm 3 is sound (i.e. if it returns* **true** *then $p \subseteq p'$) and runs in time $O(|p||p'|)$.*

In general the algorithm is incomplete, as shown by the following example.

EXAMPLE 1. *This example illustrates two patterns p, p' s.t. $p \subseteq p'$ but no homomorphism exists from p' to p. In particular, the algorithm in Fig. 9 returns* **false**, *although $p \subseteq p'$. The two patterns are shown in Figure 10. Pattern p' reduces to a pattern p'' where the right branch has the adornment "≥ 1"; p'' is not shown. Intuitively, the three branches in p and the two branches in p' impose certain conditions on the number of intermediate nodes between the c node and the d node: it should be either $= 0$, or ≥ 0 (i.e. no condition), or ≥ 1: this is noted under each branch. There is no homomorphism from p' to p (and neither from p'' to p). Since the two*

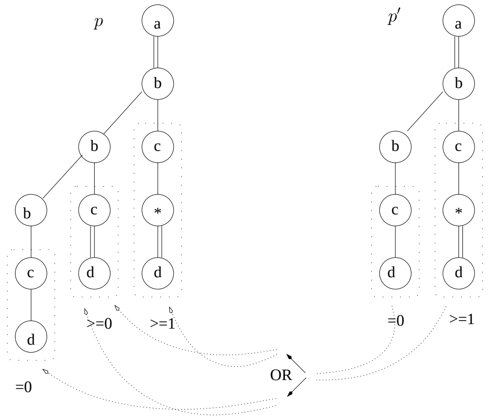


Fig. 10. Two patterns s.t. $p \subseteq p'$ but where no homomorphism exists from p' to p.

b nodes in p' are connected with a child edge, a homomorphism can map the two branches either to the first two, or the last two branches in p: but this is not possible since none of the branches in p' can be mapped to the middle branch. (The same argument applies to the reduced pattern, p'', and we omit it.) However, $p \subseteq p'$, as we show next. Let $t \in mod^z(p)$ and consider the middle branch in t. If the there are no nodes between c and d then we can embed p' to t by mapping it to the second and third branch in t. If there is at least one node between c and d then we embed p' to t by mapping it to the first and second branch. Thus, to check containment we need to "reason by cases", and the homomorphism is an incomplete test.

Thus, the algorithm is incomplete in general, but we prove next that it is complete in four important special cases. The first three are rather simple, and essentially show that the algorithm unifies and generalizes techniques from [Yannakakis 1981; Wood 2001; Amer-Yahia et al. 2001]. The fourth case is non-trivial, and is a significant generalization of the result in [Milo and Suciu 1999].

THEOREM 3. *Algorithm 3 for checking containment $p \subseteq p'$ is complete in each of the following cases:*

(1) $p \in P^{\{[],*\}}$, *or*
(2) $p' \in P^{\{[],*\}}$, *or*
(3) $p' \in P^{\{[],//\}}$, *or*
(4) $p' \in P^{\{*,//\}}$.

Clearly, as the previous example shows, the theorem cannot be generalized further in any significant way. Before giving the proof, we discuss the method used. All four cases are proven as follows: assuming no homomorphism exists from p' to p, we construct some canonical "witness" $t \in mod^z(p)$ s.t. there is no embedding from p' to t, proving that $p \not\subseteq p'$. The first three cases in the theorem are rather easy, since the witness is obtained in a generic way. The last case is the most interesting and difficult one, because when $p' \in \mathrm{P}^{\{*,//\}}$ then the witness needs to be obtained by extending some descendant edges to long chains, while others to short chains, and the exact choice depends on p' in a subtle way. This is illustrated by the example below.

EXAMPLE 2. *This example illustrates two linear patterns $p, p' \in P^{\{*,//\}}$ s.t. there exists no homomorphism $p' \to p$, but the witness $t \in mod^z(p)$ for which $p'(t)$ is false can only be obtained in a complex way. Consider:*

$$p = a/b/s//c/b/s/c//d$$
$$p' = a//b/*/c//*/d$$

Here p has two descendant edges, denote them r_1 and r_2. There is no homomorphism from p'' ($= a//^{\geq 0}b/*/c//^{\geq 1}d$) to p, and a witness $t \in mod^z(p)$ is obtained by taking $\bar{u} = (1, 0)$; that is, by some abuse of notation, the witness is:

$$t = s^z(p[\bar{u}]) = a/b/s/z/c/b/s/c/d$$

One can verify that there exists no embedding from p' to t. But the two obvious choices for canonical models for p, when all chains are short or all chains are long, do not serve as witness. If we take all chains to be short, $\bar{u} = (0, 0)$, then:

$$t = a/b/s/c/b/s/c/d$$

and $p'(t)$ is true: the embedding $p' \to t$ maps the b node in p' to the first b in t. If we make all chains long, say $\bar{u} = (1, 1)$, then

$$t = a/b/s/z/c/b/s/c/z/d$$

and $p'(t)$ is also true: the embedding maps the b node in p' to the second b in t. In fact, it is easy to see that the only witnesses are of the form $\bar{u} = (k, 0)$, for $k \geq 1$, i.e. the first chain must be long and the second must be short.

PROOF. (Of Theorem 3) We assume that p, p' have been processed according to Step 1 of the algorithm, and p' has been reduced to p'' (Step 2). To simplify the discussion we assume that p is unadorned, i.e. $\forall e \in \text{EDGES}_{//}(p)$, $\alpha(e) = 0$: the extension to adorned patterns is trivial, and not really important for us, since the main purpose of the adornments is to be used in p', not in p.

(1) Let $p \in \text{P}^{\{[\,],*\}}$. In this case there is a single canonical model for p, $mod^z(p) = \{t\}$, which is isomorphic to p except that every $*$ is replaced with z. Pick t to be the witness. Any embedding $e : p'' \to t$ immediately yields a homomorphism $p'' \to p$. Steps 1 and 2 are not need for the algorithm to be complete in this case.

(2) Let $p' \in \text{P}^{\{[\,],*\}}$. Step 1 is needed here, so assume that all leaves in p' are labeled with symbols in Σ, not with $*$. In this case no reductions are possible, hence $p'' = p'$. Let w' be the star length of p' (i.e. longest sequence of $*$'s, Sec. 3.1), and let $n = w' + 1$. Define $\bar{u} = (n, n, \ldots, n)$, and choose as witness the canonical database $t = s^z(p[\bar{u}])$. Let $e : p' \to t$ be an embedding. We observe that none of the extension nodes in t is in the image of e. Suppose it were, i.e. there are nodes $y \in \text{NODES}(p')$ and $x = e(y) \in \text{NODES}(t)$ s.t. $\text{LABEL}(x) = z$ and x is an extension node, hence belongs to a chain of n extension nodes. Clearly $\text{LABEL}(y) = *$, and going in both directions up and down from y we must find nodes y', y'' s.t. $\text{LABEL}(y') \neq *$, $\text{LABEL}(y'') \neq *$, and

$d(y', y'') \le w' + 1 = n$, because the star length of p' is w'. But this implies that $d(e(y'), e(y'')) = d(y', y'') \le n$ and none of $e(y')$ and $e(y'')$ is labeled z: this contradicts the fact that $e(y)$ is part of a chain of n consecutive z's.

(3) Let $p' \in \mathrm{P}^{\{[],//\}}$. This case is rather similar to the previous one. Here too $p'' = p'$. Define $\bar{1} = (1, 1, \ldots, 1)$, and let the witness be $t = s^z(p[\bar{1}])$. Let $e : p' \to t$ be an embedding. None of the nodes in p' is mapped to any z symbol, hence only descendant edges in p' can be mapped over the z's. It follows that e is also a homomorphism from p' to p. Steps 1 and 2 are not needed for the algorithm to be complete in this case.

(4) Let $p' \in \mathrm{P}^{\{*,//\}}$. In this case we need Steps 1 and 2 in order for the algorithm to be complete, so we assume that p' has been reduced to p''. We will further assume here, w.l.o.g., that the root nodes in both p and p'' are labeled with a symbol $a \in \Sigma$ that does not occur anywhere else in p or p'': otherwise, we replace p and p'' with a/p and a/p'' respectively, where a is a fresh symbol in Σ. Given that p'' is a linear pattern, it follows that it has a special structure that we describe next. We need the following:

DEFINITION 3. *A* block *is a linear pattern $b \in P^{\{*\}}$ (i.e. with no branches and no descendant edges), where the first and last nodes are labeled with symbols in Σ. That is,* $\mathrm{NODES}(b) = \{x_0, x_1, \ldots, x_n\}$, $\mathrm{EDGES}_/(b) = \{(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)\}$, $\mathrm{EDGES}_{//}(b) = \emptyset$, *and* $\mathrm{LABEL}(x_0), \mathrm{LABEL}(x_n) \in \Sigma$. *The size of the block b is n (the number of edges).*

The special structure of p'' is that it consists of a sequence of blocks, in which every two consecutive blocks are connected by a descendant edge. That is,

$$p'' = b_0//^{\ge k_1} b_1//^{\ge k_2} \ldots //^{\ge k_m} b_m \tag{4}$$

where $b_0, \ldots, b_m$ are blocks. Indeed, Step 1 plus our assumption about the root nodes ensures that the first and last node in p'' are labeled with symbols in Σ. Step 2 ensures that any descendant edge connects two nodes that are labeled with symbols in Σ.

For the proof of this case it is convenient to consider homomorphisms (and embeddings) that do not necessarily map the root node to the root node, but otherwise satisfy all the conditions of the homomorphism. In other words they are defined by conditions (2), (3), (4) of the homomorphism in Definition 2. We call them *un-rooted homomorphism*, and *un-rooted embedding* respectively. Then we prove the following Proposition:

PROPOSITION 6. *Suppose that there exists no un-rooted homomorphism from p'' to p. Then there exists a "witness", i.e. a tree $t \in mod^z(p)$ s.t. there exists no un-rooted embedding from p'' to t.*

We give here the intuition behind the proposition, and defer a formal proof to the appendix. Referring to the structure of p'' given by Equation (4), we attempt to find an un-rooted homomorphism from b_0 to p, searching p top-down, along all its branches, starting at the root. All descendant edges that we traverse while searching for this un-rooted homomorphism we expand to a long chain, i.e. take $u_i = n$, where n is the size of b_0. Continuing from where we managed to map b_0 to p, we attempt to map the descendant edge $//^{\geq k_1}$, along all possible branches in p: all descendant edges in p that we traverse this way we expand to short chains, i.e. $u_i = 0$. Finally, we proceed recursively, for the pattern $b_1//^{\geq k_2} \ldots //^{k_m} b_m$, along all branches in p where we succeeded in mapping both b_0 and $//^{\geq k_1}$. The formal argument is given in the appendix.

Finally we can complete the proof of Theorem 3. Suppose that for every tree $t \in mod^z(p)$ there exists an embedding $e : p'' \to t$; in particular e is also an un-rooted embedding. Hence, by Proposition 6, there exists an un-rooted homomorphism $h : p'' \to p$. Since both p'' and p have their roots labeled with some symbol $a \in \Sigma$ that does not occur elsewhere in p, h must be a (rooted) homomorphism.

□

4. CONP HARDNESS OF CONTAINMENT

We prove here that the containment problem for two tree patterns $p, p' \in P^{\{[],*,//\}}$ is coNP hard. This is in sharp contrast with the fact that containment is in PTIME for each of the three restricted classes $P^{\{[],*\}}, P^{\{[],//\}}, P^{\{*,//\}}$, hence we ask whether containment remains in PTIME if we impose some arbitrary bound on the number of occurrences of descendant edges, or wildcards, or branches. We know already from Theorem 1 that the answer is positive for descendant edges: for any $d \geq 0$, the containment problem $p \subseteq p'$ is in PTIME, where p has at most d descendant edges. We prove here that the answer is negative for the other two. Containment remains coNP hard even when we allow at most 2 wildcards, and, similarly, remains coNP hard even if we allow at most 5 branches in p and at most 3 branches in p'.

Technically, the first coNP hardness theorem is subsumed by any of the following two. We include it here, however, for the sake of the proof technique, which is simpler than that of the other two theorems.

4.1 Main coNP-Hardness

We start with a preliminary result, which is of independent interest. 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, $p(t) \to p_1(t) \vee p_2(t) \vee \cdots \vee p_k(t)$.

LEMMA 3. *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 11. 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

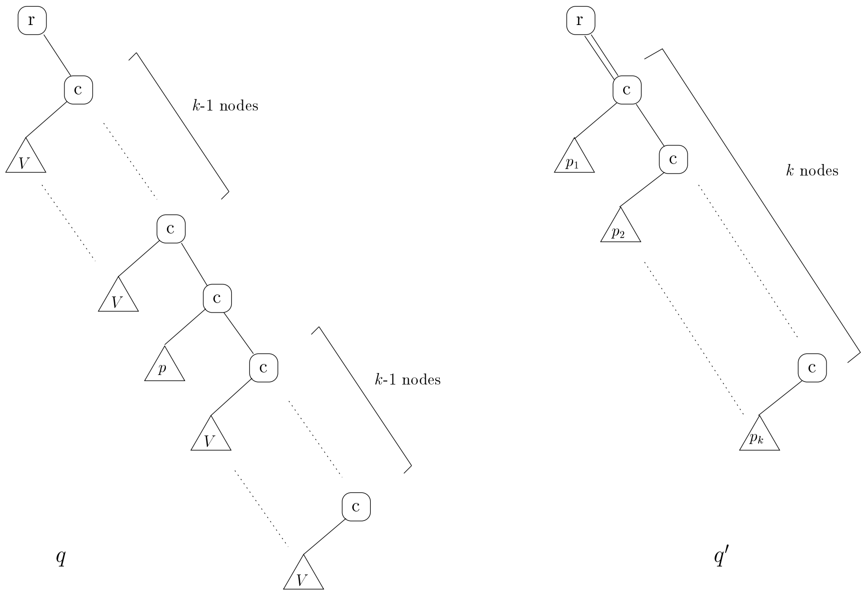


Fig. 11. Patterns q and q' from Lemma 3, constructed from $p, p_1, p_2, \ldots\ p_k$ so that $q \subseteq q'$ if and only if $p \subseteq p_1 \cup \cdots \cup p_k$

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 $t \in mod^z(q)$ we denote $t_p \in mod^z(p)$ the subtree corresponding to p (see Fig. 11).

We assume first that $p \subseteq p_1 \cup \cdots \cup p_k$, and show that for every $t \in mod^z(q)$, we have $q'(t)$ is true, which proves $q \subseteq q'$. Given $t \in mod^z(q)$, 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 the following embedding $e : q' \to t$: e maps the subpattern p_i to t_p (this is possible since $p_i(t_p)$ is true); 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, e maps the root of q' to the root of t.

Conversely, we assume $q \subseteq q'$ and show that $\forall t_p \in mod^z(p)$, $p_1(t_p) \vee \cdots \vee p_k(t_p)$: one can show that the latter implies $p \subseteq p_1 \cup \cdots \cup p_k$, using with the same argument as in Prop. 3. Let $t_p \in mod^z(p)$, and denote with t its extension to a tree $t \in mod^z(q)$, by adding the spine and $k-1$ copies of V above and below t_p in an obvious way. Since $q(t)$ is true we have $q'(t)$ also true, hence there exists an embedding $e : q' \to t$. 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

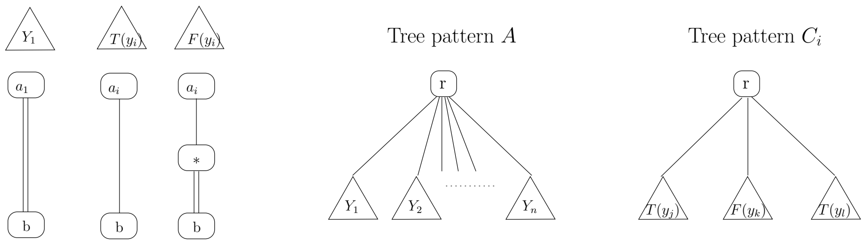


Fig. 12. The canonical models of A encode truth assignments to the literals $y_1, y_2, \ldots\ y_n$ of ψ based on the lengths of the branches. Tree pattern C_i is constructed from clause $c_i = (\neg y_j \vee y_k \vee \neg y_l)$.

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. $\square$

THEOREM 4 CONP COMPLETENESS. *The problem whether $p \subseteq p'$, for two tree patterns $p \in P^{\{[],//\}}$ and $p' \in P^{\{[],*,//\}}$, is coNP-complete.*

PROOF. We already know that it is in coNP (Proposition 4). 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 12, 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, $mod^z(A)$, encode truth assignments to the n variables of ψ. Tree pattern C_i is constructed so that the following property holds:

(*). For every $t \in mod^z(A)$, $C_i(t)$ 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 3: $(A \subseteq C_1 \cup \cdots \cup C_k) \iff$ (for every $t \in mod^z(A)$ 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 the corresponding subtree is denoted Y_i in Fig. 12. The figure defines Y_i on the left: it consists of some unique element of the alphabet a_i, which we associated to the variable y_i, connected to some node labeled b. Consider a canonical model $t \in mod^z(Y_i)$ (see Fig. 12). If t consists only of a_i followed by b, then we say that it corresponds to a truth assignment making y_i true. If t contains one or more added nodes between a_i and b, then we say t corresponds to a truth assignment making y_i false. Under this interpretation of true and false, each canonical model of A corresponds to a truth assignment of the variables $y_1, \ldots, y_n$, 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 12, 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 $F(y_k)$. The trees $T(-)$ and $F(-)$ are shown in Fig 12 on the left. This construction enforces property (*). □

4.2 coNP-Hardness for Bounded Wildcard

We strengthen here Theorem 4 by showing that only two $*$'s suffice in p', and no $*$ is needed in p.

Theorem 5 coNP - bounded wildcards. *The problem whether $p \subseteq p'$, for two tree patterns $p \in P^{\{[],//\}}$ and $p' \in P^{\{[],*,//\}}$, where p' has at most 2 label wildcards, is coNP-complete.*

Proof. The proof is by reduction from the complement of 01-integer linear programming [Garey and Johnson 1979] (01-ILP) which consists of m equations in n variables:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ \vdots \quad &\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

where each a_{ij} and b_k are 0 or 1, and we search for a solution $\vec{x} = (x_1, x_2, \dots x_n)$ of integers greater than or equal to 0. Checking whether (01-ILP) has a solution is NP-complete. If $a_{ij} = 1$ then we say that variable x_j occurs in equation i; if $a_{ij} = 0$ then we say that the variable x_j does not occur in equation i. We assume, without loss of generality, that the b_k are uniformly equal to 1. Then, we notice that there is *no* solution if and only if for every vector of non-negative integers, $\vec{x}$:

(1) there are $x_i \geq 1$ and $x_j \geq 1$ that occur in the same equation, for $i \neq j$, or
(2) there is an equation j with all occurring variables equal to zero.

Given a 01-ILP problem, we construct patterns $p, p'_0, p'_1, \dots p'_m$ such that $p \subseteq p'_0 \cup \cdots \cup p'_m$ if and only if (1) or (2) hold for every solution $\vec{x}$. The canonical models of p encode solution vectors, and we design p'_0 to hold precisely on canonical models satisfying (1), while p'_i holds on canonical models satisfying (2) for equation i.

We describe the construction of $p, p'_0, p'_1, \dots p'_m$, and will illustrate in Figure 13 with an example where $n = 5$. Let S be the set of pairs of variables (x_i, x_j), $i < j$, that occur together in the same equation, and let $w = |S|$, $w \leq n(n-1)/2$. Pattern p consists of a root labeled by r, followed by a chain of length w of b nodes, followed by n main branches, each corresponding to a variable x_i: see Figure 13. Each branch contains a sequence of w nodes labeled b, connected with child edges, then ends in the postfix $d//e/f_i$, where f_i is a unique symbol for the variable x_i. Next, we add some side-branches, as follows. Impose some order on the pairs in S, and consider pair number k, for each $k = 1, \dots, w$: let $(x_i, x_j) \in S$ be that pair. Add two side-branches, one in the main branch corresponding to x_i, the other in the main branch corresponding to x_j. Both side-branches will hang from the k'th b node on that main branch, and one will have a node labeled c_1, while the other will have a node labeled c_2. This construction is best illustrated on the example in Figure 13, where we assumed $S = \{(x_1, x_3), (x_1, x_5), (x_2, x_3), (x_1, x_4)\}$, hence $w = 4$. The first row of b's has the two side-branches in positions 1 and 3: this corresponds to the first pair in S, (x_1, x_3). The second row of b's has the two

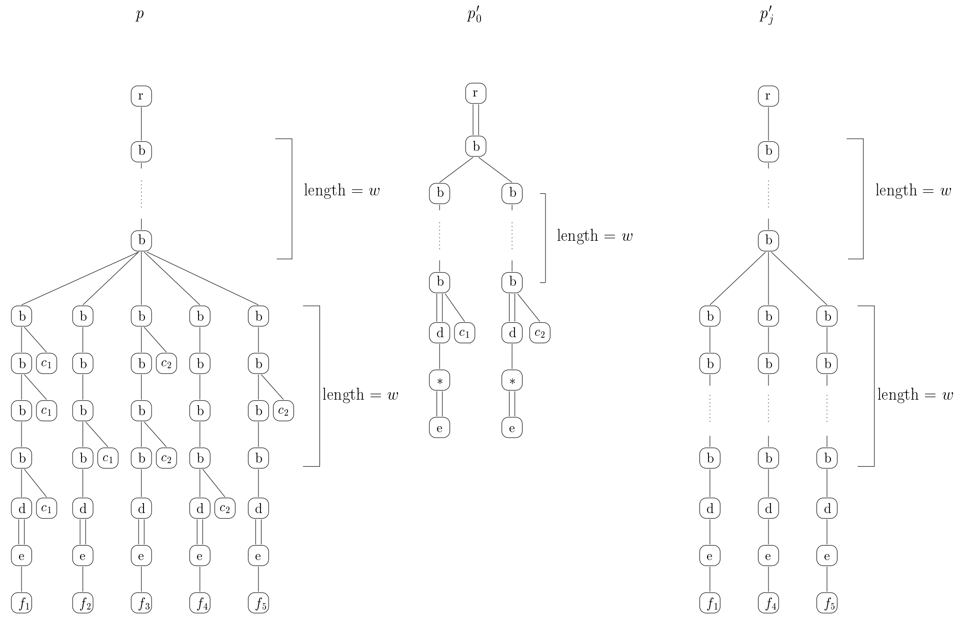


Fig. 13. Patterns p, p'_0, and p'_j (for some equation j) used in the reduction of 01-ILP to $\mathrm{P}^{\{[],*,//\}}$ containment

side-branches in positions 1 and 5, corresponding to the pair (x_1, x_5), etc. This completes the construction of p. We assume that all symbols $r, b, c_1, c_2, e, f_1, \ldots, f_n$ are distinct, and notice that there are no $*$'s in p.

Notice that each of the n main branches in p has a single descendant edge, hence that part of a canonical $t \in mod^z(p)$ represents a value $x_i \geq 0$: hence, a canonical model corresponds precisely to an n-tuple of non-negative numbers, $\vec{x}$.

Pattern p'_0 is shown in Figure 13. It consists of one descendant edge, followed by two branches with fixed height which contain a c_1 and c_2 node, respectively. If p'_0 accepts a canonical model t of p, then c_1 and c_2 must occur at the same level in t, implying that some variables x_i and x_j occur together in an equation. Further, the two branches are terminated with $d/*//e$ which implies that t encodes a solution vector in which both x_i and x_j are ≥ 1. Therefore, for any canonical model of p accepted by p'_0, condition (1) holds for the corresponding vector $\vec{x}$. Notice that there are only two wildcards in p'_0, and they are needed to check the condition ≥ 1 on both x_i and x_j. This completes the construction of p'_0, which has exactly two $*$'s.

It remains to construct $p'_1, p'_2, \ldots\ p'_m$ from the m equations of the 01-ILP instance. Suppose equation j is $x_1 + x_4 + x_5 = 1$. Then we construct p'_j as shown in Figure 13. The pattern consists of a sequence of w b nodes followed by a branch for each variable that occurs in the equation. The terminating phrase $d/e/f_i$ of each branch implies that if t is a canonical model of p accepted by p'_j then each variable occurring in equation j is zero in the solution vector encoded by t. In other words, if $p_j(t)$ is true, then condition (2) holds for the corresponding vector $\vec{x}$. This completes the construction of p'_i, which has no $*$'s.

It should be clear from the construction that $\forall t \in mod^z(p)$, $p(t) \to p'_0(t) \vee p'_1(t) \vee \ldots \vee p'_m(t)$ iff the vector $\vec{x}$ corresponding to t satisfies either condition (1) or (2). Hence, $p \subseteq p'_0 \cup \ldots \cup p'_m$ iff the 01-ILP problem instance has no solutions.

Given patterns $p, p'_0, p'_1, \ldots\ p'_m$, we use Lemma 3 to construct patterns q, q' such that $p \subseteq p'_0 \cup \cdots \cup p'_m$ if and only if $q \subseteq q'$. We can then conclude that $q \subseteq q'$ if and only if for every solution vector $\vec{x}$ either condition (1) or (2) holds. Therefore, $q \subseteq q'$ if and only if the 01-ILP instance has no solution.

It is clear that the construction of q, q' is polynomial in the size of the problem instance. In addition, there are only two label wildcards in the patterns (those in q' which are inherited from p'_0), so the Theorem follows. □

4.3 coNP-hardness for Bounded Branching

Next, we strengthen here Theorem 4 in a different direction, to show that only five branches suffices in p and only three branches in p' for co-NP hardness. We define the number of branches in a tree pattern to be the number of leaves: for example linear tree patterns have one branch.

THEOREM 6 CONP - BOUNDED BRANCHING. *The problem whether $p \subseteq p'$, for two tree patterns $p \in P^{\{[\,],*,//\}}$ and $p' \in P^{\{[\,],*,//\}}$, where p has at most 5 branches and p' has at most 3 branches is coNP-complete.*

PROOF. We show this result by reduction from the complement of satisfiability to the containment problem. Let ψ be a formula with n literals $x_1, x_2, \ldots\ x_n$, and m clauses $c_1, c_2, \ldots\ c_m$. We first define a linear pattern W whose canonical models represent assignments to the variables of ψ. Pattern W consists of a root node followed by n repetitions of $x//y//$. In a canonical model $t \in mod^z(W)$, each pair of x and y nodes corresponds to a truth assignment, as follows: `true` is the sequence $x/z/y$ while `false` is the sequence $x/y/z$; see Figure 14. Hence, only a subset of the canonical models $mod^z(W)$ of W are correct encodings of a truth assignment to the boolean variables, the others are incorrect encodings. W also has two extra nodes: a root labeled a and a leaf labeled x, hence if a canonical model of W is a correct encoding, then it has exactly $3n+2$ nodes.

Next we construct from each clause c_i a pattern C_i, s.t. for every canonical model $t \in mod^z(W)$ that is a correct encoding, $C_i(t)$ is true iff c_i is false for the corresponding truth assignment. C_i consists of $3n+2$ nodes, having 3 nodes for each variable, plus a root labeled a and a leaf labeled x. We describe now the three nodes corresponding to the boolean variable x_j. If x_j occurs negated in c_i, then these nodes are $x/*/y$; if x_j occurs positively in c_i, then the three nodes are $x/y/*$; and if x_j does not occur in c_i then the three nodes are $x/*/*$. Fig. 14 illustrates the pattern C_i for a clause c_i of the form $\bar{x}_1 \vee x_2 \vee x_4 \vee \bar{x}_5 \ldots$ (the figure shows the nodes for x_1, x_2, and x_3 only). It is straightforward to see that for every correct canonical mode $t \in mod^z(W)$, $C_i(t)$ is true iff clause c_i is false for the corresponding truth assignment. Now define C to be linear pattern obtained by concatenating $C_1, C_2, \ldots, C_m$, in notation $C_1/C_2/\ldots/C_m$. Its length is $m(3n+2)$.

Because C is much longer that W, we construct a pattern V which consists of $m-1$ copies of a pattern A followed by W and then followed by $m-1$ more copies of

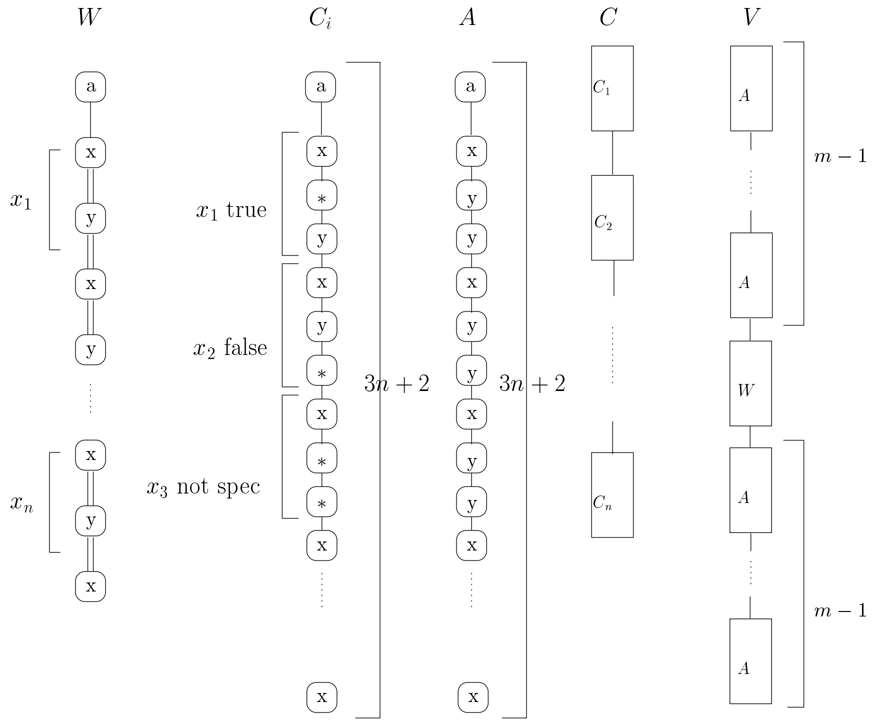


Fig. 14. Patterns used in the proof of Theorem 6.

A. Pattern A is designed so that any C_i will be true on the (single) canonical model of A: namely, it consists of n copies of the sequence $x/y/y$, plus an a root and plus an x leaf. A canonical model of V is completely determined by the canonical model of its W subpattern. A canonical model of V which represents a truth assignment consists of exactly $(2m-1)(3n+2)$ nodes.

At this point we give the main intuition of the proof, before continuing with the formal argument. Consider now the following two linear patterns (these are *not* the final patterns of our reduction):

$$\begin{aligned} p_0 &= r/V \\ p_0' &= r//C \end{aligned}$$

Here r is a new symbol, used for the root. For every correct canonical model $t \in mod^z(p_0)$, $p_0'(t)$ is true iff the formula $\psi = c_1 \wedge \ldots \wedge c_m$ is false on the truth assignment corresponding to t. This is because an embedding $e : p_0' \to t$ can map any of the clauses $C_1, \ldots, C_m$ to the portion of t corresponding to W, and conversely, each embedding must map some C_i to the W fragment. However, p_0 and p_0' are not the reductions we need, because they don't say anything about incorrect canonical databases. In fact p_0 is not contained in p_0', because p_0' is false on all incorrect canonical databases. We address the latter next.

A canonical model $t \in mod^z(V)$ is incorrect if either of the following holds:

B1. the length of t is greater than $(2m-1)(3n+2)$

B2. t contains the substring $x/y/x$.

It is a simple matter to construct patterns B_1 and B_2 that express these conditions, respectively:

$$B_1 = a/*/*...*/* \quad \text{there are } (2m-1)(3n+2)+1 \text{ nodes}$$
$$B_2 = a//x/y/x$$

Given all of the above building blocks, we are ready to construct the final queries p and p' pictured in Figure 15. Tree pattern p' has a branching node u and three branches, ending in C, B_1, and B_2 respectively: we denote these with C', B_1', B_2' to distinguish them from their isomorphic copies in p. C' is preceded by $m(2m-1)(3n+2)$ nodes labeled $*$; B_2' is preceded by one additional $*$ node, while B_1' by two additional $*$ nodes. The tree pattern p starts with two side branches, each consisting of two chains of $m(2m-1)(3n+2)$ nodes labeled $*$ followed by C. The branching nodes for these two side branches are called b_1 and b_2. Then, the main branch continues with a chain of $m(2m-1)(3n+2)$ nodes labeled $*$: we denote $v_1, v_2, \ldots, v_m$ the first node in each subchain of $(2m-1)(3n+2)$ nodes. Then there are three branches: V, and B_1, B_2, with the latter two preceded by $*/a/a/a/\ldots/a$. The three way branching node is labeled a.

We prove now that $p \subseteq p'$ if and only if ψ is not satisfiable. Assume first that ψ is not satisfiable. Let $t \in mod^z(p)$ be any canonical database. To show that $p'(t)$ is true, we will construct an embedding $e : p' \to t$. There are three cases, and in each we will explain where e maps the node u, and each of the branches C', B_1', and B_2'.

(1) t is "incorrect" because it violates condition **B1**. Then we define e as follows:

	u	C'	B_1'	B_2'
e	b_1	left C	V	B_2

That is, $e(u) = b_1$, C' is mapped precisely over the left C branch, B_1' is mapped precisely over V, while B_2' is mapped to B_2: the first a symbol in B_2' is mapped to the three-way branching a node, while the rest of B_2' (which is separated from a by a descendant edge) is mapped along the B_2 branch to the appropriate depth (i.e. $x/y/x$ over $x/y/x$). Notice that for any embedding e s.t. $e(u) = b_1$, B_1' cannot be mapped to either the B_1 branch nor to the B_2 branch, because the first symbol in B_1' is a, and, when $e(u) = b_1$, this will be mapped precisely to the z node corresponding to the leading $*$ on either the B_1 or the B_2 branch, which is impossible.

(2) t is "incorrect" because it violates condition **B2**. Then define e as follows:

	u	C'	B_1'	B_2'
e	b_2	right C	B_1 branch	V

Here $e(u)$ is one level lower, and C' is mapped to the right C branch, which is one level lower too. B_1' is mapped to the B_1 branch: more precisely it will be mapped starting to the first a node on the chain on that branch. B_2' is mapped to V. Notice that for every embedding e s.t. $e(u) = b_2$, e cannot map B_2' to either the B_1 branch nor to the B_2 branch in p: this is because the a symbol in B_2' would be mapped to the z symbol (corresponding to the $*$).

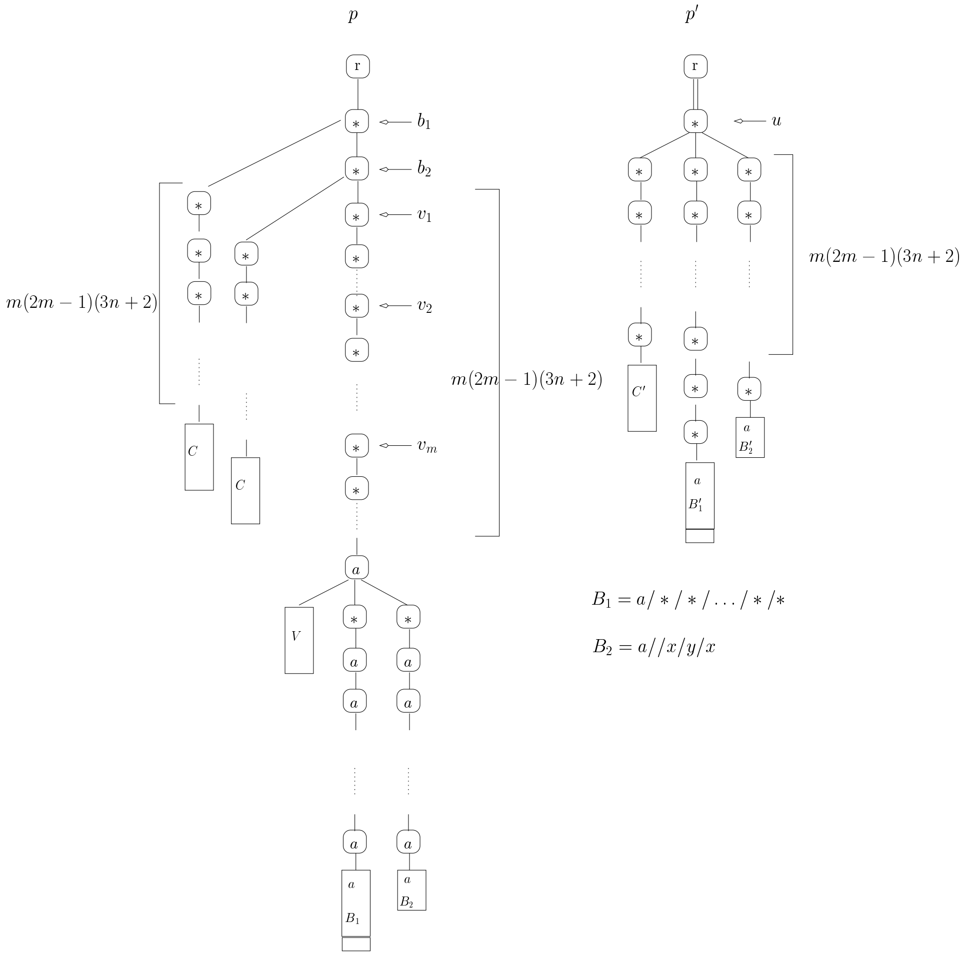


Fig. 15. Patterns used in the proof of Theorem 6. Pattern p has five branches while p' has three branches. $C' = C$, $B_1' = B_1$, and $B_2' = B_2$. The drawing shows the leading a symbol in B_1 and the fact that B_1 is one node longer than a correct canonical model in $mod^z(V)$. The drawing also shows that B_2 starts with an a symbol.

(3) t is "correct", and the corresponding assignment to boolean variables makes clause c_i false. Then we define e as follows:

	u	C'	B_1'	B_2'
e	v_i	inside V	B_1 branch	B_2 branch

The mapping of C' inside V should be obvious, according to our previous discussion. B_1' is mapped starting at some a along the B_1 branch. Since B_1' only checks that there are at least $(2m-1)(3n+1)+1$ nodes, the embedding is correct as long as we have enough a's on this branch (we need $m(2m-1)(3n+2)+1$ such a's, where the $+1$ is needed for case 2 above). Similarly, B_2' is correctly mapped to the B_2 branch: its a node is mapped to some a (again, we need to have enough: in this case we need $m(2m-1)(3n+2)$ a's, i.e. one less than on the B_1 branch), then its $x/y/y$ sequence is mapped to the corresponding $x/y/y$ sequence in B_2.

Now we prove the converse, that if $p \subseteq p'$ then ψ is not satisfiable. Suppose ψ were satisfiable, and let $t \in mod^z(p)$ be the (correct) canonical database corresponding to the truth assignment that satisfies all clauses $c_1, c_2, \ldots, c_m$. We show that there cannot be any embedding $e : p' \to t$. Indeed, suppose there were such. We consider four cases, according to where e maps the node u:

(1) $e(u) = b_1$. We have argued in this case that e cannot map B_1' to the B_1 branch or to the B_2 branch. It cannot be mapped to any of the two C branches, because they are too short. Hence, it has to be mapped to V proving that t is not "correct" because of the violation **B1**. But this contradicts our assumption.

(2) $e(u) = b_2$. We have argued already that in this case B_2' cannot be mapped to either the B_1 branch nor to the B_2 branch. It cannot be mapped to any of the two C branches, because they don't contain the sequence $x/y/x$. Hence, it has to be mapped to V, proving that t is not correct, because of the violation **B2**: but this contradicts our assumption.

(3) $e(u)$ is some node along the chain of $*$'s. Then one can see that C' has to be mapped inside V (it is easy to see that it cannot be mapped to the B_1 or B_2 branches). Because clauses are separated by the a symbol, some C_i' in C' will be mapped precisely to W in V (see Figure 14 for the structure of C and V). But then the truth assignment makes c_i false, contradicting our assumption.

(4) $e(u)$ is in one of the five branches. This is impossible because the branch B_1' is too long to mapped below $e(u)$.

□

5. DISCUSSION

This section briefly covers additional topics of interest.

5.1 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 $\mathrm{P}^{\{\mathrm{or}\}}$ 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 $\mathrm{P}^{\{\mathrm{or}\}}$ can be reduced in PTIME to the containment of two patterns in P, by a simple application of Lemma 3. However, when branches are allowed, a minor modification to the proof of Theorem 4 shows that containment for $\mathrm{P}^{\{[],\mathrm{or}\}}$ patterns is coNP-complete. No descendant edges are needed for the coNP-hardness to hold: indeed, in Theorem 4 we essentially used descendant edges to simulate disjunction, which we now get for free. On the other hand, we can extend the grammar for XPath, Eq.(1), with $q ::= q \mid q$, and denote with $\mathrm{P}^{\{[],*,//,|\}}$ this extended language. This form of disjunction is exponentially more concise than *or* nodes 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 $\mathrm{P}^{\{\mathrm{or}\}}$. If we form the fragment $\mathrm{XP}^{\{|\}}$ by including just the child axis and disjunction (in the absence of all other features) we can see that the containment problem is already coNP-complete.

THEOREM 7. *Given expressions $p, p' \in XP^{\{|\}}$, deciding containment is coNP-hard.*

PROOF. In [Stockmeyer and Meyer 1973] it is shown that deciding language equivalence is coNP-complete for string languages defined using only the regular operations of concatenation and union. (This problem is referred to as language equivalence for *star-free* regular languages in [Garey and Johnson 1979], although here complement is not present, as it is in the customary understanding of "star-free".) Such languages, over an alphabet Σ, are defined by expressions in the following simple grammar:

$$E \to a \mid E.E' \mid E \cup E'$$

for $a \in \Sigma$. We can reduce equivalence of expressions in $\mathrm{XP}^{\{|\}}$ to this problem as follows. If e is an expression in this grammar, we construct an XPath expression in $\mathrm{XP}^{\{|\}}$ in a natural way, then add a terminal symbol $t \notin \Sigma$ to the end of the expression. For instance, $e = (a \cup a.b) \cup c$ is translated to $p = (a \mid a/b) \mid c/t$. If $p, p' \in \mathrm{XP}^{\{|\}}$ are translations of expressions e, e' it's not hard to show that $L(e) \equiv L(e')$ if and only if $p \equiv p'$. □

Containment for $\mathrm{XP}^{\{|\}}$ expressions is in coNP because we can define the canonical models for an expression in $\mathrm{XP}^{\{|\}}$ (by making choices for each disjunction) and then guess a counter example to containment (closely related to Proposition 4). In fact, it's not hard to show that containment remains in coNP for $\mathrm{P}^{\{[],*,//,|\}}$, by a similar argument. The authors of [Neven and Schwentick 2003] show this result, as well as providing complexity results for these disjunctive fragments in the case of finite alphabets, discussed next.

5.2 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 [Amer-Yahia et al. 2001] no longer holds for a finite alphabet. In the presence of disjunction, finite alphabets have a substantial impact on the complexity of containment, since disjunction allows to express negation over label predicates. Neven and Schwentick show in [Neven and Schwentick 2003] a rather remarkable result: that containment is in PSPACE for $P^{\{[],*,//,|\}}$ and complete for PSPACE for $P^{\{//,|\}}$.

5.3 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, define $p(g)$ to be true if there exists an embedding $e : p \to g$. Then one can show that the containment problem on graphs is the same as the containment problem on trees: $\forall g.p(g) \Rightarrow p'(g)$ iff $\forall t.p(t) \Rightarrow p'(t)$. 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) $p(\text{unfold}(g))$ is true, (3) $\exists t \subseteq \text{unfold}(g)$, t finite and $p(t)$ is true. Thus, if there exists a witness graph g s.t. $p(g)$ is true but $p'(g)$ is false, then we can construct a witness tree t s.t. $p(t)$ is true and $p'(t)$ is false: just take $t \subseteq \text{unfold}(g)$ as above. Notice that $p'(t)$ cannot be true, since any embedding from p' to t extends to an embedding from p' to g. This shows that $\forall t.p(t) \Rightarrow p'(t)$ then $\forall g.p(g) \Rightarrow p'(g)$. The other direction is trivial.

5.4 Application to CTL

Tree patterns can be expressed in a certain fragment of computation tree logic (CTL) [Vardi 1997] consisting of $true$, $x = a$, conjunction, "eventually true" formulas EFϕ, and "successively true" formulas EXϕ. We call this fragment *conjunctive existential CTL*, ECTL$_\wedge$, and show that it is equivalent to tree patterns in $P^{\{[],*,//\}}$. Thus, 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.

We need to consider a few changes to our formalism, to align to the standard definitions in CTL. We assume Σ to be finite in this discussion: recall that all co-NP hardness results still hold in this case. We consider both finite and infinite trees: in this subsection, T_Σ denotes the set of all trees, finite and infinite. Define a *complex tree* with labels from Σ to be a tree in T_{2^Σ}. Thus, nodes in complex tree are labeled with sets of symbols from Σ. We define a *complex tree pattern* similarly: its nodes are now labeled either with $*$ or with a set of symbols in Σ. We also modify the definition of an embedding e from a tree pattern p to a tree t as follows: whenever $e(y) = x$, for $y \in \text{NODES}(p), x \in \text{NODES}(t)$, we require that either $\text{LABEL}(y) = *$ or $\text{LABEL}(y) \subseteq \text{LABEL}(x)$. With this definition, $*$ is analogous to $\emptyset$, and is actually not needed any more in tree patterns. Simple trees and patterns are a special case of complex trees and patterns, where each node label set has size one, or is $\emptyset$ (representing $*$ in a tree pattern).

Definition of ECTL$_\wedge$ We define here the fragment ECTL$_\wedge$ of interest to us, and refer the reader to [Vardi 1997] for a definition of CTL. We define below both the syntax and the semantics of formulas of ECTL$_\wedge$. For the purpose of ECTL$_\wedge$ we call the elements in Σ *propositional constants*: ECTL$_\wedge$ formulas are built from these propositional constants. The semantics of a ECTL$_\wedge$-formula ϕ is a truth value for each (possibly infinite) complex tree $t \in T_{2^\Sigma}$ and each $x \in \text{NODES}(t)$: we write

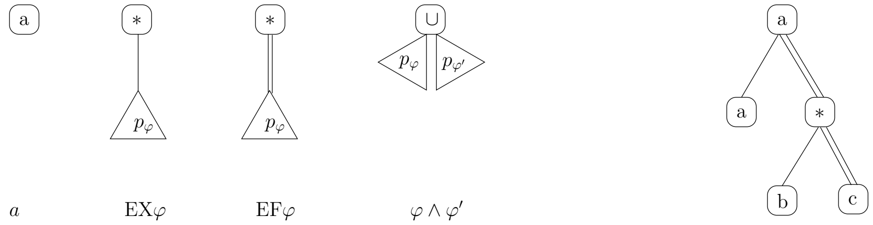


Fig. 16. ECTL$_\wedge$ formula to tree pattern translation
For each of the four kinds of formulas, an equivalent tree is pictured. The proof of Theorem 7 describes the details. The example tree pattern on the right is equivalent to formula: $a \wedge \mathrm{EX}a \wedge \mathrm{EF}(\mathrm{EX}b \wedge \mathrm{EF}c)$

$(t, x) \models \phi$ for the truth value of ϕ at t and x. Notice that in CTL the main interest is interpreting formulas over graphs, or, equivalently, over their unfoldings into infinite trees. We adopt here the same semantics for the ECTL$_\wedge$ fragment, hence allow the tree t to be infinite. The chart below defines both valid ECTL$_\wedge$ formulas and their semantics:

Formulas		**Semantics**	
a	for any $a \in \Sigma$	$(t, x) \models a$	if $a \in \textsc{label}(x)$
$\phi \wedge \phi'$	for any $\phi, \phi' \in \mathrm{ECTL}_\wedge$	$(t, x) \models \phi \wedge \phi'$	if $(t, x) \models \phi$ and $(t, x) \models \phi'$
$\mathrm{EX}\phi$	for $\phi \in \mathrm{ECTL}_\wedge$	$(t, x) \models \mathrm{EX}\phi$	if $\exists y.(x, y) \in \textsc{edges}(t)$ and $(t, y) \models \phi$
$\mathrm{EF}\phi$	for $\phi \in \mathrm{ECTL}_\wedge$	$(t, x) \models \mathrm{EF}\phi$	if $\exists x_0, \exists x_1, \ldots \exists x_k$, $k \geq 0$, with $x_0 = x$ and $(x_i, x_{i+1}) \in \textsc{edges}(t)$ for $0 \leq i < k$ and $(t, x_k) \models \phi$

For any formula ϕ, $(t, x) \models \mathrm{EX}\phi$ if x has a child satisfying ϕ, and $(t, x) \models \mathrm{EF}\phi$ if x has a descendant satisfying ϕ. The formula $\mathrm{EF}\phi$ is an abbreviation for $\mathrm{E}(\mathit{true} U \phi)$ in full CTL[2].

Given two formulas $\phi, \psi \in \mathrm{ECTL}_\wedge$, the *implication problem* asks whether for all (infinite) trees t and nodes x, $(t, x) \models \phi$ implies $(t, x) \models \psi$. We prove that the implication problem for ECTL$_\wedge$ is co-NP complete, by showing that it is equivalent to the containment problem for tree patterns. We need to handle with care the fact that ECTL$_\wedge$ implication is defined over all trees (finite and infinite) while pattern containment is only for finite trees.

We start by showing that ECTL$_\wedge$ implication is equivalent to containment of *complex* tree patterns.

PROPOSITION 7. *There exists a one-to-one translation from ECTL$_\wedge$ to complex tree patterns, $\phi \to p_\phi$, such that for any complex tree t (finite or infinite), $(t, \textsc{root}(t)) \models \phi$ if and only if $p_\phi(t)$ is true.*

[2]This highlights another restriction of ECTL$_\wedge$ that is not evident from its name, conjunctive existential CTL: full CTL allows formulas of the form $\mathrm{E}(\psi U \phi)$ for any ψ

PROOF. We describe the translation informally, by induction on the formula ϕ. See Figure 16 for an illustration. For the base case, if $\phi = a$ for $a \in \Sigma$, then p_ϕ is the tree pattern consisting of a single (root) node labeled a. Obviously, the theorem holds in this case. For the inductive cases, if φ occurs in ϕ we assume we have a complex pattern p_φ satisfying the theorem. If $\phi = \mathrm{EX}\varphi$, then p_ϕ is the tree pattern consisting of a $*$-labeled root node under which p_φ is rooted by a child edge. Similarly, for $\phi = \mathrm{EF}\varphi$, we construct p_ϕ as a $*$-labeled node with p_φ rooted beneath it by a descendant edge. If $\phi = \varphi \wedge \varphi'$ then we fuse the roots of p_φ and $p_{\varphi'}$ to form p_ϕ, setting the label for the fused node to the union of the set-labels on the roots p_φ and $p_{\varphi'}$. Note that $*$ functions like the formula *true*, so under union it will disappear. Under these constructions it is fairly obvious that $(t, \textsc{root}(t)) \models \phi$ if and only if $p_\phi(t)$ because the semantics of the formulas and our pattern embeddings are identical.

We must also confirm that for any complex pattern, we can construct an $\mathrm{ECTL}_\wedge$ formula satisfying the theorem. It is relatively clear that any complex pattern can be built, bottom up, from the tree construction operations applied above. Thus it follows that there is always an equivalent $\mathrm{ECTL}_\wedge$-formula. □

□

Now we bridge the gap between finite trees for patterns and infinite trees for formulas.

PROPOSITION 8. *If ϕ,ϕ' are $ECTL_\wedge$-formulas, and p_ϕ,$p_{\phi'}$ are their equivalent complex patterns, then $p_\phi \subseteq p_{\phi'}$ (over all finite complex trees) if and only if $\phi \to \phi'$ is valid.*

PROOF. The implication $\phi \to \phi'$ is valid if it is satisfied in every node of every (finite or infinite) tree. We must also contend with the fact that the tree need not be finite. The following equivalences prove the proposition:

$$
\begin{array}{llll}
p_\phi \subseteq p_{\phi'} & \text{IFF } p_\phi(t) \to p_{\phi'}(t) & \forall \text{ complex, finite trees} & (1)\\
& \text{IFF } p_\phi(t) \to p_{\phi'}(t) & \forall \text{ complex, finite or infinite trees} & (2)\\
& \text{IFF } (t, \textsc{root}(t)) \models \phi \to (t, \textsc{root}(t)) \models \phi' & \forall \text{ complex, finite or infinite trees} & (3)
\end{array}
$$

Line (1) is the definition of complex pattern containment. The implication (2)⇒(1) is trivial. The implication (1) ⇒ (2) follows from the fact that, if there exists an infinite witness t s.t. $p_\phi(t)$ is true and $p_{\phi'}(t)$ is false, then we can find a finite witness, t_0, by taking it to be the image of the embedding $p_\phi \to t$, and we still have $p_\phi(t_0)$ true and $p_{\phi'}(t_0)$ false. The equivalence of (2) and (3) follows directly from Proposition 7. □

Containment of complex tree patterns is in co-NP: this is relatively easy to see, since all constructions in Sec. 3.1 extend straightforwardly to complex trees and complex patterns. It follows that the implication problem for $\mathrm{ECTL}_\wedge$ is also in co-NP.

To prove that the implication problem for $\mathrm{ECTL}_\wedge$ is co-NP hard, we show that, as far as simple pattern trees are concerned, containment over complex trees is equivalent to containment over simple trees. This proves our result, since we have shown that containment for simple trees is coNP-hard.

Proposition 9. *For any simple tree patterns p and p' in $P^{\{[],*,//\}}$, $p(t) \to p'(t)$ for all complex trees if and only if $p(t) \to p'(t)$ for all simple trees.*

Proof. The forward direction of the claim is obvious: if $p(t) \to p'(t)$ for all complex trees, then this also holds for all simple trees. For the converse, suppose $p(t) \to p'(t)$ for all simple trees, and let $t \in T_{2^\Sigma}$ be a complex tree such that $p(t)$ is true and $p'(t)$ is false. Unfold t s.t. the embedding v from p into t becomes injective. Construct a new simple tree t_0 isomorphic to t by considering only the image under v, and replacing each complex label of some node $v(x)$ with the simple label of the node $x \in \text{NODES}(p)$; when x is labeled $*$, pick any label for $v(x)$, say z. Denote with t_0 the resulting tree. Clearly $p(t_0)$ is true, and $p'(t_0)$ is false, since otherwise, if there exists an embedding from p' to t_0 then we can extend it to an embedding from p' to t. $\square$

In summary, we have shown:

Theorem 8. *The implication problem for $ECTL_\wedge$ is co-NP complete.*

6. 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 [Shmueli 1993]. In [Wood 2000] 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 [Amer-Yahia et al. 2001]. Queries in $P^{\{[],*\}}$ can be viewed as conjunctive queries over tree structures. In general, containment for conjunctive queries is NP-complete [Chandra and Merlin 1977], however for acyclic conjunctive queries containment is in PTIME [Yannakakis 1981], from which it follows that $P^{\{[],*\}}$ containment is solvable in PTIME. This bound for $P^{\{[],*\}}$ was also noted in [Wood 2001].

Linear queries in $P^{\{*,//\}}$ are a special case of regular expressions on strings, for which there is a PSPACE-complete containment algorithm in general [Stockmeyer and Meyer 1973]. For the fragment of regular string expressions in $P^{\{*,//\}}$, a linear-time containment algorithm is claimed in [Milo and Suciu 1999], although the proof was not published. A PTIME algorithm for linear patterns in $P^{\{//\}}$ was provided in [Buneman et al. 2000].

On a graph-based data model, the authors of [Florescu et al. 1998] showed that for a restricted language without wildcard, similar to $P^{\{[],//\}}$, containment is NP-complete. In [Calvanese et al. 2001] 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. The authors of [Deutsch and Tannen 2001] 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. In [Neven and Schwentick 2003] the authors show that containment of patterns in $P^{\{[],*,//,|\}}$, while coNP-complete for an infinite alphabet, is in PSPACE for finite alphabets, and show that for fragment

$P^{\{//,|\}}$ containment is complete for PSPACE. In the same work, the authors also study the complexity of containment for XPath languages that include variable bindings under two different semantics.

Algorithm 1 is a particular evaluation algorithm of a small fragment of XPath on XML documents. More general techniques are studied in [Gottlob et al. 2002; 2003], which discuss evaluations of larger fragments of XPath.

Hoffman and O'Donnell [Hoffmann and O'Donnell 1982] introduce the *tree pattern matching* problem, in which a subject tree (the data) has to be matched with a set of tree patterns (the queries). The problem was motivated by several applications, and has since spawned a large amount of work [Cai et al. 1992; Thorup 1996; Cole et al. 1999]. Hoffmann and O'Donnell show that the tree patterns can be preprocessed into a data structure of exponential size, which factors out all common subpatterns, such that every subject tree can subsequently be matched bottom-up in linear time. Algorithm 2 in this paper borrows the idea of *match sets* from that work.

7. CONCLUSION

We have studied the complexity of containment and equivalence for an important core fragment of XPath. Many XML applications 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.

8. ACKNOWLEDGMENTS

We would like to thank Igor Tatarinov, Paul Beame, and Moshe Vardi for their comments, and the anonymous reviewer who pointed us the connection between dot (the current node) and union, and the alternative technique for encoding k-ary patterns into boolean patters that does not introduce extra branches. Suciu was partially supported by the NSF CAREER Grant 0092955, a gift from Microsoft, and an Alfred P. Sloan Research Fellowship.

REFERENCES

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. 2001. Minimization of tree pattern queries. *SIGMOD*.

BUNEMAN, P., DAVIDSON, S., FAN, W., HARA, C., AND TAN, W. 2000. Reasoning about keys for xml.

CAI, J., PAIGE, R., AND TARJAN, R. 1992. More efficient bottom-up multi-pattern matching in trees. *TCS 106,* 1, 21–60.

CALVANESE, D., DE GIACOMO, G., LENZERINI, M., AND VARDI, M. Y. 2001. View-based query answering and query containment over semistructured data. Submitted for publication.

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

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

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

Cole, R., Hariharan, R., and Indyk, P. 1999. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *SODA*. 245–254.

Dalvi, N. and Sanghai, S. 2002. Personal communication.

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

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

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

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

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

Gottlob, G., Koch, C., and Pichler, R. 2002. Efficient algorithms for processing xpath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*.

Gottlob, G., Koch, C., and Pichler, R. 2003. Xpath query evaluation: Improving time and space efficiency. In *Proc. of 19th International Conference on Data Engineering (ICDE 2003)*.

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

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

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

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

Neven, F. and Schwentick, T. 2003. Xpath containment in the presence of disjunction, dtds, and variables. In *Proc. of 19th International Conference on Data Engineering (ICDE 2003)*.

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

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

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

Thorup, M. 1996. Efficient preprocessing of simple binary pattern forests. *Journal of Algorithms 20,* 3, 602–612.

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

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

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

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

XSch 1999. XML Schema part 1: Structures. http://www.w3.org/TR/1999/WD-xmlschema-1-19991217/. W3C Working Draft.

Yannakakis, M. 1981. 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)*.

A. APPENDIX

PROOF. (Of Proposition 1) We instroduce first a notation. Let p be either a tree or a tree pattern, and a tuple $\bar{z} = (z_1, \ldots, z_k)$ of k nodes in p (not necessarily distinct). We denote $p[\bar{z}/\bar{s}]$ the tree or tree pattern over the alphabet $\Sigma \cup \{s_1, \ldots, s_k\}$ obtained from p as follows. Add k new nodes $y_1, \ldots, y_k$, label them with $s_1, \ldots, s_k$ respectively, and add k new edges $(z_1, y_1), \ldots, (z_k, y_k)$. We call $\bar{y}$ the *extra nodes* in $p[\bar{z}/\bar{s}]$.

Given a pattern p of arity k we translate it into the boolean pattern $p_0 = p[\bar{x}/\bar{s}]$, where $\bar{x}$ is the tuple of distinguished nodes. Thus, p_0 consists of the nodes in p plus k "extra" nodes. The relationship between p and p_0 is expressed by the property below:

$$\forall t \in T_\Sigma, \forall \bar{z} \in \text{NODES}^k(t), \bar{z} \in p(t) \iff p_0(t[\bar{z}/\bar{s}]) \text{ is true.}$$

We prove $\Longrightarrow$. Let $e : p \to t$ be an embedding s.t. $e(\bar{x}) = \bar{z}$. Then e can be extended to an embedding from p_0 $(= p[\bar{x}/\bar{s}])$ to $t[\bar{z}/\bar{s}]$, by mapping the k extra nodes in p_0 to the corresponding k extra nodes in $t[\bar{z}/\bar{s}]$, proving that $p_0(t[\bar{z}/\bar{s}])$ is true. The direction $\Longleftarrow$ is equally simple and omitted.

To prove the proposition it remains to show that $p \subseteq p'$ iff $p_0 \subseteq p'_0$, for any two tree patterns p, p'. Assume first that $p_0 \subseteq p'_0$, and let $t \in T_\Sigma$, and $\bar{z} \in p(t)$: we have to show that $\bar{z} \in p'(t)$. First we use the property above to show that $p_0(t[\bar{z}/\bar{s}])$ is true, hence $p'_0(t[\bar{z}/\bar{s}])$ is true, hence we can use the property again to conclude that $\bar{z} \in p'(t)$.

Assume now that $p \subseteq p'$ and let $t \in T_{\Sigma \cup \{s_1, \ldots, s_k\}}$ be s.t. $p_0(t)$ is true: we have to show that $p'_0(t)$ is true. The problem here is that t is not necessarily of the form $t'[\bar{z}/\bar{s}]$, hence we cannot apply the property immediately: we construct first such a t'. Let $e : p_0 \to t$ be an embedding making $p_0(t)$ true, and denote $\bar{z} = e(\bar{x})$ the images of the distinguished nodes in p. Also denote $\bar{u}$ the images under e of the extra nodes in p_0: thus the nodes $\bar{z}$ are the parents of the nodes $\bar{u}$ in the tree t. Let $s \in \Sigma$ be a label that does not appear in p', and denote $t' \in T_\Sigma$ the tree obtained from t by renaming all labels $s_1, \ldots, s_k$ with s. We prove that there exists an embedding $p_0 \to t'[\bar{z}/\bar{s}]$: define such an embedding to agree with e on $\text{NODES}(p)$, and to map the k extra nodes in p_0 to the corresponding k extra nodes in $t[\bar{z}/\bar{s}]$. It is easy to see that this is indeed an embedding. Hence $p_0(t'[\bar{z}/\bar{s}])$ is true and, by the property above, we have that $\bar{z} \in p(t')$. It follows that $\bar{z} \in p'(t')$, hence $p'_0(t'[\bar{z}/\bar{s}])$ is true, hence there exists an embedding $e' : p'_0 \to t'[\bar{z}/\bar{s}]$, which maps the distinguished nodes in p'_0 to $\bar{z}$. We construct now an embedding from p'_0 to t as follows: it agrees with e' on $\text{NODES}(p')$, and it maps the k extra nodes in p'_0 to $\bar{u}$. One can check that this is indeed an embedding. Indeed the restriction to $\text{NODES}(p')$ is an embedding since the label s does not appear in p', so replacing it with $s_1, \ldots, s_k$ in t will not result in any violations of the embedding. It maps the extra nodes in p'_0 to nodes $\bar{u}$ with the correct labels, $s_1, \ldots, s_k$. Finally, the parents of these extra nodes in p'_0 are precisely the distinguished nodes in p', and the latter are mapped to the nodes $\bar{z}$: hence the k pairs (dinstinguished node, extra node) in p_0 are mapped to the k pairs (z_i, u_i), for $z_i \in \bar{z}, u_i \in \bar{u}$, showing that the embedding respects the parent-child relationship for the extra nodes. Hence $p'_0(t)$ is true. □

PROOF. Proof of Lemma 2 We use the following lemma. Given a tree t, define a *chain* in t to be a sequence of nodes $x_1, x_2, \ldots, x_n$ s.t. x_i is the unique child of x_{i-1}, for $i = 2, 3, \ldots, n$. That is, the nodes in the chain must have a unique child, except for the last node.

LEMMA 4. *Let $t_1 \in T_\Sigma$ be a tree and p' a tree pattern such that $p'(t_1)$ is false, and let w' be the star length of p'. Let $x_1, \ldots, x_n$ be a chain in t s.t. $n > w'+1$ and none of the labels $\text{LABEL}(x_1), \ldots, \text{LABEL}(x_n)$ occurs in p'. Let t_2 be the tree obtained from t_1 by deleting the node x_n and transforming all its children into children of x_{n-1}: that is, $\text{NODES}(t_2) = \text{NODES}(t_1) - \{x_n\}$ and $(x_n, y) \in \text{EDGES}(t_1) \Leftrightarrow (x_{n-1}, y) \in \text{EDGES}(t_2)$, $\forall y \in \text{NODES}(t_2)$. Then $p'(t_2)$ is also false.*

First we show how the lemma completes the proof of Lemma 2. Indeed, assume w.l.o.g. that $u_1 > w' + 1$, hence $t_1 = s^z(p[u_1, u_2, \ldots, u_d])$ has a chain of length u_1 whose nodes are labeled with z, which does not occur in p'. Since $p'(t_1)$ is false, we can apply the lemma repeatedly and delete one by one nodes from this chain, until we obtain a tree t_2 where the chain has length $w' + 1$. It follows from the lemma that t_2 is still a witness, and, obviously $t_2 = s^z(p[w' + 1, u_2, \ldots, u_d])$. By repeating this process we for every u_i s.t. $u_i > w' + 1$, we finally obtain a witness in $mod^z_{w'+1}(p)$.

We now prove the lemma. Let t_1, p', t_2 be as in the lemma. Assume that $p'(t_2)$ is true, and let $e_2 : p' \to t_2$ be the embedding. Denote C the set $\{x_1, \ldots, x_n\}$. Define the following two sets:

$$\begin{aligned} S &= \{(z_0, z_1) \mid (z_0, z_1) \in \text{EDGES}_{//}(p'), e_2(z_1) \in C\} \\ C' &= \{z \mid e_2(z) \in C \wedge \exists (z_0, z_1) \in S, (z_1, z) \in \text{EDGES}^*(p')\} \end{aligned}$$

The set S contains all descendant edges in p' where the end node is mapped to the chain C. The set C' consist of all nodes that are mapped to C *and* are below some descedant edge in S. Now we define the following function $e_1 : \text{NODES}(p') \to \text{NODES}(t_1)$:

$$\begin{aligned} e_1(z) &= x_{i+1} \quad \text{if } z \in C' \text{ and } e_2(z) = x_i \\ e_1(z) &= e_2(z) \quad \text{if } z \notin C' \end{aligned}$$

We check now that e_1 is indeed an embedding. It is easy to see that it is root-preserving and label-preserving, so we only have to check that it is also child-edge and descendant-edge preserving. Consider an edge $(z, z') \in \text{EDGES}(p')$. One can check that $(e_1(z), e_1(z')) \in \text{EDGES}^+(t_1)$ and that that the distance $d(e_1(z), e_1(z'))$ is either $d(e_2(z), e_2(z'))$ or $d(e_2(z), e_2(z')) + 1$. This is because e_1 is either identical to e_2 or is one node below e_2. Then clearly e_1 is descendant-edge preserving, so now we check that it is child-edge preserving. The only problem here is when $(z, z') \in \text{EDGES}_/(p')$ and $d(e_1(z), e_1(z')) = 2$, and this can only happen when $x_{n-1} = e_1(z) = e_2(z)$ and $e_1(z') = e_2(z')$. In this case $e_2(z), e_2(z')$ are connected by an edge in t_2, while in t_1 we have the node x_n between them. This happens if $z \notin C'$, i.e. there is no descendant edge above it mapped to C. Consider then the path in p' from z to $\text{ROOT}(p')$: $y_1 = z, y_2, y_3, \ldots, y_k = \text{ROOT}(p')$. We have $e_2(y_1) = x_{n-1} \in C$ and let m be the largest number s.t. $e_2(y_m) \in C$; hence $y_{m+1} \notin C'$ (there exists such m, because $e_2(\text{ROOT}(p')) \notin C'$). Consider the path $y_m, y_{m-1}, \ldots, y_1$ in p'. All its nodes are mapped by e_2 to nodes in C, hence they are all labeled with $*$. Furthermore, all edges (y_{i+1}, y_i) are child edges, for $i = 1, 2, \ldots, m$, since if one where a descendant edge then $z = y_1 \in C'$, while we have already established that $z \notin C'$. Hence it is a sequence of $*$'s in p', so $m \leq w'$. On the other hand, since all edges are child edges, their image under e_2 must include all nodes $x_1, x_2, \ldots, x_{n-1}$, in other words $m = n - 1$. This imples that $n - 1 \leq w'$, which is a contradiction.

□

PROOF. Of Proposition 6
We start by proving another lemma:

LEMMA 5. *Let b be a block of size n. Let $q \in P^{\{[\,],*,//\}}$ be any tree pattern, define $\bar{u} = (n, n, \ldots, n)$, and consider the canonical model $t = s^z(q[\bar{u}]) \in mod^z(q)$. Then, if there exists an un-rooted embedding $e : b \to t$, then there exists an un-rooted homomorphism $h : b \to q$, such that $e = e_t \circ h$, where $e_t : q \to t$ is the canonical embedding (Sec. 3.1).*

PROOF. Recall that there are two kinds of nodes labeled z in t: those corresponding to $*$ nodes in q, and those corresponding to extension nodes. We show that none of the extension nodes is in the image of e. Let $\text{NODES}(b) = \{x_0, \ldots, x_n\}$. The two end points, x_0 and x_n are each labeled with some symbol in Σ (not $*$) which is different from z: hence, e cannot map them to a z-node. Suppose $y = e(x_i)$ is an extension node (hence, it is labeled with z). The node y is part of a chain of n nodes, extending some descendant edge: let u and v the nodes before and after this chain, hence $d(u, v) = n + 1$. $e(x_0)$ is either u or an ancestor of u, while $e(x_n)$ is either v or a descenant of v. Hence $d(e(x_0), e(x_n)) \geq n + 1$, while $d(x_0, x_n) = n$. This is a contradiction since the definition of an embedding requires $d(e(x_0), e(x_n)) = d(x_0, x_n)$ (since all edges in b are child edges). It follows that e maps all the nodes in b only to nodes in q, and not to the extra z-nodes introduced by the extension. Then define $h(x) = e(x)$, $\forall x \in \text{NODES}(b)$; it is easl to see that h is a homomorphism and that $e = e_t \circ h$. □

Returning to the proof of Proposition 6, we proceed by induction on the number of blocks in p''. The base case, when p'' has a single block, follows immediately from the lemma, by taking $q = p$.

We prove the inductive step, and assume that p'' has at two or more blocks. Then we can write it as $p'' = b//^{\geq k}p''_1$, where b is the first block and p''_1 has one less blocks, hence Proposition 6 holds for p''_1. To prove it for p'', assume that there is no un-rooted homomorphism from p'' to p, and we will construct a witness $t = s^z(p[\bar{u}]) \in mod^z(p)$ s.t. there is no un-rooted embedding from p'' to t. All we need in order to construct t is to define $\bar{u} = (u_1, \ldots, u_d)$. We will define each u_i to be either 0, or n, where n is the size of the block b (i.e. the number of edges), or we will obtain it inductively, from some witness for p''_1. Given $x \in \text{NODES}(p)$ denote $x\downarrow = \{y \mid (x, y) \in \text{EDGES}^+(p)\}$ the set of its strict descenants, and given $X \subseteq \text{NODES}(p)$ denote $X\downarrow = \bigcup\{x\downarrow \mid x \in X\}$. Let w be the last node in b, and w' be the first node in p_1'': that is $(w, w') \in \text{EDGES}_{//}(p'')$ and $\alpha(w, w') = k$. Then define:

$$\begin{aligned}
H &= \{h \mid h : b \to p \text{ is an un-rooted homomorphism}\} \\
X &= \{h(w) \mid h \in H\} \subseteq \text{NODES}(p) \\
\bar{X} &= \text{NODES}(p) - X\downarrow \\
Y &= \{y \mid \exists x \in X, (x, y) \in \text{EDGES}^+(p) \wedge d(x, y) = k + 1\} \\
\bar{Y} &= \text{NODES}(p) - Y\downarrow \\
Y_0 &= Y - Y\downarrow
\end{aligned}$$

We give the intuition first, then describe the construction formally. Imagine trying to find an un-rooted homomorphism h from the block b to p. We proceed top-down in p, traversing every root-to-leaf path, trying to map b into that path. The set $\bar{X}$ consists of all nodes that we need to visit when searching for h. If we succeed finding h, then we stop including further nodes down that path into $\bar{X}$. If we fail, then we end up including the entire path in $\bar{X}$. In constructing the witness t, we will extend all descendant edges in $\bar{X}$ to long chains ($u_i = n$): this prevents "accidental" un-rooted embeddings from b to this portion of t. Next, $\bar{Y}$ consists of all nodes in $\bar{X}$ plus all nodes at distance $\leq k+1$. Here we will extend the descendant edges into short chains, by taking $u_i = 0$: this prevents accidental embeddings of the edge (w, w') into this portion of the tree t. Finally, $Y_0 \subseteq \bar{Y}$ are the frontier nodes and here we consruct the witness inductively, by applying Proposition 6 to p_1''. With this basic intuition in mind, we give now the formal proof.

The sets H and X may be empty, but $\bar{X}$ is always non-empty. Notice that $\bar{X} \subseteq \bar{Y}$ and that both $\bar{X}, \bar{Y}$ are upwards closed: if $y \in \bar{X}$ and $(x, y) \in \text{EDGES}^+(p)$, then $x \in \bar{X}$, and similarly for $\bar{Y}$. Then the sets $\bar{X}$, $(\bar{Y} - \bar{X})$, and $y\downarrow$, for $y \in Y_0$ form a partition of $\text{NODES}(p)$: e.g. to check that $(y\downarrow) \cap (y'\downarrow) = \emptyset$ for every $y, y' \in Y_0$ it suffices to see that $(y, y') \notin \text{EDGES}^+(p)$. We will construct the witness t differently on each such partition.

For every $y \in Y_0$ denote p_y the subpattern of p defined by the set of nodes $\{y\} \cup y\downarrow$, i.e. $\text{NODES}(p_y) = \{y\} \cup (y\downarrow)$. There is no un-rooted homomorphism from p_1'' to p_y. Indeed, suppose there were one, $h_1 : p_1'' \to p_y$. Let $x \in X$ be such that $d(x, y) = k + 1$ (given by the definition of Y), and let $h \in H$ be s.t. $h(w) = x$ (given by the definition of X). Then one can easily check that h and h_1 together define an un-rooted homomorphism from p'' to p, contradicting the assumption in Proposition 6. By induction hypothesis the lemma holds for the pattern p_1'', hence, there exists a witness $t_y \in mod^z(p_y)$ s.t. there exists no un-rooted embedding from p_1'' to t_y. Now we define the witness t as follows. Consider each descendant edge $(x_i, y_i) \in \text{EDGES}_{//}(p)$. If $y_i \in \bar{X}$ then define $u_i = n$. If $y_i \in (\bar{Y} - \bar{X})$ then define $u_i = 0$. And if $y_i \in y\downarrow$, for some $y \in Y_0$ then define u_i as in the witness t_y. We prove that there exists no un-rooted embedding from p'' to t.

Assume $e : p'' \to t$ is such an un-rooted embedding. We first show that $e(w) \in X \cup X\downarrow$. Suppose $e(w) \notin X\downarrow$, and denote q the subpattern of p defined by the set of nodes $\bar{X}$. Then e gives us an un-rooted embedding from b to a canonical model of q, since w, the last node in b, is mapped into q. By Lemma 5, we also get an un-rooted homomorphism h from b to q and, moreover, $h(w) = e(w)$. Obviously h is also an un-rooted homomorphism from b to p, hence $h \in H$, hence $x = h(w) \in X$. Thus we have shown that $e(w) \in X \cup X\downarrow$. Then $e(w') \in Y \cup Y\downarrow$, because $d(w, w') = k + 1$, and, by the definition of an embedding, $d(e(w), e(w')) \geq k + 1$. It follows that e maps p''_1 entirely into the witness t_y, which is a contradiction.