Vertically acyclic conjunctive queries over trees *

Filip Murlak and Grzegorz Zieliński

University of Warsaw

Abstract. Seeking a manageable subclass of conjunctive queries over trees that would reach beyond tree patterns, we find that vertical acyclicity of queries is sufficient to guarantee the same complexity bounds for static analysis problems, as those enjoyed by tree patterns.

1 Introduction

Conjunctive queries (CQs) are a staple of relational databases, offering a good balance between expressive power and the computational cost of typical static analysis problems, like query containment [3]. However, over tree-structured data, like XML documents, their place is taken by tree patterns [7], which correspond to downward, acyclic CQs. The arguments in their favour are threefold.

First, allowing arbitrary CQs induces exponential increase in the complexity of query containment in the presence of schema information: for unions of tree patterns (using horizontal and vertical axes) it is ExpTime-complete in general and ExpSpace-complete under non-recursive schemas (where the depth of trees is bounded by the size of the schema) [1, 6, 9]; for unions of CQs it is 2ExpTime-complete in general (even without horizontal axes) [2] and ExpSpace-complete under non-recursive schemas (NExpTime-complete without horizontal axes) [8].

Second, each CQ (over trees) can be rewritten as a union of exponentially many polynomial-size tree patterns [5]. Hence, unions of tree patterns and unions of CQs are equiexpressive languages, but the latter is more succinct. By choosing tree patterns as the basic formalism, we put the burden of dealing with the exponential blow-up on the user's shoulders. An advantage—from the user's point of view—is that the computational cost is more directly reflected in the size of queries, making it easier to control when writing them.

Finally, over tree-structured data it is more natural to navigate in the tree, rather than declaratively specifying properties of nodes in the fashion of first-order logic. For acyclic queries, this navigational approach can be easily supported with appropriate syntax, as illustrated palpably by the popularity of the XPath query language. Certain attempts of going beyond acyclic queries have been made with the introduction of path intersection operator to XPath 2.0, but there seems to be no natural way of dealing with full CQs.

Still, following [2], one can ask how much of the succinctness of full conjunctive queries can be allowed without compromising the complexity bounds. A partial answer was given in [8]: without influencing the complexity bounds,

* Supported by Poland's National Science Center grant 2013/11/D/ST6/03075.

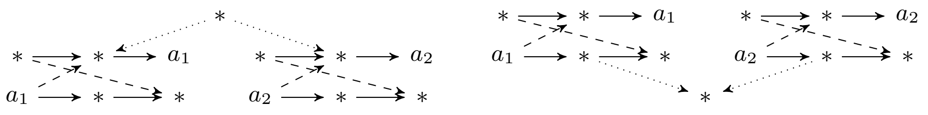

Fig. 1. A tree pattern with horizontal CQs (left) and a vertically acyclic CQ (right). Solid, dashed, and dotted edges represent next-sibling, following-sibling, and descendent relations; asterisk represents a node with an unspecified label. The left query checks if both a_1 and a_2 occur twice among siblings, separated by 1 or 2 nodes; the right query checks if this happens along a single branch.

one can extend tree patterns with arbitrary horizontal CQs over siblings (see Fig. 1); arbitrary joins with horizontal CQs cannot be allowed, as they may induce additional vertical relations, breaking the acyclicity condition. The argument combines the well-known automata construction for tree patterns with a new construction of an exponential-size deterministic automaton over words, equivalent to a given horizontal CQ.

It is well known, however, that an exponential-size automaton can be constructed for any acyclic CQ (see e.g. [4]). Since acyclic CQs can navigate up and down the tree, the construction is more involved than for tree patterns (which only go down). Can it be used to extend the result of [8] from tree patterns with horizontal CQs to *vertically acyclic* CQs (see Fig. 1)? We show that the answer is yes, but rather unexpectedly it requires substantial additional effort. The construction for the horizontal patterns provided by [8] is too weak: we need an automaton that finds all matches of the horizontal CQ in the input word, not just some match. In other words, we need a way to run multiple copies of the original automaton without affecting drastically the size of the state-space.

The reminder of the paper is organized as follows: in Section 2 we recall the basic notions, in Section 3 we construct the enhanced automaton for horizontal CQs, and in Section 4 we combine it with the construction for acyclic CQs.

2 Preliminaries

XML documents and trees. We model XML documents as unranked labelled trees. Formally, a *tree* over a finite labelling alphabet Γ is a relational structure $\mathcal{T} = \langle T, \downarrow, \downarrow^+, \rightarrow, \xrightarrow{+}, (a^{\mathcal{T}})_{a\in\Gamma}\rangle$, where

- the set T is a finite unranked tree domain, i.e., a finite prefix-closed subset of $\mathbb{N}^*$ such that $v \cdot i \in T$ implies $v \cdot j \in T$ for all $j < i$;
- the binary relations $\downarrow$ and $\rightarrow$ are the child relation ($v \downarrow v \cdot i$) and the next-sibling relation ($v \cdot i \rightarrow v \cdot (i + 1)$);
- $\downarrow^+$ and $\xrightarrow{+}$ are transitive closures of $\downarrow$ and $\rightarrow$;
- $(a^{\mathcal{T}})_{a\in\Gamma}$ is a partition of the domain T into possibly empty sets.

We write $|\mathcal{T}|$ to denote the number of nodes of tree $\mathcal{T}$. The partition $(a^{\mathcal{T}})_{a\in\Gamma}$ defines a labelling of the nodes of $\mathcal{T}$ with elements of Γ, denoted by $\ell_{\mathcal{T}}$.

Tree automata. We abstract schemas as tree automata. We use a variant in which the state in a node v depends on the states in the previous sibling and the last child of v. Formally, an automaton is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$, where Σ is the labelling alphabet (the set of element types in our case), Q is the state space with the initial state q_0 and final states F, and $\delta \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation. A *run* of $\mathcal{A}$ over a tree $\mathcal{T}$ is a labelling ρ of the nodes of $\mathcal{T}$ with the states of $\mathcal{A}$ such that for each node v with children $v \cdot 0, v \cdot 1, \ldots, v \cdot k$ and previous sibling w, $\big(\rho(w), \rho(v \cdot k), \ell_{\mathcal{T}}(v), \rho(v)\big) \in \delta$. If v has no previous sibling, $\rho(w)$ in the condition above is replaced with q_0. Similarly, if v has no children, $\rho(v \cdot k)$ is replaced with q_0. The language of trees *recognized* by $\mathcal{A}$, denoted by $L(\mathcal{A})$, consists of all trees admitting an *accepting* run of $\mathcal{A}$, i.e. a run that assigns one of the final states to the root.

A schema is *nonrecursive*, if the depth of trees it accepts is bounded by a constant dependent on the schema. For schemas defined with automata, this constant is always at most the number of states.

CQs and patterns. A conjunctive query (CQ) over alphabet Γ is a formula of first order logic using only conjunction and existential quantification, over unary predicates $a(x)$ for $a \in \Gamma$ and binary predicates $\downarrow, \downarrow^+, \to, \xrightarrow{+}$ (referred to as *child*, *descendant*, *next sibling*, and *following sibling*, respectively). Since we work only with Boolean queries, to avoid unnecessary clutter we often skip the quantifiers, assuming that all variables are by default quantified existentially.

An alternative way of looking at CQs is via patterns. A *pattern* π over Γ can be presented as $\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi \rangle$ where ℓ_π is a partial function from V to Γ, and $\langle V, E_c \cup E_d \cup E_n \cup E_f \rangle$ is a finite graph whose edges are split into child edges E_c, descendant edges E_d, next-sibling edges E_n, and following-sibling edges E_f. By $\|\pi\|$ we mean the size of the underlying graph.

We say that a tree $\mathcal{T} = \langle T, \downarrow, \downarrow^+, \to, \xrightarrow{+}, (a^{\mathcal{T}})_{a \in \Gamma} \rangle$ *satisfies* a pattern $\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi \rangle$, denoted $\mathcal{T} \models \pi$, if there exists a homomorphism $h \colon \pi \to \mathcal{T}$, i.e., a function $h : V \to T$ such that

- $h : \langle V, E_c, E_d, E_n, E_f \rangle \to \langle T, \downarrow, \downarrow^+, \to, \xrightarrow{+} \rangle$ is a homomorphism of relational structures; and
- $\ell_{\mathcal{T}}(h(x)) = \ell_\pi(x)$ for all x in the domain of ℓ_π.

Each pattern can be seen as a CQ, and vice versa. In what follows we use the terms "pattern" and "CQ" interchangeably. *Tree patterns* are patterns whose underlying graph is a directed tree with edges (of the four kinds) pointing from parents to children.

3 Finding all matches of a horizontal pattern

We write V^w for the set of positions of word w, $\{1, 2, \ldots, |w|\}$, and use $\leq$ and $+1$ for the standard order and successor on the positions of words. A *horizontal* pattern uses only $\to$ and $\xrightarrow{+}$ edges. Homomorphisms into words are defined just like for trees. Whenever we write $h : \pi \to w$, we implicitly assume that h is a homomorphism. By a *pattern π rooted at vertex $x \in V^\pi$* we mean a pair (π, x); for $i \in V^w$ we write $w, i \models (\pi, x)$ if there is $h\colon \pi \to w$ such that $h(x) = i$.

The aim of this section is to prove the following theorem, which provides an economic construction of an automaton finding all matches of a rooted horizontal pattern in the input word, used crucially in the proof of the main result.

Theorem 1. *For each rooted horizontal pattern (π, x) one can construct in polynomial working space an exponential-size deterministic automaton $\textsc{Match}_{\pi,x}$ recognizing the language of words $(a_1, \alpha_1)(a_2, \alpha_2) \dots (a_n, \alpha_n) \in (\Gamma \times \{\top, \bot\})^*$ such that for $w = a_1 a_2 \dots a_n$ and all $i \in V^w$, $\alpha_i = \top$ if and only if $w, i \models (\pi, x)$.*

We shall think of $\textsc{Match}_{\pi,x}$ as a decision procedure reading letter by letter a word $w \in \Gamma^*$ and an additional labelling $\alpha \in \{\top, \bot\}^{|w|}$, storing some information in the working memory of size polynomial in $\|\pi\|$ and independent from $|w|$.

Earliest matchings. Like in [8], $\textsc{Match}_{\pi,x}$ will look for the earliest (leftmost) matchings witnessing $w, i \models (\pi, x)$. A *word with infinity* $w\infty$ is a word w extended with an additional position ∞, greater then all original positions, that is its own successor, and bears all the labels in Γ. Over words with infinity we use $\lessdot$ to denote the composition of $\leq$ and the successor relation, which can be equivalently defined as: $x \lessdot y$ iff $x < y$ or $x = y = \infty$. The notion of homomorphism extends naturally to words with infinity. Note that since ∞ is not successor of any ordinary position, there can be no $\to$ edge between a vertex mapped to an ordinary position and a vertex mapped to ∞, but there can be a $\to$ edge between two vertices mapped to ∞. The source and target of each $\xrightarrow{+}$ edge must be mapped to positions $i \lessdot j$. Note that each homomorphism into $w\infty$ induces by restriction a partial homomorphism into w (we shall blur the distinction between them), but not every partial homomorphism into w extends to a homomorphism into $w\infty$.

Definition 1. *Let $g, h\colon \pi \to w\infty$ be two homomorphisms.*

- *We write $g \leq h$ if $g(x) \leq h(x)$ for each vertex x of π.*
- *We define $\min(g, h)\colon \pi \to w\infty$ as $\min(g, h)(x) = \min(g(x), h(x))$.*
- *We say that h* respects *relation $\eta \subseteq V^\pi \times V^w$, if $h \subseteq \eta \cup V^\pi \times \{\infty\}$.*

Note that h constantly equal to ∞ respects each η. We will be mostly interested in two special cases of η: extending a fixed partial homomorphism, and mapping the pattern's root to a fixed position (or one of several fixed positions), but for conciseness we treat them in this abstract fashion.

Lemma 1. *1. For all $g, h\colon \pi \to w\infty$, $\min(g, h)$ is a homomorphism.*
2. There exists $h_{\min}\colon \pi \to w\infty$ such that $h_{\min} \leq h$ for all $h\colon \pi \to w\infty$.
3. For each relation $\eta \subseteq V^\pi \times V^{w\infty}$, there exists $h^\eta_{\min}\colon \pi \to w\infty$ respecting η such that $h^\eta_{\min} \leq h'$ for each $h'\colon \pi \to w\infty$ respecting η. □

We call the unique $h_{\min}$ from Lemma 1 the *earliest matching* of π in $w\infty$, and $h^\eta_{\min}$ the earliest matching respecting η. Note that $h_{\min} = h^\eta_{\min}$ for $\eta = V^\pi \times V^w$. Of course, pattern π can be matched in word w if and only if the earliest matching $h_{\min}\colon \pi \to w\infty$ maps no vertex to ∞, and similarly for matchings respecting η.

Algorithm. The procedure $\textsc{Match}_{\pi,x}$ works with components of π, called *firm subpatterns*, described in Definition 3.

Definition 2. *A* $\rightarrow$-component *of* π *is a maximal connected subgraph of the* $\rightarrow$*-graph of* π*. In the* graph of $\rightarrow$-components of π*, denoted* G_π*, there is an edge from a* $\rightarrow$*-component* π_1 *to a* $\rightarrow$*-component* π_2 *if there is a* $\overset{+}{\rightarrow}$ *edge in* π *from a vertex of* π_1 *to a vertex of* π_2*.*

Definition 3. *A pattern* π *is* firm *if* G_π *is strongly connected. In general, each strongly connected component* X *of* G_π *defines a* firm subpattern *of* π*: the subgraph of* π *induced by the vertices of* $\rightarrow$*-components contained in* X*. The* DAG of firm subpatterns *of* π*, denoted* F_π*, is the standard DAG of strongly connected components of* G_π*.*

All four horizontal subpatterns in Fig. 1 are firm, but have two $\rightarrow$-components.

$\textsc{Match}_{\pi,x}(w)$ does two independent checks, the positive and the negative; it accepts when both checks accept. The negative check should reject if it finds a matching of π that maps x to a position with $\bot$, and accept otherwise. We read the input word w from left to right letter by letter, trying to build the earliest matching of π that maps x to a $\bot$ position. To process position i,

– compute the earliest matching in $w_{1..i}\infty$ that extends the constructed partial homomorphism into $w_{1..i-1}$ and maps x to a $\bot$ position.

The positive check is dual: it should accept if it can find matchings for all positions with $\top$. To keep the used space bounded, instead of running a separate check for each position with $\top$, we run a single test, but we partially reset the constructed partial homomorphism each time we see $\top$. We keep a look-ahead of size $\|\pi\|$; that is, when position i is being processed, we already have read the input word up to position $i + \|\pi\|$. To process position i,

– if $\alpha(i) = \top$, reset the partial homomorphism constructed so far: restrict it to components not reachable from the firm component containing x;
– compute the earliest matching in $w_{1..i+\|\pi\|}\infty$ that extends the currently stored partial homomorphism and maps x to the most recent position $j \leq i$ with $\alpha(j) = \top$;
– if $\alpha(i) = \top$, but the root component cannot be matched within $w_{1..i+\|\pi\|}$, reject immediately.

Accept if after processing the whole w a full homomorphism $\pi \rightarrow w$ is found.

Invariants. Correctness of the algorithm follows from the following invariants:

Invariant 1 The partial homomorphism computed by the negative check after reading the i-th letter is the earliest matching of π in $w_{1..i}\infty$ mapping x to a $\bot$ position.

Invariant 2 The partial homomorphism h computed by the positive check after processing the i-th letter is the earliest matching of π in $w_{1..i+\|\pi\|}\infty$ mapping x to the most recent $\top$ position in $w_{1..i}$ (not mapping it at all, i.e., respecting $\bigcup_{y\neq x}\{y\} \times \{1, 2, \ldots, i + \|\pi\|\}$, if there has been no $\top$ position yet); for each earlier $\top$ position j, the earliest matching $h_j : \pi \to w_{1..i+\|\pi\|}\infty$ mapping x to j, satisfies $h_j \leq h$.

Invariant 1 follows immediately from Lemma 2, by induction on i.

Lemma 2. *The earliest matchings $h : \pi \to u\infty$ and $h' : \pi \to uu'\infty$ respecting $\eta \subseteq V^\pi \times V^u$ always agree over vertices mapped to u by h.* □

For Invariant 2, we use the following properties of the earliest matchings.

Lemma 3. *Let $h_0 : \pi \to u\infty$ be the earliest matching respecting $\bigcup_{y\neq x}\{y\} \times V^u$, and let $h_i : \pi \to u\infty$ be the earliest matching that maps the root x of π to $i \in V^u$.*

1. *For every node y, whose component cannot be reached from the component of the root x, $h_i(y) = h_j(y)$ for all i, j.*
2. *If $h_i(x) = i$ and $1 \leq i \leq j$, then $h_i \leq h_j$.* □

Slightly abusing notation, we shall apply the same symbol h_j from Lemma 3 for different words; that is, $h_j : \pi \to w_{1..i-1+\|\pi\|}\infty$ is not the same as $h_j : \pi \to w_{1..i+\|\pi\|}\infty$ (although, by Lemma 2, the latter extends the former).

Before reading the first letter, Invariant 2 is satisfied: the empty partial homomorphism is the earliest matching in the empty word (with infinity), and the second part trivialises. Assume that Invariant 2 holds for $i - 1$ and let us see that it holds for i. Let g_{i-1} be the partial homomorphism constructed so far, and let g_i be the one to be constructed. By Invariant 2 for $i - 1$, g_{i-1} is the earliest matching of π in $w_{1..i-1+\|\pi\|}\infty$ that maps root x to the most recent $\top$ position, or using notation from Lemma 3, $g_{i-1} = h_j : \pi \to w_{1..i-1+\|\pi\|}\infty$ for some $0 \leq j \leq i - 1$. If $\alpha(i) = \bot$, the algorithm computes g_i as the earliest matching extending the partial homomorphism g_{i-1} and mapping x to the most recent $\top$ position in $w_{1..i+\|\pi\|}$—which is also the most recent $\top$ position in $w_{1..i-1+\|\pi\|}$—and we conclude by Lemma 2. Assume that $\alpha(i) = \top$. Then, the algorithm computes g'_{i-1}, by restricting g_{i-1} to components that are not reachable from the root component. By Lemma 2, $g'_{i-1} \subseteq h_j : \pi \to w_{1..i+\|\pi\|}\infty$. By Lemma 3 (1), $g'_{i-1} \subseteq h_i : \pi \to w_{1..i+\|\pi\|}\infty$. Hence, the earliest matching in $w_{1..i+\|\pi\|}\infty$ that extends g'_{i-1} and maps x to i is equal to $h_i : \pi \to w_{1..i+\|\pi\|}\infty$. Since this is exactly how g_i is computed, we have $g_i = h_i : \pi \to w_{1..i+\|\pi\|}\infty$. This concludes the proof of the first part of Invariant 2. To prove the second part, assume that i is not the first $\perp$ position; that is, $0 < j < i$. Since we have not rejected yet, $g_{i-1} = h_j : \pi \to w_{1..i-1+\|\pi\|}\infty$ matches the root component of π and $g_{i-1}(x) = j$. By Lemma 2, also $h_j : \pi \to w_{1..i+\|\pi\|}\infty$ maps x to j (rather than to ∞). Hence, the second part of Invariant 2 for i follows from Lemma 3 (2) and Invariant 2 for $i - 1$.

Correctness. Let us begin with a simple observation. It is routine to check that each homomorphic image of a firm pattern π_0 is a subword of length at most $\|\pi_0\|$. Consequently, if a node y of the pattern is matched at position i, then all firm components from which the component of y is reachable, must be matched before position $i + \|\pi\|$: they can reach at most $\|\pi\|$ positions forward.

If the algorithm accepts, Invariant 2 guarantees that all suitable homomorphisms exist: indeed, if the earliest matching for the last $\top$ position does not use ∞, neither do the ones for the earlier $\top$ positions.

Assume that the algorithm rejects. That means that $h_i \colon \pi \to w_{1..i+\|\pi\|}\infty$ does not match the root component (that is, matches it to ∞). Assume that $h_i \colon \pi \to w\infty$ does match the root component. Then it maps x to i, so by the observation above it maps all the components from which the component of y is reachable, within $w_{1..i+\|\pi\|}$. Restricting $h_i \colon \pi \to w\infty$ to these components and extending with infinity to other vertices of π would give a matching earlier then $h_i \colon \pi \to w_{1..i+\|\pi\|}\infty$ — a contradiction. Thus, $h_i \colon \pi \to w\infty$ is not a total homomorphism into w, so $w, i \not\models (\pi, x)$ and the algorithm rejects correctly.

Memory bound. Now that we have seen that $\textsc{Match}_{\pi,x}(w)$ is correct, we need to bound the memory it uses. We claim that while processing position i the algorithm only needs to have access to positions between $i - \|\pi\|$ and $i + \|\pi\|$. That means that the algorithm only needs to store last $2 \cdot |\pi| + 1$ symbols read, plus the matching constructed so far, restricted to this suffix.

We shall use a dualized version of the observation made previously: if a node y of the pattern is matched at position i, then all firm components reachable from the component of y must be matched after position $i - \|\pi\|$: they can reach at most $\|\pi\|$ positions back.

For the negative check, the claim is almost obvious. After reading a new symbol we match the subpatterns greedily, until no more can be matched. Among matched patterns each has an ancestor (possibly itself) that touches the last symbol (otherwise, it would have been matched before). By the dualized observation none of these patterns can reach back further than $\|\pi\|$, so it suffices to store last $\|\pi\|$ symbols read.

For the positive check the situation is similar, except that before we start matching we first check if $\|\pi\|$ positions back we had $\top$ or $\bot$. If it was $\bot$ we proceed essentially like in the negative check. But if it was $\top$, we rematch the firm components reachable from the component of the root x, in such a way that x is matched $\|\pi\|$ positions back. Again, by the dualized observation, the rematched components can reach at most $\|\pi\|$ more positions back, so they also fit within the intended window.

4 The main result

Using the matching procedure we prove our main result: we show that extending acyclic CQs using vertical axes with arbitrary horizontal CQs over siblings does not increase the complexity of the containment problem. In fact, we shall work with a more general problem of *satisfiability of Boolean combinations*, or BC-SAT: given a Boolean combination of patterns φ and an automaton $\mathcal{A}$, decide if there is a tree $\mathcal{T} \in L(\mathcal{A})$ such that $\mathcal{T} \models \varphi$. Containment for unions of CQs corresponds to non-satisfiability for Boolean combinations of the form $\pi \wedge \neg\pi_1 \wedge \neg\pi_2 \wedge \cdots \wedge \neg\pi_k$. Since we work with boolean combinations anyway, each disconnected pattern can be replaced with a conjunction of connected patterns. Thus, we shall only consider connected patterns from now on.

Assuming that vertical edges propagate through horizontal edges (e.g., next sibling of a child is also explicitly connected with a child edge), one can define *vertically acyclic* patterns as those whose $\downarrow, \downarrow^+$-subgraph is acyclic in the undirected sense; since patterns are connected, it is then an undirected tree. Without this assumption we define this notion as follows. A *horizontal component* of pattern π is a connected component of the $\rightarrow, \xrightarrow{+}$-subgraph of π. Let $H_\pi = \langle V_\pi, \downarrow, \downarrow^+ \rangle$ be a graph over horizontal components of π, where edge $X \downarrow Y$ is present if $x \downarrow y$ for some $x \in X$ and $y \in Y$, and $X \downarrow^+ Y$ is present if $x \downarrow^+ y$ for some $x \in X$ and $y \in Y$, but there is no edge $X \downarrow Y$. We say that pattern π is *vertically acyclic*, if this graph is acyclic in the undirected sense. Again, as π is connected, the graph is an undirected tree.

Given a vertically acyclic pattern π, by simple preprocessing we ensure that there is at most one edge in π between each two horizontal components: if there are more, we can merge their starting points in the parent/ancestor horizontal component, and drop all edges except one (a child edge, if there is one).

Theorem 2. *For vertically acyclic patterns,* BC-SAT *is* EXPTIME*-complete, and* PSPACE*-complete under non-recursive schemas.*

Proof. We shall see that for a vertically acyclic pattern π one can construct an equivalent automaton $\mathcal{A}_\pi$ *in polynomial working space* (states have polynomial-size representations and all ingredients of $\mathcal{A}_\pi$ can be enumerated in polynomial working space). Despite its nondeterminism, we will be able to complement $\mathcal{A}_\pi$ by simply changing accepting states. Hence, one can easily reduce BC-SAT to nonemptiness of tree automata, by constructing in polynomial working space an automaton $\mathcal{A}_\varphi$ equivalent to a given boolean combination φ. The product $\mathcal{B}$ of $\mathcal{A}_\varphi$ with a nonrecursive automaton $\mathcal{A}$ can be tested for nonemptiness by a nondeterministic algorithm using space $\mathcal{O}(\|\mathcal{A}\| \cdot \log \|\mathcal{A}\| \cdot \mathit{poly}(\|\varphi\|))$, which guesses a tree of depth at most $\|\mathcal{A}\|$ in the depth-first order and nondeterministically evaluates $\mathcal{B}$ over it, processing it in the post-order fashion. By Savitch's theorem, this gives a PSPACE algorithm for satisfiability under nonrecursive schemas. As $\mathcal{A}_\varphi$ is singly exponential in $\|\varphi\|$, the standard polynomial-time emptiness test gives an EXPTIME algorithm for satisfiability under arbitrary schemas.

The idea is that the automaton $\mathcal{A}_\pi$ guesses in each node which subpatterns are matched there, and then checks that these guesses are consistent: a subpattern is matched in some node if and only if its root can be matched there and all its sub-subpatterns can be matched in appropriate nodes. To ensure that these dependencies are well founded, we root the undirected tree forming the graph H_π, by arbitrarily fixing a root component and treating all the edges as pointing outwards. This builds the structure of a directed tree over the graph H_π; the orientation of edges in this directed tree is entirely unrelated to the character of the corresponding edges in the pattern: child and descendent edges can point both up and down the tree. We use this structure to speak precisely of subpatterns: for each horizontal component X of π we define subpattern $\pi.X$ obtained by restricting π to X and all descendants of X in the directed tree. If component X has successors $Y_1, Y_2, \ldots, Y_n$ in the directed tree, then the subpattern $\pi.X$ is composed of the nodes in the horizontal component X connected to subpatterns $\pi.Y_1, \pi.Y_2, \ldots, \pi.Y_n$ with child or descendant edges, pointing to or form X. We call patterns $\pi.Y_1, \pi.Y_2, \ldots, \pi.Y_n$ the *immediate subpatterns* of pattern $\pi.X$. Unless X is the root component of the whole directed tree, it is connected to some other horizontal component Z, with a child or descendent edge, pointing to or from X. If the edge points to X, we call $\pi.X$ a *down-subpattern.* Otherwise, we call it an *up-subpattern.* In either case, we call the vertex of X that is the end of this edge, the *root vertex* of $\pi.X$; if X is the root component of the whole tree, we choose an arbitrary vertex in X for the root vertex. We say that $\pi.X$ is matched at node v if its root vertex is mapped to v.

The automaton $\mathcal{A}_\pi$, before reading the sequence of children of a node v, non-deterministically selects the set $\mathit{Expected}(v)$ of subpatterns that can be matched at v. Similarly, $\mathcal{A}_\pi$ non-deterministically selects a set $\mathit{ExpectedAbove}(v)$ of subpatterns that can be matched at some ancestor of v. (Note that for down-subpatterns the exact place where the root vertex is matched among siblings is irrelevant; we make the distinction only for the purpose of uniform treatment by the procedure match.) After selecting $\mathit{Expected}(v)$ and $\mathit{ExpectedAbove}(v)$, the automaton proceeds to read the sequence of children of v. While doing so it computes the set $\mathit{Matched}(v)$ of subpatterns of π matched in the children of v and the set $\mathit{MatchedBelow}(v)$ of subpatterns that were matched in the children of some descendant of v, and performs a number of consistency checks using a slightly modified version of the procedure MATCH, described below.

With each subpattern $\pi.X$ we associate a horizontal pattern π_X obtained by restricting π to X and including in the label of each vertex x the set Φ of immediate subpatterns $\pi.Y$ of $\pi.X$ connected to x. In the modified procedure MATCH for π_X, a vertex labelled with (σ, Φ) can be matched in a position labelled with (τ, Ψ) only if $\sigma = \tau$ and $\Phi \subseteq \Psi$. It is straightforward to check that this does not influence the correctness of MATCH. Observe that the extended alphabet is exponential, but each symbol can be stored in polynomial memory. Hence, MATCH still works in memory polynomial in the size of the pattern.

Reading the sequence of children of v, the automaton performs simultaneously (using the product construction) the following tasks:

1. Check that $\mathit{Expected}(v)$ and $\mathit{ExpectedAbove}(v)$ do not contradict the choices for the children of v. The subpatterns expected to be matched in an ancestor of a child w must either be matched in the parent of w or an ancestor of the parent of w; that is, $\mathit{ExpectedAbove}(w) = \mathit{Expected}(v) \cup \mathit{ExpectedAbove}(v)$ must hold for each child w. If for some w it fails to hold, the computation stops and no state is assigned to v, because the guesses are inconsistent.

2. Check that *Expected*(w) was chosen correctly for each child w. This is done by simultaneously running procedure MATCH for each subpattern $\pi.X$ (with the fixed root vertex) over the children of v. For each child w, the procedure is fed with an extended symbol that consists of:
   - the label of node w,
   - the set of immediate subpatterns of $\pi.X$ matched below w (computed from *Matched*(w) and *MatchedBelow*(w), both empty for leaves), and above w (computed from *Expected*(v) and *ExpectedAbove*(v)),
   - $\top$ if $\pi.X \in$ *Expected*(w), and $\bot$ otherwise.

   If some instance of MATCH rejects, the computation stops and no state is assigned to v, since the guesses made below have proved to be incorrect.
3. Compute the sets *Matched*(v) and *MatchedBelow*(v), based on *Matched*(w), *MatchedBelow*(w), and *Expected*(w) for each child w of v.

The automaton treats the root as any other node; that is, it performs the tasks above for the single-node sequence of siblings consisting only from the root, treating it as a child of a dummy node #. The automaton accepts if: $\mathit{Expected}(\#) = \mathit{ExpectedAbove}(\#) = \emptyset$ and $\pi \in \mathit{Matched}(\#) \cup \mathit{MatchedBelow}(\#)$.

By the structural induction over subpatterns one easily proves that in any complete run (accepting or rejecting) of the described automaton $\mathcal{A}_\pi$, the sets *Expected*(v), *ExpectedAbove*(v), *Matched*(v), and *MatchedBelow*(v) are guessed or computed correctly for each node of the input tree. Hence, the construction is correct. Moreover, for each node v of the input tree, there exists exactly one correct set *Expected*(v) and one correct set *ExpectedAbove*(v). Hence, for each input tree there is exactly one complete run, modulo the last guess of *Expected*$(\#)$ and *ExpectedAbove*$(\#)$. Consequently, the automaton $\mathcal{A}_\pi$ can be complemented simply by changing the acceptance condition to: $\mathit{Expected}(\#) = \mathit{ExpectedAbove}(\#) = \emptyset$ and $\pi \notin \mathit{Matched}(\#) \cup \mathit{MatchedBelow}(\#)$. $\square$

References

1. M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM* 55(2), 2008.
2. H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. Proc. MFCS 2008: 132–143.
3. A. K. Chandra, P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. Proc. STOC 1977: 77–90.
4. N. Francis, C. David, L. Libkin. A Direct Translation from XPath to Nondeterministic Automata. Proc. AMW 2011.
5. G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53 (2006), 238–272.
6. M. Marx. XPath with conditional axis relations. Proc. EDBT 2004: 477–494.
7. G. Miklau, M. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM* 51(1): 2–45, 2004.
8. F. Murlak, M. Ogiński, M. Przybyłko. Between Tree Patterns and Conjunctive Queries: Is There Tractability beyond Acyclicity? Proc. MFCS 2012: 705-717.
9. F. Neven, T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Meth. in Comp. Sci.* 2(3): 1–30, 2006.