The Complexity of Tree Automata and XPath on Grammar-Compressed Trees

Markus Lohrey [a], Sebastian Maneth [b]

[a] *FMI, Universität Stuttgart, Germany*

[b] *National ICT Australia Ltd, Sydney, Australia*

Abstract

The complexity of various membership problems for tree automata on compressed trees is analyzed. Two compressed representations are considered: dags, which allow to share identical subtrees in a tree, and straight-line context-free tree grammars, which moreover allow to share identical intermediate parts in a tree. Several completeness results for the classes NL, P, and PSPACE are obtained. Finally, the complexity of the evaluation problem for (structural) XPath queries on trees that are compressed via straight-line context-free tree grammars is investigated.

Key words: tree automata, XPath, tree compression

1 Introduction

During the last decade, the massive increase in the volume of data has motivated the investigation of algorithms on *compressed data*, like for instance compressed strings, trees, and pictures. The general goal is to develop algorithms that directly work on compressed data without prior decompression. Considerable amount of work has been done concerning algorithms on compressed strings, see e.g. [6,17,27]. In this paper we investigate the computational complexity of algorithmic problems on *compressed trees*. Trees serve as fundamental data structure in many fields of computer science, e.g. term rewriting, model checking, XML, etc. In each of these domains, compressed trees in form of *dags* (directed acyclic graphs), which allow to share identical subtrees in a tree, are used as a key for obtaining efficient algorithms,

Email addresses: `lohrey@informatik.uni-stuttgart.de` (Markus Lohrey), `sebastian.maneth@nicta.com.au` (Sebastian Maneth).

see for instance [26] (term graph rewriting), [3] (model checking with BDDs), and [4,11,21] (querying compressed XML documents). Recently, *straight-line context-free tree grammars* (SL cf tree grammars) were proposed as another compressed representation of trees in the context of XML [19,5]. Whereas a dag can be seen as a *regular* tree grammar [7] that generates exactly one tree, an SL cf tree grammar is a *context-free tree grammar* [7] that generates exactly one tree. SL cf tree grammars allow to share identical intermediate parts in a tree. This results in better compression rates in comparison to dags: in the theoretical optimum, SL cf tree grammars lead to doubly exponential compression rates, whereas dags only allow singly exponential compression rates. In [5], a practical algorithm (BPLEX) for generating a small SL cf tree grammar that produces a given input tree is presented. Experiments with existing XML benchmark data show that BPLEX results in significantly better compression rates than dag-based compression algorithms.

In Section 3 we study the problem of evaluating compressed trees via *tree automata* [7,12]. Tree automata play a fundamental role in many applications where trees have to be processed in a systematic way. In the context of XML, for instance, tree automata are used to type check documents against an XML type [22,23]. These applications motivate the investigation of general decision problems for tree automata like emptiness, equivalence, and intersection nonemptiness. Several complexity results are known for these problems, see e.g. [7]. Membership problems for tree automata were investigated in [16] for ranked trees (see Table 1 for the results of [16]) and [28] for unranked trees from the perspective of computational complexity. Here we extend this line of research by investigating the computational complexity of membership problems for various classes of tree automata on compressed trees (dags and SL cf tree grammars). For deterministic/nondeterministic top-down/bottom-up tree automata we analyze the fixed membership problem (where the tree automaton is not part of the input) as well as the uniform membership problem (where the tree automaton is also part of the input). Moreover, we also consider subclasses of SL cf tree grammars that allow more efficient algorithms for evaluating tree automata. In particular, linearity and the restriction that for some constant k, every production of the SL cf tree grammar contains at most k parameters (variables) lead to better complexity bounds. For all cases, we present upper and lower bounds which vary from NL (nondeterministic logspace) to PSPACE (polynomial space). Our results are collected in Table 1. We also briefly consider the parameterized complexity [10] of membership problems for tree automata.

In Section 4 we consider the problem of evaluating core XPath expressions over compressed trees. XPath is a widely used language for selecting nodes in XML documents and is the core of many modern XML technologies. The query problem for XPath asks whether a given node in a given (unranked) tree is selected by a given XPath expression. For uncompressed trees, the complexity of this problem is intensively studied in [13,14]. For input trees that are represented as dags, XPath evaluation was investigated in [4,11,21]. In [11] it was shown that the evaluation problem for core XPath (the navigational part of XPath) over dag-compressed trees is PSPACE-complete. Here, we extend this result to linear SL cf tree grammars (Theorem 9). This is remarkable, since linear SL cf tree grammars lead to (provably) better compression rates than dags, which is also confirmed by our experimental results for the BPLEX-algorithm (which produces linear SL cf tree grammars) from [5].

A short version of this paper appeared in [18].

2 Preliminaries

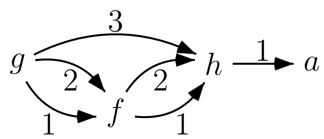
For background in complexity theory see [24]. The set of all finite strings over a (not necessarily finite) alphabet Σ is Σ^*. The empty string is ε. The length of a string u is $|u|$. We write $u \preceq v$ for $u, v \in \Sigma^*$ if u is a prefix of v. The reflexive and transitive closure of a binary relation $\to$ is denoted by $\xrightarrow{*}$.

2.1 Trees, dags, and SL cf tree grammars

A *ranked alphabet* is a pair $(\mathcal{F}, \mathrm{arity})$, where $\mathcal{F}$ is a finite set of function symbols and $\mathrm{arity} : \mathcal{F} \to \mathbb{N}$ assigns to each $\alpha \in \mathcal{F}$ its arity (or rank). Let $\mathcal{F}_i = \{\alpha \in \mathcal{F} \mid \mathrm{arity}(\alpha) = i\}$. Function symbols in $\mathcal{F}_0$ are called *constants*. In all examples we use symbols $a \in \mathcal{F}_0$ and $f \in \mathcal{F}_2$. Mostly we omit the function arity in the description of a ranked alphabet. An *$\mathcal{F}$-labeled tree* t (or *ground term* over $\mathcal{F}$) is a pair $t = (\mathrm{dom}_t, \lambda_t)$, where (i) $\mathrm{dom}_t \subseteq \mathbb{N}^*$ is finite, (ii) $\lambda_t : \mathrm{dom}_t \to \mathcal{F}$, (iii) if $v \preceq w \in \mathrm{dom}_t$, then also $v \in \mathrm{dom}_t$, and (iv) if $v \in \mathrm{dom}_t$ and $\lambda_t(v) \in \mathcal{F}_n$, then $vi \in \mathrm{dom}_t$ if and only if $1 \leq i \leq n$. Note that the edge relation of the tree t can be defined as $\{(v, vi) \in \mathrm{dom}_t \times \mathrm{dom}_t \mid v \in \mathbb{N}^*, i \in \mathbb{N}\}$. The size of t is $|t| = |\mathrm{dom}_t|$. With an $\mathcal{F}$-labeled tree t we associate a term in the usual way: If $\lambda_t(\varepsilon) = \alpha \in \mathcal{F}_i$, then this term is $\alpha(t_1, \ldots, t_i)$, where t_j is the term that corresponds to the subtree of t rooted at the node $j \in \mathbb{N}$. The set of all $\mathcal{F}$-labeled trees is $T(\mathcal{F})$. Let us fix a countable set $\mathcal{X}$ of variables. The set of all $\mathcal{F}$-labeled trees with variables from $\mathcal{X}$ is $T(\mathcal{F}, \mathcal{X})$. Formally, we consider variables as new constants and define $T(\mathcal{F}, \mathcal{X}) = T(\mathcal{F} \cup \mathcal{X})$. A tree $t \in T(\mathcal{F}, \mathcal{X})$ is *linear*, if every variable $x \in \mathcal{X}$ occurs at most once in t. A *term rewriting system*, briefly TRS, over a ranked alphabet $\mathcal{F}$ is a finite set $\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{X}) \times T(\mathcal{F}, \mathcal{X})$ such that for all $(s, t) \in \mathcal{R}$, every variable that occurs in t also occurs in s and furthermore $s \notin \mathcal{X}$. The *one-step rewrite relation* $\to_{\mathcal{R}}$ over $T(\mathcal{F}, \mathcal{X})$ is defined as usual, see for instance [2].

Dags (directed acyclic graphs) are a popular compressed representation of trees that allows to share identical subtrees. An $\mathcal{F}$-labeled *dag* is a triple $D = (V_D, \lambda_D, E_D)$ where (i) V_D is a finite set of nodes, (ii) $\lambda_D : V_D \to \mathcal{F}$ labels each node with a symbol from $\mathcal{F}$, (iii) $E_D \subseteq V_D \times \mathbb{N} \times V_D$ (i.e. edges are directed and labeled with natural numbers), (iv) every $v \in V_D$ contains precisely one i-labeled outgoing edge for every $1 \leq i \leq \mathrm{arity}(\lambda_D(v))$, and (v) (V_D, E_D) is acyclic and contains precisely one node $\mathrm{root}_D \in V_D$ without incoming edges. The size of D is $|D| = |V_D|$. A *root-path* in D is a path $v_1, i_1, v_2, i_2, \ldots, v_n$ in the graph (V_D, E_D), i.e., $v_k \in V_D$ $(1 \leq k \leq n)$ and $(v_k, i_k, v_{k+1}) \in E_D$ $(1 \leq k < n)$ that moreover starts in the root node, i.e., $v_1 = \mathrm{root}_D$. Such a path can be identified with the label-sequence $i_1 i_2 \cdots i_{n-1} \in \mathbb{N}^*$. An $\mathcal{F}$-labeled dag D over $\mathcal{F}$ can be unfolded into an $\mathcal{F}$-labeled tree $\mathrm{eval}(D)$: $\mathrm{dom}_{\mathrm{eval}(D)}$ is the set of all root-paths in D (viewed as a subset of $\mathbb{N}^*$), and if the root-path $p \in \mathbb{N}^*$ ends in the node $v \in V_D$, then we set $\lambda_{\mathrm{eval}(D)}(p) = \lambda_D(v)$. Clearly the size of $\mathrm{eval}(D)$ is bounded exponentially in $|D|$.

Example 1 *Let D be the following dag:*



We have $\mathrm{eval}(D) = g(f(h(a), h(a)), f(h(a), h(a)), h(a))$. *Moreover, the size of D is 4. We have* $\mathrm{dom}_{\mathrm{eval}(D)} = \{\varepsilon, 1, 2, 3, 11, 12, 21, 22, 111, 121, 211, 221, 31\}$.

In this paper we also consider a compressed representation of trees recently introduced in [19], which generalizes the dag-representation: *straight-line context-free tree grammars* (*SL cf tree grammars*). An SL cf tree grammar is a tuple $G = (\mathcal{F}, N, S, P)$, where (i) $N \cup \mathcal{F}$ is a ranked alphabet, (ii) N is the set of nonterminals, (iii) $\mathcal{F}$ is the set of terminals, (iv) $S \in N$ is the start nonterminal and has rank 0, (v) P (the set of productions) is a TRS over $N \cup \mathcal{F}$ that contains for every $A \in N$ exactly one rule of the form $A(x_1, \ldots, x_n) \to t_A$, where $n = \mathrm{arity}(A)$ and $x_1, \ldots, x_n$ are pairwise different variables, and (vi) the relation $\{(A, B) \in N \times N \mid B \text{ occurs in } t_A\}$ is acyclic. These conditions ensure that for every $A \in N$ of rank n there is a unique tree $\mathrm{eval}_G(A)(x_1, \ldots, x_n) \in T(\mathcal{F}, \{x_1, \ldots, x_n\})$ such that $A(x_1, \ldots, x_n) \xrightarrow{*}_P \mathrm{eval}_G(A)(x_1, \ldots, x_n)$. We define $\mathrm{eval}(G) = \mathrm{eval}_G(S) \in T(\mathcal{F})$. The size of G is $|G| = \sum_{A \in N} |t_A|$. We say that G is an SL cf tree grammar *with k parameters* ($k \geq 0$) if $\mathrm{arity}(A) \leq k$ for every $A \in N$. The SL cf tree grammar G is *linear* if for every production $A(x_1, \ldots, x_n) \to t_A$ in P the tree t_A is linear.

An SL cf tree grammar can be seen as a context free tree grammar [7] that generates exactly one tree. Alternatively, an SL cf tree grammar is a *recursive program scheme* [8] that generates a finite tree. SL cf tree grammars generalize string generating straight-line programs [27] in a natural way from strings to trees. The following example shows that SL cf tree grammars may lead to doubly exponential compression rates; thus, they can be exponentially more succinct than dags:

Example 2 *Let the (non-linear) SL cf tree grammar G_n consist of the following productions: $S \to A_0(a)$, $A_i(x) \to A_{i+1}(A_{i+1}(x))$ for $0 \leq i < n$, and $A_n(x) \to f(x,x)$. Then* $\mathrm{eval}(G_n)$ *is a complete binary tree of height 2^n. Thus, $|\mathrm{eval}(G_n)| \in O(2^{2^n})$. Note that G_n has only one parameter.*

On the other hand, it is easy to prove by induction over the number of productions that linear SL cf tree grammars can only achieve exponential compression rates. But linear SL cf tree grammars are still more succinct than dags: The tree $h(h(\cdots h(a)\cdots))$ with 2^n many occurrences of h can be generated by a linear SL cf tree grammar of size $\mathcal{O}(n)$, which is not possible with dags.

An SL cf tree grammar $G = (\mathcal{F}, N, S, P)$ with 0 parameters (i.e., $\mathrm{arity}(A) = 0$ for every nonterminal $A \in N$) can be easily transformed in logspace into an $\mathcal{F}$-labeled dag that generates the same tree: we take the disjoint union of all right-hand sides of productions from P, where the root of the right-hand side for the nonterminal A gets the additional label A. Then we merge for every nonterminal A all nodes with label A. Note that since $\mathrm{arity}(A) = 0$ for every $A \in N$, nonterminals can only occur as leafs in right-hand sides of G. Thus, this merging process results in a dag. For instance, the SL cf tree grammar with the productions $S \to g(A, A, B), A \to f(B, B), B \to h(a)$ corresponds to the dag from Example 1. Vice versa, from an $\mathcal{F}$-labeled dag we can construct in logspace an equivalent SL cf tree grammar with 0 parameters by taking the nodes of the dag as nonterminals. Thus, dags can be seen as special SL cf tree grammars, which justifies our choice to denote with eval both the evaluation function for dags and unrestricted SL cf tree grammars.

2.2 Tree automata

A (nondeterministic) *top-down tree automaton*, briefly TDTA, is a tuple $\mathcal{A} = (Q, \mathcal{F}, q_0, \mathcal{R})$, where Q is a finite set of states, $Q \cup \mathcal{F}$ is a ranked alphabet with $\mathrm{arity}(q) = 1$ for all $q \in Q$, $q_0 \in Q$ is the initial state, and $\mathcal{R}$ is a TRS such that all rules have the form $q(\alpha(x_1, \ldots, x_n)) \to \alpha(q_1(x_1), \ldots, q_n(x_n))$, where $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n$ are pairwise different variables, and $\alpha \in \mathcal{F}$ has rank n. $\mathcal{A}$ is a *deterministic TDTA* if no two rules in $\mathcal{R}$ have the same left-hand side. The tree language that is accepted by a TDTA $\mathcal{A}$ is $T(\mathcal{A}) = \{t \in T(\mathcal{F}) \mid q_0(t) \xrightarrow{*}_{\mathcal{R}} t\}$. A (nondeterministic) *bottom-up tree automaton*, briefly BUTA, is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \mathcal{R})$, where Q and $\mathcal{F}$ are as above, $Q_f \subseteq Q$ is the set of final states, and $\mathcal{R}$ is a TRS such that all rules have the form

$\alpha(q_1(x_1), \ldots, q_n(x_n)) \to q(\alpha(x_1, \ldots, x_n))$, where $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n$ are pairwise different variables, and $\alpha \in \mathcal{F}$ has rank n. $\mathcal{A}$ is a *deterministic BUTA* if no two rules in $\mathcal{R}$ have the same left-hand side. The tree language that is accepted by a BUTA $\mathcal{A}$ is $T(\mathcal{A}) = \{t \in T(\mathcal{F}) \mid \exists q \in Q_f : t \xrightarrow{*}_{\mathcal{R}} q(t)\}$. It is straight-forward to transform a nondeterministic BUTA into an equivalent nondeterministic TDTA and vice versa, and these transformations can be performed by a logspace transducer. Thus, in the following we do not distinguish between nondeterministic BUTAs and nondeterministic TDTAs, and we call them simply tree automata (TAs). A subset of $T(\mathcal{F})$ is *recognizable* if it is accepted by a TA. Using a powerset construction, every recognizable tree language can be also accepted by a deterministic BUTA, but this involves an exponential blowup in the number of states. For deterministic TDTAs the situation is different; they only recognize a proper subclass of the recognizable tree languages. The size $|\mathcal{A}|$ of a TA is the sum of the sizes of all left and right hand sides of rules. Let $\mathcal{G}$ be a class of SL cf tree grammars (e.g., the class of all dags or the class of all linear SL cf tree grammars). The *membership problem* for the fixed TA $\mathcal{A}$ and the class $\mathcal{G}$ is the following decision problem:

INPUT: $G \in \mathcal{G}$
QUESTION: Does $\mathrm{eval}(G) \in T(\mathcal{A})$ hold?

Here, the input size is $|G|$. For a class $\mathcal{C}$ of tree automata, the *uniform membership problem* for $\mathcal{C}$ and the class $\mathcal{G}$ is the following decision problem:

INPUT: $G \in \mathcal{G}$ and $\mathcal{A} \in \mathcal{C}$
QUESTION: Does $\mathrm{eval}(G) \in T(\mathcal{A})$ hold?

In this problem the input size is $|\mathcal{A}| + |G|$.

The upper part of Table 1 collects the complexity results that were obtained in [16] for uncompressed trees. The statement that for instance the membership problem for TAs is NC^1-complete means that for every fixed TA the membership problem is in NC^1 and that there exists a fixed TA for which the membership problem is NC^1-hard. More details on tree automata can be found in [7,12].

3 Membership Problems for Dags and SL CF Tree Grammars

The time bounds in the following theorem are based on dynamic programming. Note that only the number k of parameters appears in the exponent. The idea is to run the tree automaton $\mathcal{A}$ bottom up on the right-hand sides of G's productions. For the parameters we have to assume at most n^k different possibilities of states of $\mathcal{A}$ which (a deterministic simulation of) $\mathcal{A}$ maps to a

<table>
<tr><th></th><th></th><th>det. TDTA</th><th>det. BUTA</th><th>TA</th></tr>
<tr><td rowspan="2">uncompressed trees [16]</td><td>fixed</td><td colspan="3">NC^1-complete</td></tr>
<tr><td>uniform</td><td>L-complete</td><td>LOGDCFL
L-hard</td><td>LOGCFL-complete</td></tr>
<tr><td rowspan="2">dags</td><td>fixed</td><td rowspan="2">NL-complete</td><td colspan="2" rowspan="2">P-complete</td></tr>
<tr><td>uniform</td></tr>
<tr><td rowspan="2">lin. SL+fixed number para.</td><td>fixed</td><td colspan="3" rowspan="2">P-complete</td></tr>
<tr><td>uniform</td></tr>
<tr><td rowspan="2">SL+fixed number para.</td><td>fixed</td><td colspan="2" rowspan="2">P-complete</td><td></td></tr>
<tr><td>uniform</td><td>PSPACE-complete</td></tr>
<tr><td rowspan="2">general SL</td><td>fixed</td><td rowspan="2">P-complete</td><td colspan="2" rowspan="2">PSPACE-complete</td></tr>
<tr><td>uniform</td></tr>
</table>

Table 1
Complexity results for (uniform) membership problems

state of $\mathcal{A}$.

Theorem 1 *The following holds:*

(1) For a given TA $\mathcal{A}$ with n states and a linear SL cf tree grammars G with k parameters we can check in time $\mathcal{O}(n^{k+1}\cdot|G|\cdot|\mathcal{A}|)$ whether $\mathrm{eval}(G) \in T(\mathcal{A})$.
(2) For a given deterministic BUTA $\mathcal{A}$ with n states and a given SL cf tree grammars with k parameters we can check in time $\mathcal{O}(n^k \cdot |G| \cdot |\mathcal{A}|)$ whether $\mathrm{eval}(G) \in T(\mathcal{A})$.

Proof. For (1) let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \mathcal{R})$ be a nondeterministic BUTA and let $G = (\mathcal{F}, N, S, P)$ be a linear SL cf tree grammar such that $\mathrm{arity}(A) \leq k$ for every $A \in N$. We calculate for every nonterminal A and all $(q, q_1, \ldots, q_\ell) \in Q^{\ell+1}$, where $\mathrm{arity}(A) = \ell \leq k$, a boolean value $\mathrm{ok}(A, q, q_1, \ldots, q_\ell)$, where $\mathrm{ok}(A, q, q_1, \ldots, q_\ell)$ is true if and only if $\mathrm{eval}_G(A)(q_1, \ldots, q_\ell) \xrightarrow{*}_{\mathcal{R}} q$ is true. First, note that ok has at most $|Q|^{k+1} \cdot |N|$ entries, which is polynomial, since k is a constant. We calculate the entries for ok using dynamic programming. Assume that $A(x_1, \ldots, x_\ell) \to t(x_1, \ldots, x_\ell)$ is the unique production for the nonterminal A and assume that for every nonterminal B that occurs in the tree t, all boolean values $\mathrm{ok}(B, p, p_1, \ldots, p_{\mathrm{arity}(B)})$ are already calculated. We want to calculate the boolean value $\mathrm{ok}(A, q, q_1, \ldots, q_\ell)$. For this, we simulate the deterministic powerset automaton $\mathcal{P}(\mathcal{A})$ of $\mathcal{A}$ on the tree t. It is not necessary to construct $\mathcal{P}(\mathcal{A})$ explicitly (which is not possible in polynomial time), we calculate, on the fly, only the transition of $\mathcal{P}(\mathcal{A})$ that is needed in the current simulation step. Moreover, in the unique occurrence of the variable x_i ($1 \leq i \leq \ell$) in t we start the simulation in the state q_i (more precisely, in the singleton set $\{q_i\}$). If we have evaluated all children $v_1, \ldots, v_{\mathrm{arity}(B)}$ for a node v that is labeled with a nonterminal $B \in N$ in t, then we can evaluate the node v by using the already calculated boolean values $\mathrm{ok}(B, p, p_1, \ldots, p_{\mathrm{arity}(B)})$: Assume that we have calculated for the child v_i the set $U_i \subseteq Q$. Then, the node v evaluates to the set of all $p \in Q$ such that there exists $(p_1, \ldots, p_{\mathrm{arity}(B)}) \in \prod_{i=1}^{\mathrm{arity}(B)} U_i$ with $\mathrm{ok}(B, p, p_1, \ldots, p_{\mathrm{arity}(B)}) = \mathrm{true}$. The total calculation needs time $\mathcal{O}(|Q|^{k+1} \cdot |\mathcal{A}| \cdot |G|)$, because for every $A \in N$ of rank ℓ there are $|Q|^{\ell+1}$ many tuples for which ok has to be evaluated. For every specific tuple $(A, q, q_1, \ldots, q_\ell)$ the evaluation of ok needs time $\mathcal{O}(|\mathcal{A}| \cdot |t_A|)$, where t_A is the right-hand side of A, because for every node in t_A every transition of $\mathcal{A}$ has to be considered only once. Thus, in total we need time $\sum_{A \in N} \mathcal{O}(|Q|^{\mathrm{arity}(A)+1} \cdot |\mathcal{A}| \cdot |t_A|) \leq \mathcal{O}(|Q|^{k+1} \cdot |\mathcal{A}| \cdot |G|)$. At the end, we accept if and only if $\mathrm{ok}(S, q)$ is true for some final state $q \in Q_f$ of $\mathcal{A}$.

The reader might ask, at which point in the previous algorithm it is important that G is *linear*. If G is nonlinear, then a variable x_i might have exponentially many (say ℓ_i many) occurrences in the tree $\mathrm{eval}(A)(x_1, \ldots, x_\ell)$. In a certain occurrence of the tree $\mathrm{eval}(A)(x_1, \ldots, x_\ell)$ within the tree $\mathrm{eval}(G)$ these occurrences of x_i correspond to tree nodes $v_{i,1}, \ldots, v_{i,\ell_i}$. Now we cannot expect that the automaton $\mathcal{A}$ arrives at each node $v_{i,j}$ ($1 \leq j \leq \ell_i$) in the same state q_i during a successful run on $\mathrm{eval}(G)$. In other words, in the nonlinear case, it is not sufficient to describe the behavior of $\mathcal{A}$ on the tree $\mathrm{eval}_G(A)$ by the set of all tuples $(q, q_1, \ldots, q_\ell) \in Q^{\ell+1}$ such that $\mathrm{eval}_G(A)(q_1, \ldots, q_\ell) \xrightarrow{*}_{\mathcal{R}} q$. On the other hand, if $\mathcal{A}$ is deterministic, then $\mathcal{A}$ arrives at each node $v_{i,j}$ in the same state, because at each of these nodes the same subtree is rooted. In other words, it is not necessary to substitute different states for different occurrences of the same variable in $\mathrm{eval}_G(A)(x_1, \ldots, x_\ell)$, and non-linearity does not cause any problems. This proves statement (2) for deterministic BUTAs from the theorem. $\square$

Recall that a dag can be seen as a (linear) SL cf tree grammar without parameters. Thus, Theorem 1 can also be applied to dags in order to obtain a polynomial time algorithm for the uniform membership problem for TAs and dags. A reduction from the P-complete monotone circuit-value problem (see e.g. [24]), shows:

Theorem 2 *There exists a fixed deterministic BUTA $\mathcal{A}$ such that the membership problem for $\mathcal{A}$ and dags is P-hard.*

Proof. Recall that the monotone boolean circuit-value problem is P-complete, see e.g. [24]. Let the ranked alphabet $\mathcal{F}$ contain the binary function symbols $\wedge$ and $\vee$ and the constants 0 and 1. Then, a monotonic circuit corresponds to an $\mathcal{F}$-labeled dag. The state set of the deterministic BUTA $\mathcal{A}$ is $\{0, 1\}$ and the rules of $\mathcal{A}$ correspond to the evaluation rules for $\wedge$ and $\vee$. The unique final state of $\mathcal{A}$ is 1. $\square$

Remark 1 *By Theorem 1 and 2, the (uniform) membership problem for (deterministic) BUTAs on dags is P-complete. This result may appear surprising when compared with a recent result from [1]: the membership problem for so called dag automata is NP-complete. But in contrast to our approach, a dag automaton operates directly on a dag, whereas we consider ordinary tree automata that run on the unfolded dag. This makes a crucial difference for the complexity of the membership problem.*

By the next theorem, a deterministic TDTA can be evaluated on a dag in NL (nondeterministic logspace). The crucial fact is that a deterministic TDTA $\mathcal{A}$ accepts a tree t if and only if the path language of t (which is, roughly speaking, the set of all words labeling a maximal path in the tree t) is included in some regular string language L [12], where L is accepted by a finite automaton $\mathcal{B}$ that is logspace constructible from $\mathcal{A}$. Now we guess a path in the input dag and simulate $\mathcal{B}$ on this path. The NL lower bound is obtained by a reduction from the graph accessibility problem for dags.

Theorem 3 *The uniform membership problem for deterministic TDTAs and dags is in NL. Moreover, there exists a fixed deterministic TDTA such that the membership problem for $\mathcal{A}$ and dags is NL-hard.*

Proof. For the upper bound we need the concept of the path language $P(t)$ of a tree $t \in T(\mathcal{F})$. It is a language over the alphabet $\Sigma = \mathcal{F}_0 \cup \{\alpha_i \mid \alpha \in \mathcal{F}, 1 \leq i \leq \text{arity}(\alpha) > 0\}$ and inductively defined as follows: If $\lambda_t(\varepsilon) \in \mathcal{F}_0$, then $P(t) = \{\lambda_t(\varepsilon)\}$, otherwise $P(f(t_1, \ldots, t_n)) = \{f_i w \mid 1 \leq i \leq n, w \in P(t_i)\}$. For an $\mathcal{F}$-labeled dag D let $P(D) = P(\text{eval}(D))$. For instance, for the dag D from Example 1 we have $P(D) = \{g_1 f_1 h_1 a, g_1 f_2 h_1 a, g_2 f_1 h_1 a, g_2 f_2 h_1 a, g_3 h_1 a\}$. Now, for a given deterministic TDTA $\mathcal{A}$ it is easy to construct in logspace a deterministic finite automaton $\mathcal{B}$ over the alphabet Σ such that $P(t) \subseteq L(\mathcal{B})$ if and only if $t \in T(\mathcal{A})$ [12]. Thus, $P(D) \backslash L(\mathcal{B}) \neq \emptyset$ if and only if $\text{eval}(D) \notin T(\mathcal{A})$. By guessing a path from the root of D to a leaf and thereby simulating the deterministic automaton $\mathcal{B}$, we can check in NL whether $P(D) \setminus L(\mathcal{B}) \neq \emptyset$. The NL upper bound follows from the complement closure of NL, see e.g. [24].

We prove the lower bound by a reduction from the graph accessibility problem for dags, which is NL-complete, see e.g. [24]. Let (V, E) be a directed acyclic graph where w.l.o.g. every node has outdegree 0 or 2, and let $u, v \in V$ be two nodes, where u has outdegree 2 and v has outdegree 0. We construct a dag D and a deterministic TDTA $\mathcal{A}$ such that $\text{eval}(D) \in T(\mathcal{A})$ if and only if there is no path from u to v in the graph (V, E); by the complement closure of NL [24] this suffices in order to prove the NL lower bound. By adding new nodes

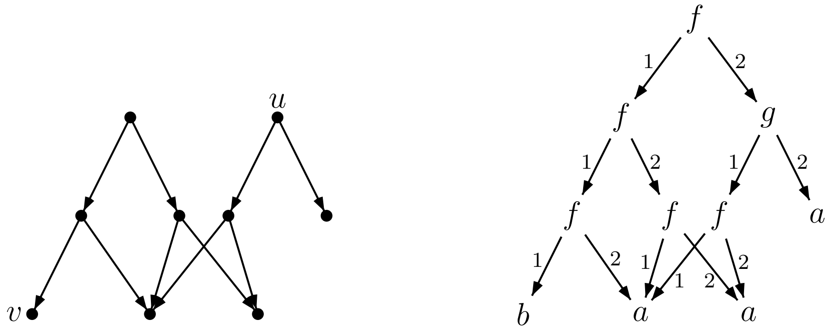


Fig. 1. A directed acyclic graph (V, E) and the dag D constructed from (V, E)

of outdegree 2 we may ensure that (V, E) has exactly one node of indegree 0. Next, we label the node u with the binary function symbol g and we label every other node of outdegree 2 with the binary function symbol f. Moreover, we label the node v with the constant b and we label every other node of outdegree 0 with the constant a. Call the resulting labeled dag D, see Figure 1 for an example. Then v is not reachable from u in the dag (V, E) if and only if every path in $\mathrm{eval}(D)$ from the root to a leaf that visits a g-labeled node also visits an a-labeled node. We can easily construct a fixed deterministic TDTA that checks the latter property. $\square$

By combining the statements in Theorem 1–3 we obtain the results for dags in Table 1.

SL cf tree grammars allow higher compression rates than dags. This makes computational problems harder when input trees are represented via SL cf tree grammars. The following result reflects this phenomenon. The PSPACE lower bound is shown by a reduction from QSAT (quantified boolean satisfiability), see e.g. [24].

Theorem 4 *The uniform membership problem for TAs and SL cf tree grammars is in PSPACE. Moreover, there exists a fixed deterministic BUTA such that the membership problem for $\mathcal{A}$ and SL cf tree grammars is PSPACE-hard.*

Proof. For the upper bound let $G = (\mathcal{F}, N, S, P)$ be an SL cf tree grammar and let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \mathcal{R})$ be a nondeterministic BUTA. Let $\mathcal{P}(\mathcal{A})$ be the deterministic BUTA that results by the usual powerset construction from $\mathcal{A}$. Thus, the state set of $\mathcal{A}$ is the powerset $\mathcal{P}(Q) = \{U \mid U \subseteq Q\}$ of Q. We will not construct $\mathcal{P}(\mathcal{A})$ explicitly, which is of course not possible in polynomial space. But we can evaluate $\mathcal{P}(\mathcal{A})$ on a given tree in polynomial space (even in polynomial time) by constructing the currently necessary production of $\mathcal{P}(\mathcal{A})$ on the fly. This was also done in the proof of Theorem 1.

In the following we will use the notation $\mathrm{ok}(A, U, U_1, \ldots, U_n)$ from the proof of Theorem 1 for the powerset automaton $\mathcal{P}(\mathcal{A})$. Thus, $A \in N$ is a nonterminal of G and $U, U_1, \ldots, U_n \subseteq Q$. The meaning of $\mathrm{ok}(A, U, U_1, \ldots, U_n)$ with

$n = \mathrm{arity}(A)$ is that the deterministic BUTA $\mathcal{P}(\mathcal{A})$ arrives at the root of $\mathrm{eval}_G(A)(x_1, \ldots, x_n)$ in state U if it starts in every occurrence of x_i in state U_i. Note that a tuple $(A, U, U_1, \ldots, U_n)$ can be stored in polynomial space. Moreover, note that since $\mathcal{P}(\mathcal{A})$ is deterministic, it is sufficient to verify in PSPACE whether an assertion $\mathrm{ok}(A, U, U_1, \ldots, U_n)$ is true, see the remarks at the end of the proof of Theorem 1. We will do this verification nondeterministically using a pushdown of polynomial size. This pushdown contains tuples of the form $(A, U, U_1, \ldots, U_n) \in N \times \mathcal{P}(Q)^{n+1}$, where $n = \mathrm{arity}(A)$. Initially the pushdown contains only one tuple of the form (S, U), where U is guessed such that $U \cap Q_f \neq \emptyset$. Now assume that the topmost tuple is $(A, U, U_1, \ldots, U_n)$ and let $A(x_1, \ldots, x_n) \to t(x_1, \ldots, x_n)$ be the unique production in P for the nonterminal A. First, we pop this tuple from the pushdown. Then we start running the deterministic BUTA $\mathcal{P}(\mathcal{A})$ on the tree $t(U_1, \ldots, U_n)$. If during this run, $\mathcal{A}$ has assigned to every child v_i $(1 \leq i \leq m)$ of a node v that is labeled with a nonterminal $B \in N$ a subset $V_i \subseteq Q$, then we guess a subset $V \subseteq Q$, assign V to the node v and push the tuple $(B, V, V_1, \ldots, V_m)$ onto the pushdown. Finally, if $\mathcal{P}(\mathcal{A})$ arrives at the root of t in state U, then we may pop the next tuple from the pushdown, otherwise we reject. We accept if the pushdown is empty. Note that the number of tuples on the pushdown is bounded by $k \cdot |N|$, where k is the maximal number of nonterminals in a right-hand side of a production. Since every tuple can be stored in polynomial space, the whole algorithm works in polynomial space.

We prove the lower bound by a reduction from QSAT (quantified satisfiability) [24]. This reduction was also used in [15] in the context of hierarchically defined graphs. Let

$$\psi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \, \varphi(x_1, \ldots, x_n)$$

be a quantified boolean formula, where $Q_i \in \{\forall, \exists\}$ and $\varphi(x_1, \ldots, x_n)$ is a boolean formula with variables from $\{x_1, \ldots, x_n\}$. Let the ranked alphabet $\mathcal{F}$ contain the binary function symbols $\wedge$ and $\vee$ and the constants 0 and 1. Then we may view $\varphi(x_1, \ldots, x_n)$ as a tree over the ranked alphabet $\mathcal{F}$ with variables $x_1, \ldots, x_n$. Let the SL cf tree grammar G be defined by the following productions:

$$\begin{aligned}
A_n(x_1, \ldots, x_n) &\to \varphi(x_1, \ldots, x_n) \\
A_{i-1}(x_1, \ldots, x_{i-1}) &\to \begin{cases} \wedge(A_i(x_1, \ldots, x_{i-1}, 0), A_i(x_1, \ldots, x_{i-1}, 1)) & \text{if } Q_i = \forall \\ \vee(A_i(x_1, \ldots, x_{i-1}, 0), A_i(x_1, \ldots, x_{i-1}, 1)) & \text{if } Q_i = \exists \end{cases} \\
&\quad \text{for } 0 < i \leq n \\
S &\to A_0
\end{aligned}$$

Finally let $\mathcal{A}$ be the deterministic BUTA from the proof of Theorem 2 for evaluating boolean formulas. Then $\mathrm{eval}(G) \in T(\mathcal{A})$ if and only if the formula ψ is true. $\square$

Only for deterministic TDTAs we obtain more efficient algorithms in the context of general SL cf tree grammars. The polynomial time upper bound in the next theorem is again based on the concept of the path language of a tree. For an SL cf tree grammar G, we show that the path language of $\mathrm{eval}(G)$ can be generated by a small context-free string grammar. The lower bound follows from a result of [20] about string straight-line programs.

Theorem 5 *The uniform membership problem for deterministic TDTAs and SL cf tree grammars is in P. Moreover, there is a fixed deterministic TDTA such that the membership problem for $\mathcal{A}$ and linear SL cf tree grammars with only one parameter is P-hard.*

Proof. For the upper bound recall the definition of the path language $P(t) \subseteq \Sigma^*$ of a tree $t \in T(\mathcal{F})$, where $\Sigma = \mathcal{F}_0 \cup \{\alpha_i \mid \alpha \in \mathcal{F}, 1 \leq i \leq \mathrm{arity}(\alpha) > 0\}$ from the proof of Theorem 3. For the input SL cf tree grammar $G = (\mathcal{F}, N, S, P)$ let $P(G) = P(\mathrm{eval}(G))$. Using a construction that is inspired by [9], we will build in polynomial time (w.r.t. $|G|$) a context-free grammar H such that $L(H) = P(G)$. The set of nonterminals M of H is $M = \{A \in N \mid \mathrm{arity}(N) = 0\} \cup \{A_i \mid A \in N, 1 \leq i \leq \mathrm{arity}(N) > 0\}$. Note that the path language $P(t)$ of every tree t over the ranked alphabet $N \cup \mathcal{F}$ is a language over the alphabet $M \cup \Sigma$. The start nonterminal of H is $S \in N \cap M$. The productions of H are defined as follows: First let $A \in N$ be a nonterminal of rank 0 and let $A \to t$ be the corresponding production in P. Then, in H we introduce the productions $A \to w_1 \,|\, w_2 \,|\, \cdots \,|\, w_m$, where $\{w_1, \ldots, w_m\} = P(t)$. Now let us take a production $A(x_1, \ldots, x_n) \to t(x_1, \ldots, x_n)$ from P where $n > 0$. Then, for every $1 \leq i \leq n$ we put into H the productions $A_i \to w_1 \,|\, w_2 \,|\, \cdots w_m$, where $\{w_1, \ldots, w_m\}$ is the set of those strings w such that wx_i belongs to the path language $P(t(x_1, \ldots, x_n))$ (here, the variable x_i is viewed as a constant). Note that the number m is bounded by the number of leafs of $t(x_1, \ldots, x_n)$ and hence by the size of G. It is easy to check that indeed $L(H) = P(G)$. Let us consider an example. Assume that G consists of the following productions:

$$\begin{aligned} S &\to A(a, b) \\ A(x_1, x_2) &\to B(B(x_1, x_2), B(x_1, x_2)) \\ B(x_1, x_2) &\to f(x_1, x_2) \end{aligned}$$

Then H contains the following productions:

$$\begin{aligned} S &\to A_1 a \mid A_2 b \\ A_1 &\to B_1 B_1 \mid B_2 B_1 \\ A_2 &\to B_1 B_2 \mid B_2 B_2 \\ B_i &\to f_i \text{ for } i \in \{1, 2\} \end{aligned}$$

Next, since $\mathcal{A}$ is a deterministic TDTA, we can construct in polynomial time a deterministic finite automaton $\mathcal{B}$ over the alphabet Σ such that $P(t) \subseteq L(\mathcal{B})$ if and only if $t \in T(\mathcal{A})$, see also the proof of Theorem 1. Thus, $L(H) \setminus L(\mathcal{B}) = \emptyset$ if and only if $\text{eval}(G) \in T(\mathcal{A})$. From H and $\mathcal{B}$ we can construct in polynomial time a context-free grammar for $L(H) \setminus L(\mathcal{B})$. Emptiness for this grammar can be checked in polynomial time.

For the lower bound we can use a result of [20] for *straight-line programs* (SLPs). An SLP is an ordinary context-free grammar $G = (N, \Gamma, S, P)$ (N is the set of nonterminals, Γ is the set of terminals, S is the start nonterminal, and $P \subseteq N \times (N \cup \Gamma)^*$ is the finite set of productions) such that: (i) for every $A \in N$ there exists exactly one production $(A, w_A) \in P$ with left-hand side A and (ii) the relation $\{(A, B) \in N \times N \mid B \text{ occurs in } w_A\}$ is acyclic. Clearly, every SLP G generates exactly one string $\text{eval}(G) \in \Gamma^*$. By [20], there exists a fixed regular language L such that it is P-complete to determine whether $\text{eval}(G) \in L$ for a given SLP G. The theorem follows immediately by using the standard encoding of strings over Γ by trees over the ranked alphabet $\Gamma \cup \{\#\}$, where $\text{arity}(a) = 1$ for all $a \in \Gamma$ and $\text{arity}(\#) = 0$: The string $a_1 a_2 \cdots a_n$ is encoded by the tree $a_1(a_2(\cdots a_n(\#)\cdots))$. Under this encoding, a regular string language can be recognized both by a deterministic bottom-up as well as a deterministic top-down tree automaton. Moreover, an SLP corresponds to a linear SL cf tree grammar with only one parameter. $\square$

From Theorem 1 and 5 (resp. Theorem 4 and 5) we obtain the complexity results for linear SL cf tree grammars with a fixed number of parameters (resp. general SL cf tree grammars) in Table 1, see lin. SL + fixed number para. (resp. general SL). The following result completes our characterization presented in Table 1.

Theorem 6 *The uniform membership problem for TAs and (non-linear) SL cf tree grammars with only one parameter is PSPACE-hard.*

Proof. We prove the theorem by a reduction from QSAT [24]. Let us take a quantified boolean formula $\psi = Q_1 x_1 \cdots Q_n x_n \, \varphi$, where $Q_i \in \{\forall, \exists\}$ and φ is a boolean formula with variables from $\mathcal{X} = \{x_1, \ldots, x_n\}$. W.l.o.g. we may assume that in φ the negation operator $\neg$ only occurs directly in front of variables. Let $\bar{\mathcal{X}} = \{\neg x \mid x \in \mathcal{X}\}$. We define an SL cf tree grammar G as follows: The set of terminals contains the binary function symbol f, a unary function symbol t_i for every $x_i \in \mathcal{X}$, and a constant a. The set of nonterminals contains the start nonterminal S, and for every subformula α of ψ it contains a nonterminal A_α of arity 1. The productions of G are:

$$
\begin{array}{rll}
S & \to A_\psi(a) & \\
A_\alpha(y) & \to y & \text{if } \alpha \in \mathcal{X} \cup \bar{\mathcal{X}} \\
A_\alpha(y) & \to f(A_\beta(t_i(y)), A_\beta(y)) & \text{if } \alpha \in \{\forall x_i \beta, \exists x_i \beta\} \\
A_\alpha(y) & \to f(A_\beta(y), A_\gamma(y)) & \text{if } \alpha \in \{\beta \wedge \gamma, \beta \vee \gamma\}
\end{array}
$$

An occurrence of the symbol t_i on a path in the tree $\text{eval}(G)$ indicates that the variable x_i is set to true. Note that from a nonterminal A_α, where α begins with a quantification $\exists x_i$ or $\forall x_i$ we first generate a branching node (labeled with the binary symbol f). Moreover, the left branch gets in addition the unary symbol t_i, which indicates that x_i is set to true. The absence of t_i in the right branch indicates that x_i is set to false.

We define a nondeterministic TDTA $\mathcal{A}$ as follows: The state set of $\mathcal{A}$ contains all subformulas of ψ plus an additional state q. The initial state of $\mathcal{A}$ is the whole formula ψ. The set $\mathcal{R}$ of transition rules of $\mathcal{A}$ consists of the following rules:

$$
\begin{array}{rll}
q(f(y,z)) \to f(q(y),q(z)) & & \\
q(t_i(y)) \to t_i(q(y)) & & \text{for all } i \\
q(a) \to a & & \\
\alpha(f(y,z)) \to f(\beta(y),q(z)) & \text{if } \alpha = \exists x_i\beta \text{ for some } i & \\
\alpha(f(y,z)) \to f(q(y),\beta(z)) & \text{if } \alpha = \exists x_i\beta \text{ for some } i & \\
\alpha(f(y,z)) \to f(\beta(y),\beta(z)) & \text{if } \alpha = \forall x_i\beta \text{ for some } i & \\
\alpha(f(y,z)) \to f(\beta(y),q(z)) & \text{if } \alpha = \beta \vee \gamma \text{ for some } \gamma & \\
\alpha(f(y,z)) \to f(q(y),\gamma(z)) & \text{if } \alpha = \beta \vee \gamma \text{ for some } \beta & \\
\alpha(f(y,z)) \to f(\beta(y),\gamma(z)) & \text{if } \alpha = \beta \wedge \gamma & \\
\alpha(t_i(y)) \to t_i(\alpha(y)) & \text{if } \alpha \in (\mathcal{X} \cup \bar{\mathcal{X}}) \setminus \{x_i, \neg x_i\} & \\
\alpha(t_i(y)) \to t_i(q(y)) & \text{if } \alpha = x_i & \\
\alpha(a) \to a & \text{if } \alpha \in \bar{\mathcal{X}} &
\end{array}
$$

Figure 2 shows the tree $\text{eval}(G)$ for the true quantified boolean formula

$$\forall x_1 \exists x_2 : (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2),$$

where in addition every node is labeled with a state of the automaton $\mathcal{A}$ such that the overall labeling is an accepting run.

By the first three rules for state q, $q(t) \xrightarrow{*}_{\mathcal{R}} t$ for every ground tree t. Thus, if we reach the state q, then the corresponding subtree is accepted. If the current state α is an existential subformula $\exists x_i\beta$, then we guess nondeterministically one of the two subtrees of the current f-labeled node (i.e., we choose an assignment for x_i) and verify β in that subtree. The other subtree is accepted by sending q to that subtree. Similarly, if the current state α is a universal subformula $\forall x_i\beta$, then we verify β in both subtrees, i.e., for both assignments for x_i. The rules for $\alpha = \beta \vee \gamma$ and $\alpha = \beta \wedge \gamma$ can be interpreted similarly. Note that by construction of G and $\mathcal{A}$, if the current state α is of the form $\exists x_i\beta$, $\forall x_i\beta$, $\beta \vee \gamma$, or $\beta \wedge \gamma$, then the current tree node in $\text{eval}(G)$ is an f-labeled node. On the other hand, if the current state is from $\mathcal{X} \cup \bar{\mathcal{X}}$, then the current tree node in $\text{eval}(G)$ is labeled with a symbol t_j or the constant a. If

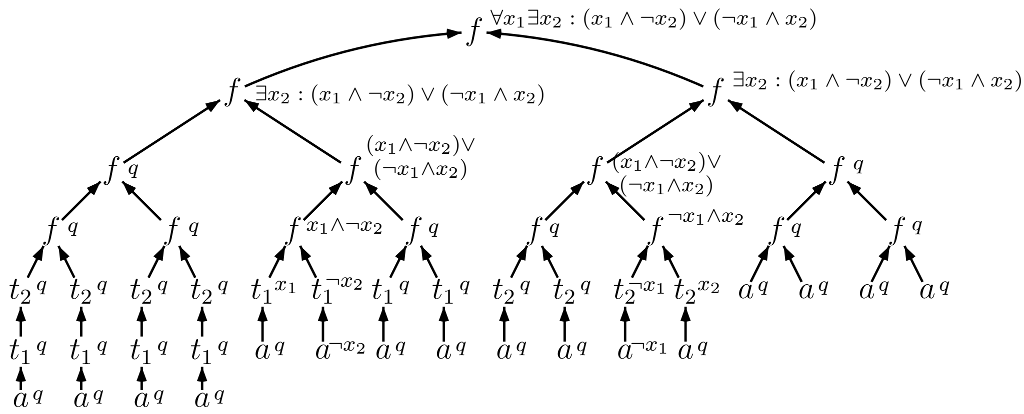


Fig. 2.

the current state is a variable x_i, then we search for the symbol t_i in the chain of t_j-labeled nodes below the current node. We accept by going into the state q ($x_i(t_i(y)) \to t_i(q(y))$) as soon as we find t_i. If we do not find t_i and end up in the constant a, then we block; note that there is no rule of form $x_i(a) \to a$. On the other hand, if the current state is a negated variable $\neg x_i$, then we verify that there is no t_i in the chain of t_j-labeled nodes below the current node. Thus, we block as soon as we find t_i; note that there is no rule with left-hand side $\neg x_i(t_i(y))$. On the other hand, if we finally reach the constant a in state $\neg x_i$, then we accept via the rule $\neg x_i(a) \to a$. From the previous discussion, it is not hard to see that the formula ψ is true if and only if $\mathrm{eval}(G) \in L(\mathcal{A})$. $\square$

From Theorem 1 and Theorems 4–6 we obtain the results for SL cf tree grammars with a fixed number of parameters in Table 1.

We end this section with two results concerning the parameterized complexity of membership problems for tree automata. *Parameterized complexity* [10] is a branch of complexity theory with the goal to understand which input parts of a hard (e.g. NP-hard) problem are responsible for the combinatorial explosion. A *parameterized problem* is a decision problem where the input is a pair $(k, x) \in \mathbb{N} \times \Sigma^*$. The first input component k is called the *input parameter* (it may also consist of several natural numbers). A typical example of a parameterized problem is the parameterized version of the clique problem, where the input is a pair (k, G), G is an undirected graph, and it is asked whether G has a clique of size k. A parameterized problem (with input (k, x)) is in the class FPT (fixed parameter tractable), if the problem can be solved in time $f(k) \cdot |x|^c$. Here c is a fixed constant and f is an arbitrary (e.g., exponential) computable function on $\mathbb{N}$. This means that the non-polynomial part of the algorithm is restricted to the parameter k.

Theorem 7 *The following parameterized problem is in FPT:*

INPUT: An SL cf tree grammar G with k parameters and a TA $\mathcal{A}$ with n

states.
INPUT PARAMETER: (k, n)
QUESTION: $\mathrm{eval}(G) \in T(\mathcal{A})$*?*

Proof. We first transform $\mathcal{A}$ into a deterministic BUTA with at most 2^n states. Then we apply Theorem 1 which gives us a running time of $2^{kn} \cdot |G| \cdot \mathcal{A}$. □

In recent years, a structural theory of parameterized complexity with the aim of showing that certain problems are unlikely to belong to FPT was developed. Underlying this theory is the notion of parameterized reductions [10]: A parameterized reduction from a parameterized problem A (with input $(k, x) \in \mathbb{N} \times \Sigma^*$) to a parameterized problem B (with input $(\ell, y) \in \mathbb{N} \times \Gamma^*$) is a mapping $f : \mathbb{N} \times \Sigma^* \to \mathbb{N} \times \Gamma^*$ such that: (i) for all $(k, x) \in \mathbb{N} \times \Sigma^*$, $(k, x) \in A$ if and only if $f(k, x) \in B$, (ii) $f(k, x)$ is computable in time $g(k) \cdot |x|^c$ for some computable function g and some constant c, and (iii) for some computable function h, if $f(k, x) = (\ell, y)$, then $\ell \leq h(k)$. A parameterized problem A is fpt-reducible to a parameterized problem B if there exists a parameterized reduction from A to B. One of the classes in the upper part of the parameterized complexity spectrum is the class AW[P]. For the purpose of this paper it is not necessary to present the quite technical definition of AW[P]. Roughly speaking, AW[P] results from taking the closure (w.r.t. fpt-reducibility) of a parameterized version of the PSPACE-complete QSAT problem. Problems that are AW[P]-hard are very unlikely to be in FPT.

Theorem 8 *The following problem is AW[P]-hard (w.r.t. fpt-reducibility):*

INPUT: A deterministic BUTA $\mathcal{A}$ and an SL cf tree grammar G with k parameters
INPUT PARAMETER: k
QUESTION: $\mathrm{eval}(G) \in T(\mathcal{A})$*?*

Proof. We will use the fact that the following problem, called pFOMC (parameterized first-order model-checking), is hard (w.r.t. fpt-reducibility) for the class AW[P], see [25]:

INPUT: A directed graph $H = (V, E)$ and a sentence ϕ of first-order logic (built up from the atomic formulas $x = y$ and $E(x, y)$ (for variables x and y) using boolean connectives and quantification over nodes of H).
INPUT PARAMETER: The number of different variables that are used in ϕ
QUESTION: Is ϕ true in the graph H?

It should be noted that it is open whether pFOMC also belongs to AW[P].

Thus, it suffices to reduce pFOMC to the problem in the theorem. Let $H = (V, E)$ be a directed graph and φ be a first-order sentence. W.l.o.g. assume that $V = \{1, \dots, n\}$. We define an SL cf tree grammar G as follows: The set of terminals contains the constants $1, \dots, n$, a unary function symbol $\neg$, and binary function symbols E, $=$, and $\wedge$. For every subformula $\psi(\bar{x})$ of φ we introduce a nonterminal A_ψ. Here, $\bar{x}$ is the sequence of free variables of ψ, and the rank of A_ψ is the number of these free variables, i.e., $\mathrm{arity}(A_\psi) = |\bar{x}|$. The start nonterminal is A_φ, which has rank 0, since φ has no free variables. The productions of G are:

$$A_\psi(\bar{x}) \to \begin{cases} E(x,y) & \text{if } \psi(\bar{x}) = E(x,y) \\ =(x,y) & \text{if } \psi(\bar{x}) = (x = y) \\ \neg(A_\theta(\bar{x})) & \text{if } \psi(\bar{x}) = \neg\theta(\bar{x}) \\ \wedge(A_{\psi_1}(\bar{x}_1), A_{\psi_2}(\bar{x}_2)) & \text{if } \psi(\bar{x}) = \psi_1(\bar{x}_1) \wedge \psi_2(\bar{x}_2) \\ \wedge(A_\theta(\bar{x},1), A_\theta(\bar{x},2), \dots, A_\theta(\bar{x},n)) & \text{if } \psi(\bar{x}) = \forall y\, \theta(\bar{x},y) \end{cases}$$

Of course, in the last line, the n-ary $\wedge$ has to be replaced by a binary tree of binary $\wedge$-operators. Clearly, the number of parameters of G is bounded by the maximal number of free variables in a subformula of ϕ. The latter number is bounded by the total number of different variables used in ϕ. Now we can easily construct from H a deterministic BUTA $\mathcal{A}$ with state set $V \cup \{\mathrm{true}, \mathrm{false}\}$ such that $\mathrm{eval}(G) \in T(\mathcal{A})$ if and only if ϕ is true in the graph H. The BUTA $\mathcal{A}$ evaluates trees of the form $=(a,b)$ and $E(a,b)$, where $a, b \in V$, to boolean values and then evaluates the resulting boolean expression. This is a parameterized reduction from pFOMC to the problem in the theorem. $\square$

4 XPath Evaluation

In this section, we consider XML-trees that are compressed via SL cf tree grammars and study the node selecting language XPath over such trees. For more background on XPath see [13,14]. We restrict our attention to linear SL cf tree grammars. Skeletons of XML documents are usually modeled as *rooted unranked labeled trees*. Analogously to Section 2, an unranked tree with labels from an (unranked) alphabet Σ can be defined as a pair $t = (\mathrm{dom}_t, \lambda_t)$, where (i) $\mathrm{dom}_t \subseteq \mathbb{N}^*$ is finite, (ii) $\lambda_t : \mathrm{dom}_t \to \mathcal{F}$, (iii) if $v \preceq w \in \mathrm{dom}_t$, then also $v \in \mathrm{dom}_t$, and (iv) if $vi \in \mathrm{dom}_t$ then also $vj \in \mathrm{dom}_t$ for every $1 \le j \le i$. For the purpose of this section, it is more suitable to view such an unranked tree $t = (\mathrm{dom}_t, \lambda_t)$ as a relational structure $t = (\mathrm{dom}_t, \text{child}, \text{next-sibling}, (Q_a)_{a\in\Sigma})$, where $Q_a = \lambda_t^{-1}(a)$, $\text{child} = \{(v, vi) \in \mathrm{dom}_t \times \mathrm{dom}_t \mid v \in \mathbb{N}^*, i \in \mathbb{N}\}$, and $\text{next-sibling} = \{(vi, v(i+1)) \in \mathrm{dom}_t \times \mathrm{dom}_t \mid v \in \mathbb{N}^*, i \in \mathbb{N}\}$. Thus, $\text{child}(u,v)$ is the child-relation in t and $\text{next-sibling}(u,v)$ if and only if v is the right sibling of u. From the basic tree relations child and next-sibling further tree relations that are called *XPath-axes* can be defined. For instance let descendant := child* (the reflexive and transitive closure of child) and following-sibling := next-sibling*. For the definition of the other XPath axes see for instance [13]. In the following we consider the four XPath axes child, descendant, next-sibling, and following-sibling; handling of other axes is straightforward and needs no further ideas.

The node selection language *core XPath* [13] can be seen as the tree navigational (or, "structural") core of XPath. Its syntax is given by the following EBNF; here, χ is an XPath-axis and $a \in \Sigma \cup \{*\}$ (where $*$ is a new symbol):

$$\begin{aligned}
\text{corexpath} &::= \text{locationpath} \mid /\,\text{locationpath} \\
\text{locationpath} &::= \text{locationstep}\,(/\,\text{locationstep})^* \\
\text{locationstep} &::= \chi :: a \mid \chi :: a\,[\text{pred}] \\
\text{pred} &::= (\text{pred and pred}) \mid (\text{pred or pred}) \mid \text{not(pred)} \mid \text{locationpath}
\end{aligned}$$

Let Q_* be the unary predicate that is true for every node of a tree t. We define the semantics of core XPath by translating a given tree

$$t = (\mathrm{dom}_t, \text{child}, \text{next-sibling}, (Q_a)_{a\in\Sigma})$$

and a given expression $\pi \in \mathcal{L}(\text{corexpath})$ (resp. $e \in \mathcal{L}(\text{pred})$) into a binary relation $\mathcal{S}[\pi, t] \subseteq \mathrm{dom}_t \times \mathrm{dom}_t$ (resp. a unary relation $\mathcal{E}[e, t] \subseteq \mathrm{dom}_t$). Let $\pi, \pi_1, \pi_2 \in \mathcal{L}(\text{locationpath})$, $e, e_1, e_2 \in \mathcal{L}(\text{pred})$, and let χ be an XPath axes (recall that ε is the root of a tree).

$$\begin{aligned}
\mathcal{S}[\chi :: a[e], t] &= \{(x,y) \in \mathrm{dom}_t \times \mathrm{dom}_t \mid (x,y) \in \chi, y \in Q_a, y \in \mathcal{E}[e,t]\} \\
\mathcal{S}[/\pi, t] &= \mathrm{dom}_t \times \{x \in \mathrm{dom}_t \mid (\varepsilon, x) \in \mathcal{S}[\pi, t]\} \\
\mathcal{S}[\pi_1/\pi_2, t] &= \{(x,y) \in \mathrm{dom}_t \times \mathrm{dom}_t \mid \exists z : (x,z) \in \mathcal{S}[\pi_1, t], (z,y) \in \mathcal{S}[\pi_2, t]\} \\
\mathcal{E}[e_1 \text{ and } e_2, t] &= \mathcal{E}[e_1, t] \cap \mathcal{E}[e_2, t] \\
\mathcal{E}[e_1 \text{ or } e_2, t] &= \mathcal{E}[e_1, t] \cup \mathcal{E}[e_2, t] \\
\mathcal{E}[\text{not}(e), t] &= \mathrm{dom}_t \setminus \mathcal{E}[e, t] \\
\mathcal{E}[\pi, t] &= \{x \in \mathrm{dom}_t \mid \exists y : (x,y) \in \mathcal{S}[\pi, t]\}
\end{aligned}$$

Recall that by definition SL cf tree grammars generate ranked trees. In order to generate XML skeletons, i.e., unranked trees with SL cf tree grammars, we encode unranked trees by binary trees (and hence ranked trees) using a standard encoding: For an unranked tree $t = (\mathrm{dom}_t, \text{child}, \text{next-sibling}, (Q_a)_{a\in\Sigma})$ define the binary encoding $\mathrm{bin}(t) = (\mathrm{dom}_t, \text{child1}, \text{child2}, (Q_a)_{a\in\Sigma})$, where (i) $(u,v) \in$ child1 if and only if $(u,v) \in$ child and there does not exist $w \in \mathrm{dom}_t$ with $(w,v) \in$ next-sibling (i.e., v is the left-most child of u), and (ii) child2 = next-sibling. Note that t and $\mathrm{bin}(t)$ have the same set of nodes. The following theorem is our main result in this section.

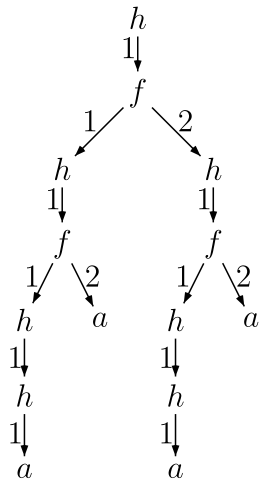
Theorem 9 *The following problem is PSPACE-complete:*

INPUT: A linear SL cf tree grammar G generating a binary tree such that $\text{eval}(G) = \text{bin}(t)$ *for some (unique) unranked tree t, two nodes u, v of* $\text{eval}(G)$*, and a core XPath expression $\pi \in$* $\mathcal{L}(\text{corexpath})$.
QUESTION: $(u, v) \in \mathcal{S}[\pi, t]$ *?*

Proof. PSPACE-hardness follows from the corresponding result for dags from [11]. The crucial point for the PSPACE-membership proof is the fact that for a linear SL cf tree grammar $G = (\mathcal{F}, N, S, P)$ we can store a node of $\text{eval}(G)$ in polynomial space. For this we list a sequence of at most $|N|$ pairs consisting of a nonterminal $A \in N$ and a node in the unique right-hand side for A. Let us illustrate the idea with an example: Let $G = (\{h, f, a\}, \{S, A, B, C\}, S, P)$, where P contains the following linear productions:

$$\begin{aligned} S &\to A(B(a), B(a)) \\ A(x_1, x_2) &\to C(C(x_1, a), C(x_2, a)) \\ C(x_1, x_2) &\to h(f(x_1, x_2)) \\ B(x) &\to h(h(x)) \end{aligned}$$

Then $\text{eval}(G)$ is the following tree:



Let us consider the f-labeled node 121 in this tree. It is generated from the unique A in the right hand side $A(B(a), B(a))$ of S. The unique A in $A(B(a), B(a))$ corresponds to the tree node ε (the root) of $A(B(a), B(a))$. Thus, we start our sequence with the pair (S, ε). Next, in the right-hand side $C(C(x_1, a), C(x_2, a))$ of A we have to move to the last C in order to arrive at our goal node 121. The last C is the right child of the root in the tree $C(C(x_1, a), C(x_2, a))$ and thus identified by the tree node 2. Hence, we proceed with the pair $(A, 2)$. Finally, in the right-hand side $h(f(x_1, x_2))$ of C we just have to specify the unique node labeled with f, which is the node 1. Thus, the node 121 is represented by the sequence $(S, \varepsilon)(A, 2)(C, 1)$. Note that this simple idea does not work for non-linear SL cf tree grammars due to the sharing of variables. In fact, it cannot work, because there may be doubly exponentially many nodes in eval(G) if G is nonlinear. Hence, any specification of a node needs at least exponentially many bits.

In the following, we assume that nodes of eval(G) (for G linear) are always represented in the polynomial size description outlined above. The following fact is easy to show:

Fact 1 *For given nodes u and v of* eval(G) *it can be checked in polynomial time whether v is the k-th child of u for some given k or whether u is labeled with some given symbol a.*

Let us now return to the XPath evaluation problem. Let G be a linear SL cf tree grammar such that eval(G) is a binary Σ-labeled tree with eval(G) $=$ bin(t) for some unranked tree t. Let us also take a core XPath expression $\pi \in \mathcal{L}(\text{corexpath})$. For the PSPACE upper bound we will use the fact that PSPACE is precisely the class of all problems that can be solved on an alternating Turing machine in polynomial time, see e.g. [24] for more details. Roughly speaking, an *alternating Turing-machine M* is a nondeterministic Turing-machine, where the set of states Q is partitioned into three sets: $Q_\exists$ (existential states), $Q_\forall$ (universal states), and F (accepting states). A configuration C with current state q is accepting, if

- $q \in F$, or
- $q \in Q_\exists$ and there exists a successor configuration of C that is accepting, or
- $q \in Q_\forall$ and every successor configuration of C is accepting.

An input word w is accepted by M if the corresponding initial configuration is accepting.

Will show that the question, whether $(u, v) \in \mathcal{S}[\pi, t]$ can be answered by an alternating Turing-machine in polynomial time. For this, it is useful to transform π into an equivalent first-order formula (with two free variables) over the signature that contains all unary predicates Q_a and all XPath-axes, i.e., the relations child, descendant, next-sibling, and following-sibling. This translation is done inductively: Every $\zeta \in \mathcal{L}(\text{corexpath})$ is translated into a first-order formula $\varphi_2(\zeta)(x, y)$ over the signature containing child, descendant, next-sibling, following-sibling, $(Q_a)_{a\in\Sigma}$. An expression $e \in \mathcal{L}(\text{pred})$ is translated into a first-order formula $\varphi_1(e)(x)$ with only one free variable. Let

$\zeta, \zeta_1, \zeta_2 \in \mathcal{L}(\text{locationpath})$ and $e, e_1, e_2 \in \mathcal{L}(\text{pred})$.

$$\begin{aligned}
\varphi_2(\chi :: a[e])(x,y) &= \chi(x,y) \wedge Q_a(y) \wedge \varphi_1(e)(y) \\
\varphi_2(/\zeta)(x,y) &= \exists z : \varphi_2(\zeta)(z,y) \wedge \neg\exists z' : \text{child}(z',z) \\
\varphi_2(\zeta_1/\zeta_2)(x,y) &= \exists z : \varphi_2(\zeta_1)(x,z) \wedge \varphi_2(\zeta_2)(z,y) \\
\varphi_1(e_1 \text{ and } e_2)(x) &= \varphi_1(e_1)(x) \wedge \varphi_1(e_2)(x) \\
\varphi_1(e_1 \text{ or } e_2)(x) &= \varphi_1(e_1)(x) \vee \varphi_1(e_2)(x) \\
\varphi_1(\text{not}(e))(x) &= \neg\varphi_1(e)(x) \\
\varphi_1(\zeta)(x) &= \exists y : \varphi_2(\zeta)(x,y)
\end{aligned}$$

These rules reformulate the semantic definition of XPath before Theorem 9 in the context of first-order logic. In the second line, the subformula $\neg\exists z'$: $\text{child}(z',z)$ expresses that z is the root of the tree. Let $\varphi(x,y) = \varphi_2(\pi)(x,y)$, where π is our input XPath expression.

It remains to check, whether $\varphi(u,v)$ is true in the unranked tree t. Next let us move to the binary tree $\text{eval}(G) = \text{bin}(t)$. Recall that $\text{eval}(G)$ contains two edge relations: child1 (left child) and child2 (right child). From child1 and child2 we define the binary relations $\text{descendant2} = \text{child2}^*$ and $\text{descendant} = (\text{child1} \cup \text{child2})^*$. From $\varphi(x,y)$ it is now straightforward to construct a formula $\varphi_{\text{bin}}(x,y)$ over the signature containing child1, child2, descendant2, descendant, $(Q_a)_{a\in\Sigma}$ such that $\varphi(u,v)$ is true in the unranked tree t if and only if $\varphi_{\text{bin}}(u,v)$ is true in $\text{bin}(t) = \text{eval}(G)$. In order to construct $\varphi_{\text{bin}}(x,y)$ we only replace the atomic subformulas of $\varphi(x,y)$ that involve XPath axes according to the following rules:

$$\begin{aligned}
\text{child}_{\text{bin}}(x,y) &:= \exists z : \text{child1}(x,z) \wedge \text{descendant2}(z,y) \\
\text{descendant}_{\text{bin}}(x,y) &:= \text{descendant}(x,y) \\
\text{next-sibling}_{\text{bin}}(x,y) &:= \text{child2}(x,y) \\
\text{following-sibling}_{\text{bin}}(x,y) &:= \text{descendant2}(x,y)
\end{aligned}$$

Thus, it suffices to check in PSPACE whether $\varphi_{\text{bin}}(u,v)$ is true in $\text{eval}(G)$. For this let us take an arbitrary first-order formula $\psi(x_1,\ldots,x_n)$ over the signature containing child1, child2, descendant2, descendant, $(Q_a)_{a\in\Sigma}$ and let $u_1,\ldots,u_n$ be nodes of $\text{eval}(G)$ that are represented as outlined above. We now describe an alternating Turing machine M that checks in polynomial time the truth of $\psi(u_1,\ldots,u_n)$ in the binary tree $\text{eval}(G)$. W.l.o.g. we may assume that the negation symbol in ψ only occurs in front of atomic formulas. Depending on the outermost operator of ψ the machine M behaves as follows:

(1) If $\psi(x_1,\ldots,x_n) = \exists y : \theta(y,x_1,\ldots,x_n)$, then M guesses in an existential state of M a node $v \in \text{dom}_t$ and proceeds with the formula $\theta(v,u_1,\ldots,u_n)$.
(2) If $\psi(x_1,\ldots,x_n) = \forall y : \theta(y,x_1,\ldots,x_n)$, then M proceeds analogously, except that the guessing of the node u is done in a universal state of M.

(3) If $\psi(x_1,\ldots,x_n) = \psi_1(x_1,\ldots,x_n) \vee \psi_2(x_1,\ldots,x_n)$, then M guesses in an existential state an $i \in \{1,2\}$ and proceeds with the formula $\psi_i(u_1,\ldots,u_n)$.
(4) If $\psi(x_1,\ldots,x_n) = \psi_1(x_1,\ldots,x_n) \wedge \psi_2(x_1,\ldots,x_n)$, then M proceeds analogously, except that the guessing of $i \in \{1,2\}$ is done in a universal state.

Since nodes of eval(G) can be stored in polynomial space with respect to the size of the grammar G, the guessing of a node u in (1) and (2) can be done in polynomial time. It remains to verify possibly negated atomic formulas. For a statement $(\neg)Q_a(u)$, $(\neg)\text{child1}(u,v)$, or $(\neg)\text{child2}(u,v)$ we can directly check the truth of the statement in polynomial time by Fact 1. For a statement descendant2(u,v) we just have to check whether there is a path in the child2-relation from u to v. This can be done in polynomial space (and hence in alternating polynomial time) by guessing the path incrementally and thereby storing only the last two nodes, for which we can check in polynomial time by Fact 1 whether they are related by the child2-relation. For $\neg$descendant2(u,v) we can use the closure of PSPACE under complement. For the relation descendant we can argue analogously. $\square$

5 Open Problems and Conclusions

An interesting class of SL cf tree grammars that is missing in our present complexity analysis is the class of *linear* SL cf tree grammar (with an unbounded number of parameters). Our algorithm BPLEX from [5] outputs linear SL cf tree grammars. Note that BPLEX, even when bounding the number of parameters by a small constant (like 2 or 3), clearly outperforms compression by dags; the results presented here show that with respect to tree automata (and XPath evaluation) exactly the same complexity bounds hold as for dags [4,11]. This motivates us to believe that linear SL cf tree grammars are better suited than dags as memory efficient representations of XML documents. Precise trade-offs between the representations have to be determined in practice; we are currently implementing our ideas as part of BPLEX. For the XPath evaluation problem, the complexity for non-linear SL cf tree grammars remains open. We conjecture that the PSPACE upper bound from Theorem 9 cannot be generalized to the non-linear case.

References

[1] S. Anantharaman, P. Narendran, and M. Rusinowitch. Closure properties and decision problems of dag automata. *Information Processing Letters*, 94(5):231–240, 2005.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[4] P. Buneman, M. Grohe, and Ch. Koch. Path queries on compressed XML. In J. C. Freytag et al., editors, *Proceedings of the 29th Conference on Very Large Data Bases (VLDB 2003)*, pages 141–152. Morgan Kaufmann, 2003.

[5] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In G. M. Bierman and Ch. Koch, editors, *Proceedings of the Tenth International Symposium on Database Programming Languages (DBPL 2005), Trondheim (Norway)*, number 3774 in Lecture Notes in Computer Science, pages 199–216. Springer, 2005.

[6] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[7] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2002.

[8] B. Courcelle. A representation of trees by languages I. *Theoretical Computer Science*, 6:255–279, 1978.

[9] B. Courcelle. A representation of trees by languages II. *Theoretical Computer Science*, 7:25–55, 1978.

[10] R. G. Downey and M. R. Fellows. *Parametrized Complexity*. Springer, 1999.

[11] M. Frick, M. Grohe, and Ch. Koch. Query evaluation on compressed trees (extended abstract). In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS'2003)*, pages 188–197. IEEE Computer Society Press, 2003.

[12] F. Gécseg and M. Steinby. *Tree automata*. Akadémiai Kiadó, 1984.

[13] G. Gottlob, Ch. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

[14] G. Gottlob, Ch. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the Association for Computing Machinery*, 52(2):284–335, 2005.

[15] T. Lengauer and K. W. Wagner. The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems. *Journal of Computer and System Sciences*, 44:63–93, 1992.

[16] M. Lohrey. On the parallel complexity of tree automata. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewrite Techniques and Applications (RTA 2001), Utrecht (The Netherlands)*, number 2051 in Lecture Notes in Computer Science, pages 201–215. Springer, 2001.

[17] M. Lohrey. Word problems on compressed word. In J. Diaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), Turku (Finland)*, number 3142 in Lecture Notes in Computer Science, pages 906–918. Springer, 2004. long version appears in SIAM Journal on Computing.

[18] M. Lohrey and S. Maneth. Tree automata and XPath on compressed trees. In J. Farré, I. Litovsky, and S. Schmitz, editors, *Proceedings of the Tenth International Conference on Implementation and Application of Automata (CIAA 2005), Sophia Antipolis (France)*, number 3845 in Lecture Notes in Computer Science. Springer, 2006. to appear.

[19] S. Maneth and G. Busatto. Tree transducers and tree compressions. In I. Walukiewicz, editor, *Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2004), Barcelona (Spain)*, number 2987 in Lecture Notes in Computer Science, pages 363–377. Springer, 2004.

[20] N. Markey and P. Schnoebelen. A PTIME-complete matching problem for SLP-compressed words. *Information Processing Letters*, 90(1):3–6, 2004.

[21] M. Marx. XPath and modal logics for finite DAG's. In M. C. Mayer and F. Pirri, editors, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2003), Rome (Italy)*, number 2796 in Lecture Notes in Computer Science, pages 150–164. Springer, 2003.

[22] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Proceedings of Extreme Markup Languages 2000*, 2000.

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

[24] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[25] C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3):407–427, 1999.

[26] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 1, pages 3–61. World Scientific, 1999.

[27] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In J. Diaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), Turku (Finland)*, number 3142 in Lecture Notes in Computer Science, pages 15–27. Springer, 2004.

[28] L. Segoufin. Typing and querying XML documents: some complexity bounds. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2003)*, pages 167–178. ACM Press, 2003.