View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Query Evaluation on Compressed Trees* (Extended Abstract)

Markus Frick Martin Grohe Christoph Koch

Abstract

This paper studies the problem of evaluating unary (or node-selecting) queries on unranked trees compressed in a natural structure-preserving way, by the sharing of common subtrees. The motivation to study unary queries on unranked trees comes from the database field, where querying XML documents, which can be considered as unranked labelled trees, is an important task.

We give algorithms and complexity results for the evaluation of XPath and monadic datalog queries. Furthermore, we propose a new automata-theoretic formalism for querying trees and give algorithms for evaluating queries defined by such automata.

1. Introduction

Semi-structured data, best-known in the syntax of XML, have caused a significant paradigm shift in the field of database systems, and have also been one of the central research topics in database theory over the last five years (see [1] and [24] for surveys). While classical relational databases can be described as relational structures, XML-documents are best modelled by unranked trees. This paper studies the problem of evaluating unary (or node-selecting) queries on unranked trees compressed in a natural structure-preserving way, by the *sharing of common subtrees*. Node-selecting queries are not only of interest as basic queries in their own right, but are also an important building block for more complex queries. In particular, the node-selecting path query language *XPath* is at the core of several major XML-related technologies, such as XML Query, XML Schema, and XSLT, the principal query language, schema definition formalism, and stylesheet language for XML, respectively. Thus, the efficient processing of XPath queries (and the study of node-selecting XML queries in general) is paramount to the overall success of all these technologies.

The compression of XML-trees into directed acyclic graphs by the sharing of subtrees has recently been proposed by Buneman, Grohe and Koch [6]. It can be seen as a direct generalisation of the compression of Boolean functions into OBDDs (cf. [5]) used so successfully in symbolic model checking [7, 8]. The approach bears the promise of advancing the state of the art in XML query processing at two fronts. First, compression allows to keep larger document trees in main memory (where they can be efficiently evaluated) and permits a substantial speedup by often avoiding the need to use slow secondary storage when trees are large and otherwise cannot be kept in main memory as a whole. Second, evaluating queries on compressed trees in practice saves time by avoiding redundant computations. In [6], it was implemented for a large fragment of XPath (*Core XPath*, which was introduced in [16]) and extensively benchmarked on practical XML documents several hundreds of Megabytes large and consisting of trees comprising tens of millions of nodes. The compression ratios obtained were very promising, and the actual efficiency of query processing obtained was astonishing.

The main objective of this paper is to create a solid theoretical foundation for the approach. The problem of evaluating Core XPath queries on compressed instances has been shown to be fixed-parameter tractable in [6]. More specifically, the problem has been shown to be solvable in time $O(k \cdot 2^k \cdot n)$, where k denotes the size of the query and n the size of the compressed instance. Complementing this result, here we prove that the problem is PSPACE-complete. Furthermore, we show that the problem of evaluating queries of the positive Core XPATH fragment (i.e., without negation) is NP-complete. Let us remark that the problem of evaluating Core XPath queries on uncompressed trees is known to be in polynomial time (actually, PTIME-complete [17]).

Even though undoubtedly very important in practice, from a theoretical perspective XPath seems to be a very ad-hoc language that leaves a lot to be desired. *Monadic second-order logic (MSO)* on trees, on the other hand, is well-known to have beautiful theoretical properties. In particular, it has well-balanced expressive power in that it is expressive enough for most purposes, but on the other hand still has good algorithmic properties due to its connection with tree automata. Indeed, MSO has been proposed as a "benchmark" for the expressive power of node-selecting XML query languages [25]. Nevertheless, MSO itself is not suitable as a practical query language because it allows to

*Author's addresses: Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK. Email: {mfrick,grohe}@inf.ed.ac.uk, koch@dbai.tuwien.ac.at

express very complex queries very concisely, which makes the query evaluation problem intractable even on uncompressed trees (cf. [27, 13]). But there are nice languages which have the same expressive power as MSO on trees, but admit much more efficient query evaluation. The modal μ-calculus may be seen as an example of such a language (at least on ranked trees). In the context of querying XML, the most promising such language is *monadic datalog*. It has the same expressive power as MSO, but admits query evaluation in time linear in both the size of the datalog program and the size of the tree [14].

We study the evaluation problem for monadic datalog queries on compressed instances. We show that, as for the strictly weaker Core XPath, the problem is PSPACE-complete. Of course the PSPACE-hardness was to be expected, but the containment of the problem in PSPACE may be viewed as mildly surprising. We then show that, again as for XPath, there is an algorithm solving the evaluation problem for monadic datalog on compressed instances in time $O(k \cdot 2^k \cdot n)$, where k denotes the size of the datalog program and n the size of the compressed instance.

Next, slightly digressing from the main focus of this paper, we discuss the connection between compressed binary and unranked trees. Even though XML-documents are naturally represented as unranked trees, we believe that it may be worthwhile to convert them into binary trees first and then only work with compressed binary trees. We show how such a conversion can be carried out without paying too high a price for it and observe that in certain situations the compressed binary instances may be exponentially smaller than the compressed unranked instances they represent. An additional advantage this may have in practice is that nodes of binary instances can be stored in a fixed size memory segment, whereas nodes of unranked instances may have adjacency lists of unbounded length, which tend to lead to high memory fragmentation. Experimental evidence (presented in the long version of the paper) suggests that the conversion to binary instances is worthwile in practice and may lead to more memory- and time-efficient XML query engines.

Returning to the query evaluation problem, an alternative approach to querying trees can be based on tree-automata.

Right from the beginning, automata-theoretic ideas have played a central role in XML-related research. Automata theory has been used for evaluating path and pattern queries [4, 25, 26, 14], as a basis for XML schema languages [20], for defining XML transducers [21], and for XML data stream processing [18]; See [24] for a survey of automata-theoretic work related to XML.

Most important in our context are *query automata*, proposed by Neven and Schwentick [26] to describe node-selecting queries on unranked trees. We suggest a similar automata model that we call *selecting tree automaton (STA)*. STAs have the same querying power and similar algorithmic properties as query automata, but we feel they are much cleaner and simpler. Even though our main interest here is in querying compressed instances, STAs are relevant in the uncompressed setting as well. We show that STA-queries on compressed instances can be evaluated in time $2^{O(s)} \cdot n$, where s is the size of the automaton and n the size of the compressed instance. Unfortunately, this seems to be too inefficient for practical purposes, because we usually cannot expect our automata to be so small that a factor in the running time that is exponential in s is acceptable. For example, translating a monadic datalog program into an equivalent STA causes an exponential blow-up in size. Therefore, we also consider a restricted model of *weak selecting tree automata*, whose definition captures the intuitions of monotonic inference in monadic datalog. This is witnessed by the fact that the above transformation from monadic datalog (which comes with the mentioned exponential blow-up) naturally yields weak STAs, and the time to evaluate such automata on compressed instances is linear. Therefore, in total, weak STAs evaluate monadic datalog programs within essentially the same time bound as the direct evaluation techniques discussed above.

The structure of this paper basically follows the order of contributions given above. Due to space limitations, the proofs of our results were beyond the scope of this extended abstract and will be supplied in the long version of the paper.

2. Compressed Trees

In this section, we review the framework for querying trees compressed by subtree sharing that has been introduced in [6]. We emphasise the central role of the familiar notion of bisimilarity in this framework.

2.1. Instances and Bisimilarity. A *schema* is a finite set of unary relation names. Let $\sigma = \{S_1, \dots, S_n\}$ be a schema. An *instance of schema* σ, or σ-*instance*, is a tuple

$$\boldsymbol{I} = \big(V^{\boldsymbol{I}}, \gamma^{\boldsymbol{I}}, root^{\boldsymbol{I}}, S_1^{\boldsymbol{I}}, \dots, S_n^{\boldsymbol{I}}\big),$$

where $V^{\boldsymbol{I}}$ is the (finite) set of vertices, $\gamma^{\boldsymbol{I}} : V^{\boldsymbol{I}} \to (V^{\boldsymbol{I}})^*$ is a function whose graph is acyclic and has a unique vertex of in-degree 0 from which all other vertices are reachable, $root^{\boldsymbol{I}}$ is this vertex of in-degree 0, and $S_1^{\boldsymbol{I}}, \dots, S_n^{\boldsymbol{I}}$ are subsets of $V^{\boldsymbol{I}}$.

Here the *graph of* γ is the directed graph with vertex set $V^{\boldsymbol{I}}$ and an edge from v to w if w occurs in $\gamma(v)$. We call this graph the *DAG of* $\boldsymbol{I}$. Occasionally we denote its edge relation by $E^{\boldsymbol{I}}$. For vertices $u, v \in V$ we write $u \xrightarrow{i} v$ if there is an $n \geq i$ and vertices $v_1, \dots, v_n$ such that $\gamma(u) = v_1 \dots v_n$ and $v = v_i$. Intuitively, if $u \xrightarrow{i} v$ then v is the *i-th child* of u and u is the *parent* of v. An instance is k-ary,

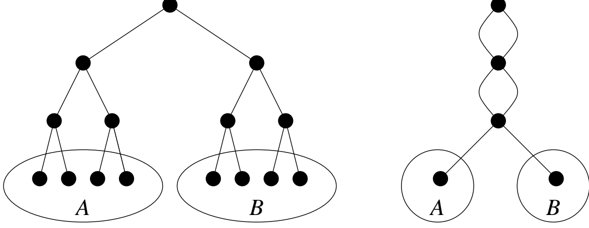

Figure 1. Two bisimilar instances over schema $\sigma = \{A, B\}$.

for some $k \geq 1$, if every vertex has at most k children. If the instance $\boldsymbol{I}$ is clear from the context, we often omit the superscript $^{\boldsymbol{I}}$.

A *tree instance* is an instance whose DAG is a tree. Tree-instances are our model of XML-documents. The unary relations represented by the schema are used to encode the relevant information carried by the vertices of an XML-tree. This may be the XML-tag of a vertex, but also information encoded in the alphanumerical data at the vertex. Indeed, the latter is of crucial importance because node-selecting queries will usually access the alphanumerical data. The implementation of Core XPath in [6] admits querying alphanumerical data through so-called *string constraints*.

A *bisimilarity relation* between two σ-instances $\boldsymbol{I}$ and $\boldsymbol{J}$ is a binary relation $\sim \subseteq V^{\boldsymbol{I}} \times V^{\boldsymbol{J}}$ such that for all $v \in V^{\boldsymbol{I}}, w \in V^{\boldsymbol{J}}$ with $v \sim w$ we have

- for all i and $v' \in V^{\boldsymbol{I}}$, if $v \xrightarrow{i} v'$ then there exists $w' \in V^{\boldsymbol{J}}$ such that $w \xrightarrow{i} w'$ and $v' \sim w'$,
- for all i and $w' \in V^{\boldsymbol{J}}$, if $w \xrightarrow{i} w'$ then there exists $v' \in V^{\boldsymbol{I}}$ such that $v \xrightarrow{i} v'$ and $v' \sim w'$, and
- for all $S \in \sigma$: $\big(v \in S^{\boldsymbol{I}} \iff w \in S^{\boldsymbol{J}}\big)$.

Thus, our notion of bisimilarity relation takes both node and edge labels that are present in the tree into account.

If there is some bisimilarity relation $\sim$ between $\boldsymbol{I}$ and $\boldsymbol{J}$ such that $v \sim w$, we call the vertices v and w *bisimilar* (we write $v \approx w$ or $(\boldsymbol{I}, v) \approx (\boldsymbol{J}, w)$). The instances $\boldsymbol{I}$ and $\boldsymbol{J}$ are *bisimilar* (we write $\boldsymbol{I} \approx \boldsymbol{J}$) if $root^{\boldsymbol{I}} \approx root^{\boldsymbol{J}}$. An example of two bisimilar instances is given in Figure 1.

If $\boldsymbol{I}$ is an instance and $\sim$ a bisimilarity relation on $\boldsymbol{I}$ (that is, between $\boldsymbol{I}$ and $\boldsymbol{I}$), then $\boldsymbol{I}/_\sim$ is the instance obtained from $\boldsymbol{I}$ by identifying all vertices v, w with $v \sim w$. Pairs of edges $v \xrightarrow{i} w$, $v' \xrightarrow{j} w'$ (with $v \sim v'$, $w \sim w'$) may only be identified in this process if $i = j$. We define a partial order $\preceq$ on the class of all σ-instances by letting $\boldsymbol{I} \preceq \boldsymbol{J}$ if there is a bisimilarity relation $\sim$ on $\boldsymbol{J}$ such that $\boldsymbol{I}$ is isomorphic to $\boldsymbol{J}/_\sim$. More precisely, $\preceq$ is a partial order on the set of all isomorphism classes of instances. But no harm is done by blurring this distinction here and in the following.

Lemma 1 ([6]). *Let $\boldsymbol{I}$ be a σ-instance and let $\mathcal{L}(\boldsymbol{I})$ be the class of all instances bisimilar to $\boldsymbol{I}$. Then $\big(\mathcal{L}(\boldsymbol{I}), \preceq\big)$ is a lattice. Its maximal element $\boldsymbol{T}(\boldsymbol{I})$ is the only tree instance in $\mathcal{L}(\boldsymbol{I})$. The minimal element $\boldsymbol{M}(\boldsymbol{I})$ is also characterised by the fact that it contains the least number of vertices of all instances in $\mathcal{L}(\boldsymbol{I})$.*

It will be necessary later to have a canonical definition of $\boldsymbol{T}(\boldsymbol{I})$, and not just a characterisation up to isomorphism. If v, v' are vertices in an instance $\boldsymbol{I}$, and there are intermediate vertices $v_1, \ldots, v_{n-1}$ such that $v \xrightarrow{i_1} v_1 \ldots v_{n-1} \xrightarrow{i_n} v'$, we say that the integer sequence $i_1 \ldots i_n$ is an *edge-path* between v and v'. For each vertex $v \in V$ we define

$$\Pi(v) = \{P \mid P \text{ is an edge-path from } root \text{ to } v\},$$

and for a set $S \subseteq V$ we let $\Pi(S) = \bigcup_{v \in S} \Pi(v)$. Note that the vertices of $\boldsymbol{T}(\boldsymbol{I})$ are in one-to-one correspondence with the elements of $\Pi(V)$. Thus we can define our canonical representation of $\boldsymbol{T}(\boldsymbol{I})$ to have vertex set $\Pi(V)$ (and all relations defined in the obvious way). Also note that for bisimilar instances $\boldsymbol{I} \approx \boldsymbol{J}$ and vertices $v \in V^{\boldsymbol{I}}, w \in V^{\boldsymbol{J}}$ we have $v \approx w$ if and only if $\Pi(v) = \Pi(w)$.

2.2. Representations and Size. Our machine model is a standard RAM model with addition and subtraction as arithmetic operations. We use a uniform cost measure. While for some of the theoretical considerations of this paper a logarithmic cost measure would be nicer (cf. Remark 14), we think that for the practical analysis of our algorithms a uniform measure is most appropriate. After all, a main motivation for this research is to give main memory algorithms for querying XML-documents, and this basically means that in our algorithms we never have to handle numbers that do not fit into a single memory word.

We represent instances in a straightforward way based on an adjacency list representation of the underlying DAG. There is one important twist: Instances may contain multiple edges, and instead of storing them all separately, we just use one adjacency list entry which contains an integer representing the multiplicity to represent consecutive edges from a vertex to a child. Since the order of the children of a vertex is important, we can only do this for consecutive edges to the same child (see Figure 2). In practice, this concise representation of multiple edges is extremely important, because XML-trees tend to be very broad and shallow, and therefore their compressed versions tend to have many multiple edges. We denote the size of the representation of an instance $\boldsymbol{I}$ by $||\boldsymbol{I}||$. Note that we have $|V^{\boldsymbol{I}}| \leq O(||\boldsymbol{I}||)$ and $||\boldsymbol{I}|| \leq O(|V^{\boldsymbol{I}}|^2)$. Moreover, we have $||\boldsymbol{I}|| \leq O(|E^{\boldsymbol{I}}|)$, but our concise representation of multiple edges and the uniform cost model imply that we cannot bound $|E^{\boldsymbol{I}}|$ in terms of $||\boldsymbol{I}||$. If we let mult$(\boldsymbol{I})$ be the maximum number of consecutive multiple edges, that is, the maximum number of

edges represented by a single adjacency list entry, we have $|E^{\boldsymbol{I}}| \leq O(||\boldsymbol{I}|| \cdot \text{mult}(\boldsymbol{I}))$.

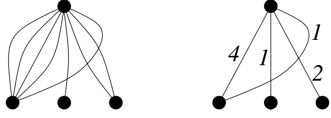

Figure 2.

The following earlier result refers to the computation of compressed instances even *with edge multiplicities*:

Theorem 2 ([6]). *There is an algorithm that, given an instance $\boldsymbol{I}$, computes the minimal bisimilar instance $\boldsymbol{M}(\boldsymbol{I})$ in time $O(||\boldsymbol{I}||)$.*

3. Queries and Query Languages

3.1. Queries. Since we think of an instance $\boldsymbol{I}$ as being a compressed representation of the tree instance $\boldsymbol{T}(\boldsymbol{I})$, we define the semantics of queries with respect to the tree instances. Our notion of query is based on the use of this term in finite model theory: A *(unary) query Q of schema σ* associates with every tree instance $\boldsymbol{T}$ of schema σ a subset $Q(\boldsymbol{T}) \subseteq V^{\boldsymbol{T}}$ in such a way that for every isomorphism π from a tree-instance $\boldsymbol{T}$ to a tree instance $\boldsymbol{T}'$ we have $\pi(Q(\boldsymbol{T})) = Q(\boldsymbol{T}')$. Note that the term "query" usually has a different meaning in the theory of semi-structured data; there a query is a mapping from tree instances to tree instances (see [1]). What we call unary query here is usually called *pattern* in this framework.

We want to evaluate queries on compressed instances, preferably without fully decompressing them. The problem is that we cannot always represent the result of a query in the instance we are given: While every subset $X \subseteq V^{\boldsymbol{I}}$ canonically corresponds to the subset $\Pi(X) \subseteq V^{\boldsymbol{T}(\boldsymbol{I})} = \Pi(V^{\boldsymbol{I}})$, unless $\boldsymbol{I} = \boldsymbol{T}(\boldsymbol{I})$ it is not the case that for every set $Y \subseteq V^{\boldsymbol{T}(\boldsymbol{I})}$ there is a set $X \subseteq V^{\boldsymbol{I}}$ such that $Y = \Pi(X)$. So to represent the answer of a query we may have to partially decompress the instance. The following definition makes this precise:

Definition 3. The *evaluation problem* for a query language L on a class C of instances is the following problem:

Input:	Instance $\boldsymbol{I} \in C$ and query $Q \in L$.
Problem:	Return an instance $\boldsymbol{J}$ and a subset $\tilde{Q} \subseteq V^{\boldsymbol{J}}$ such that $\boldsymbol{I}$ and $\boldsymbol{J}$ are bisimilar and $Q(\boldsymbol{T}(\boldsymbol{I})) = \Pi(\tilde{Q})$.

If C is not explicitly mentioned, it is understood to be the class of all instances.

Our complexity-theoretic results only refer to the *decision version* of the evaluation problem:

Input:	Instance $\boldsymbol{I} \in C$, vertex $v \in V^{\boldsymbol{T}(\boldsymbol{I})}$, and query $Q \in L$.
Problem:	Decide if $v \in Q(\boldsymbol{T}(\boldsymbol{I}))$.

In this definition, we supply a node of the *uncompressed* tree-version of the instance $\boldsymbol{I}$ with the input, since nodes of $\boldsymbol{I}$ do not necessarily exist in $Q(\boldsymbol{I})$. Clearly, a good way to formulate global properties of instances as decision problems is to check queries on the root node, since $\Pi(\textit{root}^{\boldsymbol{I}}) = \{\textit{root}^{T(\boldsymbol{I})}\}$ for all instances.

3.2. Complexity. We assume that the reader is familiar with the standard complexity classes such as PTIME, NP, and PSPACE. It is convenient to phrase some of our results in the terminology of fixed-parameter tractability (see [10], or [19] for a short introduction into the notions most relevant here). Actually, we only need one definition: The evaluation problem for L on C is *fixed parameter tractable* if there is a computable function f, a constant c, and an algorithm solving the problem in time $f(k) \cdot n^c$, where n is the size of the input instance and k the size of the input query.

3.3. Logic and relational structures. We assume that the reader is familiar with relational structures, first-order logic FO, and monadic second-order logic MSO (see, for example, [11]). In the logical context, we describe σ-tree instances as relational structures whose vocabulary consists of all unary relation symbols in the schema σ and, in addition, the unary relation symbols *Root*, *Leaf*, *Last-Sibling*, and the binary relation symbols *First-Child* and *Next-Sibling*, all with the natural meanings (cf. [14]). We occasionally call *Root*, *Leaf*, *Last-Sibling*, *First-Child*, and *Next-Sibling* the *built-in predicates*. We use $\boldsymbol{T}$ to denote both the tree instance and the relational structure representing it.

If $\boldsymbol{T}$ is a tree instance and $\varphi(x)$ an MSO-formula with one free variable, then we let $\varphi(\boldsymbol{T})$ be the set of all vertices $v \in V^{\boldsymbol{T}}$ such that $\boldsymbol{T}$ satisfies $\varphi(x)$ if x is interpreted by v. We call $\boldsymbol{T} \mapsto \varphi(\boldsymbol{T})$ the query defined by φ.

3.4. Monadic datalog. We assume that the reader is familiar with datalog, which may be viewed as logic programming without function symbols (cf. [2]). A datalog program is *monadic* if all its IDB predicates (that is, intensional predicates that appear in rule heads somewhere in the program) are unary. We interpret monadic datalog programs over tree instances. A monadic datalog program of schema σ may use as EDB predicates (that is, extensional predicates which are determined by the structure the program is interpreted over) the built-in predicates *Root*, *Leaf*, *Last-Sibling*, the binary relation symbols *First-Child* and *Next-Sibling*, the predicates in σ, and a predicate $\overline{S}$ for every $S \in \sigma$ which

is interpreted as the complement of S. Each program $\mathcal{P}$ has a distinguished *goal (IDB) predicate*. The query defined by $\mathcal{P}$ maps a tree instance $\boldsymbol{T}$ to the set of all vertices v such that $\mathcal{P}$ derives over $\boldsymbol{T}$ that v is in the goal predicate. Two programs are *equivalent* if they define the same query.

The popular fixpoint semantics of (monadic) datalog can be defined by means of a small-step[1] monotonic *immediate consequence operator* $\mathcal{T}_\mathcal{P}$. Given a set of ground (i.e., variable-free) atoms X, $\mathcal{T}_\mathcal{P}$ chooses a rule of $\mathcal{P}$ (with head predicate P) and a node v such that X satisfies the rule body and a new ground atom $P(v)$ can be inferred (then, $\mathcal{T}_\mathcal{P}(X) = X \cup \{P(v)\}$). We write the k-times iterated application of $\mathcal{T}_\mathcal{P}$ as ${\mathcal{T}_\mathcal{P}}^k$ and the fixpoint ${\mathcal{T}_\mathcal{P}}^m = {\mathcal{T}_\mathcal{P}}^{m+1}$ as ${\mathcal{T}_\mathcal{P}}^\omega$. In this context, we may consider an instance $\boldsymbol{T}$ as a set of unary and binary ground atoms and evaluate $\mathcal{P}$ on $\boldsymbol{T}$ as ${\mathcal{T}_\mathcal{P}}^\omega(\boldsymbol{T})$.

We only use monadic datalog programs with a restricted syntax described next. In a *TMNF program* ("Tree-marking Normal Form"), each rule is an instance of one of the four rule templates (with "types" 1 to 4)

$$P(x) \leftarrow U(x). \quad (1)$$
$$P(x) \leftarrow P_0(x_0) \wedge B(x_0, x). \quad (2)$$
$$P(x_0) \leftarrow P_0(x) \wedge B(x_0, x). \quad (3)$$
$$P(x) \leftarrow P_1(x) \wedge P_2(x). \quad (4)$$

where P, P_0, P_1, P_2 are IDB predicates and U, B are EDB predicates.

Proposition 4 ([14]). *Every monadic datalog program (over trees) can be translated into an equivalent TMNF program in linear time.*

Thus, in the following, we only deal with programs in TMNF. All worst-case bounds for TMNF query evaluation translate immediately to the evaluation of monadic datalog. Note that monadic datalog captures MSO over trees [14] and a program $\mathcal{P}$ can be evaluated in time $O(||\boldsymbol{T}|| * |\mathcal{P}|)$ when $\boldsymbol{T}$ is a tree-instance [14]. The following new result is based on a lazy rule-instantiation version of an algorithm by Minoux [22]:

Proposition 5. *A TMNF program $\mathcal{P}$ can be evaluated on tree-instance $\boldsymbol{T}$ in time $O(|\boldsymbol{T}| + |\mathcal{P}| * |{\mathcal{T}_\mathcal{P}}^\omega(\boldsymbol{T}) - \boldsymbol{T}|)$.*

Proof of Proposition 5: It is easy to verify by inspection that the algorithm of Figure 3 indeed computes ${\mathcal{T}_\mathcal{P}}^\omega(\boldsymbol{T}) - \boldsymbol{T}$ and stores it in Ω.

The main idea responsible for the low runtime bound is to use a queue containing newly derived atoms into which

[1] Note that this nondeterministic defi nition is slightly nonstandard ut makes certain induction proofs more straightforward. Usually, the immediate consequence operator adds to X in one step *all* the atoms that can be inferred as described here from the atoms in X. Of course, the fi xpoint of the both versions of the operator is the same on all instances and programs.

Input: program $\mathcal{P}$, tree-instance $\boldsymbol{T}$.
Output: set of ground atoms $\Omega = {\mathcal{T}_\mathcal{P}}^\omega(\boldsymbol{T}) - \boldsymbol{T}$.

Initializations:
// waiting queue for true ground atoms.
$W := \{P(v) \mid P(x) \leftarrow U(x).$ is in $\mathcal{P}$,
  $U(v)$ is true w.r.t. $\boldsymbol{T}\}$;

$\Omega := W$; // results: set of ground atoms.

// rules by predicate.
map R: predicate → set of rule-ids;
for each IDB predicate P **do**
  $R[P] := \{r \mid$ the body of r contains $P\}$;

// half-finished ground rules of type 4.
map Aux_4: rule-id → set of ground atoms;
for each rule $r \in \mathcal{P}$ of type 4 **do** $Aux_4[r] := \emptyset$;

Main loop:
while (W not empty) **do**
begin
  take a ground atom $P(v)$ off W;

  // rules of type 2 and 3
  for each rule $P'(x) \leftarrow P(x_0) \wedge B(x_0, x)$.
    in $R[P]$, $B(v, w)$ true w.r.t. $\boldsymbol{T}$ or
    $P'(x) \leftarrow P(x_0) \wedge B(x, x_0)$.
    in $R[P]$, $B(w, v)$ true w.r.t. $\boldsymbol{T}$ **do**
  // where B is either *First-Child* or *Next-Sibling*
  begin
    if ($P'(w)$ not yet in Ω) **then**
      add $P'(w)$ to W and to Ω;
  end;

  // rules of type 4
  for each rule $r = P''(x) \leftarrow P(x) \wedge P'(x)$.
    in $R[P]$ **do**
  begin
    if ($P(v)$ not in $Aux_4[r]$) **then**
      add $P'(v)$ to $Aux_4[r]$;
      // what is actually waiting now
      // is the ground rule $P''(v) \leftarrow P'(v)$.
    else
      // $P'(v)$ has been satisfied before and
      // $P(v)$ is satisfied now as well - rule fires.
      if ($P''(v)$ not yet in Ω) **then**
        add $P''(v)$ to W and to Ω;
  end;
end;

Figure 3. Modified Minoux algorithm for TMNF.

no atom is ever inserted twice and which governs the inference of further atoms. Moreover, rules r with several IDB body atoms (i.e., those of type 4) are instantiated (that is, their variable is matched and replaced with a node) immediately when a first appropriate atom a is reached in the queue W. The body atom matching a is also removed from the instantiated version of r, leading to a rule with one body atom less. (Note that we assume that rule bodies are sets and thus that there are no duplicate atoms in rule bodies.)

The initialization phase (before the start of the while loop) is dominated by the initial value assignment to W in time $O(|\mathcal{T_P}^{\omega}(\boldsymbol{T}) - \boldsymbol{T}|)$ and to R in time $O(|\mathcal{P}|)$.

The while-loop that follows iterates exactly $|\mathcal{T_P}^{\omega}(\boldsymbol{T}) - \boldsymbol{T}|$ times (once for each IDB ground atom inferred). We assume appropriate data structures (e.g. an array of size $|\mathcal{P}| * |V^{\boldsymbol{T}}|$ for Ω) that allow to do all data structure look-ups in constant time. The first for-loop runs in constant time, as it may have at most four iterations. The second for-loop (handling rules of type 4) may take time $O(|\mathcal{P}|)$, thus the overall while loop can be processed in time $O(|\mathcal{P}| * |\mathcal{T_P}^{\omega}(\boldsymbol{T}) - \boldsymbol{T}|)$. The time bound of Proposition 5 follows. □

Note that by definition, $\mathcal{T_P}^{\omega}(\boldsymbol{T})$ contains $\boldsymbol{T}$ as a set of ground atoms.

3.5. Core XPath. XPath uses thirteen binary relations – called axes – for navigating in trees [30]. We only need to introduce five of them in this paper, *Self* (the identity relation on V), *Child* (the intuitive child relation; *Child*(v, w) iff w is a child of v), *Parent* (its inverse), *Descendant* (the transitive closure of *Child*), and *Ancestor* (its inverse). Given a binary axis relation χ, an axis is its *inverse*, denoted χ^{-1}, if $\langle x, y\rangle \in \chi \Leftrightarrow \langle y, x\rangle \in \chi^{-1}$. For each XPath axis, there is also an XPath axis that is its inverse. For example, $\mathit{Self}^{-1} = \mathit{Self}$, $\mathit{Child}^{-1} = \mathit{Parent}$, and $\mathit{Descendant}^{-1} = \mathit{Ancestor}$. In the following definition of a fragment of XPath, we assume all 13 axes to be supported, even if only a few have been introduced here (for a complete formal definition of Core XPath see [16]).

Definition 6. Let $\boldsymbol{T}$ be a tree-instance. We define the syntax of *Core XPath* by the EBNF

corexpath:	locationpath | '/' locationpath
locationpath:	locationstep ('/' locationstep)*
locationstep:	χ '::' P | χ '::' P '[' pred ']'
pred:	pred 'and' pred | pred 'or' pred
	| 'not(' pred ')' | corexpath | '(' pred ')'

"corexpath" is the start production, χ stands for an axis, and P for a "node test", that is, a unary relation from σ or "*", meaning "any node" $V^{\boldsymbol{T}}$.

The semantics of Core XPath queries on tree-instances $\boldsymbol{T}$ is defined by two functions $\mathcal{S}$ and $\mathcal{E}$ (for Core XPath expressions and condition predicates, respectively):

$$
\begin{array}{rcl}
\mathcal{S} & : & \mathcal{L}(\mathit{corexpath}) \rightarrow 2^{V^{\boldsymbol{T}} \times V^{\boldsymbol{T}}} \\
\mathcal{S}[\![\chi{::}P[e]]\!] & := & \{\langle x, y\rangle \mid \chi^{\boldsymbol{T}}(x, y) \wedge y \in (P \cap \mathcal{E}[\![e]\!])\} \\
\mathcal{S}[\![/\pi]\!] & := & V^{\boldsymbol{T}} \times \{x \mid \langle \mathit{root}^{\boldsymbol{T}}, x\rangle \in \mathcal{S}[\![\pi]\!]\} \\
\mathcal{S}[\![\pi_1/\pi_2]\!] & := & \{\langle x, z\rangle \mid \exists y : \langle x, y\rangle \in \mathcal{S}[\![\pi_1]\!] \wedge \\
& & \qquad \langle y, z\rangle \in \mathcal{S}[\![\pi_2]\!]\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} & : & \mathcal{L}(\mathit{pred}) \rightarrow 2^{V^{\boldsymbol{T}}} \\
\mathcal{E}[\![e_1 \text{ and } e_2]\!] & := & \mathcal{E}[\![e_1]\!] \cap \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![e_1 \text{ or } e_2]\!] & := & \mathcal{E}[\![e_1]\!] \cup \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![\text{not}(e)]\!] & := & V^{\boldsymbol{T}} - \mathcal{E}[\![e]\!] \\
\mathcal{E}[\![\pi]\!] & := & \{x_0 \mid \exists x : \langle x_0, x\rangle \in \mathcal{S}[\![\pi]\!]\}
\end{array}
$$

Here, π, π_1 and π_2 are location paths. Query Q results in the set $\{y \mid \exists x : \langle x, y\rangle \in \mathcal{S}[\![Q]\!]\}$.

An example of a Core XPath query is

/descendant::*[child::A and child::B]/child::*,

which selects all children of descendants of the root node that (i.e., the descendants) have a child node labelled A and a child node labeled B.

Core XPath is a strict fragment of XPath [30], both syntactically and semantically.

A mapping from Core XPath to monadic datalog with stratified negation was given in [15]; we strengthen this result to (negation-free) TMNF. Note again that we assume that Core XPath supports all XPath axes.

Proposition 7. *Every Core XPath query can be translated into an equivalent TMNF program in linear time and logarithmic space.*

Proof of Proposition 7 (Sketch): To keep the encoding of this proof simple, we do not strictly adhere to TMNF syntax. However, all datalog rules can be transformed into TMNF by folding pairs of unary atoms from long rules into separate rules. For instance, a rule

$$P(y) \leftarrow P_1(x) \wedge P_2(x) \wedge P_3(x).$$

rewrites into

$$
\begin{array}{rcl}
P(y) & \leftarrow & P'(x) \wedge P_3(x). \\
P'(x) & \leftarrow & P_1(x) \wedge P_2(x).
\end{array}
$$

where P' is a new predicate. This is not an additional transformation that has to be composed with the proof, but can be applied a priori to our encoding.

The encoding presented here is closely based on one presented in [15]; however, there, negation was encoded using datalog with stratified negation, which we do not want

$$
\begin{array}{rcl}
\mathit{Child}_P(x) & \leftarrow & P(x_0) \wedge \mathit{First\text{-}Child}(x_0, x). \\
\mathit{Child}_P(x) & \leftarrow & \mathit{Child}_P(x_0) \wedge \mathit{Next\text{-}Sibling}(x_0, x). \\
\\
\overline{\mathit{Child}_P}(x) & \leftarrow & \mathit{Root}(x). \\
\overline{\mathit{Child}_P}(x) & \leftarrow & \overline{P}(x_0) \wedge \mathit{First\text{-}Child}(x_0, x). \\
\overline{\mathit{Child}_P}(x) & \leftarrow & \overline{\mathit{Child}_P}(x_0) \wedge \mathit{Next\text{-}Sibling}(x_0, x). \\
\\
\mathit{Parent}_P(x) & \leftarrow & \mathit{Aux\text{-}Parent}_P(x_0) \wedge \\
& & \mathit{First\text{-}Child}(x, x_0). \\
\mathit{Aux\text{-}Parent}_P(x) & \leftarrow & P(x). \\
\mathit{Aux\text{-}Parent}_P(x) & \leftarrow & \mathit{Aux\text{-}Parent}_P(x_0) \wedge \\
& & \mathit{Next\text{-}Sibling}(x, x_0). \\
\\
\overline{\mathit{Parent}_P}(x) & \leftarrow & \mathit{Leaf}(x). \\
\overline{\mathit{Parent}_P}(x_0) & \leftarrow & \mathit{First\text{-}Child}(x_0, x) \wedge \\
& & \mathit{Aux\text{-}}\overline{\mathit{Parent}_P}(x). \\
\mathit{Aux\text{-}}\overline{\mathit{Parent}_P}(x) & \leftarrow & \overline{P}(x) \wedge \mathit{Last\text{-}Sibling}(x). \\
\mathit{Aux\text{-}}\overline{\mathit{Parent}_P}(x) & \leftarrow & \overline{P}(x) \wedge \mathit{Next\text{-}Sibling}(x, y) \wedge \\
& & \mathit{Aux\text{-}}\overline{\mathit{Parent}_P}(y).
\end{array}
$$

Figure 4. Axis encodings (for *Child*, *Parent*, and their complements).

to use here.[2] The idea is to push down negation to unary built-in predicates, for which we, by the definitions of Section 3.4, have the complements available. The slightly tricky part is the (binary) axis relations, which are also the only part of a Core XPath query for which recursion is needed in our encoding. As in [15], for each axis χ, we can set up a basically fixed template program (modulo occurrences of IDB predicate P) defining a predicate χ_P with the intuitive meaning

$$\chi_P(x) \leftarrow P(x_0) \wedge \chi(x_0, x).$$

As TMNF with the given built-in relations provides all the machinery to check a given predicate P on *each* of the nodes of a region in the tree relevant to an axis, negated axes $\overline{\chi_P}$ (denoting the complement of the set of nodes to which χ_P evaluates) can be encoded as well. We show implementations of Child_P, Parent_P, $\overline{\mathit{Child}_P}$, and $\overline{\mathit{Parent}_P}$ in Figure 4 and an implementation of $\overline{\mathit{Ancestor}_P}$ in the example of Figure 5.

First we provide the encoding for positive Core XPath, i.e. for Core XPath without negation. This is precisely as in [15]. Given a query π, we obtain a program computing π by starting with the query predicate $\mathcal{S}_R[\![\pi]\!]$ and including rules according to the patterns shown below until all IDB predicates have been defined.

$$
\begin{array}{rcl}
\mathcal{S}_R[\![\chi{::}P[e]]\!](x) & \leftarrow & \chi_V(x) \wedge P(x) \wedge \mathcal{E}[\![e]\!](x). \\
\mathcal{S}_R[\![/\chi{::}P[e]]\!](x) & \leftarrow & \chi_{\mathit{Root}}(x) \wedge P(x) \wedge \mathcal{E}[\![e]\!](x). \\
\mathcal{S}_R[\![\pi/\chi{::}P[e]]\!](x) & \leftarrow & \chi_{\mathcal{S}_R[\![\pi]\!]}(x) \wedge P(x) \wedge \mathcal{E}[\![e]\!](x). \\
\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]]\!](x) & \leftarrow & P(x) \wedge \mathcal{E}[\![e]\!](x). \\
\mathcal{S}_L[\![\chi{::}P[e]]\!](x) & \leftarrow & \chi^{-1}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]]\!]}(x). \\
\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]/\pi]\!](x) & \leftarrow & \mathcal{S}_L[\![\pi]\!](x) \wedge P(x) \wedge \mathcal{E}[\![e]\!](x). \\
\mathcal{S}_L[\![\chi{::}P[e]/\pi]\!](x) & \leftarrow & \chi^{-1}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]/\pi]\!]}(x). \\
\mathcal{E}[\![\pi]\!](x) & \leftarrow & \mathcal{S}_L[\![\pi]\!](x). \\
\mathcal{E}[\![e_1 \text{ and } e_2]\!](x) & \leftarrow & \mathcal{E}[\![e_1]\!](x) \wedge \mathcal{E}[\![e_2]\!](x).
\end{array}
$$

$$\mathcal{E}[\![e_1 \text{ or } e_2]\!](x) \leftarrow \mathcal{E}[\![e_1]\!](x). \quad \mathcal{E}[\![e_1 \text{ or } e_2]\!](x) \leftarrow \mathcal{E}[\![e_1]\!](x).$$

Note that as in the encoding of [15], query subexpressions within brackets (i.e., conditions) are intuitively "reversed" (using the $\mathcal{S}_L$ predicates and the axis inverses χ^{-1}) to direct all computation in the query tree towards the "hot point" that selects nodes.

Now for negation, inside boolean condition expressions (with "and", "or" and "not" as operations and path expressions assumed atomic), negations are pushed down as far as possible using De Morgan's laws (which of course do not lead to an increase in formula size). Then, we proceed as above with the additional rule templates

$$
\begin{array}{rcl}
\mathcal{E}[\![\text{not}(\pi)]\!](x) & \leftarrow & \overline{\mathcal{S}_L[\![\pi]\!]}(x). \\
\overline{\mathcal{S}_L[\![\chi{::}P[e]]\!]}(x) & \leftarrow & \overline{\chi^{-1}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]]\!]}}(x). \\
\overline{\mathcal{S}_L[\![\chi{::}P[e]/\pi]\!]}(x) & \leftarrow & \overline{\chi^{-1}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]/\pi]\!]}}(x).
\end{array}
$$

Rules defining predicates such as $\overline{\mathit{Aux}\text{-}\mathcal{S}_L[\![P[e]/\pi]\!]}$ can be easily obtained by again applying De Morgan's law, by which the negation of a rule

$$P(x) \leftarrow P_1(x) \wedge \cdots \wedge P_n(x).$$

becomes

$$\overline{P}(x) \leftarrow \overline{P_1}(x). \quad \ldots \quad \overline{P}(x) \leftarrow \overline{P_n}(x).$$

Even if not obvious from this rewriting-based discussion, this translation can be carried out in linear time and logarithmic space. Figure 5 gives an example. □

A natural restriction on Core XPath is to exclude negation. We call the language obtained *positive Core XPath*. This fragment is also interesting as while both monadic datalog and Core XPath are P-complete w.r.t. combined complexity on tree-instances, positive Core XPath is LOGCFL-complete and thus effectively parallelizable [17].

[2]Even though extending monadic datalog with stratified negation changes neither the complexity nor the expressiveness of the formalism.

$$
\begin{array}{rcl}
\mathit{Child}_{\mathit{Root}}(x) & \leftarrow & \mathit{Root}(x_0) \wedge \mathit{First\text{-}Child}(x_0, x). \\
\mathit{Child}_{\mathit{Root}}(x) & \leftarrow & \mathit{Child}_{\mathit{Root}}(x_0) \wedge \mathit{Next\text{-}Sibling}(x_0, x). \\
\mathcal{S}_R[\![\text{/child::A[not(descendant::B)]}]\!](x) & \leftarrow & \mathit{Child}_{\mathit{Root}}(x) \wedge A(x) \wedge \mathcal{E}[\![\text{not(descendant::B)}]\!](x). \\
\mathcal{E}[\![\text{not(descendant::B)}]\!](x) & \leftarrow & \overline{\mathcal{S}_L[\![\text{descendant::B}]\!]}(x). \\
\overline{\mathcal{S}_L[\![\text{descendant::B}]\!]}(x) & \leftarrow & \overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x). \\
\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) & \leftarrow & \mathit{Leaf}(x). \\
\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) & \leftarrow & \mathit{First\text{-}Child}(x, y) \wedge \mathit{Aux}\text{-}\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(y). \\
\mathit{Aux}\text{-}\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) & \leftarrow & \overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) \wedge \overline{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}(x) \wedge \mathit{Last\text{-}Sibling}(x). \\
\mathit{Aux}\text{-}\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) & \leftarrow & \overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(x) \wedge \overline{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}(x) \wedge \\
 & & \mathit{Next\text{-}Sibling}(x, y) \wedge \mathit{Aux}\text{-}\overline{\mathit{Ancestor}_{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}}(y). \\
\overline{\mathit{Aux}\text{-}\mathcal{S}_L[\![B]\!]}(x) & \leftarrow & \overline{B}(x).
\end{array}
$$

Figure 5. Encoding for Core XPath query /child::A[not(descendant::B). A and B (and $\overline{B}$) are predicates in the schema and $\mathcal{S}_R[\![\text{/child::A[not(descendant::B)]}]\!]$ is the goal predicate.

4. Complexity and Evaluation of Monadic Datalog

In this section, we study the complexity of query evaluation for TMNF and Core XPath on compressed instances. Edge multiplicities, as introduced before, may lead to some difficulty; we assume such multiplicities not to be present in the instances of this section. Since they can have practical impact on the compression rate (cf. [6]), we refer to Section 5 where a mapping of unranked trees with edge multiplicities to binary trees is described.

Before we arrive at our main complexity results, we provide a polynomial-space algorithm that computes the predicates derivable for a given node, compressed instance, and TMNF program.

Let $\boldsymbol{T}$ be a tree-instance and $v \in V^{\boldsymbol{T}}$ a node. Moreover, let $\boldsymbol{T}_v$ denote the subtree of $\boldsymbol{T}$ rooted by v. $\boldsymbol{T}_v$ is again an instance. Let X be a set of ground atoms and V a set of nodes. Then, by $X/_V$ we denote $\{P(v) \in X \mid v \in V, P \in IDB(\mathcal{P})\}$, $X/_v$ is short for $X/_{\{v\}}$, and we abbreviate $X/_{V^{\boldsymbol{T}}}$ as $X/_{\boldsymbol{T}}$.

In analogy to $\boldsymbol{T}_v$ (the subtree of $\boldsymbol{T}$ rooted by v), let $\overline{\boldsymbol{T}_v}$ denote the complement (or *envelope*) of $\boldsymbol{T}_v$, the tree obtained by removing all of $\boldsymbol{T}_v$ from $\boldsymbol{T}$ except for v (which becomes a leaf). Just like $\boldsymbol{T}_v$, $\overline{\boldsymbol{T}_v}$ is again a tree-instance.

The following central lemma now states a stronger version of the fact that (a) in order to decide whether an atom inside a subtree $\boldsymbol{T}_v$ can be derived with program $\mathcal{P}$ (i.e. whether the atom is in $\mathcal{T_P}^{\omega}(\boldsymbol{T})$), all we need to know is the structure of $\boldsymbol{T}_v$ itself and the atoms that are derivable for node v on $\boldsymbol{T}$ (i.e., the atoms of $\mathcal{T_P}^{\omega}(\boldsymbol{T})/_v$), and (b) the converse fact with $\boldsymbol{T}_v$ and $\overline{\boldsymbol{T}_v}$ exchanged.

The $\mathcal{T_P}$ operator is nondeterministic, but we may compose a particular run $\mathcal{T_P}^k(X)$ into a new deterministic operator $\mathcal{O_P}$. We denote this by $\mathcal{O_P} := (\mathcal{T_P}^k)[X]$. We assume $\mathcal{O_P}$ to choose the same rules and nodes as the particular run of $\mathcal{T_P}^k$ in the same order. We extend the applicability of $\mathcal{O_P}$ to sets of ground atoms $Y \subset X$ by stepping over rules that cannot fire because their bodies are not satisfied in Y without doing anything.

Lemma 8. *Let $\mathcal{P}$ be a TMNF program, $\boldsymbol{T}$ a tree-instance, and $\mathcal{O_P} := (\mathcal{T_P}^k)[\boldsymbol{T}]$. Then,*

$$\mathcal{O_P}(\boldsymbol{T})/_{\boldsymbol{T}_v} = \mathcal{O_P}(\boldsymbol{T}_v \cup \mathcal{O_P}(\boldsymbol{T})/_v)$$

and

$$\mathcal{O_P}(\boldsymbol{T})/_{\overline{\boldsymbol{T}_v}} = \mathcal{O_P}(\overline{\boldsymbol{T}_v} \cup \mathcal{O_P}(\boldsymbol{T})/_v).$$

Proof of Lemma 8 (Sketch): By induction on the k steps of the operator. There are two cases of rules. In the first (which corresponds to TMNF rule templates (1) and (4)), the body contains only a single variable. Thus, the rule application has only strictly local impact, on a single node. In the second case (corresponding to TMNF rule templates (2) and (3)), a rule has at most one single binary atom and a single further IDB atom in the body (which may depend on previously derived facts). Here, a new fact $P(v)$ may enable the derivation of facts on *different* nodes, but as an immediate consequence, only facts on nodes adjacent in $\boldsymbol{T}$ w.r.t. the *First-Child* or *Next-Sibling* relations can be inferred. An atom $P(v)$ can only contribute to the derivation of another atom $Q(w)$, where v and w are not adjacent in $\boldsymbol{T}$, if first a trace of new atoms is computed for each of the nodes on the undirected path from v to w in $\boldsymbol{T}$. Let v' be a node on this undirected path between v and w, and $P'(v')$ this witness

```
function fixp(v0: node,
              X0: set of ground IDB atoms over v0,
              k: integer ≥ 0)
  /* let v0 have n ≥ k children v1 ... vn */
  returns set of ground IDB atoms over vk
begin
  X := X0 ∪ T/{v0,v1,...,vn};
  while no fixpoint reached do
    X := T_P(X) ∪
         fixp(v1, X/v1, 0) ∪ ··· ∪ fixp(vn, X/vn, 0);
  return X/vk;
end.
```

Figure 6. The function fixp.

atom following from $P(v)$. Then, subsequently, it suffices to know $P'(v')$ to infer $Q(w)$; $P(v)$ is not needed. □

Lemma 9. *Let $\boldsymbol{T}$ be a tree-instance, $v_0 \in V^{\boldsymbol{T}}$ a node with $n \geq 0$ children $v_1 \dots v_n$, X_0 a set of ground IDB atoms over node v_0, and $0 \leq k \leq n$ an integer. Then, the pseudocode of Figure 6 defines the function*

$$\mathit{fixp}(v_0, X_0, k) = \mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_0} \cup X_0)/_{v_k}$$

which satisfies the two equations

$$\begin{aligned} \mathcal{T_P}^{\omega}(\boldsymbol{T})/_{\mathit{root}} &= \mathit{fixp}(\mathit{root}, \emptyset, 0) \\ \mathcal{T_P}^{\omega}(\boldsymbol{T})/_{v_k} &= \mathit{fixp}(v, \mathcal{T_P}^{\omega}(\boldsymbol{T})/_{v}, k). \end{aligned}$$

Proof of Lemma 9: By induction on the tree (bottom-up):

- (Induction start) Let v_0 be a leaf. $\boldsymbol{T}_{v_0}$ is a tree with only a single node, so $\boldsymbol{T}_{v_0}$ interpreted as a set of ground atoms only consists of *unary* atoms over v_0, and the same is true for $X_0 \cup \boldsymbol{T}_{v_0}$. $\mathit{fixp}(v_0, X_0, k=0)$ (k must be 0 because there are no children.) initially sets X to $X_0 \cup \boldsymbol{T}_{v_0}$ and then iterates $X := \mathcal{T_P}(X)$. Note that since X at all times only contains unary atoms, only rules of types 1 and 4 can fire. When a fixpoint is reached (which must be reached eventually), $X/_{v_0}$ is returned. Thus, $\mathit{fixp}(v_0, X_0, 0) = \mathcal{T_P}^{\omega}(X_0 \cup \boldsymbol{T}_{v_0})/v_0$, as claimed.
- (Induction step) Let v_0 be a non-leaf node. We rewrite the pseudocode of Figure 6 by replacing the recursive calls to *fixp* using the induction hypothesis. By our induction hypothesis, the pseudocode

```
X := X0 ∪ T/{v0,v1,...,vn};
while no fixpoint reached do
  X := T_P(X) ∪
       T_P^ω(T_v1 ∪ X/v1)/v1 ∪ ··· ∪
       T_P^ω(T_vn ∪ X/vn)/vn;
return X/vk;
```

obtained must equally compute $\mathit{fixp}(v_0, X_0, k)$.

Clearly, the while-loop converges to a fixpoint, with $\mathcal{T_P}$ and set unions monotonically increasing and the set of ground atoms finite.

By the first part of Lemma 8, it is safe to decompose the computation of $\mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_0} \cup X_0)/_{v_k}$ into the local processing of the region of $\boldsymbol{T}$ consisting of v_0 and its children $v_1 \dots v_n$ and the separate processing of their subtrees $\boldsymbol{T}_{v_1} \dots \boldsymbol{T}_{v_n}$ using the *fixp* function.

It is essential to observe that X at all times only consists of the part of the tree structure that only involves nodes $v_0, v_1, \dots, v_n$ as well as unary atoms over the same nodes. Because of the limited syntax of TMNF, rules of types 2 and 3 involving binary predicates can only (and only have to) unify with neighbouring nodes in the tree – either parents and their children or adjacent siblings. Therefore, all rules that can fire on $\boldsymbol{T}_{v_0}$ involving v_0 and its children will already fire on $\boldsymbol{T}/_{v_0,v_1,\dots,v_n}$ given that the iterative computation of $\mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_i} \cup X)/_{v_i}$ on the children ($1 \leq i \leq n$) contributes the atoms over the child nodes v_i that can be computed in the subtrees $\boldsymbol{T}_{v_i}$, possibly given atoms over v_i computed locally on $\boldsymbol{T}/_{v_0,v_1,\dots,v_n}$.

Whenever $\mathcal{T_P}(X)$ adds a new ground atom over a child node v_i ($i \geq 1$) to X, $\mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_i} \cup X/_{v_i})/_{v_i}$ is re-computed immediately afterwards to make sure that all consequences over node v_i from that atom that may be due to inferences in $\boldsymbol{T}_{v_i}$ are added to X. Therefore, the while-loop only terminates when $X = \mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_0} \cup X_0)/_{v_0,v_1,\dots,v_n}$. Our claim follows.

Thus, Figure 6 indeed defines the function

$$\mathrm{fixp}(v_0, X_0, k) = \mathcal{T_P}^{\omega}(\boldsymbol{T}_{v_0} \cup X_0)/_{v_k}.$$

Since $\boldsymbol{T}_{\mathit{root}} = \boldsymbol{T}$, $\mathcal{T_P}^{\omega}(\boldsymbol{T})/_{\mathit{root}} = \mathit{fixp}(\mathit{root}, \emptyset, 0)$. Moreover, by Lemma 8, $\mathcal{T_P}^{\omega}(\boldsymbol{T})/_{v_k} = \mathit{fixp}(v, \mathcal{T_P}^{\omega}(\boldsymbol{T})/_{v}, k)$ (where v_k is the k-th child of v). □

This provides us with an algorithm to compute $\mathcal{T_P}^{\omega}(\boldsymbol{T})/_{v}$ for an arbitrary v given its edge-path $\Pi(v) = k_1 \cdots k_m$. It simply starts from $\mathit{fixp}(\mathit{root}, \emptyset, 0)$ and iteratively calls the *fixp* function for each of the segments k_i of $\Pi(v)$, always using the result of the previous call as the second and k_i as the third argument.

Note that differently from Figure 6, the second argument as well as the return value in the implementation of Figure 7 is of type "set of predicates" rather than "set of atoms". Moreover, we make the recursive calls to *fixp* more intelligently, that is, only when new atoms have been derived at a child node and there is a chance of *fixp* inferring something new.

```
function fixp(node v, set of predicates F, integer k ≥ 0)
    returns set of predicates
begin
    F_0 := F ∪ {P | P(x) ← U(x). in 𝒫, U holds at v};
    let n be the number of children of v;
    for each child v_i of v (1 ≤ i ≤ n) do
        F_i := fixp(v_i, ∅, 0);

    while no fixpoint of F_0 reached do
    begin
        if P(x) ← P_1(x) ∧ P_2(x). in 𝒫
          and P_1, P_2 ∈ F_0 then
              F_0 := F_0 ∪ {P};

        if P(x) ← P_0(x_0) ∧ First-Child(x_0, x). in 𝒫,
          n ≥ 1 and P_0 ∈ F_0 then
              F_1 := fixp(v_1, F_1 ∪ {P}, 0);

        if P(x) ← P_0(x_0) ∧ First-Child(x, x_0). in 𝒫,
          n ≥ 1 and P_0 ∈ F_1 then
              F_0 := F_0 ∪ {P};

        if P(x) ← P_0(x_0) ∧ Next-Sibling(x_0, x). in 𝒫,
          P_0 ∈ F_i and 1 ≤ i < n then
              F_{i+1} := fixp(v_{i+1}, F_{i+1} ∪ {P}, 0);

        if P(x) ← P_0(x_0) ∧ Next-Sibling(x, x_0). in 𝒫,
          P_0 ∈ F_i and 1 < i ≤ n then
              F_{i-1} := fixp(v_{i-1}, F_{i-1} ∪ {P}, 0);
    end;
    return F_k;
end.
```

Figure 7. A more detailed implementation of fixp.

Algorithm 4.1 *(Evaluation of monadic datalog)*
Input: A program $\mathcal{P}$, an instance $\boldsymbol{I}$, and a node $v \in \Pi(V^{\boldsymbol{I}})$ (that is, v is a node of $T(\boldsymbol{I})$ expressed as an edge-path).
Output: The set of IDB predicates for v that can be derived using $\mathcal{P}$ on $T(\boldsymbol{I})$.
Method:

```
Let k_1 ... k_m be the edge-path defining v (m ≥ 0);
w := root^I;    F := fixp(w, ∅, 0);    i := 1;
while i ≤ m do
begin
    F := fixp(w, F, k_i).
    w := the k_i-th child of w;
    i := i + 1;
end;
Output F.
```

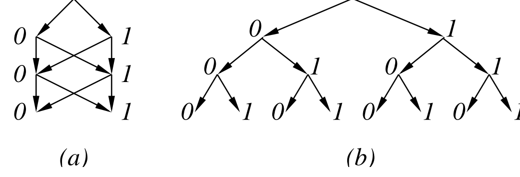

Figure 8. Instance for QSAT encoding (three variables).

where the auxiliary function

$$\text{fixp} : V^{\boldsymbol{I}} \times 2^{IDB(\mathcal{P})} \times \{0, 1, 2, 3, \dots\} \to 2^{IDB(\mathcal{P})}$$

is given in Figure 7. □

Theorem 10. *Given a TMNF program $\mathcal{P}$, an instance $\boldsymbol{I}$, and a node $v \in \boldsymbol{T}(\boldsymbol{I})$, the set $\{P(v) \in \mathcal{T}_{\mathcal{P}}{}^{\omega}(\boldsymbol{T}(\boldsymbol{I}))\}$ can be computed in space $O(||\boldsymbol{I}|| * |IDB(\mathcal{P})|)$.*

Proof of Theorem 10 (Sketch): The space requirements of the above-described algorithm are dominated by those of the fixp function. This function in turn only considers very localized regions of the instance at any point in time, namely a node v plus its children. It is recursive, and therefore $depth(\boldsymbol{T}(\boldsymbol{I}))$ fixp activation records may have to be kept on the stack. The total space required may at most amount to the *number of nodes of a path from the root to a leaf plus all of their siblings* (times space to store a set of predicates of size at most $|IDB(\mathcal{P})|$). Such a fragment of an instance has the same size no matter whether compressed or not. Thus, we obtain the space bound indicated in Theorem 10. Note that our algorithm proceeds in exactly the same way no matter whether the instance is a tree or compressed. □

Evaluating Core XPath (and thus TMNF) on compressed instances is also PSPACE-complete:

Theorem 11. *The evaluation problem for both monadic datalog and Core XPath over compressed instances is PSPACE-complete. Positive Core XPath over compressed instances is NP-complete.*

Proof of Theorem 11: PSPACE-membership for TMNF is asserted in Theorem 10, and Core XPath inherits this upper bound by Proposition 7. We show the PSPACE-hardness of both languages by providing a logspace-reduction from QSAT to the decision problem for Core XPath query evaluation.

Given a (closed) quantified boolean formula

$$Q_1 x_1 \cdots Q_n x_n \varphi$$

(where $Q_1, \dots, Q_n \in \{\forall, \exists\}$ and φ is quantifier-free), the instance is as shown in Figure 8 for $n = 3$ variables and has schema $\sigma = \{0, 1\}$.

The query is /self::*[Q_1 child[$\cdots$[Q_n child[φ']]$\cdots$]], where $\exists$ child[ψ] rewrites into child::*[ψ] and $\forall$ child[ψ] rewrites into not(child::*[not(ψ)]). φ' is obtained from the boolean formula φ by substituting $\wedge$, $\vee$, and $\neg$ by "and", "or", and "not()", respectively, and each variable x_i by parent::*/$\cdots$/parent::*/self::1, where the number of applications of the parent axis is $n - i$.

It is not difficult to see that this is indeed a correct encoding of QSAT, i.e. the query selects the root node if and only if the QSAT formula is true.

For example, the query for the QSAT formula

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

(which is true) is

/self::*[not(child::*[not(child::*[φ'])])]

with

$$\varphi' \quad = \quad \text{(not(parent::*/self::1) or self::1) and} \\ \text{(parent::*/self::1 or not(self::1)).}$$

The instance serves as a compressed version of a proof tree for the QSAT formula, where a node label (either 0 for "false" or 1 for "true") at a node at depth $i + 1$ in the tree denotes a valuation of propositional variable x_i. In our example, which can also be written as $\forall x_1 \exists x_2 (x_1 \Leftrightarrow x_2)$, the query selects the root node because both the left child of the root node labeled 0 and the right child labeled 1 ($\forall x_1$) have a child ($\exists x_2$) with their respective label ($x_1 \Leftrightarrow x_2$).

The NP-hardness of positive Core XPath follows immediately from the encoding of the previous proof, in which we can still encode SAT (i.e., all quantifiers are existential) without negation. Negation inside φ can be pushed down to the variables using De Morgan's laws, where we can rewrite e.g. not(parent::*/self::1) as parent::*/self::0. For instance, we can write φ' equivalently as

(parent::*/self::0 or self::1) and (parent::*/self::1 or self::0).

Membership in NP follows from the following observation. Let Q be a query and v be the node for which membership in the query result is to be checked. If negation is not present in Q, it can be matched in the instance $\boldsymbol{I}$ iff it can be matched on a small subtree of $\boldsymbol{T}(\boldsymbol{I})$ containing v. This "witness" tree is of polynomial size: it is at most as deep as $\boldsymbol{I}$ and is defined by a subset of $\Pi(V^{\boldsymbol{I}})$ of cardinality at most the number of nodes of the query tree of Q (basically corresponding to the number of operations such as axis applications and node tests performed in Q).

Indeed, no more than a subtree of this size can matter in the evaluation of such a query. The remaining parts of tree $\boldsymbol{T}(\boldsymbol{I})$ must be ignored and "jumped over" by the axis applications.

All we now need to do is guess such a witness tree and execute Q on it (for which we have a polynomial-time algorithm [16]). $\square$

Finally, we look at the problem of finding a practical (time-efficient) algorithm that evaluates a TMNF program $\mathcal{P}$ on a compressed instance, i.e., which produces a bisimilar instance to the nodes of which the fixpoint of $\mathcal{P}$ has been attached (employing partial de-compression).

Analogously to Definition 3 (but now, a datalog program selects several sets of nodes, one for each IDB predicate), we assume that the evaluation problem for program $\mathcal{P}$ on instance $\boldsymbol{I}$ is to find a bisimilar instance $\boldsymbol{J}$ over schema $\sigma \cup IDB(\mathcal{P})$ such that $\boldsymbol{T}(\boldsymbol{J}) = \mathcal{T_P}^{\omega}(\boldsymbol{T}(\boldsymbol{I}))$ (where we slightly abuse notation).

Theorem 12. *Let $\mathcal{P}$ be a TMNF program and $\boldsymbol{I}$ an instance. Then, an instance $\boldsymbol{J}$ s.t. $\boldsymbol{T}(\boldsymbol{J}) = \mathcal{T_P}^{\omega}(\boldsymbol{T}(\boldsymbol{I}))$ and $|V^{\boldsymbol{J}}| \leq 2^{|IDB(\mathcal{P})|} * |V^{\boldsymbol{I}}|$ can be computed in time $O(|\mathcal{P}| * ||\boldsymbol{J}||)$.*

Proof of Theorem 12: We assume the following data structure for representing nodes of an instance $\boldsymbol{I}$. For practical reasons (nodes and thus atoms true on them may have to be copied), we store inferred ground atoms *with the nodes* rather than in the set Ω used in the algorithm of Figure 3. Each node $v \in V^{\boldsymbol{I}}$ has a set $Preds(v)$ of boolean flags and a list $children(v)$ of *references to* nodes from $V^{\boldsymbol{I}}$ associated with it. By $children(v)[i]$, we access the i-th child of node v.

We make a number of changes to the algorithm of Figure 3. One is that we populate the $Preds(v)$ sets rather than the set Ω: In the initialization phase, for each rule $P(x) \leftarrow U(x)$. of the first kind and each node v,

$$Preds(v) := \{P \mid U \text{ is true on } v\}$$

In the while-loop, given that we infer a new atom $P(v)$, we add predicate P to $Preds(v)$ rather than $P(v)$ to Ω. This is now done immediately when $P(v)$ is inferred and put on the queue W, rather than later, when it is taken off W. (This is necessary because depending on the way $P(v)$ is inferred, v may have to be split into two nodes).

In the following, we say that two nodes v and w are equivalent if $Preds(v) = Preds(w)$ and $children(v) = children(w)$. (That is, $children(v)$ and $children(w)$ have the same number of elements n and $children(v)[i]$ and $children(w)[i]$ denote the same nodes – rather than just equivalent nodes – for all $1 \leq i \leq n$.)

We define a function make(v, P) which

- copies node v to a new node v' (including the list children(v)),
- adds predicate P to $Preds(v')$, and

– returns v' if no node w equivalent to v' exists in the instance[3] (otherwise, it deletes v' again and returns w). Let u be the node returned. Then, "make" also puts $P(u)$ onto W.

  If a copy v' is returned, the v entries in the data structure Aux_4 (to find those in constant time, an additional data structure is needed) are copied as well.

For rules of types 2 and 3 in the first for-loop inside the while-loop, we distinguish four cases.

case $P(x) \leftarrow P_0(x_0) \wedge \textit{First-Child}(x_0, x).$:
  if $P_0 \in \textit{Preds}(v)$ and v is not a leaf **then**
    children(v)[1] := make(children(v)[1], P);

case $P(x) \leftarrow P_0(x_0) \wedge \textit{First-Child}(x, x_0).$:
  if $P_0 \in \textit{Preds}$(children(v)[1]) **then**
  {
    $\textit{Preds}(v) := \textit{Preds}(v) \cup \{P\}$;
    add $P(v)$ to W;
  }

case $P(x) \leftarrow P_0(x_0) \wedge \textit{Next-Sibling}(x_0, x).$:
  if $P_0 \in \textit{Preds}$(children(v)[i]) and $i <$ |children(v)| **then**
    children(v)[$i+1$] := make(children(v)[$i+1$], P);

case $P(x) \leftarrow P_0(x_0) \wedge \textit{Next-Sibling}(x, x_0).$:
  if $P_0 \in \textit{Preds}$(children(v)[i]) and $i > 1$ **then**
    children(v)[$i-1$] := make(children(v)[$i-1$], P);

The second for-loop handling rules of type 4 remains essentially as in Figure 3.

Given appropriate data structures as we have assumed them earlier, this modified algorithm runs in time $O(|\mathcal{P}| * ||\boldsymbol{J}||)$, for the same reasons why the algorithm of Figure 3 runs in time linear in the size of the output times the size of the program. Note that the instance can only grow by splitting nodes and will never shrink; however, since we always check whether a split is necessary or whether an existing node can be re-used, the instance de-compresses to at most the size $2^{|IDB(\mathcal{P})|} * ||\boldsymbol{I}||$.

Note that in this simple form, the algorithm may leave some nodes in the instance that are unreachable from the root, and which need to be garbage-collected in the end. These "lost nodes" do not invalidate our claim about the running time $O(|\mathcal{P}| * ||\boldsymbol{J}||)$, as they are accounted for by the factor $|\mathcal{P}|$. □

This result is based on a variation of Minoux' algorithm [22] (for evaluating propositional logic programs) in which we lazily – only when needed – compute propositional rules (by instantiating the rules of $\mathcal{P}$ using the instance). The simple syntax of TMNF greatly facilitates this lazy grounding of the program.

[3]Here, for practical purposes, we assume a hash table of existing nodes in which we can find nodes in *constant* time.

5. Binary Structures

The concise representation of multiple consecutive edges (just storing one edge together with a multiplicity) causes a number of problems. One way to get around these is to transform arbitrary instances into binary instances first and then only to work with these binary instances. In a binary instance we can store all edges explicitly, so there is no need for the multiplicities. For each instance $\boldsymbol{I}$ we define a binary instance $\boldsymbol{B}(\boldsymbol{I})$ as follows: We first replace a vertex with i children by an almost complete binary tree of height $2^{\lceil \log(i) \rceil}$, as indicated in Figure 9. The binary instance $\boldsymbol{B}(\boldsymbol{I})$ has an additional unary relation that contains all vertices of the original instance $\boldsymbol{I}$. All other unary relations of $\boldsymbol{I}$ can be directly transferred to $\boldsymbol{B}(\boldsymbol{I})$.

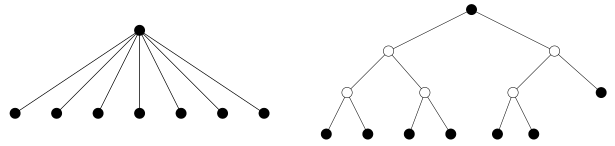

Figure 9.

We omit a formal definition of $\boldsymbol{B}(\boldsymbol{I})$. The following proposition is crucial. Its proof is straightforward; Figure 10 illustrates why there will be a blow-up in size logarithmic in the maximum edge multiplicity.

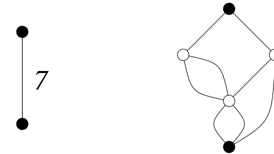

Figure 10.

Proposition 13. *There is an algorithm that, given an instance $\boldsymbol{I}$, computes a binary instance $\boldsymbol{B}$ equivalent to $\boldsymbol{B}(\boldsymbol{I})$ in time $O\big(||\boldsymbol{I}|| \cdot \log(\text{mult}(\boldsymbol{I}))\big)$.*

Proof of Proposition 13: Let $\boldsymbol{I}$ be an instance and $v \in V^{\boldsymbol{I}}$ with

$$\gamma(v) = w_1^{m_1} w_2^{m_2} \dots w_k^{m_k},$$

where $w_1, \dots, w_k \in V^{\boldsymbol{I}}$ such that $w_i \neq w_{i+1}$ for $1 \leq i \leq k$. Then we have $m_i \leq \text{mult}(\boldsymbol{I})$ for $1 \leq i \leq k$. Let $m = \sum_{i=1}^{k} k_i$ and $h = \lceil \log(m) \rceil$.

In the transformation from $\boldsymbol{T}(\boldsymbol{I})$ to $\boldsymbol{B}(\boldsymbol{T}(\boldsymbol{I}))$, vertex v and its m children are replaced by a subtree $\boldsymbol{S}$ of height h with

m leaves. We claim that in the compressed binary instance $\boldsymbol{M}(\boldsymbol{B}(\boldsymbol{I})) = \boldsymbol{M}(\boldsymbol{B}(\boldsymbol{T}(\boldsymbol{I})))$, this subtree $\boldsymbol{S}$ is compressed to a subinstance $\boldsymbol{S}'$ with $O(k \cdot \log(\text{mult}(\boldsymbol{I})))$ vertices.

To prove this claim, we assume without loss of generality that $m = 2^h$. If this is not the case, we can simply add a dummy vertex w_{k+1} and the let $\gamma(v) = w_1^{m_1} \dots w_k^{m_k} w_{k+1}^{2^h - m}$. Then $\boldsymbol{S}$ is a complete binary tree of height h. The *ith level* of $\boldsymbol{S}$ and $\boldsymbol{S}'$ consists of all vertices whose distance from v (the root of $\boldsymbol{S}$) is i.

We observe the following:

(1) On each level of $\boldsymbol{S}$ there are at most $2k-1$ bisimilarity classes of vertices (i.e., at most $2k-1$ vertices that are pairwise not bisimilar). Thus each level of $\boldsymbol{S}'$ contains at most $2k-1$ vertices.

(2) For $0 \leq \ell \leq h$, the first ℓ levels of $\boldsymbol{S}$ contain $2^{\ell+1} - 1$ vertices alltogether. Thus the first ℓ levels of $\boldsymbol{S}'$ contain at most $2^{\ell+1} - 1$ vertices.

Now let $\ell = \lceil \log(k) \rceil$. Then $\boldsymbol{S}'$ contains at most

$$\begin{aligned} & (2^{\ell+1} - 1) + (h - \ell)(2k - 1) \\ \leq\ & (8k - 1) + (\log(m) - \log(k))(2k - 1) \\ =\ & (8k - 1) + \log\left(\frac{m}{k}\right)(2k - 1) \\ =\ & (8k - 1) + (\log(\text{mult}(\boldsymbol{I}))(2k - 1)) \\ =\ & O(k \cdot \log(\text{mult}(\boldsymbol{I}))). \end{aligned}$$

This completes the proof of the claim.

Noting that, given v and $\gamma(v)$, it is easy to compute $\boldsymbol{S}'$ in time linear in the size of $\boldsymbol{S}'$, the statement of the proposition follows. ☐

Remark 14. Note that if we used a logarithmic cost model, then the logarithmic factor in Proposition 13 could be avoided, because storing m parallel edges would then require space and time $\Theta(\log(m))$.

While the binary instance computed by the algorithm in Proposition 13 may be larger by a factor of $\log(\text{mult}(\boldsymbol{I}))$ than the original instance, it is not clear that it will be larger in practice. Indeed, after being minimised it may be exponentially smaller than the original instance, even if that was also minimal. The following example illustrates this.

Example 15. Let $\sigma = \{P, Q\}$. For $\ell \geq 1$, let $\boldsymbol{I}_\ell$ be the following instance: $\boldsymbol{I}$ has 3 vertices r, p, q. r is the root and p, q are leaves. γ is defined by $\gamma(r) = (pq)^{2^\ell}$, $\gamma(p) = \gamma(q) = \emptyset$. Furthermore $P = \{p\}$ and $Q = \{q\}$. The right hand side of Figure 11 shows $\boldsymbol{I}_3$.

It is easy to see that $\boldsymbol{I}_\ell$ is minimal and that $||\boldsymbol{I}_\ell|| \in O(2^\ell)$. However, as Figure 11 illustrates, the minimal instance $\boldsymbol{M}(\boldsymbol{B}(\boldsymbol{I}_\ell))$ bisimilar to the binary instance $\boldsymbol{B}(\boldsymbol{I}_\ell)$ has size $O(\ell)$.

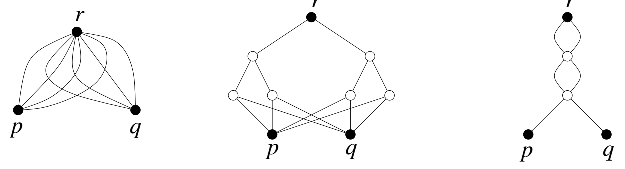

Figure 11. The instance $\boldsymbol{I}_3$ of Example 15, the binary instance $\boldsymbol{B}(\boldsymbol{I}_3)$, and its minimisation $\boldsymbol{M}(\boldsymbol{B}(\boldsymbol{I}_3))$.

Translating queries in TMNF or MSO into queries over the corresponding binary instances is easy, so it may be well worth working with binary instances only.

Let $\boldsymbol{B}^{-1}$ be the transformation that intuitively reverses the translation from unranked into binary instances and identifies each (non-auxiliary) node of $\boldsymbol{B}(\boldsymbol{T})$ with the node of $\boldsymbol{T}$ that it originates from. We assume that the schema of $\boldsymbol{B}(\boldsymbol{T})$ contains a predicate $V^{\boldsymbol{T}}$ that selects the nodes corresponding to the orginal nodes of $\boldsymbol{T}$, but not the auxiliary nodes introduced to obtain a binary tree.

Lemma 16. *Let $\mathcal{P}$ be a TMNF program. Then there is a TMNF program $\mathcal{P}'$ such that for all tree-instances $\boldsymbol{T}$,*

$$\mathcal{T_P}^\omega(\boldsymbol{T}) = \boldsymbol{B}^{-1}(\{P(v) \in \mathcal{T}^\omega_{\mathcal{P}'}(\boldsymbol{B}(\boldsymbol{T})) \mid P \in \mathit{IDB}(\mathcal{P})\}).$$

$\mathcal{P}'$ can be computed simultaneously in logarithmic space and linear time.

Proof of Lemma 16: We have to translate all occurrences of the predicates encoding the tree structure (i.e., the unary predicates *Root*, *Leaf* and *Last-Sibling* and the binary predicates *First-Child* and *Next-Sibling*) to obtain the correct behaviour over binary instances.

The *Root* and *Leaf* predicates remain the same on binary instances. The binary-instance version of *Last-Sibling* (called *Bin-Last-Sibling* below) can be defined by the following fixed program:

$$\begin{aligned} \mathit{Aux}_0(x) &\leftarrow V^{\boldsymbol{T}}(x). \\ \mathit{Aux}_1(x) &\leftarrow \mathit{Aux}_0(x_0) \wedge \\ & \quad\ \mathit{Second\text{-}Child}(x_0, x). \\ \mathit{Aux}_2(x) &\leftarrow \mathit{Aux}_0(x_0) \wedge \\ & \quad\ \mathit{Last\text{-}Sibling}(x_0). \\ \mathit{Aux}_1(x) &\leftarrow \mathit{Aux}_2(x_0) \wedge \\ & \quad\ \mathit{First\text{-}Child}(x_0, x). \\ \mathit{Aux}_0(x) &\leftarrow \mathit{Aux}_1(x) \wedge \overline{V^{\boldsymbol{T}}}(x). \\ \mathit{Bin\text{-}Last\text{-}Sibling}(x) &\leftarrow \mathit{Aux}_1(x) \wedge V^{\boldsymbol{T}}(x). \\ \mathit{Bin\text{-}Last\text{-}Sibling}(x) &\leftarrow \mathit{Root}(x). \end{aligned}$$

The idea of this encoding is to start at a parent node and to find its last child by following the rightmost edge-path down, moving to the second child whenever one exists and only otherwise moving to the first child.

As all IDB predicates have to be unary and we thus cannot define a binary relation as a predicate, we have to rewrite rules containing binary predicates. There are four cases; we discuss two (the "forward" cases using TMNF rules of the second kind); the remaining two can be encoded similarly.

Consider the rule

$$P(x) \leftarrow P_0(x_0) \wedge \textit{First-Child}(x_0, x).$$

In the binary version we essentially have to say that x is the first *First-Child*-descendent of x_0 such that $V^{\boldsymbol{T}}(x)$ holds. Therefore, we replace this rule by the following. Note that for every rule we have to introduce a new predicate *Aux*-P_0.

$$\begin{array}{rcl} \textit{Aux-}P_0(x) & \leftarrow & P_0(x_0) \wedge \textit{First-Child}(x_0, x). \\ \textit{Aux-}P_0(x) & \leftarrow & \textit{Aux-}P_0(x_0) \wedge \overline{V^{\boldsymbol{T}}}(x_0) \wedge \\ & & \textit{First-Child}(x_0, x). \\ P(x) & \leftarrow & \textit{Aux-}P_0(x) \wedge V^{\boldsymbol{T}}(x). \end{array}$$

Accordingly, we proceed for a rule

$$P(x) \leftarrow P_0(x_0) \wedge \textit{Next-Sibling}(x_0, x).$$

where we essentially check if x_0 is a *Last-Sibling*, and if so, we move to its parent if that parent is not in $V^{\boldsymbol{T}}$. Once we have found a non-*Last-Sibling* node, we go to its next sibling, and from there follow the *First-Child* path down. We select the first node in $V^{\boldsymbol{T}}$ we encounter. In particular, we replace the above rule by the following program fragment:

$$\begin{array}{rcl} \textit{Aux}_1\textit{-}P_0(x) & \leftarrow & P_0(x_0). \\ \textit{Aux}_2\textit{-}P_0(x) & \leftarrow & \textit{Aux}_1\textit{-}P_0(x) \wedge \textit{Last-Sibling}(x). \\ \textit{Aux}_3\textit{-}P_0(x) & \leftarrow & \textit{Aux}_2\textit{-}P_0(x_0) \wedge \textit{First-Child}(x, x_0). \\ \textit{Aux}_3\textit{-}P_0(x) & \leftarrow & \textit{Aux}_2\textit{-}P_0(x_0) \wedge \textit{Second-Child}(x, x_0). \\ \textit{Aux}_1\textit{-}P_0(x) & \leftarrow & \textit{Aux}_3\textit{-}P_0(x) \wedge \overline{V^{\boldsymbol{T}}}(x). \\ \textit{Aux}_4\textit{-}P_0(x) & \leftarrow & \textit{Aux}_1\textit{-}P_0(x_0) \wedge \textit{First-Child}(x, x_0). \\ \textit{Aux}_5\textit{-}P_0(x) & \leftarrow & \textit{Aux}_4\textit{-}P_0(x_0) \wedge \textit{Second-Child}(x_0, x). \\ \textit{Aux}_5\textit{-}P_0(x) & \leftarrow & \textit{Aux}_5\textit{-}P_0(x_0) \wedge \overline{V^{T}}(x_0) \wedge \\ & & \textit{First-Child}(x_0, x). \\ P(x) & \leftarrow & \textit{Aux}_5\textit{-}P_0(x) \wedge V^{\boldsymbol{T}}(x). \end{array}$$

It is easy to see that this transformation can be done in logarithmic space and linear time. □

Proposition 17. *Let $\boldsymbol{I}$ be an instance and Q a query in TMNF or MSO. Then, a query Q' in the same language can be computed in logarithmic space and linear time s.t. $Q(\boldsymbol{I}) = Q'(\boldsymbol{B}(\boldsymbol{I}))$.*

Proof of Proposition 17: For TMNF, the result follows immediately from Lemma 16, which makes an even stronger statement (namely that the translation correctly encodes all IDB predicates of $\mathcal{P}$, not just a single query predicate).

For MSO, we can directly take the encoding of a binary predicate B in TMNF and define a new binary relation φ_B in MSO with two free variables. In order to do that, we simply take the program $\mathcal{P}_r$ encoding rule r:

$$P(x) \leftarrow P_0(x_0) \wedge B(x_0, x).$$

and interpret $\leftarrow$ and $\wedge$ as implication and logical conjunction in MSO. Let $\bar{X} := \textit{IDB}(\mathcal{P}_r)$ and

$$\Psi_B(\bar{X}) := \bigwedge\{\forall \bar{x} : r \mid r \in \mathcal{P}_{P_0,B,P}, \\ \bar{x} \text{ consists of the variables in } r\}$$

Now,

$$\varphi_B(x, y) := \forall \bar{X}\big((x \in P_0 \wedge \Psi_B(\bar{X})) \rightarrow y \in P\big).$$

Given the TMNF program $\mathcal{P}_{\textit{Last-Sibling}}$ encoding *Last-Sibling* through the IDB predicate P, we proceed as above to obtain the FO-formula $\Psi_{\textit{Last-Sibling}}$. Let $\bar{X} := \textit{IDB}(\mathcal{P}_{\textit{Last-Sibling}})$. We define

$$\varphi_{\textit{Last-Sibling}}(x) := \forall \bar{X}\big(\Psi_{\textit{Last-Sibling}}(\bar{X}) \rightarrow x \in P\big).$$

By replacing all occurrences of *First-Child*, *Next-Sibling* and *Last-Sibling* using the unary and binary relations defined in this way, we obtain the desired translated MSO query. □

Core XPath is not expressive enough to accommodate such a translation (consider, for example, the very simple query /descendant::A/child::B), but this is not a problem as we translate all Core XPath queries into TMNF in our framework anyway.

Experimental Results on Binary Instances.

We have carried out a number of experiments on XML corpora, comparing the compression achieved using our notion of binary instances with instances compressed using multiple edges.

To assess the merits of our framework for binary structures, we have extended the XML compression module of the XPath query engine discussed in [6] to create binary structures in the way discussed in Section 5. We have run this compressor on a number of standard corpora (see [6]). The benchmark results can be found in Figure 12. This table is interpreted as follows.

- The first (left-most) column provides the name and size of the corpus.

- The second column states the number of nodes in the XML tree. (Subtract one to obtain the number of edges.)

	$\lvert V^T \rvert$	bisimilarity only $\lvert V^{M(T)} \rvert$	$\lvert E^{M(T)} \rvert$	bisimilarity & edge multiplicities $\lvert V^{M(T)} \rvert$	$\lvert E^{M(T)} \rvert$	bisimilarity on binary instances $\lvert V^{M(T)} \rvert$	$\lvert E^{M(T)} \rvert$	include XML tags?
SwissProt	10,903,569	83,427	1,731,391	83,427	792,620	352,185	683,153	no
(457.4 MB)		85,712	1,751,930	85,712	1,100,648	371,650	721,429	yes
DBLP	2,611,932	321	272,573	321	171,820	72,020	143,951	no
(103.6 MB)		4,481	379,524	4,481	222,755	114,000	227,470	yes
TreeBank	2,447,728	323,256	909,875	323,256	853,242	504,759	966,591	no
(55.8 MB)		475,366	1,315,645	475,366	1,301,690	783,177	1,505,197	yes
OMIM	206,454	962	25,173	962	11,921	8,430	16,744	no
(28.3 MB)		975	25,173	975	14,416	8,549	16,949	yes
XMark	190,488	3,642	28,901	3,642	11,837	9,292	17,238	no
(9.6 MB)		6,692	39,180	6,692	27,438	15,080	27,329	yes
Shakespeare	179,691	1,121	40,855	1,121	29,006	15,806	31,381	no
(7.9 MB)		1,534	48,385	1,534	31,910	18,022	35,688	yes
Baseball	28,307	26	665	26	76	104	199	no
(671.9 KB)		83	1,378	83	727	469	834	yes
TPC-D	11,765	15	357	15	161	196	378	no
(287.9 KB)		53	361	53	261	303	507	yes

Figure 12. Degree of compression of benchmarked corpora.

- The third and fourth columns provide the number of nodes and edges in the instance compressed using just bisimilarity, respectively.

  For each corpus, we present in two rows

  - in the upper row, the size of the compressed instance when XML node tags were ignored (i.e., only the bare tree structure is compressed, without any labeling information) and
  - in the lower row, the size of the compressed instance with all XML node tags inserted.
- Columns 5 and 6 report the size of the instance (nodes and edges, respectively) compressed using edge multiplicities.
- Finally, columns 7 and 8 state the size of the instance (nodes and edges, respectively) compressed as a binary instance.

Our experiences with the prototype implementation of [6] are that in order to achieve good query performance with unranked instances, in the data structures we come up with, the amount of memory required for a node is approximately the same as for an edge. Thus, to estimate the size of an unranked instance, one best adds up the node and the edge count of the instance.

We observe that binary instances compress quite well; the compression seems to compare the more favorably the larger the corpora get. Together with the fact that binary instances can be represented more efficiently in memory (the data structures required to represent nodes and edges have one degree of freedom less than unranked instances – each node only needs two fixed-size pointers to children, rather than an associated adjacency list), binary instances seem to be an interesting alternative to unranked instances compressed using multiple edges.

It is convenient to represent binary instances as relational structures in a slightly different way than arbitrary instances. We represent the edges of the DAG by two binary relations *First-Child* and *Second-Child*, representing the first-child and second-child relation.

In the following, we will give versions of our results for both unranked instances with edge multiplicities and binary instances.

6. Yet Another Query Automaton

The idea of querying XML-documents by tree automata is not new (e.g., [26, 18]). Some care needs to be taken, because in this context we are facing two problems usually not considered in classical language theory: Tree instances are not necessarily binary, but may be unranked, and queries select subsets of a tree and do not just accept or reject. Of course both of these problems can easily be resolved (and have been resolved in various ways, see e.g. [3, 12, 26]). We handle unranked trees simply by viewing them as binary trees under the *First-Child* and *Next-Sibling* relation. Our

way of handling unary queries is similar to the approach proposed by Neven and Schwentick [26] in their *query automata*: certain states are *selecting states* that select the vertices in the answer of the query. However, beyond this superficial similarity the two querying mechanisms are quite different.

A non-deterministic (bottom-up) tree automaton is a tuple $\mathfrak{A} = (Q, \Sigma, F, \delta)$, where Q is the *state space*, Σ is the *alphabet*, $F \subseteq Q$ is the set of *accepting states*, and

$$\delta : \Sigma \cup (Q \times \Sigma) \cup (Q \times Q \times \Sigma) \rightarrow 2^Q$$

the *transition function*. Tree automata work on binary trees in the usual way. Note that the transition function is defined in such a way that it accommodates leaves, nodes with one child, and nodes with two children. However, if a node has only one child, there is no way to distinguish between this child being a left or right child.

A *σ-tree automaton* is a nondeterministic tree-automaton $\mathfrak{A} = (Q, \Sigma, F, \delta)$ with $\Sigma = 2^\sigma$, that is, a tree-automaton that runs on tree instances of schema σ. If $\boldsymbol{T}$ is a σ-tree instance and $t \in V^{\boldsymbol{T}}$, we often write $\Sigma(t)$ to denote the "symbol" $\{R \in \sigma \mid t \in R^{\boldsymbol{T}}\} \in \Sigma = 2^\sigma$.

We can let the same tree automaton run on binary trees and on unranked trees, viewing the latter as binary trees with respect to the first-child and next-sibling relation. In the following, we treat binary trees in some details and then briefly explain how the approach extends to unranked trees.

6.1. Binary instances. A *run* of a σ-tree automaton $\mathfrak{A} = (Q, \Sigma, F, \delta)$ on a binary tree instance $\boldsymbol{T}$ of schema σ is a mapping $\rho : V^{\boldsymbol{T}} \rightarrow Q$ such that:

- For all leaves $t \in V^{\boldsymbol{T}}$ we have $\rho(t) \in \delta\big(\Sigma(t)\big)$.
- For all nodes $t \in V^{\boldsymbol{T}}$ with one child t_1 we have
$$\rho(t) \in \delta\big(\rho(t_1), \Sigma(t)\big).$$
- For all nodes $t \in V^{\boldsymbol{T}}$ with two children t_1, t_2 we have
$$\rho(t) \in \delta\big(\rho(t_1), \rho(t_2), \Sigma(t)\big).$$

The run ρ is *accepting* if $\rho(\mathit{root}^{\boldsymbol{T}}) \in F$. The automaton $\mathfrak{A}$ *accepts* $\boldsymbol{T}$ if there is an accepting run for $\mathfrak{A}$ on $\boldsymbol{T}$.

To be able to define unary queries, we need to enhance tree automata by an additional mechanism for selecting nodes.

Definition 18. A *selecting σ-tree automaton (σ-STA)* is a tuple $\mathfrak{A} = (Q, \Sigma, F, \delta, S)$, where (Q, Σ, F, δ) is a σ-tree automaton and $S \subseteq Q$ a set of *selecting states*.

The unary query defined by a σ-STA $\mathfrak{A}$ maps every σ-tree $\boldsymbol{T}$ to the set

$$\mathfrak{A}(\boldsymbol{T}) = \big\{v \in V^{\boldsymbol{T}} \mid \text{every accepting run of } \mathfrak{A} \text{ on } \boldsymbol{T} \text{ is in a selecting state at vertex } v\big\}.$$

At first sight, this querying mechanisms may seem a bit artificial. However, it turns out that STAs have good algorithmic properties and provide a nice unifying framework for the languages considered here. Still, requiring *all* accepting runs to be in a selecting state seems somewhat arbitrary. We shall see below that we may equivalently require *at least one* accepting run to be in a selecting state (cf. Corollary 21). Of course it would even be better if we could simply use *deterministic* tree automata.[4] The following example shows that this would not be sufficient.

Example 19. Let EVEN-DEPTH be the query of schema $\emptyset$ defined by

$$\text{EVEN-DEPTH}(\boldsymbol{T}) = \{v \in V^{\boldsymbol{T}} \mid \text{depth}(v) \text{ is even}\}$$

where the depth of a vertex in a tree is the length of the path from the root to this vertex.

EVEN-DEPTH is clearly definable by an STA. It is not definable by an deterministic STA, though. To see this, just note that because there is only one run of an STA on every tree, for a query defined by a deterministic STA it only depends on the subtree below a vertex whether the vertex belongs to the answer set or not.

Neven and Schwentick's [26] query automata are deterministic, but the price for this is that a run of a query automaton may go up and down the tree several times. As the following result implies, both types of automata have the same querying power.

Theorem 20 (see also [23]). *A query (on binary tree-instances) is definable in MSO if, and only if, it is definable by an STA.*

Proof of Theorem 20: For the forward direction (i.e., that for every unary MSO query there exists an equivalent STA), we use the well known fact that a class of binary trees is definable in MSO if and only if it is recognisable by a tree automaton [9, 28]. It implies that for every MSO-formula $\psi(X)$ of vocabulary $\sigma \cup \{\mathit{First\text{-}Child}, \mathit{Second\text{-}Child}\}$ for some schema σ there is a binary $\sigma \cup \{X\}$-tree automaton $\mathfrak{A}_\psi$ such that for all binary σ-tree instances $\boldsymbol{T}$ and for all subsets $U \subseteq V^{\boldsymbol{T}}$ we have

$$(\boldsymbol{T}, U) \models \psi(X) \iff \mathfrak{A}_\psi \text{ accepts } (\boldsymbol{T}, U).$$

Here $(\boldsymbol{T}, U)$ denotes the expansion of $\boldsymbol{T}$ to the $\sigma \cup \{X\}$-instance in which X is interpreted by U. Now let $\varphi(x)$ be an MSO-formula with one free variable x. Let $\psi(X) =$

[4] A tree automaton is *deterministic* if $\delta(\bar{q}, a)$ is a one-element set for all $(\bar{a}, q) \in \Sigma \cup (Q \times \Sigma) \cup (Q \times Q \times \Sigma)$. The usual powerset construction shows that for every (nondeterministic) tree automaton $\mathfrak{A}$ there is a deterministic tree automaton $\mathfrak{A}'$ equivalent to $\mathfrak{A}$, in the sense that $\mathfrak{A}$ and $\mathfrak{A}'$ accept the same trees.

$\forall x(\varphi(x) \to Xx)$. Then

$$\varphi(\boldsymbol{T}) = \bigcap\{U \subseteq V^{\boldsymbol{T}} \mid (\boldsymbol{T}, U) \models \psi(X)\}.$$

Suppose $\mathfrak{A}_\psi = (Q, 2^{\sigma \cup \{X\}}, F, \delta)$. Let $\mathfrak{A}$ be the selecting σ-tree automaton $(Q \times \{0,1\}, 2^\sigma, F \times \{0,1\}, \delta', Q \times \{1\})$, where δ' is defined by

$$\delta'((q_1, \varepsilon_1), (q_2, \varepsilon_2), a) = \\ \big(\delta(q_1, q_2, a) \times \{0\}\big) \cup \big(\delta(q_1, q_2, a \cup \{X\}) \times \{1\}\big)$$

for all $q_1, q_2 \in Q$, $\varepsilon_1, \varepsilon_2 \in \{0,1\}$, $a \in 2^\sigma$ (similarly for $\delta(a)$ and $\delta'(q,a)$). Then accepting runs of $\mathfrak{A}$ on a tree $\boldsymbol{T}$ correspond to accepting runs of $\mathfrak{A}_\psi$ on expansions $(\boldsymbol{T}, U)$ of $\boldsymbol{T}$. Therefore,

$$\mathfrak{A}(\boldsymbol{T}) = \bigcap\{U \subseteq V^{\boldsymbol{T}} \mid \mathfrak{A}_\psi \text{ accepts } (\boldsymbol{T}, U)\} = \varphi(\boldsymbol{T}).$$

For the converse direction, let $\mathfrak{A} = (Q, 2^\sigma, F, \delta, S)$ be a σ-STA with $Q = \{q_1, \dots, q_k\}$. A run ρ of $\mathfrak{A}$ on a binary σ-tree instance $\boldsymbol{T}$ can be described by a tuple $(U_1, \dots, U_k)$ of subsets of $V^{\boldsymbol{T}}$, where $v \in U_i$ if $\rho(v) = q_i$. It is easy to define a first-order formula $\psi(X_1, \dots, X_k)$ saying that $(X_1, \dots, X_k)$ describes an accepting run. Then

$$\varphi(x) = \forall X_1 \dots \forall X_k \big(\psi(X_1, \dots, X_k) \to \bigvee_{\substack{i \\ q_i \in S}} X_i x\big)$$

defines the same query as $\mathfrak{A}$. □

Corollary 21 (see also [23]). *For every STA $\mathfrak{A}$ there exists an STA $\mathfrak{A}'$ such that for all binary tree instances $\boldsymbol{T}$,*

$$\mathfrak{A}(\boldsymbol{T}) = \big\{v \in V^{\boldsymbol{T}} \mid \text{there exists an accepting run of } \mathfrak{A}' \text{ on } \boldsymbol{T} \text{ that is in a selecting state at vertex } v\big\}.$$

Proof of Corollary 21: Theorem 20 implies that the negation of a query defined by an STA can also be defined by an STA. Let $\mathfrak{B} = (Q, 2^\sigma, F, \delta, S)$ be an STA defining the negation of the query defined by $\mathfrak{A}$. Let $\mathfrak{A}' = (Q, 2^\sigma, F, \delta, Q \setminus S)$. It is easy to see that $\mathfrak{A}'$ has the desired property. □

Since monadic datalog is contained in MSO, Theorem 20 also implies that for every monadic datalog program there is an STA defining the same query. In the following example, we give a direct construction of such an automaton for a given datalog program, which of course is more efficient than the one going through MSO.

Example 22. Let σ be a schema and $X_1, \dots, X_\ell \notin \sigma$. Let $\mathcal{P}$ be a TMNF-program of schema σ. Let X_1 be the goal predicate. We define a σ-STA $\mathfrak{A} = (Q, 2^\sigma, F, \delta, S)$ as follows: We let $Q = 2^{\{X_1, \dots, X_\ell\}}$, $F = Q$, and $S = \{q \in Q \mid X_1 \in q\}$. To define the transition function, we first define a propositional Horn formula Ψ in the variables $\sigma \cup \{X_i, X_i^1, X_i^2 \mid 1 \le i \le \ell\}$ (we consider relation names as propositional variables here) with the following clauses:

- If $X_i(x) \leftarrow R(x)$ is a rule of $\mathcal{P}$ then $X_i \leftarrow R$ is a clause of Ψ.
- If $X_i(x) \leftarrow X_j(x) \wedge X_k(x)$ is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j \wedge X_k$ is a clause of Ψ.
- If $X_i(x) \leftarrow X_j(y) \wedge \textit{First-Child}(x, y)$ is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j^1$ is a clause of Ψ.
  If $X_i(x) \leftarrow X_j(y) \wedge \textit{Second-Child}(x, y)$ is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j^2$ is a clause of Ψ.
- If $X_i(x) \leftarrow X_j(y) \wedge \textit{First-Child}(y, x)$ is a rule of $\mathcal{P}$ then $X_i^1 \leftarrow X_j$ is a clause of Ψ.
  If $X_i(x) \leftarrow X_j(y) \wedge \textit{Second-Child}(y, x)$ is a rule of $\mathcal{P}$ then $X_i^2 \leftarrow X_j$ is a clause of Ψ.

Now for all $q_1, q_2 \in Q$ and $\tau \in 2^\sigma$ we let $\Psi(q_1, q_2, \tau)$ be the formula in the variables $\{X_1, \dots, X_\ell\}$ obtained from Ψ by replacing each variable X_j^i, for $i = 1, 2$ and $1 \le j \le \ell$, by TRUE if $X_j \in q_i$ and by FALSE otherwise, and by replacing each $R \in \sigma$ by TRUE if $R \in \tau$ and by FALSE otherwise. Then we let $\delta(q_1, q_2, \tau)$ be the set of all satisfying assignments of $\Psi(q_1, q_2, \tau)$. Here a satisfying assignment is identified with the set of variables it sets TRUE.

For $q_1 \in Q$ and $\tau \in 2^\sigma$ we define a formula $\Psi(q_1, \tau)$ as $\Psi(q_1, q_2, \tau)$ above, just replacing all variables X_j^2 by FALSE. Then we let $\delta(q_1, \tau)$ be the set of all satisfying assignments of $\Psi(q_1, \tau)$. For $\tau \in 2^\sigma$ we define $\delta(\tau)$ similarly.

This completes the definition of the automaton $\mathfrak{A}$. It is not hard to prove that $\mathfrak{A}$ defines the same query as the program $\mathcal{P}$.

We now show how to evaluate STA-queries, that is, queries defined by STAs, first on tree-instances and then on compressed instances. The algorithms combine ideas from [12] with the partial decompression methods that we also use to evaluate XPath and monadic datalog queries.

Proposition 23. *The evaluation problem for STA-queries on binary tree instances can be solved in time $O(s^3 \cdot n)$, where s is the number of states of the input automaton and n the number of vertices of the input instance.*

Proof of Proposition 23: Let $\mathfrak{A} = (Q, 2^\sigma, F, \delta, S)$ be an STA and $\boldsymbol{T}$ a binary σ-tree instance.

For every $t \in V^{\boldsymbol{T}}$, let *reach*(t) be the set of all states q such that there is a run ρ of $\mathfrak{A}$ on $\boldsymbol{T}$ with $\rho(t) = q$. The states in *reach* are called the *reachable states* at t. Clearly, the mapping *reach* $: V^{\boldsymbol{T}} \to 2^Q$ can be computed in time $O(s^3 \cdot n)$ in a bottom-up pass of the tree.

For every $t \in V^{\boldsymbol{T}}$, let *succ*(t) be the set of all states q such that there is an *accepting* run ρ of $\mathfrak{A}$ on $\boldsymbol{T}$ with $\rho(t) =$

q. The states in *succ* are called the *successful states* at t. Observe that

$$\mathit{succ}(\mathit{root}^{\boldsymbol{T}}) = \mathit{reach}(\mathit{root}^{\boldsymbol{T}}) \cap F,$$

and that for a siblings $t_1, t_2 \in V^{\boldsymbol{T}}$ with parent $t \in V^{\boldsymbol{T}}$,

$$\mathit{succ}(t_1) = \{q_1 \in \mathit{reach}(t_1) \mid \exists q_2 \in \mathit{reach}(t_2), q \in \mathit{succ}(t) : \; q \in \delta(q_1, q_2, \Sigma(t))\}.$$

$\mathit{succ}(t_2)$ can be obtained analogously. Similarly, $\mathit{succ}(t_1)$ can be obtained from $\mathit{succ}(t)$ for an only-child t_1. Using this, it is easy to see that given *reach*, the mapping $\mathit{succ} : V^{\boldsymbol{T}} \to 2^Q$ can be computed in time $O(s^3 \cdot n)$ in a top-down pass of the tree.

The vertices in $\mathfrak{A}(\boldsymbol{T})$ are precisely the $t \in V^{\boldsymbol{T}}$ for which $\mathit{succ}(t) \subseteq S$. □

Remark 24. Note that storing the transition relation of an automaton with s states requires space $\Omega(s^2)$ and $O(s^3)$. So the factor s^3 in the running time of the previous algorithm is not as bad as it looks. Indeed, a closer analysis shows that the running time can be improved to $O(m \cdot n)$, where m is the size of the encoding of the automaton.

In the remainder of this section, whenever we have an automaton $\mathfrak{A}$ run on a (compressed) instance $\boldsymbol{I}$, the actual meaning is that $\mathfrak{A}$ runs on the unfolded tree-instance $\boldsymbol{T}(\boldsymbol{I})$, even if our results will be based on techniques that usually do not require complete decompression to answer queries. Thus, we continue to address the problem of evaluating queries on (compressed) trees, rather than studying automata for directed acyclic graphs.

Theorem 25. *The evaluation problem for STA-queries on binary instances can be solved in time $O(2^{s+3\log s} \cdot n)$, where s is the number of states of the input automaton and n the number of vertices of the input instance.*

Proof of Theorem 25: We proceed as on tree instances: In a first bottom up pass we compute the sets of reachable states at all vertices, and in a second, top-down pass we compute the successful states. The problem is that a reachable state $q \in \mathit{reach}(v)$ may be successful on some path from v to the root and not successful on some other path. If this happens, we have to create a copy v' of vertex v and put q into $\mathit{succ}(v')$, but not into $\mathit{succ}(v)$ (or vice versa). In the worst case, we have to create a copy of every vertex for every subset of the state space, which explains why the running time is exponential in s.

Let us make this precise. Let $\mathfrak{A} = (Q, 2^\sigma, F, \delta, S)$ be a σ-STA and $\boldsymbol{I}$ a binary σ-instance. Let $\boldsymbol{T} = \boldsymbol{T}(\boldsymbol{I})$ be the tree instance bisimilar to $\boldsymbol{I}$. Recall that $\Pi : V^{\boldsymbol{I}} \to 2^{V^{\boldsymbol{T}(\boldsymbol{I})}}$ maps each vertex $v \in V^{\boldsymbol{I}}$ to the set of vertices of $\boldsymbol{T}$ it corresponds to.

For every $t \in V^{\boldsymbol{T}}$ we define the reachable states $\mathit{reach}(t)$ and successful states $\mathit{succ}(t)$ at t as in the proof of Proposition 23. Since $\mathit{reach}(t)$ only depends on the subtree below t, for all $v \in V^{\boldsymbol{I}}$, $t, t' \in \Pi(v)$ we have $\mathit{reach}(t) = \mathit{reach}(t')$. Let $\mathit{reach}(v) = \mathit{reach}(t)$. Observe that the mapping $\mathit{reach} : V^{\boldsymbol{I}} \to 2^Q$ can be computed in $O(s^3 \cdot n)$ in a bottom-up pass of the instance $\boldsymbol{I}$. (Compression does not matter here.)

However, there may be $v \in V^{\boldsymbol{I}}$, $t, t' \in \Pi(v)$ such that $\mathit{succ}(t) \neq \mathit{succ}(t')$. For $v \in V^{\boldsymbol{I}}$ we let

$$\mathcal{S}(v) = \{\mathit{succ}(t) \mid t \in \Pi(v)\}.$$

In the top-down pass over the instance $\boldsymbol{I}$ we compute a new instance $\boldsymbol{J}$ and mappings $f : V^{\boldsymbol{J}} \to V^{\boldsymbol{I}}$ and $\mathit{succ} : V^{\boldsymbol{J}} \to 2^\sigma$ such that:

(1) $\boldsymbol{J}$ is bisimilar to $\boldsymbol{I}$,

(2) For all $w \in V^{\boldsymbol{J}}$ we have $w \approx f(w)$ and thus $\Pi(w) = \Pi(f(w))$.

(3) For all $w \in V^{\boldsymbol{J}}$ and $t \in \Pi(w)$ we have $\mathit{succ}(t) = \mathit{succ}(w)$.

(4) For every $v \in V^{\boldsymbol{I}}$ and every $X \in \mathcal{S}(t)$ there exists exactly one vertex $w \in V^{\boldsymbol{J}}$ such that $f(w) = v$ and $\mathit{succ}(w) = X$.

We start by creating a vertex $\mathit{root}^{\boldsymbol{J}}$ and let $\mathit{succ}(\mathit{root}^{\boldsymbol{J}}) = \mathit{reach}(\mathit{root}^{\boldsymbol{I}}) \cap F$ and $f(\mathit{root}^{\boldsymbol{J}}) = \mathit{root}^{\boldsymbol{I}}$.

Now suppose that w is a vertex of w that we have already created, and let $v = f(w)$, and $X = \mathit{succ}(w)$. Furthermore, suppose that v has children v_1 an v_2 and that we have not yet created any children of w. Let

$$X_1 = \{q_1 \in \mathit{reach}(v_1) \mid \exists q_2 \in \mathit{reach}(v_2), q \in X : q \in \delta(q_1, q_2, \Sigma(v))\}.$$

Then for all $t \in \Pi(w)$ with children t_1 and t_2 we have $X_1 = \mathit{succ}(t_1)$. To see this, note that $\mathit{reach}(t_1) = \mathit{reach}(v_1)$, $\mathit{reach}(t_2) = \mathit{reach}(v_2)$, $\mathit{succ}(t) = X$, and recall the corresponding claim in the proof of Proposition 23.

If there already is a vertex $w_1 \in V^{\boldsymbol{J}}$ such that $f(w_1) = v_1$ and $\mathit{succ}(w_1) = X_1$, we make w_1 the first child of w. Otherwise, we create a new vertex w_1, let $f(w_1) = v_1$ and $\mathit{succ}(w_1) = X_1$, and make w_1 the first child of w. The second child of w is created analogously. If the vertex $v = f(w)$ only has one child, we proceed similarly.

With an appropriate dictionary data structure that stores, for each vertex $v \in V^I$, the set of all pairs w with $f(w) = v$ and allows, given X, to find the w with $\mathit{succ}(w) = X$ (if it exists) in time $O(s)$, the children of a vertex of w can be created in time $O(s^3)$. Since $\boldsymbol{J}$ has at most $2^s \cdot n$ vertices, the overall running time to compute $\boldsymbol{J}$ and *succ* is $O(s^3 \cdot 2^s \cdot n) = O(2^{s+3\log s} \cdot n)$.

The output of the algorithm is $\boldsymbol{J}$ together with the set of all vertices $v \in V^{\boldsymbol{J}}$ with $\mathit{succ}(v) \subseteq S$. □

Remark 26. It is not hard to see the decision version of the evaluation problem for STA-queries is in polynomial time, because for a given σ-STA we can easily define a $\sigma \cup \{X\}$-tree automaton $\mathfrak{A}'$ such that for every σ-tree instance $\boldsymbol{T}$ and every vertex $t \in V^{\boldsymbol{T}}$ we have $t \in \mathfrak{A}(\boldsymbol{T})$ if and only if $\mathfrak{A}'$ accepts $(\boldsymbol{T}, \{t\})$. The latter can be tested in time $O(s^3 \cdot n)$, because an automaton accepts a tree if and only if there is an accepting reachable state at the root.

Corollary 27. *The evaluation problem for MSO-queries on binary instances parameterized by the size of the input formula is fi xed-parameter tractable.*

Let $\exists$MSO be the set of MSO-formulas of the form $\exists X_1, \dots, X_k \varphi(X_1, \dots, X_k)$ where φ is first-order.

Theorem 28. *The evaluation problem for $\exists$MSO-queries on binary instances is NEXPTIME-complete. Evaluation of MSO queries can be done in EXPSPACE.*

Proof of Theorem 28 (Sketch): For the upper bounds recall that the tree-instance $\boldsymbol{T}(\boldsymbol{I})$ has size $\leq 2^{|V^{\boldsymbol{I}}|}$. To decide whether $\psi = \exists \bar{X} \varphi(\bar{X})$ holds, we guess k sets A_i and check whether $\boldsymbol{T}(\boldsymbol{I}) \models \varphi(\bar{A})$. This can be done in the straighforward way in time $O(||\boldsymbol{T}(\boldsymbol{I})||^{||\varphi||}) = O(2^{||\varphi|| \cdot |V^{\boldsymbol{I}}|})$.

In case of full MSO, we use the standard PSPACE algorithm on $\boldsymbol{T}(\boldsymbol{I})$ [29], which, in total, yields EXPSPACE as upper bound.

For the hardness, assume that M is a fixed nondeterministic Turing machine with an NEXPTIME-complete acceptance problem (such a machine exists). We assume that M has alphabet $\Delta = \{a_1, \dots, a_l\}$ and works on input $w \in \Delta^n$ in time $2^{n^k} - 1$, for some $k \geq 1$. Let $Q = \{q_1, \dots, q_m\}$ be the state space of M. Then configurations are words $v \in \Delta^* Q \Delta^+$ of length 2^{n^k} meaning that if $v_i \in Q$ then M is in state $q = v_i$ and currently reading the ith cell of its (input/work) tape (or equivalently, M reads v_{i+1}).

From input $w \in \Delta^n$ for M we construct a binary instance $\boldsymbol{I}$ with $|V^{\boldsymbol{I}}| = O(n^k)$ and an $\exists$MSO-formula φ with $||\varphi|| = n^{O(k)}$ such that M accepts w iff $\boldsymbol{T}(\boldsymbol{I}) \models \varphi$. Let $\boldsymbol{I}$ be the structure from Figure 8 with paths of length $2 \cdot n^k + 1$ and let $\boldsymbol{T}$ denote its tree version $\boldsymbol{T}(\boldsymbol{I})$. Note here that $\boldsymbol{T}$ has exactly $2^{2 \cdot n^k} = 2^{n^k} \cdot 2^{n^k}$ leaves.

Recall that elements of $\boldsymbol{T}$ are edge-paths. We call every node $t \in \boldsymbol{T}$, where t is a n^k-length edge-path, a *base*. There are 2^{n^k} bases and each subtree rooted at a base has 2^{n^k} leaves. Leaves and bases are each totally ordered by the lexicographic ordering of the respective edge-paths (the first digit is the most significant one). For convenience, we denote both of them by $\prec$. For $0 \leq i \leq 2^{n^k} - 1$, by $t(i)$ we denote the i-th base of $V^{\boldsymbol{T}}$. Analogously, let $t^s(i)$ be the i-th leave of the subtree rooted at base $s \in V^{\boldsymbol{T}}$.

There is an FO-formula $\psi(x, y)$ with $||\psi(x, y)|| = O(n^{2k})$ such that $\boldsymbol{T} \models \psi(t, t')$ iff t' is the direct successor of t (w.r.t. $\prec$). For that, we simply guess two paths from the root to t and t' and check, if the edge-paths linking them are direct successors. Analogously, we can define a formula $\psi(x_1, y_1, x_2, y_2)$ saying that x_1, x_2 are bases, y_1 and y_2 are in $\boldsymbol{T}_{x_1}$, $\boldsymbol{T}_{x_2}$ resp. and the path from x_2 to y_1 is the direct successor of the path from x_1 to y_2 (w.r.t. $\prec$).

Recall that T_s, for a $s \in V^{\boldsymbol{T}}$, denotes the subtree of $\boldsymbol{T}$ rooted at s. We encode a configuration $v \in \Delta^* Q \Delta^+$ (of size 2^{n^k}) *relative* to base s as the $(m+l)$-tuple $\bar{C} \subseteq V_s^{\boldsymbol{T}}$ such that the following conditions hold:

(1) if $t \in C_\nu$ for some i, then t is a leaf

(2) if $v_i \in Q$ then $t^s(i) \in C_\nu$ iff $v_i = q_\nu$, $(\nu = 1, \dots, m)$

(3) if $v_i \in \Delta$ then $t^s(i) \in C_\nu$ iff $v_i = a_{\nu - m}$, $(\nu = m + 1, \dots, m + l)$

From now on we do not distinguish explicitly between configurations and their encodings. It is straightforward to define FO-formulas $\varphi_{start}(\bar{X}, x)$ and $\varphi_{acc}(\bar{X}, x)$, which hold, if x is a base and $\bar{X}$ relative to x is the *start* configuration or a *accepting* configuration, respectively. The intention behind these definition is to guess a tuple $\bar{C}$ over the leaves of $\boldsymbol{T}$ and to check, whether the configurations relative to the bases in $\boldsymbol{T}$ form an accepting run. Here it is crucial to observe that $\psi(x_1, y_1, x_2, y_2)$ allows us to identify consecutive positions (y_1, y_2) in two different configurations (relative to x_1, x_2).

Next, we define $\varphi_{next}(\bar{X}, x, y)$, which essentially says that x and y are bases and direct $\prec$-successors, and that $\bar{X}$ relative to x and $\bar{X}$ relative to y are configurations. We let denote these configurations by C_x and C_y. Now $\varphi_{next}(\bar{X}, x, y)$ says that C_x "equals" C_y, or that we can pass from C_x to C_y in a single step. In particular, this means that there are $i \leq n^k$, $\nu \leq m$ such that $t^x(i) \in X_\nu$, i.e. in configuration C_x we have the situation that M is state q_ν and reads letter a_j (j such that $t^x(i+1) \in X_{m+j}$). Since, using ψ, we are able to directly adress successors of $t^x(i)$ and $t^y(i)$, it is easy to devise a formula $\varphi'(\bar{X}, x, y)$ saying that the neighborhoods around $t^x(i)$ and $t^y(i)$ conform to the transition relation of M.

The sought formula now looks as follows. Let

$$\varphi = \exists \bar{X} \big(\varphi_{start}(\bar{X}, 0) \wedge \varphi_{acc}(\bar{X}, 1) \wedge \\ \forall \text{bases } x, y \; (\psi(x, y) \rightarrow \varphi_{next}(\bar{X}, x, y)) \big),$$

where we use 0 and 1 to denote the first and last base w.r.t. $\prec$, respectively. It is easy to see that φ behaves as it is supposed to. More precisely, it states that there is a sequence $v_1, \dots, v_{2^{n^k}}$ of configurations such that v_1, $v_{2^{n^k}}$ is a starting, accepting configuration, respectively. Furthermore, it states that for each $i < 2^{n^k}$ either v_{i+1} is equal to v_i or a direct successor of v_i (here w.r.t. the transition relation defined by M). Together, this means that M on input w has an accepting run.

The size of φ is bounded by $n^{O(k)}$, caused by the subformulas needed to identify $\prec$-successors of leaves and bases. The remaining subformulas solely depend on the (fixed) machine M. □

If we convert a monadic datalog program into an STA in the way described in Example 22 and then use Theorem 25 to evaluate the query, the resulting algorithm is doubly exponential in the size of the datalog program, which is much worse than the direct algorithms we saw before. We shall now introduce a restricted version of our STA, which is somewhat closer to monadic datalog and admits more efficient query evaluation.

Definition 29. A *weak selecting σ-tree automaton (σ-WSTA)* is a tuple $\mathfrak{A} = (Q, \Sigma, F, \delta, S, \preceq)$, where $(Q, \Sigma, F, \delta, S)$ is a σ-STA and $\preceq$ a partial order on Q s.t.

- F is downward closed w.r.t. $\preceq$.
- S is upward closed w.r.t. $\preceq$.
- For all $(\bar{q}, a) \in \Sigma \cup (Q \times \Sigma) \cup (Q \times Q \times \Sigma)$ and $r, r' \in \delta(\bar{q}, a)$ there is an $r'' \in \delta(\bar{q}, a)$ such that $r'' \preceq r, r'$.
- δ is monotone with respect to $\preceq$ in the sense that for all $(\bar{q}, a), (\bar{q}', a) \in (Q \times \Sigma) \cup (Q \times Q \times \Sigma)$, if $\bar{q} \preceq \bar{q}'$ (componentwise) then for every $r' \in \delta(\bar{q}', a)$ there is an $r \in \delta(\bar{q}, a)$ such that $r \preceq r'$.

Example 30. The automaton constructed from a monadic datalog program in Example 22 with set-inclusion as a partial order on the state space $2^{\{X_1,\ldots,X_n\}}$ is a WSTA. To see this, just note that the set of satisfying assignments of a propositional Horn formula is closed under intersection and monotone with respect to the Boolean constants in the formula.

Proposition 31. *For every STA there is a WSTA that defi nes the same query.*

Proof of Proposition 31: This follows from the fact that every MSO-query is definable by a monadic datalog program [14], Theorem 20, and the previous example. □

Note that if $\mathfrak{A} = (Q, \Sigma, F, \delta, S, \preceq)$ is a σ-WSTA and $\boldsymbol{T}$ a binary σ-tree instance, then $\preceq$ induces a partial order on the runs of $\mathfrak{A}$ on $\boldsymbol{T}$, which we also denote by $\preceq$, and which is defined by $\rho \preceq \rho'$ if $\rho(t) \preceq \rho'(t)$ for all $t \in V^{\boldsymbol{T}}$.

Lemma 32. *Let $\mathfrak{A} = (Q, \Sigma, F, \delta, S, \preceq)$ be a σ-WSTA. Then for every binary σ-tree instance $\boldsymbol{T}$ there is a unique minimal run $\rho_{\min}$ of $\mathfrak{A}$ on $\boldsymbol{T}$ with respect to the order $\preceq$ on the runs. Moreover, if $\mathfrak{A}$ accepts $\boldsymbol{T}$ then $\rho_{\min}$ is an accepting run and*

$$\mathfrak{A}(\boldsymbol{T}) = \{t \in V^{\boldsymbol{T}} \mid \rho_{\min}(t) \in S\}.$$

Proof: In a straightforward bottom-up fashion we can construct for any two runs ρ, ρ' of $\mathfrak{A}$ on $\boldsymbol{T}$ a run ρ'' such that $\rho'' \preceq \rho, \rho'$. Since there are only finitely many runs, this implies the existence of a unique minimal run.

If there is some accepting run ρ, then all runs $\rho' \preceq \rho$ are also accepting, because F is downward closed. Thus in particular, $\rho_{\min}$ is accepting. $\mathfrak{A}(\boldsymbol{T}) = \{t \in V^{\boldsymbol{T}} \mid \rho_{\min}(t) \in S\}$ follows from the fact that S is upward closed w.r.t. $\preceq$, which implies that for runs $\rho \preceq \rho'$, every vertex selected by ρ is also selected by ρ'. □

Theorem 33. *The evaluation problem for WSTA-queries on binary instances can be solved in time $O(s^3 \cdot n)$, where s is the number of states of the input automaton and n the number of vertices of the input instance.*

Proof of Theorem 33 (Sketch): We essentially proceed as in the proof of Theorem 25 using Lemma 32 to restrict attention to success states instead of success sets. Let $\mathfrak{A} = (Q, \Sigma, F, \delta, S, \preceq)$ be a σ-WSTA and $\boldsymbol{I}$ a binary σ-instance. Let $\boldsymbol{T} = \boldsymbol{T}(\boldsymbol{I})$ be the tree instance bisimilar to $\boldsymbol{I}$. For $t \in V^{\boldsymbol{T}}$ define $reach(t)$ and $succ(t)$ as before. Furthermore, let $succ'(t)$ be the minimal state of $succ(t)$ (w.r.t. $\preceq$) and let $\mathcal{S}'(v) = \{succ'(t) \mid t \in \Pi(v)\}$. We proceed as in the case of general STAs, but compute minimal states instead of sets of states. By the previous lemma this is sufficient to determine the sought result.

Again we start with a bottom-up pass computing $reach(t)$ for all $t \in V^{\boldsymbol{T}}$. Here note that $reach(t) = reach(t')$ for all $t, t' \in \Pi(v)$ for an arbitrary $v \in V^{\boldsymbol{I}}$. Then we go top-down and gradually compute a new instance $\boldsymbol{J}$ and mappings $f : V^{\boldsymbol{J}} \to V^{\boldsymbol{I}}$ and $succ' : V^{\boldsymbol{J}} \to Q$ that satisfy the conditions (1) to (4) with $succ$ and $\mathcal{S}$ replaced by $succ'$ and $\mathcal{S}'$, respectively. Note that, since F is downward closed, $\rho(t) := succ'(v)$ for $t \in \Pi(v)$ is the sought minimal accepting run ρ of $\mathfrak{A}$ on $\boldsymbol{T}$.

We start by creating a vertex $root^{\boldsymbol{J}}$ and let $succ'(root^{\boldsymbol{J}}) = reach(root^{\boldsymbol{I}}) \cap F$ and $f(root^{\boldsymbol{J}}) = root^{\boldsymbol{I}}$.

If w is a vertex of $V^{\boldsymbol{J}}$ that has already been created, let $v = f(w)$ and $q = succ'(w)$. Suppose that v has children v_1, v_2 and let $X_1 = \{q_1 \in reach(v_1) \mid \exists q_2 \in reach(v_2) \text{ s.t. } q \in \delta(q_1, q_2, \Sigma(v)\}$. Let q_1 be the minimum of X_1.

Now we check if there is a $w_1 \in V^{\boldsymbol{J}}$ with $f(w_1) = v_1$ and $succ'(w_1) = q_1$. If so, we make w_1 the first child of w. Otherwise, we create a new w_1 in $\boldsymbol{J}$ and define $f(w_1) = v_1$, $succ'(w_1) = q_1$ and make this w_1 the first child of w.

Now we see that for all $t \in \Pi(w)$ with children t_1, t_2 we have $q_1 = succ'(t_1)$. This shows that the procedure yields the intended result. Then we proceed analogously for the second child v_2.

But in contrast to the general case, now we create at most s new vertices for one vertex in $\boldsymbol{I}$, hence we have at most $s \cdot n$ vertices in $V^{\boldsymbol{J}}$.

An ad-hoc implementation yields an s^4 factor. This can

be improved as follows: at each node $v \in V^{\boldsymbol{I}}$ with children v_1, v_2 we compute the sets $pre_1(v,q) = \{q_1 \mid \exists q_2 \in reach(v_2), q \in \delta(q_1, q_2, \Sigma(v))\}$. Note that this set coincides with X_1, computed for $q = succ'(w)$ as defined above.

Analogously, we compute the sets $pre_2(v,q)$. Note that, using suitable data structures, this can be done in a single loop over all tuples $(q_1, q_2, \Sigma(v), q) \in \delta$, hence requires $O(s^3)$ steps for each $v \in V^{\boldsymbol{I}}$. Then we simply look up these tables to determine the values $succ'(t)$ (we find the minimum in $O(s)$ steps).

Together, this gives the $O(s^3 \cdot n)$ time bound. □

6.2. Unranked tree automata. Even though we can avoid unranked instances entirely if we transform them to binary instances as described in Section 5, we think it is still worthwhile to briefly discuss STAs on unranked instances. We have mentioned earlier that we run a tree automaton on an unranked tree by running it on the corresponding binary tree induced by the *First-Child* and *Next-Sibling* relation. Note that in this binary tree, the last siblings are precisely those that only have one (first) child.

To distinguish them clearly from the "binary" automata considered in the previous section, we call the automata considered here *unranked tree automata*. Let us emphasise, however, that the automata themselves are the same, they only run in a different way.

Once we have defined an appropriate notion of run and acceptance for unranked tree automata, we can define *unranked STAs* and *unranked WSTAs*. Then the basic results stated for binary automata in the previous section have analogous versions for unranked automata. In particular, every MSO-query on unranked trees (viewed as $\sigma \cup \{Root, Leaf, Last\text{-}Sibling, First\text{-}Child, Next\text{-}Sibling\}$-structures) is definable by an STA and vice versa. Moreover, monadic datalog programs over unranked trees can be translated into WSTAs whose state space is the set of all IDB predicates.

We only state the unranked version of the main result. The bounds on the running time are weaker, which is mainly due to the compact representation of multiple edges in compressed instances (cf. Section 2.2).

Theorem 34. *The evaluation problem for*

(1) STA-queries can be solved in time $O(2^{3s} + 2^s \cdot m)$.

(2) WSTA-queries can be solved in time $O(2^{3s} + s \cdot m)$.

Here s is the number of states of the input automaton and m the size of the input instance.

Proof of Theorem 34 (Sketch): Basically, we proceed as in the binary case. When computing the reachable and successful states, it is convenient to associate them with edges of the instance instead of vertices. For example, $reach(e)$ for an edge $e = (v, w)$ represents the set $reach(\Pi(w))$. The reason we have to do this is that vertices may play different roles as children of different parents — they may be the first child of some, but not for all parents. If we transfer the algorithms in a straightforward way, we obtain running times of $O(2^{s+3\log s} \cdot |E|)$ for STAs and $O(s^3 \cdot |E|)$ for WSTAs, where $|E|$ is the number of edges of the input instance. However, due to the compact representation of multiple edges, the size of the input instance may be much smaller than $|E|$.

Let $\mathfrak{A}$ be an STA and $\boldsymbol{I}$ an instance. Consider a node u of the input instance $\boldsymbol{I}$ which has outgoing edges $e_1, \dots, e_n$. Suppose $e_p, \dots, e_q$, for some $1 \le p < q < n$, have the same endpoint v, and e_{p-1} (if $p > 1$) and e_{q+1} have an endpoint distinct from v. Then in the adjacency list of u, edges $e_p, \dots, e_q$ are represented as a single edge $\boldsymbol{e}$ of multiplicity $r = (q - p + 1)$. Let w be the first child of v and f the edge from v to w. Let x be the endpoint of e_{q+1}, i.e., the next sibling of v. Suppose that during the computation of the reachable states we have already computed $R = reach(f)$ and $R' = reach(e_{q+1})$. To compute $reach(\boldsymbol{e})$, we not only have to simulate one step of the computation of the automaton $\mathfrak{A}$, but actually a sequence of r steps to compute $R_1 = reach(e_q), R_2 = reach(e_{q-1}), \dots, R_r = reach(e_p) = reach(\boldsymbol{e})$. The crucial observation is that since there are only 2^s distinct subsets of the state space, there are $i < j \le 2^{s+1}$ such that $R_i = R_j$. Of course this implies that for all $k \ge i$ we have $R_k = R_\ell$ for

$$\ell = \big((k - i) \bmod (j - i)\big) + i.$$

Thus once we have found i and j, we can stop the simulation and compute R_r by simple arithmetic.

Instead of running this simulation every time it is needed, we can pre-compute a table that contains, for all subsets R and R', the values i and j and the sets $R_1, \dots, R_j$. Such a table can be computed in time 2^{3s}. Of course the table then simplifies the computation of the mappings *reach* and *succ*, because essentially we are dealing with a deterministic automaton now. □

Acknowledgments

We thank an anonymous reviewer for pointing out to us that an unpublished part of Frank Neven's PhD thesis [23], pp. 128–130, defines the notion of *nondeterministic query automata* which precisely equals ours of STAs on (uncompressed) trees. Theorem 20 and Corollary 21 are proven there as well.

The third author's visit to LFCS, University of Edinburgh, was sponsored by Erwin Schrödinger Grant J2169 of the Austrian Research Fund (FWF).

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann Publishers, 2000.

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

[3] A. Brüggemann-Klein, M. Murata, and D. Wood. "Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001". Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science and Technology, Hong Kong SAR, China, 2001.

[4] A. Brüggemann-Klein and D. Wood. "Caterpillars: A Context Specification Technique". *Markup Languages*, **2**(1):81–106, 2000.

[5] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, **C-35**(8):677–691, Aug. 1986.

[6] P. Buneman, M. Grohe, and C. Koch. "Path Queries on Compressed XML", 2003. Submitted for publication.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, 1990.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[9] J. Doner. "Tree Acceptors and some of their Applications". *Journal of Computer and System Sciences*, **4**:406–451, 1970.

[10] R. Downey and M. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[11] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1999. Second edition.

[12] J. Flum, M. Frick, and M. Grohe. "Query Evaluation via Tree-Decompositions". In J. Van den Bussche and V. Vianu, editors, *Proc. of the 8th International Conference on Database Theory (ICDT'01)*, volume 1973 of *Lecture Notes in Computer Science*, pages 22–38, London, UK, Jan. 2001. Springer.

[13] M. Frick and M. Grohe. "The Complexity of First-order and Monadic Second-order Logic Revisited". In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 215–224, 2002.

[14] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages", Nov. 2002. Extended version of PODS'02 paper, submitted. Available as CoRR report arXiv:cs.DB/0211020.

[15] G. Gottlob and C. Koch. "Monadic Queries over Tree-Structured Data". In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 189–202, Copenhagen, Denmark, July 2002.

[16] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[17] G. Gottlob, C. Koch, and R. Pichler. "The Complexity of XPath Query Processing". In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Diego, California, USA, June 2003. To appear.

[18] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT'03)*, 2003.

[19] M. Grohe. "Parameterized Complexity for the Database Theorist". *SIGMOD Record*, **31**(4), 2002.

[20] H. Hosoya and B. C. Pierce. "Regular Expression Pattern Matching for XML". In *Proceedings of 28th Symposium on Principles of Programming Languages (POPL'01)*, pages 67–80. ACM Press, 2001.

[21] T. Milo, D. Suciu, and V. Vianu. "Typechecking for XML Transformers". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'00)*, pages 11–22, 2000.

[22] M. Minoux. "LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation". *Information Processing Letters*, **29**(1):1–12, 1988.

[23] F. Neven. *Design and Analysis of Query Languages for Structured Documents – A Formal and Logical Approach*. PhD thesis, Limburgs Universitair Centrum, 1999.

[24] F. Neven. "Automata Theory for XML Researchers". *SIGMOD Record*, **31**(3), Sept. 2002.

[25] F. Neven and T. Schwentick. "Expressive and Efficient Pattern Languages for Tree-Structured Data". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'00)*, pages 145–156, Dallas, Texas, USA, 2000. ACM Press.

[26] F. Neven and T. Schwentick. "Query Automata on Finite Trees". *Theoretical Computer Science*, **275**:633–674, 2002.

[27] L. Stockmeyer and A. Meyer. "Word Problems Requiring Exponential Time". In *Proceedings of the 5th ACM Symposium on Theory of Computing*, pages 1–9, 1973.

[28] J. Thatcher and J. Wright. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic". *Mathematical Systems Theory*, **2**(1):57–81, 1968.

[29] M. Y. Vardi. "The Complexity of Relational Query Languages". In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 137–146, San Francisco, CA USA, May 1982.

[30] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.