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 [23] 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 [15]) 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 [16]).

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 [24]. 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. [26, 12]). 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 [13].

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, 24, 25, 13], as a basis for XML schema languages [19], for defining XML transducers [20], and for XML data stream processing [17]; See [23] for a survey of automata-theoretic work related to XML.

Most important in our context are *query automata*, proposed by Neven and Schwentick [25] 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,

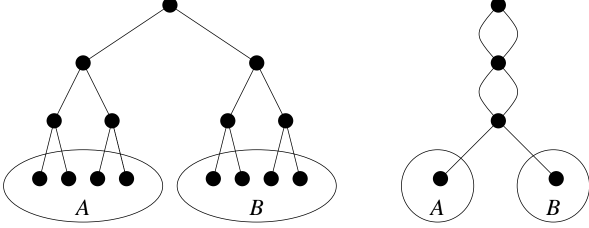

Figure 1. Two bisimilar instances over schema $\sigma = \{A, B\}$.

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 12), 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 $|E^{\boldsymbol{I}}| \leq O(||\boldsymbol{I}|| \cdot \text{mult}(\boldsymbol{I}))$.

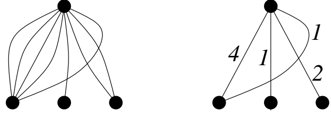

Figure 2.

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 [9], or [18] 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, [10]). 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. [13]). 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.

A *valuation* on a datalog rule r is a mapping φ which maps each variable of r to a domain element. Given an atom $p(x_1, \ldots, x_m)$, let

$$\varphi(p(x_1, \ldots, x_m)) := p(\varphi(x_1), \ldots, \varphi(x_n)).$$

The popular fixpoint semantics of (monadic) datalog can be defined by means of a monotonic *immediate consequence operator* $\mathcal{T}_{\mathcal{P}}$. Given a set of ground (i.e., variable-free) atoms X,

$$\mathcal{T}_{\mathcal{P}}(X) := X \cup \{\varphi(h) \mid \text{there is a rule } h \leftarrow b_1, \ldots, b_n. \text{ in } \mathcal{P} \text{ and a valuation } \varphi \text{ on the rule s.t. } \varphi(b_1), \ldots, \varphi(b_n) \in X\}$$

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)

$$\begin{aligned} P(x) &\leftarrow U(x). & (1)\\ P(x) &\leftarrow P_0(x_0) \wedge B(x_0, x). & (2)\\ P(x_0) &\leftarrow P_0(x) \wedge B(x_0, x). & (3)\\ P(x) &\leftarrow P_1(x) \wedge P_2(x). & (4) \end{aligned}$$

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

Proposition 4 ([13]). *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 [13] and a program $\mathcal{P}$ can be evaluated in time $O(||\boldsymbol{T}|| * |\mathcal{P}|)$ when $\boldsymbol{T}$ is a tree-instance [13]. The following new result is based on a lazy rule-instantiation version of an algorithm by Minoux [21]:

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

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

3.5. Core XPath. XPath uses thirteen binary relations – called axes – for navigating in trees [27]. 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). 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 [15]).

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):

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

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

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 [27], both syntactically and semantically.

A mapping from Core XPath to monadic datalog with stratified negation was given in [14]; we strengthen this re-

sult 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.*

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 [16].

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.

Theorem 8. *Given a TMNF program $\mathcal{P}$, an instance $\boldsymbol{I}$, and a node $v \in \boldsymbol{T}(\boldsymbol{I})$, the set $\{P(v) \in \mathcal{T_P}^{\omega}(\boldsymbol{T}(\boldsymbol{I}))\}$ can be computed in space $O(||\boldsymbol{I}|| * |IDB(\mathcal{P})|)$.*

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

Theorem 9. *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.*

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 10. *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 $O(|\mathcal{P}| * ||\boldsymbol{J}||)$.*

This result is based on a variation of Minoux' algorithm [21] (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.

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 3. 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 $\boldsymbol{B}(\boldsymbol{I})$.

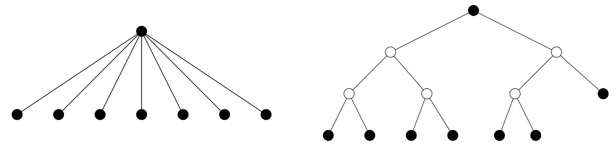

Figure 3.

We omit a formal definition of $\boldsymbol{B}(\boldsymbol{I})$. The following proposition is crucial. Its proof is straightforward; Figure 4 illustrates why there will be a blow-up in size logarithmic in the maximum edge multiplicity.

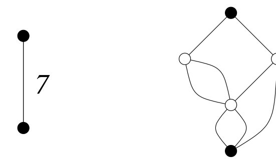

Figure 4.

Proposition 11. *There is an algorithm that, given an instance $\boldsymbol{I}$, computes a binary instance $\boldsymbol{B}$ equivalent to $\boldsymbol{B}(\boldsymbol{I})$ in time $O\big(||\boldsymbol{I}|| \cdot \log(\text{mult}(\boldsymbol{I}))\big)$.*

Remark 12. Note that if we used a logarithmic cost model, then the logarithmic factor in Proposition 11 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 11 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 13. 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 5 shows $\boldsymbol{I}_3$.

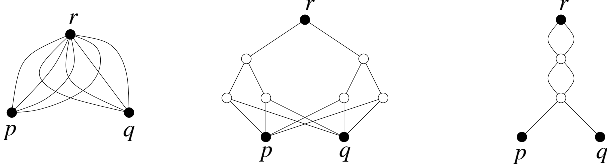

Figure 5. The instance $\boldsymbol{I}_3$ of Example 13, the binary instance $\boldsymbol{B}(\boldsymbol{I}_3)$, and its minimisation $\boldsymbol{M}(\boldsymbol{B}(\boldsymbol{I}_3))$.

It is easy to see that $\boldsymbol{I}_\ell$ is minimal and that $||\boldsymbol{I}_\ell|| \in O(2^\ell)$. However, as Figure 5 illustrates, the minimal instance $\boldsymbol{M}(\boldsymbol{B}(\boldsymbol{I}_\ell))$ bisimilar to the binary instance $\boldsymbol{B}(\boldsymbol{I}_\ell)$ has size $O(\ell)$.

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.

Proposition 14. *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. $Q(\boldsymbol{I}) = Q'(\boldsymbol{B}(\boldsymbol{I}))$.*

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.

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., [25, 17]). 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, 11, 25]). 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 [25] 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

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

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 $\rho : V^{\boldsymbol{T}} \to Q$ 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 $\rho(root^{\boldsymbol{T}}) \in F$. 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 15. 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 18). Of course it would even be better if we could simply use *deterministic* tree automata.[1] The following example shows that this would not be sufficient.

Example 16. 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 [25] 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 17 (see also [22]). *A query (on binary tree-instances) is definable in MSO if, and only if, it is definable by an STA.*

Corollary 18 (see also [22]). *For every STA $\mathfrak{A}$ there exists an STA $\mathfrak{A}'$ such that for all binary tree instances $\boldsymbol{T}$,*

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

[1] 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.

Since monadic datalog is contained in MSO, Theorem 17 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 19. Let σ be a schema and $X_1, \ldots, X_\ell \notin \sigma$. 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 $Q = 2^{\{X_1,\ldots,X_\ell\}}$, $F = Q$, 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 $\{X_1, \ldots, X_\ell\}$ 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 [11] with the partial decompression methods that we also use to evaluate XPath and monadic datalog queries.

Proposition 20. *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.*

Remark 21. 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 22. *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.*

Remark 23. 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 24. *The evaluation problem for MSO-queries on binary instances parameterized by the size of the input formula is fixed-parameter tractable.*

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

Theorem 25. *The evaluation problem for $\exists$MSO-queries on binary instances is NEXPTIME-complete. Evaluation of MSO queries can be done in EXPSPACE.*

If we convert a monadic datalog program into an STA in the way described in Example 19 and then use Theorem 22 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 26. 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 27. The automaton constructed from a monadic datalog program in Example 19 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 28. *For every STA there is a WSTA that defines the same query.*

Theorem 29. *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.*

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-Sibling*, *First-Child*, *Next-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 30. *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.

Acknowledgments

We thank an anonymous reviewer for pointing out to us that an unpublished part of Frank Neven's PhD thesis [22], pp. 128–130, defines the notion of *nondeterministic query automata* which precisely equals ours of STAs on (uncompressed) trees. Theorem 17 and Corollary 18 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] R. Downey and M. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

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