Enumeration for MSO-Queries on Compressed Trees

MARKUS LOHREY, Universität Siegen, Germany
MARKUS L. SCHMID, Humboldt-Universität zu Berlin, Germany

We present a linear preprocessing and output-linear delay enumeration algorithm for MSO-queries over trees that are compressed in the well-established grammar-based framework. Time bounds are measured with respect to the size of the compressed representation of the tree. Our result extends previous work on the enumeration of MSO-queries over uncompressed trees and on the enumeration of document spanners over compressed text documents.

CCS Concepts: • **Theory of computation** → *Tree languages*; *Finite Model Theory*; *Database query languages (principles)*; *Database query processing and optimization (theory).*

Additional Key Words and Phrases: MSO-Enumeration, Unranked Trees, Evaluation Over Compressed Data, Straight-Line Programs

ACM Reference Format:
Markus Lohrey and Markus L. Schmid. 2024. Enumeration for MSO-Queries on Compressed Trees. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 78 (May 2024), 17 pages. https://doi.org/10.1145/3651141

1 INTRODUCTION

The paradigm of *algorithmics on compressed data* (ACD) aims at solving fundamental computational tasks directly on compressed data objects, without prior decompression. This allows us to work in a completely compressed setting, where our data is always stored and processed in a compressed form. ACD works very well with respect to *grammar-based compression* with so-called straight-line programs (SLPs). Such SLPs use grammar-like formalisms in order to specify how to construct the data object from small building blocks. For example, an SLP for a string w is just a context-free grammar for the language $\{w\}$, which can be seen as a sequence of instructions that construct w from the terminal symbols. For instance, the SLP $S \to AA$, $A \to BBC$, $B \to ba$, $C \to cb$ (where S, A, B, C are nonterminals and a, b, c are terminals) produces the string $babacbbabacb$. String SLPs are very popular and many results exist that demonstrate their wide-range applicability (see, e. g., [4, 12, 17, 19] for some recent publications, and the survey [26]). Moreover, SLPs achieve very good compression rates in practice (exponential in the best case) and are tightly related to dictionary based compression, in particular LZ77 and LZ78 [13, 34].

An important point is that the ACD pradigm may lead to substantial running time improvements over the uncompressed setting. Indeed, the algorithm's running time only depends on the size of the compressed input, so the smaller size of the input may directly translate into a lower running time. For example, if the same problem can be solved in linear time both in the uncompressed and in the compressed setting, then in the case that the input can be compressed from size n to size $O(\log(n))$, the algorithm in the compressed setting is exponentially faster. This is not just

Authors' addresses: Markus Lohrey, lohrey@eti.uni-siegen.de, Universität Siegen, Hölderlinstr. 3, 57076, Siegen, Germany; Markus L. Schmid, MLSchmid@MLSchmid.de, Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099, Berlin, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/5-ART78
https://doi.org/10.1145/3651141

hypothetically speaking. In the field of string algorithms several fundamental problems are known with this behaviour, e. g., string pattern matching [17].

Recently, the ACD paradigm has been combined with the enumeration perspective of query evaluation. In [32, 35, 36], the information extraction framework of document spanners is investigated in the compressed setting, and it has been shown that the results of regular spanners over SLP-compressed text documents can be enumerated with linear preprocessing and constant delay. Applying SLP-based ACD in the framework of document spanners suggests itself, since this is essentially a query model for string data (or sequences), and ACD is most famous in the realm of string algorithms.

We consider a different classical evaluation problem in the compressed setting: MSO-queries on trees. It is known that MSO-queries can be enumerated with linear preprocessing and output-linear delay [1, 2, 14, 24]. Output linear delay means that the delay is linearly bounded by the size of the next element that is enumerated. If the elements of the solution have constant size (like, e. g., for document spanners), then constant delay and output-linear delay are the same. Our main result extends the work of [2, 14, 24] to the setting, where the tree is given in a compressed representation. Our data models are unranked forests, and as compression scheme, we consider forest straight-line programs (FSLPs) [20]. This is a convenient and versatile grammar-based compression scheme for unranked forests, which also covers other SLP-models from the literature. As usual, MSO-queries are represented by tree automata. Let us now explain the concept of SLP-compression of trees and forests in more detail.

Grammar-compression of trees and forests: An advantage of grammar-based compression is that it can be easily extended to other data types. For example, by using a context-free grammar formalism for trees, we can define tree SLPs [27, 30, 31] and use them for the compression of ranked trees. However, in the context of database theory, we are rather interested in unranked trees and forests as data model. A typical example of such data are XML tree structures. Therefore, we use *forest straight-line programs* (FSLPs) introduced in [20]. Let us sketch this model.

The set of rules of a string SLP essentially define a DAG (directed acyclic graph) whose inner nodes (resp., leaves) are the non-terminals (resp., terminals) of the grammar. The edges of the DAG are defined by the productions of the grammar (e. g., $S \to AB$ means that from S there is a left edge to A and a right edge to B). This DAG can also be seen as an *algebraic circuit* over the free monoid Σ^* where Σ is the terminal alphabet. FSLPs are also DAGs that represent circuits, but the underlying algebraic structure is the *forest algebra* (see [6]). The elements of this algebra are forests over the alphabet $\Sigma \cup \{*\}$ (i.e., ordered sequences of unranked trees with nodes labelled by symbols from $\Sigma \cup \{*\}$) with the restriction that the special symbol $*$ occurs at most once in a forest and moreover only at a leaf position. Intuitively, $*$ represents a substitution point where another forest can be plugged in. Such forests can be built from atomic forests of the form $a \in \Sigma$ (representing a single node-labelled a) and a_* (representing a node-labelled Σ with a single child labelled $*$) using two operations: the operation $\ominus$ horizontally concatenates two forests (where $*$ is allowed to occur in at most one of them) and the operation $\varobar$ substitutes in a forest F_1 containing $*$ the unique occurrence of $*$ by a second forest F_2. The tree T_e in Figure 1 shows an expression in the forest algebra that evaluates to the tree T at the right of Figure 2. Now the DAG D at the left of Figure 2 is an FSLP, since it is a natural DAG representation of T_e. An FSLP produces a unique forest that we denote with $[\![D]\!]$. For the FSLP at the left of Figure 2 it is exactly the tree T at the right of Figure 2. Consequently, FSLPs follow a two stage approach: A forest is represented as the parse-tree of a forest algebra expression, and this parse-tree is then folded into a DAG.

Let us motivate our choice of FSLPs as compression scheme. First, FSLPs can describe node-labelled unranked forests and therefore cover a large number of tree structures (e.g. XML tree

structures). Moreover, FSLPs also cover other popular tree compression schemes like top dags [5, 15, 23] and tree straight-line programs [18, 28]. Furthermore, there exist compressors such as TreeRePair that produce FSLPs[1] and show excellent compression ratios in practice. For a corpus of typical XML documents, the number of edges of the original tree is reduced to approximately 3% using TreeRePair on the first-child next-sibling encoding of the XML tree [28]. Other available grammar-based tree compressors are BPLEX [10] and CluX [7]. It is also known that for every forest with n nodes one can construct in linear time an FSLP of size $O(n \log |\Sigma| / \log n)$ (so $O(n/\log n)$ for a fixed Σ) [18]. Finally, notice that FSLPs generalize string SLPs (only use the horizontal concatenation $\ominus$).

Main result: The following is a preliminary statement of our main result. Upper bounds refer to data complexity.

Theorem 1.1. *For an MSO-query $\psi(X)$ with a free set variable X and a forest straight-line program D one can compute in preprocessing time $O(|D|)$ a data structure that allows to enumerate with output-linear delay all $S \subseteq \mathsf{nodes}(\llbracket D \rrbracket)$ such that $\llbracket D \rrbracket \models \Psi(S)$.*

The MSO-query $\psi(X)$ will be represented by a tree automaton $\mathcal{A}$ (for an automaton model that is equivalent to MSO on unranked forests). We first reduce the enumeration problem from Theorem 1.1 to the following enumeration problem for binary trees: given a DAG D that represents a binary node-labelled tree T and a deterministic bottom-up tree automaton $\mathcal{B}$, enumerate all subsets $S \subseteq \mathsf{leaves}(T)$ such that $\mathcal{B}$ accepts the tree (T, S) (obtained by marking all leaves from S with a 1). This enumeration problem is then instantiated by the FSLP D (a DAG) and the forest algebra parse tree T represented by D. The nodes of the forest $\llbracket D \rrbracket$ uniquely correspond to the leaves of T (see Figures 1 and 2). The bottom-up tree automaton $\mathcal{B}$ is obtained from the automaton $\mathcal{A}$ using a known construction [25]. Since T is a binary tree, we could enumerate all sets $S \subseteq \mathsf{leaves}(T)$ such that $\mathcal{B}$ accepts (T, S) with linear preprocessing and output-linear delay by one of the algorithms in the literature, e. g., the one by Bagan [2]. But T is given by the DAG D and we cannot afford to explicitly construct T. Consequently, we have to adapt Bagan's algorithm in such a way that it can be used directly on DAG-compressed trees, which is our main technical contribution.

Related results: Forest algebras have been also used in the context of MSO-enumeration on trees in [25, 33] for the purpose of enabling updates of the queried tree in logarithmic time by updating and re-balancing a forest algebra expression for the tree. However no compression is achieved in [25, 33]. Working with a DAG-representation of a forest algebra expression (i.e., an FSLP) is a new aspect of our work and requires new algorithmic techniques. We discuss in Section 8 the possibilities and difficulties for also supporting updates in a compressed setting.

The arguably most simple way of compressing a tree is to fold it into a DAG. In the context of database theory, this type of compression has been investigated in [9, 16] for XPath and monadic datalog queries, but the enumeration perspective is not investigated. Moreover, the simple DAG-compression has other disadvantages in comparison to the grammar-based approach. In general, DAG-compression can achieve exponential compression, but there are also trees non-compressible by DAGs, but still exponentially compressible by FSLPs, e. g., unary trees of the form $f(f(\ldots f(a)\ldots))$. The experimental study of [28] also shows that in a practical setting DAG-compression cannot compete with grammar-based tree compression. Further work on the compression performance of DAGs for XML can be found in [8, 29].

Our work can also be seen as extending the results of [32, 35, 36] to the case of MSO queries on trees, instead of regular spanners on text documents. In this regard, we also provide an alternative

[1] Actually, TreeRePair works for ranked trees. It can produce a tree SLP for the first-child-next-sibling of an unranked forest F. This tree SLP can be transformed in linear time into an equivalent FSLP for F [20].

proof for the result that regular spanners can be enumerated with linear preprocessing and constant delay over SLP-compressed texts [32] (the latter uses so-called enumerable compact sets with shifts).

In addition to Bagan's algorithm, there are also other enumeration algorithms for MSO queries over uncompressed trees [14, 24]. However, Bagan's algorithm seems to be the one that can be extended the best to the compressed setting.

2 PRELIMINARIES

2.1 Trees and Forests

Different rooted and finite trees will appear in this work. Note that forests are simply sequences of trees, and we should keep in mind that each definition of a certain tree model yields the corresponding concept of forests (which are just sequences of such trees).

Node-labelled ordered trees are trees, where nodes are labelled with symbols from some alphabet Σ and may have an arbitrary number of children (i.e., the trees are unranked). Moreover, the children are linearly ordered. A typical example of such trees are XML tree structures. A node-labelled ordered tree can be defined as structure $T = (V, E, R, \lambda)$, where V is the set of nodes, E is the set of edges, R is the sibling relation (i.e., $(u, v) \in R$ if and only if v is the right sibling of u), and $\lambda : V \longrightarrow \Sigma$ is the function that assigns labels to nodes. In the following, when we speak of a tree, we always mean a node-labelled ordered tree. A *forest* is a (possibly empty) ordered sequence of trees. We write $\mathcal{F}(\Sigma)$ for the set of all forests. The size $|F|$ of a forest is the number of nodes of F. Forests are the structures on which we want to enumerate query results.

We also use a term representation for forests, i. e., we write elements of $\mathcal{F}(\Sigma)$ as strings over the alphabet $\Sigma \cup \{(,)\}$.

Node-labelled binary trees (binary trees for short) are the special case of the trees from the previous paragraph, where every node is either a leaf or has two children (a left and a right child). It is then more common to replace the two relations E (edge relation) and R (sibling relation) by the relations E_ℓ (left edges) and E_r (right edges), where $(u, v) \in E_\ell$ (resp., $(u, v) \in E_r$) if v is the left (resp., right) child of u. We write $E = E_\ell \cup E_r$ for the set of all edges. Our binary trees have the additional property that Σ is partitioned into two disjoint sets Σ_0 and Σ_2 labelling leaves and internal nodes, respectively. We use the above term notation for general trees also for binary trees. With $\mathsf{leaves}(T)$ we denote the set of leaves of the binary tree T. We number the leaves of T from left to right starting with 0. The number assigned to $v \in \mathsf{leaves}(T)$ is the *leaf number* of v; see the binary tree in Figure 1, where the leaf numbers are written in blue color. Binary trees will be used for describing algebraic expressions (mainly expressions in the so-called forest algebra).

Unordered trees are trees, without node labels and without an order on the children of a node. They will be used as auxiliary data structures in our algorithms. An unordered tree will be defined as a pair (V, E), where V is the set of nodes and E is the edge relation. An unordered forest is a disjoint union of unordered trees.

Trees with edge weights: We also have to consider (possibly unordered) trees, where the edges are labelled with weights from a monoid M. We then add to the description of the tree a function $\gamma : E \longrightarrow M$ that assigns weights to edges. In our specific application, M will be the monoid $(\mathbb{N}, +)$. But there are further applications, where a different choice of M is needed. Since $M = (\mathbb{N}, +)$ in our application, we write the monoid additively, i.e., we use + for the monoid operation and 0 for the neutral element. But let us emphasize that we do not use that M is commutative. The weights can be lifted from edges to paths in the natural way: let $(v_1, v_2, v_3, \cdots, v_{d-1}, v_d)$ be the unique path from v_1 to a descendant v_d in the tree, i.e., $(v_i, v_{i+1}) \in E$ for all $i \in [1, d-1]$. We then define $\gamma(v_1, v_d)$ as

the sum $\sum_{i=1}^{d-1} \gamma(v_i, v_{i+1})$. For a leaf v of the tree T we define $\gamma(v) = \gamma(r, v)$, where r is the root of T, and for a set of leaves S we define $\gamma(S) = \{\gamma(v) : v \in S\}$.

2.2 Monadic Second Order Logic Over Forests

We consider MSO-formulas that are interpreted over forests $F = (V, E, R, \lambda)$. Since any first-order variable x (that takes elements from V as values) can be replaced by a set variable X (an MSO-formula can express that X is a singleton set), we can restrict to MSO-formulas where all free variables are set variables. If Ψ is an MSO-formula with free set variables $X_1, \dots, X_k$ and $S_1, \dots, S_k \subseteq V$ are node sets, then we write $(F, S_1, \dots, S_k) \models \Psi$ if the formula Ψ holds in the forest F if the variable X_i is set to S_i. To make the exposition less technical, we further restrict to MSO-formulas with a single free set variable X. This is actually no restriction. The restriction to MSO-formulas with a single free set variable is a common one that can be found elsewhere in the literature; see, e. g., [3, 16].

2.3 Tree Automata

We consider two types of tree automata: deterministic bottom-up tree automata that work on binary trees and nondeterministic stepwise tree automata that work on general trees. Since they should implement queries on trees, they will be interpreted as selecting nodes from trees (this aspect is explained in more detail later on).

A **deterministic bottom-up tree automaton** (over the alphabets Σ_0, Σ_2) is represented by a tuple $\mathcal{B} = (Q, \Sigma_0, \Sigma_2, \delta_0, \delta_2, Q_f)$, where Q is a finite set of states, Σ_0 is the set of leaf node labels, Σ_2 is the set of labels for internal nodes, $Q_f \subseteq Q$ is the set of final states, $\delta_0 : \Sigma_0 \to Q$ assigns states to leaves of a tree, and $\delta_2 : Q \times Q \times \Sigma_2 \to Q$ assigns states to internal nodes depending on the node label and the states of the two children. For a given binary tree T we define the state $\mathcal{B}(T)$ as the unique state to which $\mathcal{B}$ evaluates the tree T. It is inductively defined as follows, where $a \in \Sigma_0$ and $f \in \Sigma_2$:

- $\mathcal{B}(a) = \delta_0(a)$ and
- $\mathcal{B}(f(T_1, T_2)) = \delta_2(\mathcal{B}(T_1), \mathcal{B}(T_2), f)$ for trees T_1 and T_2.

The binary tree T is accepted by $\mathcal{B}$ if and only if $\mathcal{B}(T) \in Q_f$. With $L(\mathcal{B})$ we denote the set of binary trees accepted by $\mathcal{B}$. We use the abbreviation dBUTA for deterministic bottom-up tree automaton.

Stepwise tree automata (nSTAs for short) are an automaton model for forests that is equivalent to MSO-logic [11]. In fact, we only use known results for nSTAs (Theorems 2.1 and 4.2 below). Therefore we skip the definition of nSTAs. With $L(\mathcal{A}) \subseteq \mathcal{F}(\Sigma)$ we denote the set of forests accepted by $\mathcal{A}$.

Tree automata can represent queries on trees and forests as follows. For a forest F (note that this includes trees) and a subset S of its nodes, we identify the pair (F, S) with the forest that is obtained from F by relabelling every a-labelled node v of F ($a \in \Sigma$) with $(a, \beta) \in \Sigma \times \{0, 1\}$, where $\beta = 1$ if and only if $v \in S$. Intuitively, (F, S) represents the forest F from which the nodes in S have been selected (or the forest F together with a possible query result S). Our nSTAs become node-selecting, by taking $\Sigma \times \{0, 1\}$ as the set of node labels. Such an nSTA $\mathcal{A}$ selects the node set S from a forest $F \in \mathcal{F}(\Sigma)$ if and only if $(F, S) \in L(\mathcal{A})$.

Our dBUTAs only need the ability to select leaves of binary trees, which means that we define them over the alphabets $\Sigma_0 \times \{0, 1\}$ (for leaf nodes) and Σ_2 (for internal nodes), i. e., we run them on pairs (T, S), where T is a binary tree and S is a subset of its leaves.

In the following, we assume that all nSTAs and dBUTAs are node-selecting in the above sense. For the forest $F = (V, E, R, \lambda)$ and an nSTA $\mathcal{A}$ we write

$$\text{select}(\mathcal{A}, F) = \{S \subseteq V : (F, S) \in L(\mathcal{A})\}$$

for the node set selected by the nSTA $\mathcal{A}$. Similarly, for a binary tree T and a dBUTA $\mathcal{B}$ we define

$$\mathsf{select}(\mathcal{B}, T) = \{S \subseteq \mathsf{leaves}(T) : (T, S) \in L(\mathcal{B})\}.$$

It is known that MSO-formulas (that are interpreted over forests) can be translated into equivalent automata (and vice versa). More precisely, we use the following well-known fact:

THEOREM 2.1 (C.F. [11]). *From an MSO-formula $\Psi(X)$ one can construct an nSTA $\mathcal{A}$ such that for every forest $F \in \mathcal{F}(\Sigma)$ with node set V we have* $\mathsf{select}(\mathcal{A}, F) = \{S \subseteq V : (F, S) \models \Psi(X)\}$.

Our main goal is to enumerate all sets $S \subseteq V$ such that $(F, S) \models \Psi(X)$ holds. By Theorem 2.1 this is equivalent to enumerate all S such that (F, S) is accepted by an nSTA. We will therefore ignore MSO logic in the following and directly start from an nSTA.

2.4 Enumeration Algorithms

We use the standard RAM model (with two restrictions for the register length; see the end of this section and the end of Section 3.2). An enumeration algorithm A produces on input I an *output sequence* $(s_1, s_2, \dots, s_m, \mathsf{EOE})$, where EOE is the *end-of-enumeration* marker. We say that A on input I *enumerates* a set S if and only if the output sequence is $(s_1, s_2, \dots, s_m, \mathsf{EOE})$, $|S| = m$ and $S = \{s_1, s_2, \dots, s_m\}$. The *preprocessing time* (of A on input I) is the time that elapses between starting $A(I)$ and the output of the first element, and the *delay* is the time that elapses between any two elements of the output sequence. The preprocessing time and the delay of A is the maximum preprocessing time and delay, respectively, over all possible inputs of length at most n (viewed as a function of n).

The gold standard in the area of enumeration algorithms is (i) *linear preprocessing* and (ii) *output-linear delay*. Linear preprocessing means that the preprocessing phase needs time $O(|I|)$ and output-linear delay means that for every output s_i the delay is $O(|s_i|)$. If every output s_i has constant size (which for the RAM model means that it occupies a constant number of registers), then output-linear delay is the same as constant delay.

We are interested in enumeration algorithms that enumerate the set $\mathsf{select}(\mathcal{A}, F)$ for a fixed nSTA $\mathcal{A}$ and a forest F. The input is F, while $\mathcal{A}$ is fixed and not part of the input, i. e., we measure data complexity. Theorem 4.1 below also addresses the dependence in the size of $\mathcal{A}$, but we made no effort to optimize the latter.

The special feature of this work is that the input forest F is not given explicitly, but in a potentially highly compressed form, and the enumeration algorithm must be able to handle this compressed representation rather than decompressing it. This aspect shall be explained in detail in the next section.

Recall that we deal with weights from a monoid M (written in additive notation). We assume that an element of M can be stored in a single register of our RAM and that for given $\gamma_1, \gamma_2 \in M$, $\gamma_1 + \gamma_2$ can be computed in constant time on the RAM. In our application, M is the monoid $(\mathbb{N}, +)$, for which this assumption clearly holds (in Section 3.2 we say more about the bit lengths of the integers).

3 TREE COMPRESSION

We are interested in enumerating the result of MSO-queries on *compressed forests*. For compression, we use the well-established grammar-based framework that has been investigated in the context of trees and forests before [5, 20, 22, 27, 28]. For this, we first need to introduce the representation of trees by directed acyclic graphs.

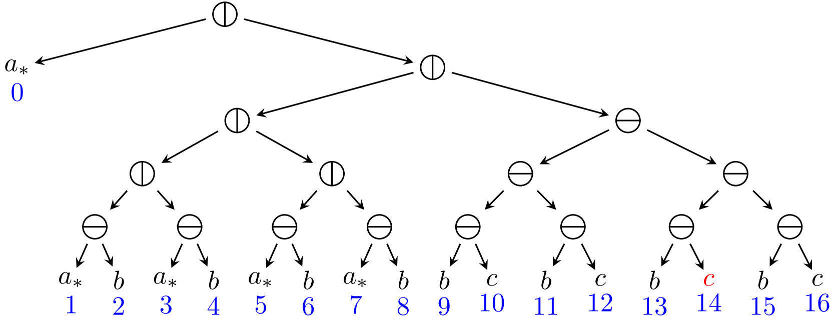

Fig. 1. A forest algebra expression.

3.1 Representing Trees by DAGs

An *unordered DAG* (with edge weights from the monoid M) is a triple $D = (V, E, \gamma)$, where V is a finite set of vertices, $E \subseteq V \times I \times V$ is the set of edges and the function $\gamma : E \longrightarrow M$ assigns weights to edges. Here, I is a finite (unordered) index set, whose elements are used in order to distinguish multiple edges between the same endpoints. For D being a DAG, we require that the binary relation $\{(u, v) \in V \times V : \exists i \in I : (u, i, v) \in E\}$ is acyclic. The size $|D|$ of D is defined as $|E|$. The outdegree (resp., indegree) of a vertex v is the number of edges of the form (v, i, u) (resp., (u, i, v)).

A path (from v_1 to v_n) is a word $\pi = v_1 i_1 \cdots v_{n-1} i_{n-1} v_n$ such that $n \geq 1$ and $(v_k, i_k, v_{k+1}) \in E$ for all $1 \leq k \leq n-1$. The length of this path π is $n-1$ and its weight is the sum $\gamma(\pi) = \sum_{k=1}^{n-1} \gamma(v_k, i_k, v_{k+1})$. If $n = 1$ (in which case we have $\pi = v_1$) we speak of an empty path. To define the (unordered) tree $\mathsf{unfold}_D(v)$ for a vertex v we take for the node set all paths $v_1 d_1 \cdots v_{n-1} d_{n-1} v_n$ with $v_1 = v$ and $n \geq 1$. There is an edge from a path $\pi = v_1 i_1 \cdots v_{n-1} i_{n-1} v_n$ to every path of the form $\pi' = \pi j v'$ ($j \in I, v' \in V$) and the weight of this edge is $\gamma(v_n, j, v')$. For $v \in V$ and $U \subseteq V$ let $\mathsf{path}_D(v, U)$ be the set of all paths from v to some vertex in U.

A (node-labelled) *binary DAG* (with edge weights) is a tuple $D = (V, E, \lambda, \gamma, v_0)$, where (V, E, γ) is a DAG as above with the index set $I = \{\ell, r\}$, v_0 is the root vertex and $\lambda : V \longrightarrow \Sigma_0 \cup \Sigma_2$ is the vertex-labelling function. Moreover, for every $v \in V$, if v is a leaf in D (i.e., has outdegree zero), then $\lambda(v) \in \Sigma_0$, and if v has non-zero outdegree, then $\lambda(v) \in \Sigma_2$ and v has exactly two outgoing edges of the form (v, ℓ, v_1) (a left edge) and (v, r, v_2) (a right edge). We also omit the edge weight function γ from the description of the binary DAG D if it is not important and write $D = (V, E, \lambda, v_0)$ in this case.

We define the tree $\mathsf{unfold}_D(v)$ for a binary DAG D as above. It becomes a binary node-labelled tree if we define the node labelling function λ as follows: for a path $\pi = v_1 i_1 \cdots i_{n-1} i_{n-1} v_n$ we set $\lambda(\pi) = \lambda(v_n)$. Moreover, the edge from a path π to a path $\pi j v'$ ($j \in \{\ell, r\}$, $v' \in V$) is a left (resp., right) edge if $j = \ell$ (resp., $j = r$). Finally, we define the binary tree $\mathsf{unfold}(D)$ as $\mathsf{unfold}_D(v_0)$.

3.2 Forest Straight-Line Programs

We next define the concept of forest straight-line programs, which combines the DAG-representation of trees with forest algebras.

Forest algebras: Recall that $\mathcal{F}(\Sigma)$ is the set of all forests with node labels from Σ. Let us fix a distinguished symbol $* \notin \Sigma$. The set of forests $F \in \mathcal{F}(\Sigma \cup \{*\})$ such that $*$ has a unique occurrence in F and this occurrence is at a leaf node is denoted by $\mathcal{F}_*(\Sigma)$. Elements of $\mathcal{F}_*(\Sigma)$ are called *forest contexts*. Note that $\mathcal{F}(\Sigma) \cap \mathcal{F}_*(\Sigma) = \emptyset$. Following [6], we define the *forest algebra* as the 2-sorted algebra $\mathsf{F}(\Sigma) = (\mathcal{F}(\Sigma) \cup \mathcal{F}_*(\Sigma), \ominus, \mathbin{\varobar}, \varepsilon, *)$, where $\varepsilon \in \mathcal{F}(\Sigma)$ is the empty forest, $* \in \mathcal{F}_*(\Sigma)$ is the

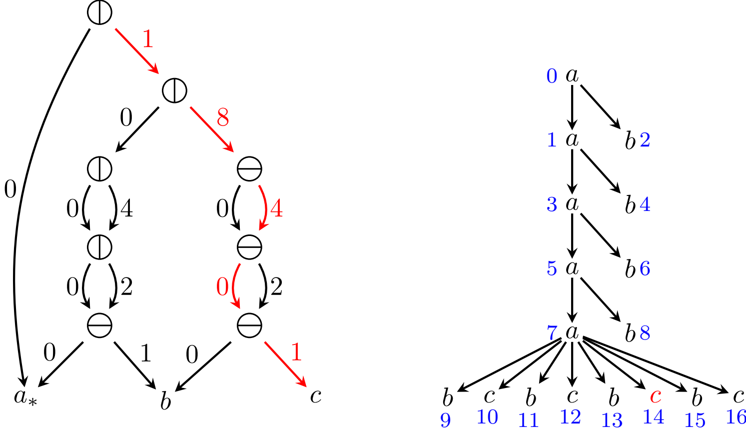

Fig. 2. An example FSLP D (left side) that describes the tree $[\![D]\!]$ on the right side. The forest algebra expression that corresponds to D's unfolding is shown in Figure 1.

empty forest context, and $\ominus$ (*horizontal concatenation*) and $\varobar$ (*vertical concatenation*) are partially defined binary operations on $\mathcal{F}(\Sigma) \cup \mathcal{F}_*(\Sigma)$ that are defined as follows:

- For $F_1, F_2 \in \mathcal{F}(\Sigma) \cup \mathcal{F}_*(\Sigma)$ such that $F_1 \in \mathcal{F}(\Sigma)$ or $F_2 \in \mathcal{F}(\Sigma)$, we set $F_1 \ominus F_2 = F_1 F_2$ (i.e., we concatenate the corresponding sequences of trees).
- For $F_1 \in \mathcal{F}_*(\Sigma)$ and $F_2 \in \mathcal{F}(\Sigma) \cup \mathcal{F}_*(\Sigma)$, $F_1 \varobar F_2$ is obtained by replacing in F_1 the unique occurrence of $*$ by F_2.

Note that $(\mathcal{F}(\Sigma), \ominus, \varepsilon)$ and $(\mathcal{F}_*(\Sigma), \varobar, *)$ are monoids. For $a \in \Sigma$, we write a_* for the forest context $a(*)$ which consists of an a-labelled root with a single child labelled with $*$. Note that $a = a_* \varobar \varepsilon$. In [6] the forest algebra is introduced as a two sorted algebra with the two sorts $\mathcal{F}(\Sigma)$ and $\mathcal{F}_*(\Sigma)$. Our approach with partially defined concatenation operators is equivalent.

A *forest algebra expression* is an expression over the algebra $\mathsf{F}(\Sigma)$ with atomic subexpressions of the form a and a_* for $a \in \Sigma$. Such an expression can be identified with a node-labelled binary tree, where every internal node is labelled with the operator $\ominus$ or $\varobar$ and every leaf is labelled with a symbol a or a_* for $a \in \Sigma$. Not all such trees are valid forest algebra expressions. For instance $a_* \ominus a_*$ is not valid, since it would produce a forest with two occurrences of $*$. It is easy to check with a dBUTA $\mathcal{B}_0$ with three states, whether a binary tree is a valid forest algebra expression. We will only consider valid forest algebra expressions in the following. We write $\mathcal{E}(\Sigma)$ for the set of all valid forest algebra expressions. Elements of $\mathcal{E}(\Sigma)$ will be denoted with T_e. With $[\![T_e]\!] \in \mathcal{F}(\Sigma) \cup \mathcal{F}_*(\Sigma)$ we denote the forest or forest context obtained by evaluating T_e in the forest algebra. The empty forest ε and the empty forest context $*$ are not allowed in forest algebra expressions, which is not a restriction as long as we only want to produce non-empty forests and forest contexts; see [19, Lemma 3.27].

The leaves of $T_e \in \mathcal{E}(\Sigma)$ are in a one-to-one correspondence with the nodes of the forest $[\![T_e]\!]$. The T_e-number of a node of $[\![T_e]\!]$ is defined as the leaf number of the corresponding leaf of T_e. Note that every node of $[\![T_e]\!]$ is uniquely identified by its T_e-number. Figure 1 shows a forest algebra expression. The forest produced by this expression is shown on the right of Figure 2. Every node is labelled with its T_e-number in blue.

A **forest straight-line program** (over Σ), FSLP for short, is a binary DAG $D = (V, E, \lambda, v_0)$ (without edge weights for the moment) such that $\mathsf{unfold}(D) \in \mathcal{E}(\Sigma)$ and $F := [\![\mathsf{unfold}(D)]\!] \in \mathcal{F}(\Sigma)$. We also

write $[\![D]\!]$ for this forest F. FSLPs were introduced in [30] in a more grammar-like but equivalent way. An FSLP where all internal vertices are labelled with $\ominus$ is just a string straight-line program.

We now add edge weights to an FSLP. Consider an FSLP $D = (V, E, \lambda, v_0)$ with $T_e = \mathsf{unfold}(D)$. Recall that we refer to the nodes of $F := [\![D]\!] = [\![T_e]\!]$ by their T_e-numbers. We will call these numbers also the D-numbers. Another representation of the nodes of $[\![D]\!]$ is given by paths from the root vertex v_0 to the leaves of the DAG D. These paths correspond to the leaves of the expression T_e and hence to the nodes of the forest F. The D-number of a node of F can be easily computed when walking down a path π from the root v_0 to a leaf of D if we assign to every edge $(u, d, v) \in E$ an integer weight $\gamma(u, d, v)$, called the *offset* of the edge. For left edges we set $\gamma(u, \ell, v) = 0$. Now consider a right edge (u, r, v_2) and let (u, ℓ, v_1) be the corresponding left edge for u. Then we define $\gamma(u, r, v_2)$ as the number of leaves in the tree $\mathsf{unfold}_D(v_1)$. With this weight function γ, the D-number of the node of F that corresponds to the path π is exactly the weight $\gamma(\pi)$. Note that the D-number of a node v is not the same as the preorder number of v.

Figure 2 shows an FSLP D on the left with the corresponding forest $[\![D]\!]$ (actually, a tree) on the right. Every edge of D is labelled with its offset. The red path in D determines the red c-labelled node in the tree $[\![D]\!]$. The D-number of this node is 14, which is the sum of edge weights of the red path.

We make the assumption that RAM-algorithms for an FSLP-compressed forest F have registers of word length $O(\log |F|)$. This is a standard assumption in the area of algorithms for compressed data. It allows to store the D-number of a node of F in a register.

4 MAIN RESULT AND ITS PROOF OUTLINE

In this section we outline our enumeration algorithm for MSO-queries on FSLP-compressed forests. As explained before, we can directly start with an nSTA $\mathcal{A}$. The main result of the paper is:

THEOREM 4.1. *From an nSTA $\mathcal{A}$ with m states and an FSLP D one can compute in preprocessing time $O(|D|) \cdot 2^{O(m^4)}$ a data structure that allows to enumerate the set* $\mathsf{select}(\mathcal{A}, [\![D]\!])$ *with output-linear delay. In the enumeration, nodes of $[\![D]\!]$ are represented by their D-numbers.*

The first step in our proof of Theorem 4.1 is a reduction to binary trees. We use the following result that is implicitly shown in [25]. Let $\Sigma_0 = \{a, a_* : a \in \Sigma\}$ and $\Sigma_2 = \{\ominus, \oplus\}$:

THEOREM 4.2 (C.F. [25]). *From an nSTA $\mathcal{A}$ over Σ with m states one can construct a dBUTA $\mathcal{B}$ over Σ_0, Σ_2 with $2^{m^2} + 2^{m^4} + 1$ states such that $L(\mathcal{B}) = \{T_e \in \mathcal{E}(\Sigma) : [\![T_e]\!] \in L(\mathcal{A})\}$.*

Theorem 4.2 is stated in [25] (except for the size bound) using the concept of finite transition algebras, which is equivalent to nSTAs.

PROOF SKETCH FOR THEOREM 4.1. We can apply Theorem 4.2 to the nSTA $\mathcal{A}$ from Theorem 4.1. First recall that the leaves of $T_e \in \mathcal{E}(\Sigma)$ (with $[\![T_e]\!] \in \mathcal{F}(\Sigma)$) can be identified with the nodes of the forest $[\![T_e]\!]$. Hence, for a subset $S \subseteq \mathsf{leaves}(T_e)$, we can view (T_e, S) (i. e., the tree T_e with the leaves from S selected; see Section 2.3) as a forest algebra expression from $\mathcal{E}(\Sigma \times \{0, 1\})$ such that $[\![(T_e, S)]\!] = ([\![T_e]\!], S)$ (for this, we identify $(a, i)_*$ with (a_*, i) for $i \in \{0, 1\}$).

With Theorem 4.2 we obtain from the nSTA $\mathcal{A}$ in Theorem 4.1 a dBUTA $\mathcal{B}$ such that for every forest algebra expression T_e and every subset $S \subseteq \mathsf{leaves}(T_e)$ we have: $([\![T_e]\!], S) \in L(\mathcal{A})$ iff $(T_e, S) \in L(\mathcal{B})$.

If the forest algebra expression T_e would be given explicitly (say, by a pointer structure) then we could use Bagan's algorithm from [2] in order to enumerate all sets S with $(T_e, S) \in L(\mathcal{B})$. But in the situation of Theorem 4.1 the expression T_e is not given explicitly but it is the unfolding $T_e = \mathsf{unfold}(D)$ of the input FSLP D, which is a binary DAG. Let $F := [\![D]\!] = [\![T_e]\!]$ be the forest

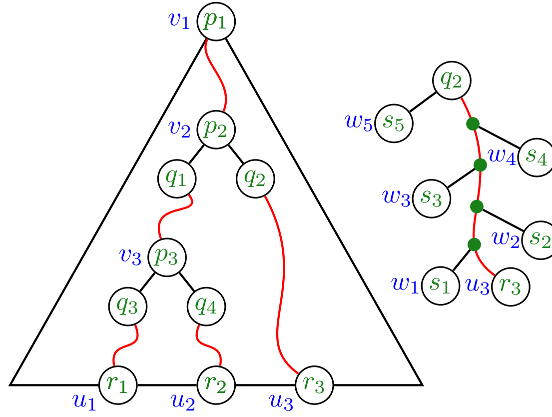

Fig. 3. A witness tree: automaton states are in green, node names (if written) are in blue. The path from q_2 to r_3 is shown on the right with the nodes that branch off from the path.

produced by the FSLP D. The nodes of F correspond to paths from the root to a leaf in the DAG D. We built an edge-labelled DAG from D by labelling every edge of the DAG D by its offset. The offsets are positive integers. All offsets can be easily computed in time $O(|D|)$ by computing in a bottom-up parse for every node v of the DAG D the number of leaves of the tree $\mathsf{unfold}_D(v)$. Recall that the sum of the offsets on a path π from the root to a leaf in D is exactly the D-number of the node of F that corresponds to the path π. It therefore suffices to prove the following theorem:

Theorem 4.3. *From a dBUTA $\mathcal{B}$ with m states and a node-labelled binary DAG $D = (V, E, \lambda, \gamma, v_0)$ with edge weights from the monoid M, one can compute in preprocessing time $O(|D| \cdot m^2)$ a data structure that allows to enumerate the set $\{\gamma(S) : S \in \mathsf{select}(\mathcal{B}, \mathsf{unfold}(D))\}$ with output-linear delay.*[2]

We obtain Theorem 4.1 by specializing M to $(\mathbb{N}, +)$ and taking for D an FSLP, whose edge weights are the offsets. By choosing for M a different, more complicated, monoid consisting of affine transformations on $\mathbb{N}$, one can also prove a variant of Theorem 4.1, where every node of $[\![D]\!]$ is represented by its preorder number (which is a more canonical representative of the node compared to the D-number).

We will prove Theorem 4.3 (and hence Theorem 4.1) in Section 6 with the last missing piece moved to Section 7. The proof builds on Bagan's enumeration algorithm from [2], which solves the problem from Theorem 4.3 (without the weight function γ) for the case of a binary tree that is given explicitly and not as the unfolding of a DAG. In the following, we explain Bagan's original algorithm. Then, in Section 7 we explain the main algorithmic tool that allows us to extend Bagan's algorithm to DAGs. □

5 BAGAN'S ALGORITHM FOR BINARY TREES

Let $T = (V, E_\ell, E_r, \lambda, \gamma)$ be a node-labelled binary tree with edge weights as described in Section 2.1. The weights are from a monoid M. For a node $v \in V$ we denote with T_v the subtree rooted in v. For $S \subseteq \mathsf{leaves}(T)$ and $v \in V$ we define $S_v = S \cap \mathsf{leaves}(T_v)$.

[2]For a general weight function γ, there may exist different sets $S_1, S_2 \in \mathsf{select}(\mathcal{B}, \mathsf{unfold}(D))$ such that $\gamma(S_1) = \gamma(S_2)$. In this case, our enumeration algorithm will enumerate the set $\gamma(S_1)$ twice. On the other hand, for the above weight function defined by offsets, we have $\gamma(S_1) \neq \gamma(S_2)$ whenever $S_1 \neq S_2$.

Let $\mathcal{B} = (Q, \Sigma_0 \times \{0,1\}, \Sigma_2, \delta_0, \delta_2, Q_f)$ be a dBUTA. We will first ignore the weights from the monoid M. Hence, we are interested in the non-empty sets in $\mathsf{select}(\mathcal{B}, T)$ (whether $\emptyset \in \mathsf{select}(\mathcal{B}, T)$ can be checked in the preprocessing). Bagan [2] was the first who presented an algorithm with output-linear delay for enumerating these sets. In the following we explain Bagan's algorithm in a slightly different way that will be convenient for our extension to DAG-compressed binary trees, which will be the main building block for our enumeration algorithm for FSLP-compressed forests. Our discussion will be informal.

The goal of Bagan's algorithm is to enumerate all non-empty sets $S \in \mathsf{select}(\mathcal{B}, T)$ together with a witness that $(T, S) \in L(\mathcal{B})$ holds. A first step towards such a witness is to replace in T every node $v \in V$ by the pair (v, q) (a so-called *configuration*), where q is the unique state $q = \mathcal{B}(T_v, S_v)$ at which $\mathcal{B}$ arrives in node v. Let us call this tree the *configuration tree*.

The delay for producing a solution S must be in $O(|S|)$ (i. e., we need output-linear delay), but the configuration tree has size $|T|$ and is therefore too big. A next step towards a solution is to prune the configuration tree by keeping only those nodes that are on a path from the root to a leaf from S. This yields a tree with only $|S|$ many leaves that we call the *pruned configuration tree*. It is shown in Figure 3 for an example, where $S = \{u_1, u_2, u_3\}$. The old node names of T are written in blue, automaton states are written in green. All tree nodes of the new pruned configuration tree are from the following set of so-called *active configurations*:

$$\mathsf{Conf}^a(T) = \{(v, q) : \exists S \subseteq \mathsf{leaves}(T_v) : S \neq \emptyset, q = \mathcal{B}(T_v, S)\}.$$

But the pruned configuration tree is still too big because it may contain long paths of unary nodes (nodes with a single child except for the last node on the path). In Figure 3 these are the red paths. The edges on these paths can be described as follows. The configurations that were removed in the pruning are from the set

$$\mathsf{Conf}^{\emptyset}(T) = \{(v, q) \in V \times Q : q = \mathcal{B}(T_v, \emptyset)\}.$$

The configurations $(w_1, s_1), \ldots, (w_5, s_5)$ in Figure 3 on the right are from this set. On the set of active configurations $\mathsf{Conf}^a(T)$ we define a weighted edge relation $\rightarrow$ as follows: For active configurations $(u, p), (v, q) \in \mathsf{Conf}^a(T)$ with u internal in T and labelled with $f \in \Sigma_2$, there is an edge $(u, p) \rightarrow (v, q)$ with weight $\gamma((u, p), (v, q)) = \gamma(u, v)$ if there is $(v', q') \in \mathsf{Conf}^{\emptyset}(T)$ such that one of the following two cases holds:

- $(u, v) \in E_\ell$, $(u, v') \in E_r$ and $\delta_2(q, q', f) = p$,
- $(u, v) \in E_r$, $(u, v') \in E_\ell$ and $\delta_2(q', q, f) = p$.

Then all the edges of the unary paths in the pruned configuration tree (the red paths in Figure 3) are of the above form $(u, p) \rightarrow (v, q)$. The configuration $(v', q') \in \mathsf{Conf}^{\emptyset}(T)$ is a configuration to which an additional edge branches off from the red unary paths (configurations $(w_1, s_1), \ldots, (w_5, s_5)$ in Figure 3). We define the edge weighted graph $T \otimes \mathcal{B} = (\mathsf{Conf}^a(T), \rightarrow, \gamma)$. Since $\mathcal{B}$ is deterministic and T is a tree, $T \otimes \mathcal{B}$ is an unordered forest.

The final idea is to contract the red paths in Figure 3 to single edges; this results in a tree of size $O(|S|)$, which is called a *witness tree* W. To define (and construct) witness trees it is useful to define a further set of configurations, the so-called *useful configurations*: An active configuration $(v, q) \in \mathsf{Conf}^a(T)$ is useful if either v is a leaf in T or v has the children v_1, v_2 in T and there exist states $q_1, q_2 \in Q$ such that $\delta_2(q_1, q_2, \lambda(v)) = q$ and $(v_1, q_1), (v_2, q_2) \in \mathsf{Conf}^a(T)$. We denote the set of useful configurations with $\mathsf{Conf}^u(T)$. Note that $\mathsf{Conf}^u(T) \subseteq \mathsf{Conf}^a(T)$. The useful configurations are the leaves and the binary nodes in witness trees. In Figure 3 these are the configurations $(u_1, r_1), (u_2, r_2), (u_3, r_3)$ and $(v_2, p_2), (v_3, p_3)$.

Let us now give a the formal definition of witness trees:

Definition 5.1. A *witness tree* W for a configuration $(v, q) \in \mathsf{Conf}^a(T)$ is a tree with root (v, q). It is constructed recursively: If $v \in \mathsf{leaves}(T)$ then (v, q) is the only node of W. Assume now that $v \notin \mathsf{leaves}(T)$. Then (v, q) has a single child, which is a configuration from $(v', q') \in \mathsf{Conf}^u(T)$ such that $(v, q) \to^* (v', q')$ (we may have $(v', q') = (v, q)$ in which case we introduce a copy of (v, q)).[3] If $v' \in \mathsf{leaves}(T)$ then (v', q') is a leaf of W. Otherwise, let $(v', v_1) \in E_\ell$ and $(v', v_2) \in E_r$. Then there exist states q_1, q_2 such that $\delta_2(q_1, q_2, \lambda(v')) = q'$, $(v_1, q_1), (v_2, q_2) \in \mathsf{Conf}^a(T)$ and the two children of (v', q') are (v_1, q_1) and (v_2, q_2). The node (v_i, q_i) ($i \in \{1, 2\}$) is the root for a witness tree for (v_i, q_i).

For a witness tree W, let $S(W)$ be the set of all $v \in \mathsf{leaves}(T)$ such that some (v, q) is a leaf of W.

Lemma 5.2. *The following holds for every $(v, q) \in \mathsf{Conf}^a(T)$:*

- *Every witness tree W contains at most $4|S(W)|$ many nodes.*
- *If $\mathcal{B}(T_v, S) = q$ for some non-empty $S \subseteq \mathsf{leaves}(T_v)$ then there is a unique witness tree W for (v, q) with $S = S(W)$.*
- *If there is a witness W for (v, q) then $\mathcal{B}(T_v, S(W)) = q$.*

By this lemma, it suffices to enumerate all witness trees W for (r, q_f) where r is the root of T and q_f goes over all final states. Fix one q_f. The construction of witness trees for (r, q_f) follows the recursive definition of Definition 5.1. In order to systematically enumerate all witness trees, we have to fix a linear order on all witness trees such that from a witness tree W we can construct the next witness tree W'. One option is to list the nodes of a witness tree in preorder, say $w_1, w_2, \ldots, w_n$. When we construct for a node $w = (v, q)$ of a witness tree the children of w, there are several choices according to Definition 5.1. We can order those choices linearly and assign to node w the choice number $c(w)$ if we took for w the $c(w)$-th choice during the construction of the witness tree (for a leaf w we may set $c(w) = 0$). The choice sequence $c(w_1), c(w_2), \ldots, c(w_n)$ then uniquely encodes the witness tree (since it can be constructed from the choice sequence). Moreover, we can order choice sequences lexicographically. It is then easy to construct from a given witness tree W the lexicographically next witness tree W' in time $O(|W'|)$.

One can also show that all precomputations can be done in linear time.

Lemma 5.3. *The sets $\mathsf{Conf}^a(T)$, $\mathsf{Conf}^u(T)$, $\mathsf{Conf}^\emptyset(T)$, and the forest $T \otimes \mathcal{B}$ can be computed bottom-up on the tree T in time $O(|T| \cdot |Q|^2)$.*

6 BAGAN'S ALGORITHM FOR DAGS

Assume now that the binary tree T is given by a node-labelled binary DAG $D = (V, E, \lambda, \gamma, v_0)$ with edge weights from the monoid M. Thus, we have $T = \mathsf{unfold}(D)$. Then, the forest $T \otimes \mathcal{B}$ can also be represented by a small DAG. To see this, first recall that the sets $\mathsf{Conf}^a(T)$, $\mathsf{Conf}^u(T)$, and $\mathsf{Conf}^\emptyset(T)$ can be computed bottom-up for T; see Lemma 5.3. In particular, if the subtrees T_v and T_w are isomorphic ($T_v \cong T_w$ for short) then for every state $q \in Q$ and every $x \in \{a, u, \emptyset\}$ we have $(v, q) \in \mathsf{Conf}^x(T)$ if and only if $(w, q) \in \mathsf{Conf}^x(T)$. We can therefore define sets $\mathsf{Conf}^x(D) \subseteq V \times Q$ by saying that $(v, q) \in \mathsf{Conf}^x(D)$ if and only if $(\tilde{v}, q) \in \mathsf{Conf}^x(T)$, where $\tilde{v}$ is any one of the tree nodes represented by the DAG node v (formally, $\tilde{v}$ is a path in D from v_0 to v). The sets $\mathsf{Conf}^a(D)$, $\mathsf{Conf}^u(D)$, and $\mathsf{Conf}^\emptyset(D)$ can be precomputed in time $O(|D| \cdot |Q|^2)$ using exactly the same bottom-up computation as for Lemma 5.3.

We then define, analogously to $T \otimes \mathcal{B}$, the unordered DAG $D \otimes \mathcal{B} = (\mathsf{Conf}^a(D), E', \gamma')$ with edge weights and index set $I = \{\ell, r\}$. To define the edge set $E' \subseteq \mathsf{Conf}^a(D) \times \{\ell, r\} \times \mathsf{Conf}^a(D)$, let $d \in \{\ell, r\}$ and $(v, q), (v', q') \in \mathsf{Conf}^a(D)$ such that $\lambda(v) = f \in \Sigma_2$. Then, there is an edge

[3] Here, '$\to^*$' is the reflexive-transitive closure of '$\to$'.

$((v,q), d, (v',q')) \in E'$ of weight $\gamma(v,d,v')$ iff there is $(v'',q'') \in \mathsf{Conf}^{\emptyset}(D)$ such that one of the following cases holds:

- $d = \ell$, $(v,\ell,v'), (v,r,v'') \in E$ and $\delta_2(q',q'',f) = q$.
- $d = r$, $(v,r,v'), (v,\ell,v'') \in E$ and $\delta_2(q'',q',f) = q$.

By a bottom-up computation along D, we can compute $D \otimes \mathcal{B}$ in time $O(|D| \cdot |Q|^2)$ analogously to $T \otimes \mathcal{B}$.

If $T_v \cong T_w$ then $T_v \otimes \mathcal{B} \cong T_w \otimes \mathcal{B}$. More precisely, if ι is the isomorphism from T_v to T_w, then the mapping $(x,q) \mapsto (\iota(x),q)$ is an isomorphism from $T_v \otimes \mathcal{B}$ to $T_w \otimes \mathcal{B}$. We therefore obtain:

Lemma 6.1. $T \otimes \mathcal{B} \cong \mathsf{unfold}(D \otimes \mathcal{B})$.

Our goal is to run the enumeration algorithm from Section 5 for the tree $T = \mathsf{unfold}(D)$. Of course, we cannot afford to construct T explicitly; it can be of exponential size. The solution is the following: we enumerate witness trees W in the same way as we did before, but we want to work with the DAG $D \otimes \mathcal{B}$ instead of the forest $T \otimes \mathcal{B}$. More precisely, the algorithm should produce all witness trees W for $T = \mathsf{unfold}(D)$. There is only one step, which is problematic. In Definition 5.1 one has to choose a configuration $(v',q') \in \mathsf{Conf}^u(T)$ that can be reached from a configuration $(v,q) \in \mathsf{Conf}^a(T)$ in the forest $T \otimes \mathcal{B}$. The original variant of the algorithm of [2] precomputes a fixed linear order on the set of these (v',q'). In the DAG-version of the witness tree construction we have a configuration $(v,q) \in \mathsf{Conf}^a(D)$ and we want to extend the witness tree W below (v,q). It is clearly not sufficient to merely enumerate all $(v',q') \in \mathsf{Conf}^u(D)$ that are reachable from (v,q) in $D \otimes \mathcal{B}$ and continue the witness tree construction in (v',q'). What we have to do is to enumerate all paths from $\mathsf{path}_{D\otimes\mathcal{B}}((v,q), \mathsf{Conf}^u(D))$, i. e., all paths π from (v,q) to some configuration in $\mathsf{Conf}^u(D)$. These paths correspond to the configurations in $\mathsf{Conf}^u(T)$ that are reachable from $(\tilde{v},q)$ in $T \otimes \mathcal{B}$, where $\tilde{v}$ is any one of the tree nodes represented by the DAG node v. In fact, it is not necessary to print out the actual paths $\pi \in \mathsf{path}_{D\otimes\mathcal{B}}((v,q), \mathsf{Conf}^u(D))$. We only need to keep from the path π its endpoint that we denote with $\omega(\pi)$ and its weight $\gamma(\pi)$. Hence, our new goal is to proof the following result:

Theorem 6.2. *Let $D = (V,E,\gamma)$ be an unordered DAG with edge weights from the monoid M and let $V_0 \subseteq V$ be a distinguished set of target vertices. In time $O(|D|)$ one can compute a data structure that allows to enumerate for a given start node $s \in V$ in constant delay all pairs $\langle \omega(\pi_1), \gamma(\pi_1)\rangle, \ldots, \langle \omega(\pi_k), \gamma(\pi_k)\rangle$ where* $\mathsf{path}_D(s,V_0) = \{\pi_1,\ldots,\pi_k\}$.[4]

Note that the data structure that is computed in the preprocessing phase from D and V_0 can be used for every start vertex s.

Once we have shown Theorem 6.2, it is easy to complete Bagan's algorithm for DAGs and thus prove Theorem 4.3. We build in the preprocessing phase the data structure from Theorem 6.2 for the DAG $D \otimes \mathcal{B}$ and take for V_0 the set $\mathsf{Conf}^u(D)$. If during the construction of a witness tree W, we want to extend the current partial witness tree at a copy of configuration $(v,q) \in \mathsf{Conf}^a(D)$,[5] we have to start the enumeration algorithm from Theorem 6.2 with $s = (v,q)$. Each time a pair $\langle (v',q'), \gamma' \rangle$ is produced by the algorithm, we extend the current partial witness tree with an edge from (v,q) to (v',q') of weight $\gamma(v')$. Then we continue the extension of the witness tree in (v',q'). As in Section 5, the size of the witness tree W is $4|S(W)|$ (see Lemma 5.2), and we spend constant time for each node of W. For the latter, it is important that the algorithm from Theorem 6.2 works in constant delay. During the construction of the witness tree W we store for every node $w = (q,v)$

[4]In general, we might have $\langle \omega(\pi_i), \gamma(\pi_i)\rangle = \langle \omega(\pi_j), \gamma(\pi_j)\rangle$ for $i \neq j$, although this does not happen in our specific application, where the edge weights are offsets.

[5]In contrast to Section 5 the witness tree W may contain many copies of the same configuration (v,q), i.e., several nodes that are labelled with the same configuration $(v,q) \in \mathsf{Conf}^a(D)$.

Algorithm 1: path_enumeration(s)

```
   variables: v ∈ V, γ ∈ M, stack S ∈ (V_2 × M)*, flag ∈ {0, 1}
 1 v := s ; γ := 0 ; stack := ε ; flag := 1 ;
 2 while true do
 3   | if flag = 1 then print ⟨ω_r[v], γ + γ_r[v]⟩ end;
 4   | flag := 1;
 5   | if v ∈ V_2 then
 6   |   | if v[r] ∈ V_2 then S.push⟨v[r], γ + γ(v, r, v[r])⟩ end;
 7   |   | v := v[ℓ]; γ := γ + γ(v, ℓ, v[ℓ]);
 8   | else if stack ≠ ε then
 9   |   | ⟨v′, γ′⟩ := S.pop; v := v′; γ := γ′; flag := 0;
10   | else
11   |   | print EOE;  stop;
```

of W the sum of the edge weights from the root of W to w. For every leaf $w = (q, v)$ of the witness tree we then print the weight stored for w; this yields the sets $\gamma(S)$ in Theorem 4.3.

7 PATH ENUMERTION IN DAGS

In this section we sketch the proof of Theorem 6.2. In a preprocessing phase, taking time $O(|D|)$, we reduce to the simpler case, where (i) the DAG $D = (V, E, \gamma)$ is *binary* (every vertex v has outdegree zero or two) and (ii) V_0 is the set of leaves of D. For a vertex $v \in V_2 := V \setminus V_0$ we write $v[\ell]$ (resp., $v[r]$) for the left (resp., right) child of v. For a vertex v we denote by $\omega_r[v] \in V_0$ the unique leaf that is reached from v by only following right edges, and we define $\gamma_r[v]$ to be the sum of the weights along this path of right edges. This data can be easily computed in time $O(|D|)$. Now, the task is to enumerate for a given vertex s all pairs $\langle\omega(\pi_1), \gamma(\pi_1)\rangle, \ldots, \langle\omega(\pi_k), \gamma(\pi_k)\rangle$, where $\{\pi_1, \ldots, \pi_k\} = \mathrm{path}_D(s, V_0)$. We assume for simplicity that $\gamma(\pi_i) \neq \gamma(\pi_j)$ whenever $i \neq j$ (which holds in our application; see the footnote in Theorem 6.2).

We will now explain our path enumeration algorithm (Algorithm 1). We ignore Lines 3 and 4, and the flag variable for the moment. Let T_s be the tree obtained from unfolding the DAG D starting from s. By the assumption from the previous paragraph, the nodes of T_s can be identified with the pairs $\langle v, \gamma\rangle$, where γ is the weight of a path from s to $v \in V$. Such a pair is stored by the algorithm. The current pair $\langle v, \gamma\rangle$ is updated according to a preorder traversal of T_s: If v is an inner vertex (Line 5), then we move on to its left child (Line 7) and store its right child on the stack (Line 6). When we reach a leaf (Line 8), we pop the topmost stack element (Line 9) and continue the traversal there, unless the stack is empty, in which case the algorithm terminates (Line 11). The weight γ is correctly updated in Lines 6, 7 and 9.

We also print all pairs $\langle\omega(\pi_1), \gamma(\pi_1)\rangle, \ldots, \langle\omega(\pi_k), \gamma(\pi_k)\rangle$ from Theorem 6.2, which are exactly the pairs $\langle v, \gamma\rangle$ where v is a leaf of D and γ is the weight of a path from s to v. However, printing these pairs when we see them during the preorder traversal would not result in a constant delay. Instead, whenever we visit a pair $\langle v, \gamma\rangle$, we print the pair $\langle\omega_r[v], \gamma + \gamma_r[v]\rangle$, i. e., the leaf reached from v by only moving along right edges together with the weight of the corresponding path (note that this pair is among those pairs that have to be enumerated). This is done in Line 3 (assume for now that the flag is 1). This, however, leads to duplicates whenever the variable v is updated in Line 9 due to a pop of the stack, since then $\langle\omega_r[v], \gamma + \gamma_r[v]\rangle$ has already been produced in an earlier iteration where we have visited an ancestor of $\langle v, \gamma\rangle$ (this was the reason why $\langle v, \gamma\rangle$ ended up on the stack in the first place). Consequently, we use the flag to avoid this: whenever the new

pair $\langle v, \gamma \rangle$ is obtained by popping from the stack (Line 9), we set the flag to 0 to avoid that we output $\langle \omega_r[v], \gamma + \gamma_r[v] \rangle$ in the next iteration of Line 3. Moreover, since v is an inner vertex (otherwise $\langle v, \gamma \rangle$ would not be on the stack), we know that we will next descend from $\langle v, \gamma \rangle$ to its left child in Line 7, which corresponds to visiting a new node of T_s. Thus, we know that $\langle \omega_r[v], \gamma + \gamma_r[v] \rangle$ has not been produced before, and therefore it is now correct to output $\langle \omega_r[v], \gamma + \gamma_r[v] \rangle$ when we reach Line 3 the next time. This explains why we set the flag back to 1 every time we reach Line 3 with the flag being 0.

Let $\langle u, \zeta \rangle$ be a leaf of T_s. If $\langle u, \zeta \rangle$ is a left child, then it is produced when $v = u$ and $\gamma = \zeta$ (because $\omega_r[u] = u$ and $\gamma_r[\zeta] = 0$), which happens at some point. If $\langle u, \zeta \rangle$ is a right child then it is not explicitly visited, since only right children that are inner nodes are pushed on the stack (Line 6). We nevertheless print $\langle u, \zeta \rangle$ when we visit the highest ancestor (in the sense of being closest to the root) of $\langle u, \zeta \rangle$ in T_s from which only right edges lead to $\langle u, \zeta \rangle$. Hence, we produce exactly $\langle \omega(\pi_1), \gamma(\pi_1) \rangle, \ldots, \langle \omega(\pi_k), \gamma(\pi_k) \rangle$ where $\{\pi_1, \ldots, \pi_k\} = \mathsf{path}_D(s, V_0)$.

The delay of Algorithm 1 is constant since there cannot be two consecutive iterations of the while loop where the flag is 0.

8 CONCLUSIONS

We conclude with a brief discussion of related aspects of our result. It is possible to support in time $O(\log |F|)$ relabelling updates, i. e., updates that change the current label of a given node v to a given label $a \in \Sigma$. By updating the path in the FSLP that represents v, such an update can be carried out in time linear in the depth of the FSLP. By using the linear time balancing theorem from [19, Corollary 3.28], the depth can be assumed to be $O(\log |F|)$. We conjecture that the deletion and insertion updates from [25] can be also implemented in time $O(\log |F|)$ when F is given by an FSLP. However, [25] uses a quite technical notion of balancedness for forest algebra parse trees and it is not clear how to preserve this notion of balancedness when the parse tree is compressed as a DAG.

For the case that M is a group, Theorem 6.2 can also be proven by using a known technique for the real-time traversal of SLP-compressed strings (see [21, 30]). However, if we want to represent the nodes of the solution sets in Theorem 4.1 by their preorder numbers (the arguably more intuitive representation) instead of their D-numbers we need for M a monoid, which is not a group. Another disadvantage is that the real-time traversal of SLP-compressed strings needs a tree data structure that provides so-called next link queries. While such data structures can be constructed in linear time, this is not straightforward and would significantly complicate an implementation of our algorithm. In general, we believe that our approach is simple to implement, which makes an experimental analysis in the vein of [28] possible.

ACKNOWLEDGMENTS

The second author is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – project number 522576760 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 522576760).

REFERENCES

[1] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Enumeration on Trees with Tractable Combined Complexity and Efficient Updates. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019*. ACM, 89–103. https://doi.org/10.1145/3294052.3319702

[2] Guillaume Bagan. 2006. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *Proceedings of the 20th International Workshop on Computer Science Logic, CSL 2006 (Lecture Notes in Computer Science, Vol. 4207)*. Springer, 167–181. https://doi.org/10.1007/11874683_11

[3] Guillaume Bagan. 2009. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries).* Ph.D. Dissertation. University

of Caen Normandy, France. https://tel.archives-ouvertes.fr/tel-00424232
[4] Hideo Bannai, Momoko Hirayama, Danny Hucke, Shunsuke Inenaga, Artur Jez, Markus Lohrey, and Carl Philipp Reh. 2021. The Smallest Grammar Problem Revisited. *IEEE Transactions on Information Theory* 67, 1 (2021), 317–328. https://doi.org/10.1109/TIT.2020.3038147
[5] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. 2015. Tree compression with top trees. *Information and Computation* 243 (2015), 166–177. https://doi.org/10.1016/J.IC.2014.12.012
[6] Mikołaj Bojańczyk and Igor Walukiewicz. 2008. Forest algebras. In *Proceedings of Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]. (Texts in Logic and Games, Vol. 2).* Amsterdam University Press, 107–132.
[7] Stefan Böttcher, Rita Hartel, and Christoph Krislin. 2010. CluX - Clustering XML Sub-trees. In *Proceedings of the 12th International Conference on Enterprise Information Systems, ICEIS 2010, Volume 1, DISI.* SciTePress, 142–150.
[8] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Nöth. 2015. XML Compression via Directed Acyclic Graphs. *Theory of Computing Systems* 57, 4 (2015), 1322–1371. https://doi.org/10.1007/s00224-014-9544-x
[9] Peter Buneman, Martin Grohe, and Christoph Koch. 2003. Path Queries on Compressed XML. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003.* Morgan Kaufmann, 141–152. https://doi.org/10.1016/B978-012722442-8/50021-5
[10] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. 2008. Efficient memory representation of XML document trees. *Information Systems* 33, 4-5 (2008), 456–474. https://doi.org/10.1016/j.is.2008.01.004
[11] Julien Carme, Joachim Niehren, and Marc Tommasi. 2004. Querying Unranked Trees with Stepwise Tree Automata. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications, RTA 2004 (Lecture Notes in Computer Science, Vol. 3091)*, Vincent van Oostrom (Ed.). Springer, 105–118. https://doi.org/10.1007/978-3-540-25979-4_8
[12] Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid. 2021. On the Complexity of the Smallest Grammar Problem over Fixed Alphabets. *Theory of Computing Systems* 65, 2 (2021), 344–409. https://doi.org/10.1007/s00224-020-10013-w
[13] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7 (2005), 2554–2576. https://doi.org/10.1109/TIT.2005.850116
[14] Bruno Courcelle. 2009. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics* 157, 12 (2009), 2675–2700. https://doi.org/10.1016/J.DAM.2008.08.021
[15] Bartlomiej Dudek and Pawel Gawrychowski. 2018. Slowing Down Top Trees for Better Worst-Case Compression. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018 (LIPIcs, Vol. 105).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:8. https://doi.org/10.4230/LIPIcs.CPM.2018.16
[16] Markus Frick, Martin Grohe, and Christoph Koch. 2003. Query Evaluation on Compressed Trees (Extended Abstract). In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science LICS 2003.* IEEE Computer Society, 188. https://doi.org/10.1109/LICS.2003.1210058
[17] Moses Ganardi and Pawel Gawrychowski. 2022. Pattern Matching on Grammar-Compressed Strings in Linear Time. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022.* SIAM, 2833–2846. https://doi.org/10.1137/1.9781611977073.110
[18] Moses Ganardi, Danny Hucke, Artur Jez, Markus Lohrey, and Eric Noeth. 2017. Constructing small tree grammars and small circuits for formulas. *J. Comput. System Sci.* 86 (2017), 136–158. https://doi.org/10.1016/j.jcss.2016.12.007
[19] Moses Ganardi, Artur Jez, and Markus Lohrey. 2021. Balancing Straight-line Programs. *Journal of the ACM* 68, 4 (2021), 27:1–27:40. https://doi.org/10.1145/3457389
[20] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. 2020. Grammar-Based Compression of Unranked Trees. *Theory of Computing Systems* 64, 1 (2020), 141–176. https://doi.org/10.1007/s00224-019-09942-y
[21] Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. 2005. Real-Time Traversal in Grammar-Based Compressed Files. In *Proceedings of the 2005 Data Compression Conference, DCC 2005.* IEEE Computer Society, 458. https://doi.org/10.1109/DCC.2005.78
[22] Pawel Gawrychowski and Artur Jez. 2016. LZ77 Factorisation of Trees. In *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016 (LIPIcs, Vol. 65).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 35:1–35:15. https://doi.org/10.4230/LIPIcs.FSTTCS.2016.35
[23] Lorenz Hübschle-Schneider and Rajeev Raman. 2015. Tree Compression with Top Trees Revisited. In *Proceedings of the 14th International Symposium on Experimental Algorithms, SEA 2015 (Lecture Notes in Computer Science, Vol. 9125).* Springer, 15–27. https://doi.org/10.1007/978-3-319-20086-6_2
[24] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of monadic second-order queries on trees. *ACM Transaction on Computational Logic* 14, 4 (2013), 25:1–25:12. https://doi.org/10.1145/2528928
[25] Sarah Kleest-Meißner, Jonas Marasus, and Matthias Niewerth. 2023. MSO Queries on Trees: Enumerating Answers under Updates Using Forest Algebras. https://doi.org/10.48550/ARXIV.2208.04180 arXiv:2208.04180 [cs.LO]

[26] Markus Lohrey. 2012. Algorithmics on SLP-Compressed Strings: A Survey. *Groups Complexity Cryptology* 4, 2 (2012), 241–299. https://doi.org/10.1515/GCC-2012-0016

[27] Markus Lohrey. 2015. Grammar-based tree compression. In *Proceedings of the 19th International Conference on Developments in Language Theory, DLT 2015 (Lecture Notes in Computer Science, Vol. 9168)*. Springer, 46–57. https://doi.org/10.1007/978-3-319-21500-6_3

[28] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. 2013. XML tree structure compression using RePair. *Information Systems* 38, 8 (2013), 1150–1167. https://doi.org/10.1016/j.is.2013.06.006

[29] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. 2017. Compression of Unordered XML Trees. In *Proceedings of the 20th International Conference on Database Theory, ICDT 2017 (LIPIcs, Vol. 68)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:17. https://doi.org/10.4230/LIPIcs.ICDT.2017.18

[30] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. 2018. Constant-Time Tree Traversal and Subtree Equality Check for Grammar-Compressed Trees. *Algorithmica* 80, 7 (2018), 2082–2105. https://doi.org/10.1007/S00453-017-0331-3

[31] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. 2012. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. System Sci.* 78, 5 (2012), 1651–1669. https://doi.org/10.1016/j.jcss.2012.03.003

[32] Martin Muñoz and Cristian Riveros. 2023. Constant-Delay Enumeration for SLP-Compressed Documents. In *Proceedings of the 26th International Conference on Database Theory, ICDT 2023 (LIPIcs, Vol. 255)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:17. https://doi.org/10.4230/LIPIcs.ICDT.2023.7

[33] Matthias Niewerth. 2018. MSO Queries on Trees: Enumerating Answers under Updates Using Forest Algebras. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. ACM, 769–778. https://doi.org/10.1145/3209108.3209144

[34] W. Rytter. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science* 302, 1–3 (2003), 211–222. https://doi.org/10.1016/S0304-3975(02)00777-6

[35] Markus L. Schmid and Nicole Schweikardt. 2021. Spanner Evaluation over SLP-Compressed Documents. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2021*. ACM, 153–165. https://doi.org/10.1145/3452021.3458325

[36] Markus L. Schmid and Nicole Schweikardt. 2022. Query Evaluation over SLP-Represented Document Databases with Complex Document Editing. In *Proceedings of 41st Symposium on Principles of Database Systems, PODS 2022*. ACM, 79–89. https://doi.org/10.1145/3517804.3524158

Received June 2023; revised August 2023; accepted September 2023