MSO Queries on Trees: Enumerating Answers under Updates Using Forest Algebras*

Matthias Niewerth
Bayreuth University

Abstract

We investigate efficient enumeration of answers to MSO-definable queries over trees which are subject to local updates. We exhibit an algorithm that uses an $O(n)$ preprocessing phase and enumerates answers with $O(\log(n))$ delay between them. When the tree is updated, the algorithm can avoid repeating expensive preprocessing and restart the enumeration phase within $O(\log(n))$ time. This improves over previous results that require $O(\log^2(n))$ time after updates and have $O(\log^2(n))$ delay. Our algorithms and complexity results in the paper are presented in terms of node-selecting tree automata representing the MSO queries. To present our algorithm, we introduce a balancing scheme for parse trees of forest algebra formulas that is of its own interest to lift results from strings to trees.

ACM Reference Format:
Matthias Niewerth. 2018. MSO Queries on Trees: Enumerating Answers under Updates Using Forest Algebras. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209144

1 Introduction

Efficient query evaluation is one of the most central problems in databases. Given a query Q and a database D, we are asked to compute the set $Q(D)$ of tuples of Q on D. In general, the number of tuples in $Q(D)$ can be extremely large: when Q has arity k and D has size n, then $Q(D)$ can contain up to n^k tuples. Since databases are typically very large, it may be unfeasible to compute $Q(D)$ in its entirety.

A straightforward solution to this problem is top-k query answering, where one is interested in the k most relevant answers according to some metric. Another way to deal with this problem is to produce the answers one by one without repetition. This is known as *query enumeration* (see, e.g., [4, 17, 19, 22, 23, 31]). More precisely, query enumeration aims at producing a small number of answers first and then, on demand, producing further small batches of answers as long as the user desires or until all answers are depleted. Most existing algorithms for query enumeration consist of two phases: the *preprocessing phase*, which lasts until the first answer is produced, and the *enumeration phase* in which next answers are produced without repetition. It is natural to try to optimize two kinds of time intervals in this procedure: the time of the preprocessing phase and the *delay* between answers, which is the time required between two answers in the enumeration phase. Thus, when one can answer $Q(D)$ with preprocessing time p and delay d, one can compute $Q(D)$ in time $p + d \cdot |Q(D)|$, where $|Q(D)|$ is the number of answers.

Much attention has been given to finding algorithms that answer queries with a linear-time preprocessing phase and constant-time delay between answers[31]. The preprocessing phase is usually used to build an index that allows for efficient enumeration. Since databases can be subjected to frequent updates and preprocessing typically costs linear time, it is usually not an option to recompute the index after every update. We want to address this concern and investigate what can be done if one wants to deal with such updates more efficiently than simply re-starting the preprocessing phase.

We study the enumeration problem for MSO queries with free node variables over *trees*. Furthermore, the trees can be subjected to local updates. We consider updates that relabel a node or insert/delete a leaf. Our aim is to provide an index structure that can be efficiently updated, when the underlying tree changes. This makes the enumeration phase *insensitive to such updates*: when our algorithm is producing answers with a small delay in the enumeration phase and the underlying data D is updated, we can update the index and re-start enumerating on the new data within the same delay.

There are algorithms that can enumerate certain classes of conjunctive queries with constant delay and sublinear update time [7, 8]. Similarly, there are algorithms that can enumerate FO+MOD queries with constant delay and constant update time on bounded degree databases [9]. However, to the best of our knowledge, there are no such algorithms for enumeration of MSO queries. That is all existing constant delay solutions for MSO query enumeration on strings and trees have the drawback that they are static: Whenever the underlying data D changes, one needs to restart the preprocessing phase before answers can be enumerated again. Only very recently, there was a solution, that allowed for relabeling updates [3]. It is yet unclear, whether this approach can be extended to structural updates of the tree.

The complexity results in this article are presented in terms of the size of the tree; the arity k of the query; and the number $|Q|$ of states of a non-deterministic node-selecting finite tree automaton for the query. The connection between run-based node-selecting automata and MSO-queries is well known, see, e.g. [28, 33].

When measuring complexity in terms of query size, we have to keep in mind that MSO queries can be non-elementarily smaller than their equivalent non-deterministic node-selecting tree automata. Therefore, our enumeration algorithm is non-elementary in terms of the MSO formula, which cannot be avoided unless P =

*Supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe)

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.
LICS '18, July 9–12, 2018, Oxford, United Kingdom
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209144

Update	Delay	Remarks	Reference
$O(\log^2(n))$	–	only Boolean queries; $O(\log(n))$ on strings; uses heavy path decomp.	[5] Balmin et al. 2004
$O(\log^2(n))$	–	Boolean XPath queries	[10] Björklund et al. 2010
–	$O(1)$	updates in $O(n)$ by recomputation	[4] Bagan 2006
–	$O(1)$	different proof using decomposition forests	[23] Kazana and Segoufin 2013
–	$O(1)$	different proof using circuits	[2] Amarilli et al. 2017
$O(\log^2(n))$	$O(\log^2(n))$	complexities drop to $O(\log(n))$ on strings; uses heavy path decomp.	[24] Losemann and Martens 2014
$O(\log(n))$	$O(1)$	only works on strings; huge constants	[29] Niewerth and Segoufin 2018
$O(\log(n))$	$O(1)$	only relabel updates; uses circuits and tree decompositions	[3] Amarilli et al. 2018
$O(\log(n))$	$O(\log(n))$	uses forest algebras	this work

Table 1. Data complexity of existing solutions. Preprocessing time is always in $O(n)$.

NP [20]. For this reason, MSO is usually not used as a query language in practice; although it is widely regarded as a good yardstick for expressiveness.

Our complexities are exponential in the arity k of the queries. However for practical scenarios, k is usually very small. We note that $k = 2$ suffices for modelling XPath queries, which are central in XML querying.

Although we do not obtain constant-delay algorithms as in previous work on static trees, we can prove that, in the dynamic setting $O(\log(n))$ delay is possible. This means that, after receiving an update, we do not need to restart the $O(n)$ preprocessing phase but only require $O(\log(n))$ time to produce the first answer on the updated tree and continue enumerating from there. We allow updates to arrive at any time: If an update arrives during the enumeration phase, we immediately start the enumeration phase for the new structure.

Previous Results on MSO Queries on Trees We have collected previous results on evaluation and enumeration of MSO queries on strings and trees in Table 1.

For MSO sentences, this problem has been studied by Balmin, Papakonstantinou, and Vianu [5]. Balmin et al. show how one can efficiently maintain satisfaction of a finite tree automaton (and therefore, an MSO property) on a tree t which is subjected to updates. More precisely, when an update transforms t to t', they want to be able to decide very quickly after the update whether t' is accepted by the automaton. Taking n as the size of t, they show that, using a one-time preprocessing phase of time $O(n)$ to construct an auxiliary data structure, one can always decide within time $O(\log^2(n))$ after the update whether t' is accepted. The delay between answers is irrelevant in the setting of Balmin et al. since their queries always have a Boolean answer. Björklund et al. show a similar result for XPath queries, which are less expressive than MSO but can be exponentially more succint than tree automata, which leads to better constants. Losemann and Martens [24] extended the work of Balmin et al. to enumeration of k-ary queries under updates with $O(\log^2(n))$ delay and update time. Our goal is to improve the delay and update time to $O(\log(n))$.

The enumeration problem of static trees was studied by Bagan [4], who showed that (fixed) monadic second-order (MSO) queries can be evaluated with linear time preprocessing and constant delay over structures of bounded tree-width. Independently, another constant delay algorithm (but with $O(n\log(n))$ preprocessing time) was obtained by Courcelle et al. [17]. Recently, Kazana and Segoufin [23] provided an alternative proof of Bagan's result based on a deterministic factorization forest theorem by Colcombet [16], which is itself based on a result of Simon [32]. Such (deterministic) factorization forests provide a good divide-and-conquer strategy for words and trees, but it is unclear how they can be maintained under updates. It seems that they would have to be recomputed entirely after an update which is too expensive for our purposes.

With exception of [4], which presents an algorithm that is cubic in terms of the tree automaton, these papers present complexities in terms of the size of the trees only, that is, they consider the MSO formula to be constant. To the best of our knowledge, the data structures in these approaches cannot be updated efficiently if the underlying tree is updated. An overview of enumeration algorithms with constant delay was given in [31].

Heavy Path Decomposition vs. Forest Algebras A main idea in [5] is a decomposition of trees into heavy paths which allows one to decompose the problem for trees into $O(\log(n))$ similar problems on words, for which a solution was given by Patnaik and Immerman in [30]. This allows to solve the incremental evaluation problem with $O(n)$ preprocessing time and $O(\log^2(n))$ update time, where one log factor stems from the heavy path decomposition and the other from solving the problem over strings using monoids of finite string automata.

The approach of [5] was later extended to enumeration of MSO queries by Losemann and Martens [24]. They tweaked the monoid to contain additional information needed to find the symbols that appear in query results, which allows logarithmic delay and update time. Just as Balmin et al., Losemann and Martens use heavy path decomposition to lift the algorithm from words to trees resulting again in an additional logarithmic factor in the delay and update time.

We adapt the algorithm of Losemann and Martens from monoids to forest algebras to avoid the heavy path decomposition. This saves us a logarithmic factor compared to their results. We believe that the framework we introduce in Section 3 to compute and maintain forest algebra formulas with logarithmic height can be applied in other areas to lift results from strings to trees.

Tree Decomposition vs. Forest Algebras In [3], Amarilli et al. use tree decompositions [12] to convert arbitrary trees to trees with logarithmic height. Having a tree of logarithmic height is central in their algorithm to allow enumeration of MSO queries over trees under relabeling updates. Amarilli et al. say that the biggest obstacle in generalizing their work to allow structural updates

(insertion/deletion of nodes) of the tree is the unability to update the tree decomposition when the input tree changes. Replacing tree decompositions with the framework we develop in Section 3 solves this key problem. It is an easy exercise to rewrite existing MSO queries over trees to equivalent queries that take parse trees of forest algebra formulas as input.

Thus, the present paper could be an important step towards reaching the goal and combining the best results on enumeration (constant delay) with the best results on maintaining answers under updates (logarithmic update time).

Further Related Work There are implementations for and experimental results on incremental evaluation of XML documents wrt. DTDs [6] and regular expressions with counters on strings [11].

The query evaluation problem has also been studied from a descriptive complexity point of view, e.g., for conjunctive queries [34] and the reachability query on graphs [18].

Structure of the Paper In the next section, we provide the formal background, trees, tree automata, and forest algebras. In Section 3, we give a framework that allows the representation of trees by forest algebra formulas whose parse trees have logarithmic height. We show how to update these formulas while maintaining the height bound in a similar way as AVL trees do. In Section 4, we show how the framework of Section 3 can be applied to incremental evaluation. This section features the description of transition algebras for stepwise tree automata that might be of its own value. In Sections 5 and 6 we present a highlevel description and the technical details of our enumeration algorithm. The datastructure described in Section 6 uses an extension of transition algebras as described in Section 4. We conclude in Section 7.

2 Definitions

Trees, Forests, Contexts *Trees* in this paper are labeled and rooted and the children of each node are ordered. For every tree t, we denote the *set of nodes of t* by $\text{Nodes}(t)$ and the *number of nodes* (or the *size*) of t by $|t|$. Nodes in trees which have no children are called *leaves*. The (unique) Σ-label of node v is denoted by $\text{lab}(v)$. A *forest* is an ordered list of trees. A *context* is like a forest, with the difference that exactly one leaf is a hole, denoted by the special label $\square \notin \Sigma$. The *height* of a forest or context is the length of the longest path from a root to a leaf.

Given a context c and a forest f, we denote by $c \odot f$ the context application of c on f. The resulting forest $c \odot f$ is derived from c by replacing the hole with all roots of f. Context application of c on another context c' is defined likewise, with the difference, that the result is a context, as $c \odot c'$ has a hole, that results from the hole in c'.

We depict an example for the context application of a context on a forest in Figure 1.

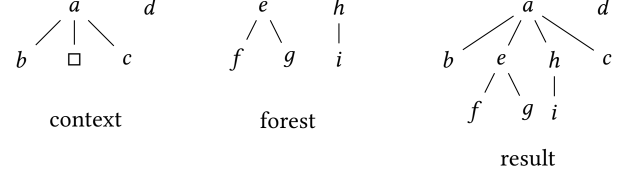


Figure 1. Example for context application

Stepwise Automata and Runs Stepwise tree automata were first described in [15] using a curry encoding of unranked trees. For convenience, we use the definition from [25] that directly works on unranked trees.

A *stepwise nondeterministic tree automaton* or *NFTA* is a tuple $N = (Q, \Sigma, \delta, \text{Init}, F)$ where Q is the finite set of states, Σ is a finite alphabet, $F \subseteq Q$ is the set of accepting states, $\text{Init}\colon \Sigma \to 2^Q$ assigns a set of initial states to every symbol of Σ, and $\delta\colon Q \times Q \to 2^Q$ is a transition function. By $\delta^*\colon Q \times Q^* \to 2^Q$ we denote the extension of δ to strings of states.

Intuitively, a stepwise tree automaton computes a run bottom-up. After assigning states $q_1, \ldots, q_n$ to the n children of some node v, it assigns a state to v, by starting in some initial state (determined by the label of v) and reading the string $q_1 \ldots q_n$. Whether a run is accepting is determined by the state of the root.

A *run* of N on a labeled tree t is an assignment of nodes to states $\lambda\colon \text{Nodes}(t) \to Q$ such that for every node v with children $v_1, \ldots, v_k$, we have $\lambda(v) \in \delta^*(q, \lambda(v_1) \ldots \lambda(v_k))$, where $q \in \text{Init}(\text{lab}(v))$. A run is *accepting* if $\lambda(r) \in F$, where r is the root of t. A tree t is *accepted* if there exists an accepting run on t. The set of all accepted trees is denoted by $L(N)$. By $\text{states}(\lambda)$ we denote the image of λ, i.e., the set of states visited by the run.

Convention 1. *Given any NFTA N, we will assume w.l.o.g. that there are special states $q_0, q_F \in Q$, such that the transitions using q_0 and q_F are exactly $\{(q_0, q, q_F) \mid q \in F\}$.*

Convention 1 is equivalent to change the mode of acceptance as follows: A run is accepting, if and only if $q_F \in \delta^*(q_0, \lambda(\text{root}(t)))$. This mode of acceptance is more similar to string automata and simplifies definitions of transition algebras later on. The convention can be easily enforced on a given NFTA by adding q_0, q_F and transitions $\{(q_0, q, q_F) \mid q \in F\}$.

Runs over forests and contexts f are defined like runs over trees with the exception that no state is assigned to the hole and the parent v of the hole can have any state q such that there exists a forest f' and run λ' over $f \odot f'$ such that $\lambda'(v) = q$.

Signatures of Runs A *signature* of a run λ over a forest with roots $v_1, \ldots, v_k$ is a pair of states (q_1, q_2) such that

$$q_2 \in \delta^*(q_1, \lambda(v_1)\lambda(v_2) \cdots \lambda(v_k)) .$$

If the run λ is over a context, then a signature is a pair of pairs of states $((q_1, q_2), (q_3, q_4))$ satisfying the following conditions that stem from the intuition that replacing the hole with a forest that has a run with signature (q_3, q_4) yields a run over the resulting forest with signature (q_1, q_2).

If the hole u_i is not a root, we let $u_1, \ldots, u_{k'}$ be the children of $\text{Parent}(u_i)$. The signature has to satisfy

$$\begin{aligned} q_2 &\in \delta^*\big(q_1, \lambda(v_1) \cdots \lambda(v_k)\big) \\ q_3 &\in \delta^*\big(p, \lambda(u_1) \cdots \lambda(u_{i-1})\big) \\ \lambda(\text{Parent}(u_i)) &\in \delta^*\big(q_4, \lambda(u_{i+1}) \cdots \lambda(u_{k'})\big) \end{aligned}$$

for some p with $p \in \text{Init}(\text{lab}(\text{Parent}(u_i)))$

If the hole v_i is a root, the signature has to satisfy

$$\begin{aligned} q_3 &\in \delta^*\big(q_1, \lambda(v_1) \cdots \lambda(v_{i-1})\big) \\ q_2 &\in \delta^*\big(q_4, \lambda(v_{i+1}) \cdots \lambda(v_k)\big) \end{aligned}$$

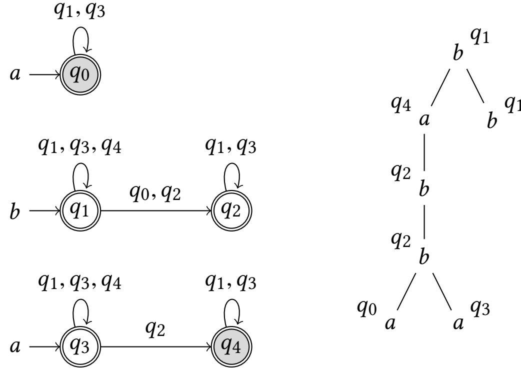


Figure 2. 2-NFSTA M with $S = \{(q_0, q_4)\}$ (left) and tree with accepting run (right).

While a run over a forest can have different signatures only if the automaton is non-deterministic, a run over a context usually has many different signatures, even if the automaton is deterministic, as the pair of states that is the "signature of the hole" is not fixed.

We denote the set of all possible signatures over the states Q with $\text{SIG}_Q = Q^2 \cup (Q^2)^2$. Whenever a run λ has signature $x \in \text{SIG}_Q$ in the automaton N, we denote this by $\lambda \models_N x$.

Selecting Automata We use (node- and tuple-) selecting finite tree automata (see, e.g., [21, 27]) as formalism for queries. It is well-known that these can express MSO queries with free node variables over unranked trees [28, Theorem 7].

For $k \in \mathbb{N}$, a *k-ary non-deterministic finite selecting tree automaton (k-NFSTA) M* is a pair (N, S), where N is a NFTA over Σ with states Q and $S \subseteq Q^k$ is a set of *selecting tuples*. The *size* of M is defined as $|Q| + |S|$. When M reads a tree t, it computes a set of tuples in $\text{Nodes}(t)^k$. More precisely, we define

$$M(t) = \Big\{ (v_1, \ldots, v_k) \;\Big|\; \text{there is an accepting run } \lambda \text{ of } N \text{ on } t \text{ and a tuple } (p_1, \ldots, p_k) \in S \text{ such that } \lambda(v_\ell) = p_\ell \text{ for } \ell \in \{1, \ldots, k\} \Big\} .$$

Notice that, if $t \notin L(N)$ then $M(t) = \emptyset$.

Example 1. Figure 2 illustrates a 2-NFSTA M over $\Sigma = \{a, b\}$ that outputs each pair of a-labeled nodes that are connected by a path of b-labeled nodes. The automaton consists of three parts. The first part guesses the start node of some ab^+a-path. The second part checks for each b-node whether it is—directly or by a b-path—connected to this a-node. The third part checks whether some a-node is the end of such a path. The automaton has two initial states for a-nodes and one initial state for b-nodes. We note that all parts check—by lack of q_0-, q_2-, and q_4-transitions—that at most one a-node uses the first part of the automaton. All states are accepting, because a lack of ab^+a-paths is already detected by not being able to create a run which contains a q_4-state and thus the only selecting tuple would not be matched.

Next to the automaton we depict a tree with some accepting run that returns the a-node below the root and the left a-leaf. A symmetric run returns a pair of nodes with the other a-leaf.

Forest Algebras Here, we introduce forest algebras that were first described by Bojańczyk and Walukiewicz [14]. We prefer the syntax used in the Handbook of Automata Theory [13] that also provides a nice introduction.

A *forest algebra*

$$(H, V, \odot_{VH}, \oplus_{HV}, \oplus_{VH})$$

consists of two monoids, $H = (H, \oplus_{HH}, \varepsilon)$ and $V = (V, \odot_{VV}, \square)$ along with three monodial actions: $\oplus_{HV} : H \times V \to V$, $\oplus_{VH} : V \times H \to V$, and $\odot_{VH} : V \times H \to H$.

Intuitively, each element of H represents a forest and each element in V represents a context. The monoid operations correspond to concatenation of forests and context application (on a context), respectively. The neutral elements of H and V correspond to the empty forest and empty context. The monodial actions correspond to concatenation of a forest and a context (or the other way round) and context application of a context on a forest.

As $\oplus_{HV}$, $\oplus_{VH}$, and $\odot_{VH}$ are monodial actions, the following hold for any $f_1, f_2 \in H$ and $c_1, c_2 \in V$.

$$\begin{aligned}
(f_1 \oplus_{HH} f_2) \oplus_{HV} c_1 &= f_1 \oplus_{HV} (f_2 \oplus_{HV} c_1) \\
c_1 \oplus_{VH} (f_1 \oplus_{HH} f_2) &= (c_1 \oplus_{VH} f_1) \oplus_{VH} f_2 \\
(c_1 \odot_{VV} c_2) \odot_{VH} f_1 &= c_1 \odot_{VH} (c_2 \odot_{VH} f_1) \\
(f_1 \oplus_{HV} c_1) \oplus_{VH} f_2 &= f_1 \oplus_{HV} (c_1 \oplus_{VH} f_2) \\
\varepsilon \oplus_{HV} c_1 &= c_1 \\
c_1 \oplus_{VH} \varepsilon &= c_1 \\
\square \odot_{VH} f_1 &= f_1
\end{aligned}$$

We use f and c (possibly with indices), to denote forests and contexts, respectively. Whenever it is not clear whether we refer to a forest or a context, we use d, which is between c and f. We use v and w to denote nodes of either a given tree t, or a parse tree of a forest algebra formula Ψ. To keep the notation clean, we identify leaves of Ψ with nodes of t, whenever we have a formula Ψ describing a tree t. Given a node v of the parse tree of Ψ, we use Ψ_v to denote the subformula of Ψ rooted at v and t_v to denote the forest or context described by Ψ_v.

We will often drop the indices of the monoid operations and monodial actions, i.e., we will just use $\oplus$ and $\odot$. Which operation is needed is clear from the operands. In some cases, we do not even specify, whether we refer to concatenation or context application. In this case, we use $\bigcirc$. Given a formula Ψ and an inner node v of the parse tree, we denote by $\bigcirc_v$ the operation at node v.

The *free forest algebra* over an alphabet Σ is defined as $\mathrm{T}_\Sigma = (H, V)$, where H are all forests and V are all contexts. The necessary monoid operations and monodial actions are given by forest concatenation and context application. For every symbol a of Σ, we denote with a_t the tree consisting only of an a-labeled root and by $a_\square$ the context consisting of an a-labeled root having the hole as its only child.

The *size* of the forest algebra formula is the number of nodes in its parse tree.

A homomorphism $\mathrm{h} = (\mathrm{h}_H, \mathrm{h}_V)$ from a forest algebra $\mathrm{T}_1 = (H_1, V_1)$ to a forest algebra $\mathrm{T}_2 = (H_2, V_2)$ is given by two monoid morphisms $\mathrm{h}_H : H_1 \to H_2$, $\mathrm{h}_V : V_1 \to V_2$ that additionally satisfy

$$\mathrm{h}_H(c \odot f) = \mathrm{h}_V(c) \odot \mathrm{h}_H(f)$$

for all $c \in V_1$ and $f \in H_1$. To simplify notation, we will omit the indices H and V and use h for both morphisms.

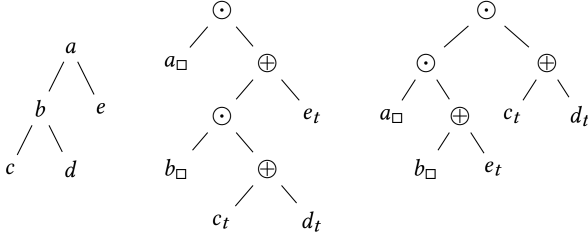


Figure 3. Tree t and the parse trees of two different formulas Ψ_1 and Ψ_2 representing t

A homomorphism h from the free forest algebra to some forest algebra $\mathrm{T} = (H, V)$, is uniquely defined by the mappings from the set $\{a_\square \mid a \in \Sigma\}$, as all other mappings are implied, e.g., $\mathrm{h}(a_t) = \mathrm{h}(a_\square) \odot \varepsilon$.

For every forest algebra $\mathrm{T} = (H, V)$, we will always have an input homomorphism h from the free forest algebra to T.

The *parse tree* of a forest algebra formula (over an algebra $T = (V, H)$) is a binary tree with inner nodes marked by $\oplus$ and $\odot$ and leaf nodes marked by some element of $V \cup H$. Whether a $\oplus$ node corresponds to the concatenation of two forests or the concatenation of a forest with a context is clear from the operands. Similarly it is clear, whether a $\odot$ node corresponds to the context application between two contexts or between a context and a forest.

The *balance factor* $\mathrm{B}(v)$ of an inner node v of the parse tree is the height difference of the two trees, rooted at the children of v. Positive values for $\mathrm{B}(v)$ denote that the right subtree is higher than the left subtree and vice versa.

In this paper, we will never consider parse trees that use nodes that correspond to the empty forest or empty context. Therefore, parse trees of forest algebra formulas will have exactly one leaf for every node of the represented forest or context. As parse trees are always binary trees, they will have one inner node less than leaves. Therefore, a parse tree is always of roughly twice the size than the represented forest or context.

In Figure 3 we depict a tree t, and the parse trees of two different formulas encoding t.

To simplify notation, we identify forest algebra formulas with their parse trees and leaves of parse trees with nodes of the represented forest or context. Whenever we have a parse tree Ψ, we assume that for any node v, the algebra element $\mathrm{h}(t_v)$ is stored together with the node, where h is the input homomorphism. Especially we assume that given v, the element $\mathrm{h}(t_v)$ is available in constant time.

Incremental Evaluation and Enumeration Let M be a selecting automaton, t the input tree for M, and $M(t)$ be the answer of M on t. We are interested in efficiently maintaining $M(t)$ under updates of t. This means that we can have an update u to t, yielding another tree t', and we wish to efficiently compute $M(t')$. The latter cost should be more efficient than computing $M(t')$ from scratch. We consider the following *updates* on trees: (i) Replace the current label of a specified node by another label, (ii) insert a new node as only child of a specified node making all existing children of the existing node children of the new node, (iii) insert a new leaf node as left or right sibling of an existing node, and (iv) delete a specified leaf node

We allow a single preprocessing phase in which we can compute an *auxiliary data structure* $\mathrm{Aux}(t)$ that we can use for efficient query answering. When t is updated to t', we therefore want to efficiently compute $M(t')$ and efficiently update $\mathrm{Aux}(t)$ to $\mathrm{Aux}(t')$.

If M is simply an NFTA (i.e., a 0-ary NFSTA), then this problem is known as *incremental evaluation* and was studied by, e.g., [5]. Here, we perform *incremental enumeration*, meaning that we extend the setting of Balmin et al. from 0-ary queries to k-ary queries. We measure the complexity of our algorithms in terms of the following parameters: *(i)* size of $\mathrm{Aux}(t)$, *(ii)* time needed to compute $\mathrm{Aux}(t)$, *(iii)* time needed to update $\mathrm{Aux}(t)$ to $\mathrm{Aux}(t')$, and *(iv)* time delay we can guarantee between answers of $M(t')$. The underlying model of computation is a random access machine (RAM) with uniform cost measure.

In the remainder we use IncEval and IncEnum to refer to the incremental evaluation and enumeration problems, respectively.

3 Maintaining Parse Trees under Updates

For our evaluation and enumeration algorithms, we need a data structure that can represent a tree t by a forest algebra formula Ψ (from some algebra T). The formula should have a parse tree of logarithmic height, updates of the tree should require only logarithmic time to update the formula, and for each node v of the parse tree, the corresponding element $\mathrm{h}(t_v)$ of the algebra should be available in constant time.

To keep the parse tree of Ψ shallow, we use similar rotations, as used in AVL trees [1]. Unfortunately, the well known rotations used to balance AVL trees only work, if the underlying algebra is fully associative, which does not hold for forest algebras, as, e.g., $c \odot (f_1 \oplus f_2) \neq (c \odot f_1) \oplus f_2$. Therefore, we provide additional rotations that can be used where the traditional rotations fail.

The main result of this section is:

Proposition 2. *Given a tree t and a forest algebra T, it is possible to compute in time $O(c|t|)$ a forest algebra formula Ψ representing t, such that*

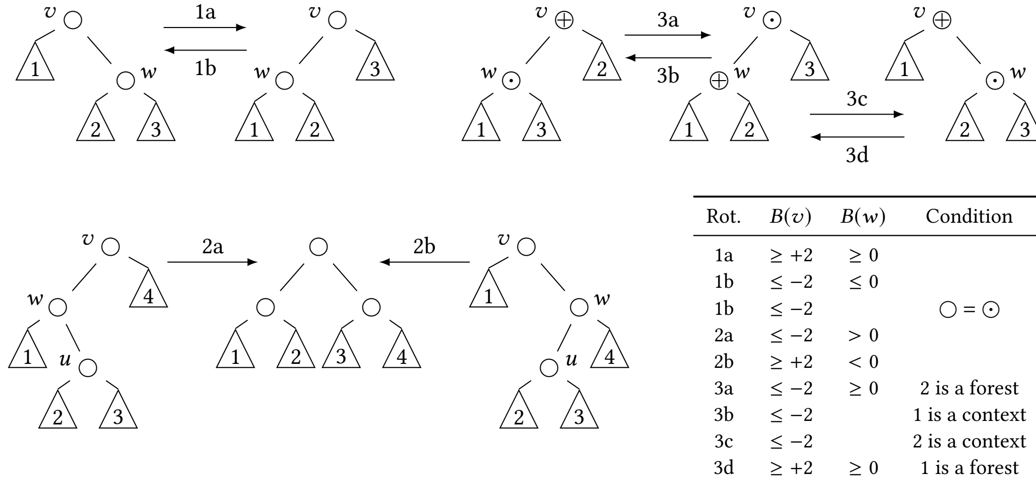
- *the parse tree of Ψ is of height at most $8\log(|t|)$; and*
- *each update of t can be translated to an update of Ψ, such that the new formula can be computed in time $O(\log(c|t|))$ and has height at most $8\log(|t|)$,*

where c is the worst-case computation time of one forest algebra operation.

We note that the coefficient c in the running times purely stems from computing and updating the annotations of the inner nodes. The remainder of the section is devoted to prove the proposition.

A node v in Ψ is *balanced* if $\mathrm{B}(v) \in \{-1, 0, 1\}$. Otherwise, v is *unbalanced*. A path going from v to a leaf w is called a *long path* of v, if it is a maximal length path among all paths leaving v.

The proof of Proposition 2 now goes as follows: We introduce rotations that can be used to balance parse trees (Figure 4), show that the rotations are sound, i.e., they preserve equivalence of the formula (Lemma 3), and that each formula where none of the rotations can be applied has at most logarithmic height (Lemmas 5 and 6). At last we show that formulas can be updated in logarithmic time (Lemma 7) and preprocessed in linear time.



Rot.	$B(v)$	$B(w)$	Condition
1a	$\geq +2$	≥ 0	
1b	≤ -2	≤ 0	
1b	≤ -2		$\bigcirc = \odot$
2a	≤ -2	> 0	
2b	$\geq +2$	< 0	
3a	≤ -2	≥ 0	2 is a forest
3b	≤ -2		1 is a context
3c	≤ -2		2 is a context
3d	$\geq +2$	≥ 0	1 is a forest

Figure 4. Rotations of Forest Algebra Formulas

We depict all rotations[1] needed to balance parse trees of forest algebra formulas in Figure 4. Embedded in the figure is a table that lists the preconditions of each rotation. For the classic AVL rotations (i.e. rotations 1a, 1b, 2a, and 2b) to work, all $\bigcirc$-nodes need to be of the same kind, i.e., either all of them are $\oplus$-nodes or all of them are $\odot$-nodes. To which exact operations these nodes refer (e.g., whether a $\odot$-node refers to $\odot_{VV}$ or $\odot_{VH}$), does not matter, as operations of the same kind are associative in forest algebras.

Conditions over balance factors are given to ensure that application of a rotation increases balancedness, while conditions that enforce that a subformula describes a forest/context are necessary to ensure that the rotation can be applied, as, e.g., it is not possible to concatenate two contexts.

The rotation 1b is special, when applied to a $\odot$-node. Likewise are the rotations 3b and 3c. All these rotations have in common that they can change the balance factor of the node at the top from -2 to $+2$, which at the first sight does not help to balance the formula. However, as the context application is not symmetric, we have a problem of balancing $\odot$-nodes that have a negative balance factor. Therefore, we strictly prefer positive balance factors over negative ones for $\odot$-nodes.

All other rotations strictly decrease the absolute value of the balance factor of the top node, due to the requirement on the balance factor of w. The same holds true for rotation 1b, when applied to a $\oplus$-node.

Lemma 3. *The rotations depicted in Figure 4 are sound, i.e., applying one operation, whose preconditions are satisfied, to some subformula Ψ_v yields an equivalent formula.*

Proof sketch. It is tedious but straightforward to verify the statement for all operations using the axioms of forest algebras, including associativity of the two monoids. Care must be taken to take into account that $\oplus$ and $\odot$ refer to different operations according to the operands. □

[1] Technically not all rewritings in the figure are rotations. We stick with this term, as it is established for rewritings that rebalance a tree.

We now define balanced formulas. We note that we cannot avoid unbalanced nodes altogether, as e.g., we have no rotation that we can apply if some node v with $B(v) \geq 2$ is a $\odot$ node with the right child being a $\oplus$-node. The definition of a balanced formula is reverse engineered from the proof of Lemma 6, while ensuring logarithmic height at the same time.

Definition 4 (Balanced Formula)**.** A formula Ψ is *balanced*, if for each unbalanced node v of Ψ it holds that there is a balanced node w at most 6 levels below v on a long path of v.

The following height bound of balanced formulas can be shown using recurrence equations in a similar way as height bounds for AVL trees can be shown.

Lemma 5. *A balanced formula Ψ with n nodes has a parse tree of height at most $8\log(n)$.*

Lemma 6. *Let Ψ be a forest algebra formula, such that no rotation from Figure 4 can be applied. Then the parse tree of Ψ is balanced.*

Proof. Assume in contradiction that no rotation can be applied and there is a long path $v_1, \ldots, v_7, \ldots$, such that the nodes v_1 to v_7 are unbalanced.[2]

We will now show the following properties of such a path:

(a) Every $\odot$-node v_i has $B(v_i) \geq 2$.
(b) There are no two consecutive $\odot$-nodes.
(c) There are no three consecutive $\oplus$-nodes.
(d) There is at most one $\oplus$-node followed directly by a $\odot$-node.

The lemma statement then follows from (b) to (d), as the longest possible path ($\odot$–$\oplus$–$\oplus$–$\odot$–$\oplus$–$\oplus$) fulfilling (b) to (d) has 6 nodes. It remains to show (a) to (d).

If (a) does not hold, we can apply either 1b, 3b, or 3c, as the left child is either a $\odot$-node or a $\oplus$-node. In the latter case, one of the children of the $\oplus$-node needs to describe a context, because v_i is a $\odot$-node.

If (b) does not hold, we can apply 1a, as by (a) all $\odot$-nodes on the path have $B(v_i) \geq 2$, i.e, the second consecutive $\odot$-node is right

[2] In this case, every long path of v_1 uses the nodes $v_2, \ldots, v_7$, as the next node on a long path is always in the direction indicated by the balance factor and we assume all nodes v_i to be unbalanced.

child of the first one and satisfies the condition on $B(w)$ to apply 1a. Similarly, if (c) does not hold, we can apply one of 1a, 1b, 2a and 2b.

It remains to show (d). If v_i is a $\oplus$-node and v_{i+1} is a $\odot$-node, we can conclude that t_{v_i} is a context and $t_{v_{i+1}}$ is a forest. Otherwise 3a or 3d could be applied. We remind that at most one child of a $\oplus$-node can be a context. However, if $t_{v_{i+1}}$ is a forest, then all t_{v_j} with $j > i$ are forests, as the path always goes to the right child of each $\odot$-node by (a). Therefore, there can be no other $\oplus$-node that has a $\odot$-child. We can conclude (d) and the proof. □

Towards Proposition 2, it remains to show two things, how to maintain balancedness under updates and how to compute a balanced parse tree in the first place.

Lemma 7. *There exists an algorithm that takes as input a balanced forest algebra formula Ψ for a tree t and an update of the kinds (i) to (iv) with the following properties:*

- *The runtime is $O(c \cdot \text{height}(t))$.*
- *The resulting formula represents the updated tree t' and has height at most $8\log(|t'|)$.*
- *The height bound is maintained under repeated updates.*

Proof sketch. Our update algorithm works as follows:

1. Do the actual update by some local change at some leaf u of Ψ.
2. Do a bottom up pass from u to the root and for each node v
   - if a rotation is possible[3] at v: do the rotation
   - if a rotation is possible at the sibling of v: do the rotation
3. Recompute the node annotations by a bottom up pass from u to the root. Also recompute the node annotations for nodes that were changed in step 2.

It is straightforward to verify, that each of the updates (i) to (iv) can indeed be achieved by a local change at some leaf of the formula, e.g., adding a new leaf with label a as a sibling of some existing node v can be done by replacing v with $a_t \oplus v$. The correctness of the algorithm follows from Lemma 3.

The runtime follows from the fact that each rotation takes constant time, each algebra computation takes time c, and there are at most two rotations in each level of the parse tree.

The trickiest part is to show that the algorithm maintains an upper bound of $8\log(|\Psi|)$ on the height of Ψ. This can be achieved by adding some invariants and doing induction over the update sequence. □

One might wonder why we do not always apply all possible rotations after an update. This would trivially maintain the height bound by Lemma 5. Unfortunately, there are some bad update sequences, where one update allows for logsquare many rotations afterwards. The underlying reason is that there might be some unbalanced nodes v in the parse tree with $|B(v)| \approx \text{height}(\Psi)$. After an update it might be possible to apply logarithmically many rotations at v. This could happen on several nodes on the bottom up pass. Therefore we only apply at most two rotations at each level, which keeps the update time low and still is enough to maintain the $8\log(\text{height}(\Psi))$ height bound.

The data structure can be initialized in linear time by starting with the formula representing the root of a given tree t and successively inserting the nodes of t. While the naïve analysis yields runtime $O(n \cdot \log(n))$, a more involved amortized analysis yields linear time, as the amortized cost of an insertion is $O(1)$ not counting navigation, especially the amortized number of rotations after each update is constant. We note, that we need i rebalancing operations approximately every 2^i insertion operations. Navigation is in $O(1)$ for each insertion operation if we keep a pointer to the last inserted node. The amortized cost of insertions for AVL trees has been analyzed in [26]. Compared to an AVL tree, we need less rotations, as we do not require every node to be balanced.

[3] The conditions in Figure 4 are satisfied.

4 Incremental Evaluation

Towards incremental evaluation, we first define the *transition algebra* of a stepwise tree automaton. It is the generalization of the transition monoid of a finite string automaton. Intuitively, each element of the transition algebra captures the signatures of all possible runs over a forest or context. Our auxiliary data structure for incremental evaluation will be a balanced transition algebra formula representing the given tree.

Formally, the *transition algebra* of a given stepwise tree automaton $N = (\Sigma, Q, \delta, \text{Init}, F)$ is defined as $\mathrm{T} = (H, V, \odot_{VH}, \oplus_{HV}, \oplus_{VH})$ using $H = (2^{Q^2}, \oplus_{HH}, \text{id}_Q)$ as the horizontal monoid and $V = (2^{(Q^2)^2}, \odot_{VV}, \text{id}_{Q^2})$ as the vertical monoid. Here, id_X is the identity function over X. Both monoid operations are given by

$$d_1 \bigcirc d_2 = \{(x_1, x_3) \mid (x_1, x_2) \in d_1, (x_2, x_3) \in d_2\} ,$$

where x_1, x_2, x_3 are states in the case of the horizontal multiplication and pairs of states in the case of the vertical multiplication. The actions are defined by

$$\begin{aligned} c \odot_{VH} f &= \{(q_1, q_2) \mid ((q_1, q_2), (q_3, q_4)) \in c, (q_3, q_4) \in f\} \\ f \oplus_{HV} c &= \{((q_1, q_3), (q_4, q_5)) \mid \\ &\qquad (q_1, q_2) \in f, ((q_2, q_3), (q_4, q_5)) \in c\} \\ c \oplus_{VH} f &= \{((q_1, q_5), (q_3, q_4)) \mid \\ &\qquad ((q_1, q_2), (q_3, q_4)) \in c, (q_2, q_5) \in f\} \end{aligned}$$

We define the input homomorphism by

$$\mathrm{h}(a_\square) = \{((q_1, q_2), (q_3, q_4)) \mid q_3 \in \text{Init}(a), q_2 \in \delta(q_1, q_4)\}$$

for all symbols $a \in \Sigma$. It is straightforward to verify that T is indeed a forest algebra. Furthermore, we have the following observation:

Observation 8. *All operations in* T *can be performed in at most $O(|Q|^6)$ time using join operations, where the costliest operation is the multiplication in the vertical monoid.*

We note that the horizontal monoid of the transition algebra is defined just like the transition monoid for a string automaton over the alphabet Q. The intuition about the vertical monoid can be best seen by the definition of the action $\odot_{VH}$. If $((q_1, q_2), (q_3, q_4)) \in \mathrm{h}(c)$ for some context c, then applying c to some forest f with $(q_3, q_4) \in \mathrm{h}(f)$ results in a forest f' such that $(q_1, q_2) \in \mathrm{h}(f')$. This intuition is formalized in Lemma 9.

Lemma 9. *Let d be a forest or context. It holds that $x \in \mathrm{h}(d)$ if and only if there exists a run λ of N on d such that $\lambda \models_N x$.*

The lemma can be proven by a straightforward induction. Now we have all ingredients to show the main result of this section.

Theorem 10. *IncEval for an NFTA $N = (\Sigma, Q, \delta, \text{Init}, F)$ and a tree t can be solved with a preprocessing phase of time $O(|Q|^6 \cdot |t|)$, auxiliary structure of size $O(|Q|^4 \log(|t|))$, and with update time $O(|Q|^6 \log(|t|))$ after each new update.*

Algorithm 1 Enumeration of $M(t)$

Input: k-NFSTA $M = ((Q, \Sigma, \delta, F), S)$, tree t, incomplete answer A
Output: Enumeration of all answers in $M(t)$ that are compatible with A

```
1: function Enum(M, t, A)
2:     if |A| = k then Output(A)
3:     else
4:         A' ← Complete(A, ⊥)
5:         while A' ≠ ⊥ do
6:             Enum(M, t, A')
7:             v ← A'_{|A'|}
8:             A' ← Complete(A, v)
```

Proof. We use the framework of Section 3 to compute and maintain a balanced representation of t using the transition algebra of N. By Convention 1 and Lemma 9, the evaluation problem can be solved by looking whether (q_0, q_F) is contained in the forest algebra element represented at the root of the parse tree. The complexities follow from Observation 8 and Proposition 2. □

5 Highlevel Enumeration Algorithm

In this Section, we give a highlevel presentation of our enumeration algorithm that is depicted as Algorithm 1. The technical details including the auxiliary data structure and our main result are presented in the next section.

We assume a total order $\leq$ on the nodes of t that can depend on our auxiliary data structure. The algorithm then enumerates all answers in lexicographic order. To avoid some case distinctions, we assume a symbol $\perp$ such that $\perp \leq v$ for any node v.

To understand the algorithm, we need the notation of an incomplete answer: We call a tuple $A \in \text{Nodes}(t)^\ell$ with $\ell \leq k$ an *incomplete answer* if it is a prefix of some answer $B \in M(t)$. We assume that the empty tuple () is an incomplete answer, even if $M(t) = \emptyset$ to avoid some corner cases. We write $A \preceq B$ for two (in-)complete answers A and B, if A is a prefix of B. By $|A| := \ell$ we denote the number of nodes of the incomplete answer A.

To enumerate all answers, Enum has to be called with the empty answer (). The sub-procedure Complete extends a given incomplete answer A with another node according to the following definition.

Definition 11. Let $A = (v_1, v_2, \ldots, v_j)$ be an incomplete answer, then

$$\text{Complete}(A, u) := (v_1, v_2, \ldots, v_j, v) ,$$

where v is the smallest node such that $u < v$ and $(v_1, v_2, \ldots, v_j, v)$ is an incomplete answer. If no such node exists, then we define $\text{Complete}(A, u) := \perp$.

By definition of Complete, the lines 4 to 8 iterate over all incomplete answers A' that result from A by adding one additional node. Before we show how to efficiently implement Complete, we prove correctness of the highlevel enumeration algorithm.

Lemma 12. *Enum$(M, t, ())$ enumerates all answers in $M(t)$.*

Proof. We show that for every incomplete answer A, the function call Enum(M, t, A) outputs exactly the answers B such that A is a prefix of B. The lemma statement follows, as the empty answer () is a prefix of every answer.

The proof is by induction over $|A|$. The base case is $|A| = k$. In this case, the only compatible answer is A, which is output in Line 2 of the algorithm. Let now $A = (v_1, \ldots, v_\ell)$ be an incomplete answer and $B = (v_1, \ldots, v_\ell, v_{\ell+1}, \ldots, v_k)$ be some answer compatible with A. Eventually some call to Complete in Line 4 or 8 will return the incomplete answer $(v_1, \ldots, v_\ell, v_{\ell+1})$. By the induction hypotheses, the recursive call in Line 6 will output B. □

6 Technical Core

This section presents our implementation of Complete. Our auxiliary data structure is a balanced forest algebra formula Ψ that represents the tree t. We use the *extended transition algebra* that we define below.

Compared to the transition algebra it additionally contains some information about states visited in a run. Instead of adding (for each possible run) the set of used states, we only care about those subsets that are actually needed by some selecting tuple.

Let $M = (N, S)$ be a k-NFSTA. We define $\mathbb{S}(S)$ to be the set $\mathbb{S}(S) = \{Q' \subseteq Q \mid \exists s \in S : Q' \subseteq s\}$.

We define the *extended transition algebra*

$$\mathrm{T}^+ = (H^+, V^+, \oplus^+_{HV}, \oplus^+_{VH}, \odot^+_{VH})$$

using $H^+ = (2^{Q^2 \times \mathbb{S}(S)}, \oplus^+_{HH}, \text{id}_Q \times \{\emptyset\})$ as horizontal monoid and $V^+ = (2^{(Q^2)^2 \times \mathbb{S}(S)}, \odot^+_{VV}, \text{id}_{Q^2} \times \{\emptyset\})$ as vertical monoid.

We define the monoid operations and monodial actions by

$$d_1 \bigcirc^+ d_2 = \big\{(x, r) \in \text{SIG}_Q \times \mathbb{S}(S) \mid \exists (y_1, r_1) \in d_1, (y_2, r_2) \in d_2. \\ x \in \{y_1\} \bigcirc \{y_2\} \text{ and } r = r_1 \cup r_2\big\} ,$$

where $\bigcirc$ refers to the according operation in the normal transition algebra. The input homomorphism is given by

$$\mathrm{h}^+(a_\square) = \big\{\big(((q_1, q_2), (q_3, q_4)), r\big) \in (Q^2)^2 \times \mathbb{S}(S) \mid \\ ((q_1, q_2), (q_3, q_4)) \in \mathrm{h}(a_\square) \text{ and } r \subseteq \{q_4\}\big\}$$

In the case $k = 0$, T^+ is isomorphic to T, as $\mathbb{S}(S) = \{\emptyset\}$. We note that the horizontal monoid H works exactly, as illustrated by [24] in the word case. Especially, our definition of $\oplus$ is equivalent to the definition of $\bowtie$ in [24].

We call a tuple (x, r) from $\text{SIG}_Q \times \mathbb{S}(S)$ an extended signature. And use the syntax $\lambda \models_M (x, r)$ analogously to normal signatures.

Observation 13. *Given a k-NFSTA $M = (N, S)$, operations in T^+ can be carried out in time $O(|Q|^6 \cdot |S| \cdot 2^k)$ using join operations.*

The total order on the nodes of t, that we already introduced in the last section, is defined as follows: $v \leq w$ if and only if v occurs before w in the parse tree of Ψ, reading the leaves from left to right. We stress that the order $\leq$ depends on the formula Ψ. Especially, the order can change in non-obvious ways during insertion and deletion updates, as the structure of Ψ may change, due to rotations. We sketch, how to achieve enumeration in pre- or post-order at the end of this section.

We already know that elements of T^+ can be interpreted as sets of extended signatures of possible runs. However, not all runs of M on t_v (and thus not all signatures in $\mathrm{h}^+(t_v)$) are actually useful for completing an incomplete answer A. To be useful, an extended signature $(x, r) \in \mathrm{h}^+(v)$ has to satisfy two conditions: It has to be the signature of a run λ of t_v that

- is compatible with A, i.e., for the nodes in A and some selecting tuple s, it visits the correct states; and
- can be extended to some accepting run λ' over t.

Let now $A = (v_1, \ldots, v_\ell)$ be an incomplete answer and $Q_s = \{q_1^s, \ldots, q_k^s\}$ for each $s = (q_1^s, \ldots, q_k^s)$ in S. We write $\lambda \models_s A$, if $\lambda(v_i) = q_i^s$ for $i \in \{1, \ldots, \ell\}$.

Towards the above conditions, we define sets of relevant tuples of each node v of Ψ.

The sets $R^1_{A,s}$ account for the first condition, and are defined by

$$R^1_{A,s}(v) = \begin{cases} \mathrm{h}^+(t_v) \cap \mathrm{SIG}_Q \times \{\{q_i^s\}\} & \text{if } v = v_i \\ \mathrm{h}^+(t_v) \cap \mathrm{SIG}_Q \times 2^{Q_s} & \text{if } v \notin A \text{ is a leaf of } \Psi \\ R^1_{A,s}(v_l) \bigcirc_v R^1_{A,s}(v_r) & \text{if } v \text{ is not a leaf of } \Psi \end{cases}$$

Here, v_l and v_r refer to the left and right child of v, respectively. The intersection with $\mathrm{SIG}_Q \times 2^{Q_s}$ in the second row is just to optimize the computation, as we are only interested in signatures that can be used for extending the answer with selecting tuples s.

The sets $R^2_{A,s}(v)$ additionally account for the second condition:

$$R^2_{A,s}(v) = \begin{cases} R^1_{A,s}(v) \cap \{(q_0, q_F)\} \times \{Q_s\} \\ \{x \in R^1_{A,s}(v) \mid (\{x\} \bigcirc_u R^1_{A,s}(w)) \cap R^2_{A,s}(u) \neq \emptyset\} \\ \{x \in R^1_{A,s}(v) \mid (R^1_{A,s}(w) \bigcirc_u \{x\}) \cap R^2_{A,s}(u) \neq \emptyset\} \end{cases}$$

The first case applies if v is the root, while the second and third cases apply, when u is the parent of v and w is the left (second case) or right (third case) sibling of v.

We note that the definition of $R^2_{A,s}(v)$ is very similar to a semi-join. The following two lemmas are the technical core of the paper. The proofs are not very difficult but have many case distinctions, as there are 5 relevant forest algebra operations.

Lemma 14. *It holds that $(x, r) \in R^1_{A,s}(v)$ if and only if there exists a run λ on t_v such that $\lambda \models_M (x, r)$, $r \subseteq Q_s$, and $\lambda \models_s A$.*

Proof sketch. The proof is by induction, and identical to the proof of Lemma 9 with the exception that some information about states is carried through the induction. □

Lemma 15. *It holds that $(x, r) \in R^2_{A,s}(v)$ if and only if there exists a run λ on t such that λ is accepting, $\lambda_v \models_M (x, r)$ and $\lambda_v \models_s A$, where λ_v is the restriction of λ to t_v.*

Proof sketch. The proof is by a top-down induction. At the root, the claim holds by Convention 1 and the definition of $R^1_{A,s}$. The induction step propagates the condition that the run over t_v can be completed to an accepting run over t that uses all states of some selecting tuple s down to the leaves. □

The definitions of $R^1_{A,s}$ and $R^2_{A,s}$ yield straightforward algorithms to compute these sets. The computation of $R^1_{A,s}$ can be done bottom up (just as the computation of T^+), while the computation of $R^2_{A,s}$ can be done top-down.

We now have all ingredients for an implementation of the procedure Complete that we depict in Algorithm 2. We first prove correctness before we give an upper bound on the runtime. We use $\mathrm{states}(R^2_{A,s}(v))$ to denote the set of states that occur in some tuple of $R^2_{A,s}(v)$, i.e.,

$$\mathrm{states}(R^2_{A,s}(v)) = \bigcup_{(x,r) \in R^2_{A,s}(v)} r \ .$$

Algorithm 2 Procedure Complete as used in Algorithm 1

Input: incomplete answers $A = (v_1, v_2, \ldots, v_j, \bot, \ldots, \bot)$, node u
Output: the answer $\mathrm{Complete}(A, i)$ from Definition 11

```
1: function Complete(A, u)
2:     return Complete(A, Root(Ψ), u)
3: function Complete(A, v, u)
4:     compute R²_{A,s}(v) for s ∈ S
5:     if max(Nodes(t_v)) ≤ u or q^s_{j+1} ∉ states(R²_{A,s}(v))
                                  for every s ∈ S then return ⊥
6:     if isLeaf(v) then A' ← (v_1, v_2, ..., v_j, v)
7:     else
8:         A' ← Complete(A, LeftChild(v), u)
9:         if A' = ⊥ then A' ← Complete(A, RightChild(v), u)
10:    if A' ≠ ⊥ then compute R¹_{A',s}(v) for s ∈ S
11:    return A'
```

Lemma 16. *The procedure Complete correctly computes the incomplete answer as required by Definition 11.*

Proof. The main challenge of the procedure is to find a node v_{j+1} that can be used to extend the incomplete answer A. As our order of the nodes of t is induced by the order of the leaves of Ψ, we have to find the leftmost leaf of Ψ that can be used to extend A. By the definition of $R^2_{A,s}$, this is the leftmost leaf v with $v > u$ and $q^s_{j+1} \in \mathrm{states}(R^2_{A,s}(v))$.

The procedure returns in Line 5, only if it is sure that no such node v_{j+1} can be found among the descendants of v, either because all nodes below v are smaller or equal than u, or because $q_{j+1} \notin \mathrm{states}(R^2_{A.s}(v))$ and therefore also $q_{j+1} \notin \mathrm{states}(R^2_{A,s}(w))$ for any w below v by the definition of $R^2_{A,s}$.

If v is a leaf, the procedure either returns $\bot$ in Line 5 or computes the correct incomplete answer A'. We note that if the algorithm does not return in Line 5 and v is a leaf, then v is the desired node.

If v is not a leaf, the algorithm first descends into the left subtree and only if no appropriate node was found there descends into the right subtree. It thus ensures to find the leftmost leaf satisfying the conditions. □

Lemma 17. *The procedure Complete runs in time $O(\log(|t|) \cdot |Q|^6 \cdot |S| \cdot 2^k)$.*

Proof sketch. The time spent in each invocation of Complete (excluding time spent in recursive calls) is dominated by the computation of $R^2_{A,s}(v)$ and $R^1_{A',s}(v)$. Both operations can be carried out in time $O(\cdot |Q|^6 \cdot |S| \cdot 2^k)$ using the forest algebra.

It remains to show that the total number of calls is bounded by $O(\log(|t|))$. It can be shown that there is at most one node at each level, such that Complete returns $\bot$ in Line 11. We can conclude that the overall number of calls is bounded by $O(\log(|t|))$, as there are at most $\log(|t|)$ calls that return a value different from $\bot$ and calls that return already in Line 5 do not make any recursive calls. □

We now have all ingredients to show our main result.

Theorem 18. *IncEnum for a k-NFSTA M and a tree t can be solved with auxiliary data of size $O(|Q^4| \cdot |S| \cdot 2^k \cdot |t|)$ which can be computed in time $O(|Q^6| \cdot |S| \cdot 2^k \cdot |t|)$, maintained within time $O(|Q^6| \cdot |S| \cdot$*

$2^k \log(|t|))$ *per update, and which guarantees delay* $O(|Q^6| \cdot k \cdot |S| \cdot 2^k \cdot \log(|t|))$ *between answers.*

Proof. The delay follows from Lemma 17 and the fact that we need at most k calls to COMPLETE to compute the next answer. All other bounds are by Proposition 2 together with Observation 13. □

There is one unaesthetic detail in our algorithm, that we still want to fix: Our implementation enumerates the tree in a strange order that depends on the internal state of our data structure. With a slightly more complicated algorithm and forest algebra, it is possible to enumerate t in pre-order or post-order. Algorithm 2 needs to be changed so that it makes three recursive calls at each inner node that represents a context application. One to search for v in the context among the nodes before the hole, a second that searches the tree, the context is applied to, and a third searching the context again, but now on the nodes after the hole. The vertical monoid of the tree algebra needs to be extended such that it carries the information which states of S are visited before and after the hole.

7 Concluding Remarks and Further Directions

We depicted an algorithm that allows enumeration with logarithmic delay and logarithmic updates for MSO queries on trees, making this the best currently known algorithm that works for relabeling and structural updates on the tree. Still, the main open question is, whether there exists an algorithm that allows for constant delay enumeration and logarithmic updates at the same time. Amarilli et al. [3] made an important step in this direction by providing an algorithm that works for relabeling updates. However, in practice, updates that change the structure of the tree are very common. In the current work we introduced a balancing schema for forest algebra formulas that solves a key problem in generalizing the results of [3] to insertion and deletion updates, which gives hope that combining both results could lead to such a constant delay, logarithmic update algorithm.

Towards future work, we want to investigate how far our techniques can be generalized towards graphs with bounded treewidth, using the generalization in [4]. A straightforward generalization of our algorithm will only be able to deal with relabelings since node insertions and deletions can have drastic impact on tree decompositions. To keep the complexity for relabel operations low, one can assume w.l.o.g. that the label of each node is only stored in one bag of a tree decomposition.

Other future work for which our method seems promising is efficiently computing the *difference between answers*. That is, after an update occurred on the tree, we could say which tuples no longer satisfy the query and which ones are new.

Acknowledgments

I thank Luc Segoufin for bringing me in touch with enumeration of MSO queries and the algebraic perspective and I thank Antoine Amarilli, Johannes Doleschal, and the anonymous reviewers for many helpful remarks.

References

[1] G. Adelson-Velsky and E. Landis. 1962. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR* 146, 2 (1962), 263–266.
[2] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. 2017. A Circuit-Based Approach to Efficient Enumeration. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*. 111:1–111:15.
[3] A. Amarilli, P. Bourhis, and S. Mengel. 2018. Enumeration on Trees under Relabelings. In *International Conference Database Theory (ICDT)*. 5:1–5:18.
[4] G. Bagan. 2006. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *Computer Science Logic (CSL)*. 167–181.
[5] A. Balmin, Y. Papakonstantinou, and V. Vianu. 2004. Incremental validation of XML documents. *ACM Transactions on Database Systems (TODS)* 29, 4 (2004), 710–751.
[6] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. 2004. Efficient Incremental Validation of XML Documents. In *ICDE*. 671–682.
[7] C. Berkholz, J. Keppeler, and N. Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *International Symposium on Principles of Database Systems (PODS)*. 303–318.
[8] C. Berkholz, J. Keppeler, and N. Schweikardt. 2017. Answering FO+MOD Queries Under Updates on Bounded Degree Databases. In *International Conference Database Theory (ICDT)*. 8:1–8:18.
[9] C. Berkholz, J. Keppeler, and N. Schweikardt. 2018. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *International Conference Database Theory (ICDT)*. 8:1–8:19.
[10] H. Björklund, W. Gelade, and W. Martens. 2010. Incremental XPath evaluation. *ACM Transactions on Database Systems (TODS)* 35, 4 (2010), 29:1–29:43.
[11] H. Björklund, W. Martens, and T. Timm. 2015. Efficient Incremental Evaluation of Succinct Regular Expressions. In *ACM Conference on Information and Knowledge Management (CIKM)*. 1541–1550.
[12] H. L. Bodlaender and T. Hagerup. 1998. Parallel Algorithms with Optimal Speedup for Bounded Treewidth. *SIAM J. Comput.* 27, 6 (1998), 1725–1746.
[13] M. Bojańczyk. 2010. Algebra for trees. In *In Handbook of Automata Theory. European Mathematical Society Publishing*.
[14] M. Bojańczyk and I. Walukiewicz. 2007. Forest algebras. In *Automata and Logic: History and Perspectives*. Amsterdam University Press, 107–132.
[15] J. Carme, J. Niehren, and M. Tommasi. 2004. Querying Unranked Trees with Stepwise Tree Automata. In *Rewriting Techniques and Applications (RTA)*. 105–118.
[16] T. Colcombet. 2007. A Combinatorial Theorem for Trees. In *International Colloquium on Automata, Languages and Programming (ICALP)*. 901–912.
[17] B. Courcelle. 2009. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics* 157, 12 (2009), 2675–2700.
[18] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. 2015. Reachability is in DynFO. In *International Colloquium on Automata, Languages and Programming (ICALP)*. 159–170.
[19] A. Durand and Y. Strozecki. 2011. Enumeration Complexity of Logical Query Problems with Second-order Variables. In *Computer Science Logic (CSL)*. 189–202.
[20] M. Frick and M. Grohe. 2004. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic* 130, 1–3 (2004), 3–31.
[21] M. Frick, M. Grohe, and C. Koch. 2003. Query Evaluation on Compressed Trees (Extended Abstract). In *Logic in Computer Science (LICS)*. 188.
[22] W. Kazana and L. Segoufin. 2013. Enumeration of first-order queries on classes of structures with bounded expansion. In *Symposium on Principles of Database Systems (PODS)*. 297–308.
[23] W. Kazana and L. Segoufin. 2013. Enumeration of Monadic Second-Order Queries on Trees. *ACM Transactions on Computational Logic (TOCL)* 14, 4 (2013), 25:1–25:12.
[24] K. Losemann and W. Martens. 2014. MSO Queries on Trees: Enumerating Answers under Updates. In *Joint Meeting of Computer Science Logic (CSL) and Logic in Computer Science (LICS)*.
[25] W. Martens and J. Niehren. 2007. On the minimization of XML Schemas and tree automata for unranked trees. *J. Comput. System Sci.* 73, 4 (2007), 550–583.
[26] K. Mehlhorn and A. Tsakalidis. 1986. An Amortized Analysis of Insertions into AVL Trees. *Siam Journal on Computing* 15, 1 (Feb. 1986), 22–33.
[27] F. Neven. 1999. *Design and Analysis of Query Languages for Structured Documents*. Ph.D. Dissertation. Limburgs Universitair Centrum.
[28] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. 2005. N-Ary Queries by Tree Automata. In *International Conference on Database Programming Languages (DBPL)*. 217–231.
[29] M. Niewerth and L. Segoufin. 2018. Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates. In *International Symposium on Principles of Database Systems (PODS)*. to appear.
[30] S. Patnaik and N. Immerman. 1997. Dyn-FO: A Parallel, Dynamic Complexity Class. *Journal of Computer and System Sciences (JCSS)* 55, 2 (1997), 199–209.
[31] L. Segoufin. 2013. Enumerating with constant delay the answers to a query. In *International Conference on Database Theory (ICDT)*. 10–20.
[32] I. Simon. 1990. Factorization Forests of Finite Height. *Theoretical Computer Science (TCS)* 72, 1 (1990), 65–94.
[33] J. W. Thatcher and J. B. Wright. 1968. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory* 2, 1 (1968), 57–82.
[34] T. Zeume and T. Schwentick. 2014. Dynamic Conjunctive Queries. In *International Conference Database Theory (ICDT)*. 38–49.