

MSO Queries on Trees: Enumerating Answers under Updates

Katja Losemann Wim Martens

University of Bayreuth

Abstract

We investigate efficient view maintenance for MSO-definable queries over trees or, more precisely, efficient enumeration of answers to MSO-definable queries over words and trees which are subject to local updates. For words we exhibit an algorithm that uses an $O(n)$ preprocessing phase and enumerates answers with $O(\log n)$ delay between them. When the word is updated, the algorithm can avoid repeating expensive preprocessing and restart the enumeration phase within $O(\log n)$ time. For trees, our algorithm uses $O(n)$ preprocessing time, enumerates answers with $O(\log^2 n)$ delay, and can restart enumeration within $O(\log^2 n)$ time after receiving an update to the tree. This significantly improves the cost of recomputing the answers of a query from scratch. Our algorithms and complexity results in the paper are presented in terms of node-selecting automata representing the MSO queries.

Categories and Subject Descriptors F.2.0 [Analysis of Algorithms and Problem Complexity]: General; F.4.1 [Mathematical Logic and Formal Languages]: Computational Logic; H.2.8 [Database Management]: Database Applications

General Terms Algorithms, Languages, Theory

Keywords Tree Automata, Query Enumeration, XPath

1. Introduction

Efficient query evaluation is the most central problem in databases. Given a query Q and a database D , we are asked to compute the set or multiset $Q(D)$ 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 n^k tuples. Since D is typically very large in database applications, it may be unfeasible to compute $Q(D)$ in its entirety.

This observation has triggered several lines of research that aim at addressing this problem. For example, in *top-k query answering* the goal is to find the k most relevant answers to a query (according to some heuristic). Another interesting way to deal with this problem is known as *query enumeration* (see, e.g., [1, 7, 8, 12, 13, 17]). In query enumeration, one is interested in producing the answers of $Q(D)$ one by one, preferably quickly, without repetition. 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. Existing algorithms for query enumeration usually 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 [17]. To the best of our knowledge, all existing solutions for query enumeration 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. Since databases can be subjected to frequent updates and preprocessing typically costs linear time, this can again be too costly. 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 *words* and *trees*. Furthermore, the structures can be subjected to local updates. For words we consider updates that relabel a node, insert a node, or delete a node. For trees, updates can relabel a node, or insert/delete a leaf. Our aim is to make 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 re-start enumerating on the new data within the same delay.

For MSO sentences over trees, this problem has been studied by Balmin, Papakonstantinou, and Vianu [2]. 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 subject 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 have a boolean answer. Our goal is to extend Balmin et al.'s result to MSO queries of arity k while guaranteeing a small delay between answers.

Although we do not obtain constant-delay algorithms as in the above mentioned work on static words and trees, we can prove that, in the dynamic setting $O(\log n)$ delay over words and $O(\log^2 n)$

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 ACM 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.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM 978-1-4503-2886-9/14-07...\$15.00.
<http://dx.doi.org/10.1145/2603088.2603137>

delay over trees 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 (resp., $O(\log^2 n)$ time) to produce the first answer on the updated word (resp., 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.

The complexity results in this paper are presented in terms of the size of the word or 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. [15, 20].) Two remarks should be kept in mind when measuring complexity in terms of query size. First, MSO queries can be non-elementary 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 $P = NP$ [9]. (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.) Second, the arity k of the queries is usually very small in practical scenarios. (For example, $k = 2$ suffices for modelling XPath queries, which are central in XML querying.)

Related Work

To the best of our knowledge, this paper is the first to formally study enumeration problems on dynamic trees.

Bagan [1] 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 [7]. Recently, Kazana and Segoufin [13] provided an alternative proof of Bagan’s result based on a deterministic factorization forest theorem by Colcombet [6], which is itself based on a result of Simon [18]. 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 [1], 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. A recent overview of enumeration algorithms with constant delay was given in [17].

Balmin, Papakonstantinou, and Vianu provide an algorithm that can efficiently decide if local updates on trees preserve a Boolean MSO property in time $O(\log^2 n \cdot |N|^3)$ where N is the size of the tree automaton [2]. A main idea in Balmin et al. 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 known by Patnaik and Immerman [16]. Patnaik and Immerman’s divide-and-conquer approach was also used by Björklund et al. [4] in an algorithm for maintaining whether updates preserve a property specified by an XPath query. Although the XPath dialects studied in [4] are less expressive than tree automata, they may be exponentially more succinct. These papers essentially consider Boolean queries and are not concerned with efficiently enumerating answers.

Bojanczyk and Figueira [5] consider evolutions $t_1, \dots, t_m$ of trees (which they call *document evolutions*) and evaluate two-dimensional logics over such sequences. Such logics can express properties of single trees and how such properties evolve over time. (For example, “eventually, every a -node will have a b -child”.) They

read the input as t_1 followed by a sequence of $m - 1$ local updates and give an $O(m \cdot \log n)$ algorithm to decide if a formula holds over the evolution (assuming $m > n$). Therefore, in the temporal dimension, the setting in [5] is more general than ours — we cannot compare different versions of the tree. Since they are only concerned with satisfaction of a property, they do not consider small delay algorithms for enumerating answers.

2. Definitions

By $[n]$ we denote the finite set $\{1, \dots, n\}$. The number of elements of a finite set A is denoted $|A|$. For a finite set A , we define a *multiset* m over A as a function $m : A \rightarrow \mathbb{N}$. Here, $m(a)$ is the *multiplicity* of a in m . We say that $a \in m$ if $m(a) > 0$. The size of m , denoted $|m|$, is the sum $\sum_{a \in A} m(a)$ of all multiplicities of elements in m . We denote multisets in brackets $\{\}$ and $\lfloor \rfloor$. E.g., in $m = \{1, 1, 3\}$ we have that $m(1) = 2$ and $m(3) = 1$. The union $m = m_1 \cup m_2$ (resp., intersection $m = m_1 \cap m_2$) of multisets is defined as usual, taking $m(a) = m_1(a) + m_2(a)$ (resp., $m(a) = \min\{m_1(a), m_2(a)\}$) for every $a \in A$. We say that $m_1 \subseteq m_2$ if $m_1(a) \leq m_2(a)$ for all $a \in A$.

By Σ we denote an alphabet, i.e., a finite set of *labels*. A word (over alphabet Σ) is a finite sequence $w = a_1 \dots a_n$ of labels from Σ . To a word w we associate a *set of nodes* $\text{Nodes}(w) = \{v_1, \dots, v_n\}$ such that each node v_i bears the label $\text{lab}(v_i) = a_i$. Since nodes in words are linearly ordered (due to the structure of the word) we often take $\text{Nodes}(w) = \{1, \dots, n\}$ to simplify notation. However, our results do not require that $\text{Nodes}(w) = \{1, \dots, n\}$. For v_i, v_j with $1 \leq i \leq j \leq n$ we denote by $w[v_i..v_j]$ the subword $a_i \dots a_j$.

Trees in this paper are labeled, rooted, and binary. 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 $|t|$. Therefore, each tree t has a unique root and every node has 0, 1, or 2 children. Nodes in trees which have no children are called *leaves*. The (unique) Σ -label of node v is denoted by $\text{lab}(v)$.

For a finite set A and a word $w \in A^*$ or tuple $s = (a_1, \dots, a_k) \in A^k$, we regularly need the *set of ingredients* occurring in it. We refer to this set as $\text{set}(w)$ or $\text{set}(s)$, respectively. It is defined as $\text{set}(w) := \{a \in A \mid \exists v \in \text{Nodes}(w), \text{lab}(v) = a\}$ and $\text{set}(s) = \{a_1, \dots, a_k\}$.

2.1 Automata and Selecting Automata

We use (node- and tuple-) selecting finite automata (see, e.g., [10, 14]) as formalism for queries. It is well-known that these can express MSO queries with free node variables (Section 3 of [15]). We start by recalling notation for ordinary finite automata. A *non-deterministic finite automaton (NFA)* is a tuple $N = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of states, Σ the alphabet, q_0 is the initial state, and $F \subseteq Q$ the set of accepting states. The transition function δ has signature $Q \times \Sigma \rightarrow 2^Q$. When $q_2 \in \delta(q_1, a)$, it means that, whenever N is in state q_1 , reading an $a \in \Sigma$ can bring it in state q_2 . The function δ^* extends δ to strings in the canonical way, that is, $\delta^*(q, a) = \delta(q, a)$ and $\delta^*(q, aw) = \cup_{q' \in \delta(q, a)} \delta^*(q', w)$. Intuitively, $q_2 \in \delta^*(q_1, w)$ whenever reading w can bring N from q_1 to q_2 . A *run* of N on a word $w = a_1 \dots a_n$ is a word $r = q_0 \dots q_n \in Q^*$ such that $q_i \in \delta(q_{i-1}, a_i)$ for every $i \in [n]$. For each node $i \in [n]$, we say that run r *visits node i in state q_i* , also denoted $r(i) = q_i$. The run is *accepting* if $q_n \in F$. A word w is in the *language* of N (denoted $L(N)$) if there exists an accepting run of N on w . A *partial run* of N on w is defined analogously to a run, except that we do not require the first state to be q_0 .

A (*bottom-up*) *nondeterministic tree automaton* or *NTA* is a tuple $N = (Q, \Sigma, \delta, F)$ where Q is the finite set of states, $F \subseteq Q$ is the set of accepting states, and a set of transition rules δ which

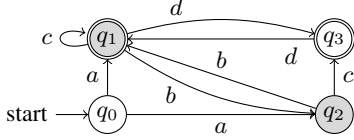


Figure 1. 2-NFSA M with $S = \{(q_1, q_2), (q_2, q_1)\}$.

are either of the form $(q_1, q_2, a) \rightarrow q$ or $a \rightarrow q$, for states $q_1, q_2, q \in Q$ and a label $a \in \Sigma$. A run of N on a labeled binary tree t is an assignment of nodes to states $\lambda : \text{Nodes}(t) \rightarrow Q$ such that for every $v \in \text{Nodes}(t)$ the following holds: if v is a leaf, then $\text{lab}(v) \rightarrow \lambda(v) \in \delta$; if v has children v_1 and v_2 then $(\lambda(v_1), \lambda(v_2), \text{lab}(v)) \rightarrow \lambda(v) \in \delta$. A run is *accepting* if $\lambda(r) \in F$ for the root r of t . Run λ visits v in q if $\lambda(v) = q$. A tree t is *accepted* if there exists an accepting run on t . The set of all accepted trees is denoted by $L(N)$.

Now we are ready to define node selecting automata and queries. For $k \in \mathbb{N}$, a k -ary non-deterministic finite selecting automaton (k -NFSA) M is a pair (N, S) , where N is an automaton 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 word w of length n , it computes a set of tuples in $\text{Nodes}(w)^k$. More precisely, we define

$$M(w) = \{(v_1, \dots, v_k) \mid \text{there is an accepting run } r \text{ of } N \\ \text{on } w \text{ and a tuple } (p_1, \dots, p_k) \in S \text{ such that,} \\ \text{for every } \ell \in [k], r \text{ visits } v_\ell \text{ in } p_\ell\}.$$

Notice that, if $w \notin L(N)$, then $M(w) = \emptyset$. The corresponding definitions for k -ary non-deterministic finite selecting tree automaton (k -NFSTA) are the same as for k -NFSA, with the only difference that N should be an NTA instead of an NFA. Figure 1 illustrates a 2-NFSA M . For the input word $w = abcd$, we get $M(w) = \{(1, 2), (2, 1), (1, 3), (3, 1), (2, 4), (4, 2)\}$.

2.2 Incremental Evaluation and Enumeration

Let M be a selecting automaton (k -NFSA or k -NFSTA), d the input for M (a word or a tree), and $M(d)$ be the answer of M on d . We are interested in efficiently maintaining $M(d)$ under updates of d . This means that we can have an update u to d , yielding another structure d' , and we wish to efficiently compute $M(d')$. The latter cost should be more efficient than computing $M(d')$ from scratch. We consider the following updates on trees (cfr. [2]): (i) *Replace the current label of a specified node by another label*, (ii) *insert a new leaf node after a specified node*, (iii) *insert a new leaf node as first child of a specified node*, and (iv) *delete a specified leaf node*. On words, we consider the updates (i), (ii), and (iv), but the word “leaf” can be omitted.

We allow a single preprocessing phase in which we can compute an *auxiliary data structure* $\text{Aux}(d)$ that we can use for efficient query answering. When d is updated to d' , we therefore want to efficiently compute $M(d')$ and efficiently update $\text{Aux}(d)$ to $\text{Aux}(d')$.

If M is simply an NTA or NFA (i.e., a 0-ary NFSA or NFSTA), then this problem is known as *incremental evaluation* and was studied by, e.g., Balmin et al. [2]. 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 $\text{Aux}(d)$, (ii) time needed to compute $\text{Aux}(d)$, (iii) time needed to update $\text{Aux}(d)$ to $\text{Aux}(d')$, and (iv) time delay we can guarantee between answers of $M(d')$. The underlying model of computation is a random access machine (RAM) with uniform cost measure.

In the remainder of the paper we use INCEVAL and INCENUM to refer to the incremental evaluation and enumeration problems, respectively.

2.3 Two Remarks

In the technical part of this paper we only consider updates of the kind (i), i.e., *relabeling updates*. The first remark is that this is sufficient. Balmin et al. [2] argue why one can use self-balancing auxiliary tree structures to generalize the techniques for updates (i) to updates of the kind (ii)–(iv).

Second, all results we present for binary trees can be immediately generalized to *unranked trees*, in which nodes can have arbitrarily many children. Unranked trees are particularly relevant in the context of XML, since XML documents naturally abstract as unranked trees. The formal argument why it is sufficient to consider binary trees is that one can naturally encode unranked trees in binary ones. For more details we refer to [2].

3. The Word Case

In this section, we show how to solve incremental enumeration for a k -NSFA and a word efficiently. Therefore, we need to present a well-known algorithm to solve incremental evaluation efficiently for NFAs first. In this section we assume that $\text{Nodes}(w) = [n]$ for simplicity of notation.

3.1 Incremental Evaluation

The following algorithm, first described by Patnaik and Immerman [16], solves INCEVAL for an NFA $N = (Q, \Sigma, \delta, q_0, F)$ and a word $w = a_1 \dots a_n \in \Sigma^*$. For simplicity, we assume here that n is a power of 2, say $n = 2^m$. In preprocessing, the algorithm builds the following auxiliary structure.

Definition 1. For a word w with $\text{Nodes}(w) = \{1, \dots, n\}$, the *auxiliary tree* N_w^{aux} is defined as follows:

- the root of N_w^{aux} is v_{1n} ;
- each node v_{xy} , for which $y - x > 0$, has children v_{xz} and $v_{(z+1)y}$ where $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$; and
- nodes v_{xx} are the leaves, for all $1 \leq x \leq n$.

We identify the nodes x of w with leaves v_{xx} in N_w^{aux} . That is, the nodes of w are leaves in N_w^{aux} .

Every node v_{xy} in N_w^{aux} is associated to the subword $w[x..y]$ and holds information about how N ’s state can change when reading $w[x..y]$:

Definition 2. Let $v_{xy} \in \text{Nodes}(N_w^{\text{aux}})$, then the *transition relation* $T(v_{xy})$ is defined as:

- if $x = y$: $T(v_{xx}) := \{(q_1, q_2) \mid q_2 \in \delta(q_1, a_x)\}$
- otherwise, v_{xy} has left child v_{xz} , right child $v_{(z+1)y}$, and $T(v_{xy}) := \{(q_1, q_2) \mid \exists q \in Q \text{ such that } (q_1, q) \in T(v_{xz}) \text{ and } (q, q_2) \in T(v_{(z+1)y})\}$

Thus, $(q_1, q_2) \in T(v_{xy})$ if and only if $q_2 \in \delta^*(q_1, w[x..y])$, i.e., reading $w[x..y]$ can bring N from q_1 to q_2 . We can compute $T(v_{xy})$ from $T(v_{xz})$ and $T(v_{(z+1)y})$ in time $\mathcal{O}(|Q|^3)$ (this corresponds to joining two binary relations). Since N_w^{aux} has $2n - 1$ nodes and $\mathcal{O}(\log n)$ depth, N_w^{aux} and T can be computed in time $\mathcal{O}(|Q|^3 \cdot n)$. Finally, $w \in L(N)$ if and only if $(q_0, q_F) \in T(v_{1n})$ for some $q_F \in F$.

We now describe how updates are maintained. Assume that we change label a_x to b , that is, the new word is $w = a_1 \dots a_{x-1} b a_{x+1} \dots a_n$. The relations T that are affected by the update are those lying on the path from the leaf v_{xx} to the root v_{1n} ($\mathcal{O}(\log n)$ many). These can each be updated in time $\mathcal{O}(|Q|^3)$ in a bottom-up

pass through N_w^{aux} , yielding a total time of $\mathcal{O}(|Q|^3 \cdot \log n)$ for one update.

Theorem 3 ([2, 16]). *INCEVAL for an NFA and a word w can be solved with a preprocessing phase of time $\mathcal{O}(|Q|^3 \cdot n)$, auxiliary structure of size $\mathcal{O}(|Q|^2 \cdot n)$, and within time $\mathcal{O}(|Q|^3 \cdot \log n)$ after each new update.*

The above approach can easily be adapted to words whose lengths are not a power of 2.

3.2 Auxiliary Data Structure for Enumeration

We now extend the mapping T on N_w^{aux} such that we can use it to enumerate answers for a k -NSFA on a word w with logarithmic delay. This structure constitutes the auxiliary data we store for our enumeration algorithm during updates. We can construct it in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot n)$ and, whenever w receives an update, we can update the structure in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ and recommence logarithmic-delay enumeration. We fix the following notation for the remainder of the section. By $M = ((Q, \Sigma, \delta, q_0, F), S)$, we denote a k -NSFA and by $w = a_1 \cdots a_n \in \Sigma^*$ the input word. By Q_S we denote the set of all states that appear in some selecting tuple, i.e., $Q_S = \bigcup_{s \in S} \text{set}(s)$.

The structure is based on the auxiliary tree N_w^{aux} from Section 3.1, but now we store tuples that contain, in addition to the pair of states, a set of selecting states which can be reached by a run on the subword associated to the node. We denote this new relation by T^+ .

Definition 4. For each $v_{xy} \in N_w^{\text{aux}}$, we define $T^+(v_{xy})$ to be the set of tuples $(q_1, q_2, I) \in (Q^2 \times 2^{Q_S})$ for which there exist a selecting tuple $s \in S$ and partial run $r = q_1 \cdots q_2$ on $w[x..y]$ such that $I = \text{set}(r) \cap \text{set}(s)$.

Notice that all $T^+(v_{xy})$ can be computed efficiently:

- If $x = y$ then $T^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta(q_1, a_x) \text{ and } I = \{q_2\} \cap Q_S\}$. (The condition on I states that $I = \{q_2\}$ if q_2 appears in some selecting tuple s ; and $I = \emptyset$ otherwise.)
- Otherwise, let v_1 and v_2 be the left and right child of v_{xy} in N_w^{aux} , then $T^+(v_{xy}) = T^+(v_1) \bowtie^+ T^+(v_2)$.

Here, we define $T^+(v_1) \bowtie^+ T^+(v_2) := \{(q_1, q_2, I) \mid \exists p \in Q, \exists I_1, I_2 \subseteq Q_S, \exists s \in S \text{ such that } (q_1, p, I_1) \in T^+(v_1), (p, q_2, I_2) \in T^+(v_2), \text{ and } I = (I_1 \cup I_2) \cap \text{set}(s)\}$. (The proof that this computation is correct is a straight-forward induction.) Furthermore, we can maintain T^+ under updates analogously to relation T in Section 3.1 but with extra time needed for the $\bowtie^+$ -operation.

Lemma 5. *For a k -NSFA M and a word w of length n , the tree N_w^{aux} and T^+ have size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot n)$, can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot n)$ and updated in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$.*

This concludes the description of the dynamic data structure.

3.3 Enumerating Query Answers

We now discuss how to enumerate query answers. In this section, we assume that N_w^{aux} and T^+ are already computed. A high-level description of the enumeration algorithm is outlined in Algorithm 1. This procedure is similar to enumerating words in a dictionary in lexicographic order, but the details are rather different. The procedure **Enum** takes a k -NSFA and a word; invokes procedure “Complete” to compute the first set of answers (there can be several smallest answers); starts the enumeration by repeatedly calling **Next** (which allows us to go from one set of answers to the next) until all answers are depleted. The algorithm could either enumerate answers in *set semantics* as defined in the Definitions; or in *multiset semantics* which we will discuss in the conclusions.

Algorithm 1 Enumeration of $M(w)$

```

1: Enum( $M, w$ ) {
2:   Input:  $k$ -NSFA  $M = ((Q, \Sigma, \delta, F), S)$ , word  $w$ 
3:   Output: Enumeration of all answers in  $M(w)$ 
4:    $\mathcal{A} = \text{Complete}(\{\emptyset\})$ 
5:   while  $\mathcal{A} \neq \emptyset$  do
6:      $\text{output}(\mathcal{A})$ 
7:      $\mathcal{A} = \text{Next}(\mathcal{A})$ 
8:   }
9: Next( $\mathcal{A}$ ) {
10:  Input: set  $\mathcal{A}$  of annotated answers
11:  Output: set of smallest annotated answers larger than  $\mathcal{A}$ 
12:  while  $\text{Nextnode}(\mathcal{A}) = \emptyset$  do
13:     $\mathcal{A} \leftarrow \text{Back}(\mathcal{A})$ 
14:    if  $\mathcal{A} = \emptyset$  then return  $\emptyset$ 
15:  return  $\text{Complete}(\text{Nextnode}(\mathcal{A}))$ 
16: }
```

Our first goal in this section is to explain the operations that are used in Algorithm 1. We require some preliminary notions. First we define the *output ordering* $\preceq$ in which we will output answers to the query. For a tuple $t = (i_1, \dots, i_k) \in \mathbb{N}^k$, let $\text{sort}(t)$ denote the word obtained by sorting $i_1, \dots, i_k$ in increasing order and concatenating the result (as a word in $\mathbb{N}^*$). More precisely, $\text{sort}(t) = i_{\sigma(1)} \cdots i_{\sigma(k)}$ where σ is a permutation on $[k]$ such that $i_{\sigma(j)} \leq i_{\sigma(j+1)}$ for every $j \in [k-1]$. For example, $\text{sort}((5, 2, 2, 3, 12)) = 2\ 2\ 3\ 5\ 12$. The total order $\preceq$ between tuples $(i_1, \dots, i_k)$ is defined as the lexicographical order on $\text{sort}((i_1, \dots, i_k))$ (taking the empty word to be the lexicographically smallest word). We define $\preceq$ on multisets over $\mathbb{N}$ analogously. We denote the strict variant of $\preceq$ by $\prec$.

In the course of our algorithm we compute so-called *annotated answers*, which are multisets of pairs in $\text{Nodes}(w) \times Q$. Annotated answers contain, in addition to nodes of w , also the states that were responsible for selecting the nodes. The semantics of such a multiset are that, for each element (i, q) , there is an accepting run on w which visits node i in state q . If (i, q) occurs j times in the multiset, then there is a selecting tuple $s \in S$ (with at least j occurrences of q) and we decide to associate node i to j occurrences of q in s . (Intuitively, this means that we will eventually produce an answer to the query that has j occurrences of node i .) Formally, an *annotated answer* of $M = (N, S)$ on w is a multiset A^{full} over $\text{Nodes}(w) \times Q$ of the form

$$\{(i_1, q_1), \dots, (i_k, q_k)\}$$

such that there is an accepting run r of N on w and a $(q_1, \dots, q_k) \in S$ such that r visits i_ℓ in q_ℓ , for every $\ell \in [k]$. It holds that $|A^{\text{full}}| = k$. We sometimes also say that A^{full} is an *annotated answer w.r.t. r* if we want to emphasize the connection between A^{full} and r . An *incomplete (annotated) answer* is a (not necessarily strict) subset A of some annotated answer A^{full} . (Therefore, every incomplete answer can be completed into an answer.) For a multiset A over $\text{Nodes}(w) \times Q$, we denote by $\text{Nodes}(A)$ the multiset of nodes in A . That is, for $A = \{(i_1, q_1), \dots, (i_k, q_k)\}$ we have that $\text{Nodes}(A) = \{i_1, \dots, i_k\}$.

We extend the order $\preceq$ to multisets over $\text{Nodes}(w) \times Q$. For two such multisets A and B , we say that $A \preceq B$ if $\text{Nodes}(A) \preceq \text{Nodes}(B)$. We extend $\prec$ analogously. Furthermore, we define the set of minima for a set $\mathcal{A}$ of incomplete answers:

$$\min(\mathcal{A}) = \{A \in \mathcal{A} \mid \forall B \in \mathcal{A} : A \preceq B\}$$

We are now ready to define the semantics of the functions in Algorithm 1.

Definition 6. Let $\mathcal{A}$ be a set of multisets over $\text{Nodes}(w) \times Q$.

$$\text{Complete}(\mathcal{A}) := \min\{A^{\text{full}} \mid A^{\text{full}} \text{ is an annotated answer} \\ \text{such that } \exists A \in \mathcal{A} : A \subseteq A^{\text{full}} \text{ and } A \preceq A^{\text{full}}\}$$

Intuitively, $\text{Complete}(\mathcal{A})$ contains the smallest annotated answers of M obtained from extending elements $A \in \mathcal{A}$ on nodes of w that are all larger or equal to the maximal node already used in A . (So, $\text{Complete}(\{\emptyset\})$ is the set of smallest annotated answers.)

The procedure $\text{Next}(\mathcal{A})$ should give us, for a set of annotated answers, the set of immediate successors in output order. To describe how we compute $\text{Next}(\mathcal{A})$, we use the following ingredients in Algorithm 1. Let $A = \{(i_1, q_1), \dots, (i_j, q_j)\}$ be a multiset over $\text{Nodes}(w) \times Q$ such that, for each i_ℓ there is at most one q_ℓ such that $(i_\ell, q_\ell) \in A$. Let $i_j \in \max(\text{Nodes}(A))$, then we define $A_{\text{del}} = \{(i_1, q_1), \dots, (i_{j-1}, q_{j-1})\}$.

Definition 7. Let $\mathcal{A}$ be a set of multisets over $\text{Nodes}(w) \times Q$.

$$\text{Back}(\mathcal{A}) := \min\{A \mid A \text{ is an incomplete answer such that} \\ \exists A' \in \mathcal{A} : \text{Nodes}(A) = \text{Nodes}(A'_{\text{del}})\}$$

$\text{Back}(\mathcal{A})$ performs a kind of backtracking step. It returns all smallest incomplete answers which, compared with an element $A = \{(i_1, q_1), \dots, (i_j, q_j)\} \in \mathcal{A}$, annotate exactly the nodes $i_1, \dots, i_{j-1}$ (if we assume i_j to be maximal).

Definition 8. Let $\mathcal{A}$ be a set of multisets over $\text{Nodes}(w) \times Q$.

$$\text{Nextnode}(\mathcal{A}) := \min\{A \mid A \text{ is an incomplete answer such} \\ \text{that } \exists A' \in \mathcal{A} : |A| = |A'| \text{ and } A' \prec A \text{ and } A_{\text{del}} \subseteq A'\}$$

For a set of incomplete answers $\mathcal{A}$, the procedure $\text{Nextnode}(\mathcal{A})$ returns the incomplete answers of the same size as incomplete answers in $\mathcal{A}$, such that only the maximal node of an answer in $\mathcal{A}$ has been incremented.

Lemma 9. Let $\mathcal{A}$ be a set of annotated answers. Then $\text{Next}(\mathcal{A})$ in Algorithm 1 returns

$$\min\{A^{\text{full}} \mid A^{\text{full}} \text{ is an annotated answer} \\ \text{such that } \exists A \in \mathcal{A} : A \prec A^{\text{full}}\}.$$

Finally, the procedure $\text{output}(\mathcal{A})$ takes a set of annotated answers $\mathcal{A}$ and writes the set $\{(i_1, \dots, i_k) \mid \{(i_1, q_1), \dots, (i_k, q_k)\} \in \mathcal{A} \text{ and } (q_1, \dots, q_k) \in S\}$ to the output, in arbitrary order. Notice that this set can contain multiple tuples, but they are all equal with respect to $\preceq$. For example, one tuple can be $(1, 2, 2, 3, 4)$ and another could be $(2, 4, 3, 2, 1)$. Furthermore, the output procedure can be designed such that the delay between these tuples in the output is constant.

The proof of the next lemma relies on the following observation about function calls in Algorithm 1: All $\mathcal{A}$ in the algorithm are such that, for all $A, B \in \mathcal{A}$, we have $\text{Nodes}(A) = \text{Nodes}(B)$. This property trivially holds since all operations in Algorithm 1 return a set of *minima* of incomplete annotated answers.

Lemma 10. $\text{Enum}(M, w)$ correctly enumerates all answers in $M(w)$.

We use the following sections to explain how $\text{Complete}(\mathcal{A})$, $\text{Back}(\mathcal{A})$, and $\text{Nextnode}(\mathcal{A})$ can be implemented efficiently.

3.4 The First Answer

To compute $\text{Complete}(\{\emptyset\})$, the first answer(s) to the query w.r.t. the output ordering, we need to find the leftmost piece of informa-

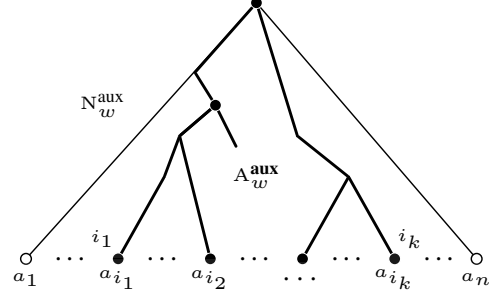


Figure 2. A_w^{aux} is the part of N_w^{aux} that is used to compute the first answers. (Note that $i_j = v_{i_j i_j}$.)

tion in N_w^{aux} that is relevant to some answer. After finding this first ingredient to an answer, we store it in a set of so-called *growing (annotated) answers*, which will evolve into the first answer of the query. Then we navigate further to the right to search for the leftmost nodes in w that can be used to add more and more information to the growing answers, until at least one growing answer is *complete*. Next, we define *growing (annotated) answers*, which contain the full information of some answer to M on w up to a node j .

Definition 11. Let $q \in Q$, $j \in [n]$, and A be a multiset over $\text{Nodes}(w) \times Q$. Then (q, A) is a *growing annotated answer up to node j* if there is an accepting run r of N on w such that

- r visits j in q ; and
- there is an annotated answer A^{full} w.r.t. r such that, for every $p \in Q$ and $i \in [n]$,
 - if $i < j$, then $A^{\text{full}}((i, p)) = A((i, p))$,
 - if $i = j$, then $A((i, p)) \leq A^{\text{full}}((i, p))$, and
 - if $i > j$, then $A((i, p)) = 0$.

The second bullet in the above definition states that A has the same information as A^{full} concerning the nodes up to j and possibly partial information about j itself. For brevity, we often refer to (q, A) as *growing answer*.

We compute growing answers as follows. Assume that $(i_1, \dots, i_k)$ (see Figure 2) is a smallest answer in $M(w)$ w.r.t. the output order. (Notice that some of the i_j can be equal.) Let A_w^{aux} be the tree induced by all ancestors of nodes i_j in N_w^{aux} . Hence, A_w^{aux} has at most k leaves, its root is the root of N_w^{aux} , and each of its leaves corresponds to a node i_j . For obtaining $(i_1, \dots, i_k)$, we perform a depth-first left-to-right traversal of A_w^{aux} . Since the depth of N_w^{aux} is logarithmic in n , such a traversal costs about $O(k \log n)$ steps (if one would magically know where to go). In particular, one can travel from one leaf in A_w^{aux} to the next within $O(\log n)$ steps. Our goal is to show that this is possible when one stores the right kind of information along the paths of A_w^{aux} .

We first explain how to compute and traverse the leftmost path of A_w^{aux} . We start at the root of N_w^{aux} and need to decide which child to choose. To this end, we compute *relevant tuples*, which are defined in the following.

Definition 12. For each $v \in N_w^{\text{aux}}$ the set of *relevant tuples* of v , denoted $R(v)$, is inductively defined as follows:

- $R(v_{1n}) = \{(q_0, q_F, \text{set}(s)) \in T^+(v_{1n}) \mid q_F \in F, s \in S\}$;
- Otherwise, if $(q_1, q_2, I) \in R(v)$ and v_1 and v_2 are left and right child of v , then we want to “split” I between v_1 and v_2 . More precisely, let $R_{v_1, v_2} = \{(q_1, q_2, J_1, q_2, q_3, J_2) \mid \exists (q_1, q_3, I) \in R(v), (q_1, q_2, I_1) \in T^+(v_1), (q_2, q_3, I_2) \in T^+(v_2) \text{ such that } J_1 \cup J_2 = I, J_1 \subseteq I_1, \text{ and } J_2 \subseteq I_2\}$. Then,

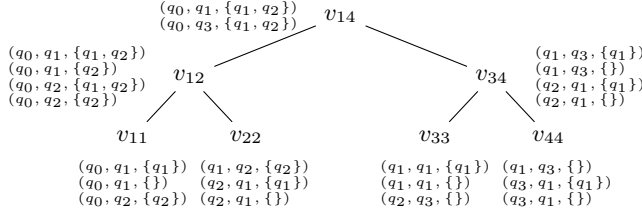


Figure 3. The relation R for the 2-NSFA M from Figure 1 and $w = abcd$.

$$R(v_1) = \{(q_1, q_2, J_1) \mid (q_1, q_2, J_1, q_2, q_3, J_2) \in R_{v_1, v_2}\} \text{ and} \\ R(v_2) = \{(q_2, q_3, J_2) \mid (q_1, q_2, J_1, q_2, q_3, J_2) \in R_{v_1, v_2}\}.$$

The relevant tuples of the root v_{1n} of N_w^{aux} are the tuples for which I is exactly a set of states appearing in some selecting tuple. Therefore, if we can match every state in I , we can produce an answer. By definition of T^+ , this means that M returns at least one answer if and only if $R(v_{1n}) \neq \emptyset$. Further down in the tree, if $(q_1, q_2, I) \in R(v)$, we split I among the children v_1 and v_2 of v in all possible ways. This is to ensure that, if we find a partial result for v_1 , we are *certain* that we can find sufficient information below v_2 to annotate every state in I .

Therefore, the sets R contain exactly the information from T^+ that is relevant for producing answers, i.e., tuples in R are associated to accepting runs that produce at least one answer. This is captured in the following lemma. Here, for an annotated answer $A^{\text{full}} = \{(i_1, q_1), \dots, (i_k, q_k)\}$ and two nodes ℓ, r of w , the *projection of A^{full} onto $[\ell, r]$* , denoted $A_{[\ell, r]}^{\text{full}}$, is defined to be the multiset $\{(i, q_i) \mid \ell \leq i \leq r\}$.

Lemma 13. *For every node $v_{xy} \in N_w^{\text{aux}}$, we have that $R(v_{xy})$ is the set of all (q_1, q_2, I) such that there is an annotated answer A^{full} w.r.t. some run r with $r(x) \in \delta(q_1, w[x])$, $r(y) = q_2$, and $I = \text{Nodes}(A_{[x, y]}^{\text{full}})$.*

For the 2-NSFA in Figure 1, the relation R is shown in Figure 3. Finally, computing R can be done by straightforward implementation of the definition in a top-down way.

Lemma 14. *Given N_w^{aux} and T^+ , we can compute $R(v_{1n})$ in time $\mathcal{O}(|Q|^2 \cdot 2^k)$ and, for every other $v \in N_w^{\text{aux}}$ with parent v_p , compute $R(v)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$ if $R(v_p)$ is known.*

We state Lemma 14 as it is because our algorithm will not compute $R(v)$ for every node v of N_w^{aux} but only among paths of A_w^{aux} .

3.4.1 The First Part of the First Answer

In order to find the leftmost path of A_w^{aux} , we start at the root of N_w^{aux} and iteratively perform the following: Whenever we are in a node v , we compute the sets of relevant tuples of its two children. We proceed to the leftmost child for which the set of relevant tuples contains a tuple (q_1, q_2, I) with $I \neq \emptyset$ and stop when we reach a leaf. We claim that this leaf is the leftmost node i_1 in N_w^{aux} that can be used in some smallest answer of $M(w)$ (see Figure 2). Notice that we only know that i_1 is used in such a smallest answer but not necessarily as the leftmost element. (For example, answers of the form $(i_2, i_1, \dots)$ with $i_2 > i_1$ are possible too.)

Lemma 15. *Let u be the leftmost leaf of N_w^{aux} such that $R(u)$ has a tuple (q_1, q_2, I) with $I \neq \emptyset$. Then u is the node i_1 in w .*

This allows us to define our first set G of growing answers:

$$G(i_1) := \{(q_2, \{(i_1, q_2)\}) \mid (q_1, q_2, \{q_2\}) \in R(i_1)\}$$

By Lemma 13 and 15, every element in $G(i_1)$ is a growing answer up to node i_1 . By Lemma 14, i_1 and, therefore, the set $G(i_1)$ can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ by traversing the path from the root of N_w^{aux} to i_1 . For the running example in Figure 1, we have $G(1) = \{(q_1, \{(1, q_1)\}), (q_2, \{(1, q_2)\})\}$. In this way, we know how to compute i_1 , if it exists.

3.4.2 Growing Until the First Answer is Complete

We assume that from now on we know some j for which the set $G(i_j)$ is defined and not empty. We will explain how to compute the set $G(i_{j+1})$ containing similar information for the node i_{j+1} . To this end, we first have to find the node i_{j+1} itself (recall that not necessarily $i_{j+1} \neq i_j$) and then all the information which is needed to calculate the correct set of growing answers. We will navigate from i_j to the right and only keep track of the relevant tuples that are *compatible* with our growing answer(s). Our next aim is to define this compatibility. In the following, the *projection of a multiset $A = \{(i_1, q_1), \dots, (i_k, q_k)\}$ of tuples over $\mathbb{N} \times Q$ onto Q* , denoted $\pi_Q(A)$, is defined as $\{q_1, \dots, q_k\}$.

Definition 16 (Compatibility). Let v_{xy} be a node of N_w^{aux} . For an annotated answer A^{full} w.r.t. run r , we say that a tuple

- $(q_1, q_2, I) \in R(v_{xy})$ is *compatible with A^{full} and r* if $r(x) \in \delta(q_1, w[x])$, $r(y) = q_2$, and $I = \text{set}(\pi_Q(A_{[x, y]}^{\text{full}}))$.

(Here, $I = \text{set}(\pi_Q(A_{[x, y]}^{\text{full}}))$ ensures that I is the set of selecting states in A^{full} used between nodes x and y in w .) Furthermore, for (q, A) a growing answer up to node i ,

- (q, A) is *compatible with A^{full} and r* if $r(i) = q$, $A_{[1, i-1]} = A_{[1, i-1]}^{\text{full}}$ and $A_{[i, i]} \subseteq A_{[i, i]}^{\text{full}}$.

Finally, $(q_1, q_2, I) \in R(v_{xy})$ is *compatible with (q, A)* if there exists an annotated answer A^{full} w.r.t. some run r such that both (q_1, q_2, I) and (q, A) are compatible with A^{full} and r .

Now, we can define the node i_{j+1} in terms of compatibility.

Proposition 17. *The node $i_{j+1} \geq i_j$ is the smallest node in w for which there exists a tuple $(q_1, q_2, I) \in R(i_{j+1})$ with $I \neq \emptyset$ which is compatible with some $(q, A) \in G(i_j)$.*

Once we have i_{j+1} we can also define the set $G(i_{j+1})$:

$$G(i_{j+1}) = \{(q_2, A \cup \{(i_{j+1}, q_2)\}) \mid \\ \text{there exists some } (q_1, q_2, \{q_2\}) \in R(i_{j+1}) \\ \text{compatible with some } (q, A) \in G(i_j)\}$$

In this way, our algorithm will successively compute sets $G(i_j)$ for increasing values of j . The next lemma states that the last such set, $G(i_k)$, contains indeed the answer(s) we want.

Lemma 18. *Let $\mathcal{A}^{\text{first}}$ be the set of smallest annotated answers. Then, it holds that $G(i_k) = \{(q, A) \mid A \in \mathcal{A}^{\text{first}} \text{ and } (q, A) \text{ is compatible with } A\}$.*

Regarding the example from Figure 1, we have $k = 2$ and, thus, $G(2) = \{(q_1, \{(1, q_2), (2, q_1)\}), (q_2, \{(1, q_1), (2, q_2)\})\}$. It remains to show how to compute i_{j+1} and $G(i_{j+1})$ efficiently. From the last section, we know that we can compute $G(i_1)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$. Next, we prove that we can compute i_{j+1} in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ when $G(i_j)$ is given. Afterwards, we examine the computation of the set $G(i_{j+1})$.

To begin with, we extend the notion of the relevant relation. Intuitively, this relation stores which tuples from R remain relevant for constructing the smallest possible answer, given the knowledge we have at node i_j . We call such tuples *j-relevant*.

Definition 19. For $v_{xy} \in N_w^{\text{aux}}$ and $j \in \{0, \dots, k\}$, we define the set of j -relevant tuples of v_{xy} , denoted $R_j(v_{xy})$, as follows:

- $R_0(v_{xy}) := R(v_{xy})$ and,
- for each $j \geq 1$,
 - if $y < i_j$, then $R_j(v_{xy}) := R_{j-1}(v_{xy})$,
 - otherwise, $R_j(v_{xy}) := \{(q_1, q_2, I) \in R(v_{xy}) \mid (q_1, q_2, I) \text{ compatible with some } (q, A) \in G(i_j)\}$.

In Figure 3, we have that, if $i_1 = 1$ then every tuple is in the relation R_1 except $(q_0, q_1, \{\}) \in R(v_{11})$. Furthermore by Definition 19, we can reformulate Proposition 17 such that i_{j+1} is the smallest node in w for which there is a tuple $(q_1, q_2, I) \in R_j(i_{j+1})$ with $I \neq \emptyset$. Notice that if $R_j(i_j)$ itself contains such a tuple, then $i_{j+1} = i_j$. Otherwise, we can compute i_{j+1} by traversing the tree N_w^{aux} using the following lemma.

Lemma 20. For a node v_{xy} of N_w^{aux} , we can compute $R_j(v_{xy})$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$ in each of the following cases:

- (1) v_{xy} is a leaf, $v_{xy} = i_j$, and we know $G(i_j)$ and $R_{j-1}(i_j)$;
- (2) v_{xy} has parent v , $x > i_j$, and we already know $R_j(v)$; and
- (3) v_{xy} has child v , $y \geq i_j$, and we know $R_j(v)$ and $R_{j-1}(v_{xy})$.

In the following we argue that we need at most $\log n$ operations of the kind (1) to (3) to find i_{j+1} from i_j . We start at node i_j where $G(i_j)$ and $R_{j-1}(i_j)$ are known. We compute $R_j(i_j)$ using (1) and test whether $i_{j+1} = i_j$. If this is not the case we follow the path p from i_j to the root of N_w^{aux} and calculate R_j on the way. Since we always calculate the new relation R_j for every node on p we can always apply case (3). Because p is of length $\log n$ this needs $\log n$ operations. Afterwards, we do a second bottom-up traversal of p and, at each node, compute R_j for every right child (applying case (2)). We stop when we find such a right child v_r which is not on p and where $R_j(v_r)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. Again, this can be done with at most $\log n$ operations. By definition of R_j , we know that the subtree rooted at v_r has at least one leaf node u such that $R_j(u)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. The leftmost such leaf u will be i_{j+1} . To arrive at i_{j+1} , we go down from v_r . On this path, we always compute R_j for both children (applying case (2)) and choose the leftmost child for which R_j has a tuple (q_1, q_2, I) with $I \neq \emptyset$. We are done when we reach a leaf. Altogether, we navigated through $\mathcal{O}(\log n)$ nodes in the tree.

The following characterization demonstrates how we can obtain $G(i_{j+1})$ from $G(i_j)$ using j -relevant tuples:

$$G(i_{j+1}) = \{(q_2, A \cup \{(i_{j+1}, q_2)\}) \mid \exists (q_1, q_2, \{q_2\}) \in R_j(i_{j+1}) \\ \exists (q, A) \in G(i_j), \text{ and } q_1 \in \delta^*(q, w[i_j + 1..i_{j+1}])\}$$

By maintaining reachable states in δ^* when going from i_j to i_{j+1} , we can compute i_{j+1} from $G(i_j)$ within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$. This leads to the following.

Lemma 21. Given N_w^{aux} , T^+ , and $\ell \in [k]$, we can compute $G(i_\ell)$ in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot \ell \log n)$.

The additional term $|S| \cdot k!$ comes from the size of the $G(i_\ell)$ which is naively $\mathcal{O}(|Q|^k)$ but can be shown to be $\mathcal{O}(|S| \cdot k!)$. Combining Lemma 18 and 21 we then have the following.

Theorem 22. Given N_w^{aux} and T^+ , we can compute the first answer of M on w in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$.

In particular, we have that $\text{Complete}(\{\emptyset\})$ returns the set $\{A \mid (q, A) \in G(i_k)\}$.

3.5 From One Answer to the Next

The previous section showed how to compute $\text{Complete}(\{\emptyset\})$, i.e., the first answer(s) of the query on w . We now show how to go from

one answer to the next, i.e., the details of the procedure $\text{Next}(\mathcal{A})$ in Algorithm 1.

Lemma 23. Complete, Back, and Nextnode can be implemented such that Algorithm 1 correctly computes $\text{Next}(\mathcal{A})$. Furthermore, $\text{Next}(\mathcal{A})$ runs in $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$ time.

Proof sketch. To this end, recall from Section 3.3 that every $\mathcal{A}$ at each call of Complete, Back, or Nextnode has the property that all $A \in \mathcal{A}$ use the same multiset of nodes $\{i_1, \dots, i_\ell\}$. We denote this multiset by $\text{Nodes}(\mathcal{A})$ and assume that the following information is available at the time we call Complete, Back, Nextnode, or $\text{Next}(\mathcal{A})$: the tree N_w^{aux} with T^+ (entirely), the relations R_j , and sets G as described in the invariants (I1) and (I2) below.

- (I1) Let A_w^{aux} be the tree induced by all ancestors in N_w^{aux} of nodes in $\text{Nodes}(\mathcal{A})$. For every $v_{xy} \in A_w^{\text{aux}}$, we know the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$ where $i_j \in \text{Nodes}(\mathcal{A})$ is the maximal node with $x \leq i_j \leq y$.
- (I2) For every $i_j \in \text{Nodes}(\mathcal{A})$, we know $G(i_j)$. Here, $G(i_j) = \{(q, A) \mid \text{Nodes}(A) \subseteq \text{Nodes}(\mathcal{A}), |A| = j, \text{ and there is an annotated answer } A^{\text{full}} \text{ such that } \text{Nodes}(A_{[1, i_j-1]}) = \text{Nodes}(A_{[1, i_j-1]}^{\text{full}}), \text{Nodes}(A_{[i_j, i_j]}) \subseteq \text{Nodes}(A_{[i_j, i_j]}^{\text{full}}), \text{ and } (q, A) \text{ is compatible with } A^{\text{full}}\}$.

From Section 3.4 we can infer that (I1) and (I2) hold after calling $\text{Complete}(\{\emptyset\})$. Furthermore, we can generalize the description in Section 3.4 to compute $\text{Complete}(\mathcal{A})$ for an arbitrary $\mathcal{A}$ occurring in Algorithm 1. To this end, we have to change the definition of the tuple $(i_1, \dots, i_k)$ in Section 3.4. In particular, $(i_1, \dots, i_k)$ should be the smallest answer of the query such that $\{i_1, \dots, i_j\} = \text{Nodes}(\mathcal{A})$ and, for every $\ell > j$, i_ℓ is at least the largest number i_j in $\text{Nodes}(\mathcal{A})$. (Notice that we only have that $\text{Nodes}(\mathcal{A}) = \emptyset$ in the very first call of Complete, at line 4 of Algorithm 1.) Therefore, we can leave all the $G(i_1), \dots, G(i_j)$ untouched and only recompute the sets $G(i_\ell)$ for $\ell > j$. This concludes the description of $\text{Complete}(\mathcal{A})$. If (I1) and (I2) hold before calling $\text{Complete}(\mathcal{A})$, they also hold after the call is completed.

The procedure $\text{Back}(\mathcal{A})$ is implemented as follows:

$$\text{Back}(\mathcal{A}) = \begin{cases} \{(A \mid (q, A) \in G(i_{j-1}))\} & \text{if } j \geq 2, \\ \emptyset & \text{otherwise.} \end{cases}$$

If (I1) and (I2) are satisfied before calling $\text{Back}(\mathcal{A})$, they are also satisfied afterwards since we do not touch any R and G . Correctness holds using (I2).

Finally, for the implementation of $\text{Nextnode}(\mathcal{A})$, let i_j be maximal in $\text{Nodes}(\mathcal{A})$. Then, $\text{Nextnode}(\mathcal{A})$ checks whether there is an annotated answer A^{full} with $\text{Nodes}(A^{\text{full}}) = \{i_1, \dots, i_{j-1}, i_{j+1}, \dots, i_k\}$ for nodes $i_{j+1}, \dots, i_k$ larger i_j . It returns the following set of incomplete answers, if it exists:

$$\text{Nextnode}(\mathcal{A}) = \begin{cases} \{(A \mid (q, A) \in G(i_{j+1}))\} & \text{if } A^{\text{full}} \text{ exists,} \\ \emptyset & \text{otherwise.} \end{cases}$$

Notice that, $\text{Nextnode}(\mathcal{A})$ does recompute only the single node i_{j+1} and the set $G(i_{j+1})$. The computation is analogous to the one in Section 3.4.2 with only one difference: One has to ensure that i_{j+1} is strictly larger than i_j ; if it exists. This can be done by skipping the step where we check whether $i_{j+1} = i_j$ in the beginning of the computation. Again, if (I1) and (I2) are satisfied before calling $\text{Nextnode}(\mathcal{A})$, they are also satisfied afterwards. By Lemma 9, it directly follows that the computation of $\text{Next}(\mathcal{A})$ in Algorithm 1 is correct. Again, the term $|S| \cdot k!$ in the runtime is due to the size of the sets G of growing answers. $\square$

We therefore obtain the following main result.

Theorem 24. INCENUM for a k -NSFA M and a word w with $|w| = n$ can be solved with auxiliary data of size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot n)$ which can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot n)$, maintained within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ per update, and which guarantees delay $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$ between answers.

Here, Q is the state set of M and S is the set of selecting tuples of M . If M is constant, then the delay is $\mathcal{O}(\log n)$.

4. Incremental Enumeration for Trees

We extend the algorithm from Section 3 to trees. More precisely we show that, for a k -NSTA M (with states Q and selecting tuples S) and a tree t we can enumerate answers with $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$ delay, using an auxiliary data structure of size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot |t|)$ that can be updated within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$. This generalizes a result by Balmin et al. [2] who showed the following:

Theorem 25. [2] INCEVAL for an NTA N and tree t can be solved with auxiliary data of size $\mathcal{O}(|Q|^2 \cdot |t|)$ which can be updated in time $\mathcal{O}(|Q|^3 \cdot \log^2 |t|)$ per update.

We generalize Theorem 25 in two directions: from boolean queries to k -ary queries and we show that answers can be enumerated with small delay. The main observation in this section is that the techniques of Balmin et al. can be used together with the methods we developed in Section 3. Roughly, Balmin et al. maintain a set of NFAs over *heavy paths* in the tree t . Denote by t_v the subtree of t rooted at node v . For a node v , the *heavy path* $\text{hp}(v)$ of v is defined as follows [11, 19]:

- v belongs to $\text{hp}(v)$;
- if $v' \in \text{hp}(v)$ has children v_1 and v_2 , then v_1 belongs to $\text{hp}(v)$ if $|t_{v_1}| \geq |t_{v_2}|$.

A heavy path of v is *maximal* if it is not included in another heavy path (i.e., in the heavy path of v 's parent). The (*maximal*) *heavy path* of t , denoted $\text{hp}(t)$, is the path $\text{hp}(r)$ where r is the root of t . The set $\text{HPaths}(t)$ is the set of all maximal heavy paths of nodes in t . For a binary tree t , the set $\text{HPaths}(t)$ can be calculated in time and space linear in t . In Figure 4 (left), we illustrate the set $\text{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$ for the tree t : each heavy path is encircled and depicted by a separate shape of nodes.

Balmin et al. evaluate an NTA $N = (Q, \Sigma, \delta, F)$ on a tree t by encoding it into NFAs on maximal heavy paths of t (Sec. 5 in [2]). The NFAs operate on an enhanced alphabet that, for every $v \in t$, stores the label of v and all states of the NTA reachable at v by reading the subtree t_v rooted at v in a bottom-up way. These new labels can be calculated during one bottom-up pass through the tree in time $\mathcal{O}(|Q| \cdot |\delta| \cdot |t|)$. Then, every NFA simulates a part of the tree automaton by only allowing transitions compatible with the tree automaton on its path. If a node v changes its label, then the auxiliary structure for the NFA responsible for the heavy path containing v receives an update similar to Section 3.1. The change to this structure can in turn trigger changes in all auxiliary structures for NFAs on heavy paths closer to the root. However, the number of these heavy paths is at most logarithmic in $|t|$.

Lemma 26 (Lemma 1 in [19]). *Let t be a binary tree. The maximum number of distinct maximal heavy paths crossed by any path from the root of t to some leaf is at most $\log |t|$.*

In short, processing an update for incremental NTA evaluation essentially boils down to processing $\log |t|$ updates on words for NFAs, which explains the complexity upper bound in Theorem 25. We fix the following notation for the remainder of the section. We denote by $M = ((Q, \Sigma, \delta, F), S)$ a k -NSTA, by t the binary input tree, and by Q_S the set $\bigcup_{s \in S} \text{set}(s)$.

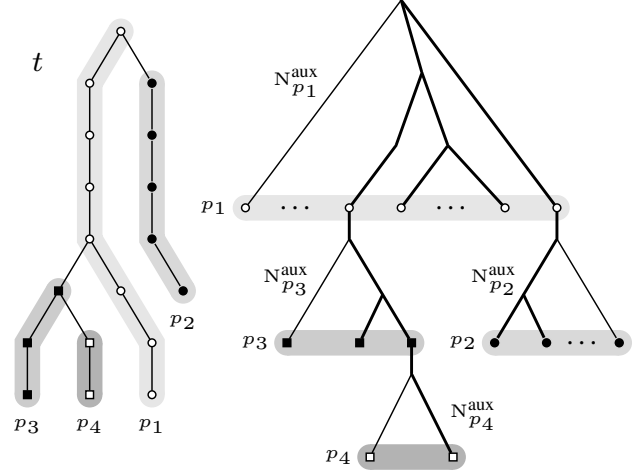


Figure 4. An input tree t with $\text{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$. In our algorithm we traverse the transition relation trees $N_{p_i}^{\text{aux}}$ for paths $p_i \in \text{HPaths}(t)$ which are linked according to the alignment of all maximal heavy paths in t .

4.1 Preprocessing: The Dynamic Auxiliary Structure

The first step in preprocessing is that we store, for each node v of t , a set of pairs $\text{Reach}(v) \subseteq (Q \times 2^Q)$ defined as follows:

- if v is a leaf:
 $\text{Reach}(v) = \{(q, I) \mid (\text{lab}(v) \rightarrow q) \in \delta, I = \{q\} \cap Q_S\}$
- if v has children v_1 (left) and v_2 (right):
 $\text{Reach}(v) = \{(q, I) \mid ((q_1, q_2, \text{lab}(v)) \rightarrow q) \in \delta, (q_1, I_1) \in \text{Reach}(v_1), (q_2, I_2) \in \text{Reach}(v_2) \text{ and } I = I_1 \cup I_2 \cup (\{q\} \cap Q_S)\}$

In other words, if we denote by M^q the NTA $(Q, \Sigma, \delta, \{q\})$, then $\text{Reach}(v)$ contains all pairs (q, I) such that $t_v \in L(M^q)$, $I \subseteq Q_S$, and there is a run of M^q on t_v that uses all states in I . By following the above definition, one can compute the sets $\text{Reach}(v)$ for all $v \in t$ in time $\mathcal{O}(|Q| \log |Q| \cdot |\delta| \cdot 2^k \cdot |t|)$.

Analogously to Balmin et al., we define a new labeling function $\text{lab}'(t)$ for all nodes $v \in t$ which constitutes the alphabet on which the NFAs will operate. Let $v_n \dots v_1$ be a heavy path in $\text{HPaths}(t)$ such that v_1 is a leaf and v_n is closest to the root. Then lab' is inductively defined as follows:

- for $i = 1$, $\text{lab}'(v_1) := \text{lab}(v_1)$,
- for $i > 1$, let v'_{i-1} be the child of v_i not on $\text{hp}(v_i)$,
 - if v'_{i-1} is the right child of v_i , then
 $\text{lab}'(v_i) := (\text{lab}(v_i), \text{Reach}(v'_{i-1}))$, and
 - if v'_{i-1} is the left child of v_i , then
 $\text{lab}'(v_i) := (\text{Reach}(v'_{i-1}), \text{lab}(v_i))$.

We define an NFA N_p for every path $p \in \text{HPaths}(t)$. The NFAs N_p read the word $\text{lab}'(v_1) \dots \text{lab}'(v_n)$ (where $p = v_n \dots v_1$), use a common state set $Q \uplus \{q_0\}$ (where q_0 is a fresh initial state), and use a common transition function δ_N :

- $\delta_N(q_0, a) := Q_0$ where $Q_0 = \{q \mid (a \rightarrow q) \in \delta\}$,
- $\delta_N(q, (a, R)) := \bigcup_{(q', I) \in R} \delta(q, q', a)$ for all $a \in \Sigma$,
- $\delta_N(q, (R, a)) := \bigcup_{(q', I) \in R} \delta(q', q, a)$ for all $a \in \Sigma$.

So, each NFA N_p simulates the tree automaton on path p by only allowing transitions compatible with the tree automaton. Note that our definition of δ_N is analogous to the one used by Balmin et al.

except that we consider an even more extended labeling function. Its alphabet is of size $\mathcal{O}(|\Sigma| \cdot 2^{|Q|} \cdot 2^k)$. However, we will not store the entire alphabet or the transition function of the NFAs explicitly. Instead, we store the sets $\text{Reach}(v)$ for every node $v \in t$ and compute δ_N on-the-fly from the tree automaton.

Let Δ be the alphabet of the labeling function lab' . Then we define $N_{\text{hp}(t)} = (Q \cup \{q_0\}, \Delta, \delta_N, q_0, F)$. The NFA $N_{\text{hp}(t)}$ accepts the word $\text{lab}'(v_1) \cdots \text{lab}'(v_n)$ if and only if the k -NSTA M accepts t . For all other paths $p \in \text{HPaths}(t)$ we define $N_p = (Q \cup \{q_0\}, \Delta, \delta_N, q_0, Q)$. For these paths the automata are needed for propagating the correct updates to the new labeling function $\text{lab}'(t)$.

This concludes our description of the NFAs that we will maintain in the auxiliary structure. Next we discuss how we do this. For every NFA N_p and path p , we build an auxiliary tree N_p^{aux} as in Definition 1. Then we compute the accompanying relations T_p^+ (Def. 4) using the $\boxtimes^+$ -operation (Sec. 3.2) as before. The only difference with Section 3.2 is how we initialize the relations in the leaf nodes, because leaf nodes in N_p^{aux} are no longer nodes in a word but nodes in t which can have subtrees below them. (We again use the convention that leaves $v_{xx} \in N_p^{\text{aux}}$ are the nodes on p in tree t). We define

$$T_p^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta_N(q_1, \text{lab}'(v_{xx})) \text{ and} \\ I = (\cup_{(a,J) \in \text{lab}'(v_{xx})} J) \cup (\{q_2\} \cap Q_S)\}.$$

The sets T_p^+ for all other nodes v_{xy} of N_p^{aux} are defined exactly as in Section 3.2. This finishes the preprocessing step. The auxiliary structure therefore includes t , the set $\text{HPaths}(t)$, and the auxiliary trees N_p^{aux} with relations T_p^+ for each NFA N_p .

Figure 4 (right) depicts the auxiliary data structure for the tree in Figure 4 (left). Heavy paths in the left tree correspond to nodes of the same shape on the right. The auxiliary data structure can therefore be seen as a “tree of trees” in which, e.g., the root of $N_{p_4}^{\text{aux}}$ provides information for a leaf node of $N_{p_3}^{\text{aux}}$, etc. The auxiliary data can be maintained under updates by propagating changes in a bottom-up fashion through all these trees. Assume that the highest node of p_4 is relabeled. This corresponds to a relabeling of the rightmost leaf of $N_{p_4}^{\text{aux}}$. This change is then propagated on all nodes on the path to the root of $N_{p_1}^{\text{aux}}$, going through the auxiliary structures $N_{p_3}^{\text{aux}}$ and $N_{p_2}^{\text{aux}}$. In principle, this procedure is very similar to the incremental evaluation algorithm of tree automata as described by Balmin et al. [2]. The only difference is that we maintain more involved sets $\text{Reach}(v)$ (which explain the extra 2^k factor in complexity). Notice that $k = 0$ in [2].

Lemma 27. *Given a k -NSTA M and a binary tree t , the auxiliary structure has size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot |t|)$, can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot |t|)$ and updated in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$.*

4.2 Enumerating Answers

The construction of the auxiliary structure is such that we can enumerate the answers of $M(t)$ in a similar way as it is done on words in Section 3. The main differences are that (1) we now have to maintain several auxiliary trees N_p^{aux} and their T_p^+ (see Sec. 4.1); (2) our enumeration procedure has to check, for a leaf of some N_p^{aux} , whether to stop or go to the next tree $N_{p'}^{\text{aux}}$.

Note that similar to the word case, the relation $T_{\text{hp}(t)}^+$ of $N_{\text{hp}(t)}^{\text{aux}}$ contains a tuple $(q_0, q_F, \text{set}(s))$ if and only if there exists an accepting run of the k -NSTA M which produces an answer for selecting tuple s on t . To enumerate all these answers we traverse the auxiliary trees. Therefore, we construct Algorithm 1 from Section 3 such, that we can use it for an auxiliary tree which we will build from the trees N_p^{aux} (see Fig. 4 (right)). However, we have to redefine the relevant relation R in this case such that it fits to

the tree automaton and the input tree. The new definition differs from Definition 12 only at the root nodes of the trees N_p^{aux} for every $p \in \text{HPaths}(t)$.

Definition 28. Let $p = v_n \cdots v_1 \in \text{HPaths}(t)$ (v_1 is a leaf). Then, we define for the root node r of N_p^{aux}

- if $p = \text{hp}(t)$:
 $R(r) = \{(q_0, q_F, \text{set}(s)) \in T_p^+(r) \mid q_F \in F, s \in S\}$
- if $p \neq \text{hp}(t)$: consider the parent v_p of v_n in t , then
 - if v_n is a left child: $R(r) = \{(q_0, q_1, I') \mid \exists(q_0, q_1, I) \in T_p^+(r) \text{ with } I' \subseteq I, \exists(q_2, q, J) \in R(v_p), q \in \delta(q_1, q_2, \text{lab}(v_p)), \text{ and } J \subseteq I' \cup \{q\}\}$
 - if v_n is a right child: $R(r) = \{(q_0, q_2, I') \mid \exists(q_0, q_2, I) \in T_p^+(r) \text{ with } I' \subseteq I, \exists(q_1, q, J) \in R(v_p), q \in \delta(q_1, q_2, \text{lab}(v_p)), \text{ and } J \subseteq I' \cup \{q\}\}$

Notice that the above definition admits that $q \in I'$. Intuitively, the tuples in the above relation are associated with partial runs of the tree automaton that are relevant for constructing an answer to the query $M(t)$.

When we want to enumerate answers of M on t , we consider a tree as depicted in the right of Figure 4 (which is composed of all trees N_p^{aux}) and perform an enumeration procedure similar to the one on words. We refer to this tree as N_t^{aux} . However, nodes that can be selected by M no longer correspond to leaves of N_t^{aux} as in the word case. Now, also internal nodes of N_t^{aux} (but leaves of individual N_p^{aux}) can be selected. Therefore we extend the output order by comparing individual nodes in N_t^{aux} in terms of their postfix order. Then, for every visited tree in this traversal, we run Algorithm 1 where we interpret the relation R as given in Definition 28. The postfix-order on the nodes ensures that the Definition of j -relevant tuples and sets G can be used as defined in Section 3.4. In this way, we can compute i_{j+1} when $G(i_j)$ is given in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$ according to the size of the auxiliary structure for trees. A set of complete answers, e.g., $G(i_k)$, is computed in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$. This gives us the following result.

Theorem 29. *INCENUM for a k -NSTA M and a tree t can be solved with auxiliary data of size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot |t|)$ which can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot |t|)$ and maintained in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$ per update, and which guarantees at most $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$ delay between answers.*

Notice that, for practically important query languages such as core XPath queries or variants of regular XPath (see [3] for a survey), we have that $k = 2$, for which the theorem gives an $\mathcal{O}(|Q|^2 \cdot |t|)$ upper bound on the auxiliary data and at most $\mathcal{O}(|Q|^3 \cdot \log^2 |t|)$ delay between answers.

5. Concluding Remarks and Further Directions

All our algorithms work equally well when we would consider query evaluation under *multiset semantics*. The result of M on w under multiset semantics is denoted $M_{\text{ms}}(w)$ and is a function that maps tuples $(i_1, \dots, i_k) \in [n]^k$ to $\mathbb{N}$. More precisely, for each $(i_1, \dots, i_k) \in [n]^k$, we define

$$M_{\text{ms}}(w)((i_1, \dots, i_k)) = |\{(p_1, \dots, p_k) \in S \mid \text{there is an} \\ \text{accepting run } r \text{ of } N \text{ on } w \text{ such that,} \\ \text{for every } \ell \in [k], r \text{ visits } i_\ell \text{ in } p_\ell\}|$$

(and similarly for trees). Intuitively, the multiset contains a tuple $(i_1, \dots, i_k)$ as often as there are selecting tuples and runs that select it. For example, for the 2-NSFA M in Figure 1 and the word

$w = abcd$, we have that

$$M_{\text{ms}}(w) = \{(1, 2), (1, 2), (2, 1), (2, 1), (1, 3), (3, 1), (2, 4), (4, 2)\}.$$

Thereby, the difference between the enumeration procedure for set semantics or multiset semantics only lies in the procedure output($\mathcal{A}$) in Algorithm 1. Either we output every tuple once (set semantics) or we output every tuple as often as we have an annotated answer for it (multiset semantics).

Towards future work, we want to investigate if our techniques can be generalized towards graphs with bounded treewidth, using the generalization in Bagan [1]. 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. Furthermore, a single node relabel in the graph can induce $m > 1$ relabels in the tree decomposition which will influence complexity. 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.

Finally we want to investigate whether we can efficiently maintain the number of answers to a query (under set or multiset semantics). Notice that the number of times that a tuple $(i_1, \dots, i_k)$ is in the answer under multiset semantics is simply the number of tuples in $G(i_k)$. Computing the number of answers efficiently can be interesting if we want to decide whether a constant-delay algorithm with linear time preprocessing would be able to output the whole output faster than our logarithmic-delay algorithm which would not require preprocessing after an update. Roughly, when the output of a query contains at most $\mathcal{O}(n/\log n)$ outputs, the logarithmic-delay algorithm will finish more quickly than a constant-delay procedure with linear preprocessing. Moreover, the logarithmic-delay algorithm produces the first answers more quickly. Estimating the number of answers to a query can therefore help to decide which kind of procedure is desirable.

Acknowledgment

We thank Matthias Niewerth for many useful comments on a draft of this paper.

References

- [1] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Computer Science Logic (CSL)*, pages 167–181, 2006.
- [2] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems (TODS)*, 29(4):710–751, 2004.
- [3] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys (CSUR)*, 41(1):3:1–3:54, 2009.
- [4] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. *ACM Transactions on Database Systems (TODS)*, 35(4):29:1–29:43, 2010.
- [5] M. Bojanczyk and D. Figueira. Efficient evaluation for a temporal logic on changing XML documents. In *Symposium on Principles of Database Systems (PODS)*, pages 259–270, 2011.
- [6] T. Colcombet. A combinatorial theorem for trees. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 901–912, 2007.
- [7] B. Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.
- [8] A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Computer Science Logic (CSL)*, pages 189–202, 2011.
- [9] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.
- [10] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Logic in Computer Science (LICS)*, page 188, 2003.
- [11] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing (SICOMP)*, 13(2):338–355, 1984.
- [12] W. Kazana and L. Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Symposium on Principles of Database Systems (PODS)*, pages 297–308, 2013.
- [13] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):25:1–25:12, 2013.
- [14] F. Neven. *Design and Analysis of Query Languages for Structured Documents*. PhD thesis, Limburgs Universitair Centrum, 1999.
- [15] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *International Conference on Database Programming Languages (DBPL)*, pages 217–231, 2005.
- [16] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences (JCSS)*, 55(2):199–209, 1997.
- [17] L. Segoufin. Enumerating with constant delay the answers to a query. In *International Conference on Database Theory (ICDT)*, pages 10–20, 2013.
- [18] I. Simon. Factorization forests of finite height. *Theoretical Computer Science (TCS)*, 72(1):65–94, 1990.
- [19] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences (JCSS)*, 26(3):362–391, 1983.
- [20] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–82, 1968.