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 $O(\log^2 n)$

http://dx.doi.org/10.1145/

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 $\mathcal{O}(\log^2 n \cdot |N|^3)$ 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 $t_1, \ldots, t_m$ 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 $\{1, \ldots, n\}$. 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 $m : A \to \mathbb{N}$. 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 $\}\!\!\}$. 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 $w = a_1 \cdots a_n$ of labels from Σ. To a word w we associate a *set of nodes* $\mathrm{Nodes}(w) = \{v_1, \ldots, v_n\}$ such that each node v_i bears the label $\mathrm{lab}(v_i) = a_i$. Since nodes in words are linearly ordered (due to the structure of the word) we often take $\mathrm{Nodes}(w) = \{1, \ldots, n\}$ to simplify notation. However, our results do not require that $\mathrm{Nodes}(w) = \{1, \ldots, n\}$. For v_i, v_j with $1 \leq i \leq j \leq n$ we denote by $w[v_i..v_j]$ the subword $a_i \cdots a_j$.

Trees in this paper are labeled, rooted, and binary. For every tree t, we denote the *set of nodes of t* by $\mathrm{Nodes}(t)$ 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 $\mathrm{lab}(v)$.

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

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 $Q \times \Sigma \to 2^Q$. 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 $w = a_1 \cdots a_n$ is a word $r = q_0 \cdots q_n \in Q^*$ 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

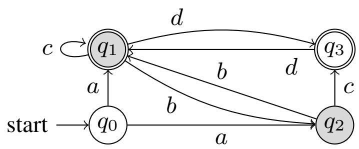

Figure 1. 2-NFSA M with $S = \{(q_1, q_2), (q_2, q_1)\}$.

are either of the form $(q_1, q_2, a) \to q$ or $a \to q$, 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 λ : $\text{Nodes}(t) \to Q$ such that for every $v \in \text{Nodes}(t)$ the following holds: if v is a leaf, then $\text{lab}(v) \to \lambda(v) \in \delta$; if v has children v_1 and v_2 then $(\lambda(v_1), \lambda(v_2), \text{lab}(v)) \to \lambda(v) \in \delta$. 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

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

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* 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 Aux(d) to 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 Aux(d), *(ii)* time needed to compute Aux(d), *(iii)* time needed to update Aux(d) to 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 $w = a_1 \cdots a_n \in \Sigma^*$. 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 $\text{Nodes}(w) = \{1, \ldots, n\}$, the *auxiliary tree* $\mathrm{N}_w^{\text{aux}}$ is defined as follows:

- the root of $\mathrm{N}_w^{\text{aux}}$ 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 $\mathrm{N}_w^{\text{aux}}$. That is, the nodes of w are leaves in $\mathrm{N}_w^{\text{aux}}$.

Every node v_{xy} in $\mathrm{N}_w^{\text{aux}}$ 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 $v_{xy} \in \text{Nodes}(\mathrm{N}_w^{\text{aux}})$, then the *transition relation* $\mathrm{T}(v_{xy})$ is defined as:

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

Thus, $(q_1, q_2) \in \mathrm{T}(v_{xy})$ 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 $\mathrm{T}(v_{xy})$ from $\mathrm{T}(v_{vz})$ and $\mathrm{T}(v_{(z+1)y})$ in time $\mathcal{O}(|Q|^3)$ (this corresponds to joining two binary relations). Since $\mathrm{N}_w^{\text{aux}}$ has $2n - 1$ nodes and $\mathcal{O}(\log n)$ depth, $\mathrm{N}_w^{\text{aux}}$ and T can be computed in time $\mathcal{O}(|Q|^3 \cdot n)$. Finally, $w \in L(N)$ if and only if $(q_0, q_F) \in \mathrm{T}(v_{1n})$ 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 $w = a_1 \cdots a_{x-1} b$ $a_{x+1} \cdots a_n$. 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 $\mathrm{N}_w^{\mathrm{aux}}$, 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 $\mathrm{N}_w^{\mathrm{aux}}$ such that we can use it to enumerate answers for a k-NFSA 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., $Q_S = \cup_{s \in S} \mathrm{set}(s)$.

The structure is based on the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$ 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 $v_{xy} \in \mathrm{N}_w^{\mathrm{aux}}$, we define $\mathrm{T}^+(v_{xy})$ 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 $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.

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

- If $x = y$ then $\mathrm{T}^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta(q_1, a_x)$ 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 $\mathrm{N}_w^{\mathrm{aux}}$, then $\mathrm{T}^+(v_{xy}) = \mathrm{T}^+(v_1) \bowtie^+ \mathrm{T}^+(v_2)$.

Here, we define $\mathrm{T}^+(v_1) \bowtie^+ \mathrm{T}^+(v_2) := \{(q_1, q_2, I) \mid \exists p \in Q, \exists I_1, I_2 \subseteq Q_S, \exists s \in S$ such that $(q_1, p, I_1) \in \mathrm{T}^+(v_1)$, $(p, q_2, I_2) \in \mathrm{T}^+(v_2)$, and $I = (I_1 \cup I_2) \cap \mathrm{set}(s)\}$. (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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $M(w)$

```
Enum(M, w) {
Input: k-NSFA M = ((Q, Σ, δ, F), S), word w
Output: Enumeration of all answers in M(w)
𝒜 = Complete({∅})
while 𝒜 ≠ ∅ do
    output(𝒜)
    𝒜 = Next(𝒜)
}
Next(𝒜) {
Input: set 𝒜 of annotated answers
Output: set of smallest annotated answers larger than 𝒜
while Nextnode(𝒜) = ∅ do
    𝒜 ← Back(𝒜)
    if 𝒜 = ∅ then return ∅
return Complete(Nextnode(𝒜))
}
```

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 $t = (i_1, \ldots, i_k) \in \mathbb{N}^k$, let $\mathrm{sort}(t)$ denote the word obtained by sorting $i_1, \ldots, i_k$ in increasing order and concatenating the result (as a word in $\mathbb{N}^*$). More precisely, $\mathrm{sort}(t) = i_{\sigma(1)} \cdots i_{\sigma(k)}$ 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, $\mathrm{sort}((5, 2, 2, 3, 12)) = 2\ 2\ 3\ 5\ 12$. The total order $\preceq$ between tuples $(i_1, \ldots, i_k)$ is defined as the lexicographical order on $\mathrm{sort}((i_1, \ldots, i_k))$ (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 $\mathrm{Nodes}(w) \times Q$. 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 $\mathrm{Nodes}(w) \times Q$ of the form

$$\{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$$

such that there is an accepting run r of N on w and a $(q_1, \ldots, q_k) \in S$ such that r visits i_ℓ in q_ℓ, for every $\ell \in [k]$. It holds that $|A^{\mathrm{full}}| = k$. 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 $\mathrm{Nodes}(w) \times Q$, we denote by $\mathrm{Nodes}(A)$ the multiset of nodes in A. That is, for $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ we have that $\mathrm{Nodes}(A) = \{\!\{i_1, \ldots, i_k\}\!\}$.

We extend the order $\preceq$ to multisets over $\mathrm{Nodes}(w) \times Q$. For two such multisets A and B, we say that $A \preceq B$ if $\mathrm{Nodes}(A) \preceq \mathrm{Nodes}(B)$. 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 $\text{Nodes}(w) \times Q$.

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

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 **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 **Next**($\mathcal{A}$), we use the following ingredients in Algorithm 1. Let $A = \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\}$ 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 $A_{\text{del}} = \{\!\{(i_1, q_1), \ldots, (i_{j-1}, q_{j-1})\}\!\}$.

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 $A = \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\} \in \mathcal{A}$, annotate exactly the nodes $i_1, \ldots, i_{j-1}$ (if we assume i_j to be maximal).

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

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

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 **Next**($\mathcal{A}$) in Algorithm 1 returns*

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

Finally, the procedure $output(\mathcal{A})$ takes a set of annotated answers $\mathcal{A}$ and writes the set $\{(i_1, \ldots, i_k) \mid \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\} \in \mathcal{A} \text{ and } (q_1, \ldots, q_k) \in S\}$ 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. ***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 information in $\mathrm{N}_w^{\text{aux}}$ 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.

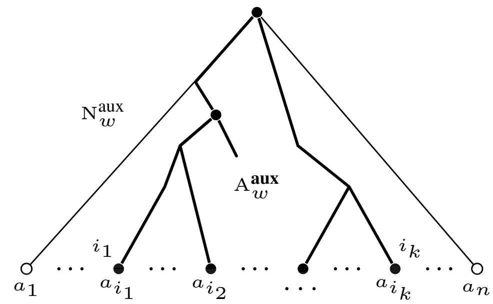

Figure 2. $\mathrm{A}_w^{\text{aux}}$ is the part of $\mathrm{N}_w^{\text{aux}}$ that is used to compute the first answers. (Note that $i_j = v_{i_j i_j}$.)

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 $(i_1, \ldots, i_k)$ (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 $\mathrm{A}_w^{\text{aux}}$ be the tree induced by all ancestors of nodes i_j in $\mathrm{N}_w^{\text{aux}}$. Hence, $\mathrm{A}_w^{\text{aux}}$ has at most k leaves, its root is the root of $\mathrm{N}_w^{\text{aux}}$, and each of its leaves corresponds to a node i_j. For obtaining $(i_1, \ldots, i_k)$, we perform a depth-first left-to-right traversal of $\mathrm{A}_w^{\text{aux}}$. Since the depth of $\mathrm{N}_w^{\text{aux}}$ 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 $\mathrm{A}_w^{\text{aux}}$ 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 $\mathrm{A}_w^{\text{aux}}$.

We first explain how to compute and traverse the leftmost path of $\mathrm{A}_w^{\text{aux}}$. We start at the root of $\mathrm{N}_w^{\text{aux}}$ 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 $v \in \mathrm{N}_w^{\text{aux}}$ the set of *relevant tuples of* v, denoted $R(v)$, is inductively defined as follows:

- $R(v_{1n}) = \{(q_0, q_F, \text{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}$;
- 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 $R_{v_1,v_2} = \{(q_1, q_2, J_1, q_2, q_3, J_2) \mid \exists (q_1, q_3, I) \in R(v), (q_1, q_2, I_1) \in \mathrm{T}^+(v_1), (q_2, q_3, I_2) \in \mathrm{T}^+(v_2)$ such that $J_1 \cup J_2 = I$, $J_1 \subseteq I_1$, and $J_2 \subseteq I_2\}$. Then,

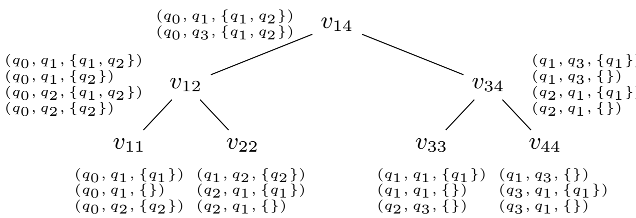

Figure 3. The relation R for the 2-NSFA M from Figure 1 and $w = abcd$.

$R(v_1) = \{(q_1, q_2, J_1) \mid (q_1, q_2, J_1, q_2, q_3, J_2) \in R_{v_1,v_2}\}$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $A^{\mathrm{full}} = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ and two nodes ℓ, r of w, the *projection of A^{full} onto* $[\ell, r]$, denoted $A^{\mathrm{full}}_{[\ell,r]}$, is defined to be the multiset $\{\!\{(i, q_i) \mid \ell \leq i \leq r\}\!\}$.

Lemma 13. *For every node $v_{xy} \in \mathrm{N}_w^{aux}$, 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 $I = \mathit{Nodes}(A^{full}_{[x,y]})$.*

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 $v \in \mathrm{N}_w^{aux}$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ but only among paths of $\mathrm{A}_w^{\mathrm{aux}}$.

3.4.1 The First Part of the First Answer

In order to find the leftmost path of $\mathrm{A}_w^{\mathrm{aux}}$, we start at the root of $\mathrm{N}_w^{\mathrm{aux}}$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $(i_2, i_1, \ldots)$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ of tuples over $\mathbb{N} \times Q$ onto Q, denoted $\pi_Q(A)$, is defined as $\{\!\{q_1, \ldots, q_k\}\!\}$.

Definition 16 (Compatibility)**.** Let v_{xy} be a node of $\mathrm{N}_w^{\mathrm{aux}}$. 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 $I = \mathrm{set}(\pi_Q(A^{\mathrm{full}}_{[x,y]}))$.

(Here, $I = \mathrm{set}(\pi_Q(A^{\mathrm{full}}_{[x,y]}))$ 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 $r(i) = q$, $A_{[1,i-1]} = A^{\mathrm{full}}_{[1,i-1]}$ and $A_{[i,i]} \subseteq A^{\mathrm{full}}_{[i,i]}$.

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 $G(i_{j+1})$:

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

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 $\mathcal{A}^{first}$ be the set of smallest annotated answers. Then, it holds that $G(i_k) = \{(q, A) \mid A \in \mathcal{A}^{first}$ and (q, A) 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 $v_{xy} \in \mathrm{N}_w^{\mathrm{aux}}$ and $j \in \{0, \ldots, k\}$, 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, $R_j(v_{xy}) := \{(q_1, q_2, I) \in R(v_{xy}) \mid (q_1, q_2, I)$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $\mathrm{N}_w^{\mathrm{aux}}$ 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 $O(\log n)$ nodes in the tree.

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

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

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 naïvely $\mathcal{O}(|Q|^k)$ 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 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 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 **Next**($\mathcal{A}$) in Algorithm 1.

Lemma 23. *Complete, Back, and Nextnode can be implemented such that Algorithm 1 correctly computes Next($\mathcal{A}$). Furthermore, 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 $\{\!\{i_1, \ldots, i_\ell\}\!\}$. We denote this multiset by Nodes($\mathcal{A}$) and assume that the following information is available at the time we call Complete, Back, Nextnode, or Next($\mathcal{A}$): the tree $\mathrm{N}_w^{\mathrm{aux}}$ with T^+ (entirely), the relations R_j, and sets G as described in the invariants (I1) and (I2) below.

(I1) Let $\mathrm{A}_w^{\mathrm{aux}}$ be the tree induced by all ancestors in $\mathrm{N}_w^{\mathrm{aux}}$ of nodes in Nodes($\mathcal{A}$). For every $v_{xy} \in \mathrm{A}_w^{\mathrm{aux}}$, we know the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$ where $i_j \in$ Nodes($\mathcal{A}$) is the maximal node with $x \leq i_j \leq y$.

(I2) For every $i_j \in$ Nodes($\mathcal{A}$), we know $G(i_j)$. Here, $G(i_j) = \{(q, A) \mid \text{Nodes}(A) \subseteq \text{Nodes}(\mathcal{A}), |A| = j$, and there is an annotated answer A^{full} such that $\text{Nodes}(A_{[1,i_j-1]}) = \text{Nodes}(A^{\text{full}}_{[1,i_j-1]})$, $\text{Nodes}(A_{[i_j,i_j]}) \subseteq \text{Nodes}(A^{\text{full}}_{[i_j,i_j]})$, and (q, A) is compatible with $A^{\text{full}}\}$.

From Section 3.4 we can infer that (I1) and (I2) hold after calling Complete($\{\emptyset\}$). Furthermore, we can generalize the description in Section 3.4 to compute Complete($\mathcal{A}$) for an arbitrary $\mathcal{A}$ occurring in Algorithm 1. To this end, we have to change the definition of the tuple $(i_1, \ldots, i_k)$ in Section 3.4. In particular, $(i_1, \ldots, i_k)$ should be the smallest answer of the query such that $\{i_1, \ldots, i_j\} =$ Nodes($\mathcal{A}$) and, for every $\ell > j$, i_ℓ is at least the largest number i_j in Nodes($\mathcal{A}$). (Notice that we only have that Nodes($\mathcal{A}$) $= \emptyset$ in the very first call of Complete, at line 4 of Algorithm 1.) Therefore, we can leave all the $G(i_1), \ldots, G(i_j)$ untouched and only recompute the sets $G(i_\ell)$ for $\ell > j$. This concludes the description of Complete($\mathcal{A}$). If (I1) and (I2) hold before calling Complete($\mathcal{A}$), they also hold after the call is completed.
The procedure 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 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 Nextnode($\mathcal{A}$), let i_j be maximal in Nodes($\mathcal{A}$). Then, Nextnode($\mathcal{A}$) checks whether there is an annotated answer $\mathcal{A}^{\text{full}}$ with Nodes(A^{full}) $= \{\!\{i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k\}\!\}$ for nodes $i_{j+1}, \ldots, i_k$ larger i_j. It returns the following set of incomplete answers, if it exists:

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

Notice that, 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 Nextnode($\mathcal{A}$), they are also satisfied afterwards. By Lemma 9, it directly follows that the computation of 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 $\mathcal{O}(|Q|^3 \cdot \log^2 \cdot |t|)$ 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* $\mathrm{hp}(v)$ of v is defined as follows [11, 19]:

- v belongs to $\mathrm{hp}(v)$;
- if $v' \in \mathrm{hp}(v)$ has children v_1 and v_2, then v_1 belongs to $\mathrm{hp}(v)$ 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 $\mathrm{hp}(t)$, is the path $\mathrm{hp}(r)$ where r is the root of t. The set $\mathrm{HPaths}(t)$ is the set of all maximal heavy paths of nodes in t. For a binary tree t, the set $\mathrm{HPaths}(t)$ can be calculated in time and space linear in t. In Figure 4 (left), we illustrate the set $\mathrm{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$ 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 $\cup_{s \in S} \mathrm{set}(s)$.

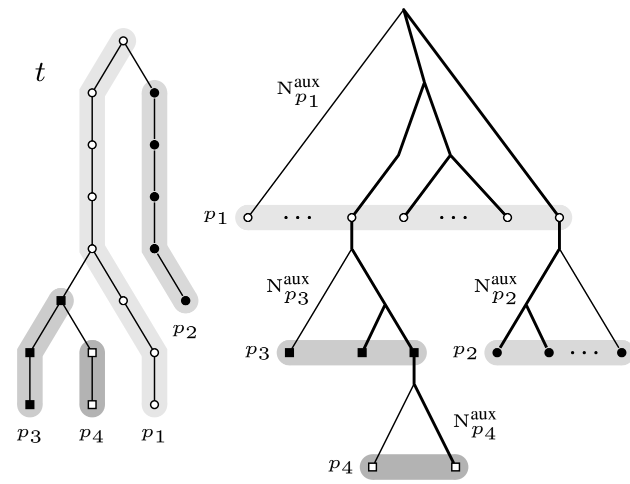

Figure 4. An input tree t with $\mathrm{HPaths}(t) = \{p_1, p_2, p_3, p_4\}$. In our algorithm we traverse the transition relation trees $\mathrm{N}^{\mathrm{aux}}_{p_i}$ for paths $p_i \in \mathrm{HPaths}(t)$ which are linked according to the alignment of all maximal heavy paths in t.

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 $\mathrm{Reach}(v) \subseteq (Q \times 2^Q)$ defined as follows:

- if v is a leaf:
  $\mathrm{Reach}(v) = \{(q, I) \mid (\mathrm{lab}(v) \to q) \in \delta, I = \{q\} \cap Q_S\}$
- if v has children v_1 (left) and v_2 (right):
  $\mathrm{Reach}(v) = \{(q, I) \mid ((q_1, q_2, \mathrm{lab}(v)) \to q) \in \delta,$
  $(q_1, I_1) \in \mathrm{Reach}(v_1), (q_2, I_2) \in \mathrm{Reach}(v_2)$
  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 $\mathrm{Reach}(v)$ 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 $\mathrm{Reach}(v)$ 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 $\mathrm{lab}'(t)$ for all nodes $v \in t$ which constitutes the alphabet on which the NFAs will operate. Let $v_n \cdots v_1$ be a heavy path in $\mathrm{HPaths}(t)$ such that v_1 is a leaf and v_n is closest to the root. Then lab' is inductively defined as follows:

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

We define an NFA N_p for every path $p \in \mathrm{HPaths}(t)$. The NFAs N_p read the word $\mathrm{lab}'(v_1) \cdots \mathrm{lab}'(v_n)$ (where $p = v_n \cdots v_1$), 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 $Q_0 = \{q \mid (a \to q) \in \delta\}$,
- $\delta_N(q, (a, R)) := \cup_{(q', I) \in R} \delta(q, q', a)$ for all $a \in \Sigma$,
- $\delta_N(q, (R, a)) := \cup_{(q', I) \in R} \delta(q', q, a)$ 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 $\bowtie^+$-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 $v_{xx} \in \text{N}_p^{\text{aux}}$ are the nodes on p in tree t). We define

$$\text{T}_p^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta_N(q_1, \text{lab}'(v_{xx})) \text{ and } I = (\cup_{(a,J)\in \text{lab}'(v_{xx})} J) \cup (\{q_2\} \cap Q_S)\}.$$

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 $\text{N}_{p_4}^{\text{aux}}$ provides information for a leaf node of $\text{N}_{p_3}^{\text{aux}}$, 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 $\text{N}_{p_4}^{\text{aux}}$. This change is then propagated on all nodes on the path to the root of $\text{N}_{p_1}^{\text{aux}}$, going through the auxiliary structures $\text{N}_{p_3}^{\text{aux}}$ and $\text{N}_{p_1}^{\text{aux}}$. 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 $\text{N}_{p'}^{\text{aux}}$.

Note that similar to the word case, the relation $\text{T}_{\text{hp}(t)}^+$ of $\text{N}_{\text{hp}(t)}^{\text{aux}}$ 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: $R(r) = \{(q_0, q_1, I') \mid \exists(q_0, q_1, I) \in \text{T}_p^+(r) \text{ with } I' \subseteq I, \exists(q_2, q, J) \in R(v_p), q \in \delta(q_1, q_2, \text{lab}(v_p)), \text{ and } J \subseteq I' \cup \{q\}\}$
  - if v_n is a right child: $R(r) = \{(q_0, q_2, I') \mid \exists(q_0, q_2, I) \in \text{T}_p^+(r) \text{ with } I' \subseteq I, \exists(q_1, q, J) \in R(v_p), q \in \delta(q_1, q_2, \text{lab}(v_p)), \text{ and } J \subseteq I' \cup \{q\}\}$

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 $(i_1, \ldots, i_k) \in [n]^k$ to $\mathbb{N}$. More precisely, for each $(i_1, \ldots, i_k) \in [n]^k$, we define

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

(and similarly for trees). Intuitively, the multiset contains a tuple $(i_1, \ldots, i_k)$ 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 $(i_1, \ldots, i_k)$ 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.

In this appendix we provide proofs and details for which there was no space in the body of the paper.

Proofs of Section 3 (Words)

Proofs of Section 3.2

Lemma 30. *For every $v_{xy} \in \mathrm{N}_w^{aux}$, the relation $\mathrm{T}^+(v_{xy})$ is the set of tuples $(q_1, q_2, I) \in (Q^2 \times 2^Q)$ 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 $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.*

Proof. Let, for a node v of $\mathrm{N}_w^{\mathrm{aux}}$, the depth $\mathrm{d}(v)$ of v, be the length of the path from the root of $\mathrm{N}_w^{\mathrm{aux}}$ to v. The proof is by induction on decreasing values of the depth, $\mathrm{d}(v_{xy})$, of nodes v_{xy} in the auxiliary tree $\mathrm{N}_w^{\mathrm{aux}}$.

For the base case, that is, a leaf node of $\mathrm{N}_w^{\mathrm{aux}}$, we have that

$$\mathrm{T}^+(v_{xx}) = \{(q_1, q_2, I) \mid q_2 \in \delta(q_1, a_x) \text{ and } I = \{q_2\} \cap (\cup_{s \in S} \mathrm{set}(s))\}.$$

Thus, the partial run $r = q_1 q_2$ on $w[x]$ proves the assumption.
For the induction case, we have to show that the assumption holds for $\mathrm{T}^+(v_{xy}) = \mathrm{T}^+(v_{xz}) \bowtie^+ \mathrm{T}^+(v_{(z+1)y})$ where $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$, and $\mathrm{T}^+(v_{xz})$ and $\mathrm{T}^+(v_{(z+1)y})$ are computed correctly. We show "$\Rightarrow$" and "$\Leftarrow$" separately.
"$\Rightarrow$": Let (q_1, q_2, I) be a tuple in $\mathrm{T}^+(v_{xy})$. Then, we know that there exist tuples $(q_1, p, I_1) \in \mathrm{T}^+(v_{xz})$ and $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$ such that $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$ for some $s \in S$. By induction hypothesis, we know that there exist

- a partial run $r_1 = q_1 \cdots p$ on $w[x..z]$ such that $I_1 = \mathrm{set}(r_1) \cap \mathrm{set}(s)$, and
- a partial run $r_2 = p \cdots q_2$ on $w[z+1..y]$ such that $I_2 = \mathrm{set}(r_2) \cap \mathrm{set}(s)$.

Since $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$, we can join r_1 and r_2 to a partial run $r = q_1 \cdots q_2$ on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$.
"$\Leftarrow$": Let s be a selecting tuple and $r = q_1 \cdots q_2$ be a partial run on $w[x..y]$ such that $I = \mathrm{set}(r) \cap \mathrm{set}(s)$. Then, we can decompose r in two partial runs r_1 and r_2 such that

- $r_1 = q_1 \cdots p$ on $w[x..z]$ such that $I_1 = \mathrm{set}(r_1) \cap \mathrm{set}(s)$,
- $r_2 = p \cdots q_2$ on $w[z+1..y]$ such that $I_2 = \mathrm{set}(r_2) \cap \mathrm{set}(s)$,
- and $(I_1 \cup I_2) \cap \mathrm{set}(s) = I$.

By applying the induction hypothesis on v_{xz} and $v_{(z+1)y}$, there exist tuples $(q_1, p, I_1) \in \mathrm{T}^+(v_{xz})$ and $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$. Therefore, by definition of $\bowtie^+$, there exists a tuple $(q_1, q_2, I) \in \mathrm{T}^+(v_{xy})$. This concludes the proof. $\square$

PROOF OF 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 $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$.*

Proof. The tree $\mathrm{N}_w^{\mathrm{aux}}$ has $O(n)$ nodes by construction. For every node $v \in \mathrm{N}_w^{\mathrm{aux}}$, the transition relation $\mathrm{T}^+(v)$ is of size $\mathcal{O}(|Q|^2 \cdot 2^k)$. Therefore, our auxiliary data has size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot n)$ in total. The relation T^+ can be computed in a bottom-up pass through $\mathrm{N}_w^{\mathrm{aux}}$. For leaf nodes, T^+ can be calculated in time and space $\mathcal{O}(|Q|^2)$. Afterwards, we need to compute n $\bowtie^+$-joins where each join can be done in time $\mathcal{O}(|Q|^3 \cdot 2^k)$. All together, $\mathrm{N}_w^{\mathrm{aux}}$ and T^+ can be built in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot n)$.

We now discuss an update. Therefore, consider an update at node v. Then, we update every relation on the path from v to the root of $\mathrm{N}_w^{\mathrm{aux}}$, i.e., we update $\log n$ many relations. This can be done by $\log n$ times a $\bowtie^+$-operation yielding a total time of $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ for one update. Analogously to the proof of Lemma 30, it follows that the relation T^+ is still correct after the update. $\square$

Proofs of Section 3.3

PROOF OF LEMMA 9: *Let $\mathcal{A}$ be a set of annotated answers. Then **Next**$(\mathcal{A})$ in Algorithm 1 returns*

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

Proof. By Definition 6 (the definition of Complete), we know that **Next**$(\mathcal{A})$ will always return a minima of a set of annotated answers. Therefore, it remains to show that for every annotated answer A^{full} in **Next**$(\mathcal{A})$, there exists an $A \in \mathcal{A}$ such that $A \prec A^{\mathrm{full}}$.

Observe that in Algorithm 1 **Next**$(\mathcal{A})$ returns the set Complete$(\mathcal{B})$ where $\mathcal{B} = \mathrm{Nextnode}(\mathrm{Back}^b(\mathcal{A}))$ and $b \in \mathbb{N}$ is the minimal b with $\mathrm{Nextnode}(\mathrm{Back}^b(\mathcal{A})) \neq \emptyset$. (The superscript ^{b} denotes that we apply the function b times.)

Let $A^{\mathrm{full}} = \{\!\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\!\} \in$ **Next**$(\mathcal{A})$. Then, we know that there exists an $A = \{\!\!\{(\ell_1, p_1), \ldots, (\ell_k, p_k)\}\!\!\} \in \mathcal{A}$ such that,

$$\{\!\!\{(\ell_1, p_1), \ldots, (\ell_{k-b}, p_{k-b})\}\!\!\} = \{\!\!\{(i_1, q_1), \ldots, (i_{k-b}, q_{k-b})\}\!\!\} \in \mathrm{Back}^b(\mathcal{A}).$$

By definition of Nextnode (see Definition 8), we get that $\ell_{k-b+1} < i_{k-b+1}$. Finally, we know that, for every $j > k - b + 1$, the node i_j is larger or equal to the node i_{k-b+1} by the definition of Complete. Therefore, $A \prec A^{\mathrm{full}}$ which concludes the proof. $\square$

PROOF OF LEMMA 10: ***Enum**(M, w) correctly enumerates all answers in $M(w)$.*

Proof. By definition of annotated answers, we know that there exists an annotated answer for every answer in $M(w)$. Therefore, it remains to show that **Enum**(M, w) computes all annotated answers for M and w and does not output an answer twice.

Let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be the sequence of sets of annotated answers that are given to the output during **Enum**(M, w). It holds that $\mathcal{A}_1 = \text{Complete}(\{\emptyset\})$. By Definition of Complete (see Definition 6), $\mathcal{A}_1$ contains the set of smallest annotated answers to $M(w)$. In the output we will delete duplicate answers of the set $\mathcal{A}_1$. For every set $\mathcal{A}_i$ with $i > 1$, we show that it is the complete set of minimal annotated answers that are larger than an answer in $\mathcal{A}_{i-1}$. Therefore, observe that every set $\mathcal{A}_i$ with $i \in [m]$ is a set Complete$(\mathcal{B})$ for some set of annotated answers $\mathcal{B}$. Thus, we know that, for all $A, B \in \mathcal{A}_i$, we have $\text{Nodes}(A) = \text{Nodes}(B)$. It follows that all answers in set $\mathcal{A}_i$ are strictly larger than all answers in $\mathcal{A}_{i-1}$. Since the output will delete duplicate answers of the set $\mathcal{A}_i$ we will never output an answer twice. Finally, by Lemma 23, we know that the answers in $\mathcal{A}_i$ are minimal and larger which means there is no annotated answer $C \in M(w)$ such that $C \notin \mathcal{A}_i$ for all $i \in [m]$. This concludes the proof. □

Proofs of Section 3.4

PROOF OF LEMMA 13: *For every node $v_{xy} \in \mathrm{N}_w^{aux}$, 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 $I = \text{Nodes}(A^{full}_{[x,y]})$.*

Proof. The proof is by induction on increasing values of the depth, $\mathrm{d}(v_{xy})$, of v_{xy} in the auxiliary tree $\mathrm{N}_w^{\text{aux}}$. Here, the depth $\mathrm{d}(v_{xy})$ of a node in $\mathrm{N}_w^{\text{aux}}$ is the length of the path from the root to v_{xy}.

For the base case, that is, the root $v_{1n} \in \mathrm{N}_w^{\text{aux}}$, we have that

$$R(v_{1n}) = \{(q_0, q_F, \text{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}.$$

The assumption holds by correctness of the relation T^+ (see Lemma 30).

For the induction case, we have to show that the assumption holds for a left-child-node and for a right-child-node v_{xy}. However, both cases are analogous and we only show the case where v_{xy} is a left child in the following. Let v_{xy} be a left child of a node v_{xz} and let $v_{(y+1)z}$ be the right child of v_{xz}. Remember that, by Definition 12,

$$\begin{aligned} R(v_{xy}) = \cup_{(q_1,q_3,I)\in R(v_{xz})}\{(q_1, q_2, J_1) \mid \exists (q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy}) \text{ such that } J_1 \subseteq I_1, \\ \exists (q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z}) \text{ such that } J_2 \subseteq I_2, \text{ and } J_1 \cup J_2 = I\}. \end{aligned}$$

We show the assumption by proving "⇒" and "⇐" separately.

"⇒": Let (q_1, q_2, J_1) be a tuple in $R(v_{xy})$. Then there are, according to Definition 12, tuples $(q_1, q_3, I) \in R(v_{xz})$, $(q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy})$ and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z})$ such that $J_1 \subseteq I_1$, and there exists a set $J_2 \subseteq I_2$ where $J_1 \cup J_2 = I$. By correctness of T^+, we know that there exists

- a partial run $r_1 = q_1 \cdots q_2$ on $w[x..y]$ such that $J_1 \subseteq \text{set}(r_1)$, and
- a partial run $r_2 = q_2 \cdots q_3$ on $w[y + 1..z]$ such that $J_2 \subseteq \text{set}(r_2)$.

By applying the induction hypothesis on the parent node v_{xz}, we know that for the tuple $(q_1, q_3, I) \in R(v_{xz})$ there is an annotated answer A w.r.t. some run r such that $r(x) \in \delta(q_1, w[x])$, $r(z) = q_3$, and $I = \text{Nodes}(A_{[x+1,z]})$. Therefore, it holds that $r = r_\ell r_m r_r$,

- r_ℓ is a partial run $q_0 \cdots q_1$ on $w[1..x - 1]$, and
- r_r is a partial run $q_3 \cdots q_F$ on $w[z + 1..n]$.

Observe that $\text{set}(r_\ell) \cup \text{set}(r_r) \cup I = \text{set}(s)$ for some $s \in S$. Altogether, the run $r_\ell r_1 r_2 r_r$ is an accepting run that produces an annotated answer A^{full} which proves the assumption for v_{xy}.

"⇐": Towards contradiction, assume that (q_1, q_2, J_1) is a tuple not in $R(v_{xy})$ 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 $J_1 = \text{Nodes}(A^{\text{full}}_{[x+1,y]})$. Applying the induction hypothesis on the parent node v_{xz}, we know that there exists a tuple $(q_1, q_3, I) \in R(v_{xz})$ which is compatible with A^{full} and r. But then it directly follows, by correctness of T^+ (see Lemma 30), that there are tuples $(q_1, q_2, I_1) \in \mathrm{T}^+(v_{xy})$ such that $J_1 \subseteq I_1$, and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_{(y+1)z})$ such that, there is a set $J_2 \subseteq I_2$ with $J_1 \cup J_2 = I$. This directly contradicts the assumption that $(q_1, q_2, J_1) \notin R(v_{xy})$. □

PROOF OF 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 $v \in \mathrm{N}_w^{aux}$ with parent v_p, compute $R(v)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$ if $R(v_p)$ is known.*

Proof. For the root node v_{1n}, we have that $R(v_{1n}) = \{(q_0, q_F, \text{set}(s)) \in \mathrm{T}^+(v_{1n}) \mid q_F \in F, s \in S\}$. Thus, we only need to traverse the relation $\mathrm{T}^+(v_{1n})$ which is of size $\mathcal{O}(|Q|^2 \cdot 2^k)$. For all other nodes, we have to distinguish whether they are a left or a right child of its parent v_p. Let v_1 be the left and v_2 be the right child of v_p. To calculate $R(v_1)$ and $R(v_2)$ we traverse the relation $R(v_p)$ and check for a tuple $(q_1, q_3, I) \in R(v_p)$ whether there exist tuples $(q_1, q_2, I_1) \in \mathrm{T}^+(v_1)$ and $(q_2, q_3, I_2) \in \mathrm{T}^+(v_2)$ such that there are subsets $J_1 \subseteq I_1$ and $J_2 \subseteq I_2$ with $J_1 \cup J_2 = I$. If J_1 and J_2 exist, we add to $R(v_1)$ a tuple (q_1, q_2, J_1) for every such set J_1, and a tuple (q_2, q_3, J_2) for every such set J_2 to $R(v_2)$. Because relations R and T^+ contain at most $\mathcal{O}(|Q|^2 \cdot 2^k)$ different tuples for every node, it needs time $\mathcal{O}(|Q|^3 \cdot 2^k)$ to calculate $R(v_1)$ and $R(v_2)$. □

Proofs of Section 3.4.1

PROOF OF 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.*

Proof. Assume that u is the node in $\mathrm{N}_w^{\text{aux}}$ that is given by the above lemma. Towards contradiction, assume that $i_1 \neq u$, i.e., there exists a leaf node $v < u$ such that there is an annotated answer A^{full} with $v \in \text{Nodes}(A^{\text{full}})$. By Lemma 13, there exists a tuple $(p, q, J) \in R(v)$ with $J \neq \emptyset$. This directly contradicts the assumption. $\square$

Lemma 31. *The node i_1 can be computed within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$.*

Proof. The node i_1 can be computed by one top-down pass of the path from the root of $\mathrm{N}_w^{\text{aux}}$ to i_1. Therefore, we start at the root and, whenever we are in a node v, we compute the sets of relevant tuples of its two children v_1 and v_2 (if these exist; if not, we are done) and proceed to the leftmost child for which the set of relevant tuples contains a tuple (p, q, I) with $I \neq \emptyset$. We are done when we reach a leaf. At every node on this path we have to calculate two sets of relevant tuples. By Lemma 14, this can be done in time $\mathcal{O}(|Q|^3 \cdot 2^k)$ for each node. Since a path from the root to a leaf in $\mathrm{N}_w^{\text{aux}}$ is of length $\log n$ we need time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ in total. This concludes the proof. $\square$

Proofs of Section 3.4.2

PROOF OF LEMMA 18: *Let $\mathcal{A}^{first}$ be the set of smallest annotated answers. Then $G(i_k) = \{(q, A) \mid A \in \mathcal{A}^{first}$ and (q, A) is compatible with $A\}$.*

Proof. We prove the lemma by showing the following claim.

Claim 32. *Let $\mathcal{A}^{first}$ be the set of smallest annotated answers. Let $\mathcal{A}^{cand}$ be the set of annotated answers A^c for which there exists an $A^f \in A^{first}$ with*

$$\textit{Nodes}(A^c_{[1,i_j-1]}) = \textit{Nodes}(A^f_{[1,i_j-1]}) \textit{ and } \textit{Nodes}(A^c_{[i_j,i_j]}) \subseteq \textit{Nodes}(A^f_{[i_j,i_j]}).$$

Then $G(i_j) = \{(q, A) \mid |A| = j$ and $\exists A^c \in \mathcal{A}^{cand}$ such that (q, A) is compatible with $A^c\}$.

The statement of Lemma 18 holds directly by Claim 32 for $j = k$. It remains to prove Claim 32.

Proof of Claim 32: The proof is by induction on increasing values j.
For $j = 1$, we have that

$$G(i_1) := \{(q, \{\!\!\{(i_1, q)\}\!\!\}) \mid (p, q, \{q\}) \in R(i_1)\}.$$

Obviously, it holds that $|\{\!\!\{(i_1, q)\}\!\!\}| = 1$. By Definition of R and Lemma 13, there is an annotated answer A^{full} w.r.t. some run r such that $A^{\text{full}}_{[i_1,i_1]} = \{\!\!\{(i_1, q), \ldots, (i_1, q)\}\!\!\}$. By Lemma 15, we know that i_1 is the smallest position that can be assigned in any annotated answer. Therefore, it holds that $A^{\text{full}}_{[1,i_1]} = A^{\text{full}}_{[i_1,i_1]}$ which means that $(q, \{\!\!\{(i_1, q)\}\!\!\})$ is compatible with A^{full}. It remains to show that $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$. Therefore, let A^f be an annotated answer in $\mathcal{A}^{\text{first}}$. By correctness of i_1 (see Lemma 15), it holds that

$$\text{Nodes}(A^{\text{full}}_{[1,i_j-1]}) = \emptyset = \text{Nodes}(A^f_{[1,i_j-1]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) \subseteq \text{Nodes}(A^f_{[i_j,i_j]}).$$

Thus, $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$ which proves the assumption for i_1.

Assume that the statement holds for j. Then, we have that

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

Since the statement holds for j, we have that $|A| = j$. Therefore, $|A \cup \{\!\!\{(i_{j+1}, q)\}\!\!\}| = j + 1$. By the definition of compatibility (see Definition 16), there is an annotated answer A^{full} w.r.t some run r such that $(p, q, \{q\})$ and (q', A) are compatible with A^{full} and r. It directly follows that, $(q, A \cup \{\!\!\{(i_{j+1}, q)\}\!\!\})$ is compatible with A^{full} and r. Therefore, it remains to show that $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$. By applying the induction hypothesis on $(q', A) \in G(i_j)$, we know that there exists annotated answer $A^f \in \mathcal{A}^{\text{first}}$ with

$$\text{Nodes}(A^{\text{full}}_{[1,i_j-1]}) = \text{Nodes}(A^f_{[1,i_j-1]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) \subseteq \text{Nodes}(A^f_{[i_j,i_j]}).$$

By correctness of i_{j+1} (see Lemma 23), it follows that A^{full} and A^f have the same node i_{j+1}, i.e., it follows that also

$$\text{Nodes}(A^{\text{full}}_{[i_j,i_j]}) = \text{Nodes}(A^f_{[i_j,i_j]}) \text{ and } \text{Nodes}(A^{\text{full}}_{[i_{j+1},i_{j+1}]}) \subseteq \text{Nodes}(A^f_{[i_{j+1},i_{j+1}]}).$$

Altogether, it holds that $A^{\text{full}} \in \mathcal{A}^{\text{cand}}$ which proves the assumption. $\square$

PROOF OF 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 $R_{j-1}(v_{xy})$.

Proof. We prove cases (1) to (3) separately. However, notice that, by Definition 19, we have to show that $R_j(v_{xy})$ for each case (1) to (3) is the set of all tuples (q_1, q_2, I) such that

- $(q_1, q_2, I) \in R(v_{xy})$, and
- (q_1, q_2, I) is compatible with some $(q, A) \in G(i_j)$.

(1) We define $R_j(i_j) = \{(p, q, \{q\}) \in R_{j-1}(i_j) \mid \exists (q, A) \in G(i_j)\}$ which needs time $\mathcal{O}(|Q|^3 \cdot 2^k)$. Every tuple $(p, q, \{q\})$ belongs to $R_j(i_j)$ because it is compatible with $(q, A) \in G(i_j)$ and, therefore, also relevant. Since every tuple $(p, q, \{q\}) \in R(i_j)$ that is compatible with some $(q, A) \in G(i_j)$ is as well compatible with some $(q', A') \in G(i_{j-1})$, it follows that $(p, q, \{q\}) \in R_{j-1}(i_j)$. That is, the set $R_j(i_j)$ is also complete.

(2) To prove (2), we distinguish two subcases (a) and (b) whether v_{xy} is a left or a right child.

(a) If v_{xy} is a left child and v is the parent of v_{xy}, then let v' be the right child of v. We define

$$R_j(v_{xy}) = \{(q_1, p, J) \mid \exists (q_1, p, I_1) \in \mathrm{T}^+(v_{xy}), \exists (p, q_2, I_2) \in T^+(v'), \exists (q_1, q_2, I) \in R_j(v), \text{ such that } J \subseteq I_1 \text{ and } J \cup I_2 = I\}.$$

(b) If v_{xy} is a right child and v is the parent of v_{xy}, then let v' be the left child of v. We define

$$R_j(v_{xy}) = \{(p, q_2, J) \mid \exists (p, q_2, I_2) \in \mathrm{T}^+(v_{xy}), \exists (q_1, p, I_1) \in T^+(v'), \exists (q_1, q_2, I) \in R_j(v) \text{ such that } J \subseteq I_2 \text{ and } J \cup I_1 = I\}.$$

We prove that the above definition for $R_j(v_{xy})$ is correct for the case (2)(a). The proof for the case (2)(b) is analogous.
We first prove that every tuple (q_1, p, J) belongs to $R_j(v_{xy})$. Since $(q_1, q_2, I) \in R_j(v)$, we know that (q_1, q_2, I) is compatible with some $(q, A) \in G(i_j)$. Because $x > i_j$, it holds that $\mathrm{Nodes}(A_{[x,y]}) = \emptyset$. Thus, (q_1, p, J) is also compatible with (q, A) and, therefore, also relevant. This proves that (q_1, p, J) belongs to $R_j(v_{xy})$. Towards contradiction, assume that the above definition is not complete for $R_j(v_{xy})$. Then, there is a tuple $(q_1, p, J) \in R(v_{xy})$ which is compatible with some $(q, A) \in G(i_j)$ but which is not captured by the right side in the above definition. However, we know that $(q_1, p, J) \in R(v_{xy})$. By Definition 12, it directly follows that there is a tuple $(q_1, p, I_1) \in \mathrm{T}^+(v_{xy})$ with $J \subseteq I_1$, and there are tuples $(p, q_2, I_2) \in T^+(v')$ and $(q_1, q_2, I) \in R_j(v)$ such that $J \cup I_2 = I$. Therefore, $(q_1, p, J) \in R_j(v_{xy})$ which directly contradicts the assumption. We can compute $R_j(v_{xy})$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$.

(3) To prove (3), we distinguish two subcases (a) and (b) whether v_{xy} has a left or a right child.

(a) If v_{xy} has a left child v, then let v' be the right child of v_{xy}. We define

$$R_j(v_{xy}) = \{(q_1, q_2, I) \mid \exists (q_1, q_2, I) \in R_{j-1}(v_{xy}), \exists (q_1, p, I_1) \in R_j(v), \exists (p, q_2, I_2) \in T^+(v') \text{ such that } I_1 \cup I_2 = I\}.$$

(b) If v_{xy} has a right child v, then let v' be the left child of v_{xy}. We define

$$R_j(v_{xy}) = \{(q_1, q_2, I) \mid \exists (q_1, q_2, I) \in R_{j-1}(v_{xy}), \exists (q_1, p, I_1) \in T^+(v'), \exists (p, q_2, I_2) \in R_j(v) \text{ such that } I_1 \cup I_2 = I\}.$$

We prove that the above definition for $R_j(v_{xy})$ is correct for the case (3)(a). The proof for the case (3)(b) is analogous.
We first prove that every tuple (q_1, q_2, I) belongs to $R_j(v_{xy})$. Because $(q_1, q_2, I) \in R_{j-1}(v_{xy})$, it directly holds that $(q_1, q_2, I) \in R(v_{xy})$. It remains to show that there is an annotated answer A^{full} w.r.t. some run r_A such that (q_1, q_2, I) and some $(q, A) \in G(i_j)$ are compatible with A^{full} and r_A. In the following, we construct such annotated answer A^{full} and run r_A from the given information about (q_1, q_2, I). First, because $(q_1, q_2, I) \in R_{j-1}(v_{xy})$ we know that (q_1, q_2, I) is compatible with some $(p, B) \in G(i_{j-1})$. That is, there is an annotated answer B^{full} w.r.t some run r_B such that (q_1, q_2, I) and (p, B) are compatible with B^{full} and r_B. Observe it already holds that $\mathrm{Nodes}(B) = \mathrm{Nodes}(A_{[1,i_{j-1}]})$. Furthermore, by definition of compatibility, we know that $\mathrm{Nodes}(B^{\text{full}}_{[x,y]}) = I$ and $r_B = r_1 r_2 r_3$ such that r_2 is a partial run $q_1 \cdots q_2$ on $w[x..y]$ and $I \subseteq \mathrm{set}(r_2)$. However, this is not enough to show that B^{full} is compatible with $(q, A) \in G(i_j)$ because B^{full} does not have to agree with A on the node i_j. If this is the case, then we can construct a new run r_A from r_B such that r_A produces the desired A^{full}. Therefore, we define $r_A = r_1 \cdot r_v \cdot r_{v'} \cdot r_3$ where

- $r_v = q_1 \cdots p$ is a partial run on $w[x..z]$ such that $I_1 \subseteq \mathrm{set}(r_v)$ and $r[i_j] = q$ for $(i_j, q) \in A$, and
- $r_{v'} = p' \cdots q_2$ with $p' \in \delta(p, w[z+1])$ is a partial run on $w[z+2..y]$ where $I_2 \subseteq \mathrm{set}(r_{v'})$.

Using Definition 19 and that $(q_1, p, I_1) \in R_j(v)$, it follows that r_v is well-defined. Applying Lemma 30 for the tuple $(p, q_2, I_2) \in \mathrm{T}^+(v')$, it follows that $r_{v'}$ is well-defined. Because $I_1 \cup I_2 = I$, the run r_A produces the annotated answer A^{full} which proves the assumption. Towards contradiction, assume that the above definition is not complete for $R_j(v_{xy})$. Then, there is a tuple $(q_1, q_2, I) \in R(v_{xy})$ which is compatible with some $(q, A) \in G(i_j)$ but which is not captured by the right side in the above definition. Let the annotated answer A^{full} w.r.t. some run r be such, that (q_1, q_2, I) and (q, A) are compatible with A^{full} and r. By definition of compatibility, we have that $\mathrm{Nodes}(A^{\text{full}}_{xy}) = I$. Then there exist I_1 and I_2 such that, for $z = x - 1 + \lfloor \frac{y-x+1}{2} \rfloor$, it holds that $\mathrm{Nodes}(A^{\text{full}}_{xz}) = I_1$, $\mathrm{Nodes}(A^{\text{full}}_{(z+1)y}) = I_2$ and $I = I_1 \cup I_2$. It follows that there exist a tuple $(q_1, p, I_1) \in R_j(v_{xz})$ for the left child v_{xz} of v, and a tuple $(p, q_2, I_2) \in \mathrm{T}^+(v_{(z+1)y})$ for the right child $v_{(z+1)y}$ of v. Therefore, $(q_1, q_2, I) \in R_j(v_{xy})$ which directly contradicts the assumption. We can compute $R_j(v_{xy})$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$.

□

Proposition 33. *All sets $G(i_j)$ (for $j \in [k]$) can be stored together in one data structure of size $\mathcal{O}(|S| \cdot k!)$. Furthermore, this data structure can, given j, generate $G(i_j)$ within time $\mathcal{O}(|S| \cdot k!)$.*

Proof. We first show that $\mathcal{O}(|S| \cdot k!)$ records suffice for storing a single $G(i_j)$ and then show how we can store them all together.

For a given j, we defined $G(i_j)$ as a set of tuples (q, A) where A is a multiset over $\text{Nodes}(w) \times Q$. Furthermore, by construction of $G(i_j)$, every annotated answer A for which (q, A) in $G(i_j)$ uses the same multiset of j nodes $\{\!\{i_1, \ldots, i_j\}\!\}$. Since k is an upper bound for j, a naïve upper bound for the size of $G(i_j)$ would therefore be $\mathcal{O}(k + |Q| \times |Q|^k)$. (That is, size at most k for storing the tuple $(i_1, \ldots, i_j)$ and at most $|Q| \times |Q|^k$ for all possibilities of $(q, (q_1, \ldots, q_j))$ for which $(q, \{\!\{(i_1, q_1), \ldots, (i_j, q_j)\}\!\})$ is in $G(i_j)$.

However, for every $(q, A) \in G(i_j)$, it holds that all tuples of the form $(i_j, q_j) \in A$ have $q_j = q$, by construction of $G(i_j)$. Therefore, instead of storing all possibilities $(q, (q_1, \ldots, q_j))$ it suffices to store $(q_1, \ldots, q_j)$.

But we can optimize even more. Observe that each element in $G(i_j)$ does not contain arbitrary states but rather states in some set $\text{set}(s)$ for some $s \in S$. Therefore, since we already have $(i_1, \ldots, i_j)$, we could alternatively store, for every tuple $(q_1, \ldots, q_k) \in S$, all injections $\sigma : [j] \to [k]$ such that $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\} \in G(i_j)$. Hence, for $j = k$ we store the tuple $(i_1, \ldots, i_k)$ once and, for every $(q_1, \ldots, q_k) \in S$, the set of permutations $\sigma : [k] \to [k]$ such that $(q_{\sigma(k)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_k, q_{\sigma(k)})\}\!\}) \in G(i_k)$. The total size is $\mathcal{O}(k + k! \cdot |S|)$, which is in $\mathcal{O}(k! \cdot |S|)$. (The representation size for $j < k$ is smaller.)

In fact, we can even store all $G(i_j)$ together in a single data structure of size $\mathcal{O}(k! \cdot |S|)$. To see this, consider a $(q_1, \ldots, q_k)$ and all injections mentioned above. If $\sigma : [j] \to [k]$ is an injection such that $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\} \in G(i_j)$ then there are two options for $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\}$. Either we can extend it in $G(i_{j+1})$ or not. In the former case, there is also a non-empty set of injections of the form $\sigma' : [j+1] \to [k]$ such that

- σ' extend σ, that is, $\sigma'(\ell) = \sigma(\ell)$ for every $\ell \in [j]$ and
- $(q_{\sigma'(j+1)}, \{\!\{(i_1, q_{\sigma'(1)}), \ldots, (i_{j+1}, q_{\sigma'(j+1)})\}\!\} \in G(i_{j+1})$.

In the latter case, there exists no such non-empty set of injections extending σ.

Notice that, in the former case, σ can be immediately inferred from σ', so σ does not need to be stored separately. For every j, we therefore only store the $\sigma : [j] \to [k]$ that cannot be extended to some $\sigma' : [j] \to [k+1]$ for $G(i_{j+1})$. In particular, for $j = k$, we store all permutations that encode answers.

We claim that the total number of injections we store is $\mathcal{O}(k! \cdot |S|)$. This is now easy to see: for every tuple $s \in S$, each injection σ that we store cannot be extended to a permutation with the correct properties. That is, there exist permutations that extend σ, but these do not have the property that they produce the answer $(i_1, \ldots, i_k)$. Therefore, the total number of injections that we store is $O(k!)$. The size of a single such injection is not larger than the size of a tuple in S, which means that $O(|S| \cdot k!)$ records suffice in total.

Finally, producing $G(i_j)$ for a given j can be done by returning all $(q_{\sigma(j)}, \{\!\{(i_1, q_{\sigma(1)}), \ldots, (i_j, q_{\sigma(j)})\}\!\}$ for which we stored a σ that is defined on j. □

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

Proof. In the following, we show first how much time we need to compute the node i_ℓ. Afterwards, we will examine the time that was spend to calculate the set $G(i_\ell)$ during the computation.

For $\ell = 1$, we know that i_1 can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ by Lemma 31. Next, we show how to calculate $i_{\ell+1}$ in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ when i_ℓ is given. Therefore, we use that the following information is available. Besides the auxiliary structure $\mathrm{N}_w^{\text{aux}}$ with T^+ and all nodes in the multiset $N = \{\!\{i_1, \ldots, i_\ell\}\!\}$, we assume that we know

(a) $G(i_\ell)$ and $R(u_{i_\ell})$,
(b) the tree $\mathrm{A}_w^{\text{aux}}$ induced by all ancestors of nodes in N, and,
(c) for every node $v_{xy} \in \mathrm{A}_w^{\text{aux}}$, we know the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$, where $i_j \in N$ is the highest position with $x \le i_j \le y$.

After the computation of $G(i_1)$, $\mathrm{A}_w^{\text{aux}}$ consists only of the path from the root of $\mathrm{N}_w^{\text{aux}}$ to i_1. Since, $R_0(v_{xy}) = R(v_{xy})$, we calculate $R_0(v_{xy})$ for every node on this path during the computation of i_1. Therefore, information (a) to (c) is available after we have computed $G(i_1)$.
Now, assume that we have all necessary information up to node i_ℓ. By Proposition 17, we know that

- $i_{\ell+1} \ge i_\ell$ is the leftmost node in w for which
  there exists a tuple $(q_1, q_2, I) \in R(i_{\ell+1})$ with $I \ne \emptyset$ which is compatible with some $(q, A) \in G(i_\ell)$.

By Definition 19, this is equivalent to

- $i_{\ell+1} \ge i_\ell$ is the leftmost node in w for which there exists a tuple $(q_1, q_2, I) \in R_\ell(i_{\ell+1})$ with $I \ne \emptyset$.

In the following, we show how we can compute $i_{\ell+1}$ in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$ providing that information (a) to (c) is available afterwards. Therefore, we argue that we need at most $\log n$ operations of the kind (1) to (3) in Lemma 20 to find $i_{\ell+1}$ from i_ℓ. We start at node i_ℓ where we assume that $G(i_\ell)$ and $R_{\ell-1}(i_\ell)$ are known. We compute $R_\ell(i_\ell)$ applying (1) in Lemma 20 and test whether $i_{\ell+1} = i_\ell$. If this is not the case we follow the path p from i_ℓ to the root of $\mathrm{N}_w^{\text{aux}}$ and calculate R_ℓ on the way. (Notice that, all necessary information (a) to (c) is still available afterwards.) Since we always calculate the new relation R_ℓ for every node on p we can always apply case (3) in Lemma 20. 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_ℓ for every right child (applying case (2) in Lemma 20). Again, all necessary information (a) to (c) is still available afterwards. We stop when we find such a right child v_r which is not on p and where $R_\ell(v_r)$ contains a tuple (q_1, q_2, I) with $I \ne \emptyset$. Again, this can be done with at most $\log n$ operations. By definition of R_ℓ, we know that the subtree rooted at v_r has at least one leaf node u such that $R_\ell(u)$ contains a tuple (q_1, q_2, I) with $I \neq \emptyset$. The leftmost such leaf u will be $i_{\ell+1}$. To arrive at $i_{\ell+1}$, we go down from v_r. On this path, we always compute R_ℓ for both children (applying case (2) in Lemma 20) and choose the leftmost child for which R_ℓ contains 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 $\mathrm{N}_w^{\text{aux}}$ using time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \ell \log n)$ in total. Furthermore, we computed R_ℓ for every node on the path from the root of $\mathrm{N}_w^{\text{aux}}$ to $i_{\ell+1}$ therefore obtaining the necessary information for (b) and (c).

Finally, the following characterization shows how we can obtain $G(i_{\ell+1})$ from $G(i_\ell)$ using j-relevant tuples (see Definition 19):

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

Afterwards, information (a) is available as well. When going from i_ℓ to $i_{\ell+1}$, we can maintain reachable states in δ^* within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$. By Proposition 33, we have shown that the computation of all $G(i_j)$ for $j \in [\ell]$ together need in time $\mathcal{O}(|S| \cdot k!)$. Therefore, we used time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot \log n)$ to compute $G(i_\ell)$ in total. □

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

Proof. Let $(v_1, \ldots, v_k)$ be the first answer of M on w such that there is an accepting run r of N on w and a tuple $(p_1, \ldots, p_k) \in S$ where, for every $\ell \in [k]$, r visits v_ℓ in p_ℓ. By Lemma 18, the set $G(i_k)$ contains the set of smallest annotated answers. Therefore, $G(i_k)$ contains an annotated answers $A^{\text{full}} = \{\!\{(v_1, p_1), \ldots, (v_k, p_k)\}\!\}$. By Lemma 21, $G(i_k)$ can be computed within time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$. □

Proofs of Section 3.5

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

Proof. Recall from Section 3.3 that every set of annotated answers $\mathcal{A}$ at each call of Complete, Back, or Nextnode has the property that all $A \in \mathcal{A}$ use the same multiset of nodes $\{\!\{i_1, \ldots, i_\ell\}\!\}$. We denote this multiset by Nodes($\mathcal{A}$). Furthermore, we assume that the following information is available when we call Complete, Back, Nextnode, or Next($\mathcal{A}$): the tree $\mathrm{N}_w^{\text{aux}}$ with T^+ and the relations R_j and sets G as described in invariants (I1) and (I2) below.

(I1) Let $\mathrm{A}_w^{\text{aux}}$ be the tree induced by all ancestors in $\mathrm{N}_w^{\text{aux}}$ of nodes in Nodes($\mathcal{A}$). For every $v_{xy} \in \mathrm{A}_w^{\text{aux}}$, we know the relation $R_{j-1}(v_{xy})$ or $R_j(v_{xy})$ where $i_j \in$ Nodes($\mathcal{A}$) is the maximal node with $x \leq i_j \leq y$.

(I2) For every $i_j \in$ Nodes($\mathcal{A}$), we know $G(i_j)$. Here, $G(i_j) = \{(q, A) \mid$ Nodes(A) $\subseteq$ Nodes($\mathcal{A}$), $|A| = j$ and there is an annotated answer A^{full} s.t. Nodes($A_{[1,i_j-1]}$) $=$ Nodes($A^{\text{full}}_{[1,i_j-1]}$), Nodes($A_{[i_j,i_j]}$) $\subseteq$ Nodes($A^{\text{full}}_{[i_j,i_j]}$), and (q, A) is compatible with $A^{\text{full}}\}$.

In the following we show that we can implement the aforementioned procedure correctly and such that (I1) and (I2) hold afterwards.

Complete($\mathcal{A}$): Remember that, in Algorithm 1, we always call Complete($\{\emptyset\}$) in the beginning. In Section 3.4, we proved that the call will return the smallest set of annotated answers in $M(w)$ if they exist; otherwise $M(w) \neq \emptyset$ and we are done. Furthermore, we know that (I1) holds by the computation done in the proof of Lemma 21 and that (I2) holds by Claim 32 (which we proved for Lemma 18). We now generalize the description in Section 3.4 to compute Complete($\mathcal{A}$) for an arbitrary occurrence of $\mathcal{A}$ in Algorithm 1. To this end, we have to change the definition of tuple $(i_1, \ldots, i_k)$ in Section 3.4. In particular, $(i_1, \ldots, i_k)$ should be the smallest answer of the query such that $\{\!\{i_1, \ldots, i_j\}\!\} =$ Nodes($\mathcal{A}$) and, for every $\ell > j$, i_ℓ is at least the largest number i_j in Nodes($\mathcal{A}$). That is, we only recompute $G(i_\ell)$ for $\ell > j$ and leave all $G(i_1), \ldots, G(i_j)$ untouched. Because we have the relation R_j available as it is given in (I1), we can compute i_{j+1} and the new set $G(i_{j+1})$ analogous to the case for Complete($\{\emptyset\}$) applying Lemma 20. Again, this computation satisfies that (I1) holds for the set Nodes(Complete($\mathcal{A}$)). At the newly computed node i_{j+1}, we compute the set $G(i_{j+1})$ equally to the case for Complete($\{\emptyset\}$), i.e.,

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

However, notice that by (I2) the semantics of the sets $G(i_j)$ differ from Section 3.4. At the end, we define

$$\text{Complete}(\mathcal{A}) = \{A \mid (q, A) \in G(i_k)\}.$$

Analogously to Lemma 21, the computation of Complete($\mathcal{A}$) needs time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$. It remains to show that (I2) holds after the computation. Observe that, when Complete($\mathcal{A}$) is finished, we have that Nodes(Complete($\mathcal{A}$)) $= \{\!\{i_1, \ldots, i_j, i_{j+1}, \ldots, i_k\}\!\}$ where $i_{j+1}, \ldots, i_k$ were recomputed by the procedure. Then, for all sets $G(i_\ell)$ with $\ell \leq j$, (I2) holds by assumption. Next, we show that (I2) holds for all all sets $G(i_\ell)$ where $\ell > j$. Therefore we prove that, if $G(i_j)$ fulfills (I2) then $G(i_{j+1})$ given by the above computation fulfills (I2) as well. First, it holds that Nodes(A) $\subseteq$ Nodes($\mathcal{A}$) by definition. By applying (I2) for $G(i_j)$, we know that $|A| = j$, i.e., $|A \cup \{\!\{(i_{j+1}, q_2)\}\!\}| = j + 1$. By the same argument, we also know that there exists an annotated answer A_1^{full} w.r.t. some run r_1 such that $(q, A) \in G(i_j)$ is compatible with A_1^{full} and r_2. Then, by the definition of R, we know that there is another annotated answer A_2^{full} w.r.t. some run r_2 such that $(q_1, q_2, \{q_2\}) \in R_j(i_{j+1})$ is compatible with A_2^{full} and r_2. Since we know that q_1 is reachable from q according to the right subword of w, it is straightforward to obtain an annotated answer A^{full} w.r.t. some run r such that $(q_1, q_2, \{q_2\})$ and (q, A) are compatible with A^{full} and r. It directly follows that (I2) holds for $G(i_{j+1})$ which concludes the description of Complete($\mathcal{A}$).

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

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

Because (I1) and (I2) hold before the call of Back($\mathcal{A}$) they are satisfied afterwards since we do not touch any R and G. Since $G(i_{j-1})$ is already computed, each call Back($\mathcal{A}$) needs time linear in the size of $G(i_{j-1})$, i.e., $\mathcal{O}(|S| \cdot k!)$ by Proposition 33. It remains to prove that our definition is correct. Therefore, observe that all $A \in$ Back($\mathcal{A}$) are incomplete answers of size $j-1$ when applying (I2) on $G(i_{j-1})$. Furthermore and by the same argument, these A are all possible incomplete answers A of M on w where Nodes$(A) = \{\!\{i_1, \dots, i_{j-1}\}\!\}$. This concludes the description of Back($\mathcal{A}$).

Nextnode($\mathcal{A}$): Finally, to compute Nextnode($\mathcal{A}$) let i_j be a maximal node in Nodes($\mathcal{A}$). Then, Nextnode($\mathcal{A}$) has to do two steps in particular: (1) check whether there is an annotated answer $\mathcal{A}^{\text{full}}$ with Nodes$(\mathcal{A}^{\text{full}}) = \{\!\{i_1, \dots, i_{j-1}, i_{j+1}, \dots i_k\}\!\}$ for new nodes $i_{j+1}, \dots, i_k$ strictly larger than i_j, and (2) compute the set $G(i_{j+1})$ for a single new position strictly larger than i_j. We define

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

In Section 3.4.2, we have already showed how to implement (1) and (2) to find a new node i_{j+1} for the case Complete($\mathcal{A}$). In the case of Nextnode($\mathcal{A}$), the proof is analogous with only one difference: one has to ensure that i_{j+1} is strictly larger than i_j; if it exists. But this can be done by skipping the step where we check whether $i_{j+1} = i_j$ in the beginning of the computation. The definition of $G(i_{j+1})$ is equal to the aforementioned case of Complete($\mathcal{A}$). Analogously to the case of Complete($\mathcal{A}$) the computation satisfies (I1) and (I2) afterwards. Altogether, the call of Nextnode($\mathcal{A}$) needs time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot \log n)$. (Remember that, it recomputes at most one node i_j and one set $G(i_j)$.)

Next($\mathcal{A}$): In Lemma 9, we have already proved that Next($\mathcal{A}$) is correct when Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$) can be implemented correctly. Therefore, it directly follows that Next($\mathcal{A}$) can be implemented correctly as given in Algorithm 1. Afore, we proved that Nextnode needs time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot \log n)$, Back can be implemented in time $\mathcal{O}(|S| \cdot k!)$, and Complete runs in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$. During a call of Next($\mathcal{A}$) there are at most $\mathcal{O}(k)$ calls of Nextnode and Back and there is only one call of Complete. Altogether, Next($\mathcal{A}$) needs time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$ in total. □

PROOF OF 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.*

Proof. In the preprocessing phase, we build the tree $\mathrm{N}_w^{\text{aux}}$ and relation T^+ which serve as our auxiliary data. By Lemma 5, the data has size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot n)$ and can be computed within time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot n)$. By the same lemma, it follows that updates can be maintained in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log n)$. By Lemma 10, we know that we can use Algorithm 1 to correctly enumerate all answers in $M(w)$ if Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$) can be computed correctly within the required running time. In Theorem 22, we showed that Complete($\{\emptyset\}$) can be implemented correctly such that it runs in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$. Finally we showed, in Lemma 23, that Next($\mathcal{A}$) (and thereby Complete($\mathcal{A}$), Back($\mathcal{A}$), and Nextnode($\mathcal{A}$)) can be computed correctly within time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log n)$. □

Proofs of Section 4 (Trees)

Proofs of Section 4.1

PROOF OF 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 $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$.*

Proof. Observe that, all trees $\mathrm{N}_p^{\mathrm{aux}}$ for every $p \in \mathrm{HPaths}(t)$ together have $|t|$ leaf nodes with at most $\log |t|$ depth, i.e., they have $\mathcal{O}(|t|)$ nodes altogether. For every node v in some $\mathrm{N}_p^{\mathrm{aux}}$, the transition relation T_p^+ is of size $\mathcal{O}(|Q|^2 \cdot 2^k)$. Therefore, our auxiliary structure has size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot |t|)$ in total. The transition relation T_p^+ is build bottom-up for every tree $\mathrm{N}_p^{\mathrm{aux}}$ analogously to the word case. For leaf nodes, T_p^+ can be calculated in time and space $\mathcal{O}(|Q|^2)$. Afterwards, we need to compute $|t|$ $\bowtie^+$-operations where each join can be done in time $\mathcal{O}(|Q|^3 \cdot 2^k)$. Altogether, the auxiliary structure can be build in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot |t|)$.

Now, consider an update at node $v \in t$ and let v be a node of path $p \in \mathrm{HPaths}(t)$. Then, we update every relation on the path from v to the root of $\mathrm{N}_p^{\mathrm{aux}}$, i.e., we update $\log |t|$ many relations. This can be done by $\log |t|$ $\bowtie^+$-operations yielding time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log |t|)$ for one tree $\mathrm{N}_p^{\mathrm{aux}}$. Analogously, to the proof of Lemma 30, it follows that the update is processed correctly for the relation T_p^+. Afterwards, this update of the tree $\mathrm{N}_p^{\mathrm{aux}}$ triggers an update in every tree $\mathrm{N}_{p'}^{\mathrm{aux}}$ for every path $p' \in \mathrm{HPaths}(t)$ that is crossed by the path from v to the root in t. By Lemma 26, there exist at most $\log |t|$ such paths p'. Therefore, we have to update at most $\log |t|$ trees $\mathrm{N}_{p'}^{\mathrm{aux}}$ each with costs $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log |t|)$. Thus, we need time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$ to update all trees $\mathrm{N}_p^{\mathrm{aux}}$ on the path from v to the root of t which means an update costs $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$ in total. □

Details for Section 4.2

The algorithms for INCENUM on words are similar to INCENUM on trees but we must update a few definitions. Here we discuss how the techniques of enumeration on words (Section 3) should be changed so that we can use them for enumeration on trees. We provide further details for which there was no space in the body of the paper.

In the following, we denote by $M = ((Q, \Sigma, \delta, F), S)$ a k-NSTA and by t a binary input tree with $|t| = n$.

The Auxiliary Structure and the Order of Nodes

First, we formally define our auxiliary structure for trees. Let $\mathrm{HPaths}(t)$ be the set of maximal heavy paths for the tree t. For every $p \in \mathrm{HPaths}(t)$, let $\mathrm{N}_p^{\mathrm{aux}}$ be the auxiliary tree for every NFA N_p on p. We now define a new auxiliary tree $\mathrm{N}_t^{\mathrm{aux}}$ which is built from all $\mathrm{N}_p^{\mathrm{aux}}$. Intuitively, $\mathrm{N}_t^{\mathrm{aux}}$ refers to the tree depicted on the right side of Figure 4.

Definition 34. The auxiliary tree $\mathrm{N}_t^{\mathrm{aux}}$ is defined as follows:

- the root of $\mathrm{N}_t^{\mathrm{aux}}$ is the root of $\mathrm{N}_{\mathrm{hp}(t)}^{\mathrm{aux}}$, where $\mathrm{hp}(t)$ is the heavy path that contains t's root;
- for every maximal heavy path p and leaf node $v \in \mathrm{N}_p^{\mathrm{aux}}$, if v has a child in t that is root of another heavy path p', then the child of v in $\mathrm{N}_t^{\mathrm{aux}}$ is the root of the tree $\mathrm{N}_{p'}^{\mathrm{aux}}$; otherwise v is a leaf in $\mathrm{N}_t^{\mathrm{aux}}$.

Therefore, $\mathrm{N}_t^{\mathrm{aux}}$ can be seen as the union of all trees $\mathrm{N}_p^{\mathrm{aux}}$ with extra edges that connect the different $\mathrm{N}_p^{\mathrm{aux}}$. A node in $\mathrm{N}_t^{\mathrm{aux}}$ is always also a node in some $\mathrm{N}_p^{\mathrm{aux}}$ and, if it is a leaf in $\mathrm{N}_p^{\mathrm{aux}}$, then it is also a node in t.

On words, the order $\preceq$ in which we compare nodes is inherently given by the word. On trees we compare single nodes in $\mathrm{N}_t^{\mathrm{aux}}$ by saying that $u \preceq_{\mathrm{t}} v$ if $u = v$ or u comes before v in the (left-to-right) post-order traversal of $\mathrm{N}_t^{\mathrm{aux}}$. Again, we define $\prec_{\mathrm{t}}$ as the strict version of $\preceq_{\mathrm{t}}$ and we extend $\preceq_{\mathrm{t}}$ to tuples analogously as we did for words.

Observe that, since all nodes of t also appear in $\mathrm{N}_t^{\mathrm{aux}}$, $\preceq_{\mathrm{t}}$ (resp. $\prec_{\mathrm{t}}$) is also well-defined for every node $v \in t$.

Enumerating Query Answers for Trees

In the following, we prove that we can use Algorithm 1 interpreted for a k-NSTA M and a tree t (instead of a word w) to correctly enumerate all answers to the query $M(t)$. We adopt the definition of an *(incomplete) annotated answer* (see Section 3.3) for trees. Formally, an *annotated answer* of a k-NSTA $M = (N, S)$ on t is a multiset A^{full} over $\mathrm{Nodes}(t) \times Q$ of the form

$$\{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$$

such that there is an accepting run λ of N on t and a tuple $(q_1, \ldots, q_k) \in S$ such that λ visits i_ℓ in q_ℓ, for every $\ell \in [k]$. For a node $v \in \mathrm{N}_t^{\mathrm{aux}}$, let t_v be the subtree of $\mathrm{N}_t^{\mathrm{aux}}$ rooted at v. For an annotated answer $A = \{\!\{(i_1, q_1), \ldots, (i_k, q_k)\}\!\}$ and a set of nodes $V \subseteq \mathrm{Nodes}(\mathrm{N}_t^{\mathrm{aux}})$, the *projection of A onto $[V]$*, denoted $A_{[V]}$, is defined to be the multiset $\{\!\{(i, q_i) \mid \exists v \in V : \text{ such that } i \in t_v\}\!\}$. All remaining notions for annotated answers are used as well but are defined equally to the word case. Now the proof of the following lemma is analogous to the proof of Lemma 10, when using the tree definitions for annotated answers.

Lemma 35. *Let M be a k-NSTA and t a binary tree. Then,* ***Enum***(M, t) *correctly enumerates all answers in $M(t)$.*

The First Answer for Trees

Following the lines of the word case, we explain next how to compute the first set of answers to the query $M(t)$. Therefore, we show how to implement $\mathrm{Complete}(\{\emptyset\})$ for trees. First, we have to adapt the notion of a *growing annotated answer up to position j* to trees. Again, we take the definition for words (see Definition 11) but interpret it over $\mathrm{Nodes}(t) \times Q$:

Definition 36. Let $q \in Q$ and A be a multiset over $\text{Nodes}(t) \times Q$. Then (q, A) is a *growing annotated answer up to position* j if there is an accepting run λ of N on t and some node $j \in \text{Nodes}(t)$ such that

- λ visits j in q; and
- there is an annotated answer A^{full} w.r.t. λ such that, for every $p \in Q$ and $i \in \text{Nodes}(t)$,
  - $i \prec_{\text{t}} j$, we have $A^{\text{full}}((i,p)) = A((i,p))$,
  - $i = j$, $A((i,p)) \leq A^{\text{full}}((i,p))$, and
  - $i \succ_{\text{t}} j$, we have $A((i,p)) = 0$.

The definition of the *relevant relation* has to be changed a bit more drastically to adapt it to the tree case (see Definition 28). In the following, we prove that Definition 28 is correct for our purpose. That is, we prove a statement similar to the one of Lemma 13 for trees. For a node v_{xy} in a tree $\mathrm{N}_p^{\text{aux}}$ where $p = v_n \cdots v_1$ (v_1 is a leaf), we define the *projected path* $\text{pp}(v_{xy})$ as $v_x \cdots v_y$. Notice that in $\text{pp}(v_{xy})$ we reverse the order on nodes $v_i \in p$ since we always have that $x \leq y$. Therefore, the definition of projected paths fits to the input words for the automata N_p because their read the labels of the path p bottom-up by definition.

Lemma 37. *Let $v_{xy} \in \mathrm{N}_p^{aux}$, $\text{pp}(v_{xy}) = v_x \cdots v_y$ and $V = \{v_x, \ldots, v_y\}$. Then, $R(v_{xy})$ is the set of all tuples (q_1, q_2, I) such that, there is an annotated answer A^{full} w.r.t. some run λ on t with $\lambda(v_y) = q_2$, $I = Nodes(A^{full}_{[V]})$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$.*

Proof. Let $v_{xy} \in \mathrm{N}_p^{\text{aux}}$, $\text{pp}(v_{xy}) = v_x \cdots v_y$ and $V = \{v_x, \ldots, v_y\}$. Balmin et al. [2] showed that one can maintain an NTA on a tree by decomposing the tree into maximal heavy paths and maintaining an NFA for every such path. The NFAs in their approach use almost the same transition relations as we do. More precisely, they use only the first two items in the triples from T_p^+. Since, for every tuple $(q_1, q_2, I) \in R(v_{xy})$ there is a tuple $(q_1, q_2, J) \in \mathrm{T}_p^+$ for some sets I and J, it follows analogously to the result of Balmin et al. that there is a run λ on t such that $\lambda(v_y) = q_2$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$. The proof that the set I is correct is a straightforward induction on the position of a path in $\text{HPaths}(t)$ compared to t. ☐

Furthermore, we adopt the definition of compatibility (see Definition 16) to trees.

Definition 38 (Compatibility)**.** Let $v_{xy} \in \mathrm{N}_t^{\text{aux}}$, $\text{pp}(v_{xy}) = v_x \cdots v_y$ and $V = \{v_x, \ldots, v_y\}$. For an annotated answer A^{full} w.r.t. some run λ, we say that a tuple

- $(q_1, q_2, I) \in R(v_{xy})$ *is compatible with A^{full} and λ* if $\lambda(v_y) = q_2$, $I = \text{Nodes}(A^{\text{full}}_{[V]})$, and, for $q_1 \neq q_0$, $\lambda(v_x) = q_1$.

Furthermore, for (q, A) a growing answer up to position i,

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

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 λ such that both (q_1, q_2, I) and (q, A) are compatible with A^{full} and λ.

Observe that we only changed the notion of compatibility for a tuple in R and an annotated answer accordingly to the semantics of R (see Lemma 37). The rest of the definition is exactly the same as for words. We can now give a Definition of the nodes i_1 (see Lemma 15) and i_{j+1} (see Proposition 17) for trees.

For Lemmas 39, Proposition 40, Definition 41, and Lemma 42, let $(i_1, \ldots, i_k)$ again be the smallest answer of M on t.

Lemma 39. *Let u be the smallest node in t (w.r.t. $\preceq_t$) such that $R(u)$ has a tuple (q_1, q_2, I) with $q_2 \in I$. Then u is the node i_1 in t.*

Next, we define the sets G of growing answers for trees:

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

Proposition 40. *The node $i_{j+1} \succeq_t i_j$ is the smallest node (regarding $\preceq_t$) in t for which there exists a tuple $(q_1, q_2, I) \in R(i_{j+1})$ with $q_2 \in I$ which is compatible with some $(q, A) \in G(i_j)$.*

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

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

The proof of correctness for the tree case is analogous to the proof of Lemma 18, but using the above definitions for trees. For the computation of the positions $\{\!\{i_1, \ldots, i_k\}\!\}$, we do a very similar traversal of the auxiliary structure as for the word case but this time in the tree $\mathrm{N}_t^{\text{aux}}$. During this traversal the notion of *j-relevance* (see Definition 19) was central for the word case. The definition of j-relevance on trees is essentially the same one as for words, but we use the definition of relation R on trees and the tree notion of compatibility:

Definition 41. For $v \in \mathrm{N}_t^{\text{aux}}$ and $j \in \{0, \ldots, k\}$, we define the set of *j-relevant tuples of* v, denoted $R_j(v)$, as follows:

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

As we said before, we traverse the tree $\mathrm{N}_t^{\text{aux}}$ in the same way as the tree $\mathrm{N}_w^{\text{aux}}$ for words. However, the tree $\mathrm{N}_t^{\text{aux}}$ has $\mathcal{O}(\log^2 |t|)$ depth, whereas $\mathrm{N}_w^{\text{aux}}$ had $\mathcal{O}(\log |w|)$ depth. Therefore, we need time $\mathcal{O}(\log^2 |t|)$ to go from a node i_j to i_{j+1}. Furthermore, we do not collect output of the query only at the leaves of $\mathrm{N}_t^{\text{aux}}$ because nodes of t appear as internal nodes of $\mathrm{N}_t^{\text{aux}}$ (but as leaf nodes of subtrees $\mathrm{N}_p^{\text{aux}}$). However maintaining the j-relevant tuple relation along the path from i_j to i_{j+1} is essentially the same than for words. We provide a version of Lemma 20 for trees.

Lemma 42. *For a node $v \in \mathrm{N}_t^{aux}$, we can compute $R_j(v)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$ in each of the following cases:*

(1) v is a leaf in N_t^{aux}, $v = i_j$, and we know $G(i_j)$ and $R_{j-1}(i_j)$;
(2) v has a parent v_p, $i_j \preceq_t v_p$, and we already know $R_j(v_p)$; and
(3) v has a child v_c, $i_j \prec_t v$, and we know $R_j(v_c)$ and $R_{j-1}(v)$.

Proof. We prove cases (1) to (3) separately. However, notice that, by Definition 41, we have to show that $R_j(v)$ for each case (1) to (3) is the set of all tuples (q_1, q_2, I) such that

- $(q_1, q_2, I) \in R(v)$, and
- (q_1, q_2, I) is compatible with some $(q, A) \in G(i_j)$.

(1) For case (1), the proof is analogous to the proof of case (1) of Lemma 20.
(2) To prove (2), we distinguish two subcases (a) and (b).
  (a) If v and v_p are nodes in the same tree $\mathrm{N}_p^{\mathrm{aux}}$, then the proof is analogous to case (2) in Lemma 20.
  (b) Otherwise, we know that v is the root node of some $\mathrm{N}_p^{\mathrm{aux}}$. (Notice that this is the point where Definition 28 differs from Definition 12.) Let $p = v_n \cdots v_1$ such that v_1 is a leaf in t. Then, we distinguish two subcases whether v_n is a left of a right child in t. However, both cases are analogous to each other and we show only the case where v_n is a left child in the following. In this case, we define

$$R_j(v) = \{(q_0, q_1, I') \mid \exists (q_0, q_1, I) \in \mathrm{T}_p^+(v) \text{ with } I' \subseteq I, \exists (q_2, q, I) \in R_j(v_p), q \in \delta(q_1, q_2, \mathrm{lab}(v_p)) \text{ and } I' \cup \{q\} = J\}.$$

  Since $(q_2, q, I) \in R(v_p)$, we know that (q_2, q, I) is compatible with some $(q, A) \in G(i_j)$. Because $i_j \preceq_{\mathrm{t}} v_p$, it holds that $\mathrm{Nodes}(A_{[\{v\}]}) = \emptyset$. But then it directly follows that (q_0, q_1, I') is also compatible with (q, A) and, therefore, that the tuple is also relevant. Now, assume there is a tuple $(q_0, q, I') \in R(v)$ which is compatible with some $(q, A) \in G(i_j)$ but is not in $R_j(v)$. Since $(q_0, q, I') \in R(v)$ it holds, by Definition 28, that there is a tuple $(q_0, q, I) \in \mathrm{T}_p^+(v)$ with $I' \subseteq I$. Furthermore by the Definition of compatibility, we know that there also has to exist a tuple $(q_2, q, I) \in R_j(v_p)$ which is compatible with $(q, A) \in G(i_j)$ and that $q \in \delta(q_1, q_2, \mathrm{lab}(v_p))$ such that $I' \cup \{q\} = J$. Altogether, this contradicts the assumption that $(q_0, q, I') \notin R_j(v)$. We can compute $R_j(v)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$.
(3) To prove (3), we distinguish two subcases (a) and (b).
  (a) If v and v_c are nodes in the same tree $\mathrm{N}_p^{\mathrm{aux}}$, then the proof is analogous to case (3) in Lemma 20.
  (b) Otherwise, we know that v is a leaf node of some $\mathrm{N}_p^{\mathrm{aux}}$. (Notice that this is the point where Definition 28 differs from Definition 12.) Let $p = v_n \cdots v_1$ such that v_1 is a leaf in t. Then, we distinguish two subcases whether v_n in t is a left child or not. However, both cases are analogous to each other and we show only the case where v_n is a left child in the following. In this case, we define

$$R_j(v) = \{(q_2, q, I) \mid (q_2, q, I) \in R_{j-1}(v), (q_0, q_1, J) \in R_j(v_c), \\ q \in \delta(q_1, q_2, \mathrm{lab}(v)) \text{ and } I = (\{q\} \cup J) \cap \mathrm{set}(s) \text{ for some } s \in S\}.$$

  Because $(q_2, q, I) \in R_{j-1}(v)$, it directly holds that $(q_2, q, I) \in R(v)$. It remains to show that (q_2, q, I) is compatible with some $(q, A) \in G(i_j)$. That is, we need to find an annotated answer A^{full} w.r.t some run r_A such that (q_2, q, I) and $(q, A) \in G(i_j)$ are compatible with A^{full} and r_A. However, the construction of such A^{full} and r_A is completely analogous to case (3)(b) in Lemma 20. We can compute $R_j(v)$ in time $\mathcal{O}(|Q|^3 \cdot 2^k)$.

□

The following results can be proved analogous to Lemma 21 and Theorem 22.

Lemma 43. *Given N_t^{aux}, T_p^+, and $\ell \in [k]$, we can compute $G(i_\ell)$ in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot \ell \log^2 |t|)$.*

Theorem 44. *Given N_t^{aux} and T_p^+, we can compute the first answer of M on t in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$.*

From One Answer to the Next for Trees

We can also obtain a similar lemma as given by Lemma 23 for trees.

Lemma 45. *Complete, Back, and Nextnode can be implemented such that Algorithm 1 for trees correctly computes Next($\mathcal{A}$). Furthermore, Next($\mathcal{A}$) runs in $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$ time.*

Finally, we are able to obtain the main result for the tree case.

PROOF OF 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.*

Proof. In the preprocessing phase, we build the tree $\mathrm{N}_t^{\mathrm{aux}}$ and relation T_p^+. By Lemma 27, the data has size $\mathcal{O}(|Q|^2 \cdot 2^k \cdot |t|)$ and can be computed in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot |t|)$. By the same lemma, it follows that updates can be maintained in time $\mathcal{O}(|Q|^3 \cdot 2^k \cdot \log^2 |t|)$. By Lemma 35, we know that we can use Algorithm 1 for trees to correctly enumerate all answers in $M(w)$ if Complete$(\mathcal{A})$, Back$(\mathcal{A})$, and Nextnode$(\mathcal{A})$ can be implemented correctly within the required running time. By Theorem 44, Complete$(\{\emptyset\})$ can be implemented correctly such that it runs in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$. Finally, by Lemma 45, it holds that Next$(\mathcal{A})$ can be implemented correctly such that it runs in time $\mathcal{O}(|S| \cdot k! + |Q|^3 \cdot 2^k \cdot k \log^2 |t|)$. $\square$