

Enumeration on Trees with Tractable Combined Complexity and Efficient Updates

Antoine Amarilli

LTCI, Télécom ParisTech, Université Paris-Saclay

Stefan Mengel

CNRS, CRIL UMR 8188

Pierre Bourhis

CRISTAL, CNRS UMR 9189 & Inria Lille

Matthias Niewerth

University of Bayreuth

ABSTRACT

We give an algorithm to enumerate the results on trees of monadic second-order (MSO) queries represented by *non-deterministic* tree automata. After linear time preprocessing (in the input tree), we can enumerate answers with linear delay (in each answer). We allow updates on the tree to take place at any time, and we can then restart the enumeration after logarithmic time in the tree. Further, all our combined complexities are polynomial in the automaton.

Our result follows our previous circuit-based enumeration algorithms based on deterministic tree automata, and is also inspired by our earlier result on words and non-deterministic sequential extended variable-set automata in the context of document spanners. We extend these results and combine them with a recent tree balancing scheme by Niewerth, so that our enumeration structure supports updates to the underlying tree in logarithmic time (with leaf insertions, leaf deletions, and node relabelings). Our result implies that, for MSO queries with free first-order variables, we can enumerate the results with linear preprocessing and constant-delay and update the underlying tree in logarithmic time, which improves on several known results for words and trees.

Building on lower bounds from data structure research, we also show unconditionally that up to a doubly logarithmic factor the update time of our algorithm is optimal. Thus, unlike other settings, there can be no algorithm with constant update time.

ACM Reference Format:

Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Enumeration on Trees with Tractable Combined Complexity and Efficient Updates. In *Proceedings of 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '19)*. ACM, New York, NY, USA, 27 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Query evaluation is one of the central tasks in databases. A prominent framework for this task in theoretical work are so-called *enumeration algorithms*. Such algorithms print out query answers one after the other, without duplicates, and bound the time between each two answers, called the *delay* of the algorithm. Enumeration algorithms were introduced in database theory by the pioneering work of Durand and Grandjean [21], in which the goal was to achieve a constant bound on the delay, called *constant-delay enumeration*. In such algorithms, the delay between different solutions only depends on the query to be evaluated, in particular it is independent of the database. To ensure such a bound, the algorithm is allowed to first perform a *preprocessing phase* on the database, which in most cases is required to run in linear time. While these time bounds sound very restrictive, a surprising range of query evaluation problems allow enumeration in this regime, as surveyed, e.g., in [36]. Most notably, constant-delay enumeration with linear-time preprocessing can be achieved for acyclic free-connex queries on arbitrary databases [9] (with extensions for the case of functional dependencies [19] and for unions of conjunctive queries [13]), for first-order queries on bounded degree databases [11] and nowhere dense databases [35] (with pseudo-linear time preprocessing), for regular path queries [28], and for monadic second-order queries on trees and bounded-treewidth structures [8, 25] which is the setting that we study.

Enumeration algorithms have recently been extended to handle the fact that practical databases are rarely static but

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PODS '19, Extended version, Appendices included

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Work	Data	Delay	Updates
Bagan [8], Kazana and Segoufin [25]	trees	$O(1)$	N/A
Losemann and Martens [26]	words	$O(\log n)$	$O(\log n)$
Losemann and Martens [26]	trees	$O(\log^2 n)$	$O(\log^2 n)$
Niewerth and Segoufin [31]	words	$O(1)$	$O(\log n)$
Niewerth [30]	trees	$O(\log n)$	$O(\log n)$
Amarilli, Bourhis, Mengel [4]	trees	$O(1)$	$O(\log n)^1$
this paper	trees	$O(1)$	$O(\log n)$

¹: Only supports *relabeling updates*

Table 1: State of the art for enumeration of MSO query results (for free FO variables) on trees under updates. All these algorithms have linear time preprocessing.

change frequently. Specifically, it is important for enumeration algorithms to efficiently handle updates to the underlying database, and maintain their precomputed index structures without recomputing them from scratch at each update. This approach was inspired by the work of Balmin et al. [10] that showed how to maintain queries on trees efficiently under updates, but only for Boolean queries, i.e., without the ability to enumerate results. The work of Losemann and Martens [26] was the first to extend this to an update-aware enumeration algorithm, which handled efficient enumeration of queries in monadic second order logic (MSO) queries on words and trees. Note that the efficiency in such update-aware enumeration algorithms is measured by three parameters: the enumeration delay, the time for an initial preprocessing, and the time necessary for each update. Since [26], several such algorithms have been designed in several settings, namely, CQs [12] and UCQs [13], FO+MOD queries on bounded degree databases [11], and MSO queries over words and trees (see Table 1).

In this paper, we focus on the setting where we evaluate MSO queries on trees [8]. This task is motivated by, e.g., querying tree-shaped data such as XML or JSON documents, and the important special case of query evaluation on words, e.g., for information extraction using *document spanners* [22, 23]. It is already known that for any MSO query the answers on a given tree can be enumerated in linear time [8, 25], and this was extended to updates by Losemann and Martens [26] but at the cost of increasing the enumeration delay to a polylogarithmic term (in particular making it depend on the database). This delay and update time were recently made logarithmic in [30], and two incomparable algorithms can achieve constant delay in special cases: when restricting the structure to be a word [31], or when restricting updates to *relabelings*, i.e., disallowing structural changes on the input tree. See Table 1 for an overview.

Our first contribution in this paper is to show an update-aware enumeration algorithm for MSO query evaluation on trees that achieves linear-time preprocessing, constant-delay enumeration, and supports updates in logarithmic time including node insertions and deletions. This strictly outperforms all previously known algorithms and shows that updates can be handled in this setting without worsening the delay. Further, our algorithm can also handle the case of monadic second-order queries with free second-order variables, for which query answers no longer have constant size: in this case the delay is linear in the size of each produced answer, as in the static case [8].

Our result is shown using the *circuit-based approach* to enumeration developed in [2, 3] and also used in [4] to show the result for relabelings. In this approach, the input is first translated into a circuit representation, and we impose on the circuit some properties inspired by similar concepts in knowledge compilation, namely, that it is a complete structured DNNF whose width only depends on the query. Then the actual enumeration is performed on this circuit by relying on these properties. This approach of translating MSO queries on trees into circuits was first proposed for probabilistic query evaluation and provenance computation [7], and has the advantage of being applicable to different enumeration problems with very few adaptations, and of being modular (i.e., in the enumeration we no longer need to refer to the database or query). Our circuit based approach is also related to factorized [32] representations where query results are first encoded into a compact representation before performing other tasks on it, e.g., aggregation.

The circuit-based approach to enumeration lends itself well to relabeling updates, as we already presented in [4]: we can toggle the value of gates when labels change, and then propagate information upwards in the circuit. The update time then depends on the *depth* of the circuit, which in our construction is linear in the height of the input tree. In [4] we side-stepped this problem by computing a balanced tree decomposition on the input during the preprocessing phase using [14], so that we applied our construction to a tree of logarithmic height to achieve logarithmic update time when relabeling nodes of the tree. However, we did not know how to maintain such a balanced tree decomposition under other updates like leaf additions and deletions.

Our first contribution is shown by observing that we can combine the techniques of [4] with those developed in the recent independent work [30]. There, it was shown that one can represent trees as balanced *forest algebra* terms [15, 16] and efficiently maintain them under updates. We show that this technique can also be used in the circuit-based approach, formalizing the forest algebra updates via *tree hollowings* (in Section 7) and giving an entirely bottom-up presentation of circuit construction and enumeration (in Sections 3–6). This

allows us to show our main result that generalizes the incomparable results of [8, 25], [31], [30], and [4].

The second contribution of this paper is an enumeration algorithm which is tractable *in combined complexity*. Indeed, the complexities given in Table 1 only apply to data complexity, i.e., complexity in the database when assuming that the query is fixed. However, if we want our enumeration algorithms to be applicable, it is important that the complexity remains tractable, e.g., polynomial, when the query is given as part of the input. This is an unreasonable hope when the query is written in MSO, as the combined complexity of Boolean MSO query evaluation (even without enumeration) is generally nonelementary [29]. However, if the query is given as a *tree automaton*, then Boolean evaluation has tractable combined complexity. But achieving combined tractability for enumeration is challenging: all results in Table 1 are either intractable in the query or are only tractable when assuming that the input query is represented as a *deterministic tree automaton*. This assumption is problematic already in the case of words, as we cannot, e.g., convert an input regular expression query on words to a deterministic automaton in polynomial time. In our recent work [5], we showed that enumeration *on words* (and without updates) could be performed with linear-preprocessing and constant delay while being polynomial in an input *nondeterministic automaton* (presented in the context of information extraction and *document spanners* [22]). In the present paper, we extend these techniques to make our enumeration algorithm polynomial in an input nondeterministic tree automaton representation of the query; in other words we perform enumeration on circuits that are not *deterministic* [20]. This is performed in Sections 5–6 by extending the techniques of [5] to trees. Our results also imply efficient enumeration algorithms to evaluate the matches of automata on words and efficiently maintain these results under updates, generalizing some results of [5] to the case of dynamic words.

Our third contribution is to show that our algorithm is close to optimal. We leverage known lower bounds from data structure research [1] to show unconditionally that the update time for an enumeration algorithm for MSO queries has to be $\Omega(\log n / \log \log n)$ when the delay is constant, or even when it is allowed to be in $o(\log n / \log \log n)$. Thus, our result is optimal up to a doubly logarithmic factor.

Paper Structure. We give preliminaries in Section 2. We present in Section 3 how to construct from the input tree and nondeterministic automaton a circuit representation of the results. We present our enumeration algorithm on such circuits in Sections 4–6, starting with a simple scheme in Section 4 which enumerates the results with duplicates and large delays, refining it in Section 5 to avoid duplicates, and showing how to achieve the right delay bound in Section 6.

In Section 7, we show how the algorithm can support updates using [30], leading to our main results in Section 8. We present our lower bounds in Section 9, before concluding in Section 10. Full proofs are deferred to the full version [6].

2 PRELIMINARIES

We define words, trees, and valuations, present our automata and a homogenization lemma, and state our problem.

Trees and Valuations. In this paper, we work with trees that are all *rooted* and *ordered*, i.e., there is an order on the children of each node. Given a set Λ of tree labels, a Λ -*tree* T (or *tree* when Λ is clear from context) is a pair of a rooted tree (also written T) and of a *labeling function* λ that maps each node n of T to a *label* $\lambda(n) \in \Lambda$. We write $\text{Leaf}(T)$ for the set of leaves of T . We abuse notation and identify T with its set of nodes, i.e., we can write that $\text{Leaf}(T) \subseteq T$. An *internal node* is a node of $T \setminus \text{Leaf}(T)$. Until Section 7, all trees in this paper will be *binary*, i.e., every internal node has exactly two children, which we refer to as *left* and *right* child.

When evaluating a query with *variables* X on a Λ -tree T , we will see its possible results as *valuations*: an X -*valuation* of T is a function $v : \text{Leaf}(T) \rightarrow 2^X$ that assigns to every leaf n of T a set of variables $v(n) \subseteq X$ called the *annotation* of n . Note that our variables are second-order, i.e., each variable can be interpreted as a set of nodes of T . We represent valuations concisely as *assignments*: an X -*assignment* is a set S of *singletons* which are pairs of the form $\langle Z : n \rangle$, where $Z \in X$ and $n \in \text{Leaf}(T)$. The *size* $|S|$ of S is simply the number of singletons that it contains. There is a clear bijection between X -valuations and X -assignments, so we write $\alpha(v)$ for an X -valuation v to mean the assignment $\{\langle Z : n \rangle \mid Z \in v(n)\}$, and write $|v| := |\alpha(v)|$. We often write $\langle \mathcal{Y} : n \rangle$ for some $n \in T$ and some *non-empty* set $\mathcal{Y} \subseteq X$, to mean the non-empty set of singletons $\{\langle Z : n \rangle \mid Z \in \mathcal{Y}\}$.

Tree Variable Automata. We will write our query on trees using *tree automata* that can express, e.g., queries in *monadic second-order logic* (MSO): see [37] and [3, Appendix E.1]. Formally, a *tree variable automaton* TVA on binary Λ -trees for variable set X (or Λ, X -TVA) is a tuple $A = (Q, \iota, \delta, F)$, where Q is a finite set of *states*, $\iota \subseteq \Lambda \times 2^X \times Q$ is the *initial relation*, $\delta \subseteq \Lambda \times Q \times Q \times Q$ is the *transition relation*, and $F \subseteq Q$ is the set of *final states*. The *size* $|A|$ of A is $|Q| + |\iota| + |\delta|$. This definition only applies to *binary* Λ -trees; the analogous automata for unranked trees will be introduced in Section 7.

To simplify notation we often see δ as a tuple of functions, i.e., for each $l \in \Lambda$ we have a function $\delta_l : Q \times Q \rightarrow 2^Q$ defined by $\delta_l(q_1, q_2) = \{q \in Q \mid (l, q_1, q_2, q) \in \delta\}$: this intuitively tells us to which states the automaton can transition on an internal node with label l when the states of the two children are respectively q_1 and q_2 . Note that, following our

definition of a valuation and of ι , the automaton is only reading annotations on leaf nodes.

Having fixed Λ and X , given a Λ -tree T and an X -valuation ν of T , given a Λ, X -TVA $A = (Q, \iota, \delta, F)$, a *run* of A on T under ν is a function $\rho : T \rightarrow Q$ satisfying the following:

- For every $n \in \text{Leaf}(T)$, we have $(\lambda(n), \nu(n), \rho(n)) \in \iota$;
- For every internal node n with label l and children n_1, n_2 , we have $\rho(n) \in \delta_l(\rho(n_1), \rho(n_2))$.

The run is *accepting* if it maps the root of T to a state in F , and we say that A *accepts* T under ν if there is an accepting run of A on T under ν . The *satisfying valuations* of A on T is the set of the X -valuations ν of T such that A accepts T under ν , and the *satisfying assignments* are the corresponding assignments $\alpha(\nu)$. Thus, the automaton A defines a query on Λ -trees with second-order variables X , and its results on a Λ -tree T are the satisfying assignments of A on T .

Homogenization. It will be useful to assume a *homogenization* property on automata. Given a Λ, X -TVA $A = (Q, \iota, \delta, F)$, we call $q \in Q$ a *0-state* if there is some Λ -tree T and run ρ of A on T that maps the root of T to q under the *empty* X -valuation ν_\emptyset of T defined as $\nu_\emptyset(n) := \emptyset$ for each $n \in \text{Leaf}(T)$. We call q a *1-state* if there is some Λ -tree T and run ρ of A on T mapping the root of T to q under some *non-empty* X -valuation, i.e., a valuation ν different from the empty valuation. Intuitively, a 0-state is a state that A can reach by reading a tree annotated by the empty valuation, and a 1-state can be reached by reading a tree with at least one non-empty annotation. In general a state can be both a 0-state and a 1-state, or it can be neither if there is no way to reach it. We say that A is *homogenized* if every state is either a 0-state or a 1-state and no state is both a 0-state and a 1-state. We can easily make automata homogenized, by duplicating the states to remember if we have already seen a non-empty annotation:

LEMMA 2.1. *Given a Λ, X -TVA A , we can compute in linear time a Λ, X -TVA A' which is homogenized and equivalent to A .*

Problem statement. Our goal in this paper is to efficiently enumerate the results of queries on trees. The inputs to the problem are the Λ -tree T and the query given as a Λ, X -TVA A , with Λ the tree alphabet and X the variables. The output is the set of the satisfying assignments of A and T . We present an *enumeration algorithm* to produce them, which first runs a *preprocessing phase* on A and T : we compute a concise representation of the output as an *assignment circuit* (Section 3), and compute an *index structure* on it (Section 6). Second, the *enumeration phase* produces each result to the query, with no duplicates, while bounding the maximal *delay* between two successive answers (Sections 4–6). Third, our algorithm must handle *updates* to T , i.e., given an edit

operation on T , efficiently update the assignment circuit and index and restart the enumeration on the updated tree (Section 7).

Our main result shows how to solve this problem with preprocessing linear in T and polynomial in A ; with delay independent from T , polynomial in A , and linear in each produced assignment; and with update time logarithmic in T and polynomial in A . This result is formally stated in Section 8.

3 BUILDING ASSIGNMENT CIRCUITS

In this section, we start describing the preprocessing phase of our enumeration algorithm. Given the input TVA and binary tree, we will build an *assignment circuit* that concisely represents the set of satisfying assignments: we do this by adapting the circuit constructions in our earlier work [2, 4] to give them better support for updates and handle non-deterministic automata.

We first define our circuit formalism, which we call *set circuits*, and their semantics. Second, we define when a set circuit can serve as an *assignment circuit* for a TVA on a binary tree. Third, we present properties that our circuits will satisfy, namely, they are *complete structured DNNFs*, and we can bound a *width* parameter for them. Last, we state our main circuit construction result at the end of the section (Lemma 3.7): it shows that, given a homogenized TVA A and binary tree T , we can construct an assignment circuit of A on T in time $O(|A| \times |T|)$ while respecting our properties and controlling the width parameter.

The circuit produced by Lemma 3.7 will then be fed to the enumeration algorithms presented in Sections 4–6. To extend enumeration to updates in Section 7, we will leverage the fact that all our constructions work by processing T bottom-up.

Set Circuits. A *set circuit* (or just *circuit*) $C = (G, W, \mu)$ consists of a directed acyclic graph (G, W) where G are the *gates* and $W \subseteq G \times G$ are the *wires*, and of a function μ mapping each gate to a type among $\top, \perp, \text{var}, \times, \cup$: we will accordingly call g a \top -gate, \perp -gate, var-gate, \times -gate, or \cup -gate depending on $\mu(g)$. There is also an injective function S_{var} mapping each var-gate g to some set $S_{\text{var}}(g)$ of *variables*. We write C_{var} to refer to the variables occurring in C , i.e., $C_{\text{var}} := \bigcup_{g \in G \mid \mu(g) = \text{var}} S_{\text{var}}(g)$. The set of *inputs* of a gate $g \in G$ is $\{g' \in G \mid (g', g) \in W\}$. We require that:

- \top -, \perp -, and var-gates have no inputs;
- \times -gates have exactly 2 inputs;
- \cup -gates have at least 1 input;
- \top -gates and \perp -gates are never used as inputs.

The *maximal fan-in* of C is the maximum number of inputs of a gate and its *depth* is the maximum length of a directed path of wires.

The goal of set circuits is to concisely represent sets. Formally, each gate *captures* a set, in the following sense:

Definition 3.1. Given a set circuit $C = (G, W, \mu)$, we define for each gate $g \in G$ a *captured set* $S(g)$ which is a set of subsets of C_{var} defined inductively as follows:

- If g is a var-gate then $S(g) := S_{\text{var}}(g)$.
- If g is a \perp -gate then $S(g) := \emptyset$.
- If g is a \top -gate then $S(g) := \{\emptyset\}$.
- If g is a \times -gate with inputs g_1 and g_2 then $S(g) := \{S_1 \cup S_2 \mid S_1 \in S(g_1), S_2 \in S(g_2)\}$.
- If g is a \cup -gate then, letting g_1, \dots, g_m be the inputs of g , we define $S(g) := \bigcup_{1 \leq i \leq m} S(g_i)$.

Example 3.2. Consider the circuit C featuring a \times -gate g with one input g'_1 being a var-gate with $S_{\text{var}}(g'_1) = \{x\}$, and one input g_2 being a \cup -gate with two input var-gates g'_2 and g''_2 with $S_{\text{var}}(g'_2) = \{y\}$ and $S_{\text{var}}(g''_2) = \{y, z\}$. We have $C_{\text{var}} = \{x, y, z\}$ and $S(g) = \{\{x, y\}, \{x, y, z\}\}$.

Assignment Circuits. An *assignment circuit* for a TVA on a tree is a set circuit where, for each automaton state q and tree node n , there is a gate $\gamma(q, n)$ capturing the assignments for which we reach state q on node n . Formally:

Definition 3.3. Given a binary Λ -tree T and a Λ, \mathcal{X} -TVA $A = (Q, \iota, \delta, F)$, an *assignment circuit* of T on A consists of a circuit C where C_{var} is the set of singletons $\{\langle Z : n \rangle \mid Z \in \mathcal{X}, n \in \text{Leaf}(T)\}$, and a mapping $\gamma : T \times Q \rightarrow C$ such that for any $n \in T$ and $q \in Q$, the gate $\gamma(n, q)$ is a \cup -gate, \top -gate, or \perp -gate, and for any \mathcal{X} -valuation v of the subtree T_n of T rooted at n , we have $\alpha(v) \in S(\gamma(n, q))$ iff there is a run of A on T_n under v that maps the root node n to state q .

Note that an assignment circuit concisely represents the satisfying assignments of A on T : they are $\bigcup_{q \in F} S(\gamma(n, q))$ where n is the root of T . Thus, our goal is to use assignment circuits to efficiently enumerate satisfying assignments. To be able to do this, we will impose certain properties on assignment circuits, which we now define.

Complete Structured DNNFs. The circuits that we build will be *complete structured DNNFs*, and we will control their *width*. The notion of structured DNNF is inspired by knowledge compilation [34], with DNNF meaning *decomposable negation normal form*, and completeness is also inspired by that field [17]; we adapt it here to set circuits rather than Boolean circuits (as we also did in [2, 4]), with no negations, and with *decomposability* intuitively implying that no variable can occur in assignments in both the left and the right input gate of some \times -gate. Formally:

Definition 3.4. A *v-tree* \mathcal{T} for a set circuit C is a binary tree whose leaves are labeled by sets of variables that form a partition of C_{var} . A *structuring function* σ from C to \mathcal{T} maps each gate g of C to a node $\sigma(g)$ of \mathcal{T} such that:

- For every var-gate g , the node $\sigma(g)$ is a leaf of \mathcal{T} , and the variables that label g in C are a subset of the variables of $\sigma(g)$ in \mathcal{T} ; formally, letting \mathcal{Y} be the set of variables that labels $\sigma(g)$ in \mathcal{T} , we have $S_{\text{var}}(g) \subseteq \mathcal{Y}$.
- Whenever a gate g' is an input gate to a gate g , then either g and g' are mapped to the same v-tree node, or the input g' must be a \cup -gate that is mapped to the child of the node of g . Formally, for every wire $(g', g) \in W$ of C , either $\sigma(g) = \sigma(g')$, or g' is a \cup -gate and $\sigma(g')$ is a child of $\sigma(g)$ in \mathcal{T} .
- Each \times -gate g has one input gate g_1 such that $\sigma(g_1)$ is the left child of $\sigma(g)$ and one input gate g_2 such that $\sigma(g_2)$ is the right child of $\sigma(g)$; we call g_1 and g_2 the *left* and *right* inputs. Note that, by the previous point, g_1 and g_2 must be \cup -gates.

Note that these points ensure that C is *decomposable*, namely, for any \times -gate g with inputs g_1 and g_2 , no variable gate in C has a path both to g_1 and to g_2 . In particular, when defining $S(g)$ according to Definition 3.1, there can never be any duplicate in the union that defines the relational product.

A *complete DNNF structured by \mathcal{T}* is a set circuit C together with a v-tree \mathcal{T} and a structuring function σ from C to \mathcal{T} .

Example 3.5. The circuit C in Example 3.2 is not a complete structured DNNF. However, consider C' built from C where the first input of the \times -gate g is now a \cup -gate g_1 having the var-gate g'_1 as its only input. Then C' is a complete structured DNNF for the v-tree \mathcal{T}' whose root has a left child n_1 labeled $\{x\}$ and a right child n_2 labeled $\{y, z\}$; the structuring function σ maps g to the root of \mathcal{T}' , maps g_1 and g'_1 to n_1 , and g_2, g'_2 , and g''_2 to n_2 .

When we have a complete structured DNNF C with a structuring function σ to a v-tree \mathcal{T} , we see the gates of C as partitioned into *boxes*, with each box being the preimage of some node of \mathcal{T} by σ . We use $\text{BOX}(g)$ to denote the box of some gate g (formally, $\text{BOX}(g) := \sigma^{-1}(\sigma(g))$). We talk about the *tree of boxes* to mean the structure on boxes that follows \mathcal{T} . In particular, given a box B , letting n be the node of \mathcal{T} such that $B = \sigma^{-1}(n)$, if n is an internal node then we call B a *non-leaf box* and denote by $\text{LEFT}(B)$ and $\text{RIGHT}(B)$ its *left child box* and *right child box* in the tree of boxes, i.e., $\sigma^{-1}(n_1)$ and $\sigma^{-1}(n_2)$ respectively, where n_1 and n_2 are the children of n in \mathcal{T} . We will use boxes to define a structural parameter of complete structured DNNFs, called *width*, and similar to width in [17].

Definition 3.6. The *width* of a structured complete DNNF is the maximal number of \cup -gates in a box, i.e.,

$$\max_B |\{g \in B \mid \mu(g) = \cup\}|.$$

While this notion of width only bounds the number of \cup -gates in each box, we can always rewrite a structured complete DNNF of width w in linear time to ensure that the number of \times -gates in each box is also bounded by w^2 [17, Observation 3]. Intuitively, each \times -gate in a box B has two \cup -gates as input, one in $\text{LEFT}(B)$ and one in $\text{RIGHT}(B)$, so there are at most w^2 non-equivalent combinations. Hence, we will assume that this bound holds on all circuits that we manipulate (and in fact our circuit construction obeys it directly, with no rewriting needed).

Building Assignment Circuits. We have defined the assignment circuits that we want to compute, defined the notion of a structured complete DNNF and a width parameter for them. We can now state our main result for this section, namely, that we can efficiently construct assignment circuits. Observe that, while the depth of the circuit depends on the input tree, the width only depends on $|Q|$, which will be crucial for our delay bounds.

LEMMA 3.7. *Given any binary Λ -tree T and homogenized $\Lambda, X\text{-TVA } A = (Q, \iota, \delta, F)$, we can construct in time $O(|T| \times |A|)$ a structured complete DNNF C which is an assignment circuit of A and T , a v -tree \mathcal{T} , and a structuring function from C to \mathcal{T} , such that C has width $|Q|$ and depth $O(\text{height}(T))$.*

PROOF SKETCH. We construct \mathcal{T} by taking T , removing all node labels, and labeling each leaf node n by the set of singletons $\langle X : n \rangle$: thus, \mathcal{T} is a v -tree for the set of variables $C_{\text{var}} = \{\langle Z : n \rangle \mid Z \in X, n \in T\}$ of C given by the definition of assignment circuits.

We now build C bottom up. For a leaf node n of T with label $l \in \Lambda$, we build the box B_n for n by setting $\gamma(n, q)$ for all $q \in Q$ as:

- a \perp -gate, if there are no tuples of the form $(l, \mathcal{Y}, q) \in \iota$,
- a \top -gate, if $(l, \emptyset, q) \in \iota$, and
- a \cup -gate having as inputs one variable gate labeled by $\langle \mathcal{Y} : n \rangle$ for each non-empty $\mathcal{Y} \subseteq X$ such that $(l, \mathcal{Y}, q) \in \iota$, otherwise.

As A is homogenized, the first and last case are disjoint, i.e., we cannot have both $(l, \emptyset, q) \in \iota$ and $(l, \mathcal{Y}, q) \in \iota$ with $\mathcal{Y} \neq \emptyset$.

For an inner node n of T with label l and child nodes n_1 and n_2 , we construct the box B_n as follows. For every triple $(q_1, q_2, q) \in \delta_l$, we define a \times -gate g^{q_1, q_2} with inputs $\gamma(n_1, q_1)$ and $\gamma(n_2, q_2)$. If there is no such triple, we let $\gamma(n, q)$ be a \perp -gate. Otherwise, we let $\gamma(n, q)$ be a \cup -gate that has all such \times -gates g^{q_1, q_2} as input.

In terms of accounting, it is clear that there are at most $|Q|$ \cup -gates in each B_n , that the depth of the circuit is as stated, and the construction of the whole circuit is in time $O(|A| \times |T|)$ as promised. Now, a straightforward bottom-up induction on T shows that the gates $\gamma(n, q)$ capture the correct set for any n , i.e., that for any leaf node n and any

$q \in Q$ we have: $S(\gamma(n, q)) = \{\langle \mathcal{Y} : n \rangle \mid (\lambda(n), \mathcal{Y}, q) \in \iota\}$ and for any internal node n with label l and children n_1 and n_2 and any $q \in Q$ we clearly have:

$$S(\gamma(n, q)) = \bigcup_{(q_1, q_2, q) \in \delta_l} S(\gamma(n_1, q_1)) \times S(\gamma(n_2, q_2))$$

It is easy to check that all rules of assignment circuits are respected with the exception of the rule that \top - and \perp -gates are never allowed as inputs to other gates. In particular, all \cup -gates and \times -gates have the right fan-in. To ensure that \top - and \perp -gates are never used as inputs, one can use a slightly modified construction (see appendix) that avoids adding outgoing wires to \top - and \perp -gates by treating these cases in a special way. This uses the fact that, as A is homogenized, there is no gate $\gamma(n, q)$ that captures both the empty assignment and some non-empty assignment. \square

4 SIMPLE ENUMERATION ALGORITHM

In the three following sections, we will present how to enumerate the set of assignments captured by gates of assignment circuits. We start in this section by presenting an algorithm which is simple but has two important drawbacks. First, the worst-case delay is $O(\text{depth}(C))$, i.e., linear in the depth of the circuit C . Second, assignments are output multiple times. We will refine this algorithm in Section 5 to ensure that every assignment is enumerated exactly once. Last in Section 6, we show how to bound the delay by the width of the circuit (instead of the depth).

To define our enumeration algorithms in this and the following sections, we introduce some useful notation. For any \cup -gate g of C , for any gate g' of C , we write $g' \rightsquigarrow g$ if there is a path $g' = g_1, \dots, g_n = g$ from g' to g in C where each (g_i, g_{i+1}) is a wire in W and where all intermediate gates g_2, \dots, g_n are \cup -gates. We then write $\downarrow(g)$ to mean the set of var-gates and \times -gates g' such that $g' \rightsquigarrow g$. The following observation shows why $\downarrow(g)$ is useful for enumeration: it is proven by an immediate induction on \cup -gates:

OBSERVATION 4.1. *For any \cup -gate $g \in C$, we have that $S(g) = \bigcup_{g' \in \downarrow(g)} S(g')$.*

We observe that we can enumerate $\downarrow(g)$ for every gate g by doing a simple preorder traversal of the circuit: however, doing this naively only ensures a delay of $O(\text{depth}(C))$, and it enumerates each gate g' as many times as there are paths that witness $g' \rightsquigarrow g$ in C . We denote by $\text{ENUM}_{\downarrow}^{\text{DUPES}}(g)$ this naive procedure.

Using this procedure, we present our enumeration algorithm for $S(g)$ as Algorithm 1. The algorithm applies to any decomposable set circuit and does not use the v -tree or the structuring function.

Algorithm 1 is presented using “output” statements to produce new results (like, e.g., Python’s “yield”). When we

Algorithm 1 Simple enumeration algorithm

```

1: procedure  $\text{ENUM}_S^{\text{DUPES}}(g)$ 
2:   for  $g' \in \text{ENUM}_S^{\text{DUPES}}(g)$  do
3:     if  $g'$  is a var-gate then output  $\{S_{\text{var}}(g')\}$ 
4:     else  $\triangleright g'$  is a  $\times$ -gate
5:       for  $S_L \in \text{ENUM}_S^{\text{DUPES}}(\text{left input of } g')$  do
6:         for  $S_R \in \text{ENUM}_S^{\text{DUPES}}(\text{right input of } g')$  do
7:           output  $S_L \cup S_R$ 

```

recursively use the enumeration algorithm on a subcircuit (as in lines 5 and 6), we assume that this enumeration is started in another thread, which will run until the first output is produced. Afterwards the new thread pauses until the calling thread requests the next value. Whenever the calling thread requests a new value, the called thread runs until it produces the next output. The following result is now not hard to see.

PROPOSITION 4.2. *Given a structured complete DNNFC and \cup -gate g , Algorithm 1 enumerates $S(g)$ (with duplicates) with delay $O(\text{depth}(C) \times |S|)$, where S is the produced assignment.*

PROOF. It is clear that Algorithm 1 is correct, because it directly follows Observation 4.1, and the inputs to \times -gates are always \cup -gates so we always call the algorithm on a \cup -gate. In terms of delay, each assignment of size k produced by the algorithm required at most $2k - 1$ recursive calls, each of which correspond to a var-gate or \times -gate used when producing the assignment: note that this uses the decomposability of C , and uses the fact that \top -gates are never used as inputs to another gate. Now, each recursive call has delay $O(\text{depth}(C))$ for the call to $\text{ENUM}_S^{\text{DUPES}}(g)$, hence the delay of the algorithm is as claimed, which completes the proof. \square

As a side remark, note that the number of times that Algorithm 1 enumerates each assignment is related to the number of runs of the TVA for this assignment. Specifically, up to redefining Definition 3.1 with multisets, and up to small changes in Lemma 3.7, we could ensure that each assignment in $S(\gamma(n, q))$ is enumerated exactly as many times as there are runs on the subtree rooted at n under the corresponding valuation such that the root node is mapped to q .

5 ELIMINATING DUPLICATES

In this section, we adapt Algorithm 1 to enumerate satisfying assignments without duplicates. A simple idea would be to change Algorithm 1 to enumerate the gates of $\downarrow(g)$ without duplicates. Sadly, this would not suffice: imagine that we enumerate $S(g)$ for some \cup -gate g having two inputs g_1 and g_2 for which $S(g_1) \cap S(g_2) \neq \emptyset$, then if we consider g_1 and g_2 separately we will enumerate their common assignments twice. However, the crucial point is that $S(g_1) \cap$

$S(g_2) \neq \emptyset$ implies that g_1 and g_2 are in the same box, thanks to the following property of structured complete DNNFs:

LEMMA 5.1. *For any structured complete DNNF C , for any var-gate or \times -gate g of C and assignment S , if we have $S \in S(g)$, then the box of g is the (unique) least common ancestor of the boxes that contain the var-gates whose variables occur in S .*

PROOF SKETCH. This is because \top -gates are not allowed as input to any gate and \times -gates always use inputs from both subtrees. \square

This observation leads to the idea of *boxwise enumeration*, i.e., simultaneously considering a set of gates that are all in the same box, and enumerate *simultaneously* the assignments that they capture, without duplicates. This idea is reminiscent of evaluating a nondeterministic automaton on a word by determinizing the automaton on-the-fly, and it was already used in [4] in the case of words; we will extend it to trees. We will accordingly call *boxed set* a set Γ of gates that are all \cup -gates and that all belong in the same box. We write B_\cup for a box B to mean the \cup -gates of B .

Given a boxed set Γ in some box B , let us denote by $S(\Gamma)$ the set of assignments $\bigcup_{g \in \Gamma} S(g)$, which we want to enumerate without duplicates, and let us write $\downarrow(\Gamma) := \bigcup_{g \in \Gamma} \downarrow(g)$. Our enumeration will rely on a procedure $\text{BOX-ENUM}(\Gamma)$ that enumerates the boxes B' such that $\downarrow(\Gamma) \cap B' \neq \emptyset$ and produces for each such box B' the \cup -reachability relation between B' and Γ , i.e., the binary relation $R(B', \Gamma)$ describing which gates of B'_\cup have a path of \cup -gates to Γ . Formally, we define the \cup -reachability relation between any sets of gates G' and G as $R(G', G) := \{(g', g) \in G'_\cup \times G_\cup \mid g' \xrightarrow{\cup} g\}$. Pay attention to the fact that each call to BOX-ENUM returns the complete relation $R(B', B)$ for one of the boxes B' (i.e., we do not enumerate the pairs of $R(B', B)$), and the relation for each box B' is returned only once (i.e., there should not be duplicate boxes).

It is straightforward to show that $\text{BOX-ENUM}(\Gamma)$ can be implemented with delay $O(\text{depth}(C) \times \text{poly}(w))$, where w is the width of the circuit, by exploring the boxes from B (traversing only \cup -gates and at most one other gate per level) and maintaining the information $R(B', \Gamma)$ for the boxes B' that we visit. In the next section, we show how we can implement BOX-ENUM more efficiently.

The point of BOX-ENUM is the following easy consequence of Observation 4.1:

OBSERVATION 5.2. *For any boxed set Γ in any box, we have:*

$$S(\Gamma) = \bigcup_{\substack{R(B', \Gamma) \in \text{BOX-ENUM}(\Gamma) \\ \text{with } \mu(g') \in \{\text{var}, \times\}}} S(g')$$

Algorithm 2 Enumeration algorithm without duplicates

```

1: procedure ENUMS( $\Gamma$ )
2:    $B \leftarrow \text{BOX}(\Gamma)$ 
3:   for  $R(B', \Gamma) \in \text{BOX-ENUM}(\Gamma)$  do
4:      $G' \leftarrow \pi_1(W \circ R(B', \Gamma)) \triangleright$  project to first component
5:      $G_{\text{var}} \leftarrow \{h \in G' \mid \mu(h) = \text{var}\}$ 
6:     for  $g'' \in G_{\text{var}}$  do
7:       output  $(S_{\text{var}}(g''), \{g''\} \circ W \circ R(B', \Gamma))$ 
8:      $G_{\times} \leftarrow \{h \in G' \mid \mu(h) = \times\}$ 
9:      $\Gamma_L \leftarrow$  set of left inputs of  $G_{\times}$ 
10:    for  $(S_L, \Gamma'_L) \in \text{ENUM}_S(\Gamma_L)$  do
11:       $G'_{\times} \leftarrow$  set of gates of  $G_{\times}$  with left input in  $\Gamma'_L$ 
12:       $\Gamma_R \leftarrow$  set of right inputs of  $G'_{\times}$ 
13:      for  $(S_R, \Gamma'_R) \in \text{ENUM}_S(\Gamma_R)$  do
14:         $G''_{\times} \leftarrow$  set of gates of  $G'_{\times}$  with right input in  $\Gamma'_R$ 
15:         $\Gamma' \leftarrow G''_{\times} \circ W \circ R(B', \Gamma)$ 
16:        output  $(S_L \cup S_R, \Gamma')$ 

```

where W is the set of wires of C , where $W \circ R(B', \Gamma)$ denotes the composition of the two binary relations, and where the outermost union is without duplicates.

PROOF. To see why the equality holds, first observe that for each $R(B', \Gamma)$, the inner union goes over all gates of $\downarrow(\Gamma) \cap B'$. Indeed, the definition of the \cup -reachability relation ensures that $R(B', \Gamma)$ denotes the \cup -gates g' of B' such that $g' \rightsquigarrow g$ for some $g \in \Gamma$, so the inner union goes over all their inputs that are \times -gates and var-gates (and which must also be in box B'). Now, by definition of BOX-ENUM , the outermost union goes over all gates of $\downarrow(\Gamma)$. Thus, we can conclude thanks to Observation 4.1. The fact that the outermost union is disjoint follows immediately from Lemma 5.1. \square

Observation 5.2 suggests that we can perform enumeration without duplicates recursively, simply by re-applying the scheme on the inputs of the gates of $\downarrow(\Gamma) \cap B'$ to perform enumeration. The details are subtle, however, and this is why we designed BOX-ENUM to return more than just the set $\pi_1(R(B', \Gamma))$ of the \cup -gates of B' having a path of \cup -gates to Γ (with π_1 denoting projection to the first component): we will really need the complete relation $R(B', \Gamma)$ for the recursive calls, in order to avoid duplicate assignments across multiple \times -gates in $\downarrow(\Gamma) \cap B'$.

Our algorithm to enumerate $S(\Gamma)$ is presented as Algorithm 2. The semantics are changed a bit relative to Algorithm 1. Algorithm 2 takes as input a boxed set Γ , and the output is the enumeration of $S(\Gamma)$ without duplicates. Moreover, for each assignment S in this set, the algorithm also returns its *provenance relative to Γ* , i.e., the subset $\text{Prov}(S, \Gamma) := \{g \in \Gamma \mid S \in S(g)\}$, which again is used for the recursive calls.

THEOREM 5.3. *Given a structured complete DNNF C and given a boxed set Γ , we can enumerate $S(\Gamma)$ (without duplicates) with delay $O(|S| \times (\Delta + w^3))$, where S is the produced assignment, Δ is the delay of BOX-ENUM , and w is the width of C . Further, we correctly produce for each assignment S its provenance $\text{Prov}(S, \Gamma)$ relative to Γ .*

PROOF SKETCH. We have to show three things to establish correctness: (1) For every output (S, Γ') of the algorithm, we have $\Gamma' \subseteq \Gamma$ and $S \in S(g)$ for every $g \in \Gamma'$. (2) For every assignment $S \in S(g)$ with $g \in \Gamma$, we have some output (S, Γ') with $g \in \Gamma'$. (3) No assignment S is outputted twice.

Statements (1) and (2) can be shown by induction on the number of variable gates that contribute to the assignment. For the induction base case, one can verify that the assignments that only use one variable gate are correctly handled in Line 7. For the induction step, one can verify that the top-most \times -gate g_{\times} that is involved in computing an assignment is handled correctly in the lines 8 to 16.

Statement (3) follows from Lemma 5.1 and the fact that for each box $\text{BOX-ENUM}(\Gamma)$ returns at most one relation.

The proof for the runtime delay is similar to that of Proposition 4.2, with the difference that the local computations are more expensive as we need to compute relational compositions, e.g., $W \circ R(B', \Gamma)$. \square

6 ENUMERATING BOXES EFFICIENTLY

We have shown in the previous section how to enumerate the assignments captured by an assignment circuit without duplicates. However the delay depends on the delay of BOX-ENUM , which for the naive implementation we discussed before was linear in the depth of the circuit. This leads to a delay for the assignment enumeration that is also linear in the depth of the circuit. In this section, we show how to speed up BOX-ENUM and make its delay independent from the depth of the circuit, using a similar idea to *jump pointers* from [5]. As the delay added by Algorithm 2 only depends on the circuit width, this will establish our overall delay bound, which we state at the end of the section.

Interesting and Bidirectional Boxes. To speed up BOX-ENUM , we will need to perform some linear-time preprocessing on the input circuit, following the tree of boxes. Let us first give the required definitions. For each boxed set Γ , the set of *interesting boxes* for Γ is defined as $\{B' \mid B' \cap \downarrow(\Gamma) \neq \emptyset\}$: these are the boxes that $\text{BOX-ENUM}(\Gamma)$ must consider. We also define the set of *bidirectional boxes* for Γ as

$$\{B' \mid \downarrow(\Gamma) \text{ intersects boxes in both subtrees of } B'\}$$

These boxes are necessarily non-leaf boxes and have interesting boxes as descendants of their left and right child. Note that a bidirectional box may also be interesting.

One key idea to optimize BOX-ENUM is to “jump” from a box B to a bidirectional descendant box B' , skipping boxes on the path from B to B' that are neither interesting nor bidirectional. To do so, we need to precompute to which box we can jump from the boxed set Γ ; specifically, we need to know the *first interesting box* and the *first bidirectional box*. We thus define $\text{FIB}(g)$ (resp., $\text{FBB}(g)$) for a gate g as the first interesting box (resp., bidirectional box) seen in the *preorder traversal* of T where we first visit the box of g , then recursively traverse its left subtree, and last traverse its right subtree. We then extend these definitions to Γ by:

$$\text{FIB}(\Gamma) = \min_{g \in \Gamma} \text{FIB}(g) \quad (1)$$

$$\text{FBB}(\Gamma) = \text{LCA}(\{\text{FBB}(g) \mid g \in \Gamma\}) \quad (2)$$

where the min operator is according to the preorder traversal and LCA denotes the least common ancestor of a set of boxes in the tree of boxes.

Intuitively, the first bidirectional box will tell us where to jump, and the first interesting box will compensate the time spent visiting bidirectional boxes in the enumeration (as these boxes do not otherwise allow us to make progress). In addition to FIB and FBB, when jumping from a box B to a descendant box B' , we will also need to know the \cup -reachability relation $R(B', B)$ from the \cup -gates of B' to the \cup -gates of B , i.e., a special case of what we have defined in Section 5, where all \cup -gates of B appear. Hence, our preprocessing will consider each box B and compute the \cup -reachability relation $R(B', B)$ for every box B' to which we may jump from B .

Index Structure. We summarize all information that needs to be computed by formally defining the index structure:

Definition 6.1. The *index structure* $I(C)$ of a structured complete DNNF C consists of the following, for each box B :

- For each \cup -gate $g \in B$, the first interesting box $\text{FIB}(g)$ of Γ and the reachability relation $R(\text{FIB}(g), \text{BOX}(g))$
- For each boxed set $\Gamma \subseteq B$ with $1 \leq |\Gamma| \leq 2$, the first bidirectional box $\text{FBB}(\Gamma)$ of Γ , and the reachability relation $R(\text{FBB}(\Gamma), \text{BOX}(\Gamma))$
- Letting \mathcal{B} be the set of boxes of the form $\text{FIB}(g)$ or $\text{FBB}(g)$ for g a \cup -gate of B , letting $\mathcal{B}' = \{\text{LCA}(B_1, B_2) \mid B_1, B_2 \in \mathcal{B}\}$ (hence $\mathcal{B}' \supseteq \mathcal{B}$), we precompute \mathcal{B}' and the linear order implied by preorder traversal over \mathcal{B}' .

At first glance, the index seems weaker than what we need, because we will want to determine $\text{FIB}(\Gamma)$ and $\text{FBB}(\Gamma)$ for boxed sets Γ of arbitrary size. However, Equation (1) implies that $\text{FIB}(\Gamma)$ can be evaluated from $\text{FIB}(g)$ for individual gates $g \in \Gamma$, using the fact that we have precomputed min. The same is true for FBB and boxed sets Γ of size at most two, thanks to Equation (2) and the following elementary fact about least common ancestors:

Algorithm 3 Box enumeration

```

1: procedure BOX-ENUM( $\Gamma$ )
2:   B-ENUM( $\text{BOX}(\Gamma)$ ,  $\{(g, g) \mid g \in \Gamma\}$ )
3: procedure B-ENUM( $B, R$ )
4:    $B_1 \leftarrow \text{FIB}(\pi_1(R))$  ▷ first interesting box
5:    $R_1 \leftarrow R(B_1, B) \circ R$  ▷  $R(B_1, B)$  is in the index
6:   output  $R_1$  ▷ relation to  $B_1$ 
7:    $B_L \leftarrow \text{LEFT}(B_1); R_L \leftarrow R(B_L, B_1) \circ R_1$ 
8:   if  $R_L \neq \emptyset$  then B-ENUM( $B_L, R_L$ ) ▷ left subtree of  $B_1$ 
9:    $B_R \leftarrow \text{RIGHT}(B_1); R_R \leftarrow R(B_R, B_1) \circ R_1$ 
10:  if  $R_R \neq \emptyset$  then B-ENUM( $B_R, R_R$ ) ▷ right subtree of  $B_1$ 
11:   $B' \leftarrow B; B \leftarrow \text{FBB}(\pi_1(R))$  ▷ jump to the 1st bidir. box
12:  while  $B$  is defined and is a strict ancestor of  $B_1$  do
13:     $R \leftarrow R(B, B') \circ R$  ▷  $R(B, B')$  is in the index
14:     $B_R \leftarrow \text{RIGHT}(B); R_R \leftarrow R(B_R, B) \circ R$ 
15:    B-ENUM( $B_R, R_R$ ) ▷ right subtree of  $B$ 
16:     $B' \leftarrow \text{LEFT}(B); R \leftarrow R(B', B) \circ R$  ▷ left child
17:     $B \leftarrow \text{FBB}(\pi_1(R))$  ▷ 1st bidir. box

```

OBSERVATION 6.2. For any set \mathcal{B} of boxes, the least common ancestor of \mathcal{B} is the minimal box B in the preorder traversal that is a least common ancestor of two (possibly equal) boxes B_1 and B_2 of \mathcal{B} . Formally: $\text{LCA}(\mathcal{B}) = \min\{\text{LCA}(B_1, B_2) \mid B_i \in \mathcal{B}\}$

We now show how to compute the index structure:

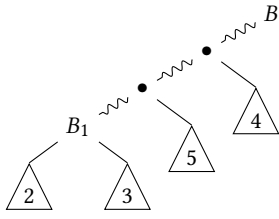
LEMMA 6.3. Given a structured complete DNNF circuit C with v -tree \mathcal{T} , we can compute $I(C)$ in time $O(|\mathcal{T}| \times w^4)$, where w is the width of C .

PROOF SKETCH. We compute the first interesting boxes $\text{FIB}(g)$ for all \cup -gates g by an easy bottom-up processing of C , and we do the same for all $\text{FBB}(\Gamma)$: here we rely on Observation 6.2 to know that, when we recursively compute $\text{FBB}(\Gamma')$ for a boxed set in a child box, we can do so from the $\text{FBB}(\Gamma'')$ for $\Gamma'' \subseteq \Gamma'$ with $|\Gamma''| \leq 2$.

We compute the reachability relations by considering all boxes B bottom-up and computing $R(B', B)$ for all descendant boxes B' where this is required: we show that we can always do so from the relation $R(B'', B)$ for B'' the child of B in the direction of B' , which is easy to compute from the wires, and from the relation $R(B', B'')$ which we argue must have been computed when considering B'' . The complexity is $O(w^3)$, which is bounded by the complexity of computing $R(B', B'') \circ R(B'', B)$ with the naïve join algorithm. \square

Efficient Implementation of BOX-ENUM. We now present the algorithm for efficient enumeration of $\text{BOX-ENUM}(\Gamma)$:

LEMMA 6.4. Given a structured complete DNNF C and the index structure $I(C)$, we can implement BOX-ENUM with delay $O(w^3)$, where w is the width of C .



B_1 is the first interesting box and \bullet indicates bidirectional boxes. All interesting boxes of the subtrees indicated by triangles are visited in the indicated order.

Figure 1: Sketch of the box tree of assignment circuits

PROOF SKETCH. The algorithm $\text{BOX-ENUM}(\Gamma)$ to perform the enumeration for an arbitrary boxed set Γ is in Algorithm 3. Each call of the recursive algorithm in Line 3 receives the relation $R(B, \Gamma)$ for some box B called the *current box*, and it is expected to output the relations $R(B', \Gamma)$ for interesting boxes B' in the subtree of B .

In Figure 1, we sketched the order in which boxes are enumerated, starting with the first interesting box B_1 (output in Line 6), then all descendants of B_1 (recursive calls in lines 8 and 10), and then right children of bidirectional boxes on the path to B_1 (recursive call in the loop). By the definition of bidirectional boxes, this enumerates all interesting boxes.

It is easy to show that the delay is $O(w^3)$, which stems from the computation of relational composition using naïve joins; the main subtlety is that we need to modify Algorithm 3 slightly to ensure that each last recursive call is tail-recursive, to avoid delays when unwinding the recursion stack. \square

Putting it Together. We now state our main result about the complexity of enumerating the set of assignments captured by a boxed set in a complete structured DNNF circuit. Before we do so, however, we point out a small optimization trick that allows us to bring the complexity in the width w of the circuit from $O(w^3)$ down to $O(w^\omega)$, where $2 \leq \omega \leq 3$ is an exponent for Boolean matrix multiplication, i.e., a constant such that the product of two r -by- r Boolean matrices can be computed in time $O(r^\omega)$. The best possible value for ω is an open question, with the best known bound being $\omega < 2.3728639$, see [24]. Observe that, in Theorem 5.3, Lemma 6.3 and Lemma 6.4, the complexity bottleneck is to compute expressions of the form $R \circ R'$ for relations R and R' over sets of size $\leq w$, with all other operations having complexity $O(w^2)$ at most. We have used the naïve join algorithm to bound this by $O(w^3)$, but we can instead represent these relations as Boolean matrices and use any matrix multiplication algorithm to compute the product in $O(w^\omega)$.

This leads to our final enumeration result on set circuits:

THEOREM 6.5. *Let ω be an exponent for the Boolean matrix multiplication problem. Given any complete structured*

DNNF C of width w with its v -tree \mathcal{T} and structuring function, we can preprocess C in $O(|\mathcal{T}| \times w^{\omega+1})$ to be able, given any boxed set Γ , to enumerate the assignments of $S(\Gamma)$ with delay $O(|S| \times w^\omega)$ for each produced assignment S .

PROOF. We preprocess C using Lemma 6.3, and we then perform the enumeration using Theorem 5.3 with the efficient implementation of BOX-ENUM given in Lemma 6.4; modifying the algorithms to perform matrix multiplication in time $O(w^\omega)$ instead of using the naïve join algorithm. \square

7 UPDATES AND BALANCING

We have shown our circuit construction result (Lemma 3.7) and enumeration result (Theorem 6.5). We will put them together to show our enumeration results for automata and queries over trees. However, before this we need to explain how we can handle updates efficiently, i.e., how we can recompute the circuit (Lemma 3.7) and the index of the enumeration structure (Lemma 6.3) whenever the underlying tree is modified.

The crucial insight is that the circuit in Lemma 3.7 is computed *bottom-up* on the input tree T , and the precomputation in Lemma 6.3 is also performed bottom-up on the tree of boxes whose structure is isomorphic to T . Hence, whenever T is updated at some node n , we can modify the circuit and the index accordingly by recomputing everything bottom-up starting at node n . The complexity of this process will be linear in the *height* of T . This is why, in this section, we will want to work on trees that are *balanced*, i.e., whose height is logarithmic in their size: this is what will guarantee that updates can be handled in logarithmic time.

As trees are in general not balanced, our technique will be to code the input tree as binary balanced trees. We will also use this as a way to allow arbitrary unranked trees as input (not just binary trees), which is more convenient because binary trees do not behave well under edit operations (e.g., adding or deleting a single leaf). Specifically, given the input unranked Λ -tree T and Λ, X -TVA A running on unranked trees, we will encode T to a balanced binary tree T' over a different alphabet Λ' , and we will translate the automaton A in polynomial time to a Λ', X -TVA A' , while ensuring that A and A' have the same satisfying assignments. Our balanced binary tree formalism will further ensure that, whenever an update is performed on T , we can update T' (and keep it balanced) to reflect the change, and we can efficiently update the circuit C and the index for this update.

In this section, we first present our model for the input unranked tree T and the edit operations that we allow on it. We then present our formalism for the automaton A , which runs on unranked trees, and the notion of *tree hollowings* to describe which kinds of updates can happen on the balanced

binary tree T' : intuitively, whenever we apply an edit operation on T , then we will be able to update T' in logarithmic time with a tree hollowing; and we will show that we can update the circuit and index in the same time bound. Then, we formalize the notion of encoding unranked trees to binary trees and of faithfully translating automata, and we state the result from [30] which explains how this can be efficiently performed. This allows us state our main enumeration results for automata and queries in the next section.

Edit Operations on Unranked Trees. We first present the language of edit operations that we allow on the *unranked* tree which is the input to our enumeration scheme. Fixing a set X of variables, we will define an X -valuation of an unranked Λ tree T as a function v mapping each node n of T to a set $v(n) \subseteq X$. Note that a valuation of an unranked tree annotates all its nodes, not just the leaf nodes.

The update operations that we allow on unranked trees are leaf insertions, leaf deletions, and relabelings. More precisely:

Definition 7.1. Given an unranked Λ -tree T , a node n of T and a label $l \in \Lambda$, we allow the following edit operations:

- **DELETE**(n): remove n from T (only if n is a leaf)
- **INSERT**(n, l): insert an l -node as first child of n
- **INSERT_R**(n, l): insert an l -node as right sibling of n
- **RELABEL**(n, l): change the label of n to l

For any edit operation τ , we call $\tau(T)$ the resulting tree.

Automata on Unranked Trees. Following our use of unranked binary trees, we must also extend the definition of TVAs to work on unranked trees. Our automaton model for unranked trees are stepwise tree automata extended with variables. Stepwise tree automata were introduced in [18]. We use the formalism from [27], as it is closer to our tree model. A Λ, X -TVA on unranked Λ -trees for the variable set X is a tuple $A = (Q, \iota, \delta, F)$, where Q are the states, $\iota \subseteq \Lambda \times 2^X \times Q$ is the *initial relation*, $\delta \subseteq Q \times Q \times Q$ is the *transition relation*, and $F \subseteq Q$ are the *final states*. Given a Λ -tree T and an X -valuation v of T , we again define a *run* of A on T as a function $\rho: T \rightarrow Q$ satisfying some conditions. However, compared to binary trees, the use of ι and δ is different: ι assigns a set of possible initial states to every state (not only to leaves). For inner nodes, δ then consumes the states of children state by state, just as a word automaton reads its input symbol by symbol. The assigned state of a node is the state after having read the states of all children.

To formalize this, let us see δ as a function $\delta: 2^Q \times Q \rightarrow 2^Q$ by setting $\delta(Q', q') = \cup_{q \in Q'} \{q'' \mid (q, q', q'') \in \delta\}$, and let us inductively define the function $\delta^*: 2^Q \times Q^* \rightarrow 2^Q$ by $\delta^*(Q', \epsilon) := Q'$ and $\delta^*(Q', q_1 \dots q_n) := \delta^*(\delta(Q', q_1), q_2 \dots q_n)$. A run $\rho: T \rightarrow Q$ of A on T must then satisfy the following: For every node n with children n_1, \dots, n_m , we have

$\rho(n) \in \delta^*(\iota(\lambda(n), v(n)), \rho(n_1)\rho(n_2) \dots \rho(n_m))$. Note that TVAs on unranked trees now read annotations at all nodes (and not only at leaves), as per the definition of valuations.

The definitions of accepting runs, satisfying valuations and assignments on a Λ, X -TVA on unranked trees is the same as for binary trees: note that assignments now consist of singletons of the form $\langle Z : n \rangle$ where n can be any node of T , not just a leaf. Incidentally, observe that any Λ, X -TVA on binary trees can clearly be converted to a Λ, X -TVA on unranked trees which accepts exactly the same trees.

Tree Hollowings. We now formalize our language of updates on *binary trees*, which we call tree hollowings. Intuitively, when performing an update on the input unranked tree T , we want to reflect it on its balanced binary representation T' . Our encoding scheme will ensure that the edit operations of Definition 7.1 can be performed as *tree hollowings* whose *trunks* have logarithmic size. Here are the relevant definitions:

Definition 7.2. Let T' be a binary Λ' -tree. Two nodes of T' are *incomparable* if neither is a descendant of the other (in particular, they must be different). An *antichain* in T' is a set of nodes in T' that are pairwise incomparable.

A *tree hollowing* $H = (T'', \eta)$ of T' consists of a $(\Lambda' \cup \{\square\})$ -tree T'' called the *trunk*, in which all internal nodes must have a label in Λ' , and an injective function η from the leaves of T'' with label \square to T' , such that the image of η is an antichain of T' . The *result* H^{res} of the tree hollowing H is the Λ' -tree obtained by taking T'' and replacing each \square -labeled leaf n of T'' by the subtree of T' rooted at $\eta(n)$.

Intuitively, a tree hollowing describes how to build a new tree while reusing disjoint subtrees of the original tree. We will require that each update to the original tree T should be translatable in logarithmic time to a tree hollowing of the binary balanced representation T' —so in particular the trunk will have logarithmic size, even though the result of the hollowing will not.

The reason why hollowings are a good update language is because our constructions are strictly bottom up. Thus, given a binary Λ' -tree T' for which we have computed an assignment circuit C for some Λ', X -TVA A , and given the index $I(C)$, we can follow a tree hollowing $H = (T'', \eta)$ of T' to update the circuit to an assignment circuit C^{res} of A on H^{res} and to update the index to $I(C^{\text{res}})$. Formally:

LEMMA 7.3. *Given any Λ' -tree T' and Λ', X -TVA A' with state space Q' , given an assignment circuit C of A' on T' which is a structured complete DNNF of width $|Q'|$ and given the index structure $I(C)$, given any tree hollowing $H = (T'', \eta)$ of T' , we can compute in time $O(|T''| \times |Q'|^{\omega+1})$ a circuit C^{res} and the index structure $I(C^{\text{res}})$ such that C^{res} is a structured*

complete DNNF of width $|Q'|$ which is an assignment circuit of A' on T'^{res} , and $I(C^{\text{res}})$ is the index structure for C^{res} .

PROOF. We use the circuit construction from Lemma 3.7 to compute a box B_n for each node n in T'' that is not labeled \square (total time $O(|T''| \times |A'|)$) and we thus compute C^{res} , which has size $O(|T''| \times |A'|)$. Afterwards we use the computation from the proof of Lemma 6.3 (using efficient matrix multiplication as explained in the proof of Theorem 6.5) to compute the index structure for the new boxes in total time $O(|T'| \times w^{\omega+1})$. \square

Encoding Unranked Trees in Balanced Binary Trees. We will now explain how we can encode an unranked tree T into a balanced binary tree T' on which update operations can be reflected, i.e., every update on T in the language of Definition 7.1 translates to an update of T' that can be represented by a tree hollowing whose trunk has logarithmic size. It will be more convenient to formalize the encoding by describing the *decoding function* which decodes a binary tree to an unranked tree. Specifically, a *tree encoding scheme* for the tree alphabet Λ consists of a tree alphabet Λ' and of a function ω defined on some subset of the binary Λ' -trees (called the *well-formed trees*) and which decodes any such binary Λ' -tree T' to an unranked Λ -tree T and to a bijection $\varphi_{T'}$ from the leaves of T' to the nodes of T . We require ω to be surjective, i.e., every unranked Λ -tree has some preimage in ω . Now, given a set X of variables, a Λ, X -TVA A and a binary Λ, X' -TVA A' , we say that ω is A, A' -faithful if for any Λ -tree T , for any preimage T' of T in ω , letting $\varphi_{T'}$ be the bijection from the leaves of T' to the nodes of T , for any X -valuation ν of T , we have that A accepts T under ν iff A' accepts T' under the valuation $\nu \circ \varphi_{T'}$.

Our tree encoding method can then be formalized as the following result, which easily follows from [30]:

LEMMA 7.4. *For any tree alphabet Λ and set X of variables, there is an encoding scheme ω for Λ such that:*

- *The encoding is linear-time computable, i.e., given any unranked Λ -tree T , we can compute in linear time some Λ' -tree T' with $\omega(T') = T$, as well as the bijection $\varphi_{T'}$.*
- *The encoded trees have logarithmic height, i.e., each well-formed Λ' -tree T' has height in $O(\log |T'|)$.*
- *We can efficiently perform updates, i.e., given any binary Λ -tree T , preimage T' of T , and update τ on T in the language of Definition 7.1, we can compute in time $O(\log |T|)$ a tree hollowing $H = (T'', \eta)$ of T' such that $\omega(H^{\text{res}}) = \tau(\omega(T'))$.*
- *We can efficiently translate automata, i.e., given any unranked Λ, X -TVA A with state space Q , we can build in time $O(|Q|^6)$ a binary Λ', X' -TVA A' with $O(|Q|^4)$ states and $O(|Q|^6)$ transitions such that ω is A, A' -faithful. Furthermore, A' has a single accepting state.*

PROOF SKETCH. The idea is to convert unranked trees to binary trees that represent terms in the free forest algebra. Intuitively, a forest algebra term T' is a binary Λ' -tree that describes an unranked Λ -tree: the leaves of T' correspond to the nodes of T , and each internal node corresponds to a forest computed from its children by one of two operations: forest concatenation or context application. In [30, Section 3], it is described how we can convert an unranked tree T in linear time to a balanced term T' that represents T , and how updates on T can be reflected on T' . Although this is not stated explicitly in [30], the resulting updates on T' can be described as tree hollowings of logarithmic size. We can also easily show that an automaton A on unranked Λ -trees can be converted in PTIME to an automaton A' on binary Λ' -trees. \square

8 MAIN RESULTS

We can now present our main results by combining our results about tree balancing and hollowing updates (Lemma 7.4 and Lemma 7.3) with our circuit construction and enumeration results (Lemma 3.7 and Theorem 6.5). The first phrasing of our main result deals with the enumeration of the satisfying assignments to TVAs under updates, while ensuring the right complexity bounds, and in particular guaranteeing tractable combined complexity:

THEOREM 8.1. *Let ω be an exponent for the Boolean matrix multiplication problem. Given an unranked Λ, X -TVA A with state space Q and an unranked Λ -tree T , we can enumerate the satisfying assignments of A on T with preprocessing time $O(|T| \times |Q|^{4(\omega+1)})$, update time $O(\log(|T|) \times |Q|^{4(\omega+1)})$, and delay $O(|Q|^{4\omega} \times |S|)$, where S is the produced assignment.*

Thanks to Theorem 8.1, we can now obtain as a corollary our main result on enumeration for monadic second-order logic (MSO). Recall that MSO is a logic that extends first-order logic on Λ -trees defined on a signature featuring the edge relation of the tree, the order relation among siblings, and unary predicates for the node labels. MSO extends first-order logic by adding the ability to quantify over sets. Given a Λ -MSO formula $\Phi(X_1, \dots, X_n)$ and a Λ -tree T where all free variables are second-order, letting $X = \{X_1, \dots, X_n\}$, the *satisfying valuations* of Φ on T are the X -valuations ν of T such that T satisfies the Boolean Λ -MSO formula Φ where the predicates X_1, \dots, X_n are interpreted according to ν . Similarly to what we did for TVAs, we represent any X -valuation ν as an *assignment* $\alpha(\nu) := \{\langle Z : n \rangle \mid Z \in \nu(n)\}$, and the *satisfying assignments* of Φ on T are the images by α of the satisfying valuations.

Our main result for MSO enumeration is then the following; it improves on the earlier results for MSO enumeration

on trees under updates [4, 26, 30]. Note that unlike Theorem 8.1, this result only considers *data complexity*, i.e., complexity in the input tree (and in the produced assignments), assuming that the MSO query is fixed.

COROLLARY 8.2. *For any fixed tree alphabet Λ , for any fixed Λ -MSO query Φ with free second-order variables, given an unranked Λ -tree T , after preprocessing T in linear time, we can enumerate the satisfying assignments of Φ with delay linear in each produced assignment, and we can handle updates to T in logarithmic time in T .*

PROOF. Let X be the variables of Φ . We see Φ as a Boolean query on the tree signature $\Lambda \cup 2^X$, and rewrite Φ to a tree automaton on $\Lambda \cup 2^X$ using the well-known result by Thatcher and Wright [37]: see also [3, Appendix E.1]. The automaton can equivalently be seen as an unranked Λ, X -TVA, so we can enumerate its assignments on T using Theorem 8.1. \square

In the common case of an MSO formula $\Phi(x_1, \dots, x_n)$ with free *first-order* variables, Corollary 8.2 implies that we can enumerate the satisfying assignments on a tree T , with the delay being constant, i.e., independent from T .

COROLLARY 8.3. *For any fixed tree alphabet Λ , for any fixed Λ -MSO query Φ with free first-order variables, given an unranked Λ -tree T , after preprocessing T in linear time, we can enumerate the satisfying assignments of Φ with constant delay, and we can handle updates to T in logarithmic time in T .*

Note that enumerating satisfying assignments also allows us to enumerate the *answer tuples* $(a_1, \dots, a_n) \in T^n$ such that T satisfies $\Phi(a_1, \dots, a_n)$, as we can simply translate each satisfying assignment in linear time to an answer tuple.

Results on Words. We conclude the section by presenting consequences of our results for *words*. As words are a special case of trees, our results on trees imply results for the enumeration on words of the satisfying assignments of word automata, with the ability to efficiently handle updates to the underlying word. This can be used in the context of *document spanners* [22] for information extraction, allowing us to efficiently enumerate the matches of a document spanner represented as a sequential extended VA [23], and to update the enumeration structure when the word changes.

Given a set Λ of labels, we call a Λ -word a finite sequence w of letters from Λ . Given a variable set X , an X -valuation of w is a function $v : \{1, \dots, |w|\} \rightarrow 2^X$, and the corresponding X -assignment $\alpha(v)$ is the set $\{\{Z : n\} \mid Z \in v(n)\}$. On words we allow the usual local edits: (1) inserting a character, (2) deleting a character, and (3) replacing a character.

We next present the formalism that we use to write queries on words, which is analogous to *extended sequential variable automata* [23]. A *word variable automaton* WVA on Λ -words for variable set X (or Λ, X -WVA) is a tuple $A = (Q, \delta, I, F)$,

where Q is a set of *states*, I is the set of initial states, $\delta \subseteq Q \times \Lambda \times 2^X \times Q$ is the *transition relation*, and $F \subseteq Q$ is the set of *final states*. Note that, like an unranked TVA, a WVA can read variables at any position.

When working with WVAs rather than TVAs, the translation to balanced binary trees can be done with a better complexity. Specifically, the following is obvious from the details of the construction of A' in Lemma 7.4.

COROLLARY 8.4. *Lemma 7.4 holds for words instead of trees and a WVA A instead of an unranked TVA as input, with the difference that the binary tree automaton A' can be constructed in time $O(|Q|^3)$, has $O(|Q|^2)$ states and $O(|Q|^3)$ transitions, where Q is the state space of A .*

We can conclude the following theorem.

THEOREM 8.5. *Given a Λ, X -WVA A with state space Q and Λ -word w , we can enumerate the satisfying assignments with preprocessing time $O(|w| \times |Q|^{2(\omega+1)})$, update time $O(\log(|w|) \times |Q|^{2(\omega+1)})$, and delay $O(|Q|^{2\omega} \times |v|)$, for v the current valuation.*

In terms of document spanners, this theorem is the analogue of the result of [5] which showed that we can efficiently enumerate the results of an extended sequential non-deterministic VA on an input word, and it extends this result to handle updates to the word in logarithmic time. However, in exchange for the support for updates, the complexity in the automaton is less favorable, with a higher polynomial degree (due to the need to balance the word); and the memory usage of Theorem 8.5 is not constant like it was in [5]. We also note that our results in this paper do *not* recapture the results for *non-extended* VAs from [5]: we believe that our techniques here would extend to such automata, but that it would require some changes to the tree automaton model.

9 LOWER BOUND

In this section, we will show that the logarithmic update time of Section 8 is optimal up to a doubly logarithmic factor. In fact, we will show that either the update time or the enumeration delay of any MSO enumeration algorithm on trees must be $\Omega\left(\frac{\log(n)}{\log \log(n)}\right)$. In particular, in contrast to the settings in [11, 12], there is no algorithm with constant update time for MSO queries on trees even if we allow the enumeration delay to be slightly superconstant.

To show our lower bound, we rely heavily on a result from [1], so let us introduce some notation from there: consider a tree T on n nodes in which some of the nodes are *marked* while the others are *unmarked*. For every node v of T , let $\pi(v)$ denote the path from v to the root of T . The *existential marked ancestor query* is, given a node v , to decide if v has a marked ancestor, i.e., there is a marked node on

$\pi(v)$. An *update* in the marked ancestor problem is an operation that marks or unmarks a node of T . An algorithm for the marked ancestor problem is an algorithm that maintains a data structure that allows updates and marked ancestor queries. A main result of [1] is then the following:

THEOREM 9.1. *Consider an algorithm to solve the marked ancestor problem and let t_u be a bound on the time to handle an update and t_q be a bound on the time to answer a query. Then we have $t_q = \Omega\left(\frac{\log(n)}{\log(t_u \log(n))}\right)$.*

We remark that Theorem 9.1 is unconditional and holds for the standard model of unit cost RAMs with logarithmic word size; for different word sizes the result is slightly different. We also remark that Theorem 9.1 even stays true if all runtimes are assumed to be amortized, so in a non-worst case setting. Finally, Theorem 9.1 makes no assumption whatsoever about the runtime of a potential preprocessing phase. So even with generous runtime in the preprocessing, the result holds.

We now show that we get a lower bound for MSO query enumeration easily from Theorem 9.1.

THEOREM 9.2. *There is an MSO query Φ on trees such that any enumeration algorithm for Φ under relabelings with update time \hat{t}_u and enumeration delay \hat{t}_e has*

$$\max(\hat{t}_u, \hat{t}_e) \geq \Omega\left(\frac{\log(n)}{\log \log(n)}\right).$$

PROOF SKETCH. We reduce from the existential marked ancestor problem. To this end, we consider trees in which nodes can have three labels: marked, unmarked, or special. Fix the query $\Phi(x)$ that selects all special nodes that have a marked ancestor: we can easily write it in MSO.

We give an algorithm for the marked ancestor problem: We start with the input tree T , i.e., a tree without any node marked special. To answer the marked ancestor query for a node v , we label v as a special node, enumerate the answer to Φ , and make v non-special again. Finally we return 'yes' if and only if we enumerated any answer to Φ .

To see that this algorithm is correct, observe that v is the only special node in T when evaluating Φ . So either we enumerate v or nothing depending on if v has a marked ancestor, and the answer we give is correct. Now, Theorem 9.1 applies to the marked ancestor queries, i.e., to $2\hat{t}_u + \hat{t}_e$, from which we can mathematically derive our claimed bound. \square

From Theorem 9.2 we get easily that the update time in Theorem 8.1 is optimal up to a doubly logarithmic factor even if we allow the enumeration delay to be close to logarithmic.

COROLLARY 9.3. *There is an MSO query Φ on trees such that any enumeration algorithm for Φ under relabelings with enumeration delay $o\left(\frac{\log(n)}{\log \log(n)}\right)$ has update time $\Omega\left(\frac{\log(n)}{\log \log(n)}\right)$.*

As Theorem 9.1 from [1], our Theorem 9.2 also works for amortized time and with arbitrary preprocessing. Note also that the query Φ in the proof does not use the full power of MSO but can in fact be expressed in first-order logic with transitive closure, so our lower bound already holds for that fragment. Note that it is not clear if we can show an analogous result for FO without transitive closure: in particular, we cannot apply the results of [11] about enumeration for FO on bounded-degree structures, because the lower bound in [1] uses trees of unbounded degree.

10 CONCLUSION

We have shown an efficient algorithm to enumerate the assignments of MSO formulae on trees under updates (relabeling, leaf insertions, leaf deletions), with linear preprocessing in the input tree, linear delay in each produced assignment (so constant if all free variables are first-order), and with logarithmic update time (in the tree). Our work is the first to match the bounds of the original results on MSO enumeration on trees without updates, while allowing general updates on trees (refer back to Table 1 for a comparison). Our algorithm is also the first to be tractable in combined complexity when the query is given as a (generally nondeterministic) tree variable automaton, with the preprocessing phase, enumeration delay, and update time being polynomial in the automaton; this extends our previous results on words [5] to trees and to efficient updates.

Our results leave several directions open for future work. One question is to improve the complexity in terms of the automaton, i.e., lowering the polynomial degree, as we have shown to be possible in the case of queries on words. A related question would be to perform enumeration for a more concise automaton model, in particular allowing the automaton to represent possible sets of captured variables more concisely, like non-extended VAs [5, 23]: we believe that this should be possible. We could also aim for more expressive automata models, e.g., alternating automata or two-way automata; or other query languages on trees, e.g., tree pattern queries. Another open question is the support for more expressive update operations. In the case of words, it would be natural to support bulk updates, i.e., moving a part of the text to a different place (see the conclusion of [5]). We believe that our techniques could adapt for such updates on words. As for trees, we currently do not know how to handle updates that split a subtree or attach a subtree.

One issue that we have not explored in the current paper is memory usage. Constant memory usage was achieved in [5] (for nondeterministic sequential VAs) and in [5, 8, 25]

(for MSO queries with free first-order variables), but it does not hold for our results (the memory usage may be linear in the circuit). We do not know if we can achieve constant memory.

Finally, there is a gap of $\log \log(n)$ between our upper bound and the lower bound, which it would be interesting to close. Note that the marked ancestor problem in fact has an algorithm with update complexity $O(\log(n)/\log \log(n))$, so we cannot hope to close the gap by improving the lower bound in Theorem 9.1. It may be the case that better lower bounds can be shown for our enumeration problem, but, going beyond $\Omega(\log(n)/\log \log(n))$ is generally considered very challenging. Indeed, the only paper that we are aware of that achieves a lower bound of $\Omega(\log(n))$ for any dynamic problem is [33], but it does not imply a lower bound in our setting because they allow more powerful updates than we do. Alternatively, it may be possible to improve our update complexity to $O(\log(n)/\log \log(n))$, e.g., by adapting the techniques of [1], but we have not been able to do so.

It would also be interesting to see what happens if we allow more generous enumeration delay, say $O(\log(n))$. Can we get an algorithm with updates that is faster than the lower bound of Corollary 9.3 in that case, maybe even constant? This might be an interesting trade-off in applications where updates are much more frequent than enumeration queries.

ACKNOWLEDGMENTS

Bourhis was partially founded by the DeLTA French ANR project (ANR-16-CE40-0007). Niewerth was supported by the Deutsche Forschungsgemeinschaft (MA 4938/4-1).

REFERENCES

- [1] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. 1998. Marked ancestor problems. In *FOCS*.
- [2] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. 2017. A Circuit-based approach to efficient enumeration. In *ICALP*.
- [3] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. 2019. A Circuit-based approach to efficient enumeration. Extended version with proofs. <https://arxiv.org/abs/1702.05589>.
- [4] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. 2018. Enumeration on trees under relabelings. In *ICDT*.
- [5] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*.
- [6] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Enumeration on trees with tractable combined complexity and efficient updates. Extended version with proofs. <https://arxiv.org/abs/1812.09519>.
- [7] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2015. Provenance circuits for trees and treelike instances. In *ICALP*.
- [8] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*.
- [9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic conjunctive queries and constant delay enumeration. In *CSL*.
- [10] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental validation of XML documents. *TODS* 29, 4 (2004).
- [11] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering FO+MOD queries under updates on bounded degree databases. In *ICDT*.
- [12] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering conjunctive queries under updates. In *PODS*.
- [13] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2018. Answering UCQs under updates and in the presence of integrity constraints. In *ICDT*.
- [14] Hans L. Bodlaender and Torben Hagerup. 1998. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.* 27, 6 (1998).
- [15] M. Bojańczyk. 2010. Algebra for trees. In *Handbook of Automata Theory*. European Mathematical Society Publishing.
- [16] M. Bojańczyk and I. Walukiewicz. 2007. Forest algebras. In *Automata and Logic: History and Perspectives*. Amsterdam University Press.
- [17] Florent Capelli and Stefan Mengel. 2019. Tractable QBF by knowledge compilation. In *STACS*.
- [18] J. Carme, J. Niehren, and M. Tommasi. 2004. Querying unranked trees with stepwise tree automata. In *RTA*.
- [19] Nofar Carmeli and Markus Kröll. 2018. Enumeration complexity of conjunctive queries with functional dependencies. In *ICDT*.
- [20] Adnan Darwiche. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics* 11, 1-2 (2001).
- [21] Arnaud Durand and Etienne Grandjean. 2007. First-order queries on structures of bounded degree are computable with constant delay. *TOCL* 8, 4 (2007).
- [22] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummen. 2015. Document spanners: A formal approach to information extraction. *JACM* 62, 2 (2015).
- [23] Fernando Florenzano, Cristian Riveros, Martin Ugarte, Stijn Vansummen, and Domagoj Vrgoc. 2018. Constant delay algorithms for regular document spanners. In *PODS*.
- [24] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *ISSAC*.
- [25] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of monadic second-order queries on trees. *TOCL* 14, 4 (2013).
- [26] Katja Losemann and Wim Martens. 2014. MSO queries on trees: enumerating answers under updates. In *CSL-LICS*.
- [27] W. Martens and J. Niehren. 2007. On the Minimization of XML Schemas and tree automata for unranked trees. *JCSS* 73, 4 (2007).
- [28] Wim Martens and Tina Trautner. 2018. Evaluation and enumeration problems for regular path queries. In *ICDT*.
- [29] Albert R Meyer. 1975. Weak monadic second order theory of successor is not elementary-recursive. In *Logic Colloquium*.
- [30] Matthias Niewerth. 2018. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*.
- [31] Matthias Niewerth and Luc Segoufin. 2018. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*.
- [32] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *SIGMOD Record* 45, 2 (2016).
- [33] Mihai Patrascu and Erik D. Demaine. 2006. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.* 35, 4 (2006).
- [34] Knot Pipatsrisawat and Adnan Darwiche. 2008. New compilation languages based on structured decomposability. In *AAAI*.
- [35] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. 2018. Enumeration for FO queries over nowhere dense graphs. In *PODS*.
- [36] Luc Segoufin. 2014. A glimpse on constant delay enumeration (Invited talk). In *STACS*.
- [37] James W. Thatcher and Jesse B. Wright. 1968. Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Systems Theory* 2, 1 (1968).

A PROOFS FOR SECTION 2 (PRELIMINARIES)

LEMMA 2.1. *Given a Λ, X -TVA A , we can compute in linear time a Λ, X -TVA A' which is homogenized and equivalent to A .*

PROOF. Let $A = (Q, \iota, \delta, F)$. Intuitively, we build A' as a product of A with an automaton with two states that remember whether some non-empty annotation has been seen. Formally, let A' be $(Q', \iota', \delta', F')$ where $Q' := Q \times \{0, 1\}$, where $F' := F \times \{0, 1\}$, where $\iota' := \{(l, \emptyset, (q, 0)) \mid (l, \emptyset, q) \in \iota\} \cup \{(l, \mathcal{Y}, (q, 1)) \mid (l, \mathcal{Y}, q) \in \iota, \mathcal{Y} \neq \emptyset\}$, and where $\delta' := \{((q_1, i_1), (q_2, i_2), (q, i_1 \vee i_2)) \mid (q_1, q_2, q) \in \delta, (i_1, i_2) \in \{0, 1\}^2\}$. We can clearly construct A' from A in linear time. We then modify A' to *trim* it, i.e., removing states that cannot be reached by any run, which is clearly doable in linear time by a simple reachability test. It is clear from the inductive definition of 0- and 1-states that A' is homogenized, i.e., each (q, i) for $q \in Q$ and $i \in \{0, 1\}$ is an i -state and it is not an $(1 - i)$ -state. Further, it is immediate by induction that, for any Λ -tree T and X -valuation ν of T , there is a bijection between the runs of A on T under ν and the runs of A' on T under ν , that maps a run $\rho : T \rightarrow Q$ to $\rho' : T \rightarrow Q'$ defined by keeping the first component as-is and filling the second component on each node n by 1 or 0 depending on whether some descendant of n has a non-empty annotation or not. \square

B PROOFS FOR SECTION 3 (BUILDING ASSIGNMENT CIRCUITS)

LEMMA 3.7. *Given any binary Λ -tree T and homogenized Λ, X -TVA $A = (Q, \iota, \delta, F)$, we can construct in time $O(|T| \times |A|)$ a structured complete DNNF C which is an assignment circuit of A and T , a ν -tree \mathcal{T} , and a structuring function from C to \mathcal{T} , such that C has width $|Q|$ and depth $O(\text{height}(T))$.*

PROOF. We construct \mathcal{T} by taking T , removing all node labels, and labeling each leaf node n by the set of singletons $\langle X : n \rangle$: note that \mathcal{T} is indeed a ν -tree for the set of variables $C_{\text{var}} = \{\langle Z : n \rangle \mid Z \in X, n \in T\}$ of C given by the definition of assignment circuits.

We now present the construction of C bottom up. We first describe the case of a leaf node n of T with label $l \in \Lambda$. In this case, we construct the box B_n for n as follows:

- For every 0-state q of A , we set $\gamma(n, q)$ to be a \top -gate if $(l, \emptyset, q) \in \iota$, and a \perp -gate otherwise.
- For every 1-state q of A with no tuples of the form (l, \mathcal{Y}, q) in ι , we set $\gamma(n, q)$ to be a \perp -gate.
- For every 1-state q of A with at least one tuple of the form (l, \mathcal{Y}, q) , we set $\gamma(n, q)$ to be a \cup -gate having as inputs one variable gate labeled by $\langle \mathcal{Y} : n \rangle$ for each $\mathcal{Y} \subseteq X$ such that $(l, \mathcal{Y}, q) \in \iota$. Note that \mathcal{Y} is then nonempty because q is a 1-state.

It is clear that B_n has at most $|Q|$ \cup -gates and that all restrictions for structured complete DNNFs are met.

For an inner node n of T with label l and child nodes n_1 and n_2 , we construct the box B_n as follows. First, for every 0-state of A , we set $\gamma(n, q)$ to be a \top -gate if and only if there are states q_1 and q_2 in A such that $(q_1, q_2, q) \in \delta_l$ and $\gamma(n_1, q_1)$ and $\gamma(n_2, q_2)$ are both \top -gates. Otherwise, we set $\gamma(n, q)$ to be a \perp -gate.

Second, for every 1-state q of A and every triple $(q_1, q_2, q) \in \delta_l$, let $g_1 := \gamma(n_1, q_1)$ and $g_2 := \gamma(n_2, q_2)$.

We define a gate g^{q_1, q_2} such that we have the equality:

$$S(g^{q_1, q_2}) = S(g_1) \times S(g_2) \quad (*)$$

but while respecting the rule that \top and \perp -gates can never be used as input to another gate. Specifically:

- If one of g_1, g_2 is a \perp -gate, we set g^{q_1, q_2} to be a \perp -gate, which clearly satisfies (*);
- If one of g_1, g_2 is a \top -gate, we set g^{q_1, q_2} to be the other gate; this also satisfies (*);
- Otherwise we set g^{q_1, q_2} to be a \times gate with inputs g_1 and g_2 .

Having created the necessary gates g^{q_1, q_2} for the triples of δ_l , we now create $\gamma(n, q)$ for every 1-state q as a gate that satisfies:

$$S(\gamma(n, q)) = \bigcup_{(q_1, q_2, q) \in \delta_l} S(g^{q_1, q_2}) \quad (**)$$

Specifically:

- If all g^{q_1, q_2} in the union are \perp -gates (in particular if the union is empty), we set $\gamma(n, q)$ to also be a \perp -gate, respecting (**);
- Otherwise we exclude all \perp -gates from the union and set $\gamma(n, q)$ to be a \cup -gate, which has all remaining gates g^{q_1, q_2} as input, satisfying (**).

We can easily check that all rules of assignment circuits are respected. In particular, all \cup -gates and \times -gates have the right fan-in. To check that we never use \top and \perp as input to another gate, the only subtlety is that, when defining the \cup -gate $\gamma(n, q)$ for a 1-state q , we must check that g^{q_1, q_2} can never be a \top -gate, but this is because one of q_1 and q_2 must be a 1-state, hence it cannot be a 0-state because A is homogenized; now it can be seen by induction that whenever $\gamma(n', q')$ is a \top -gate then q' is a 0-state. It is also clear that the definition of a structured complete DNNF is respected, in particular the inputs to \times -gates are \cup -gates in the two child boxes.

In terms of accounting, it is clear that there are at most $|Q|$ \cup -gates in each B_n , that the depth of the circuit is as stated, and the construction of the whole circuit is in time $O(|A| \times |T|)$ as promised. Last, a straightforward bottom-up induction on T shows that the gates $\gamma(n, q)$ capture the correct set for any n , i.e., that for any leaf node n and any $q \in Q$ we have:

$$S(\gamma(n, q)) = \{ \langle \mathcal{Y} : n \rangle \mid (\lambda(n), \mathcal{Y}, q) \in \iota \}$$

and for any internal node n with label l and children n_1 and n_2 and any $q \in Q$ we clearly have the following, by (*) and (**) (and their analogues in the case of 0-states):

$$S(\gamma(n, q)) = \bigcup_{(q_1, q_2, q) \in \delta_l} S(\gamma(n_1, q_1)) \times S(\gamma(n_2, q_2))$$

Hence, the construction is correct, which concludes the proof. \square

C PROOFS FOR SECTION 5 (ELIMINATING DUPLICATES)

LEMMA 5.1. *For any structured complete DNNF C , for any var-gate or \times -gate g of C and assignment S , if we have $S \in S(g)$, then the box of g is the (unique) least common ancestor of the boxes that contain the var-gates whose variables occur in S .*

PROOF. The definition of a structured complete DNNF clearly ensures that, letting $B := \text{box}(g)$, for any var-gate g' whose variables occur in S , the leaf box B' that contains g' must be a descendant of B in the tree of boxes. Hence, B must be a common ancestor of the boxes that contain the var-gates with the variables of S . Now, if B is a leaf box then it is necessarily the lowest common ancestor. Otherwise, g must be a \times -gate, so it has two inputs g_1 and g_2 which must be in the two child boxes B_1 and B_2 , and there are two assignments $S_1 \in S(g_1)$ and $S_2 \in S(g_2)$ such that $S_1 \cup S_2 = S$; further, as \top -gates are never used as input to another gate, neither S_1 nor S_2 are empty. Hence, B_1 and B_2 are both ancestors of some of the boxes that contain the var-gates whose variables occur in S , which clearly implies that their parent B cannot be a strict ancestor of the lowest common ancestor of this set of boxes. \square

THEOREM 5.3. *Given a structured complete DNNF C and given a boxed set Γ , we can enumerate $S(\Gamma)$ (without duplicates) with delay $O(|S| \times (\Delta + w^3))$, where S is the produced assignment, Δ is the delay of BOX-ENUM , and w is the width of C . Further, we correctly produce for each assignment S its provenance $\text{Prov}(S, \Gamma)$ relative to Γ .*

PROOF. We first show (1). The proof is by induction over the number of var-gates whose variables occur in the assignment. The base case is that of the assignments produced at Line 7, whose assignments are clearly correct.

For the induction case, we consider assignments produced at Line 16. By the induction hypotheses $S_L \in S(g_L)$ for every $g_L \in \Gamma'_L$ and $S_R \in S(g_R)$ for every $g_R \in \Gamma'_R$. Note now that the gates of G''_X have their left input in Γ'_L and their right input in Γ'_R , so indeed we have $S \in S(g'')$ for every $g'' \in G''_X$. Now, for every $g'' \in G''_X \circ W \circ R(B', \Gamma)$, we know that there is a \cup -gate g' and a gate $g \in \Gamma$ such that $(g'', g') \in W$ and $(g', g) \in R(B', \Gamma)$, i.e., by definition, $g'' \stackrel{\cup}{\sim} g$, so this witnesses that $S \in S(g)$ for some $g \in \Gamma$, concluding the proof of (1).

Now we show (2). The proof is again by induction over the size of assignments. The base case is again Line 7, which correctly produces all assignments that only involve one variable gate, by definition of $\text{BOX-ENUM}(\Gamma)$.

Let now $S \in S(\Gamma)$ be some assignment involving multiple var-gates. Clearly there must be a \times -gate $g_X \in \downarrow(\Gamma)$ witnessing that $S \in S(\Gamma)$, i.e., such that we have $S \in S(g_X)$. Let g_L and g_R be the left and right inputs of g_X , and let $S'_L \in S(g_L)$ and $S'_R \in S(g_R)$ be the assignments witnessing that $S \in S(g_X)$, i.e., we have $S = S'_L \cup S'_R$.

Let B' be the box of g_X . As $g_X \in \downarrow(\Gamma)$, we know that $\text{BOX-ENUM}(\Gamma)$ returns some relation $R(B', \Gamma)$, and then we know that G_X contains g_X . Therefore, we can conclude that $g_L \in \Gamma_L$. By the induction hypothesis we know that $(S'_L, \Gamma'_L) \in \text{ENUM}_S(\Gamma_L)$ and that $g_L \in \Gamma'_L$. Therefore, we can conclude that $g_X \in G'_X$, so that $g_R \in \Gamma_R$ and, again using the induction hypothesis, we have $(S'_R, \Gamma'_R) \in \text{ENUM}_S(\Gamma_R)$ with $g_R \in \Gamma'_R$. It follows that $g_X \in G''_X$. Now, as $R(B', \Gamma)$ was correctly computed by $\text{BOX-ENUM}(\Gamma)$, we have $(g_X, g) \in R(B', \Gamma)$, so that $g \in G''_X \circ W \circ R(B', \Gamma)$. Hence, we indeed produce (S, Γ') with a set Γ' that contains g .

At last, we show (3), i.e., that no assignment is output more than once. First observe that, by Lemma 5.1, the assignments captured by the gates of different boxes are disjoint, so that it suffices to show the claim for each $R(B', \Gamma)$. For assignments involving only one var-gate, the claim is immediate as the assignments that are produced all involve a different variable gate, and the labeling function S_{var} of variable gates is injective. For assignments involving multiple var-gates, we use the fact that by induction the recursive calls on Γ_L and Γ_R output each assignment once, and the properties of a structured complete DNNF ensures that each assignment $S \in S(g)$ for a \times -gate g has a unique partition (given following the v -tree) as $S_L \cup S_R$ with $S_L \in S(g_L)$ and $S_R \in S(g_R)$ for g_L and g_R the left and right inputs of g , respectively.

For delay, the analysis is similar to that of Proposition 4.2: producing an assignment S requires again $2|S| - 1$ recursive calls, and the delay of each call includes the delay Δ of the call to BOX-ENUM , plus the delay of the operations performed in Algorithm 2 which are bounded by $O(w^3)$ for w the width of the circuit. Indeed, remember in particular that each box contains at most w^2 \times -gates; the number of var-gates in each leaf box is unbounded but we make progress each time we examine one such gate. \square

D PROOFS FOR SECTION 6 (ENUMERATING BOXES EFFICIENTLY)

LEMMA 6.3. *Given a structured complete DNNF circuit C with v -tree \mathcal{T} , we can compute $I(C)$ in time $O(|\mathcal{T}| \times w^4)$, where w is the width of C .*

PROOF. We can compute the first interesting boxes for all \cup -gates in time $O(|\mathcal{T}| \times w)$ by a bottom up traversal of the \cup -gates of the circuit using the following equation:

$$\text{FIB}(g) = \begin{cases} \text{BOX}(g) & \text{if } g \text{ has a non-}\cup \text{ input} \\ \min_{(g', g) \in W_{\cup}} \text{FIB}(g') & \text{otherwise} \end{cases} \quad (3)$$

where relation W_{\cup} is the restriction of W to \cup -gates.

Likewise, the first bidirectional boxes of at most two \cup -gates can be computed in time $O(|\mathcal{T}| \times w^2)$ by

$$\text{FBB}(\Gamma) = \begin{cases} \text{UNDEF} & \text{if } \Gamma = \emptyset \\ \text{BOX}(\Gamma) & \text{if } \Gamma \text{ is bidirectional} \\ \text{FBB}(\{g' \mid (g', g) \in W_{\cup}, g \in \Gamma\}) & \text{otherwise} \end{cases} \quad (4)$$

where we say that Γ is *bidirectional* if some gate $g_L \in \Gamma$ has some input in the left child box of $\text{BOX}(\Gamma)$ and some gate $g_R \in \Gamma$ has some input in the right child box of $\text{BOX}(\Gamma)$ (note that we may take $g_L = g_R$). Observe that in the third case we call FBB on a boxed set for a child box: while this set may have cardinality > 2 , remember that we can evaluate FBB from the values computed for the child box, simply by applying Observation 6.2 the definition of FBB in Equation (2).

As for reachability relations, remember that we want to compute $R(\text{FIB}(g), B)$ and $R(\text{FBB}(\Gamma), B)$ for each \cup -gate $g \in B$ and for each boxed set $\Gamma \subseteq B$ such that $1 \leq |\Gamma| \leq 2$. Let us call the *target boxes* of B the boxes B' for which we want to compute $R(B', B)$. First observe that:

- If $B = B'$ then $R(B', B) = \{(g, g) \mid g \in B\}$
- If B' is a child of B , then $R(B', B)$ is easily computed from W_{\cup}
- If B' is a descendant of B but not a child, we have

$$R(B', B) = R(B', B'') \circ R(B'', B), \quad (5)$$

where B'' is the child of B in the direction of B' .

The crucial observation is that, in the last case, we must already have precomputed $R(B', B'')$ when processing the child B'' of B , i.e., B' is a target box of B'' . But indeed:

- If $B' = \text{FIB}(g)$ for some \cup -gate g of B , then if $B' \neq B$, the equation for FIB ensures that we must have $B' = \text{FIB}(g'')$ for some \cup -gate $g'' \in B''$, so B' was a target box of B'' .
- If $B' = \text{FBB}(\Gamma)$ for some boxed set Γ , then if $B' \neq B$, the equation for FBB ensures that we must have $B' = \text{FBB}(\Gamma'')$ for some boxed set Γ'' of B'' . Now using Observation 6.2 we know that we have $B' = \text{FBB}(\Gamma_2'')$ for some subset $\Gamma_2'' \subseteq \Gamma''$ of size at most 2, justifying that B' is a target box of B'' .

To bound the complexity, let us first analyze how many reachability relations we have to compute per box: first there are w for all possible $\text{FIB}(g)$ and another w for all possible $\text{FBB}(g)$. For the least common ancestors of all $\text{FBB}(g)$ at first glance it looks like we have to compute up to w^2 many reachability relations for each box. But thankfully, the set of possible least common ancestors is in fact of linear size. That is because the set $\{ B \mid B = \text{LCA}(\mathcal{B}'), \mathcal{B}' \subseteq \mathcal{B} \}$ is of size at most $|\mathcal{B}|$. Therefore it suffices to overall compute at most $3w$ many relations for each box. Thus altogether we compute $O(|\mathcal{T}| \times w)$ many relations, which takes time $O(|\mathcal{T}| \times w^4)$ altogether, as each relation can be computed in time $O(w^3)$ using the naïve join algorithm. \square

LEMMA 6.4. *Given a structured complete DNNFC and the index structure $I(C)$, we can implement BOX-ENUM with delay $O(w^3)$, where w is the width of C .*

PROOF. The algorithm $\text{BOX-ENUM}(\Gamma)$ to perform the enumeration for an arbitrary boxed set Γ is depicted as Algorithm 3. Each call of the recursive algorithm in Line 3 receives the relation

$R(B, \Gamma)$ for some box B called the *current box* and is expected to output the relations $R(B', \Gamma)$ for interesting boxes B' in the subtree of B .

In Line 6, the algorithm outputs the relation $R(B_1, \Gamma)$ for the first interesting box B_1 , so as to immediately make progress. Afterwards it does recursive calls in lines 8 and 10 that output all interesting boxes below the first interesting box if there are any. Finally, the algorithm walks down all bidirectional boxes on the path from B to B_1 and does recursive calls for the right children of these bidirectional boxes in Line 15 to enumerate all interesting boxes whose preorder traversal number is greater than the last interesting box enumerated in the subtree rooted at B_1 . By the definition of bidirectional boxes, we thus enumerate all interesting boxes. We have sketched the order in which the boxes are enumerated in Figure 1.

All relations $R(B_2, B_3)$ that are used in the algorithm are either part of the index structure $I(C)$ or are the identity ($B_2 = B_3$) or between a box and a child box (B_2 is a child of B_3). Therefore, all relational compositions can be computed in time $O(w^3)$ using the naïve join algorithm.

We show now that Algorithm 3 enumerates with constant delay, neglecting a small issue with the call stack that we discuss afterwards. The most important observation is that, by definition of bidirectional boxes, each recursive call will produce some output, and will do so after time at most $O(w^3)$, namely, the time needed to identify the first interesting box and produce the corresponding output. Then the time until we do the next recursive call (which itself will produce output in time $O(w^3)$) is also bounded by $O(w^3)$. Hence, the delay is dominated by the time spent computing the joins of the relations R .

The only subtlety in the delay analysis concerns the call stack. Indeed, its depth can be as large as the depth of C , so cleaning up the stack might take too much time. To avoid this problem, we need to ensure that between two outputs we do not have to clean up too many stack frames, which we do by modifying our code to apply tail recursion elimination as we now explain. We modify the procedure `B-ENUM` so that, during each call to the procedure, just before we do a recursive call, we test whether the rest of the current execution of the procedure will be making another recursive call. For example, before doing the recursive call in Line 15, we have to check whether there is another bidirectional box that we need to visit in the next iteration of the while loop. Adding these tests does not impact the delay. Now, if we notice that a recursive call is the last one in the current execution of the procedure, we perform tail recursion, i.e., we do the recursive call by setting the argument of `B-ENUM` and jumping to Line 4 without adding anything to the call stack. It is clear that this change does not modify what the algorithm computes, as whenever we do this we have checked that the rest of the current execution of the procedure will not be making any more recursive calls; nor does the change make the delay worse.

To understand why the algorithm is in delay $O(w^3)$ after this change, observe that, with the modified algorithm, we no longer need to clean up more than one stack frame before we can produce the next output. Indeed, if we clean up a stack frame, then we know that the call to which we return will be making another recursive call. By our analysis, it does so after a delay of at most $O(w^3)$, and then we know that the call produces an output after an additional delay of at most $O(w^3)$. Hence it is indeed the case that the overall delay in is $O(w^3)$, which concludes the proof. \square

E PROOFS FOR SECTION 7 (UPDATES AND BALANCING)

In this appendix, we give some more explanations about the proof of Lemma 7.4 using [30]. We start with some prerequisites about forest algebra terms.

Forest Algebra Terms. A *forest algebra pre-term* is a term in the free forest algebra as defined in [30] that does not use the empty forest and empty context. We repeat the definition for self-containedness.

A Λ -*forest* is an ordered list of Λ -trees. A Λ -*context* is a $(\Lambda \cup \{\square\})$ -forest, where the special “hole” label \square is applied to no internal node and to exactly one leaf.

A *forest algebra pre-term* on an alphabet Λ is a binary tree whose internal nodes are labeled \oplus_{HH} (for forest concatenation), \oplus_{HV} , \oplus_{VH} (for concatenation of forest and context or vice versa), \odot_{VV} (for context application between contexts) and \odot_{VH} (for context application on a forest), whose leaves are labeled with a_t for $a \in \Lambda$ (for a node labeled a), or a_\square for $a \in \Lambda$ (for a context node labeled a), and where we require that some typing constraints are respected. Specifically, each node of the tree is typed as a forest or as a context, with a_t being a forest and a_\square being a context, and the type of inner nodes is defined by induction:

- \oplus_{HH} both inputs must be forests, and the result is a forest;
- \oplus_{HV} the left input must be a forest, the right input must be a context, and the result is a context;
- \oplus_{VH} the right input must be a forest, the left input must be a context, and the result is a context;
- \odot_{VV} both inputs must be contexts, and the result is a context;
- \odot_{VH} the left input must be a context, the right input must be a forest, and the result is a forest.

We say that a forest algebra term *represents* an unranked forest or context on the alphabet Λ , which is defined by induction, preserving the invariant that a node typed as a forest represents a Λ -forest and a node typed as a context represents a Λ -context:

- a_t represents the forest with a singleton root labeled a ;
- a_\square represents the context with a singleton root labeled a and having a single child labeled \square ;
- for a \oplus -node, given the contexts or forests F_1 and F_2 represented by the first and second input, the result is $F_1 \cup F_2$;
- for a \odot -node, given the context F_1 and the forest or context F_2 represented respectively by the first and second input, the result is obtained by replacing the one \square -labeled node of F_1 by F_2 , i.e., the \square -labeled node n of F_1 is removed and the roots of the trees in F_2 are inserted in the list of children of the parent of n at the point where n was, in their order according to F_2 ;

A *forest algebra term* is a forest algebra pre-term that represents a Λ -forest (i.e., the root has type forest), and where this forest contains exactly one tree.

LEMMA 7.4. *For any tree alphabet Λ and set X of variables, there is an encoding scheme ω for Λ such that:*

- *The encoding is linear-time computable, i.e., given any unranked Λ -tree T , we can compute in linear time some Λ' -tree T' with $\omega(T') = T$, as well as the bijection $\varphi_{T'}$.*
- *The encoded trees have logarithmic height, i.e., each well-formed Λ' -tree T' has height in $O(\log |T'|)$.*
- *We can efficiently perform updates, i.e., given any binary Λ -tree T , preimage T' of T , and update τ on T in the language of Definition 7.1, we can compute in time $O(\log |T|)$ a tree hollowing $H = (T'', \eta)$ of T' such that $\omega(H^{res}) = \tau(\omega(T'))$.*

- We can efficiently translate automata, i.e., given any unranked Λ, \mathcal{X} -TVA A with state space Q , we can build in time $O(|Q|^6)$ a binary Λ', \mathcal{X} -TVA A' with $O(|Q|^4)$ states and $O(|Q|^6)$ transitions such that ω is A, A' -faithful. Furthermore, A' has a single accepting state.

PROOF. We let Λ' be the alphabet of forest algebra terms over alphabet Λ , and we define ω to be the function that maps forest algebra terms over alphabet Λ to the unranked Λ -trees that they represent. We artificially restrict ω to be only defined on those forest algebra terms that have logarithmic height in order to satisfy the second condition.

It is easy to see that every tree can be encoded as a forest algebra term. In [30], it is shown that for each tree there exists a forest algebra term of logarithmic height and that we can efficiently perform updates. Although this is not stated explicitly in [30], all updates of forest algebra terms can be described as tree hollowings of logarithmic size. The initial update on the forest algebra term is of constant size and all successive rotations done for rebalancing are either performed on the path from the updated node to the root or performed on a direct child of a node from this path, so the result of the update on the forest algebra is indeed representable as a trunk of logarithmic size to which we connect subtrees of the original forest algebra term.

There is a natural bijection between leaves of forest algebra terms and nodes in the represented tree.

The last point is to explain how we convert an unranked Λ, \mathcal{X} -TVA $A = (Q, \delta, F)$ to a binary Λ', \mathcal{X} -TVA $A' = (Q', \delta', F')$. We do so as follows, where we assume w.l.o.g. that we have added to Q some special states q_0, q_f such that $\delta \cap (\{q_0\} \times Q \times \{q_f\}) = \{q_0\} \times F \times \{q_f\}$. This will help us later to identify the accepting runs of A .

$$\begin{aligned}
\Lambda' &= \{a_t \mid a \in \Lambda\} \cup \{a_\square \mid a \in \Lambda\} \cup \{\oplus_{HH}, \oplus_{HV}, \oplus_{VH}, \odot_{VV}, \odot_{VH}\} \\
Q' &= Q^2 \cup (Q^2)^2 \\
F' &= \{(q_0, q_f)\} \\
\iota' &= \left\{ (a_t, \mathcal{Y}, (q_1, q_2)) \mid (q_1, p, q_2) \in \delta \text{ for some } p \in \iota(a, \mathcal{Y}) \right\} \cup \\
&\quad \left\{ (a_\square, \mathcal{Y}, ((q_1, q_2), (q_3, q_4))) \mid (q_1, q_4, q_2) \in \delta, q_3 \in \iota(a, \mathcal{Y}) \right\} \\
\delta'_{\oplus_{HH}} &= \left\{ \left((q_1, q_2), (q_2, q_3), (q_1, q_3) \right) \mid q_1, q_2, q_3 \in Q \right\} \\
\delta'_{\oplus_{HV}} &= \left\{ \left((q_1, q_2), ((q_2, q_3), (q_4, q_5)), ((q_1, q_3), (q_4, q_5)) \right) \mid q_1, \dots, q_5 \in Q \right\} \\
\delta'_{\oplus_{VH}} &= \left\{ \left(((q_1, q_2), (q_3, q_4)), (q_2, q_5), ((q_1, q_5), (q_3, q_4)) \right) \mid q_1, \dots, q_5 \in Q \right\} \\
\delta'_{\odot_{VV}} &= \left\{ \left(((q_1, q_2), (q_3, q_4)), ((q_1, q_2), (q_3, q_4)), ((q_1, q_2), (q_3, q_4)) \right) \mid q_1, \dots, q_6 \in Q \right\} \\
\delta'_{\odot_{VH}} &= \left\{ \left(((q_1, q_2), (q_3, q_4), (q_3, q_4), (q_1, q_2))) \mid q_1, \dots, q_4 \in Q \right\}
\end{aligned}$$

The construction of A' is directly derived from the definition of the transition algebra as given in [30, Section 4]. The resulting automaton has $O(|Q|^4)$ states and $O(|Q|^6)$ transitions and can be computed in time $O(|Q|^6)$.

We have to prove faithfulness. Rather than repeating the definition of transition algebra form [30], we give a direct description on how A' captures runs of A . This is similar to the proof of Lemma 9 in [30].

In the following, we let T' be an input tree for A' and $T = \omega(T')$ and let ψ be a function mapping nodes of T' to the forest or context represented by the forest algebra pre-term rooted at n .

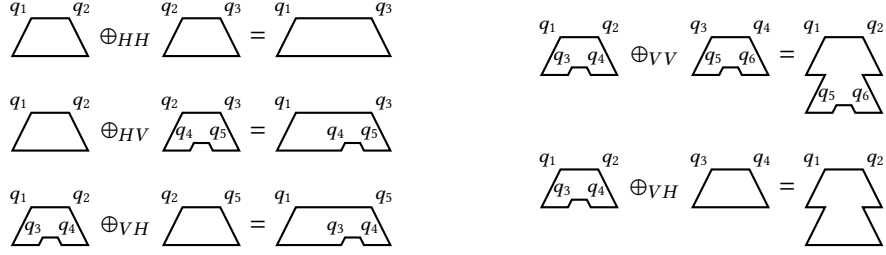


Figure 2: Visualisation of the monoid operations. Forests are depicted as trapezoids and contexts as trapezoids with a cutout. States from A that are memorized in states of A' are indicated at the corresponding corners.

It can be verified by induction that the state q assigned to a node n in a run of A' satisfies the following conditions that describe the partition of Q' into states Q^2 for nodes of type forest and $(Q^2)^2$ for nodes of type context.

- If $\psi(n)$ is a forest then $q = (q_1, q_2) \in Q^2$ such that there exists a run ρ on $\psi(n)$ such that we have $\delta(q_1, \rho(n_1) \cdots \rho(n_m)) = q_2$, where n_1, \dots, n_m are the roots of $\psi(n)$.
- If $\psi(n)$ is a context then $q = ((q_1, q_2), (q_3, q_4))$ such that—after replacing the hole in $\psi(n)$ with a forest that allows the transition from q_1 to q_2 —there exists a run ρ such that $\delta(q_3, \rho(n_1) \cdots \rho(n_m)) = q_4$, where n_1, \dots, n_m are the roots of $\psi(n)$.

The induction base case is at the leaves of T' . If n is labeled a_t with $a \in \Lambda$, A' nondeterministically guesses a pair of states such that $\delta(q_1, p) = q_2$, where p is a state that A could assign to a leaf with label a , i.e., it has to be an initial state that is possible for label a and the given variable assignment at the node.

If n is labeled a_\square , then A' guesses a pair of states $((q_1, q_2), (q_3, q_4))$ such that the forest that will be inserted into the hole allows a transition from q_3 to q_4 . Furthermore A has to allow a transition from q_1 to q_2 when reading the state assigned to $\varphi_{T'}(n)$.

For the induction step, one has to look at the transitions allowed by A' on inner nodes, i.e., on nodes with a label from $\{\oplus_{HH}, \oplus_{HV}, \oplus_{VH}, \oplus_{VV}, \oplus_{VH}\}$. There A' verifies deterministically whether the nondeterministic choices done at the leaves are consistent. In the following n will always be an inner node with left child n_L and right child n_R . We depicted a sketch of all five monoid operations in Figure 2. There we also indicate the state names used in the definition of δ' .

If n has label \oplus_{HH} , A' has to check that the state q_2 after reading the (roots of the) forest $\psi(n_L)$ has to be the same as the state before reading the (roots of the) forest $\psi(n_R)$. Furthermore, A' propagates the states (q_1, q_2) upwards, where q_1 is the state before reading $\psi(n_L)$ and q_3 is the state after reading $\psi(n_R)$.

If n has label \oplus_{VH} , A' checks that the forest $\psi(n_R)$ actually allows the transition that was guessed for the hole at some leaf with some label a_\square .

In the two described cases, the resulting type is forest. In the remaining three cases, where the resulting type is context, A' additionally has to propagate the guess for the hole upwards, such that it can be verified later, when the automaton reaches a node labeled \oplus_{VH} .

The correctness of this verification at inner nodes can be verified using the definition of δ' . At last, the automaton has to check that all the guesses are not only consistent, but belong to an accepting run of A . This is reflected by our choice of F' that verifies that A can do a transition

from q_0 to q_f when reading the state assigned to the root of $\omega(T)$. According to our assumption, this implies that the root is assigned a state from F and thus A accepts $\omega(T)$.

We note that the translation preserves runs in the sense that for every unique run of A on $\omega(T)$, there is a unique run of A' on T' . Especially, if A is deterministic or unambiguous, then A' is unambiguous. Furthermore, for nondeterministic automata A , the number of runs is preserved. \square

F PROOFS FOR SECTION 8 (MAIN RESULTS)

THEOREM 8.1. *Let ω be an exponent for the Boolean matrix multiplication problem. Given an unranked Λ, X -TVA A with state space Q and an unranked Λ -tree T , we can enumerate the satisfying assignments of A on T with preprocessing time $O(|T| \times |Q|^{4(\omega+1)})$, update time $O(\log(|T|) \times |Q|^{4(\omega+1)})$, and delay $O(|Q|^{4\omega} \times |S|)$, where S is the produced assignment.*

PROOF. Let ω be the encoding scheme of Lemma 7.4. In the preprocessing phase, we apply Lemma 7.4 to compute a Λ' -tree T' such that $\omega(T') = T$, and the bijection $\varphi_{T'}$, in time $O(|T|)$. We also know that the height of T' is logarithmic. We also translate the automaton A in time $O(|Q|^6)$ to a binary Λ', X -TVA A' with state space Q' such that $|Q'| = O(|Q|^4)$ and such that A' has $O(|Q|^6)$ -transitions and has only one final state. We then use Lemma 2.1 to process A' and ensure that it is homogenized; the construction clearly ensures that A' then has exactly two final states: one final 0-state $q_{f,0}$ and one final 1-state $q_{f,1}$. Further, we know that ω is A, A' -faithful, meaning in particular that for any X -valuation ν of the unranked tree T , letting $\nu' := \nu \circ \varphi_{T'}$, we have that A accepts T under ν iff A' accepts T' under ν' .

We now apply Lemma 3.7 to construct in time $O(|T'| \times |A'|)$, i.e., $O(|T| \times |Q|^6)$, a structured complete DNNF C which is an assignment circuit of A and T , such that the width w of C is $|Q'|$, i.e., $O(|Q|^4)$, and its depth is $O(\text{height}(T'))$, i.e., $O(\log(\text{height}(T)))$. We will then perform enumeration using Theorem 6.5, which includes a preprocessing phase in time $O(|C| \times w^\omega)$, i.e., $O(|T| \times w^{4\omega+6})$, to compute the index $I(C)$. Hence, the preprocessing time is as we claimed.

After the preprocessing, letting $g_{f,1} := \varphi(n, q_{f,1})$ where n is the root of T' , we can enumerate the assignments of $S(\{q_{f,1}\})$ with delay $O(|S| \times w^{\text{omega}})$, i.e., $O(|S| \times |Q|^{4\omega})$, where S is the produced assignment, which is the time bound that we claimed. To see why this is correct, observe that by definition of an assignment circuit, this enumerates precisely the assignments corresponding to valuations ν' such that A' has a run mapping the root of T' to $q_{f,1}$, and as $q_{f,1}$ is the only final 1-state of A' it is exactly the set of assignments corresponding to non-empty valuations ν' such that A' accepts T' under ν' . Last, we handle the case of the empty valuation by considering $g_{f,0} := \varphi(n, q_{f,0})$ for n the root of T' , which must clearly be either a \top -gate or \perp -gate, and we produce the empty assignment iff $g_{f,1}$ is a \top -gate: by the same reasoning this produces the empty assignment iff A' accepts T' under the empty valuation. Thus, we correctly enumerate the set of assignments corresponding to valuations ν' such that A' accepts T' under ν' , i.e., the satisfying assignments of A' on T' . As ω is A, A' -faithful, this enumerates exactly the satisfying assignments of A on T . Hence, the enumeration is correct.

Now, whenever an update is performed on T , we know by Lemma 7.4 that we can compute in time $O(\log T)$ a tree hollowing $H = (T'', \eta)$ of T' such that $\omega(H^{\text{res}}) = \tau(\omega(T))$. Now, we know that we can reflect this change on C and on the index $I(C)$ to obtain a new circuit C^{res} and index $I(C^{\text{res}})$ such that C^{res} is a structured complete DNNF of width $|Q'|$, i.e., the same as C , which is an assignment circuit of A' on the new Λ' -tree H^{res} , and $I(C^{\text{res}})$ is the index structure for C^{res} . This update takes time $O(|T'| \times |Q'|^{\omega+1})$, i.e., $O(|T| \times |Q|^{4\omega+1})$. Thus the update time is as we claimed, which concludes the proof. \square

COROLLARY 8.3. *For any fixed tree alphabet Λ , for any fixed Λ -MSO query Φ with free first-order variables, given an unranked Λ -tree T , after preprocessing T in linear time, we can enumerate the satisfying assignments of Φ with constant delay, and we can handle updates to T in logarithmic time in T .*

PROOF. We do the standard rewriting of $\Phi(x_1, \dots, x_n)$ to an MSO query $\Phi'(X_1, \dots, X_n)$ with free second-order variables where we add for each i a conjunct asserting that X_i is a singleton (e.g., $\exists x X_i(x) \wedge (\forall xy X_i(x) \wedge X_i(y) \rightarrow x = y)$) and we add an existential quantification $\exists x_i X_i(x_i)$, then we reuse the body of Φ . This rewriting is independent of T , so runs in constant time. Now, we use Corollary 8.2 to enumerate the satisfying assignments of Φ' on T . The definition of Φ' ensures that each satisfying assignment has cardinality exactly n , so the enumeration proceeds in constant time. This produces the desired result because there is a clear bijection from the satisfying assignments of Φ' on T to the answer tuples of Φ on T , which we can apply to each satisfying assignment in constant time. \square

COROLLARY 8.4. *Lemma 7.4 holds for words instead of trees and a WVA A instead of an unranked TVA as input, with the difference that the binary tree automaton A' can be constructed in time $O(|Q|^3)$, has $O(|Q|^2)$ states and $O(|Q|^3)$ transitions, where Q is the state space of A .*

PROOF. We interpret strings as forests, where each tree has exactly one node. We can use the same balancing schema that happens to work exactly like AVL trees, if the only operation is concatenation (\oplus_{HH}).

We also reuse the automaton construction with the difference that we can drop everything that is related to contexts. Especially, we can use $Q' = Q^2$, as there are no nodes of type context. Furthermore, we only need $\delta'_{\oplus_{HH}}$ from the definition of δ' , as there are no other operators. W.l.o.g. we assume that A has only one initial state q_0 and one final state q_F .

This can be easily achieved by adding a new initial state q_0 and final state q_F , such that q_0 gets all outgoing transitions from all initial states and q_F gets all incoming transitions from all existing final states. Afterwards we use q_0 and q_F as single initial and final state, respectively.

We use (q_0, q_F) as the single final state of A' . \square

G PROOFS FOR SECTION 9 (LOWER BOUND)

THEOREM 9.2. *There is an MSO query Φ on trees such that any enumeration algorithm for Φ under relabelings with update time \hat{t}_u and enumeration delay \hat{t}_e has*

$$\max(\hat{t}_u, \hat{t}_e) \geq \Omega\left(\frac{\log(n)}{\log \log(n)}\right).$$

PROOF. The beginning of the proof is as explained in the sketch: we now give the computation for the lower bound. The runtime of the marked ancestor queries as we implemented them is $t_\Phi = 2\hat{t}_u + \hat{t}_e$. From Theorem 9.1 we get

$$t_\Phi = 2\hat{t}_u + \hat{t}_e \geq c \frac{\log(n)}{\log(\hat{t}_u \log(n))} = c \frac{\log(n)}{\log(\hat{t}_u \log(n))} \quad (6)$$

for some constant c .

Now, assume first that $\hat{t}_u > \hat{t}_e$. Then we get from (6) that

$$3\hat{t}_u \geq c \frac{\log(n)}{\log(\hat{t}_u \log(n))}.$$

Now assume w.l.o.g. that $\hat{t}_u \leq \log(n)$ (otherwise we are done), then by substituting on the right-hand side we get

$$3\hat{t}_u \geq c \frac{\log(n)}{\log(\log(n) \log(n))} \geq \frac{\log(n)}{2 \log \log(n)}$$

which completes the proof.

If $\hat{t}_e \geq \hat{t}_u$, then we get from (6) that

$$3\hat{t}_e \geq c \frac{\log(n)}{\log(\hat{t}_e \log(n))}$$

and reasoning as before completes the proof. □