Ranked Enumeration for MSO on Trees via Knowledge Compilation

Antoine Amarilli
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Pierre Bourhis
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Florent Capelli
Univ. Artois, CNRS, UMR 8188, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France

Mikaël Monet
Université de Lille, CNRS, Inria, UMR 9189 – CRIStAL, F-59000 Lille, France

Abstract

We study the problem of enumerating the satisfying assignments for certain circuit classes from knowledge compilation, where assignments are ranked in a specific order. In particular, we show how this problem can be used to efficiently perform ranked enumeration of the answers to MSO queries over trees, with the order being given by a ranking function satisfying a subset-monotonicity property.

Assuming that the number of variables is constant, we show that we can enumerate the satisfying assignments in ranked order for so-called *multivalued circuits* that are smooth, decomposable, and in negation normal form (smooth multivalued DNNF). There is no preprocessing and the enumeration delay is linear in the size of the circuit times the number of values, plus a logarithmic term in the number of assignments produced so far. If we further assume that the circuit is deterministic (smooth multivalued d-DNNF), we can achieve linear-time preprocessing in the circuit, and the delay only features the logarithmic term.

2012 ACM Subject Classification Information systems → Relational database model

Keywords and phrases Enumeration, knowledge compilation, monadic second-order logic

Digital Object Identifier 10.4230/LIPIcs.ICDT.2024.25

Related Version *Full Version (with all Proofs)*: `https://arxiv.org/abs/2310.00731` [2]

Funding *Antoine Amarilli*: Partially supported by the ANR project EQUUS ANR-19-CE48-0019, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758, and by the ANR project ANR-18-CE23-0003-02 ("CQFD"). This work was done in part while the author was visiting the Simons Institute for the Theory of Computing.
Florent Capelli: This work was supported by project ANR KCODA, ANR-20-CE48-0004.
Mikaël Monet: This work was done in part while the author was visiting the Simons Institute for the Theory of Computing.

1 Introduction

Data management tasks often require the evaluation of queries on large datasets, in settings where the number of query answers may be very large. For this reason, the framework of *enumeration algorithms* has been proposed as a way to distinguish the *preprocessing time* of query evaluation algorithms and the maximal *delay* between two successive answers [32, 37]. Enumeration algorithms have been studied in several contexts: for conjunctive queries [8] and unions of conjunctive queries [10, 16] over relational databases; for first-order logic over bounded-degree structures [23], structures with local bounded expansion [33], and nowhere dense graphs [31]; and for monadic second-order logic (MSO) over trees [7, 25, 3].

We focus on the setting of MSO over trees. In this context, the following enumeration result is already known. For any fixed MSO query Q (i.e., in *data complexity*) where the free variables are assumed to be first-order, considering the answers of Q on a tree T given as input (i.e., the functions that map the variables of Q to nodes of T in a way that satisfies Q), we can enumerate them with linear preprocessing on the tree T and with constant delay. If the free variables are second-order, then the delay is output-linear, i.e., linear in each produced answer [7, 3]. Further results are known when the query is not fixed but given as input as a potentially non-deterministic automaton [4, 5], or when maintaining the enumeration structure under tree updates [28, 5].

However, despite their favorable delay bounds, a shortcoming of these enumeration algorithms is that they enumerate answers in an opaque order which cannot be controlled. This is in contrast with application settings where answers should be enumerated, e.g., by decreasing order of relevance, or focusing on the *top-k* most relevant answers. This justifies the need for enumeration algorithms that can produce answers in a user-defined order, even if they do so at the expense of higher delay bounds.

This task, called *ranked enumeration*, has recently been studied in various contexts. For instance, Carmeli et al. [17, 13, 14] study for which order functions one can efficiently perform ranked direct access to the answers of conjunctive queries: here, efficient ranked direct access implies efficient ranked enumeration. Ranked enumeration has also been studied to support order-by operators on factorized databases [9]. Other works have studied ranked enumeration for document spanners [21], which relate to the evaluation of MSO queries over words. Closer to applications, some works have studied the ranked enumeration of conjunctive query answers, e.g., Deep et al. [20, 19] or Tziavelis et al. [35, 36]. Variants of in-order enumeration have been also studied on knowledge compilation circuit classes, for instance top-k, with a pseudo-polynomial time algorithm [11]. Closest to the present work, Bourhis et al. [12] have studied enumeration on *words* where the ranking function on answers is expressed in the formalism of MSO cost functions. They show that enumeration can be performed with linear preprocessing, with a delay between answers which is no longer constant but logarithmic in the size of the input word. However, their result does not apply in the more general context of trees.

Contributions. In this paper, we embark on the study of efficient ranked enumeration algorithms for the answers to MSO queries on trees, assuming that all free variables are first-order. We define this task by assigning *scores* to each so-called *singleton assignment* $[x \to d]$ describing that variable x is assigned tree node d, and combining these values into a *ranking function* while assuming a *subset-monotonicity property* [36]: intuitively, when extending two partial assignments in the same manner, then the order between them does not change. This setting covers many ranking functions, e.g., those defined by order, sum, or a lexicographic order on the variables. Our main contribution is then to show the following results on the data complexity of ranked enumeration for MSO queries on trees:

▶ **Result 1.** *For any fixed MSO query $Q(x_1, \ldots, x_n)$ with free first-order variables, given as input a tree T and a subset-monotone ranking function w on the partial assignments of $x_1, \ldots, x_n$ to nodes of T, we can enumerate the answers to Q on T in nonincreasing order of scores according to w with a preprocessing time of $O(|T|)$ and a delay of $O(\log(K + 1))$, where K is the number of answers produced so far.*

Note that, as the total number of answers is at most $|T|^n$, and as n is constant in data complexity, the delay of $O(\log(K+1))$ can alternatively be bounded by $O(n \log |T|)$, or $O(\log |T|)$. This matches the bound of [12] on words, though their notion of rank is different. Further, our bound shows that the first answers can be produced faster, e.g., for top-k computation.

Our results for MSO queries on trees are shown in the general framework of *circuit-based enumeration methods*, introduced by [3]. In this framework, enumeration results are achieved by first translating the task to a class of structured circuits from knowledge compilation, and then proposing an enumeration algorithm that works directly on the structured class. This makes it possible to re-use enumeration algorithms across a variety of problems that compile to circuits. In this paper, as our task consists in enumerating assignments (from first-order variables of an MSO query to tree nodes), we phrase our results in terms of *multivalued circuits*. These circuits generalize Boolean circuits by allowing variables to take values in a larger domain than $\{0, 1\}$: intuitively, the domain will be the set of the tree nodes. We assume that circuits are *decomposable*, i.e., that no variable has a path to two different inputs of a $\wedge$-gate: this yields *multivalued DNNFs*, which generalize usual DNNFs. We also assume that the circuits are *smooth*: intuitively, no variable is omitted when combining partial assignments at an $\vee$-gate. Multivalued circuits can be smoothed while preserving decomposability, in quadratic time or faster in some cases [34]. Smooth multivalued DNNF circuits can alternatively be understood as factorized databases, but we do not impose that they are *normal* [30], i.e., the depth can be arbitrary.

Our enumeration task for MSO on trees thus amounts to the enumeration of satisfying assignments of smooth multivalued DNNFs, following a ranking function which we assume to be subset-monotone. However, we are not aware of existing results for ranked enumeration on circuits in the knowledge compilation literature. For this reason, the second contribution of this paper is to show efficient enumeration algorithms on these smooth multivalued DNNFs.

We first present an algorithm for this task that runs with no preprocessing and polynomial delay. The algorithm can be seen as an instance of the Lawler-Murty [26, 29] procedure. We show:

▶ **Result 2.** *For any constant $n \in \mathbb{N}$, given a smooth multivalued DNNF circuit C with domain D and with n variables, given a subset-monotone ranking function w, we can enumerate the satisfying assignments of C in nonincreasing order of scores according to w with delay $O(|D| \times |C| + \log(K+1))$, where K is the number of assignments produced so far.*

We then show a second algorithm, which allows for a better delay bound at the expense of making an additional assumption on the circuit; it is with this algorithm that we prove Result 1. The additional assumption is that the circuit is *deterministic*: intuitively, no partial assignment is captured twice. This corresponds to the class of *smooth multivalued d-DNNF circuits*. For our task of enumerating MSO query answers, the determinism property can intuitively be enforced on circuits when we compute them using an deterministic tree automaton to represent the query. We then show:

▶ **Result 3.** *For any constant $n \in \mathbb{N}$, given a smooth multivalued d-DNNF circuit C with n variables, given a subset-monotone ranking function w, we can enumerate the satisfying assignments of C in nonincreasing order of scores according to w with preprocessing time $O(|C|)$ and delay $O(\log(K+1))$, where K is the number of assignments produced so far.*

Paper structure. We give preliminary definitions in Section 2. We first study in Section 3 the ranked enumeration problem for smooth multivalued DNNF circuits (Result 2). We then move on to a more efficient algorithm on smooth multivalued d-DNNF circuits (Result 3) in Section 4. We show how to apply the second algorithm to ranked enumeration for the answers to MSO queries (Result 1) in Section 5. We conclude in Section 6. Missing proofs can be found in the full version [2].

2 Preliminaries

For $n \in \mathbb{N}$, we write $[n]$ for the set $\{1, \ldots, n\}$.

Assignments. For two finite sets D of *values* and X of *variables*, an *assignment on domain D and variables X* is a mapping from X to D. We write D^X the set of such assignments. We can see assignments as sets of *singleton assignments*, where a *singleton assignment* is an expression of the form $[x \to d]$ with $x \in X$ and $d \in D$.

Two assignments $\tau \in D^Y$ and $\sigma \in D^Z$ are *compatible*, written $\tau \simeq \sigma$, if we have $\tau(x) = \sigma(x)$ for every $x \in Y \cap Z$. In this case, we denote by $\tau \bowtie \sigma$ the assignment of $D^{Y \cup Z}$ defined following the natural join, i.e., for $y \in Y \setminus Z$ we set $(\tau \bowtie \sigma)(y) := \tau(y)$, for $z \in Z \setminus Y$ we set $(\tau \bowtie \sigma)(z) := \sigma(z)$, and for $x \in Z \cap Y$, we set $(\tau \bowtie \sigma)(x)$ to the common value $\tau(x) = \sigma(x)$. Two assignments $\tau \in D^Y$ and $\sigma \in D^Z$ are *disjoint* if $Y \cap Z = \emptyset$: then they are always compatible and $\tau \bowtie \sigma$ corresponds to the relational product, which we write $\tau \times \sigma$.

Given $R \subseteq D^Y$ and $S \subseteq D^Z$, we define $R \wedge S = \{\tau \bowtie \sigma \mid \tau \in R, \sigma \in S, \tau \simeq \sigma\}$: this is a subset of $D^{Y \cup Z}$. Note how, if the domain is $D = \{0, 1\}$, then this corresponds to the usual conjunction for Boolean functions, and in general we can see it as a relational join, or a relational product whenever $Y \cap Z = \emptyset$. Further, we define $R \vee S = \{\tau \in D^{Y \cup Z} \mid \tau|_Y \in R \text{ or } \tau|_Z \in S\}$, which is again a subset of $D^{Y \cup Z}$. Again observe how, when $D = \{0, 1\}$, this corresponds to disjunction; and in general we can see this as relational union except that assignments over Y and Z are each implicitly completed in all possible ways to assignments over $Y \cup Z$.

Multivalued circuits. A *multivalued circuit C on domain D and variables X* is a DAG with labeled vertices which are called *gates*. The circuit also has a distinguished gate r called the *output gate of C*. Gates having no incoming edges are called *inputs* of C. Moreover, we have:
- Every input of D is labeled with a pair of the form $\langle x : d \rangle$ with $x \in X$ and $d \in D$;
- Every other gate of D is labeled with either $\vee$ (a $\vee$-*gate*) or $\wedge$ (a $\wedge$-*gate*).

We denote by $|C|$ the number of edges in C.

Given a gate v of C, the *inputs of v* are the gates w of C such that there is a directed edge from w to v. The *set of variables below v*, denoted by $\mathsf{var}(v)$, is then the set of variables $x \in X$ such that there is an input w which is labeled by $\langle x : d \rangle$ for some $d \in D$ and which has a directed path to v. Equivalently, if v is an input labeled by $\langle x : d \rangle$ then $\mathsf{var}(v) := \{x\}$, otherwise $\mathsf{var}(v) := \bigcup_{i=1}^{k} \mathsf{var}(v_i)$ where $v_1, \ldots, v_k$ are the inputs of v. We assume that the set X of variables of the circuit is equal to $\mathsf{var}(r)$ for r the output gate of C: this can be enforced without loss of generality up to removing useless variables from X.

For each gate v of C, the *set of assignments* $\mathsf{rel}(v) \subseteq D^{\mathsf{var}(v)}$ *of v* is defined inductively as follows. If v is an input labeled by $\langle x : d \rangle$, then $\mathsf{rel}(v)$ contains only the assignment $[x \mapsto d]$. Otherwise, if v is an internal gate with inputs $v_1, \ldots, v_k$ then $\mathsf{rel}(v) := \mathsf{rel}(v_1) \mathbin{\mathsf{op}} \cdots \mathbin{\mathsf{op}} \mathsf{rel}(v_k)$ where $\mathsf{op} \in \{\vee, \wedge\}$ is the label of v. The *set of assignments* $\mathsf{rel}(C)$ of C is that of its output gate. Note that, if $D = \{0, 1\}$, then the set of assignments of C precisely corresponds to its satisfying valuations when we see C as a Boolean circuit in the usual sense.

We say that a $\wedge$-gate v is *decomposable* if all its inputs are on disjoint sets of variables; formally, for every pair of inputs $v_1 \neq v_2$ of v, we have $\mathsf{var}(v_1) \cap \mathsf{var}(v_2) = \emptyset$. A $\vee$-gate v is *smooth* if all its inputs have the same set of variables (so that implicit completion does not occur); formally, for every pair of inputs v_1, v_2 of v, we have $\mathsf{var}(v_1) = \mathsf{var}(v_2)$. A $\vee$-gate v is *deterministic* if every assignment of v is computed by only one of its inputs; formally, for every pair of inputs $v_1 \neq v_2$ of v, if $\tau \in \mathsf{rel}(v)$ then either $\tau|_{\mathsf{var}(v_1)} \notin \mathsf{rel}(v_1)$ or $\tau|_{\mathsf{var}(v_2)} \notin \mathsf{rel}(v_2)$.

Let v be an internal gate with inputs $v_1, \ldots, v_k$. Observe that if v is decomposable, then $\mathsf{rel}(v) = \bigtimes_{i=1}^{k} \mathsf{rel}(v_i)$. If v is smooth then $\mathsf{rel}(v) = \bigcup_{i=1}^{k} \mathsf{rel}(v_i)$. If moreover v is deterministic, then $\mathsf{rel}(v) = \biguplus_{i=1}^{k} \mathsf{rel}(v_i)$, where $\uplus$ denotes disjoint union. Accordingly, we denote decomposable $\wedge$-nodes as $\times$-*nodes*, denote smooth $\vee$-nodes as $\cup$-*nodes*, and denote smooth deterministic $\vee$-nodes as $\uplus$-*nodes*.

A multivalued circuit is *decomposable* (resp., *smooth*, *deterministic*) if every $\wedge$-gate is decomposable (resp., every $\vee$-gate is smooth, every $\vee$-gate is deterministic). A *multivalued DNNF on domain D and variables X* is then a decomposable multivalued circuit on D and X. A *multivalued d-DNNF on domain D and variables X* is a deterministic multivalued DNNF on D and X. In all this paper, we only work with circuits that are both decomposable and smooth, i.e., smooth multivalued DNNFs. Note that smoothness can be ensured on Boolean circuits in quadratic time [34], and the same can be done on multivalued circuits.

Ranking functions. Our notion of ranking functions will give a score to each assignment, but to state their properties we define them on partial assignments. Formally, a *partial assignment* is a mapping $\nu : X \to D \cup \{\bot\}$, where $\bot$ is a fresh symbol representing *undefined*. We denote by $\overline{D^X}$ the set of partial assignments on domain D and variables X. The *support* $\mathsf{supp}(\nu)$ of ν is the subset of X on which ν is defined.

We extend the definitions of compatibility, of $\bowtie$, and of disjointness, to partial assignments in the following way. Two partial assignments $\tau \in \overline{D^Y}$ and $\sigma \in \overline{D^Z}$ are *compatible*, again written $\tau \simeq \sigma$, when for every $x \in Y \cap Z$, if $\tau(x) \neq \bot$ and $\sigma(x) \neq \bot$ then $\tau(x) = \sigma(x)$. In this case, we denote by $\tau \bowtie \sigma$ the partial assignment of $\overline{D^{Y \cup Z}}$ defined by: for $y \in Y \setminus Z$ we have $(\tau \bowtie \sigma)(y) := \tau(y)$, for $z \in Z \setminus Y$ we have $(\tau \bowtie \sigma)(z) := \sigma(z)$, and for $x \in Z \cap Y$, if $\tau(x) \neq \bot$ then $(\tau \bowtie \sigma)(x) = \tau(x)$, otherwise $(\tau \bowtie \sigma)(x) = \sigma(x)$. We call τ and σ *disjoint* if $Y \cap Z = \emptyset$; then again they are always compatible and we write $\tau \times \sigma$ for $\tau \bowtie \sigma$.

We then consider ranking functions defined on partial assignments $\overline{D^X}$, on which we will impose *subset-monotonicity*. Formally, a (D, X)-*ranking function* w is a function[1] $\overline{D^X} \to \mathbb{R}$ that gives a score to every partial assignment. Such a ranking function induces a *weak ordering*[2] $\preceq$ on $\overline{D^X}$, with $\mu \preceq \mu'$ defined as $w(\mu) \leq w(\mu')$. We always assume that ranking functions can be computed efficiently, i.e., with running time that only depends on X, not D.

By a slight notational abuse, we define the score $w(\tau)$ of partial assignment $\tau \in \overline{D^Y}$ with $Y \subseteq X$ by seeing τ as a partial assignment on X which is implicitly extended by assigning $\bot$ to every $z \in X \setminus Y$. Following earlier work [20, 36, 19], we then restrict our study to ranking functions that are *subset-monotone* [36]:

▶ **Definition 2.1.** *A (D, X)-ranking function $w : \overline{D^X} \to \mathbb{R}$ is* subset-monotone *if for every $Y \subseteq X$ and partial assignments $\tau_1, \tau_2 \in \overline{D^Y}$ such that $w(\tau_1) \leq w(\tau_2)$, for every partial assignment $\sigma \in \overline{D^{X \setminus Y}}$ (so disjoint with τ_1 and τ_2), we have $w(\sigma \times \tau_1) \leq w(\sigma \times \tau_2)$.*

We use in particular the following consequence of subset-monotonicity, where we call $\tau \in \overline{D^X}$ *maximal* (or maximum) for $w : \overline{D^X} \to \mathbb{R}$ when for every $\tau' \in \overline{D^X}$ we have $w(\tau') \leq \tau(\tau)$:

[1] As usual, when we write $\mathbb{R}$, we assume a suitable representation, e.g., as floating-point numbers.
[2] Recall that a weak ordering $\preceq$ on A is a total preorder on A, i.e., $\preceq$ is transitive and we have either $x \preceq y$ or $y \preceq x$ for every $x, y \in A$. In particular, it can be the case that two distinct elements x and y are tied, i.e., $x \preceq y$ and $y \preceq x$.

▶ **Lemma 2.2.** *Let $R \subseteq \overline{D^Y}$ and $S \subseteq \overline{D^Z}$ with $Y \cap Z = \emptyset$, and let $w : \overline{D^{Y \cup Z}} \to \mathbb{R}$ be subset-monotone. If τ is a maximal element of R and σ is a maximal element of S with respect to w, then $\tau \times \sigma$ is a maximal element of $R \wedge S$ with respect to w.*

We give a few examples of subset-monotone ranking functions. Let $W : X \times D \to \mathbb{R}$ be a function assigning scores to singleton assignments, and define the (D, X)-ranking function $\mathsf{sum}_W : \overline{D^X} \to \mathbb{R}$ by $\mathsf{sum}_W(\tau) = \sum_{x \in X, \tau(x) \neq \perp} W(x, \tau(x))$. Then sum_W is subset-monotone. Similarly define $\mathsf{max}_W : \overline{D^X} \to \mathbb{R}$ by $\mathsf{max}_W(\tau) = \max_{x \in X, \tau(x) \neq \perp} W(x, \tau(x))$, or prod_W in a similar manner (with non-negative scores for singletons); then these are again subset-monotone. In particular, we can use sum_W to encode lexicographic orderings on $\overline{D^X}$.

Enumeration and problem statement. Our goal in this article is to efficiently enumerate the satisfying assignments of circuits in nonincreasing order according to a ranking function. We will in particular apply this for the ranked enumeration of the answers to MSO queries on trees, as we will explain in Section 5. We call this problem RankEnum. Formally, the input to RankEnum consists of a multivalued circuit C on domain D and variables X, and a (D, X)-ranking function w that is subset-monotone. The output to enumerate consists of all of $\mathsf{rel}(C)$, without duplicates, in nonincreasing order of scores (with ties broken arbitrarily).

Formally, we work in the RAM model on words of logarithmic size [1], where memory cells can represent integers of value polynomial in the input length, and on which arithmetic operations take constant time. We will in particular allocate arrays of polynomial size in constant time, using lazy initialization [24]. We measure the performance of our algorithms in the framework of *enumeration algorithms*, where we distinguish two phases. First, in the *preprocessing phase*, the algorithm reads the input and builds internal data structures. We measure the running time of this phase as a function of the input; in general the best possible bound is *linear preprocessing*, e.g., preprocessing in $O(|C|)$. Second, in the *enumeration phase*, the algorithm produces the assignments, one after the other, without duplicates, and in nonincreasing order of scores; the order of assignments that are tied according to the ranking function is not specified. The *delay* is the maximal time that the enumeration phase can take to produce the next assignment, or to conclude that none are left. We measure the delay as a function of the input, as a function of the produced assignments (which each have size $|X|$), and also as a function of the number of results that have been produced so far. The best delay is *output-linear delay*, i.e., $O(|X|)$, which can be achieved for (non-ranked) enumeration of MSO queries on trees [7, 25, 3]. In our results, we will always fix $|X|$ to a constant (for technical reasons explained in the next section), so the corresponding bound would be *constant delay*, but, like [12], we will not be able to achieve it. Also note that the memory usage of the enumeration phase is not bounded by the delay, but can grow as enumeration progresses.

Brodal queues. Similar to [12], our algorithms in this paper will use priority queues, in a specific implementation called a *(functional) Brodal queue* [15]. Intuitively, Brodal queues are priority queues which support union operations in $O(1)$, and which are *purely functional* in the sense that operations return a queue without destroying the input queue(s). More precisely, a *Brodal queue* is a data structure which stores a set of priority-data pairs of the form $(\mathbf{p} : \text{foo}, \mathbf{d} : \text{bar})$ where foo is a real number and bar an arbitrary piece of data, supporting operations defined below. Brodal queues are *purely functional and persistent*, i.e., for any operation applied to some input Brodal queues, we obtain as output a new Brodal queue Q', such that the input queues can still be used. Note that the structures of Q' and of the input Brodal queues may be sharing locations in memory; this is in fact necessary, e.g., to guarantee constant-time bounds. However, this is done transparently, and both Q' and the input Brodal queues can be used afterwards[3]. Brodal queues support the following:

- *Initialize*, in time $O(1)$, which produces an empty queue;
- *Push*, in time $O(1)$, which adds to Q a priority-data pair;
- *Find-Max*, in time $O(1)$, which either indicates that Q is empty or otherwise returns some pair $(\mathbf{p} : \text{foo}, \mathbf{d} : \text{bar})$ with foo being maximal among the priority-data pairs stored in Q (ties are broken arbitrarily);
- *Pop-Max*, in time $O(\log(|Q|))$, which either indicates that Q is empty or returns two values: first the pair p returned by Find-Max, second a queue storing all the pairs of Q except p;
- *Union*, in time $O(1)$, which takes as input a second Brodal queue Q' and returns a queue over the elements of Q and Q'.

3 Ranked Enumeration for Smooth Multivalued DNNFs

In this section, we start the presentation of our technical results by giving our algorithm to solve the ranked enumeration problem for DNNFs under subset-monotone orders. This is Result 2 from the introduction, which we restate below:

▶ **Theorem 3.1.** *For any constant $n \in \mathbb{N}$, we can solve the* RankEnum *problem on an input smooth multivalued DNNF circuit C on domain D and variables X with $|X| = n$ and a subset-monotone (D, X)-ranking function with no preprocessing and with delay $O(|D| \times |C| + \log(K + 1))$, where K is the number of assignments produced so far.*

Note how the number n of variables is assumed to be constant in the result statement. This is for a technical reason: we will need to store partial assignments in memory, but in the RAM model we can only index polynomially many memory locations [24, page 3], so we must ensure that the total number of assignments is polynomial. The circuit itself and the domain can however be arbitrarily large, following the application to MSO queries over trees studied in Section 5: the variables of the circuit will be the variables of the MSO query (which is fixed because we will work in data complexity), and the size of the circuit and that of the domain will be linear in the size of the tree (which represents the data).

Our algorithm can be seen as an instance of the Lawler-Murty [26, 29] procedure, that has been previously used to enumerate paths in DAGs in decreasing order of weight in [36]. Interestingly, the result does not require that the input circuit is deterministic. However, it is less efficient than the method presented in Section 4 where determinism is exploited.

We prove Theorem 3.1 in the rest of this section. Let us fix a smooth multivalued DNNF C on domain D and variables X, and a subset-monotone ranking function $w \colon \overline{D^X} \to \mathbb{R}$. For a partial assignment τ, we denote by $w_C(\tau) = \max\{w(\tau \times \sigma) \mid \sigma \in D^{X \setminus \mathsf{supp}(\tau)} \text{ and } \tau \times \sigma \in \mathsf{rel}(C)\}$ the score of the maximal completion of τ to a satisfying assignment of C if it exists and $w_C(\tau) = \bot$ if no such completion exists. Our algorithm relies on the following folklore observation:

▶ **Lemma 3.2.** *Given a partial assignment τ, one can compute $w_C(\tau)$ in time $O(|C|)$.*

[3] This is similar to how persistent linked lists can be modified by removing the head element or concatenating with a new head element. Such operations can run in constant time and return the modified version of the list without invalidating the original list; with both lists sharing some memory locations in a transparent fashion.

Algorithm 1 Algorithm for Theorem 3.1.

Data: Smooth multivalued DNNF C with n variables, subset-monotone ranking function w.
Result: Enumeration of the satisfying assignments of C in nonincreasing order of scores by w.

$Q \leftarrow$ empty priority queue;
Push the empty assignment $[]$ into Q with priority $w_C([])$;
while *Q is not empty* **do**
  Pop into γ the assignment with maximum w_C-score from Q;
  for $j \leftarrow |\mathsf{supp}(\gamma)| + 1$ *to* n **do**
    foreach $d \in D$ **do**
      Construct $\alpha_d = \gamma \times \langle x_j : d \rangle$;
      Compute $w_C(\alpha_d)$ using Lemma 3.2;
    end
    $\gamma \leftarrow \alpha_{d_0}$ such that $w_C(\alpha_{d_0})$ is not $\perp$ and is maximal;
    Push into Q all $\alpha_{d'}$ for $d' \neq d_0$ where $w_C(\alpha_{d'}) \neq \perp$, with priority $w_C(\alpha_{d'})$;
  end
  Output γ;
end

Proof. Let X be the variables of C. It is enough to show that we can compute, given a smooth multivalued DNNF C' and monotone ranking function w', some $\sigma' \in \mathsf{rel}(C')$ that maximizes $w'(\sigma')$, in $O(|C'|)$. Indeed, if this is the case we can first compute the conditioning[4] C' of C on τ in time $O(|C'|)$: specifically, C' is a multivalued circuit on domain D and variables $X \setminus \mathsf{supp}(\tau)$ such that, for $\sigma' \in D^{X \setminus \mathsf{supp}(\tau)}$ we have that $\sigma' \in \mathsf{rel}(C')$ iff $\tau \times \sigma' \in \mathsf{rel}(C)$. Then, letting w' be the ranking function on $\overline{D^{X \setminus \mathsf{supp}(\tau)}}$ defined by $w'(\sigma') := w(\sigma' \times \tau)$ (which is subset-monotone), find one such $\sigma' \in \mathsf{rel}(C')$ in time $O(|C|)$, and then return $w(\sigma' \times \tau)$. This is correct thanks to subset-monotonicity of w, more precisely, by Lemma 2.2.

Now the algorithm to do this proceeds by bottom-up induction as follows: for each gate v of C', we compute $\sigma_v \in \mathsf{rel}(v)$ such that $w'(\sigma_v) = \max\{w'(\sigma) \mid \sigma \in \mathsf{rel}(v)\}$. If v is an input then $\mathsf{rel}(v)$ is a singleton assignment, and we let σ_v be this assignment. Now, if v is a $\times$-gate with inputs $v_1, \ldots, v_k$, we let $\sigma_v = \sigma_{v_1} \times \cdots \times \sigma_{v_k}$. By Lemma 2.2, σ_v is maximal for $\mathsf{rel}(v)$ if each σ_{v_i} is maximal for $\mathsf{rel}(v_i)$ which is the case by induction. Finally, if v is a $\cup$-gate with input $v_1, \ldots, v_k$, we define $\sigma_v = \arg\max_{i=1}^{k} w'(\sigma_{v_i})$, which is clearly maximal in $\mathsf{rel}(v) = \bigcup_{i=1}^{k} \mathsf{rel}(v_i)$ if σ_{v_i} is maximal in $\mathsf{rel}(v_i)$ for each i because v is smooth, which is the case by induction. ◀

With this in place, we are ready to describe the algorithm. Notice that our definition of multivalued circuits implies that $\mathsf{rel}(C)$ can never be empty, because all gates except input gates have inputs, and the circuit is decomposable. We fix an arbitrary order on $X = \{x_1, \ldots, x_n\}$ and, for $i \in \{1, \ldots, n+1\}$, we denote by $X_{<i}$ the set $\{x_1, \ldots, x_{i-1}\}$ (which is empty for $i = 1$). A partial assignment $\tau \in \overline{D^X}$ is called a *prefix assignment* if $\mathsf{supp}(\tau) = X_{<i}$ for some $i \in \{1, \ldots, n+1\}$.

[4] See [18, Definition 5.4] for the definition of conditioning on Boolean circuits, which easily adapts to multivalued circuits.

The enumeration algorithm is then illustrated as Algorithm 1, which we paraphrase in text below. The algorithm uses a variable γ holding a prefix assignment and a priority queue Q containing prefix assignments. The priorities in the queue are the w_C-score, i.e., the priority of each prefix assignment is the score returned by w_C on this assignment. We initialize Q to contain only the empty partial assignment (i.e., the assignment that maps every variable to $\perp$, denoted [] in Algorithm 1): note that the w_C-score of [] is not $\perp$ because $\mathsf{rel}(C) \neq \emptyset$. We then do the following until the queue is empty. We pop (i.e., call *Pop-Max*) from the queue a prefix assignment (of maximal w_C-score) that we assign to γ; we will inductively see that γ is a prefix assignment of $D^{<i}$ for some $i \in \{1, \ldots, n+1\}$ and that its w_C-score is not $\perp$. We then do the following for $j := i$ to n (i.e., potentially zero times, in case $i = n+1$ already). For every possible choice of domain element $d \in D$, we let α_d be the prefix assignment that extends γ by assigning x_i to d, and we compute the value $w_C(\alpha_d)$ using Lemma 3.2. Among these values, the definition of w_C ensures that one has a w_C-score which is not $\perp$, because this is true of γ. We thus pick a value $d_0 \in D$ such that $w_C(\alpha_{d_0})$ is maximal (in particular non-$\perp$). We set γ to α_{d_0}, and we push into Q all other prefix assignments $\alpha_{d'}$ for $d' \neq d_0$ for which we have $w_C(\alpha_{d'}) \neq \perp$. Once we have run this for all values of j, we have $i = n+1$, hence γ is a total assignment, and we output it. We then continue processing the remaining contents of the queue.

Correctness of the algorithm. We can show (see the full version [2]) that the following invariants hold at the beginning and end of every while loop iteration:

1. For every $\tau \in Q$, no satisfying assignment of C compatible with τ has been outputted so far;
2. For every $\tau, \tau' \in Q$, if $\tau \neq \tau'$ then $\tau \not\simeq \tau'$;
3. For every $\sigma \in \mathsf{rel}(C)$ that has not yet been outputted by the algorithm, there exists some $i \in \{1, \ldots, n+1\}$ such that $\sigma|_{X_{<i}} \in Q$ (in fact, the previous point then implies there is at most one such i);
4. The number of elements in Q is at most $n \times |D| \times (K+1)$, where K is the number of assignments produced so far.

We explain next why they imply correctness.

▷ Claim 3.3. Algorithm 1 terminates, enumerates $\mathsf{rel}(C)$ without duplicates and in non-increasing order, and runs with delay $O(|D| \times |C| + \log(K+1))$ with K the number of assignments produced so far.

Proof. We first show that the algorithm terminates. Indeed, notice that we pop a prefix assignment from the queue at the beginning of every while loop iteration. Let us show that, once a prefix assignment τ has been popped from Q, it cannot be pushed again into Q for the rest of the algorithm's execution. Indeed, observe that once we pop τ from Q, we will first push to Q assignments that are strict extensions of τ (hence different from τ), and then output a satisfying assignment τ' of C that is compatible with τ, after which the current iteration of the while loop ends. Now, by invariant (1), no partial assignment compatible with τ' can ever be added to Q, and in particular it is the case that τ cannot ever be added to Q. Thus the queue becomes empty and the algorithm terminates.

Since the queue eventually becomes empty, by invariant (3), the algorithm outputs at least all of $\mathsf{rel}(C)$. The fact that there are no duplicates follows from invariant (1), using a similar reasoning to how we proved termination. Furthermore, it is clear that only assignments of $\mathsf{rel}(C)$ are ever outputted. Therefore the algorithm indeed enumerates exactly all of $\mathsf{rel}(C)$ with no duplicates.

To check that assignments are enumerated in nonincreasing order, consider an iteration of the while loop where we output $\tau \in \mathsf{rel}(C)$. Let $\sigma \in \mathsf{rel}(C)$ be an assignment that has not yet been outputted, and assume by contradiction that $w(\tau) < w(\sigma)$. Consider the prefix assignment γ that was popped from the queue Q at the beginning of that iteration; clearly by construction we have $w_C(\gamma) = w(\tau)$. But by invariant (3), there exists a prefix assignment γ' in Q of which σ is a completion, hence for this γ' we have $w_C(\gamma') \geq w(\sigma)$ by definition of w_C, and this is strictly bigger than $w(\gamma)$, contradicting the fact that γ had maximal priority.

Last, we check that the delay between any two consecutive outputs is indeed $O(|D| \times |C| + \log(K+1))$. The $O(|D| \times |C|)$ term corresponds to the at most $n \times |D|$ applications of Lemma 3.2 during a for loop until we produce the next satisfying assignment (remember that n is constant so it is not reflected in the delay). The $O(\log(K+1))$ term corresponds to the unique pop operation performed on the priority queue during a while loop iteration. Indeed, by invariant (4) the queue contains less than $n \times |D| \times (K+1)$ prefix assignments and the complexity of a pop operation is logarithmic in this. Since n is constant we obtain $O(\log|D| + \log(K+1))$, and the $O(\log|D|)$ gets absorbed in the $O(|D| \times |C|)$ term. ◁

Thus, up to showing that the invariants hold (see the full version [2]), we have concluded the proof of Theorem 3.1.

4 Ranked Enumeration for Smooth Multivalued d-DNNFs

Having shown our polynomial-delay ranked enumeration algorithm for DNNF circuits, we move on in this section to our main technical contribution. Specifically, we present an algorithm for smooth multivalued DNNF circuits that are further assumed to be *deterministic*, but which achieves linear-time preprocessing and delay $O(\log(K+1))$, where K denotes the number of satisfying assignments produced so far. This proves Result 3, which we restate below:

▶ **Theorem 4.1.** *For any constant $n \in \mathbb{N}$, we can solve the* RankEnum *problem on an input smooth multivalued d-DNNF circuit C with n variables and a subset-monotone ranking function, with preprocessing $O(|C|)$ and delay $O(\log(K+1))$, where K is the number of assignments produced so far.*

Let us fix for this section the set X of variables of C (with $|X| = n$) and the domain D.

The rest of this section is devoted to proving Theorem 4.1. It is structured in three subsections, corresponding to the three main technical difficulties to overcome. First, we explain in Section 4.1 the preprocessing phase of the algorithm, where in particular we use Brodal queues to quickly "jump" over $\uplus$-gates. Second, in Section 4.2, we present a simple algorithm, that we call the $A \odot B$ *ranked enumeration algorithm*, which conveys in a self-contained fashion the idea of how we handle $\times$-gate during the enumeration phase of the main algorithm. Last, we present the enumeration phase in Section 4.3.

4.1 Preprocessing Phase

The preprocessing phase is itself subdivided in four steps, described next.

Preprocessing: first step. We preprocess C in $O(|C|)$ to ensure that the $\times$-gates of the circuit always have exactly two inputs. This can easily be done as follows. Remember that our definition of multivalued circuits does not allow $\times$-gates with no inputs, so this case does not occur. We can then eliminate $\times$-gates with one input by replacing them by their single input. Next, we can rewrite $\times$-gates with more than two inputs to replace them by a tree of $\times$-gates with two inputs. For simplicity, let us call C again the resulting smooth multivalued d-DNNF circuit in which $\times$-gates always have exactly two inputs.

Preprocessing: second step. We compute, for every gate g of C the value $\#g := |\mathsf{rel}(g)|$. This can clearly be done in linear time again, by a bottom-up traversal of C and using decomposability, determinism and smoothness. Note that $\#g$ has value at most $|D|^n$, which is polynomial (as n is a constant), so this fits into one memory cell.

Preprocessing: third step. The third step begins by initializing for every gate g of C an empty Brodal queue B_g. We then populate those queues by a (linear-time) bottom-up traversal of the circuit, described next. This traversal will add to each queue B_g some priority-data pairs of the form $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g', 1, \tau))$ where g' has a (possibly empty) directed path to g and $\tau \in \mathsf{rel}(g)$. We will shortly explain what is the exact content of these queues at the end of this third preprocessing step, but we already point out one invariant: once we are done processing a gate g in the traversal, then B_g contains at least one priority-data pair of this form, i.e., it is non-empty.

The traversal proceeds as follows:

- If g is an input gate labeled with $\langle x : d\rangle$ corresponding to the singleton assignment $\alpha = [x \mapsto d]$, then we push into B_g the priority-data pair corresponding to this assignment: $(\mathbf{p} : w(\alpha),\ \mathbf{d} : (g, 1, \alpha))$.
- If g is a $\times$-gate with inputs g_1 and g_2 then we call *Find-Max* on the Brodal queues B_{g_1} and B_{g_2} of the inputs. These gates g_1 and g_2 have already been processed, so the queues B_{g_1} and B_{g_2} are non-empty, and we obtain priority-data pairs $(\mathbf{p} : w(\tau_1),\ \mathbf{d} : (g_1', 1, \tau_1))$ and $(\mathbf{p} : w(\tau_2),\ \mathbf{d} : (g_2', 1, \tau_2))$, where $\tau_1 \in \mathsf{rel}(g_1)$ and $\tau_2 \in \mathsf{rel}(g_2)$. We push into B_g the pair $(\mathbf{p} : w(\tau_1 \times \tau_2),\ \mathbf{d} : (g, 1, \tau_1 \times \tau_2))$.
- If g is a $\uplus$-gate with input gates $g_1, \ldots, g_m$ then we set B_g to be the union of $B_{g_1}, \ldots, B_{g_m}$; recall that the union operation on two Brodal queues can be done in $O(1)$, so that this union is linear in m.

It is clear that this third preprocessing step takes time $O(|C|)$. To describe what the queues contain at the end of this step, we need to define the notion of *exit gate* of a $\uplus$-gate:

▶ **Definition 4.2.** *For a $\uplus$-gate g of C, an* exit gate of g *is a gate g' which is not a $\uplus$-gate (i.e., a $\times$-gate or an input of the circuit) such that there is a path from g' to g where every gate except g' on this path is a $\uplus$-gate. We denote by $\mathsf{exit}(g)$ the set of exit gates for g.*

We can then characterize what the queues contain:

▷ Claim 4.3. When the third preprocessing step finishes, the queues B_g are as follows:

- If g is an input gate corresponding to the singleton assignment $\alpha = [x \mapsto d]$ then B_g contains only the pair $(\mathbf{p} : w(\alpha),\ \mathbf{d} : (g, 1, \alpha))$.
- If g is a $\times$-gate then B_g contains only one pair, which is of the form $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g, 1, \tau))$ where τ is some satisfying assignment of g of maximal score (i.e., maximal among $\mathsf{rel}(g)$).
- If g is a $\uplus$-gate then B_g contains exactly the following: for every exit gate g' of g, the queue B_g contains one pair of the form $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g', 1, \tau))$ where τ is some satisfying assignment of g' of maximal score (i.e., maximal among $\mathsf{rel}(g')$).

This implies, in particular, that for every $g \in C$ the queue B_g contains a pair $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g', 1, \tau))$ (possibly $g' = g$) where τ is a satisfying assignment of g of maximal score among $\mathsf{rel}(g)$.

Proof of Claim 4.3. It is routine to prove this by bottom-up induction, in particular using Lemma 2.2 for the case of $\times$-gates. ◁

This concludes the third preprocessing step. Intuitively, the Brodal queues computed at this step will allow us to jump directly to the exits of $\uplus$-gates, without spending time traversing potentially long paths of $\uplus$-gates. Thanks to the constant-time union operation on Brodal queues, this third step takes linear time, and in fact this is the only part of the proof where we need this bound on the union operation. More precisely, in the remainder of the algorithm, we will only use on priority queues Q the operations *Initialize*, *Push* and *Find-Max* (in $O(1)$) and *Pop-Max* (in $O(\log |Q|)$).

Preprocessing: fourth step. In the fourth and last preprocessing step, we define some more data structures on every gate g of C.

First, we define for every gate g a priority queue Q_g. For all input gates and $\uplus$-gates, we simply set $Q_g := B_g$, but for $\times$-gates we will define Q_g to be new priority queues. Once this is done, we will only use the priority queues Q_g, and can forget about the priority queues B_g. We construct Q_g for each $\times$-gate g separately, in $O(1)$ time, as follows. Letting g_1 and g_2 be the inputs to g, we call Find-Max on B_g. By Claim 4.3, we obtain a pair $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g, 1, \tau))$ where τ is some satisfying assignment of g of maximal score. We split τ into $\tau_1 \times \tau_2$ where $\tau_i \in \mathsf{rel}(g_i)$ for $i \in \{1, 2\}$, and we define the priority queue Q_g to contain one priority-data pair, namely, $(\mathbf{p} : w(\tau),\ \mathbf{d} : (1, 1, \tau_1, \tau_2))$.

Second, we allocate for every gate g a table T_g of size $\#g$ (indexed starting from 1), that will later hold satisfying assignments of g in nonincreasing order of scores, stored into contiguous memory cells starting at the beginning of T_g. We do not bother initializing these tables, but we initialize integers i_g to 0, that will store the current number of assignments stored in T_g.

Last, we also initialize to 0 a bidimensional bit table R_g for every $\times$-gate g, of size $\#g_1 \times \#g_2$ with g_1, g_2 the two inputs of g. This can be done in $O(1)$ with the technique of *lazy initialization*, see e.g., [24, Section 2.5]. The role of these tables will be explained later.

This concludes the description of the preprocessing phase of our algorithm. In what follows, we will rely on the priority queues Q_g, the tables T_g, the integers i_g storing their size, and the tables R_g, The following should then be clear:

$\triangleright$ Claim 4.4. Once we finish the fourth preprocessing step (concluding the preprocessing), all integers i_g are 0, all tables T_g and R_g are empty, and the queues Q_g contain the following:

- If g is an input gate corresponding to the singleton assignment $\alpha = [x \mapsto d]$, then Q_g contains only the pair $(\mathbf{p} : w(\alpha),\ \mathbf{d} : (g, 1, \alpha))$.
- If g is a $\times$-gate with inputs g_1, g_2, then Q_g contains only one priority-data pair which is of the form $(\mathbf{p} : w(\tau_1 \times \tau_2),\ \mathbf{d} : (1, 1, \tau_1, \tau_2))$, where $\tau_1 \times \tau_2$ is some satisfying assignment of g of maximal score (among $\mathsf{rel}(g)$).
- If g is a $\uplus$-gate, then Q_g contains, for every exit gate g' of g, one pair of the form $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g', 1, \tau))$ where τ is some satisfying assignment of g' of maximal score (among $\mathsf{rel}(g')$).

Again, this in particular implies that each Q_g stores a satisfying assignment of g of maximal score (but the way in which it is represented depends on the type of g).

4.2 $A \odot B$ Ranked Enumeration Algorithm

Having described the preprocessing phase, we present in this section a component of the enumeration phase of our algorithm, called the $A \odot B$ *ranked enumeration* algorithm. This simple algorithm will be used at every $\times$-gate g during the enumeration phase to enumerate all ways to combine the assignments of the two inputs of g.

Algorithm 2 Algorithm for $A \odot B$ ranked enumeration.

Data: Two arrays A, B of real numbers of size n_1, n_2 (indexed from 1), sorted in nonincreasing order; An operation $\odot$ as described in the main text.
Result: An enumeration of the pairs $\{(i,j) \mid (i,j) \in \{1,\dots,n_1\} \times \{1,\dots,n_2\}\}$ in nonincreasing order of the score $A[i] \odot B[j]$

$R \leftarrow$ bidimensional array of size $n_1 \times n_2$ lazily initialized to 0;
$Q \leftarrow$ empty priority queue;
Push $(1,1)$ into Q with priority $A[1] \odot B[1]$;
$R[1,1] \leftarrow$ true;
while Q *is not empty* **do**
    Pop into (i,j) the pair with maximal priority from Q;
    Output (i,j);
    for $(p,q) \in \{(i+1,j),(i,j+1)\}$ **do**
        if $p \leq n_1$ *and* $q \leq n_2$ *and* $R[p][q] = 0$ **then**
            Push (p,q) into Q with priority $A[p] \odot B[q]$;
            $R[p][q] \leftarrow$ true;
        end
    end
end

Let $\odot : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be an operation which is computable in $O(1)$ and such that, for all $a \leq a'$ and $b \leq b'$ we have $a \odot b \leq a' \odot b'$ (this is similar to subset-monotonicity, and is in fact equivalent; cf. the full version [2]). We explain in this section how, given as input two tables (indexed starting from 1) A, B of reals of size n_1, n_2 sorted in nonincreasing order, we can enumerate the set of integer pairs $\{(i,j) \mid (i,j) \in \{1,\dots,n_1\} \times \{1,\dots,n_2\}\}$, in nonincreasing order of the score $A[i] \odot B[j]$, with $O(1)$ preprocessing and a delay $O(\log K)$ where K is the number of pairs outputted so far.

Intuitively, this will be applied at every $\times$-gate g, with $[n_1]$ (resp., $[n_2]$) representing the satisfying valuations of the first (resp., second) input of g sorted in a nonincreasing order, as in the table A (resp., B).

The algorithm is shown in Algorithm 2, but we also paraphrase it in text with more explanations. We initialize a two-dimensional bit table R of size $n_1 \times n_2$ to contain only zeroes (again using lazy initialization [24, Section 2.5]), whose role will be to remember which pairs have been seen so far, and a priority queue Q containing only the pair ($\mathbf{p}$: $A[1] \odot B[1]$, $\mathbf{d} : (1,1)$); we set $R[1,1]$ to true because the pair $(1,1)$ has been seen. Then, while the queue is not empty, we do the following. We pop (call *Pop-Max*) from Q, obtaining a priority-data pair of the form ($\mathbf{p} : A[i] \odot B[j]$, $\mathbf{d} : (i,j)$). We output the pair (i,j). Then, for each $(p,q) \in \{(i+1,j),(i,j+1)\}$ that is in the $[n_1] \times [n_2]$ grid, if the pair (p,q) has not been seen before, then we push into Q the pair ($\mathbf{p} : A[p] \odot B[q]$, $\mathbf{d} : (p,q)$) and mark (p,q) as seen in R. We show the following in the full version [2].

▷ Claim 4.5. This $A \odot B$ ranked enumeration algorithm is correct and runs with the stated complexity.

Proof sketch. The proof is simple and hinges on the following two invariants:

1. For any pair (i,j) not enumerated so far, there exists a pair (i',j') (possibly $(i,j) = (i',j')$) such that (i',j') is in Q, and a simple path in the $[n_1] \times [n_2]$ grid from (i',j') to (i,j) with nondecreasing first and second coordinates such that none of the pairs in that path have been outputted yet.
2. The queue contains at most $K+1$ pairs for K the number of pairs outputted so far. ◁

4.3 Enumeration Phase

We last move on to the enumeration phase. We first give a high-level description of how the enumeration phase works, before presenting the details.

The operation Get(g, j). We will define a recursive operation Get, running in complexity $O(\log(K + 1))$, that applies to a gate g and integer $1 \leq j \leq i_g + 1$ and does the following. If $j \leq i_g$ then $\mathsf{Get}(g, j)$ simply returns the satisfying assignment of g that is stored in $T_g[j]$ (i.e., this assignment has already been computed). Otherwise, if $j = i_g + 1$, then $\mathsf{Get}(g, j)$ finds the next assignment to be enumerated, inserts it into T_g, and returns that assignment. Note that, in this case, calling $\mathsf{Get}(g, j)$ modifies the memory for g and some other gates g'. Specifically, it modifies the tables $T_{g'}$ and $R_{g'}$, the queues $Q_{g'}$, and the integers $i_{g'}$ for various gates g' having a directed path to g (i.e., including $g' = g$).

When we are not executing an operation Get, the memory will satisfy the following invariants, for every g of C:

- The table T_g contains assignments $\tau \in \mathsf{rel}(g)$, ordered by nonincreasing score and with no duplicates; and i_g is the current size of T_g;
- For any assignment $\tau \in \mathsf{rel}(g)$ that does not occur in T_g, it is no larger than the last assignment in T_g, i.e., we have $w(\tau) \leq w(T_g[i_g])$.
- The queues Q_g will also satisfy some invariants, which will be presented later.
- The tables R_g for the $\times$-gates record whether we have already seen pairs of satisfying assignments of the two children, similarly to how this is done in the $A \odot B$ algorithm.

The tables T_g store the assignments in the order in which we find them, which is compatible with the ranking function. This allows us, in particular, to obtain in constant time the j-th satisfying assignment of $\mathsf{rel}(g)$ if it has already been computed, i.e., if $j \leq i_g$. The reason why we keep the assignments in the tables T_g is because we may reach the gate g via many different paths throughout the enumeration, and these paths may be at many different stages of the enumeration on g.

At the top level, if we can implement Get while satisfying the invariants above, then the enumeration phase of the algorithm is simple to describe: for j ranging from 1 to $\#r$, we output $\mathsf{Get}(r, j)$, where r is the output gate of C.

Implementing Get. We first explain the intended semantics of data values in the queues Q_b:

- If g is a $\uplus$-gate then Q_b will always contain pairs of the form $(\mathbf{p} : w(\tau),\ \mathbf{d} : (g', j, \tau))$ where $g' \in \mathsf{exit}(g)$ and $j \in \{1, \ldots, i_{g'} + 1\}$ and $\tau \in \mathsf{rel}(g')$, and the idea is that at the end of the enumeration τ will be stored at position j in $T_{g'}$.
- If g is a $\times$-gate, letting g'_1 and g'_2 be the input gates, then Q_b will always contain pairs of the form $(\mathbf{p} : w(\tau_1 \times \tau_2),\ \mathbf{d} : (j_1, j_2, \tau_1, \tau_2))$ with $\tau_i \in \mathsf{rel}(g_i)$ and at the end of the enumeration τ_i will be at position j_i in T_{g_i} with $j_i \in \{1, \ldots, i_{g'_i} + 1\}$ for all $i \in \{1, 2\}$.
- If g is an input gate, then Q_b initially contains the only assignment captured by g, becomes empty the first time we call $\mathsf{Get}(g, 1)$, and remains empty thereafter.

The implementation of Get is given in Algorithm 3. Intuitively, the algorithm for $\uplus$-gates simply consists of interleaving the maximal assignments of its exit gates, similarly to how one builds a sorted list for the union of two or more sorted lists. Here, determinism ensures that we do not get duplicates. The algorithm for $\times$-gates proceeds similarly to the $A \odot B$ algorithm, as explained in the previous section.

This concludes the presentation of the function Get, and with it that of the enumeration phase of the algorithm. The discussion of the delay bound can be found in the full version [2].

Algorithm 3 Implementation of $\mathsf{Get}(g, j)$ for the enumeration phase.

Data: The tables T_g, R_g, queues Q_g, integers $\#g, i_g$, ranking function w, a gate g, and integer $j \in \{1, \ldots i_g + 1\}$.
Result: The j-th satisfying assignment of g.

if $j \leq i_g$ **then return** $T_g[j]$;
// From now on, we have $j = i_g + 1$
if *g is an input gate* **then**
  $(\mathbf{p} : \delta,\ \mathbf{d} : (g, 1, \tau')) \leftarrow$ Pop from Q_j;
  $\tau \leftarrow \tau'$;
end
else if *g is a $\uplus$-gate* **then**
  $(\mathbf{p} : \delta,\ \mathbf{d} : (g', j', \tau')) \leftarrow$ Pop from Q_j;
  $\tau \leftarrow \tau'$;
  if $j' + 1 \leq \#g'$ **then**
    $\tau'' \leftarrow \mathsf{Get}(g', j' + 1)$;
    Push into Q_g the priority-data pair $(\mathbf{p} : w(\tau''),\ \mathbf{d} : (g', j' + 1, \tau''))$;
  end
end
else if *g is a $\times$ gate* **then**
  $(\mathbf{p} : \delta,\ \mathbf{d} : (j_1, j_2, \tau_1, \tau_2)) \leftarrow$ Pop from Q_j;
  $\tau \leftarrow \tau_1 \times \tau_2$;
  for $(p, q) \in \{(j_1 + 1, j_2), (j_1, j_2 + 1)\}$ **do**
    if *$p \leq \#g_1$ and $q \leq \#g_2$ and $R_g[p][q] = false$* **then**
      $\tau'_1 \leftarrow \mathsf{Get}(g_1, p)$;
      $\tau'_2 \leftarrow \mathsf{Get}(g_2, q)$;
      $\tau' \leftarrow \tau'_1 \times \tau'_2$;
      Push into Q_g the priority-data pair $(\mathbf{p} : w(\tau'),\ \mathbf{d} : (p, q, \tau'))$;
      $R_g[p][q] \leftarrow$ true;
    end
  end
end
$T_g[i_g + 1] \leftarrow \tau$;
$i_g \leftarrow i_g + 1$;
return τ

5 Application to Monadic Second-Order Queries

Having presented our results on ranked enumeration for smooth multivalued DNNFs and d-DNNFs, we present their consequences in this section for the problem of ranked enumeration of MSO query answers on trees. We first present some preliminaries on trees and MSO, formally define the evaluation problem, and explain how to reduce it to our results on circuits.

Trees and MSO on trees. We fix a finite set Λ of *tree labels*. A *Λ-tree* is then a tree T whose nodes carry a label from Λ, and which is rooted, ordered, binary, and full, i.e., every node has either no children (a *leaf*) or exactly one *left child* and one *right child* (an *internal node*). We often abuse notation and write T to refer to its set of nodes.

We consider *monadic second-order logic* (MSO) on trees, which extends first-order logic with quantification over sets. The signature of MSO on Λ-trees allows us to refer to the left child and right child relationships along with unary predicates referring to the node labels; and it can express, e.g., the set of descendants of a node. We only consider MSO queries where the free variables are first-order. We omit the precise semantics of MSO; see, e.g., [27].

Fixing an MSO query $\Phi(x_1, \dots, x_n)$ on Λ-trees, given a Λ-tree T, the *answers* of Φ on T are the assignments α on variables $X = \{x_1, \dots, x_n\}$ and domain T such that $\Phi(\alpha)$ holds on T in the usual sense. It is known that, for any such query Φ, given T and an assignment α from X to T, we can check whether $\Phi(\alpha(X))$ holds in linear time. What is more, given T, we can enumerate the answers of Φ on T with linear preprocessing and constant delay [7, 25, 3].

We now define *ranked enumeration.* For a tree T and variables $X = \{x_1, \dots, x_n\}$, a (T, X)-ranking function is simply a ranking function as in Section 2, whose domain is the set of nodes of T. We still assume that ranking functions are subset-monotone. The *ranked enumeration* problem for a fixed MSO query Φ with variables X, also denoted RankEnum, takes an input a tree T and a subset-monotone (T, X)-ranking function w, and must enumerate all answers of Φ on T, without duplicates, in nonincreasing order of scores (with ties broken arbitrarily).

Ranked enumeration for MSO. We are now ready to restate Result 1 from the introduction:

▶ **Theorem 5.1.** *For any fixed tree signature Λ and MSO query Φ on variables X on Λ-trees, given a Λ-tree T and a subset-monotone (T, X)-ranking function w, we can solve the* RankEnum *problem for Φ on T and w with preprocessing time $O(|T|)$ and delay $O(\log(K+1))$ where K is the number of answers produced so far.*

Recall that, as the total number of answers is at most $|T|^{|X|}$ and $|X|$ is constant, then this implies a delay bound of $O(\log |T|)$. The result is simply shown by constructing a smooth multivalued d-DNNF representing the query answers. This can be done in linear time with existing techniques (we provide a self-contained proof in the full version [2]):

▶ **Proposition 5.2** ([3, 5])**.** *For any fixed tree signature Λ and MSO query Φ on variables X on Λ-trees, given a Λ-tree T, we can check in time $O(|T|)$ if Φ has some answers on T, and if yes we can build in time $O(|T|)$ a smooth multivalued d-DNNF C on domain T and variables X such that* $\mathsf{rel}(C)$ *is precisely the set of answers of Φ on T.*

Note that we exclude the case where Φ has no answer on T, because our definition of multivalued circuits does not allow them to capture an empty set of assignments; of course we can do this check in the preprocessing, and if there are no answers then enumeration is trivial.

These results are intuitively shown by translating the MSO query to a tree automaton, and then computing a provenance circuit of this automaton by a kind of product construction [6]. The resulting circuit is a smooth multivalued DNNF, and is additionally a d-DNNF if the automaton is deterministic. We can then show Theorem 5.1 simply by performing the compilation (Proposition 5.2) as part of the preprocessing, and then invoking the enumeration algorithm of Section 4 (Theorem 4.1). Notice that we could also use the algorithm of Section 3 (Theorem 3.1), in particular if it is easier to obtain a nondeterministic tree automaton for the query, as its provenance circuit would then be a non-deterministic DNNF [5].

6 Conclusion

We have studied the problem of ranked enumeration for tractable circuit classes from knowledge compilation, namely, DNNFs and d-DNNFs, in the setting of multivalued circuits so as to apply these results to ranked enumeration for MSO query answers on trees. We have shown that the latter task can be solved with linear-time preprocessing and delay logarithmic in the number of answers produced so far, in particular logarithmic delay in the input tree in data complexity. This result on trees is the analogue of a previous result on words [12], achieving the same bounds but for a different notion of ranking functions.

We leave several questions open for future work. For instance, our efficient algorithms always assume that the input circuits are smooth: although this can be ensured "for free" in the setting of MSO on trees, it is generally quadratic to enforce on an arbitrary input circuit [34]. It may be possible to perform enumeration directly on non-smooth circuits, or on implicitly smoothed circuits, e.g., with special gates as in [3]. It would also be natural to study this problem in combined complexity, or for free second-order variables, though our algorithms cannot work on the RAM model if we need to store a superpolynomial number of assignments in memory. Last, it may be possible to extend our algorithms to more general ranking functions than the one we study, for instance by leveraging the framework of MSO cost functions used in [12], or using weighted logics [22], or possibly replacing subset-monotonicity by a weaker guarantee.

Last, it would be interesting to study whether our results can extend to the support of *updates*, e.g., reweighting updates to the ranking functions, or updates on the underlying circuits or (for MSO queries) on the tree, as in [28] or [5]. However, this is more difficult than the case of updates for non-ranked enumeration, because our algorithms use larger intermediate structures which are more challenging to maintain.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

2 Antoine Amarilli, Pierre Bourhis, Florent Capelli, and Mikaël Monet. Ranked enumeration for MSO on trees via knowledge compilation. *CoRR*, abs/2310.00731, 2023. Full version of this article with all proofs. `doi:10.48550/arXiv.2310.00731`.

3 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017. `doi:10.4230/LIPIcs.ICALP.2017.111`.

4 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019. `doi:10.4230/LIPIcs.ICDT.2019.22`.

5 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, 2019. `doi:10.1145/3294052.3319702`.

6 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances. In *ICALP*, 2015. `doi:10.1007/978-3-662-47666-6_5`.

7 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006. `doi:10.1007/11874683_11`.

8 Guillaume Bagan, Arnaud Durand, and Étienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007. `doi:10.1007/978-3-540-74915-8_18`.

9 Nurzhan Bakibayev, Tomáš Kociskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *PVLDB*, 6(14), 2013. `doi:10.14778/2556549.2556579`.

10 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under updates and in the presence of integrity constraints. In *ICDT*, 2018. `doi:10.4230/LIPIcs.ICDT.2018.8`.

11 Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. Pseudo polynomial-time top-k algorithms for d-dnnf circuits. *CoRR*, 2022. `doi:10.48550/arXiv.2202.05938`.

12 Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked enumeration of MSO logic on words. In *ICDT*, 2021. `doi:10.4230/LIPIcs.ICDT.2021.20`.

13 Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight fine-grained bounds for direct access on join queries. In *PODS*, 2022. `doi:10.1145/3517804.3526234`.

14 Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight fine-grained bounds for direct access on join queries. *CoRR*, abs/2201.02401, 2022. doi:10.48550/arXiv.2201.02401.

15 Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6), 1996. doi:10.1017/S095679680000201X.

16 Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. *TODS*, 46(2), 2021. doi:10.1145/3450263.

17 Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. *TODS*, 48(1), 2023. doi:10.1145/3578517.

18 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 2002. doi:10.1613/jair.989.

19 Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *arXiv preprint*, 2022. doi:10.48550/arXiv.2201.05566.

20 Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *arXiv preprint*, 2019. doi:10.48550/arXiv.1902.02698.

21 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. *Logical Methods in Computer Science*, 18, 2022. doi:10.46298/lmcs-18(1:21)2022.

22 Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In *ICALP*. Springer, 2005. doi:10.1007/11523468_42.

23 Arnaud Durand and Étienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *TOCL*, 8(4), 2007. doi:10.1145/1276920.1276923.

24 Étienne Grandjean and Louis Jachiet. Which arithmetic operations can be performed in constant time in the RAM model with addition? *arXiv preprint*, 2022. doi:10.48550/arXiv.2206.13851.

25 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013. doi:10.1145/2528928.

26 Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7), 1972.

27 Leonid Libkin. *Elements of finite model theory*. Springer, 2004. doi:10.1007/978-3-662-07003-1.

28 Katja Losemann and Wim Martens. MSO queries on trees: Enumerating answers under updates. In *CSL–LICS*, 2014. doi:10.1145/2603088.2603137.

29 Katta G Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations research*, 16(3), 1968. doi:10.1287/opre.16.3.682.

30 Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *TODS*, 40(1), 2015. doi:10.1145/2656335.

31 Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. *JACM*, 69(3), 2022. doi:10.1145/3517035.

32 Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In *STACS*, 2014. URL: https://hal.inria.fr/hal-01070893/document, doi:10.4230/LIPIcs.STACS.2014.13.

33 Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *ICDT*, 2017. doi:10.4230/LIPIcs.ICDT.2017.20.

34 Andy Shih, Guy Van den Broeck, Paul Beame, and Antoine Amarilli. Smoothing structured decomposable circuits. In *NeurIPS*, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/940392f5f32a7ade1cc201767cf83e31-Abstract.html.

35 Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *PVLDB*, 13(9), 2020. doi:10.14778/3397230.3397250.

36 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries. *arXiv preprint*, 2022. doi:10.48550/arXiv.2205.05649.

37 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, 2016. doi:10.48550/arXiv.1605.05102.