Ranked Enumeration of MSO Logic on Words

Pierre Bourhis ✉
CNRS Lille, CRIStAL UMR 9189, University of Lille, INRIA Lille, France

Alejandro Grez ✉
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Louis Jachiet ✉
LTCI, IP Paris, France

Cristian Riveros ✉
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Abstract

In the last years, enumeration algorithms with bounded delay have attracted a lot of attention for several data management tasks. Given a query and the data, the task is to preprocess the data and then enumerate all the answers to the query one by one and without repetitions. This enumeration scheme is typically useful when the solutions are treated on the fly or when we want to stop the enumeration once the pertinent solutions have been found. However, with the current schemes, there is no restriction on the order how the solutions are given and this order usually depends on the techniques used and not on the relevance for the user.

In this paper we study the enumeration of monadic second order logic (MSO) over words when the solutions are ranked. We present a framework based on *MSO cost functions* that allows to express MSO formulae on words with a cost associated with each solution. We then demonstrate the generality of our framework which subsumes, for instance, document spanners and adds ranking to them. The main technical result of the paper is an algorithm for enumerating all the solutions of formulae in increasing order of cost efficiently, namely, with a linear preprocessing phase and logarithmic delay between solutions. The novelty of this algorithm is based on using functional data structures, in particular, by extending functional Brodal queues to suit with the ranked enumeration of MSO on words.

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management; Theory of computation → Complexity theory and logic; Theory of computation → Formal languages and automata theory

Keywords and phrases Persistent data structures, Query evaluation, Enumeration algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2021.20

Funding *Pierre Bourhis*: Partially funded by the DeLTA project (ANR-16-CE40-0007).
Alejandro Grez: Partially funded by the Millennium Institute for Foundational Research on Data.
Cristian Riveros: Partially funded by the Millennium Institute for Foundational Research on Data.

1 Introduction

Managing and querying word structures such as texts has been one of the classical problems of different communities in computer science. In particular, this problem has been predominant in information extraction where the goal is to extract some subparts of a text. A logical approach that has brought a lot of attention in the database community is document spanners [13]. This logical framework provides a language for extracting subparts of a document. More specifically, regular spanners are based on regular expressions that fill relations with tuples of the texts' subparts. These relations can afterwards be queried by conjunctive or datalog-like queries.

© Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros;
licensed under Creative Commons License CC-BY 4.0

The document spanners' main algorithmic problem is the efficient evaluation of a spanner over a word. Recently, a novel approach has been to focus on the enumeration problem to obtain efficient evaluation algorithms. The principle of an enumeration algorithm is to create a representation of the set of answers efficiently depending only on the input word's size and the query, and not in the number of answers. This time is called *the preprocessing time*. The second part of an enumeration algorithm is to enumerate the outputs one by one using the previous representation. The time between two consecutive outputs is called the *delay*. As for the preprocessing time, an efficient delay should not depend on the number of outputs, but only on the input size (i.e., word and query). In general, the most efficient enumeration algorithms have linear preprocessing time and constant delay, both in data complexity (i.e. in the size of the input word).

Several people have studied the enumeration problem over words following different formalisms. For example, [3, 6, 23] studied the enumeration problem for MSO logic, [14, 2] for regular spanners (i.e., automata), and [18] for streaming evaluation in complex event processing. For all these formalisms, it is shown that there exists an enumeration algorithm with linear time preprocessing and delay constant both in data complexity (i.e. in the size of the input word).

The interest of an efficient enumeration algorithm is to provide a process that can quickly give the first answers. Unfortunately, these answers may not be relevant for the user; that is, the enumeration process does not assume how the output will be ordered. A classical manner of considering the user's preferences is to associate a score to each solution and then rank them following this score; for instance one could ask for the matches ranked by order of length, or by the number of times a second pattern appears within the match. This approach of "scoring" solutions has been used particularly in the context of information extraction. Indeed, there have been several recent proposals [9, 8] to extend document spanners with annotations from a semi-ring. The proposed annotations are typically useful to capture the confidence of each solution [9]. For instance, [8] proves that the enumeration of the answers following their scores' order is possible with polynomial-time preprocessing and polynomial delay.

In this paper, we are interested in establishing a framework for scoring outputs and improve the bounds proved in [9]. We propose using what we called MSO cost functions, which are formulas in weighted logics [11] extended with open variables. These formulas provide a simple formalism for defining the output and scoring with MSO logic. We show that one can translate each MSO cost function to a cost transducer. These machines are a restricted form of weighted functional vset-automaton [9], for which there exists at most one run for any word and any valuation. We use cost transducers to study the ranked enumeration problem: enumerate all outputs in increasing rank order. Specifically, the main result of the paper is an algorithm for enumerating all the solutions of a cost transducer in increasing order efficiently; specifically, with a preprocessing phase linear in the input word and a logarithmic delay between solutions.

Our approach generalizes an algorithm for enumerating solutions proposed in [18, 14]. The preprocessing part builds a heap containing the answers with their score, and one step of the enumeration is simply a pop of the heap. For this, we use a general data structure that we called Heap of Words (HoW), having the classical heap operations of finding/deleting the minimal element, adding an element, and melding two heaps. We also need to add two new operations that allow us to concatenate a letter to and increase the score of all elements of the heap. Finally, we require that this structure is fully-persistent [10], i.e., that each of the previous operations returns a new heap without changing the previous one. To obtain the

required efficiency, we rely on a classical persistent data structure called Brodal queue that we extend in order to capture the new operations over the stored words and scores presented above. We call this extension an incremental Brodal queue.

Finally, for ranked query evaluation, there has been recent progress in the context of conjunctive queries: on the efficient computation of top-k queries [25] and the efficient ranked enumeration [24, 7]. These advances consider relational data (which is more general than words) and conjunctive queries (which is more restricted than MSO queries); they are thus incomparable to our work. However, it is important to note some similarities with our work, such as the need for an "advanced" priority queue (the Fibonacci heap [7]), which means that our incremental queues might be of great interest there.

Contributions. The contributions of this paper are threefold: **(i)** we introduce MSO cost functions, a framework to express MSO queries and scores, generalizing the proposals of document spanners; **(ii)** we give a ranked enumeration scheme that has linear preprocessing time and logarithmic delay in data complexity with a polynomial combined complexity; **(iii)** we introduce two new data structures for our scheme: the Heaps of Words and the incremental Brodal queues. Both of these structures might be of interest in other ranked enumerations schemes.

Organization. Section 2 introduces the ranked enumeration problem for MSO queries on words. Section 3 presents MSO cost functions and state the main result. In Section 4 we show an application of the main result to the setting of document spanners. Section 5 describes our enumeration scheme that rely on two data structures: the Heap of Words described in Section 6 and the incremental Brodal queues presented in Section 7. We finish with some conclusions in Section 8.

2 Preliminaries

Words. We denote by Σ a finite alphabet, Σ^* all words over Σ, and ϵ the empty-word of 0 length. Give a word $w = a_1 \dots a_n$, we write $w[i] = a_i$. For two words $u, v \in \Sigma^*$ we write $u \cdot v$ as the concatenation of u and v. We denote by $[n] = \{1, \dots, n\}$.

Ordered groups. A group is a pair $(\mathbb{G}, \oplus, \mathbb{0})$ where $\mathbb{G}$ is a set of elements, $\oplus$ is a binary operation over $\mathbb{G}$ that is associative, $\mathbb{0} \in \mathbb{G}$ is a neutral element for $\oplus$ (i.e., $\mathbb{0} \oplus g = g \oplus \mathbb{0} = g$) and every $g \in \mathbb{G}$ has an inverse with respect to $\oplus$ (i.e., $g \oplus g^{-1} = \mathbb{0}$ for some $g^{-1} \in \mathbb{G}$). A group is abelian if, in addition, $\oplus$ is commutative (i.e., $g_1 \oplus g_2 = g_2 \oplus g_1$). From now on, we assume that all groups are abelian. We say that $(\mathbb{G}, \oplus, \mathbb{0}, \preceq)$ is an ordered group if $(\mathbb{G}, \oplus, \mathbb{0})$ is a group and $\preceq$ is a total order over $\mathbb{G}$ that respects $\oplus$, namely, if $g_1 \preceq g_2$ then $g_1 \oplus g \preceq g_2 \oplus g$ for every $g, g_1, g_2 \in \mathbb{G}$. Examples of (abelian) ordered groups are $(\mathbb{Z}, +, 0, \leq)$ and $(\mathbb{Z}^k, +, (0, \dots, 0), \leq_k)$ where $\leq_k$ represents the lexicographic order over $\mathbb{Z}^k$.

MSO. We use monadic second-order logic for defining properties over words. As usual, we encode words as logical structures with an order predicate and unary predicates to represent the order and the letters of each positions of the word, respectively. More formally, fix an alphabet Σ and let $w \in \Sigma^*$ be a word of length n. We encode w as a structure $([n], \leq, (P_a)_{a \in \Sigma})$ where $[n]$ is the domain, $\leq$ is the total order over $[n]$, and $P_a = \{i \mid w[i] = a\}$. By some abuse of notation, we also use w to denote its corresponding logical structure.

A MSO-formula φ over Σ is given by:

$$\varphi \quad := \quad x \leq y \;\mid\; P_a(x) \;\mid\; x \in X \;\mid\; \varphi \wedge \varphi \;\mid\; \neg\varphi \;\mid\; \exists x.\, \varphi \;\mid\; \exists X.\, \varphi$$

where $a \in \Sigma$, x and y are first-order (FO) variables, and X is a monadic second order (MSO) variable (i.e., a set variable). We write $\varphi(\bar{x}, \bar{X})$ where $\bar{x}$ and $\bar{X}$ are the sets of free FO and MSO variables of φ, respectively. An assignment σ for w is a function $\sigma : \bar{x} \cup \bar{X} \to 2^{[n]}$ such that $|\sigma(x)| = 1$ for every $x \in \bar{x}$ (note that we treat FO variables as a special case of MSO variables). As usual, we denote by $\mathrm{dom}(\sigma) = \bar{x} \cup \bar{X}$ the domain of the function σ. Then we write $(w, \sigma) \models \varphi$ when σ is an assignment over w, $\mathrm{dom}(\sigma) = \bar{x} \cup \bar{X}$, and w satisfies $\varphi(\bar{x}, \bar{X})$ when each variable in $\bar{x} \cup \bar{X}$ is instantiated by σ (see [20]). Given a formula $\varphi(\bar{x}, \bar{X})$, we define $[\![\varphi]\!](w) \;=\; \{\sigma \mid (w, \sigma) \models \varphi(\bar{x}, \bar{X})\}$. For the sake of simplification, from now on we will only use $\bar{X}$ to denote the free variables of $\varphi(\bar{X})$ and use $X \in \bar{X}$ for an FO or MSO variable.

For any assignment σ over w, we define the support of σ, denoted by $\mathrm{supp}(\sigma)$, as the set of positions mentioned in σ; formally, $\mathrm{supp}(\sigma) = \{i \mid \exists v \in \mathrm{dom}(\sigma).\, i \in \sigma(v)\}$. Furthermore, we encode assignments as sequences over the support as follows. Let $\mathrm{supp}(\sigma) = \{i_1, \ldots, i_m\}$ such that $i_j < i_{j+1}$ for every $j < m$. Then we define the (word) encoding of σ as:

$$\mathrm{enc}(\sigma) \;=\; (\bar{X}_1, i_1)(\bar{X}_2, i_2)\ldots(\bar{X}_m, i_m)$$

such that $\bar{X}_j = \{X \in \mathrm{dom}(\sigma) \mid i_j \in \sigma(X)\}$ for every $j \leq m$. That is, we represent σ as an increasing sequence of positions, where each position is labeled with the variables of σ where it belongs. This is the standard encoding used to represent assignments for running algorithms regarding MSO formulas [3, 6]. Finally, we define the size of σ as $|\mathrm{enc}(\sigma)| = |\mathrm{dom}(\sigma)| \cdot m$.

Enumeration algorithms. Given a formula φ and a word w, the main goal of the paper is to study the enumeration of assignments in $[\![\varphi]\!](w)$. We give here a general definition of enumeration algorithm and how we measure its delay. Later we use this to define the ranked enumeration problem of MSO.

As it is standard in the literature [3, 6, 23], we consider algorithms on Random Access Machines (RAM) with uniform cost measure [1] equipped with addition and subtraction as basic operations. A RAM has read-only input registers (containing the input I), read-write work memory registers and write-only output registers. We assume that group elements can be stored in constant space and that all group-related operations (i.e., to evaluate $g_1 \oplus g_2$, $g_1 \preceq g_2$ and g^{-1}) take constant time. We say that an algorithm $\mathcal{E}$ is an enumeration algorithm for MSO evaluation if $\mathcal{E}$ runs in two phases, for every MSO-formula φ and a word w.

1. The first phase, called the preprocessing phase, does not produce output, but may prepare data structures for use in the next phase.
2. The second phase, called the enumeration phase, occurs immediately after the precomputation phase. During this phase, the algorithm:
   - writes $\#\,\mathrm{enc}(\sigma_1)\#\,\mathrm{enc}(\sigma_2)\#\ldots\#\,\mathrm{enc}(\sigma_k)\#$ to the output registers where # is a distinct separator symbol, and σ_1, …, σ_k is an enumeration (without repetition) of the assignments of $[\![\varphi]\!](w)$;
   - it writes the first # as soon as the enumeration phase starts,
   - it stops immediately after writing the last #.

The separation of $\mathcal{E}$'s operation into a preprocessing and enumeration phase is done to be able to make an output-sensitive analysis of $\mathcal{E}$'s complexity. Formally, we say that $\mathcal{E}$ has preprocessing time $f : \mathbb{N}^2 \to \mathbb{N}$ if there exists a constant C such that the number of instructions that $\mathcal{E}$ executes during the preprocessing phase on input (φ, w) is at most $C \times f(|\varphi|, |w|)$ for every MSO-formula φ and word w. Furthermore, we measure the delay as follows. Let

$\text{time}_i(\varphi, w)$ denote the time in the enumeration phase when the algorithm writes the i-th # (if it exists) when running on input (φ, w). Define $\text{delay}_i(\varphi, w) = \text{time}_{i+1}(\varphi, w) - \text{time}_i(\varphi, w)$. Further, let $\text{output}_i(\varphi, w)$ denote the i-th element that is output by $\mathcal{E}$ when running on input (φ, w), if it exists. We say that $\mathcal{E}$ has delay $g : \mathbb{N}^2 \to \mathbb{N}$ if there exists a constant D such that, for all φ and w, it holds that:

$$\text{delay}_i(\varphi, w) \;\leq\; D \times |\text{output}_i(\varphi, w)| \times g(|\varphi|, |w|)$$

for every $i \leq |[\![\varphi]\!](w)|$. Furthermore, if $[\![\varphi]\!](w)$ is empty, then $\text{delay}_1(\varphi, w) \leq k$, namely, it ends in constant time. Finally, we say that $\mathcal{E}$ has preprocessing time $f : \mathbb{N} \to \mathbb{N}$ and delay $g : \mathbb{N} \to \mathbb{N}$ in *data complexity*, if there exists a function $c : \mathbb{N} \to \mathbb{N}$ such that $\mathcal{E}$ has preprocessing time $c(|\varphi|) \times f(|w|)$ and has delay $c(|\varphi|) \times g(|w|)$ (i.e., f and g describe the complexity once φ is considered as fixed).

It is important to notice that, although we fix a particular encoding for assignments and we restrict the enumeration algorithms to this encoding, we can use any encoding for the assignments whenever there exists a linear transformation between $\text{enc}(\cdot)$ and the new encoding. Given the definition of delay, if we use an encoding $\text{enc}'(\sigma)$ for σ, and there exists a linear time transformation between $\text{enc}(\sigma)$ and $\text{enc}'(\sigma)$ for every σ, then the same enumeration algorithm works for $\text{enc}'(\cdot)$. In particular, whenever the encoding depends linearly over $\text{supp}(\sigma)$ and $|\bar{x} \cup \bar{X}|$, then the aforementioned property holds.

Ranked enumeration. For an MSO formula φ and $w \in \Sigma^*$, we consider the ranked enumeration of the set $[\![\varphi]\!](w)$. For this, we need to assign an order to the outputs and we do this by mapping each element to a total order set. Fix a set C with a total order $\preceq$ over C. A cost function is any partial function κ that maps words $w \in \Sigma^*$ and assignments σ to elements in C. Without loss of generality, we assume that κ is defined only over pairs (w, σ) such that σ is an assignment over w.

Let φ be an MSO formula and κ a cost function over $(C, \preceq)$. We define the ranked enumeration problem of (φ, κ) as

Problem:	RANK-ENUM$[\varphi, \kappa]$
Input:	A word $w \in \Sigma^*$.
Output:	Enumerate all $\sigma_1, \ldots, \sigma_k \in [\![\varphi]\!](w)$ without repetitions and such that $\kappa(w, \sigma_i) \preceq \kappa(w, \sigma_{i+1})$.

Note that we consider the version of the problem in data-complexity where φ and κ are fixed. We say that RANK-ENUM$[\varphi, \kappa]$ can be solved with preprocessing time $f(n)$ and delay $g(n)$ if there exists an enumeration algorithm $\mathcal{E}$ that runs with preprocessing time $f(n)$ and delay $g(n)$ and, for every $w \in \Sigma^*$, $\mathcal{E}$ enumerates $[\![\varphi]\!](w)$ in increasing ordered according to κ. In the next section, we give a language to define cost functions and we state our main result.

3 MSO cost functions

To state our main result about ranked enumeration of MSO, first we need to choose a formalism to define cost functions. We do this by staying in the same setting of MSO logic by considering weighted logics over words [11, 12, 19]. Functions defined by extensions of MSO has been studied by using weighted automata, but also people have found it counterparts by extending MSO with a semiring. We use here a fragment of weighted MSO parametrized by an ordered group to fit our purpose.

Fix an ordered group $(\mathbb{G}, \oplus, \mathbb{0}, \preceq)$. A weighted MSO-formula α over Σ and $\mathbb{G}$ is given by the following syntax:

$$\alpha \ := \ [\varphi \mapsto g] \ \mid \ \alpha \oplus \alpha \ \mid \ \Sigma\, x.\, \alpha$$

where φ is an MSO-formula, $g \in \mathbb{G}$, and x is an FO variable. Further, we assume that the $\Sigma\, x$ quantifier cannot be nested. For example, $(\Sigma\, x.[\varphi \mapsto g]) \oplus (\Sigma\, y.[\varphi' \mapsto g'])$ is a valid formula but $\Sigma\, x.\Sigma\, y.[\varphi \mapsto g]$ is not. Similar than for MSO formulas, we write $\alpha(\bar{x}, \bar{X})$ to state explicitly the sets of FO-variables $\bar{x}$ and of MSO variables $\bar{X}$ that are free in α.

Let σ be an assignment. For any FO-variable x and $i \in \mathbb{N}$ we denote by $\sigma[x \to i]$ the extension of σ with x assigned to i, namely, $\mathrm{dom}(\sigma[x \to i]) = \{x\} \cup \mathrm{dom}(\sigma)$ such that $\sigma[x \to i](x) = \{i\}$ and $\sigma[x \to i](y) = \sigma(y)$ for every $y \in \mathrm{dom}(\sigma) \setminus \{x\}$. We define the semantics of a weighted MSO formula α as a function from words and assignments to elements in $\mathbb{G}$. Formally, for every $w \in \Sigma$ and every assignment σ over w we define the output $[\![\alpha]\!](w, \sigma)$ recursively as follows:

$$[\![[\varphi \mapsto g]]\!](w,\sigma) = \begin{cases} g & \text{if } (w,\sigma) \models \varphi \\ \mathbb{0} & \text{otherwise.} \end{cases} \qquad \begin{aligned} [\![\alpha \oplus \alpha']\!](w,\sigma) &= [\![\alpha]\!](w,\sigma) \oplus [\![\alpha']\!](w,\sigma) \\ [\![\Sigma\, x.\, \alpha]\!](w,\sigma) &= \bigoplus_{i=1}^{|w|} [\![\alpha]\!](w, \sigma[x \to i]) \end{aligned}$$

where φ is any MSO-formula, α and α' are weighted MSO formulas, and $g \in \mathbb{G}$. By some abuse of notation, in the following we will not make distinction between α and $[\![\alpha]\!]$, that is, the cost function over $\mathbb{G}$ defined by α.

▶ **Example 1.** Consider the alphabet $\{a, b\}$ and suppose that we want to define a cost function that counts the number of a-letters between two variables x and y. This can be defined in weighted MSO over $\mathbb{Z}$ as follows:

$$\alpha_1 \ := \ \Sigma\, z.\big[(x \leq z \wedge z \leq y \wedge P_a(z)) \mapsto 1\big]$$

Here, α_1 uses z to count over all positions of the word and we count 1 whenever z is labeled with a and is between x and y, and we count 0, otherwise, which is the identity of $\mathbb{Z}$.

▶ **Example 2.** Consider again the alphabet $\{a, b\}$ and suppose that we want a cost function to compare assignments over variables (x, y) lexicographically. For this, we can write a weighted MSO-formula over $\mathbb{Z}^2$ that maps each assignment σ over x and y to a pair $(\sigma(x), \sigma(y))$. This can be defined in weighted MSO over $\mathbb{Z}^2$ as follows:

$$\alpha_2 \ := \ \big(\Sigma\, z_1.\, \big[(z_1 \leq x) \mapsto (1,0)\big]\big) + \big(\Sigma\, z_2.\, \big[(z_2 \leq y) \mapsto (0,1)\big]\big)$$

Similar than for the previous example, we use the Σ-quantifier to add in the first and second component the value of x and y, respectively. In fact, for every assignment $\sigma = \{x \to i, y \to j\}$ over $w \in \Sigma^*$ it holds that $[\![\alpha_2]\!](w, \sigma) = (i, j)$.

Strictly speaking, the syntax and semantics of weighted MSO defined above is a restricted version of weighted logics [11], in the sense that weighted logics is usually defined over a semiring, which has two binary operations $\oplus$ and $\odot$. Indeed, it will be interesting to have a better understanding of the expressibility of MSO cost functions, or to extend our results for weighted logics over semiring. We leave this for future work.

We are ready to state the main result of the paper about ranked enumeration of MSO.

▶ **Theorem 3.** *Fix an alphabet Σ and an ordered group $\mathbb{G}$. For every MSO formula φ over Σ and every weighted MSO formula α over Σ and $\mathbb{G}$ the problem RANK-ENUM$[\varphi, \alpha]$ can be solved with linear preprocessing time and logarithmic delay.*

As it is common for MSO logic over words, we prove this result by developing an enumeration algorithm using automata theory. Specifically, we define a weighted automata model, that we called cost transducer, and show that its expressiveness is equivalent to the combination of (boolean) MSO and weighted MSO logic.

From now on, fix an input alphabet Σ and an output alphabet Γ. Furthermore, fix an ordered group $(\mathbb{G}, \oplus, \mathbb{0}, \preceq)$. A *cost transducer* over $\mathbb{G}$ is a tuple $\mathcal{T} = (Q, \Delta, \kappa, I, F)$, where Q is the set of states, $\Delta \subseteq Q \times \Sigma \times 2^\Gamma \times Q$ is the transition relation, $\kappa : \Delta \to \mathbb{G}$ is a function that associates a cost to every transition of Δ, and $I : Q \to \mathbb{G}$, $F : Q \to \mathbb{G}$ are partial functions that associate a cost in $\mathbb{G}$ to (some) states in Q. The functions I and F are partial functions because they naturally define the set of initial and final states as $\text{dom}(I)$ and $\text{dom}(F)$, respectively. A run of $\mathcal{T}$ over a word $w = a_1 a_2 \ldots a_n$ is a sequence of transitions $\rho : q_0 \xrightarrow{a_1/\bar{X}_1} q_1 \xrightarrow{a_2/\bar{X}_2} \ldots \xrightarrow{a_n/\bar{X}_n} q_n$ such that $q_0 \in \text{dom}(I)$ and $(q_{i-1}, a_i, \bar{X}_i, q_i) \in \Delta$ for every $i \leq n$. We say that ρ is accepting if $q_n \in \text{dom}(F)$.

For a run ρ as defined above, let $\{i_1, \ldots, i_m\} \subseteq [n]$ be all the positions of ρ such that $\bar{X}_{i_j} \neq \emptyset$ and $i_j < i_{j+1}$ for all $j \leq m$. Then we define the output of ρ as the sequence:

$$\text{out}(\rho) \;=\; (\bar{X}_{i_1}, i_1)(\bar{X}_{i_2}, i_2) \ldots (\bar{X}_{i_m}, i_m)$$

Moreover, we extend κ over accepting runs ρ by adding the costs of all transitions of ρ plus the initial and final cost, namely:

$$\kappa(\rho) = I(q_0) \oplus \bigoplus_{i=1}^{|w|} \kappa((q_{i-1}, a_i, \bar{X}_i, q_i)) \oplus F(q_n).$$

Note that $\text{out}(\rho)$ defines the encoding of some assignment σ over w with $\text{dom}(\sigma) = \Gamma$ and $\text{out}(\rho) = \text{enc}(\sigma)$. Of course, the opposite direction is not true: for some assignment σ there could be no run ρ that defines σ and, moreover, there could be two runs ρ_1 and ρ_2 such that $\text{out}(\rho_1) = \text{out}(\rho_2) = \text{enc}(\sigma)$, but $\kappa(\rho_1) \neq \kappa(\rho_2)$. For this reason, we impose an additional restriction to cost transducers: we assume that all cost transducers in this paper are unambiguous, that is, for every $w \in \Sigma^*$ there does not exist two distinct runs ρ_1 and ρ_2 of w such that $\text{out}(\rho_1) = \text{out}(\rho_2)$. In other words, a cost transducers satisfies that for every $w \in \Sigma^*$ and assignment σ there exists at most one run ρ such that $\text{out}(\rho) = \text{enc}(\sigma)$.

Given the unambiguous restriction of cost transducers, we can define a partial function from pairs (w, σ) to $\mathbb{G}$ as $\text{cost}_\mathcal{T}(w, \sigma) = \kappa(\rho)$ whenever there exists a run ρ of w such that $\text{out}(\rho) = \text{enc}(\sigma)$. Otherwise $\text{cost}_\mathcal{T}(w, \sigma)$ is not defined. Given that for some pairs (w, σ) the function $\text{cost}_\mathcal{T}$ is not defined, we can define the set $[\![\mathcal{T}]\!](w) = \{\sigma \mid \text{cost}_\mathcal{T}(w, \sigma) \text{ is defined}\}$ of all outputs of $\mathcal{T}$ over w.

It is important to notice that, given $w \in \Sigma^*$, a cost transducer $\mathcal{T}$ is in charge of (1) defining the set of assignments $[\![\mathcal{T}]\!](w)$ and (2) assigning a cost $[\![\mathcal{T}]\!](w, \sigma)$ for each output $\sigma \in [\![\mathcal{T}]\!](w)$. These two tasks are separated in our setting of ranked MSO enumeration by having a MSO formula φ that defines the outputs $[\![\varphi]\!]$ and a weighted MSO formula α to assign a cost to each pair (w, σ). In fact, one can show that cost transducers are equally expressive than combining MSO plus weighted MSO.

▶ **Proposition 4.** *For every cost transducer $\mathcal{T}$, there exists a MSO formula $\varphi_\mathcal{T}$ and weighted MSO formula $\alpha_\mathcal{T}$ such that $[\![\mathcal{T}]\!] = [\![\varphi_\mathcal{T}]\!]$ and $\text{cost}_\mathcal{T}(w, \sigma) = [\![\alpha_\mathcal{T}]\!](w, \sigma)$ for every $\sigma \in [\![\mathcal{T}]\!](w)$. Moreover, for every MSO formula φ and weighted MSO formula α, there exists a cost transducer $\mathcal{T}_{\varphi,\alpha}$ such that $[\![\varphi]\!] = [\![\mathcal{T}_{\varphi,\alpha}]\!]$ and $[\![\alpha]\!](w, \sigma) = \text{cost}_{\mathcal{T}_{\varphi,\alpha}}(w, \sigma)$ for every $\sigma \in [\![\varphi]\!](w)$.*

By the previous result, we can represent pairs of formulas (φ, α) by using cost transducers and vice-versa. Similar than for MSO [22], there exists a non-elementary blow-up for going from (φ, α) to a cost transducer and this blow-up cannot be avoided [16].

To solve the problem RANK-ENUM$[\varphi, \alpha]$ we can use a cost transducer $\mathcal{T}_{\varphi,\alpha}$ to enumerate all its outputs following the cost assigned by this machine. More concretely, we study the following rank enumeration problem for cost transducers:

Problem:	RANK-ENUM-T
Input:	A cost transducer $\mathcal{T}$ and a word $w \in \Sigma^*$.
Output:	Enumerate all $\sigma_1, \ldots, \sigma_k \in [\![\mathcal{T}]\!](w)$ without repetitions and such that $\text{cost}_{\mathcal{T}}(w, \sigma_i) \preceq \text{cost}_{\mathcal{T}}(w, \sigma_{i+1})$.

Note that for RANK-ENUM-T we consider the cost transducer as part of the input[1]. Indeed, for this case we can provide an enumeration algorithm with stronger guarantees regarding the preprocessing time in terms of $\mathcal{T}$. We now give the theorem formalizing the main result of this paper, which will be proven in the next section:

▶ **Theorem 5.** *The problem RANK-ENUM-T can be solved with* $|\mathcal{T}| \cdot |w|$ *preprocessing time and* $\log(|\mathcal{T}| \cdot |w|)$*-delay.*

In the rest of the paper, we present the above mentioned ranked enumeration algorithm. We start by showing a general algorithm based on a novel data structure called a Heap of Words. In Section 6, we provide the implementation of this structure. In Section 7, we show how to implement the incremental Brodal queues, a technical data structure needed to obtain the required efficiency. Before presenting the technical details of this algorithm, in the next section we show an application of this result in the framework of document spanners [13, 9].

4 Application: document spanners

The framework of document spanners was proposed in [13] as a formalization of rule-based information extraction and has attracted a lot of attention both in terms of the formalism [15, 21] and the enumeration problem associated to it [14]. Recently, an extension of document spanners has been proposed to enhance the extraction process with annotations [9, 8]. These annotations serve as auxiliary information of the extracted data such as confidence, support, or confidentiality measures. To extend spanners, this framework follows the approach of provenance semiring by annotating the output with elements from a semiring and propagating the annotations by using the semiring operators. Next we give the core definitions of [9] and we state the application of our main results to this setting.

We start by defining the central elements of document spanners: documents and spans. Fix a finite alphabet Σ. A document over Σ (or just a document) is a string $d = a_1 \ldots a_n \in \Sigma^*$ and a span is a pair $s = [i, j\rangle$ with $1 \leq i \leq j \leq n + 1$. A span represents a continuous region of d, whose content is the substring from positions i to $j - 1$. Formally, the content of span $[i, j\rangle$ is defined as $d[i, j\rangle = a_i \ldots a_{j-1}$; if $i = j$, then $d[i, i\rangle = \epsilon$. Fix a finite set of variables $\mathbf{X}$. A mapping μ over d is a function from $\mathbf{X}$ to the spans of d. A document spanner (or just spanner) is a function that maps each document d to a set of mappings over d.

To annotate mappings, we need to introduce semirings. A semiring $(K, \oplus, \odot, \mathtt{0}, \mathtt{1})$ is an algebraic structure where K is a non-empty set, $\oplus$ and $\odot$ are binary operations over K, and $\mathtt{0}, \mathtt{1} \in K$. Furthermore, $\oplus$ and $\odot$ are associative, $\mathtt{0}$ and $\mathtt{1}$ are the identities of $\oplus$ and $\odot$ respectively, $\oplus$ is commutative, $\odot$ distributes over $\oplus$, and $\mathtt{0}$ annihilates K (i.e.,

[1] In Section 2 we introduce the setting of ranked enumeration for MSO formulas and cost functions. One can easily extend this setting and the definiton of enumeration algorithms for cost transducer.

$0 \odot k = k \odot 0 = 0$ for all $k \in K$). We will use $\bigoplus_X$ or $\bigodot_X$ for the $\oplus$- or $\odot$-iteration over all elements in some set X, respectively. An ordered semiring $(K, \oplus, \odot, 0, 1, \preceq)$ is a semiring extended with a total order $\preceq$ over K such that $\preceq$ preserves $\oplus$ and $\odot$, namely, $k_1 \preceq k_2$ implies $k_1 * k \preceq k_2 * k$ for $* \in \{\oplus, \odot\}$. From now on, we will assume that all semirings are ordered. A semifield [17] is a semiring $(K, \oplus, \odot, 0, 1)$ where each $k \in K \setminus \{0\}$ has a multiplicative inverse (i.e., $(K \setminus \{0\}, \odot, 1)$ forms a group). Examples of ordered semifields are the tropical semiring $(\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0, \leq)$ and the semiring of non-negative rational numbers $(\mathbb{Q}_{\geq 0}, +, \times, 0, 1, \leq)$.

Fix a semiring $(K, \oplus, \odot, 0, 1)$. Let $\mathbf{X}$ be a set of variables and define $\mathcal{C}(\mathbf{X})$ as the set of captures $\{x\vdash, \dashv x \mid x \in \mathbf{X}\}$. To extend spanners with annotations, we use the formalism of weighted variable set automata [9] which defines the class of all regular spanners with annotations, also called regular annotators. A weighted variable set automaton (wVA) over K is a tuple $\mathcal{A} = (\mathbf{X}, Q, \delta, I, F)$ such that $\mathbf{X}$ is a finite set of variables, Q is a finite set of states, $\delta : Q \times (\Sigma \cup \mathcal{C}(\mathbf{X})) \times Q \to K$ is a weighted transition function and $I : Q \to K$ and $F : Q \to K$ are the initial and final weight functions, respectively. A run ρ over a document $d = a_1 \cdots a_n$ is a sequence of the form:

$$\rho \;:=\; (q_0, i_0) \xrightarrow{o_1} (q_1, i_1) \xrightarrow{o_2} \;\ldots\; \xrightarrow{o_m} (q_m, i_m)$$

where (1) $1 = i_0 \leq i_1 \leq \cdots \leq i_m = n+1$, (2) each $q_j \in Q$ with $I(q_0) \neq 0 \neq F(q_m)$, (3) $\delta(q_j, o_{j+1}, q_{j+1}) \neq 0$, and (4) $i_{j+1} = i_j$ if $o_{j+1} \in \mathcal{C}(\mathbf{X})$ and $i_{j+1} = i_j + 1$ otherwise. In addition, we say that a run ρ is valid if for every $x \in \mathbf{X}$ there exist exactly one index i with $o_i = x\vdash$, exactly one index j with $o_j = \dashv x$, and $i < j$. We denote by $\mathrm{Run}_{\mathcal{A}}(d)$ the set of all valid runs of $\mathcal{A}$ over d. Note that for some wVA $\mathcal{A}$ and document d there could exist runs of $\mathcal{A}$ over d that are not valid. For this reason, we say that $\mathcal{A}$ is functional if every run ρ of $\mathcal{A}$ over d is valid for every document d. Given that some decision problems for non-functional variable-set automata are NP-hard [15, 21], from now on we assume that all wVA are functional.

A valid run ρ like above naturally defines a mapping μ^ρ over $\mathbf{X}$ that maps each x to the span $[i_j, i_{j'}\rangle$ where $o_{i_j} = x\vdash$ and $o_{i_{j'}} = \dashv x$. Furthermore, we associate a weight in K to ρ by multiplying all the weights of the transitions as follows:

$$W(\rho) \;:=\; I(q_0) \odot \bigodot_{j=1}^{m} \delta(q_j, o_{j+1}, q_{j+1}) \odot F(q_m).$$

We define the set of output mappings of $\mathcal{A}$ over d as $[\![\mathcal{A}]\!](d) = \{\mu^\rho \mid \rho \in \mathrm{Run}_{\mathcal{A}}(d)\}$. Given a mapping $\mu \in [\![\mathcal{A}]\!](d)$, we associate the weight $W_{\mathcal{A},d}(\mu) = \bigoplus_{\rho \in \mathrm{Run}_{\mathcal{A}}(d) : \mu = \mu^\rho} W(\rho)$. Intuitively, each $\mu \in [\![\mathcal{A}]\!](d)$ contains relevant data extracted by $\mathcal{A}$ from d, and $W_{\mathcal{A},d}(\mu)$ is the annotation attached to μ obtained during the extraction process, e.g. confidence or support.

In [9], the problem of ranked annotator enumeration was proposed, which for the sake of completeness we present next[2]:

Problem:	RA-ENUM
Input:	A wVA $\mathcal{A}$ over an ordered semiring K and a document d.
Output:	Enumerate all $\mu_1, \ldots, \mu_k \in [\![\mathcal{A}]\!](d)$ without repetitions and such that $W_{\mathcal{A},d}(\mu_1) \preceq W_{\mathcal{A},d}(\mu_{i+1})$.

[2] In [9] they considered positively ordered semiring, which is slightly more general that the notion of ordered semiring used here.

RA-ENUM was studied in [9] and an enumeration algorithm was provided with polynomial preprocessing and polynomial delay in terms of $|\mathcal{A}|$ and $|d|$. By using the framework of MSO cost functions, we can give a better algorithm for a special case of RA-ENUM. We say that a wVA $\mathcal{A}$ is unambiguous if, for every document d and $\mu \in [\![\mathcal{A}]\!](d)$, there exists at most one run $\rho \in \text{Run}_{\mathcal{A}}(d)$ such that $\mu = \mu^\rho$. The connection between cost transducers and wVA is direct, although the former works over groups and wVA works over semirings. For this reason, we restrict wVA to semifields and give the following result.

▶ **Corollary 6.** *The problem RA-ENUM can be solved with $|\mathcal{A}| \cdot |d|$ preprocessing time and $\log(|\mathcal{A}| \cdot |d|)$-delay when $\mathcal{A}$ is unambiguous and K is an ordered semifield.*

Although the previous result is a restricted case of RA-ENUM and a direct consequence of Theorem 5, to the best of our knowledge this is the first non-trivial ranked enumeration algorithm proposed for the framework of document spanner.

5 Ranked enumeration algorithm

In this section, we will see how novel data structures can solve the ranked enumeration problem for cost transducers on words. We provide an algorithm for the RANK-ENUM-T problem, which uses a structure called *Heap of Words* (HoW) as a black box. We specify the interface of the HoW, to then present the full algorithm. The HoW structure is addressed in detail in the next section. This structure has the property of being fully-persistent. Given that this is a crucial property, we start with a brief introduction to this concept.

Fully-persistent data structures. A data structure is said *fully-persistent* [10] when no operation can modify the data structure. In a fully-persistent data structure, all the operations return new data structures, without changing the original ones. While this seems to be a restriction on the possible operations, it allows "sharing". For instance, with a fully-persistent linked list data structure, we can keep two lists l_1, l_2 with l_1 being some value followed by the content of l_2 and since no operation modifies the content of l_1 or l_2 there is no risk that an access to l_1 modifies indirectly l_2. In contrast, if we had allowed an operation that modifies the first value of a list in place (i.e., without returning a new list containing the modification), the applying this new operation on l_2 would have modified both l_1 and l_2.

All data structures that we use in this paper are fully-persistent. We use these data structures to store and enumerate the outputs of the cost transducer while, at the same time, share and modify the outputs without any risk of losing them. For more information of fully-persistent data structures, we refer the reader to [10].

The HoW data structure. A Heap of Words (HoW) over an ordered group $(\mathbb{G}, \oplus, \mathbb{0}, \preceq)$ is a data structure h that stores a finite set $\{[w_1 : g_1], \ldots, [w_n : g_n]\}$ where each $[w_i : g_i]$ is a pair composed by a word $w_i \in \Sigma^*$ and a priority $g_i \in \mathbb{G}$. Further, we assume that $w_i \neq w_j$ for every $i \neq j$, namely, the stored words form a set too. We define $[\![h]\!] = \{w_1, \ldots, w_n\}$ as the content of h and, given the previous restriction, there is a one-to-one correspondence between $[w_i : g_i]$ and w_i. Notice that we will usually write $h = \{[w_1 : g_1], \ldots, [w_n : g_n]\}$ to denote that h stores $[w_1 : g_1], \ldots, [w_n : g_n]$ but, strictly speaking, h is a data structure (i.e., a heap). Finally, we denote by $\emptyset$ the empty HoW.

The purpose of a HoW h is to store words and retrieve quickly the pair $[w : g]$ with minimum priority with respect to the order $\preceq$ of the group. We also want to manage h by *deleting* the word with minimum priority, *adding* new words, *increasing* the priority of

Algorithm 1 Preprocessing and enumeration phases for RANK-ENUM-T.

```
input: 𝒯 = (Q, Δ, κ, I, F) and w = a_1 ... a_n.
 1: procedure PREPROCESSING(𝒯, w)
 2:     for each q ∈ dom(I) do
 3:         h_q^0 ← ADD(∅, [ϵ : I(q)])
 4:     for i from 1 to n do
 5:         for each t = (p, a_i, X̄, q) ∈ Δ do
 6:             h ← h_p^{i-1}
 7:             if X̄ ≠ ∅ then
 8:                 h ← EXTENDBY(h, (X̄, i))
 9:             h ← INCREASEBY(h, κ(t))
10:             h_q^i ← MELD(h_q^i, h)
11:     for each q ∈ dom(F) do
12:         h ← INCREASEBY(h_q^n, F(q))
13:         h_out ← MELD(h_out, h)
14:     return h_out
```

```
input: A heap of words h.
 1: procedure ENUMERATION(h)
 2:     write #
 3:     while h ≠ ∅ do
 4:         write FINDMIN(h)
 5:         h ← DELETEMIN(h)
 6:         write #
```

all elements by some $g \in \mathbb{G}$, or *extending* all words with a new letter $a \in \Sigma$. Furthermore, we want to build the union of two HoWs. More formally, we consider the following set of functions to manage HoWs. For HoWs h, h_1, and h_2, $w \in \Sigma^*$, $g \in \mathbb{G}$, and $a \in \Sigma$ we define:

$$
\begin{array}{lll@{\qquad}llll}
w' & := & \textsc{FindMin}(h) & h' & := & \textsc{Meld}(h_1, h_2) & \text{s.t. } [\![h_1]\!] \cap [\![h_2]\!] = \emptyset \\
h' & := & \textsc{DeleteMin}(h) & h' & := & \textsc{Add}(h, [w\!:\!g]) & \text{s.t. } w \notin [\![h]\!] \\
h' & := & \textsc{IncreaseBy}(h, g) & h' & := & \textsc{ExtendBy}(h, a) &
\end{array}
$$

where h' is a new HoW and $w' \in \Sigma^*$. In general, each of such functions receives a HoW and outputs a HoW h'. As it was explained before, this data structure is fully-persistent and, therefore, after applying any of these functions, both the output h' and its previous version h are available. Now, we define the semantics of each operation. Let $h = \{[w_1\!:\!g_1], \ldots, [w_n\!:\!g_n]\}$. The FINDMIN of h returns a word w' such that $[w'\!:\!g']$ is stored in h and g' is minimal among all the priorities stored, formally, $[w'\!:\!g'] \in h$ and $g' = \min\{g \mid [w\!:\!g] \in h\}$. If there are several w' satisfying this property, one is picked arbitrarily. Operation DELETEMIN returns a new HoW h' that stores the set represented by h without the pair of the word returned by FINDMIN(h), that is, $h' = h \setminus \{[w'\!:\!g']\}$ where $w' = \textsc{FindMin}(h)$ and $g' = \min\{g \mid [w\!:\!g] \in h\}$. Finally, the functions ADD, INCREASEBY, EXTENDBY, and MELD are formally defined as:

$$
\begin{array}{ll}
\textsc{Meld}(h_1, h_2) := h_1 \cup h_2 & \textsc{IncreaseBy}(h, g) := \{[w_1\!:\!(g_1 \oplus g)], \ldots, [w_n\!:\!(g_n \oplus g)]\} \\
\textsc{Add}(h, [w\!:\!g]) := h \cup \{[w\!:\!g]\} & \textsc{ExtendBy}(h, a) := \{[(w_1 \cdot a)\!:\!g_1], \ldots, [(w_n \cdot a)\!:\!g_n]\}
\end{array}
$$

We assume that ADD, INCREASEBY, EXTENDBY and MELD take constant time and FINDMIN takes $\mathcal{O}(|w'|)$ where $w' = \textsc{FindMin}(h)$. For DELETEMIN(h), if h was built using n operations ADD, INCREASEBY, EXTENDBY and MELD followed by some number of operations DELETEMIN then computing DELETEMIN(h) takes $\mathcal{O}(|w'| \cdot \log(n))$ where $w' = \textsc{FindMin}(h)$. In the next section we show how to implement HoWs in order to satisfy these requirements. For now, we assume the existence of this data structure and use it to solve RANK-ENUM-T.

The algorithm. In Algorithm 1, we show the preprocessing phase and the enumeration phase to solve RANK-ENUM-T. On one hand, the PREPROCESSING procedure receives

a cost transducer $\mathcal{T} = (Q, \Delta, \kappa, I, F)$ and a word $w \in \Sigma^*$, and computes a HoW h_{out}. On the other hand, the Enumeration procedure receives a HoW (i.e., h_{out}) and enumerates $\text{enc}(\sigma_1), \ldots, \text{enc}(\sigma_k)$ such that $\{\sigma_1, \ldots, \sigma_k\} = [\![\mathcal{T}]\!](w)$ and $\text{cost}_{\mathcal{T}}(w, \sigma_i) \preceq \text{cost}_{\mathcal{T}}(w, \sigma_{i+1})$.

In both procedures we use HoW to compute the set of answers. Indeed, for each $q \in Q$ and each $i \in \{0, \ldots, |w|\}$ we compute a HoW h_q^i, and also compute a h_{out} to store the final results. We assume that all HoWs are empty (i.e., $h_{\text{out}} = \emptyset$ and $h_q^i = \emptyset$) when the algorithm starts. For each i, we call the set $\{h_q^i \mid q \in Q\}$ the i-level of HoW. Starting from the 0-level (lines 2-3), the preprocessing phase goes level by level, updating the i-level with the previous $(i-1)$-level (lines 4-10). It is important to note here that the Meld(h_q^i, h) call (line 10) is well-defined since $\mathcal{T}$ is unambiguous (i.e., $[\![h_q^i]\!] \cap [\![h]\!] = \emptyset$). After reaching the last n-level, the algorithm joins all HoWs $\{h_q^n \mid q \in \text{dom}(F)\}$ into h_{out}, by incrementing first their cost with $F(q)$ and melding them into h_{out} (lines 11-13). Finally, the preprocessing phase return h_{out} as output (line 14).

In order to understand the preprocessing algorithm, one has to notice that all the evaluation is based on a very simple fact. Let $w_i = a_1 \ldots a_i$ and define the set $\text{Run}_{\mathcal{T}}(q, w_i)$ of all partial runs of $\mathcal{T}$ over w_i that end in state q. For any of such runs $\rho = q_0 \xrightarrow{a_1/\bar{X}_1} \ldots \xrightarrow{a_i/\bar{X}_i} q_i \in \text{Run}_{\mathcal{T}}(q, w_i)$, define the partial cost of ρ as $\kappa^*(\rho) = I(q_0) \oplus \bigoplus_{j=1}^{i} \kappa((q_{j-1}, a_j, \bar{X}_j, q_j))$. After executing Preprocessing, it will hold that: $h_q^i = \{ [\text{out}(\rho) : \kappa^*(\rho)] \mid \rho \in \text{Run}_{\mathcal{T}}(q, w_i) \}$. This is certainly true for h_q^0 after lines 2-3 are executed. Then, if this is true for $(i-1)$-level, after the i-th iteration of lines 5-10 we will have that h_q^i contains all pairs of the form $[\text{out}(\rho) \cdot (\bar{X}, i) : \kappa^*(\rho) \oplus \kappa(t)]$ for each $t = (p, a_i, \bar{X}, q) \in \Delta$, plus all pairs $[\text{out}(\rho) : \kappa^*(\rho) \oplus \kappa(t)]$ for each $t = (p, a_i, \emptyset, q) \in \Delta$ and $\rho \in \text{Run}_{\mathcal{T}}(p, w_{i-1})$. Given that each line takes constant time, we can conclude that the preprocessing phase takes time $\mathcal{O}(|\mathcal{T}| \cdot |w|)$ as expected.

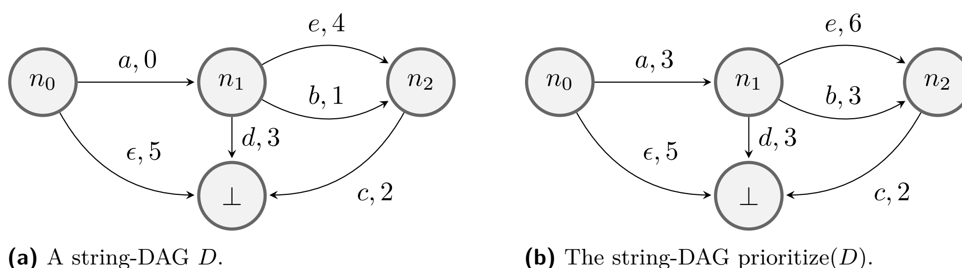
For the enumeration phase, we extract each output from h_{out}, one by one, by alternating between the FindMin and DeleteMin procedures. Since with DeleteMin we remove the minimum element of h after printing it, the correctness of the enumeration phase is straightforward. Notice that this enumeration will print all outputs in increasing order of priority. Furthermore, it will not print any output twice given that h_{out} contains no repetitions. To bound the time, notice that the number of Add, IncreaseBy, ExtendBy and Meld functions used during the pre-processing is at most $\mathcal{O}(|\mathcal{T}| \cdot |w|)$. For this reason, the delay between each output w' is bounded by $\mathcal{O}(\log(|\mathcal{T}| \cdot |w|) \cdot |w'|)$, satisfying the promised delay between outputs.

We want to finish this section by emphasizing that the ranked enumeration problem of cost transducers reduces to computing efficiently the HoW's methods. Moreover, it is crucial in this algorithm that this data structure is fully-persistent, and each operation takes constant time. Indeed, this allows us to pass the outputs between levels very efficiently and without losing the outputs of the previous levels.

6 The implementation of HoW data structure

In this section we focus on the HoW data structure and explain its implementation using yet another structure called incremental Brodal queue. We begin by explaining the general technique we use to store sets of strings with priorities, and end by giving a full implementation of the functions to manage HoWs.

Let Σ be a possibly infinite alphabet and $\mathbb{G} = (\mathbb{G}, \oplus, \mathbb{0}, \preceq)$ an order group. A *string-DAG* over Σ and $\mathbb{G}$ is a DAG $D = (V, E)$ where the edges are annotated with symbols in $\Sigma \cup \{\epsilon\}$ and priorities in $\mathbb{G}$. Formally, each edge has the form $e = (u, a, g, v)$, where $u, v \in V$, $a \in \Sigma \cup \{\epsilon\}$ and $g \in \mathbb{G}$. Given a path $\rho = v_1 \xrightarrow{a_1, g_1} \ldots \xrightarrow{a_k, g_k} v_k$, let $[w_\rho : g_\rho]$ be the pair defined by



(a) A string-DAG D.

(b) The string-DAG prioritize(D).

Figure 1 A string-DAG D and the result of prioritize(D).

ρ, where $w_\rho = a_1 \dots a_k$ and $g_\rho = g_1 \oplus \dots \oplus g_k$. We make two more assumptions that any string-DAG must satisfy. First, we assume that there is a special sink vertex $\bot \in V$ that is reachable from any $v \in V$, has no outgoing edges, and that all edges with ϵ must point to $\bot$. Second, we assume that, for every $v \in V$ and every two different paths ρ and ρ' from v to $\bot$, it holds that $w_\rho \neq w_{\rho'}$. Given these two assumptions, we say that each $v \in V$ encodes a set of pairs $[\![D]\!](v)$: for $v = \bot$ this set is the empty set, while for all $v \neq \bot$ this set is defined by all the paths from v to $\bot$, i.e., $[\![D]\!](v) = \{[w_\rho : g_\rho] \mid \rho \text{ is a path from } v \text{ to } \bot\}$. By these two assumptions, there is a correspondence between the words in $[\![D]\!](v)$ and the paths from v to $\bot$. For instance, the strings associated with n_0 in the string-DAG depicted in Figure 1a are ad with priority $0+3=3$, abc with priority $0+1+2=3$, aec with priority $0+4+2=6$ and ϵ with priority 5.

This structure is useful to store a big number of strings in a compressed manner. Further, since ϵ can only appear at the last edge of a path, by doing a DFS it can be used to retrieve all of them without repetitions and taking time linear in the length of each string. However, one can see that it is not very useful when we want to enumerate them by rank order. This motivates the following string-DAG construction. We define a function prioritize(D) that receives a string-DAG $D = (V, E)$ and returns a string-DAG $D' = (V, E')$ where each edge (u, a, g, v) of E is replaced by an edge $(u, a, g \oplus g', v)$ in E', where g' is the minimum priority in $[\![D]\!](v)$. For instance, Figure 1b shows the string-DAG resulting after applying prioritize to D of Figure 1a. Having prioritize(D) makes finding the string with minimum priority of a vertex much easier: we simply need to follow recursively the edge with minimum priority. In n_0 of Figure 1b we make the path $n_0 \xrightarrow{a,3} n_1 \xrightarrow{b,3} n_2 \xrightarrow{c,2} \bot$ and compute the minimum pair $[abc, 3]$ (the priority is retrieved from the first edge).

Before presenting the HoW implementation, we need to introduce another fully-persistent data structure. This structure is based on the Brodal queue [4], a known worst-case efficient priority queue, which we extend with the new function increaseBy. Formally, an *incremental Brodal queue*, or just a *queue*, is a fully-persistent data structure Q which stores a set $P = \{[\mathcal{E}_1 : g_1] \dots [\mathcal{E}_k : g_k]\}$, where each $\mathcal{E}_i$ is a stored element and g_i is its priority. As an abuse of notation, we often write $Q = P$. The functions to manage incremental Brodal queues include all functions for HoW except ExtendBy, namely findMin, deleteMin, add, increaseBy and meld; their definition also remains the same as for HoW. Note that we use different fonts to distinguish the operations over HoWs versus the operations over incremental Brodal queues. For example, we write FindMin for HoWs and findMin for queues. Further, this queue has two additional functions: isEmpty, that checks if the queue is the empty queue $\emptyset$; and minPrio, that returns the value g of the minimal priority among all the priorities stored. For the rest of this section we assume the existence of an incremental Brodal queue structure such that all functions run in time $\mathcal{O}(1)$ except for deleteMin, which runs in $\mathcal{O}(\log(n))$, where

Algorithm 2 HoW's implementation of ADD, EXTENDBY, FINDMIN and DELETEMIN.

```
 1: procedure ADD(⟨Q⟩, [a : g])
 2:     return ⟨add(Q, [(a, ⟨∅⟩) : g])⟩
 3: procedure EXTENDBY(⟨Q⟩, a)
 4:     if isEmpty(Q) then
 5:         return ⟨∅⟩
 6:     return ⟨add(∅, [(a, ⟨Q⟩) : minPrio(Q)])⟩
 7: procedure FINDMIN(⟨Q⟩)
 8:     (a, ⟨Q′⟩) ← findMin(Q)
 9:     if isEmpty(Q′) then
10:         return a
11:     return FINDMIN(⟨Q′⟩) · a
12: procedure DELETEMIN(⟨Q⟩)
13:     if isEmpty(Q) then
14:         return ⟨∅⟩
15:     (a, ⟨R⟩) ← findMin(Q)
16:     Q′ ← deleteMin(Q)
17:     ⟨R′⟩ ← DELETEMIN(⟨R⟩)
18:     if isEmpty(R′) then
19:         return ⟨Q′⟩
20:     δ ← minPrio(R′) ⊕ (minPrio(R))^−1
21:     g ← minPrio(Q) ⊕ δ
22:     return ⟨add(Q′, [(a, ⟨R′⟩) : g])⟩
```

n is the number of pairs stored in the queue. Finally, all these operations are fully-persistent. The in-detail explanation of this structure is derived to the next section.

With the previous intuition and the structure above, we can now present the implementation for Heap of Words. A HoW h is implemented as an incremental Brodal queue Q that stores a set $\{[(a_1, h_1) : g_1], \ldots, [(a_k, h_k) : g_k]\}$, where each $a_i \in \Sigma \cup \{\epsilon\}$, each h_i is a HoW and each $g_k \in \mathbb{G}$. We write $h = \langle Q \rangle$ to make clear that we are talking about a HoW and not the queue. The empty HoW is simply the empty queue $\langle \emptyset \rangle$. Intuitively, the recursive references to HoWs are used to encode a string-DAG D; more specifically, we use it to encode $\mathsf{prioritize}(D) = (V, E)$ and store the edges using the queue structure. For every $u \in V$, we define a HoW $h_u = \langle Q \rangle$ such that each pair $[(a, h_v) : g]$ stored in Q represents an edge $(u, a, g, v) \in E$. For instance, continuing with the example of Figure 1b, we have a HoW for each vertex: $h_\bot = \langle \emptyset \rangle$, $h_{n_2} = \langle \{[(c, h_\bot) : 2]\} \rangle$, $h_{n_1} = \langle \{[(e, h_{n_2}) : 6], [(b, h_{n_2}) : 3], [(d, h_\bot) : 3]\} \rangle$ and $h_{n_0} = \langle \{[(a, h_{n_1}) : 3], [(\epsilon, h_\bot) : 5]\} \rangle$.

We now explain the implementation of the functions defined in Section 5 to manage HoW. Consider a HoW $h = \langle Q \rangle$. For each $\textsc{op} \in \{\textsc{Meld}, \textsc{IncreaseBy}\}$, the function is just applied directly to the queue, i.e., $\textsc{op}(\langle Q \rangle) = \langle \mathsf{op}(Q) \rangle$. The implementation of ADD and the other functions is now described and presented in Algorithm 2.

In the case of $\textsc{Add}(h, a)$, an edge is added that points to $\langle \emptyset \rangle$; this can be extended to add a word w instead by allowing that edges keep words instead of single letters. To implement $\textsc{ExtendBy}(\langle Q \rangle, a)$, we simply need to create a new queue containing the element $[(a, \langle Q \rangle) : \mathsf{minPrio}(Q)]$. Note that the appended letter is added not at the end, but at the beginning, meaning that the strings are actually being stored in inverted order. This is managed in $\textsc{FindMin}(\langle Q \rangle)$, where the output string is inverted back. For $\textsc{FindMin}(\langle Q \rangle)$, to get the minimum element we recursively use $\mathsf{findMin}(Q)$ to find the outgoing edge with minimum priority, as we explained when the prioritize function was introduced. For DELETEMIN, in order to delete the string with minimum priority, we use the fact that the set of all paths, minus the one with minimal priority, is composed by: (1) all the paths that do not start with the minimal edge, and (2) all the paths starting with the minimal edge that are followed by any path minus the one with minimal priority. For instance, in Figure 1b, the minimal path from n_0 is $\pi = n_0 \xrightarrow{a} n_1 \xrightarrow{d} \bot$. Then, the set of paths minus π is composed by (1) $n_0 \xrightarrow{\epsilon} \bot$, and (2) $n_0 \xrightarrow{a} n_1 \xrightarrow{e} n_2 \xrightarrow{c} \bot$, $n_0 \xrightarrow{a} n_1 \xrightarrow{b} n_2 \xrightarrow{c} \bot$. In procedure DELETEMIN, $\langle Q' \rangle$ stores the paths of (1), while $\langle R' \rangle$ stores the paths from (2) minus the first edge (lines 16-17).

Further, since the minimal path was removed, a new priority needs to be computed for this edge, which is computed and stored as g (line 20-21). This priority is used to create an edge to R', i.e., $[(a, \langle R' \rangle) : g]$, which together represent the paths of (2). This is connected with the paths of (1), i.e., $\langle Q' \rangle$, and the result is returned in line 22. The border case case where (2) is empty is managed by lines 18-19, in which case it simply returns $\langle Q' \rangle$.

It is straightforward to check that this data structure achieves the time and space bounds given in Section 5. We end this section by arguing that the implementation of HoW is fully-persistent. For this, note that the performance of HoW relies on the implementation of incremental Brodal queues. Indeed, given that these queues are fully-persistent and each method in Algorithm 2 creates new queues without modifying the previous ones, the whole data structure is fully-persistent. Therefore, it is left to prove that we can extend Brodal queues as we already mentioned. We will show this in the last section.

7 Incremental brodal queues

In this section, we discuss how to implement an incremental Brodal queue, the last ingredient of our ranked enumeration algorithms for MSO cost functions. This data structure extends Brodal queues [4] by including the increaseBy procedure. Indeed, our construction of incremental Brodal queues follows the same approach as in [4]. We start by defining what we call an *incremental binomial heap*, for which most operations take logarithmic time, to then show how to extend it to lower the cost to constant time, except for deleteMin that takes logarithmic time. The relevant aspects for this extension (i.e., to support increaseBy) appear in the definition of the incremental binomial heap. For this reason, in this section we present only the implementation of the incremental binomial heap. The details of how to extend it to an incremental Brodal queue can be found in [4]. We start by introducing some notation.

A *multitree structure* is a pair $M = (V, \mathsf{first}, \mathsf{next}, v^0)$ where V is a set of nodes, $\mathsf{first} : V \to V \cup \{\bot\}$ and $\mathsf{next} : V \to V \cup \{\bot\}$ are functions such that $\bot \notin V$ and $v^0 \in V$ is a special node. Further, we assume that the directed graph $G_M = (V, \{(u, v) \mid \mathsf{first}(u) = v \text{ or } \mathsf{next}(u) = v\})$ is a multitree, namely, it is a directed acyclic graph (DAG) in which the set of vertices reachable from any vertex induces a tree. Let V_{v^0} denotes the reachable nodes from v^0 and $G_{v^0} = (V_{v^0}, \{(u, v) \mid \mathsf{first}(u) = v \text{ or } \mathsf{next}(u) = v\})$ the graph induced by V_{v^0}, which is a tree by definition. Note that G_{v^0} is using the first-child next-sibling encoding to form an ordered forest. To see this, let $\mathsf{next}^*(v)$ be the smallest subset of V such that $v \in \mathsf{next}^*(v)$ and $\mathsf{next}(u) \in \mathsf{next}^*(v)$ whenever $u \in \mathsf{next}^*(v)$. Then the set $\mathsf{roots} = \mathsf{next}^*(v^0)$ represents the roots of the forest and for each $v \in V_{v^0}$ the set $\mathsf{children}(v) = \mathsf{next}^*(\mathsf{first}(v))$ are the children of the node v in the forest where $\mathsf{children}(v) = \emptyset$ when $\mathsf{first}(v) = \bot$. Here both sets are ordered by the next function, then we will usually write $\mathsf{roots} = v_1, \ldots, v_j$ or $\mathsf{children}(v) = u_1, \ldots, u_k$ to denote both the elements of the set and its order. Also, we write $\mathsf{parent}(v) = u$ if $v \in \mathsf{children}(u)$ and we say that v is a leaf if $\mathsf{first}(v) = \bot$. Note that in M a node could have different "parents" (i.e., G_M is a DAG) depending on the node v^0 that we start. We say that M *forms a tree* if $\mathsf{next}(v^0) = \bot$. Furthermore, for $v \in V$ we denote by M_v the tree hanging from v, namely, M_v is equal to M with the exception that $v^0_{M_v} = v$ and $\mathsf{next}_{M_v}(v) = \bot$. As it will clear below, this encoding will be helpful to build the data structure and assure the persistent requirement.

A binomial tree of rank k is recursively defined as follows. A binomial tree of rank 0 is a leaf and a binomial tree of rank $k+1$ is a multitree structure M that forms a tree such that $\mathsf{children}(v^0) = u_k, \ldots, u_0$ and M_{u_i} is a binomial tree of rank i. If M is a binomial tree we denote its rank by $\mathrm{rank}(M)$. One can easily show by induction over the rank (see [5]) that for

every binomial tree M of ranked k, it holds that $|V_{v^0}| = 2^k$ and, thus, the number of children of each node is of logarithmic size with respect to the size of T, i.e., $|\mathsf{children}(v)| \leq \log(|V_{v^0}|)$ for every $v \in V_{v^0}$. We use this property several times throughout this section.

Fix an ordered group $(\mathbb{G}, \oplus, \mathbb{0}, \preceq)$. An incremental binomial heap over $\mathbb{G}$ is defined as a pair $H = (V, \mathsf{first}, \mathsf{next}, v^0, \Delta, \mathsf{elem}, \delta^0)$ where $(V, \mathsf{first}, \mathsf{next}, v^0)$ is a multitree structure, $\Delta : V \to \mathbb{G}$ is the delta-priority function, $\mathsf{elem} : V \to \mathcal{E}$ is the element function where $\mathcal{E}$ is the set of elements that are stored, and $\delta^0 \in \mathbb{G}$ is an initial delta value. Further, if M is the multitree structure defined by $(V, \mathsf{first}, \mathsf{next}, v^0)$ and $\mathsf{roots} = v_1, \ldots, v_n$ are its roots, then each M_{v_i} is a binomial tree with $\mathrm{rank}(M_{v_i}) < \mathrm{rank}(M_{v_{i+1}})$ for each $i < n$. In other words, an incremental binomial heap has the same underlying structure than a standard binomial heap [5]. Usually in the literature [4], a binomial heap is imposed a min-heap property, meaning that a node always has lower priority than its children, which is crucial for dequeuing elements in order. Instead, we give to our heap a different semantics by keeping the difference between nodes with the Δ-function and computing the real priority function $\mathsf{pr}_{v^0} : V_{v^0} \to \mathbb{G}$ as follows: $\mathsf{pr}_{v^0}(v) := \delta^0 \oplus \Delta(v)$ whenever v is a root of the underlying multitree structure, and $\mathsf{pr}_{v^0}(v) := \mathsf{pr}_{v^0}(u) \oplus \Delta(v)$ whenever $\mathsf{parent}(v) = u$. Given that $\mathsf{parent}(v)$ depends on the starting node v^0, then pr_{v^0} also depends on v^0. In addition, we assume that a min-heap property is satisfied over the real priority function, namely, $\mathsf{pr}_{v^0}(u) \preceq \mathsf{pr}_{v^0}(v)$ whenever $\mathsf{parent}(v) = u$. Then H is a heap where each node $v \in V$ keeps a pair $(\mathsf{elem}(v), \mathsf{pr}_{v^0}(v))$ where $\mathsf{elem}(v)$ is the stored element and $\mathsf{pr}_{v^0}(v)$ its priority in the heap. This principle of storing the deltas between nodes instead of the real priority is crucial for supporting the increased-by operation of the data structure.

Next, we show how to implement the operations of an incremental Brodal queue stated in Section 6, namely, isEmpty, increaseBy, findMin minPrio, deleteMin, add, and meld. We implement this with an incremental binomial heap where the only difference is that isEmpty and increaseBy will take constant time, and findMin minPrio, deleteMin, add, and meld will take logarithmic time. To extend incremental binomial heaps to lower the complexity of findMin minPrio, deleteMin, and add to constant time, one can use the same techniques as in [4]. Most operations of incremental binomial heaps are similar to the operations on binomial heaps (see [5]), however, for the sake of completeness we explain each one in detail, highlighting the main differences to manage the delta priorities.

From now on, fix an incremental binomial heap $H = (V, \mathsf{first}, \mathsf{next}, v^0, \Delta, \mathsf{elem}, \delta^0)$. Given that all operations must be persistent, we will usually create a copy H' of H by extending H with new fresh nodes. More precisely, we will say that H' is an *extension* of H (denoted by $H \subseteq H'$) iff $V_H \subseteq V_{H'}$ and $\mathsf{op}_{H'}(v) = \mathsf{op}_H(v)$ for every $v \in V_H$ and $\mathsf{op} \in \{\mathsf{first}, \mathsf{next}, \Delta, \mathsf{elem}\}$ (note that v^0 and δ^0 may change). Furthermore, for $H \subseteq H'$ we will say that a node $v' \in V_{H'} \setminus V_H$ is a *fresh copy* of $v \in V_H$ if v' in H' has the same structure as v in H where only the differences are defined explicitly, namely, we omit the functions that are the same as for v. For example, if we say that "v' is a fresh copy of v such that $\mathsf{next}_{H'}(v') := \perp$", this means that $\mathsf{next}_{H'}(v') := \perp$ and $\mathsf{op}_{H'}(v') = \mathsf{op}_H(v)$ for every $\mathsf{op} \neq \mathsf{next}$.

The first operation, $\mathsf{isEmpty}(H)$, can easily be implemented in constant time, by just checking whether $v^0 = \perp$ or not. Similarly, $\mathsf{increaseBy}(H, \delta)$ can be implemented in constant time by just updating δ^0 to $\delta^0 \oplus \delta$, which is the purpose of having δ^0. For $\mathsf{findMin}(H)$ or $\mathsf{minPrio}(H)$, a bit more of work is needed. Recall that a k-rank binomial tree with $|V|$ nodes satisfies $|V| = 2^k$. Given that $\mathsf{roots} = v_1, \ldots, v_n$ is a sequence of binomial trees ordered by rank, one can easily see that $n \in \mathcal{O}(\log(|V_{v^0}|)$. Therefore, we need at most a logarithmic number of steps to find the node v_i with the minimum priority and return $\mathsf{elem}(v_i)$ or $\mathsf{pr}_{v^0}(v_i)$ whenever $\mathsf{findMin}(H)$ or $\mathsf{minPrio}(H)$ is asked, respectively.

For $\mathsf{add}(H, e, g)$ or $\mathsf{deleteMin}(H)$, we reduce them to melding two heaps. For the first operation, we create a heap H' whose multitree structure has one node, call it v, $\Delta_{H'}(v) := g$, $\mathsf{elem}_{H'}(v) := e$, and $\delta^0_{H'} := \mathbb{0}$. Then we apply $\mathsf{meld}(H, H')$ obtaining a heap where the new node (e, g) is added to H. For the second operation, we remove the minimum element by creating two heaps and then apply the meld operation. Specifically, let $\mathsf{roots} = v_1, \ldots, v_n$ be the roots of H and v_i be the root with the minimum priority. Then we build two heaps H_1 and H_2 such that $H \subseteq H_i$ for $i \in \{1, 2\}$. For H_1, we extend H by creating fresh copies of all v_j, $j \neq i$. Formally, define $V_{H_1} = V_H \cup \{v'_1, \ldots, v'_n\}$ where each v'_j is a fresh copy of v_j with the exception of v'_{i-1} that we set $\mathsf{next}_{H_1}(v'_{i-1}) := v'_{i+1}$. Finally, define $v^0_{H_1} = v'_1$ as the starting node of H_1. Now, for H_2 we extend H by creating a copy of the children of v_i in H in reverse order and updating δ^0_H to $\delta^0_H \oplus \Delta_H(v_i)$ (recall that the children of a binomial tree are ordered by decreasing rank). Formally, if $\mathsf{children}_H(v_i) = u_1, \ldots, u_k$, then $V_{H_2} = V_H \cup \{u'_1, \ldots, u'_k\}$ where each u'_j is a fresh copy of u_j such that $\mathsf{next}_{H_2}(u'_j) := u'_{j-1}$ for $j > 1$ and $\mathsf{next}_{H_2}(u'_1) := \perp$. Finally, define $v^0_{H_2} := u'_k$ and $\delta^0_{H_2} := \delta^0_H \oplus \Delta_H(v_i)$. The reader can check that H_1 and H_2 are valid incremental binomial heaps and, furthermore, H_1 is H without v_i and H_2 contains only the children of v_i in reverse order. Therefore, to compute $\mathsf{deleteMin}(H)$ we return $\mathsf{meld}(H_1, H_2)$. Given that the construction of H_1 and H_2 takes at most logarithmic time in the size of H (i.e., there is at most a log number of roots or children), then the procedure takes logarithmic time. Furthermore, H was never touched and then the operation is fully-persistent.

For $\mathsf{meld}(H_1, H_2)$, we use the same algorithm as for melding two binomial heaps with two modifications that are presented here. For melding two binomial heaps, we point the reader to [5] in which this operation is well explained. For the first change, we need to update the link operation [5] of two binomial trees to support the use of the delta priorities. Given a incremental binomial heap H and its underlying multitree structure M, let v_1 and v_2 be two nodes in H such that $\Delta(v_1) \preceq \Delta(v_2)$ and M_{v_1} and M_{v_2} has the same rank k. Then the *link* of v_1 and v_2, denoted by $\mathtt{link}(H, v_1, v_2)$, outputs a pair (H', v'_1) such that H' is an extension of H and $M'_{v'_1}$ is a binomial tree of rank $k+1$ containing the nodes of M_{v_1} and M_{v_2}. Formally, $V_{H'} := V_H \cup \{v'_1, v'_2\}$ and v'_1 and v'_2 are fresh copies of v_1 and v_2 such that $\mathsf{first}_{H'}(v'_1) := v'_2$, $\mathsf{next}_{H'}(v'_2) := \mathsf{first}_H(v_1)$ and $\Delta_{H'}(v'_2) := \Delta_H(v_1)^{-1} \oplus \Delta_H(v_2)$. Note that the new node v'_1 defines a binomial tree $M'_{v'_1}$ of rank $k+1$ containing all nodes of M_{v_1} and M_{v_2}, maintaining the priorities of H and such that $\mathsf{pr}_{H'}(u) \preceq \mathsf{pr}_{H'}(u')$ whenever $u = \mathsf{parent}(u')$. The second change of the algorithm in [5] is that, before melding H_1 and H_2, we push each initial delta value to the roots of the corresponding data structures. For this, given an incremental binomial heap H we construct $H^{\downarrow}$ with $H \subseteq H^{\downarrow}$ as follows. Let $\mathsf{roots}_H = v_1, \ldots, v_k$. Then $V_{H^{\downarrow}} = V_H \cup \{v'_1, \ldots, v'_k\}$ where $v'_1, \ldots, v'_k$ are fresh copies of $v_1, \ldots, v_k$ and $\Delta_{H^{\downarrow}}(v'_i) := \delta^0_H \oplus \Delta_H(v_i)$. Furthermore, we define $v^0_{H^{\downarrow}} := v'_1$ and $\delta^0_{H^{\downarrow}} := \mathbb{0}$. Note that in $H^{\downarrow}$ we can forget about the initial delta value given that this is included in the root of each binomial tree. Finally, to meld H_1 and H_2 we construct $H_1^{\downarrow}$ and $H_2^{\downarrow}$ and then apply the melding algorithm of [5] with the updated version of the link function, $\mathtt{link}(H, v_1, v_2)$. Overall, the operation takes logarithmic time to build $H_1^{\downarrow}$ and $H_2^{\downarrow}$, and logarithmic time to meld both heaps. Moreover, given that $\mathtt{link}(H, v_1, v_2)$ and the construction of $H_1^{\downarrow}$ and $H_2^{\downarrow}$ do not modify the initial heap H, then the meld operation is persistent as well.

To finish this section, we recall that the next step is to extend the incremental binomial heap to an incremental Brodal queue. For this, we follow the same approach as [4] to lower the time complexity of find-min, add, and meld operation from logarithmic to constant time.

8 Conclusions

This paper presented an algorithm to enumerate the answers of queries over words, in an order defined by a cost function, that has a linear preprocessing and a logarithmic delay in the size of the words. We first introduced the notion of MSO cost functions, to then present a ranked enumeration scheme. This scheme relies on a particular data structure called HoW. The complexity of our algorithms depends mainly on the performance of the operations of HoW. To implement them, we extend a well known persistent data structure called Brodal queue. Thanks to this data structure, we obtain the bounds of our algorithm.

For future work, we would like to find a lower bound that justifies the logarithmic delay or whether one can achieve a better delay. We also plan to study how the introduced data structures and algorithms could be used in other enumeration schemes (e.g., relational databases). Finally, we would also like to validate our approach in practical settings.

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
2 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.
3 Guillaume Bagan. Mso queries on tree decomposable structures are computable with linear delay. In *International Workshop on Computer Science Logic*, pages 167–181. Springer, 2006.
4 Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, 1996.
5 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
6 Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.
7 Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *CoRR*, abs/1902.02698, 2019.
8 Johannes Doleschal, Noa Bratman, Benny Kimelfeld, and Wim Martens. The complexity of aggregates over extractions by regular expressions. In *ICDT*, 2021.
9 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. In *ICDT*, volume 155, pages 8:1–8:18, 2020.
10 James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.
11 Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In *ICALP*, volume 3580, pages 513–525, 2005.
12 Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.
13 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015.
14 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.*, 45(1):3:1–3:42, 2020.
15 Dominik D Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *Proceedings of PODS*, pages 137–149, 2018.
16 Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Log.*, 130(1-3):3–31, 2004.
17 Jonathan S Golan. *Semirings and their Applications*. Springer Science & Business Media, 2013.

18 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A Formal Framework for Complex Event Processing. In *ICDT*, pages 5:1–5:18, 2019.

19 Stephan Kreutzer and Cristian Riveros. Quantitative monadic second-order logic. In *LICS*, pages 113–122, 2013.

20 Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

21 Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of PODS*, pages 125–136. ACM, 2018.

22 Klaus Reinhardt. The complexity of translating logic to finite automata. In *Automata logics, and infinite games*, pages 231–238. Springer, 2002.

23 Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*, pages 10–20, 2013.

24 Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *VLDB*, 13(9):1582–1597, 2020.

25 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *SIGMOD*, pages 2659–2665. ACM, 2020.