Revisiting Weighted Information Extraction: A Simpler and Faster Algorithm for Ranked Enumeration

PAWEŁ GAWRYCHOWSKI, University of Wrocław, Poland
FLORIN MANEA, Computer Science Department and CIDAS, Universität Göttingen, Germany
MARKUS L. SCHMID, Humboldt-Universität zu Berlin, Germany

Information extraction from textual data, where the query is represented by a finite transducer and the task is to enumerate all results without repetition, and its extension to the weighted case, where each output element has a weight and the output elements are to be enumerated sorted by their weights, are important and well studied problems in database theory. On the one hand, the first framework already covers the well-known case of regular document spanners, while the latter setting covers several practically relevant tasks that cannot be described in the unweighted setting.

It is known that in the unweighted case this problem can be solved with linear time preprocessing $O(|D|)$ and output-linear delay $O(|s|)$ in data complexity, where D is the input data and s is the current output element. For the weighted case, Bourhis, Grez, Jachiet, and Riveros [ICDT 2021] recently designed an algorithm with linear time preprocessing, but the delay of $O(|s| \cdot \log |\mathsf{D}|)$ depends on the size of the data.

We first show how to leverage the existing results on enumerating shortest paths to obtain a simple alternative algorithm with linear preprocessing and a delay of $O(|s_i| + \min\{\log i, \log |\mathsf{D}|\})$ for the i^{th} output element s_i (in data complexity); thus, substantially improving the previous algorithm. Next, we develop a technically involved rounding technique that allows us to devise an algorithm with linear time preprocessing and output-linear delay $O(|s|)$ with high probability. To this end, we combine tools from algebra, high-dimensional geometry, and linear programming.

CCS Concepts: • **Information systems** → *Information retrieval*; • **Theory of computation** → *Automata extensions*; *Regular languages*; *Design and analysis of algorithms*; *Database query languages (principles).*

Additional Key Words and Phrases: Information Extraction, Document Spanners, Ranked Enumeration

ACM Reference Format:
Paweł Gawrychowski, Florin Manea, and Markus L. Schmid. 2024. Revisiting Weighted Information Extraction: A Simpler and Faster Algorithm for Ranked Enumeration. *Proc. ACM Manag. Data* 2, 5 (PODS), Article 222 (November 2024), 19 pages. https://doi.org/10.1145/3695840

1 Introduction

The term *information extraction* usually refers to the task of compiling structured information from text documents. Its most famous instance is the framework of so-called *document spanners* introduced in the seminal paper [11] (see the surveys [2, 25, 26] or [7, 8, 15, 24] for some recent publications on document spanners). A document spanner (over a set $\mathcal{X}$ of variables) is a function that maps a document D over some alphabet Σ to a finite table with a column for each variable from $\mathcal{X}$ and the entries of the cells of the table are just pairs of positions of D. This can be illustrated as follows (where $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $\mathcal{X} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$):

Authors' Contact Information: Paweł Gawrychowski, gawry@cs.uni.wroc.pl, University of Wrocław, Poland; Florin Manea, florin.manea@informatik.uni-goettingen.de, Computer Science Department and CIDAS, Universität Göttingen, Germany; Markus L. Schmid, MLSchmid@MLSchmid.de, Humboldt-Universität zu Berlin, Germany.

ACM 2836-6573/2024/11-ART222
https://doi.org/10.1145/3695840

$\mathsf{D} = \mathsf{abbabccabc} \quad \Longrightarrow$

x	y	z
(2, 5)	(4, 7)	(1, 10)
(3, 5)	(5, 8)	(4, 7)
(1, 3)	(3, 10)	(2, 4)
...	...	...

The pairs (i, j) are called *spans* and are interpreted as pointers to factors of D, e. g., $(4, 7)$ refers to abcc, since this is the factor that starts at position 4 and ends at position 7. A row of the table is called a *span tuple* and the whole table is called a *span relation*. Spanners are a suitable formalisation of relevant information extraction tasks (see the introductions of the papers mentioned above).

The most fundamental class of spanners is the class of regular spanners (see [11]), which can be described by finite automata or regular expressions that represent span tuples by using brackets $\langle_{\mathsf{x}} \ldots {}_{\mathsf{x}}\rangle$ with $\mathsf{x} \in \mathcal{X}$ for marking the start and end points of the spans in the input string. For example,

$$r = (\mathsf{c} + \mathsf{b})^* \langle_{\mathsf{x}} (\mathsf{a} + \mathsf{b})^* {}_{\mathsf{x}}\rangle \mathsf{a}^* \langle_{\mathsf{y}} \mathsf{a}^* {}_{\mathsf{y}}\rangle \mathsf{c}^*$$

is a regular expression that describes a regular spanner over $\mathcal{X} = \{\mathsf{x}, \mathsf{y}\}$: From the input document $\mathsf{cbcabaaac}$, it can extract the span tuple $((4, 6), (8, 8))$ since $\mathsf{cbc}\langle_{\mathsf{x}}\mathsf{aba}_{\mathsf{x}}\rangle\mathsf{a}\langle_{\mathsf{y}}\mathsf{a}_{\mathsf{y}}\rangle\mathsf{c} \in L(r)$, or the span tuple $((4, 5), (6, 8))$ since $\mathsf{cbc}\langle_{\mathsf{x}}\mathsf{ab}_{\mathsf{x}}\rangle\langle_{\mathsf{y}}\mathsf{aaa}_{\mathsf{y}}\rangle\mathsf{c} \in L(r)$.

A regular spanner can also be described by a finite transducer that simply marks some of the input positions (i. e., it marks input positions with the special markers $\langle_{\mathsf{x}}, {}_{\mathsf{x}}\rangle$, $\langle_{\mathsf{y}}, {}_{\mathsf{y}}\rangle$, etc.). This suggests the more general information extraction framework of *annotation transducers* that properly extends regular spanners and that has already been used in [7, 24]. In this work, we will also focus on annotation transducers; the case of regular spanners is therefore a special case of our results.

An annotation transducer is a classical finite automaton, but transitions are not only labelled by an input symbol from Σ, but also by a marker symbol from a set $\mathcal{M}$ of *markers*, which includes the special *empty marker* $\diamond$. Consequently, a computation of an annotation transducer marks the letters of the input by elements from $\mathcal{M}$, so it outputs a string over the alphabet $\mathcal{M} \times \Sigma$ whose restriction to Σ equals the input. The *output set* consists then in all the outputs of accepting computations, but we represent them in a contracted way: marked input letters are represented by their positions and all letters marked with the empty marker are ignored. For example, an annotation transducer may mark an input document abaabc as $(\diamond, \mathsf{a})(\gamma, \mathsf{b})(\delta, \mathsf{a})(\diamond, \mathsf{a})(\diamond, \mathsf{b})(\delta, \mathsf{c})$, which means that the corresponding *output tuple* is $((\gamma, 2), (\delta, 3), (\delta, 6))$.

The arguably most important computational problem for information extraction based on annotation transducers is that of producing the set of output tuples for a given annotation transducer $\mathcal{T}$ and a given input document D, i. e., the associated query evaluation problem. As usual for query evaluation problems, we consider this in the following enumeration perspective: After a preprocessing with a running time that is only linear in $|\mathsf{D}|$, we wish to enumerate all the output tuples with an output-linear delay, i. e., the time needed to produce the next output tuple is linear in the size of this tuple, but independent from $|\mathsf{D}|$. Measuring only in dependency on $|\mathsf{D}|$ and neglecting the dependency on the query size $|\mathcal{T}|$ covers the plausible scenario where the data is rather large, while the query is human-readable and therefore comparatively small. This complexity measurement is standard in query evaluation problems and is also called *data complexity*. In this setting, linear-time preprocessing and output-linear delay in data complexity is the best we can hope for. The fact that the size of the output tuples is not lower bounded in the size of the input data can make it challenging to enumerate them with output-linear delay.

It is well-known that this enumeration task can be solved with linear preprocessing and output-linear delay in data complexity. Phrased for regular spanners, this has been shown in [2, 3, 12], and, in the context of MSO-enumeration over trees (which properly extend strings), it has already

been solved in [6]. It also follows from the result of [24], which extends the scenario to the SLP-compressed setting.

We are interested in the extension to the weighted scenario, which we will discuss next.

Weighted Information Extraction and Ranked Enumeration. One of the main motivations for the enumeration perspective when approaching query evaluation problems is that the result set is potentially very large (even exponentially), which means that a user has to wait a long time until the whole result set is materialised. Instead, the idea is to provide the user with the first elements of the result set as fast as possible (hence only linear dependency on |D| of the preprocessing) and the possibility to receive a new element on demand as fast as possible (hence, a delay that is linear with respect to size of the next element, but independent from |D|). While this setting covers many practically relevant aspects, it ignores that most likely not all answers to a query are equally interesting, especially if there are many answers. Thus, in the worst case, the relevant answers may appear at the very end of the enumeration and the user again has to wait for a long time. The weighted case of information extraction is a natural way to deal with this issue: Every output tuple has a weight and the enumeration algorithm should produce all output tuples sorted by their weights.

The weighted case of information extraction has been investigated in [7–9]. It can be easily formalised by simply adding weights to the model of annotation transducers. More precisely, a *cost transducer* is an annotation transducer whose transitions are not only labelled by an input symbol and a marker, but also by a *weight*, which is an element from an ordered abelian group[1] $\mathbb{G}$. In this setting, every accepting computation does not only produce an output tuple, but also allocates a weight to it (i. e., the sum of all the weights on the transitions of the computation). Now, every possible output tuple has a weight and, since these weights are ordered, the output tuples are ordered. As shown in [7], cost transducers cover the expressive power of cost functions expressed in monadic second order logic.

Consequently, we can talk about *ranked enumeration*, which is the same task as described above, just that the output tuples must be produced in increasing order with respect to their weights (or decreasing order, which amounts to the same problem). The obvious question is of course, whether we can still achieve linear preprocessing and output-linear delay in this setting. Before we explain what is known about the problem of ranked information extraction and outline our respective contributions, let us discuss some relevant use-cases.

Use-Cases for Ranked Information Extraction. We first discuss some application scenarios of ranked information extraction in the context of regular spanners.

Assume that we want to compile a table with columns “name”, “address”, “country”, “email”, “age”, etc. from a text document that contains customer information, i. e., each row of the extracted table then represents a customer. A respective spanner would therefore have a variable per field “name”, “address”, etc. (we neglect the actual structure of the spanner, which anyway depends on the format of the data). However, our data might be incomplete in the sense that a customer might be missing the field “country” or “age”. In this case, rather than ignoring this customer altogether, it seems more useful to still extract a span tuple for it, but with "country" or "age" being undefined. Hence, spanners with undefined variables is a practically relevant extension first considered in [22] and since then generally adopted to the model.

Nevertheless, we might have a strong priority of complete span tuples over incomplete ones and we wish this to be reflected in our enumeration algorithm, i. e., it seems natural that we want to first get all complete tuples, then those with one undefined variable (possibly ordered with

[1]For simplicity, let us here assume that $\mathbb{G} = (\mathbb{N}, +, 0, \leq)$; formal definitions follow below.

respect to some preferences with respect to the individual variables, e. g., a missing “age” field is less problematic than a missing “email” field), then those with two missing variables and so on. It is easy to see that this is covered by ranked enumeration based on cost transducers (i. e., by adding weights to transitions that start allocating a span to a variable).

But also without undefined variables, there are interesting applications of document spanners in the weighted case. For example, we can use weights to force the enumeration of the customer entries to be alphabetically ordered by the "name" field (by simply adding weights to the transitions that read the first letter of the “name” field). We can also prioritise span tuples with long spans over span tuples with shorter ones by adding a negative weight to the transitions that read those parts of the input that are extracted by variables. Or we could prioritise span-tuples whose spans are not too far apart from each other by giving non-negative weights to the letters read between spans.

A particularly interesting application of weighted information extraction seems to be that it provides a way of handling uncertain data. Assume that our textual data is faulty in the sense that some letters might have been replaced by other letters. This might be due to typos caused by humans (e. g., if we are dealing with data entered by hand), or errors introduced by technical problems when storing or communicating the data, or due to the fact that the data results from scientific experiments and exhibits the typical margin of measurement errors. A user might have knowledge of the format of the data, but not where the actual errors occur. This would not be too bad, if it would only mean that we extract the errors as well, but, in fact, a single erroneous letter can also mean that our information extraction query does not extract anything anymore. For example, if some conversion of the data has changed all occurrences of the "@"-sign to the letter "a", then an annotation transducer that marks the factors of a text document that satisfy the format of email addresses would be quite useless, since it does not discover any email addresses anymore and therefore does not produce any result.

In general, we can deal with this scenario by allowing the annotation transducer to make mistakes as follows. For every original transition for some letter $x \in \Sigma$, we simply add transitions for every other letter from Σ as well (with the same source and origin state). The idea is of course that this modified annotation transducer can also extract email addresses with the "@"-sign swapped into an "a"-sign, by using one of the new alternative transitions for this faulty symbol. But the modified transducer is way too general and will extract a lot of rubbish. So we could just punish the newly added transitions with a weight of 1, while all other original transitions have a weight of 0. If we now enumerate in increasing order, then we first get all the original results (i. e., those extracted by the original transducer), then all results that the transducer can extract by correcting exactly one symbol (e. g., email addresses with "a" instead of "@"), then those with two corrections and so on. The same rubbish as before will be produced, but only at the end of the enumeration, which can be interrupted as soon as we get results with a number of corrections that is over some threshold.

Obviously, in actual applications, we would fine-tune this setting by adding the mistake-transitions only for certain symbols that are known to be crucial, like the “@”-sign for extracting email addresses, or by using more complicated weighing schemes, reflecting, e.g., some sort of similarity between the newly added transitions leaving a state and the already existing ones.

Our Contribution. The central computational task investigated in this work is as follows (formal definitions follow soon): Given a text document D and an *unambiguous*[2] cost transducer $\mathcal{T}$, we wish to enumerate all output tuples in increasing order by their weights.

It is shown in [7] that this can be achieved with a preprocessing of $O(|\mathcal{T}||D|)$ and every output tuple s has delay $O(|s| \cdot \log(|\mathcal{T}||D|))$.

[2]This means that every output is produced by exactly one possible computation.

Remark 1.1. Note that for the rest of the paper, we state all running times in dependency on $|\mathrm{D}|$ and $|\mathcal{T}|$, i. e., we measure the running times of our algorithms in *combined complexity*. In this way, it will be clear that the running times of our algorithms have only a polynomial dependency on the query size. However, we keep in mind that our objective is to find algorithms optimised towards the data complexity perspective, i. e., the paramount goal is a preprocessing time that is linear in $|\mathrm{D}|$ and a delay that is linear in the size of the next enumerated element and independent from $|\mathrm{D}|$. In particular, we do not optimise our algorithms regarding the dependency on the query size (other than making sure that this dependency is polynomial).

So in [7], the dependency of the preprocessing on the data size is only linear, but, unfortunately, in the delay there is a logarithmic factor that depends on $|\mathrm{D}|$. In big data scenarios this would significantly slow down the enumeration compared to the unweighted case. Given the high relevance of ranked information extraction, this is unsatisfactory, and brings the challenge of determining whether output-linear delay can be achieved, i. e., can we solve ranked information extraction in the same running times as in the unweighted case?

We approach the problem by first phrasing it in a simpler way as a problem of computing shortest paths of an edge-weighted DAG. This allows us to apply known algorithmic techniques for weighted DAGs, in particular Eppstein's DAG of shortest paths (see [10]). As a result, we obtain a simply structured enumeration algorithm with preprocessing of $\mathrm{O}(|\mathcal{T}||\mathrm{D}|)$ and, if $s_1, s_2, \ldots, s_k$ is the sorted enumeration of the output tuples, then the delay of s_i is $\mathrm{O}(|s_i| + \min\{\log i, \log(|\mathcal{T}||\mathrm{D}|)\})$ for every $i \in [k]$. On the one hand, this improves the enumeration algorithm of [7] since the logarithmic term in the delay $\mathrm{O}(|s_i| + \min\{\log i, \log(|\mathcal{T}||\mathrm{D}|)\})$ is only additive, while it is a factor in the delay $\mathrm{O}(|s_i| \cdot \log(|\mathcal{T}||\mathrm{D}|))$ of the enumeration algorithm of [7]. Moreover, since the additive logarithmic term of the delay for the i^{th} element is actually the minimum of $\log(|\mathcal{T}||\mathrm{D}|)$ and $\log i$, we know that this additive term is very small for the first elements of the enumeration and only then slowly increases to $\log(|\mathcal{T}||\mathrm{D}|)$ after we have seen $\Omega(|\mathrm{D}||\mathcal{T}|)$ elements. This is particularly desirable for ranked enumeration, where we have a guarantee that the most important elements (according to the weights) are produced at the beginning of the enumeration. In particular, for application scenarios where we are only interested in, say, the first 25 elements, the additive term is indeed constant and therefore negligible.

As our main result, we extend this algorithm in a non-trivial way in order to address the question of whether we can achieve output-linear delay. At first, it might seem that the logarithmic factor is unavoidable, as outputting the results in the sorted order is as hard as sorting, which does need a linear-logarithmic time in the comparison model. However, we make the following observation. For fixed D and $\mathcal{T}$, all the weights of output tuples have the form $\sum_{i=1}^{t} \alpha_i g_i$, where $\alpha_1, \alpha_2, \ldots, \alpha_t \in \mathbb{N} \cup \{0\}$ and $g_1, g_2, \ldots, g_t$ are the group elements from $\mathbb{G}$ that occur on transition of the cost transducer, so in particular t is bounded by the query size $|\mathcal{T}|$. Thus, while we may need to sort many weights, they cannot be all arbitrarily complicated.

At the heart of our output-linear algorithm lies a non-trivial rounding procedure, which allows us to show that we can sort any $N \geq |\mathrm{D}|$ such sums as defined above in time $\mathrm{O}(\mathrm{poly}(|\mathcal{T}|)N)$, with high probability (see Lemma 5.3). Formally, the procedure always terminates and returns the correct answer, and for any constant $c \in \mathbb{N}$ works in time $\mathrm{O}(\mathrm{poly}(|\mathcal{T}|)N)$ with probability at least $1 - 1/N^c$. Finally, our enumeration algorithm works as follows. In the preprocessing, we pre-compute the output tuples with ranks $1, 2, \ldots, |\mathrm{D}|$ and we sort them according to their weights in linear time with high probability in the size of the input. Then, in the first stage of the enumeration phase, we output these pre-computed elements, which buys us enough time to pre-compute and sort the output tuples with ranks $|\mathrm{D}| + 1, |\mathrm{D}| + 2, \ldots, 2|\mathrm{D}|$. In the next stage, we therefore output the tuples with ranks $|\mathrm{D}| + 1, |\mathrm{D}| + 2, \ldots, 2|\mathrm{D}|$ and pre-compute and sort the tuples with ranks

$2|\mathrm{D}|+1, 2|\mathrm{D}|+2, \ldots, 4|\mathrm{D}|$, and so on. For the pre-computation of the elements, we use again Eppstein's DAG of shortest paths [10] already mentioned above and a result by Frederickson [13]. This yields an algorithm with linear preprocessing and output-linear delay with high probability (see Theorem 5.6), i. e., an algorithm with the same running time guarantees as in the unweighted case with high probability.

Related Work. As mentioned above, we directly improve the results from [7].

The connection between ranked enumeration problems in database theory and computing the shortest paths of a graph (and Eppstein's approach) has also successfully been explored in [27]. However, [27] is concerned with ranked enumeration of conjunctive queries and, moreover, the output paths are represented as explicit listings (i.e., the path size is proportional to the number of its edges) instead of the potentially much smaller labels, as in [7] and as we also require in our setting.

In [1], a work that is also related to [7], the authors consider ranked enumeration of MSO-queries over trees. However, the MSO-queries of [1] have only free FO-variables, but no free set-variables, which means that this setting does not extend MSO on strings as considered in our paper (or in [7]). Moreover, the ranking function used in [1] requires some additional properties.

2 Preliminaries

Algebra. A *free monoid* is a tuple $(\mathbb{M}, \circ_{\mathbb{M}}, \varepsilon)$, where $\mathbb{M} = \Sigma_{\mathbb{M}}^*$ for a finite alphabet $\Sigma_{\mathbb{M}}$. For free monoids, we assume that each element from $\Sigma_{\mathbb{M}}$ can be stored in a single machine word, and that, for every $x, y \in \mathbb{M}$, calculating $x \circ_{\mathbb{M}} y$ takes constant time.

A *group* is a tuple $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}})$, where $\mathbb{G}$ is a set of elements, $+_{\mathbb{G}}$ is an associative binary operation on $\mathbb{G}$, $0_{\mathbb{G}} \in \mathbb{G}$ is a neutral element for $+_{\mathbb{G}}$ (i. e., $0_{\mathbb{G}} +_{\mathbb{G}} x = x +_{\mathbb{G}} 0_{\mathbb{G}} = x$ for every $x \in \mathbb{G}$) and every $x \in \mathbb{G}$ has an inverse $-x \in \mathbb{G}$ (i. e., $x +_{\mathbb{G}} (-x) = 0$). A group is *abelian*, if $+_{\mathbb{G}}$ is also commutative. An *ordered* group is a tuple $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}}, \leq_{\mathbb{G}})$, where $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}})$ is a group, and $\leq_{\mathbb{G}}$ is a total order over $\mathbb{G}$ that respects $+_{\mathbb{G}}$ (i. e., $x_1 \leq_{\mathbb{G}} x_2$ implies $x_1 +_{\mathbb{G}} y \leq_{\mathbb{G}} x_2 +_{\mathbb{G}} y$ for every $y \in \mathbb{G}$); for an ordered group, we denote by $|x|$ the greater element from x and $-x$ with respect to the order $\leq_{\mathbb{G}}$. We will assume that $\mathbb{G}$ has an effective representation, meaning that every element of $\mathbb{G}$ is stored in a single machine word, calculating $x +_{\mathbb{G}} y$ takes constant time, and so does checking if $x \leq_{\mathbb{G}} y$.

Graphs. Let $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}}, \leq_{\mathbb{G}})$ be a fixed ordered abelian group, called the *weight group*, and let $(\mathbb{M}, \circ_{\mathbb{M}}, \varepsilon)$ be a fixed free monoid, called the *label monoid*.

We now define directed graphs with edges that have a weight from $\mathbb{G}$ and a label from $\mathbb{M}$. More formally, a *directed graph with edge weights from* $\mathbb{G}$ *and edge labels from* $\mathbb{M}$ is a tuple $G = (V, E, \mathrm{weight}, \mathrm{label})$, where V is a set of nodes, E is a set of edges, where, for every $e \in E$, $\mathrm{source}(e) \in V$ is the *source* of w and $\mathrm{target}(e) \in V$ is the *target* of w, the function $\mathrm{weight} : E \to \mathbb{G}$ is the *weight function* and $\mathrm{label} : E \to \mathbb{M}$ is the *label function*. For simplicity, we also represent an edge e in the form $(\mathrm{source}(e), \mathrm{weight}(e), \mathrm{label}(e), \mathrm{target}(e))$. A path in G is a sequence of edges

$$P = ((v_0, w_1, \beta_1, v_1), (v_1, w_2, \beta_2, v_2), \ldots, (v_{k-1}, w_k, \beta_k, v_k)) .$$

The *weight* of the path P is defined by $\mathrm{weight}(P) = \sum_{i=1}^{k} w_i$ and the *label* of the path P is defined by $\mathrm{label}(P) = \prod_{i=1}^{k} \beta_i$. We also denote the weight of an edge or a path as its *length*, and by *shortest* path, we mean a path of minimal length. For $u, v \in V$, a u-to-v-path is any path from u to v.

We measure the size of a graph $G = (V, E, \mathrm{weight}, \mathrm{label})$ with edge weights from $\mathbb{G}$ and edge labels from $\mathbb{M}$ as $|G| = |V| + |E|$. This is justified, since every edge e has constant size (due to the fact that $\mathrm{weight}(e)$ and $\mathrm{label}(e)$ can be stored in single machine words).

Nondeterministic Finite Automata. A *nondeterministic finite automaton* (*NFA* for short) is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of *states*, Σ is the finite *input alphabet*, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$

is the *transition function*, $q_0 \in Q$ is the *initial state* and $F \subseteq Q$ is the set of *final states*. The semantics are defined as usual, but we shall state them in detail for the transducer models defined later.

Enumeration Algorithms. An *enumeration* of a finite set A is any sequence $(s_1, s_2, \ldots, s_m)$ such that $|A| = m$ and $A = \{s_1, s_2, \ldots, s_m\}$. A finite set A is called $\mathbb{G}$-*ranked*, if every $s \in A$ has a weight $\mathsf{weight}(s) \in \mathbb{G}$. A *ranked enumeration* of a $\mathbb{G}$-ranked set A is any enumeration $(s_1, s_2, \ldots, s_m)$ of A that satisfies $\mathsf{weight}(s_1) \leq_{\mathbb{G}} \mathsf{weight}(s_2) \leq_{\mathbb{G}} \ldots \leq_{\mathbb{G}} \mathsf{weight}(s_m)$.

An *enumeration problem* is a function P that maps an input I to a finite *output set* $P(I)$, the elements of which are called *output elements*. An enumeration problem P is called $\mathbb{G}$-*ranked*, if, for every input I, the output set $P(I)$ is a $\mathbb{G}$-ranked set.

An *enumeration algorithm* for an enumeration problem P is an algorithm A that, on input I, produces an enumeration of $P(I)$. A *ranked enumeration algorithm* for a $\mathbb{G}$-ranked enumeration problem P is an algorithm A that, on input I, produces a $\mathbb{G}$-ranked enumeration of $P(I)$. We assume that any enumeration algorithm first performs a *preprocessing phase*, which is then followed by an *enumeration phase*, in which the enumeration is produced.

Let A be an enumeration algorithm for some enumeration problem P. The *preprocessing time* of A on input I is the running-time of the preprocessing phase of A on input I. If $(s_1, s_2, \ldots, s_m)$ is the output of A on input I, then, for every $i \in [m]$, the *delay of* s_i is the time that elapses between the end of completely producing the element s_{i-1} (or the end of the preprocessing phase if $i = 1$) and the end of completely producing the next output element s_i. The delay of A on some input I is the maximum over all delays of any output element. The preprocessing time and the delay of the algorithm A is the maximum preprocessing time and maximum delay, respectively, over all possible inputs of length at most n (viewed as a function of n).

Enumeration algorithms model the requirement that in practice the user cannot afford to wait until the set $P(I)$ is completely computed before receiving an answer. Instead, the user should be quickly provided with some answer and can then stop the enumeration as soon as enough answers have been seen. The ranked perspective even provides a way to postulate that the most important answers are enumerated first, which is, in most practical scenarios, a highly desirable feature.

However, in the case that the full output set is extremely large (i. e., exponential in the size of the queried data), an enumeration algorithm may have to deal with an exponential number of elements. This is not a problem if these output elements are just written on some output storage, but if the algorithm needs to perform computations on an exponential number of output elements, one runs into problems with respect to addressing and working with these elements. Clearly, this only happens if the output is indeed of exponential size, and in the very unlikely case that the user is really interested in seeing all of these exponentially many output elements and wants the algorithm to run all the way until the end. Nevertheless, on a formal level, we have to deal with this issue, which is why we also define polynomially-bounded versions of ranked enumeration problems, where the ranked enumeration stops after a polynomially number of elements have been produced.

Let P be a $\mathbb{G}$-ranked enumeration problem and let $f : \mathbb{N} \to \mathbb{N}$ be some polynomial function. The f-*bounded* version of P, denoted by P^f, is the restriction of P to the case where, on input I, we only enumerate the $f(|I|)$ smallest elements of $P(I)$ (with respect to their weights). Let us define this more formally. If I is an input for P, where $P(I) = \{s_1, s_2, \ldots, s_m\}$ with $\mathsf{weight}(s_i) \leq_{\mathbb{G}} \mathsf{weight}(s_{i+1})$ for every $i \in [m-1]$, then $P^f(I) = \{s_1, s_2, \ldots, s'_m\}$, where $m' = \min\{m, f(|I|)\}$. Note that P^f is a $\mathbb{G}$-ranked enumeration problem in the sense defined at the beginning of this subsection.

Remark 2.1. In the following, we let f be some fixed polynomial that we will use for polynomially-bounded versions of ranked enumeration problems. Our algorithms do not depend on f and they will work for every choice of f. For example, taking f to be linear covers those practical scenarios

where we can assume that users never materialise a number of query answers that exceed the total size of the queried data itself; in the case of enumeration algorithms with linear time preprocessing, this models the case when the user allows roughly the same time for seeing the answers as for the preprocessing.

Computational Model. The computational model we use is the standard unit-cost word RAM with word-size ω (meaning that each memory word can hold ω bits). It is assumed that this model allows processing inputs of size n, where $\omega \geq \log n$; in other words, the size n of the data never exceeds (but, in the worst case, is equal to) 2^ω. Intuitively, the size of the memory word is determined by the processor, and larger inputs require a stronger processor (which can, of course, deal with much smaller inputs as well). Indirect addressing and basic arithmetical operations on such memory words are assumed to work in constant time. Note that numbers with ℓ bits are represented in $O(\ell/\omega)$ memory words, and working with them takes time proportional to the number of memory words on which they are represented. This is a standard computational model for the analysis of algorithms, as defined in [14], and used in classical works related to basic problems such as integer-sorting, see, e.g., [4, 5, 17].

In the problem considered here, we assume that we are given a text document D, of length n, over some input alphabet Σ, and an unambiguous cost transducer $\mathcal{T}$ (to be formally defined in the next section). As explained in Section 1, the runs of $\mathcal{T}$ on D determine a set of weighted output tuples, which therefore defines a ranked enumeration problem. We wish to find ranked enumeration algorithms for this problem.

As common in query evaluation tasks (and as also explained in Remark 1.1), the dependency on the query size $|\mathcal{T}|$ is considered small in comparison to the data size |D|. This is why we are mainly concerned with running times that have a low dependency on the data size. More precisely, for the preprocessing phase, we aim for a running time with only linear dependency on |D|, while the delay should ideally be independent from |D| (and only linear in the size of the output element). Assuming that we have to read the input data at least once, this is the best we can hope for with respect to the dependency on the data size. As far as the dependency on the query size is concerned, it is desired (although this is not a main concern) to still be polynomial.

For dealing with the ranked setting, our first algorithm (Theorem 5.1) has to maintain a heap data structure whose size grows with the size of the set of output elements, while our second algorithm (Theorem 5.5) has to repeatedly sort a large number of output elements. In both cases, the space complexity can become exponential if we are dealing with instances that lead to an exponential number of output elements and if the enumeration phase is carried out completely. Since such pathological cases cannot be easily handled by the RAM model with logarithmic word size (e.g., in the case when the input size is roughly 2^ω, with ω being the word size), we only consider the f-bounded variant of our ranked enumeration problem (see the explanations from above). This is a mere technicality that is required to define our algorithms and respective running time guarantees in a sound way, using the word RAM model. In all reasonable application scenarios, it is to be assumed that ranked enumeration algorithms are not run until the end in the case that the output is exponentially large.

3 Ranked Information Extraction

We next thoroughly formalise the enumeration problem investigated in this work, which has been discussed in the introduction on an intuitive level.

In the following, let $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}}, \leq_{\mathbb{G}})$ be a fixed ordered abelian group. Let Σ be a finite *input alphabet* and let $\mathcal{M}$ be a finite set of *markers*, including a designated *empty marker* $\diamond \in \mathcal{M}$. We assume that every element of Σ and every element of $\mathcal{M}$ can be stored in a single machine word.

Syntactically, a *cost transducer over* $\mathbb{G}$ is an NFA whose transitions are labelled by elements (b, w, γ), where $b \in \Sigma$, $w \in \mathbb{G}$ and $\gamma \in \mathcal{M}$, i. e., an NFA $\mathcal{T} = (Q, (\Sigma \times \mathbb{G} \times \mathcal{M}), \delta, q_0, F)$. For convenience, we treat δ as a subset of $Q \times \Sigma \times \mathbb{G} \times \mathcal{M} \times Q$. The semantics are defined as follows.

A *run* of $\mathcal{T}$ on some input $\mathsf{D} \in \Sigma^*$ is a sequence $\rho = p_0 \to p_1 \to \ldots \to p_n$ of states $p_i \in Q$, $1 \leq i \leq n$, such that $|\mathsf{D}| = n$ and, for every $i \in [n]$, there is a transition $(p_{i-1}, \mathsf{D}[i], w_i, \gamma_i, p_i)$. We also write runs in the form

$$\rho = p_0 \xrightarrow{\mathsf{D}[1], w_1, \gamma_1} p_1 \xrightarrow{\mathsf{D}[2], w_2, \gamma_2} \ldots \xrightarrow{\mathsf{D}[n], w_n, \gamma_n} p_n \,.$$

The *weight* of ρ is $\mathsf{weight}(\rho) = \sum_{i=1}^{n} w_i$, while its *output* is $\mathsf{out}(\rho) = (\gamma_{j_1}, j_1)(\gamma_{j_2}, j_2) \ldots (\gamma_{j_m}, j_m)$, where $1 \leq j_1 < j_2 < \ldots < j_m \leq n$ are exactly the positions of ρ with $\gamma_{j_\ell} \neq \diamond$ for every $\ell \in [m]$. If $p_0 = q_0$ and $p_n \in F$, then the run ρ is *accepting*.

The cost transducer $\mathcal{T}$ is *unambiguous*, if there are no two distinct accepting runs ρ_1 and ρ_2 on any input $\mathsf{D} \in \Sigma^*$ such that $\mathsf{out}(\rho_1) = \mathsf{out}(\rho_2)$.

Remark 3.1. In the following, we assume that cost-transducers are unambiguous (an assumption also made in [7]). This is a proper restriction of the model of cost-transducers, since general cost-transducer may have two runs on the same input that produce the same output, but with different weights. However, in our setting of information extraction, weights are used to allocate a priority to each output element, which in turn induces the desired order of the output elements. Consequently, the possibility to allocate several different weights to the same output element is not a useful feature. Note, however, that different output elements may have the same weights.

Finally, the function $[\![\mathcal{T}]\!] : \Sigma^* \to \mathcal{P}((\mathcal{M} \times \mathbb{N})^*)$ is defined by

$$[\![\mathcal{T}]\!](\mathsf{D}) = \{\mathsf{out}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{T} \text{ on } \mathsf{D}\} \,.$$

Remark 3.2. Note that for the sake of simplicity, we define cost transducers in a sightly different yet equivalent way compared to [7]. The main purpose of cost transducers in [7] is to express MSO-cost functions, which map set variables to sets of positions of the input string. Consequently, the cost transducers of [7] mark every position of the input string by a whole set of output symbols (which encode the set variables that contain this position). For simplicity, we just use single markers and a designated empty marker that represents the empty set of output symbols (note that positions marked with the empty marker do not appear in the output, just like in the setting of [7], where the positions marked with the empty set of output symbols do not appear in the output). Another difference is that the model from [7] can also allocate weights to the initial and final states. This is mere syntactical sugar, since by a simple modification of the underlying automaton, such initial and final weights can be moved onto the transitions leaving or entering the initial or final states, respectively.

The problem CT-Enum is the ranked enumeration problem defined as follows. For a given unambiguous cost transducer $\mathcal{T}$ and a document $\mathsf{D} \in \Sigma^*$, let $\mathsf{CT\text{-}Enum}(\mathcal{T}, \mathsf{D}) = [\![\mathcal{T}]\!](\mathsf{D})$.

Bourhis et al. [7] give a ranked enumeration algorithm for $\mathsf{CT\text{-}Enum}^f$. Worth noting, Bourhis et al. [7] do not introduce their result in the context of bounded ranked enumeration, but their algorithm also seems to be using exponential space, in the worst case, when all the solutions to CT-Enum are output. Therefore, we state their result in our setting, for the sake of soundness and uniformity.

Theorem 3.3 (Bourhis et al. [7])**.** *There is a ranked enumeration algorithm for* $\mathsf{CT\text{-}Enum}^f$ *such that, on input* $\mathcal{T}$ *and* D*, the preprocessing time is* $O(|\mathcal{T}||\mathsf{D}|)$ *and every output element* s *has delay* $O(|s| \cdot \log(|\mathcal{T}||\mathsf{D}|))$.

Before we present our algorithmic approaches to the ranked enumeration problem CT-Enum^f, we formulate it as a problem of enumerating shortest paths in an edge-labelled weighted graph.

Reducing CT-Enum to Shortest Path Enumeration. Let $\mathsf{D} \in \Sigma^*$ with $|\mathsf{D}| = n$ and let $\mathcal{T} = (Q, (\Sigma \times \mathbb{G} \times \mathcal{M}), \delta, q_0, F)$ be an unambiguous cost transducer. Without loss of generality, we can assume that $F = \{q_f\}$. Indeed, this can be achieved by first adding a new final state q_f and making all former final states non-final. Then, for every transition (p, b, w, γ, q) where q is a former final state, we add a transition (p, b, w, γ, q_f). Now we have a single final state q_f and, for any input string, the cost transducer has the same set of outputs. Moreover, unambiguity is maintained: If there are two different accepting runs on the same input with the same output (i. e., unambiguity is violated), then we can replace the last transitions $(p_1, b, w_1, \gamma, q_f)$ and $(p_2, b, w_2, \gamma, q_f)$ of these runs by the transitions $(p_1, b, w_1, \gamma, q_1)$ and $(p_2, b, w_2, \gamma, q_2)$, where q_1 and q_2 are former final states; thus, unambiguity was already violated before the construction, which is a contradiction.

We define a monoid $(\mathbb{M}, \circ_{\mathbb{M}}, \varepsilon)$, where $\mathbb{M}$ is the set of strings over the alphabet $(\mathcal{M} \setminus \{\diamond\}) \times \{1, \ldots, n\}$ with the empty word being the neutral element, i. e., $\mathbb{M} = ((\mathcal{M} \setminus \{\diamond\}) \times \{1, \ldots, n\})^*$, and the operation $\circ_{\mathbb{M}}$ is string-concatenation (i. e., ε plays the role of the empty string).

Next, we define a directed graph $G_{\mathsf{D},\mathcal{T}} = (V_{\mathsf{D},\mathcal{T}}, E_{\mathsf{D},\mathcal{T}}, \mathsf{weight}, \mathsf{label})$ with edge weights from $\mathbb{G}$ and edge labels from $\mathbb{M}$ as follows. The set of nodes is $V_{\mathsf{D},\mathcal{T}} = \{(q, i) \mid q \in Q, 0 \leq i \leq n\}$. For every $i \in [n]$ and every transition $(p, \mathsf{D}[i], w, \gamma, q) \in \delta$ with $\gamma \neq \diamond$, there is an edge $((p, i-1), w, (\gamma, i), (q, i)) \in E_{\mathsf{D},\mathcal{T}}$, and for every transition $(p, \mathsf{D}[i], w, \diamond, q) \in \delta$, there is an edge $((p, i-1), w, \varepsilon, (q, i)) \in E_{\mathsf{D},\mathcal{T}}$. It can be easily seen that there is a weight preserving one-to-one-correspondence between accepting runs of $\mathcal{T}$ on D and $(q_0, 0)$-to-(q_f, n)-paths of $G_{\mathsf{D},\mathcal{T}}$. More precisely, there is a run

$$\rho = q_0 \xrightarrow{\mathsf{D}[1], w_1, \gamma_1} q_1 \xrightarrow{\mathsf{D}[2], w_2, \gamma_2} \ldots \xrightarrow{\mathsf{D}[n], w_n, \gamma_n} q_n \,.$$

of $\mathcal{T}$ on D with $q_n = q_f$ if and only if

$$P = ((q_0, 0), w_1, \psi_1, (q_1, 1)), ((q_1, 1), w_2, \psi_2, (q_2, 2)), \ldots, ((q_{n-1}, n-1), w_n, \psi_n, (q_n, n))$$

is a path in $G_{\mathsf{D},\mathcal{T}}$ with $q_n = q_f$ and, for every $i \in \{1, 2, \ldots, n\}$, $\psi_i = (\gamma_i, i)$ if $\gamma_i \neq \diamond$, and $\psi_i = \varepsilon$ if $\gamma_i = \diamond$. In particular, $\mathsf{weight}(\rho) = \mathsf{weight}(P)$ and $\mathsf{out}(\rho) = \mathsf{label}(P)$.

Since all edges $e \in E_{\mathsf{D},\mathcal{T}}$ satisfy $\mathsf{source}(e) = (p, i-1)$ and $\mathsf{target}(e) = (q, i)$ for some $i \in [n]$, the graph $G_{\mathsf{D},\mathcal{T}}$ is a DAG. We call $G_{\mathsf{D},\mathcal{T}}$ the *DAG-representation* of $\mathcal{T}$ and D.

Observation 3.4. *$|V_{\mathsf{D},\mathcal{T}}| = \mathrm{O}(|\mathsf{D}||Q|)$, and every transition $(p, b, w, \gamma, q) \in \delta$ is responsible for at most $|\mathsf{D}|$ edges of the form $((p, i-1), w, \psi, (q, i)) \in E_{\mathsf{D},\mathcal{T}}$ with $\mathsf{D}[i] = b$, so, $|E_{\mathsf{D},\mathcal{T}}| = \mathrm{O}(|\mathsf{D}||\mathcal{T}|)$. We conclude that $|G_{\mathsf{D},\mathcal{T}}| = \mathrm{O}(|\mathsf{D}||\mathcal{T}|)$, and we can construct $G_{\mathsf{D},\mathcal{T}}$ in time $\mathrm{O}(|\mathsf{D}||\mathcal{T}|)$.*

This means that the problem CT-Enum reduces to the following ranked enumeration problem SP-Enum: For a given DAG $G = (V, E, \mathsf{weight}, \mathsf{label})$ with edge weights from $\mathbb{G}$ and edge labels from $\mathbb{M}$, and designated $s, t \in V$, let $\text{SP-Enum}(G, s, t) = \{\mathsf{label}(P) \mid P \text{ is an } s\text{-to-}t\text{-path of } G\}$. Note that the weight of an element $\mathsf{label}(P)$ of the output set is given by $\mathsf{weight}(P)$.

Consequently, we now want to solve the ranked enumeration problem SP-Enum^f. We will next see (Theorem 5.1) that an algorithm for this problem can be obtained with moderate effort by using a data structure for the shortest paths of a DAG by Eppstein [10], which we will discuss next.

4 Eppstein's DAG of Shortest Paths

Let $G = (V, E, \mathsf{weight}, \mathsf{label})$ be a graph with edge weights from $\mathbb{G}$ and edge labels from $\mathbb{M}$. Let $s, t \in V$ be distinguished nodes of G.

A *heap representation of G's s-to-t-paths* is a tree $\mathcal{H}(G)$. There is a one-to-one correspondence between the s-to-t-paths of G and the nodes of $\mathcal{H}(G)$, and $\mathcal{H}(G)$ is a min-heap with respect to the

weights of the s-to-t-paths, i. e., if a node q of $\mathcal{H}(G)$ corresponds to an s-to-t-path P_q and q has a child r, then r corresponds to an s-to-t-path P_r with $\mathsf{weight}(P_q) \leq_{\mathbb{G}} \mathsf{weight}(P_r)$.

The following result is shown in [10], although here we phrase it in a way suitable for our applications.

Theorem 4.1 (Eppstein [10]). *Let $G = (V, E)$ be a DAG with edge weights from $\mathbb{G}$ and edge labels from $\mathbb{M}$, and let $s, t \in V$. We can compute in time and space $O(|G|)$ a data structure that represents a heap representation $\mathcal{H}(G)$ of G's s-to-t-paths. The degree of $\mathcal{H}(G)$ is 4, we can move from a node q of $\mathcal{H}(G)$ to any of its children in constant time, and, for every node q of $\mathcal{H}(G)$ that represents an s-to-t-path P_q, we can retrieve $\mathsf{weight}(P_q)$ in constant time and $\mathsf{label}(P_q)$ in time $O(|\mathsf{label}(P_q)|)$.*

Note that we can only traverse the edges from the DAG $\mathcal{H}(G)$; in particular, $\mathcal{H}(G)$ does not support a remove-min operation. Nevertheless, we can use $\mathcal{H}(G)$ for a ranked enumeration algorithm for SP-Enum^f. The general idea is the same as how to obtain the k smallest elements from a min-heap (that we can only traverse) in time $O(k \log k)$ as also sketched in the introduction of [13]: We always produce the smallest element of an auxiliary min-heap (that supports a remove-min operation) as output, where the auxiliary min-heap stores all nodes of $\mathcal{H}(G)$ that are children of nodes that have already been produced as output. This allows us to obtain an enumeration algorithm with delay $O(|s_i| + \log i)$. In fact, a simple tweak keeps the size of the auxiliary min-heap upper bounded by $|G|$, making the delay $O(|s_i| + \min\{\log i, \log |G|\})$.

Corollary 4.2. *There is a ranked enumeration algorithm for SP-Enum^f such that, on input (G, s, t), the preprocessing time is $O(|G|)$ and, if $(s_1, s_2, \ldots, s_k)$ is the output of the algorithm, then the delay of s_i is $O(|s_i| + \min\{\log i, \log |G|\})$ for every $i \in [k]$.*

5 Improved Enumeration Algorithms

We now obtain a ranked enumeration algorithm for CT-Enum^f with the reduction described in Section 3 and Corollary 4.2 (the preprocessing time follows by Observation 3.4). As explained in the introduction, this constitutes a substantial improvement over Theorem 3.3.

Theorem 5.1. *There is a ranked enumeration algorithm for CT-Enum^f such that, on input $\mathcal{T}$ and D, the preprocessing time is $O(|\mathcal{T}||\mathsf{D}|)$ and, if $(s_1, s_2, \ldots, s_k)$ is the output of the algorithm, then the delay of s_i is $O(|s_i| + \min\{\log i, \log(|\mathcal{T}||\mathsf{D}|)\})$ for every $i \in [k]$.*

We will now further improve the delay. Let $g_1, g_2, \ldots, g_t \in \mathbb{G}$ and let $n \in \mathbb{N}$. Any expression $\sum_{i=1}^{t} \alpha_i g_i$, where $\alpha_1, \alpha_2, \ldots, \alpha_t \in [0, n]$ and $\sum_{i=1}^{t} \alpha_i \leq n$ is called an *n-sum (over $g_1, g_2, \ldots, g_t$)*. We will assume that every n-sum $\sum_{i=1}^{t} \alpha_i g_i$ is represented by the tuple $(\alpha_1, \alpha_2, \ldots, \alpha_t)$.

Observation 5.2. *In $O(nt)$ time, we can compute all values kg_i for every $k \in [n]$ and $i \in [t]$. This means that within an $O(nt)$ preprocessing, we can compute a data structure that allows us to compute for a given tuple $(\alpha_1, \alpha_2, \ldots, \alpha_t)$ the group-element $\sum_{i=1}^{t} \alpha_i g_i$ in time $O(t)$.*

Our main technical result for improving the delay concerns the problem of sorting given n-sums.

Lemma 5.3. *There is an algorithm that sorts any set X of n-sums over some $g_1, g_2, \ldots, g_t \in \mathbb{G}$ with $|X| = N \geq n$ in time $O(\mathrm{poly}(t)N)$ with high probability. Formally, for any $c \in \mathbb{N}$, the output of the algorithm is always correct, and the bound on the time holds with probability at least $1 - 1/N^c$.*

The proof of this lemma constitutes the main technical part of our paper, and is deferred to Section 6. Next, we show how Lemma 5.3 can be used for improving the delay of our enumeration algorithm for SP-Enum^f (and therefore CT-Enum^f).

It is worth noting that, since in our setting N is upper bounded by a polynomial in n, if $\mathbb{G}$ is the set of integers and each weight can be represented in $O(\log n)$ bits (and, as such, every n-sum can

also be represented in $O(\log n)$ bits), one can use radix sort [20] to order the elements of the set X deterministically in linear time; so computing the value of the elements of X and sorting them can be done deterministically, in that particular case, in $O(tN)$ time. If, instead, we only assume that the weights are integers which fit in a memory word (of size $\omega \geq \log n$), but we do not generally assume that they have $O(\log n)$ bits, then each n-sum will still fit in a constant number of memory words, but it is not known whether we can sort the respective N sums (and, in general, any $N \geq n$ integers, about which we only know that they fit in one memory word of size $\omega \geq \log n$) in $O(N)$ time; see, e.g., [4, 5, 17] and the references therein.

However, our setting (as defined in [7]) is more general than that, and we cannot make any assumption on $\mathbb{G}$ (except that its elements fit in one memory-word). Therefore, in the following, we will rely on Lemma 5.3.

Further, we need the following well-known result.

Lemma 5.4 (Frederickson [13])**.** *Given some $k \in \mathbb{N}$ and a min-heap H that allows us to move from a node to its children in constant time, we can find the element of rank k of H in time $O(k)$.*

Observe that Lemma 5.4 also means that we can get in $O(k)$ time all nodes of H with rank $1, 2, \ldots, k$: Compute first the element of rank k, which has some key ℓ, and then explore H starting from the root and output all elements with a key not larger than ℓ (or, in the case that there are more than one element with key ℓ, stop as soon as k elements have been produced).

For every directed graph $G = (V, E, \text{weight}, \text{label})$ with edge weights and edge labels, let $\#\text{weights}(G) = |\{\text{weight}(e) \mid e \in E\}|$ denote the number of distinct weights that occur in G.

Theorem 5.5. *There is a ranked enumeration algorithm for SP-Enum^f such that, on any input (G, s, t), the preprocessing time is $O(|G| + \text{poly}(\#\text{weights}(G))|V|)$ and every output element s has a delay of $O(|s|\,\text{poly}(\#\text{weights}(G)))$. The guarantees on the preprocessing time and of the delay hold with high probability in the size of the input and the size of the output generated so far.*

Proof. In the preprocessing phase, we construct a data structure that represents a heap representation $\mathcal{H}(G)$ with degree 4 of the s-to-t-paths of G, such that we can move from a node q of $\mathcal{H}(G)$ to any of its children in constant time, and, for every node q of $\mathcal{H}(G)$ that represents an s-to-t-path P_q, we can retrieve $\text{weight}(P_q)$ in constant time and $\text{label}(P_q)$ in time $O(|\,\text{label}(P_q)|)$. According to Theorem 4.1, this can be done in time $O(|G|)$.

We define $n = |V|$ and assume that $g_1, g_2, \ldots, g_t$ are the weights that occur in G. We observe that since each edge of G has a weight from $g_1, g_2, \ldots, g_t$ and every s-to-t-path has at most $n - 1$ edges (since G is a DAG), the weight of any s-to-t-path is an n-sum over $g_1, g_2, \ldots, g_t$. According to Lemma 5.3, we can sort any $N \geq n$ such n-sums over $g_1, g_2, \ldots, g_t$ in time $O(\text{poly}(t)N)$ with high probability in N. Still in the preprocessing, we use Lemma 5.4 to extract the smallest n elements of $\mathcal{H}(G)$ in time $O(n)$, we sort them in $O(\text{poly}(t)n)$ time with high probability in n, using the algorithm of Lemma 5.3, and store them in the sorted order. This concludes the preprocessing, which clearly can be done in time $O(|G| + \text{poly}(t)n)$ with high probability.

The enumeration phase proceeds in several epochs. For every $i = 1, 2, 3, \ldots$, the i^{th} epoch will enumerate the labels of the $2^{i-1}n$ elements of $\mathcal{H}(G)$ with ranks in $(2^{i-2}n, 2^{i-1}n]$ (or in $[1, n]$ for $i = 1$). As an invariant, we assume that when the i^{th} epoch starts, we already have at our disposal in sorted order all elements whose labels are to be enumerated in this epoch. This invariant holds for the first epoch, due to the preprocessing mentioned above.

In epoch i, we do the following: we extract the smallest $2^i n$ elements of $\mathcal{H}(G)$ with Frederickson's algorithm, sort them with Lemma 5.3, and store in the sorted order all elements with ranks from $(2^{i-1}n, 2^i n]$ (i. e., the elements to be enumerated in the next epoch $i + 1$). Due Lemmas 5.4 and 5.3, we can assume that the number of computational steps of this procedure is bounded by $cf(t)2^i n$

with high probability in $2^i n$ for some constant c and polynomial f (that do not depend on i). At the beginning of epoch i, we initialise a counter with 0 that simply counts the computational steps. Whenever the counter reaches value $\frac{cf(t)2^i n}{2^{i-1}n}$, we output in time $O(|L|)$ the label L that corresponds to the next element of the list of pre-computed elements from $\mathcal{H}(G)$ and we set the counter back to 0. Since we need at most $cf(t)2^i n$ computational steps in total, and we have $2^{i-1}n$ pre-computed elements in epoch i, we will never run out of elements to output.

It can be easily verified that in this way epoch i does in fact enumerate the labels of the $2^{i-1}n$ elements of $\mathcal{H}(G)$ with ranks in $(2^{i-2}n, 2^{i-1}n]$ (or in $[1, n]$ for $i = 1$). Moreover, the delay of any output element L is clearly bounded by $|L|\frac{cf(t)2^i n}{2^{i-1}n} \leq c' f(t)|L|$ with high probability in the size of the input and the size of the output generated so far, for a constant c'. □

We can directly use Theorem 5.5 in order to get a ranked enumeration algorithm for CT-Enum^f. Let D be a document and let $\mathcal{T}$ be a cost transducer. We compute the DAG representation $G_{\mathrm{D},\mathcal{T}}$ in time $O(|\mathcal{T}||\mathrm{D}|)$, and we observe that $G_{\mathrm{D},\mathcal{T}}$ is a DAG with $|\mathcal{T}||\mathrm{D}|$ nodes and $\#\mathrm{weights}(G_{\mathrm{D},\mathcal{T}}) = O(|\mathcal{T}|)$. Hence, Theorem 5.5 yields the following theorem, which is our main result.

Theorem 5.6. *There is a ranked enumeration algorithm for CT-Enum^f such that, on any input $(\mathcal{T}, \mathrm{D})$, the preprocessing time is $O(\mathrm{poly}(|\mathcal{T}|)|\mathrm{D}|)$ and every output element s has a delay of $O(|s|\ \mathrm{poly}(|\mathcal{T}|))$. The guarantees on the preprocessing time and of the delay hold with high probability in the size of the input and the size of the output generated so far.*

6 Sorting n-Sums in Linear Time With High Probability

This section is devoted to proving Lemma 5.3: There is an algorithm that sorts any set $X = \{x_1, x_2, \ldots, x_N\}$ of n-sums over some $g_1, g_2, \ldots, g_t \in \mathbb{G}$ with $|X| = N \geq n$ in time $O(\mathrm{poly}(t)N)$ with high probability in N. We structure this proof on several paragraphs for readability purposes.

§ *Algebraic preliminaries.* For basic results and notations regarding linear algebra and, respectively, probability theory, we refer to the standard handbooks [21] and, respectively, [23].

Let us begin with a series of preliminary results. Let $x_i = \sum_{j=1}^t \alpha_j^i g_j$ for every $i \in [N]$. Then, $(\alpha_1^i, \alpha_2^i, \ldots, \alpha_t^i) = (\alpha_1^{i'}, \alpha_2^{i'}, \ldots, \alpha_t^{i'})$ implies $x_i = x_{i'}$, so we can identify identical vectors $(\alpha_1^i, \alpha_2^i, \ldots, \alpha_t^i)$ in $O(t(N+n)) = O(tN)$ time by radix sorting and remove elements corresponding to duplicated vectors from X. After such preliminary reduction, $N \leq n^t$, so $\log N \leq t \log n$. Next, it will be convenient to assume that the actual n-sums (and not just their vectors) are pairwise distinct, which can be assumed without losing generality by the following simple reasoning.

Proposition 6.1. *Let $n \in \mathbb{N}$. Sorting any given n-sums over $g_1, g_2, \ldots, g_t \in \mathbb{G}$ can be reduced in linear time to sorting N* distinct *$(N+n)$-sums over $g_1', g_2', \ldots, g_t'$, where $g_1', g_2', \ldots, g_t'$ are elements of a new ordered group $(\mathbb{G}', +_{\mathbb{G}'}, 0_{\mathbb{G}'}, \leq_{\mathbb{G}'})$ with an effective representation.*

Proof. The elements of $\mathbb{G}'$ are pairs (g, x), where $g \in \mathbb{G}$ and $x \in \mathbb{Z}$, with $+_{\mathbb{G}'}$ being the coordinate-wise addition and $\leq_{\mathbb{G}'}$ the lexicographical comparison. It is clear that $\mathbb{G}'$ admits an effective representation. Next, we define $g_i' = (g_i, 0)$, for $i = 1, 2, \ldots, t$, and $g_{t+1}' = (0, 1)$. Now, let the k-th given n-sum be $\sum_{i=1}^t \alpha_i g_i$. We convert it to $(\sum_{i=1}^t \alpha_i g_i') + k \cdot g_{t+1}'$. This ensures that the obtained $(n+N)$-sums are pairwise distinct, as each of them has distinct second coordinate. Arranging them in the strictly increasing order gives us the original n-sums in the non-decreasing order. □

To avoid clutter, from now on we will assume that the goal is to sort N given $(N+n)$-sums over $g_1, g_2, \ldots, g_t \in \mathbb{G}$ (instead of talking about $g_1', g_2', \ldots, g_t' \in \mathbb{G}'$).

Further, we need some facts about ordered abelian groups. As we can only access $\mathbb{G}$ by calculating expressions of the form $\sum_{i=1}^t \alpha_i g_i$, we can assume that $\mathbb{G}$ is finitely generated by $\{g_1, g_2, \ldots, g_t\}$;

to avoid clutter, we will not write anymore that the sums are for i from 1 to t, and consider this implicit. In this case, known results imply the following.

Lemma 6.2. *If $(\mathbb{G}, +_{\mathbb{G}}, 0_{\mathbb{G}}, \leq_{\mathbb{G}})$ is finitely generated by $\{g_1, g_2, \ldots, g_t\}$ then there exist $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_t \in \mathbb{R}^t$ such that for any $\alpha_1, \alpha_2, \ldots, \alpha_t \in \mathbb{Z}$ we have $\sum_i \alpha_i g_i \leq_{\mathbb{G}} 0$ if and only if $\sum_i \alpha_i \vec{v}_i \leq_{lex} \vec{0}$.*

Further, we will later verify that our algorithm only accesses $\mathbb{G}$ by performing comparisons of the form $\sum_i \alpha_i g_i \leq_{\mathbb{G}} 0_{\mathbb{G}}$, where $\alpha_1, \alpha_2, \ldots, \alpha_t \in [-2(N+n), 2(N+n)]$. In such a case, we can actually think that the elements of $\mathbb{G}$ are real numbers (in fact, to this end, it is enough that every α_i comes from a finite set). We stress that these real numbers are never computed by the algorithm, and this is just a formal proof that, without losing generality, we can think that $\mathbb{G}$ is a subgroup of $(\mathbb{R}, +, 0, \leq)$, which is a crucial insight towards understanding both the correctness and complexity of this algorithm. The following lemma follows from Lemma 6.2.

Lemma 6.3. *There exist $\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_t \in \mathbb{R}$ such that, for every $\alpha_1, \alpha_2, \ldots, \alpha_t \in [-2(N+n), 2(N+n)]$, we have $\sum_i \alpha_i g_i \leq_{\mathbb{G}} 0_{\mathbb{G}}$ if and only if $\sum_i \alpha_i \hat{g}_i \leq 0$.*

§ *The point-location problem and its relation to our problem.* Before presenting the actual algorithm showing Lemma 5.3, we also need to recap the approach of Kane, Lovett, and Moran [18], and see its connections to our setting. For a finite set $H = \{h_1, h_2, \ldots, h_{|H|}\} \subseteq \mathbb{R}^t$, let the function $\mathcal{A}_H : \mathbb{R}^t \to \{-, 0, +\}^{|H|}$ be defined as $\mathcal{A}_H(x) := (\text{sign}(\langle x, h_1\rangle), \text{sign}(\langle x, h_2\rangle), \ldots, \text{sign}(\langle x, h_{|H|}\rangle))$ (where $\langle \cdot, \cdot \rangle$ denotes the inner product on the vector space $\mathbb{R}^t$, and sign is the sign-function on $\mathbb{R}$).

Kane, Lovett, and Moran [18] considered the problem of determining $\mathcal{A}_H(x)$ for any given $x \in \mathbb{R}^t$ with few queries of the form $\text{sign}(\langle x, h\rangle)$, for $h \in H$ (label queries) and $\text{sign}(\langle x, h' - h''\rangle)$, for $h', h'' \in H$ (comparison queries); this is *the point-location in an hyperplane-arrangement problem.*

Before proceeding with further definitions, let us comment on how this could be possibly useful in our problem. Recall that our current goal (according to Proposition 6.1 and the comment made after it) is to sort a set X of $N \geq n$ given $(N+n)$-sums, where the i-th given $(N+n)$-sum is $x_i = \sum_{j=1}^{t} \alpha_j^i g_j$. First, let us consider how many expressions of the form $\sum_j \alpha_j g_j \leq_{\mathbb{G}} 0_{\mathbb{G}}$ need to be evaluated in order to uniquely determine the sorting permutation π of X, and disregard the computational complexity of actually finding π given the results of all the comparisons. It is clear that it is necessary and sufficient to resolve $\sum_j \alpha_j^i g_j \leq_{\mathbb{G}} \sum_j \alpha_j^{i'} g_j$, for every i, i', or in other words $\sum_j (\alpha_j^i - \alpha_j^{i'}) g_j \leq_{\mathbb{G}} 0_{\mathbb{G}}$. From the point of view of evaluating such expression, Lemma 6.3 tells us that we can think that $g_1, g_2, \ldots, g_t \in \mathbb{R}$. Thus, we can reformulate the problem by defining $H = \{(\alpha_1^i - \alpha_1^{i'}, \alpha_2^i - \alpha_2^{i'}, \ldots, \alpha_t^i - \alpha_t^{i'}) : 1 \leq i, i' \leq N\}$ and saying that we want to determine $\mathcal{A}_H(g)$, where $g = (g_1, g_2, \ldots, g_t)$. This will be done with few label and comparison queries. We note that a label query evaluates an expression $\sum_j \beta_j g_j \leq_{\mathbb{G}} 0_{\mathbb{G}}$, where $\beta_1, \beta_2, \ldots, \beta_t \in [-(N+n), N+n]$, while a comparison query evaluates an expression $\sum_j \beta_j g_j \leq_{\mathbb{G}} 0_{\mathbb{G}}$, where $\beta_1, \beta_2, \ldots, \beta_t \in [-2(N+n), 2(N+n)]$, thus we can still think that $g_1, g_2, \ldots, g_t \in \mathbb{R}$ (and this is, indeed, the reason for having $2(N+n)$ in the statement of Lemma 6.3). The main result of Kane, Lovett, and Moran [18] shows that it is possible to determine $\mathcal{A}_H(g)$ by evaluating only $O(t \log(t(N+n)) \log N)$ such expressions corresponding to label and comparison queries. However, this is an existential proof, and does not tell us how to actually find which expressions to evaluate, and even if we knew that, it would still be quite problematic, given that we have N^2 possible expressions, and we cannot explicitly operate on all of them. Thus, we will need a more refined approach that can be implemented efficiently. This requires looking inside the approach of Kane, Lovett, and Moran [18], and some more definitions.

For $S \subseteq H$, and $h, x \in \mathbb{R}^t$, we say that S infers h at x if $\text{sign}(\langle h, x\rangle)$ is fully determined by the answers to the label and comparison queries on S. We define $\text{infer}(S, x)$ as the set of all $h \in \mathbb{R}^t$ such that S infers h at x. The inference dimension of $H \subset \mathbb{R}^t$ is defined as the smallest $d \geq 1$ such that, for any $S \subset H$ of size at least d, and for any $x \in \mathbb{R}^t$, there exists $h \in S$ such that $S \setminus \{h\}$ infers h at x.

While it might be difficult to precisely determine the inference dimension of a specific H, it can be upper bounded as follows. For $h \in \mathbb{R}^t$, where $h = (h_1, h_2, \ldots, h_t)$, we define $||h||_1 = \sum_{i=1}^{t} |h_i|$. Then, if every $h \in H$ has integer coefficients such that $||h||_1 \leq w$ then the inference dimension of H is $d = \mathrm{O}(t \log w)$. This follows from the following lemma that upper bounds the inference dimension of the set consisting of all $h \in \mathbb{Z}^t$ such that $||h||_1 \leq w$, and the fact that the inference dimension of a smaller set cannot be larger.

Lemma 6.4 ([18, Theorem 1.10]). *The inference dimension of* $\{h \in \mathbb{Z}^t : ||h||_1 \leq w\}$ *is* $d = \mathrm{O}(t \log w)$.

If we are able to bound the inference dimension of H (for example, by the above lemma), then we have the following lemma.

Lemma 6.5 (Corollary of [18, Lemma 4.2]). *Choose any* $c \in \mathbb{N}$. *Let* $H \subseteq \mathbb{R}^t$ *be a finite set with inference dimension at most* d, *and* $S \subseteq H$ *be its uniformly chosen subset of size* $s \cdot c \log n$. *Then, for every* $x \in \mathbb{R}^t$:

$$\Pr_S \left[|H \setminus \mathsf{infer}(S, x)| \geq \frac{2d}{s+1} |H| \right] \leq \frac{1}{n^c}.$$

§ *High-level overview of the algorithm.* After presenting the main technical ingredients of our algorithm, we now provide a high-level description of the algorithm.

Recall that we want to sort a set $X = \{x_1, x_2, \ldots, x_N\}$ of $N \geq n$ given $(N+n)$-sums in $\mathrm{O}(\mathrm{poly}(t)N)$ time, and recall that, for every $i \in [N]$, $x_i = \sum_j \alpha_j^i g_j$ for some $\alpha_1^i, \alpha_2^i, \ldots, \alpha_t^i \in [0, N+n]$ such that $\sum_{j=1}^{t} \alpha_j^i \leq N + n$. We apply the preprocessing from Observation 5.2, so that after $\mathrm{O}(tN)$ preprocessing we can evaluate any such expression in $\mathrm{O}(t)$ time. We will proceed in two phases. First, we will partition X into groups of size $\log^d N$ in $\mathrm{O}(\mathrm{poly}(t)N)$ time, for some $d \in \mathbb{N}$ to be determined later. The elements in each group will be in any order, but all elements in the i-th group will be smaller than the elements in the $(i+1)$-th group. Second, we will sort the elements in every group in $\mathrm{O}(\mathrm{poly}(t)N)$ total time. In both steps, the main idea is to "approximate" real numbers $\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_t$ from Lemma 6.3 with rational numbers $\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_t$. This allows us to compute a rational approximation $\tilde{x}_i = \sum_j \alpha_i^j \tilde{g}_j$ of every $x_i = \sum_j \alpha_j^i g_j$. Then, assuming that the rational numbers consist of only $\mathrm{O}(\mathrm{poly}(t) \log n)$ bits (which will be the case), we can sort $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_N$ in $\mathrm{O}(\mathrm{poly}(t)N)$ time in the Word RAM model. To find the approximations, we apply Lemma 6.5, and then use some tools from linear programming.

§ *Detailed description of the algorithm.* We now move to describing both phases of the algorithm in details. For each phase, we describe first what is done by our algorithm, and then the reasoning (and technical details) supporting the correctness and complexity of each step.

The first phase. We begin with choosing a subset $Y \subseteq X$ of size $m+1$ by always including the smallest and the largest element of X, and independently including every other element of X with probability $p = 1/\log^c N$, where $c \in \mathbb{N}$ will be fixed later so that X is partitioned into groups of size $\Theta(\log^d n)$ with $d = 2c + 3$. By a standard application of the multiplicative Chernoff bound, $m = \Theta(N/\log^c N)$ holds with high probability in N, and from now we condition on this being the case. We sort Y with $\mathrm{O}(m \log m)$ comparisons, which takes $\mathrm{O}(tm \log m) = \mathrm{O}(tN)$ time, as we can compare any two elements in $\mathrm{O}(t)$ time. Let $x_{y_0} <_{\mathbb{G}} x_{y_1} <_{\mathbb{G}} \ldots <_{\mathbb{G}} x_{y_m}$ be the elements of Y in the sorted order, and to avoid clutter assume that m is a multiple of $\log^{c+3} N$.

Once Y is defined and sorted, for $j = 0, 1, \ldots, m/\log^{c+3} N - 1$, we define the sets:

$$Z_j := \{i \in [1, N] : x_{y_{j \cdot \log^{c+3} N}} \leq_{\mathbb{G}} x_i \text{ and } x_i <_{\mathbb{G}} x_{y_{(j+1) \log^{c+3} N}}\}.$$

As we have included the smallest and the largest element of X in Y, Z_js form a partition of X. Next, we analyse the size of every Z_j. The size of a single Z_j can be bounded as follows. Recall

that every element of X is included in Y with probability p. Then, $|Z_j|$ is the number of trials until obtaining $\log^3 N$ successes in independent Bernoulli trials with probability p. A straightforward application of the multiplicative Chernoff bound implies that among $p \cdot t/p$ trials we will have at least $t/2$ successes with probability at least $1 - \exp(-t/8)$. Thus, by setting $t = \Theta(\log^3 N)$, we obtain that $|Z_j| = \mathrm{O}(\log^{3+c} N)$ holds with high probability in N. From now on, we condition on $|Z_j| = \mathrm{O}(\log^{2c+3} N)$ holding for all j, which happens with high probability in N.

We now focus on how to construct these sets Z_j. In order to effectively allocate the elements of X into the sets Z_j defined above, we will assign a rational number $\tilde{x}_i = p_i/q_i$ to every x_i, with the nominator and denominator consisting of $\mathrm{O}(\mathrm{poly}(t)\log n)$ bits, by first assigning such a rational number $\tilde{g}_i$ to every g_i, and then calculating $\tilde{x}_i = \sum_j \alpha^i_j \tilde{g}_j$. In our model, arithmetical operations on numbers consisting of $\mathrm{O}(\mathrm{poly}(t)\log n)$ bits can be performed in $\mathrm{O}(\mathrm{poly}(t))$ time, which allows us to compute the $\tilde{x}_i$s in $\mathrm{O}(\mathrm{poly}(t)N)$ time. Then, we sort the $\tilde{x}_i$s in $\mathrm{O}(\mathrm{poly}(t)N)$ time using radix sort using the following lemma. We stress that sorting the $\tilde{x}_i$ will only approximate sorting their corresponding x_is, and there will possibly be elements x_i and x_j such that $x_i <_{\mathbb{G}} x_j$ but $\tilde{x}_i \geq \tilde{x}_j$. Still, we will be able to bound for how many pairs of elements this can possibly happen.

Lemma 6.6. *Sorting N rational numbers p_i/q_i, with each nominator and denominator consisting of $\mathrm{O}(\mathrm{poly}(t)\log n)$ bits, can be done in $\mathrm{O}(\mathrm{poly}(t)N)$ time.*

Once the rational numbers $\tilde{x}_i$ are sorted, we will ultimately allocate the elements x_i of X to the sets Z_j, by partitioning the sorted list of rational numbers $\tilde{x}_i$ w.r.t. the rational numbers corresponding to the elements of Y (which was already sorted). We will explain how this is exactly done in detail also below, once we clarify how the rational numbers $\tilde{x}_i$ are obtained. The rest of the description of the first phase of our algorithm describes how to implement these steps.

So, let us now explain how to assign a rational number $\tilde{x}_i = p_i/q_i$ to every x_i. We need to keep in mind our goal: namely, to ensure that the rational numbers can be used instead of the original x_is to partition X into Z_js. First, for every i we define a point $h_i = (\alpha^i_1, \alpha^i_2, \ldots, \alpha^i_t) \in [0, N+n]^t$. As mentioned earlier, by Lemma 6.3 we can think that $g_1, g_2, \ldots, g_t \in \mathbb{R}$. Then, we observe that $x_i <_{\mathbb{G}} x_{i'}$ is equivalent to $\mathrm{sign}(\langle g, h_i - h_{i'}\rangle) = -1$. Thus, sorting the x_is could be done by determining $\mathcal{A}_H(g)$, where $H = \{h_i - h_{i'} : 1 \leq i, i' \leq N\}$. We note that clearly $|H| = N^2$, and by Lemma 6.4, the inference dimension of H is $\mathrm{O}(t\log(N+n)) = \mathrm{O}(t\log N)$. We choose a uniform subset $H' \subseteq H$ of size $s = tN/\log^c N$. By Lemma 6.5, $|H \setminus \mathsf{infer}(H', g)| = \frac{\mathrm{O}(t\log^2 N)}{s}|H| = \mathrm{O}(N\log^{c+2} N)$ with high probability in N, and from now we condition on this being the case. Next, we want to resolve all label and comparison queries on H'. This is because, as H' has been chosen uniformly at random, by Lemma 6.5 this will actually allow us to resolve many comparison queries on the whole H', which brings us closer to sorting the whole X. Label queries can be resolved directly by computing, for every $h' \in H'$, $\mathrm{sign}(\langle g, h'\rangle)$. To efficiently resolve all comparison queries, we sort the elements of H' by declaring $h' \leq h''$ if and only if $\langle g, h'\rangle \leq_{\mathbb{G}} \langle g, h''\rangle$, where $g = (g_1, g_2, \ldots, g_t)$. Sorting can be done with $\mathrm{O}(s\log s) = \mathrm{O}(tN)$ comparisons, so $\mathrm{O}(t^2 N)$ time. We denote the elements of H' arranged according to this order by $h'_1, h'_2, \ldots, h'_s$, so that $\langle g, h'_1\rangle \leq_{\mathbb{G}} \langle g, h'_2\rangle \leq_{\mathbb{G}} \cdots \leq_{\mathbb{G}} \langle g, h'_s\rangle$. For every $i = 1, 2, \ldots, s-1$, we compute $\mathrm{sign}(\langle g, h'_i - h'_{i+1}\rangle)$ (observe that this is either -1 or 0). This takes $\mathrm{O}(ts) = \mathrm{O}(tN)$ time, and in fact implicitly resolves all comparison queries on H'. Indeed, $\mathrm{sign}(\langle g, h'_i - h'_j\rangle)$ for $i < j$ is either -1 or 0 due to sorting, and it is 0 if and only if $\mathrm{sign}(\langle g, h'_k - h'_{k+1}\rangle) = 0$ for every $k = i, i+1, \ldots, j-1$.

Next, we create a system of $\mathrm{O}(tN/\log^c N)$ linear inequalities in t variables $\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_t$. Let $\tilde{g} = (\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_t)$. The goal of the inequalities is to encode the current information about g. In other words, for any $\tilde{g}$ that respects all the inequalities, the behaviour of the algorithm should be the same when run with g replaced by $\tilde{g}$. First, for every $i = 1, 2, \ldots, m-1$, we add an inequality

of the form $\langle \tilde{g}, h_{x_{y_i}} - h_{x_{y_{i+1}}} \rangle \leq -1$. Second, for $i = 1, 2, \ldots, s$, if $\text{sign}(\langle g, h'_i \rangle) = -1$ we add an inequality $\langle \tilde{g}, h'_i \rangle \leq -1$, if $\text{sign}(\langle g, h'_i \rangle) = 0$ we add inequalities $\langle \tilde{g}, h'_i \rangle \leq 0$ and $\langle \tilde{g}, h'_i \rangle \geq 0$, and if $\text{sign}(\langle g, h'_i \rangle) = 1$ we add an inequality $\langle \tilde{g}, h'_i \rangle \geq 1$. Third, for every $i = 1, 2, \ldots, s-1$ if $\text{sign}(\langle g, h'_i - h'_{i+1} \rangle) = -1$ we add an inequality $\langle \tilde{g}, h'_i - h'_{i+1} \rangle \leq -1$, and if $\text{sign}(\langle g, h'_i - h'_{i+1} \rangle) = 0$ we add inequalities $\langle \tilde{g}, h'_i - h'_{i+1} \rangle \leq 0$ and $\langle \tilde{g}, h'_i - h'_{i+1} \rangle \geq 0$. We now want to solve this system of inequalities.

We need a few (standard) definitions from combinatorial optimisation. We recommend the reader to consult the excellent book by Grötschel, Lovász, and Schrijver [16] for the details. A polyhedron K is a set of solutions of a system of inequalities, or $K = \{\vec{x} \in \mathbb{R}^t : A\vec{x} \leq b\}$. The facet-complexity of K is at most ϕ if the encoding length of each inequality is at most ϕ, meaning that each inequality has rational coefficients and the total length of the binary encodings of all rational numbers appearing in the same inequality is at most ϕ. The strong non-emptiness problem for such a polyhedra K amounts to finding some $x \in K$ (where the coordinates of x are described as rational numbers, so in fact $x \in \mathbb{Q}^t$), or detecting that K is empty. A strong separation oracle takes $y \in \mathbb{Q}^t$, checks if $y \in K$, and if not returns a hyperplane that separates it from K, i.e. $c \in \mathbb{R}^t$ such that $\langle c, y \rangle > \max\{\langle c, x \rangle : x \in K\}$. The following theorem is known, and determines our $c \in \mathbb{N}$.

Theorem 6.7 ([19]). *For some $c \in \mathbb{N}$, the strong non-emptiness problem for a t-dimensional polyhedra with facet-complexity ϕ given by a strong separation oracle can be solved in $\mathrm{O}(\mathrm{poly}(t)\phi^c)$ time and calls to the oracle. The length of the binary encoding of the found $x \in K$ (if any) is $\mathrm{O}(\mathrm{poly}(t)\phi)$.* [3]

Our system of linear inequalities clearly forms a t-dimensional polyhedron K. Further, its facet-complexity is $\mathrm{O}(t \log(N + n)) = \mathrm{O}(t \log N)$. We need to establish that K is non-empty. Recall that, by Lemma 6.3, we can think that $g \in \mathbb{R}^t$. It is not necessarily the case that $g \in K$, as we have encoded every constraint of the form $\text{sign}(\langle g, h \rangle) = -1$ as $\langle \tilde{g}, h \rangle \leq -1$. However, for any $\alpha > 0$, $\text{sign}(\langle g, h \rangle) = \text{sign}(\langle \alpha g, h \rangle)$. Thus, we can choose sufficiently large $\alpha > 0$ so that in fact for each of the finitely many relevant constraints of the form $\text{sign}(\langle g, h \rangle) = -1$ we actually have $\langle g, h \rangle \leq -1$, and so $\alpha g \in K$, hence K non-empty. Thus, Theorem 6.7 allows us to find $\tilde{g} \in K$ with every coordinate being a rational number with nominator and denominator consisting of $\mathrm{O}(\mathrm{poly}(t) \log N) = \mathrm{O}(\mathrm{poly}(t) \log n)$ bits, assuming that we can implement an efficient strong separation oracle. A strong separation oracle can be implemented by simply iterating over all the inequalities for a given y, checking if $\langle y, h \rangle \leq 0$ and returning h as the hyperplane that separates y from K otherwise. This takes $\mathrm{O}(t^2 N / \log^c N)$ time per call, making the overall time $\mathrm{O}(\mathrm{poly}(t)(t \log n)^c t^2 N / \log^c N) = \mathrm{O}(\mathrm{poly}(t) N)$.

Having found $\tilde{g} = (\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_t)$, we calculate $\tilde{x}_i = \langle \tilde{g}, h_i \rangle$, for $i = 1, 2, \ldots, N$. Every $\tilde{x}_i$ is a sum of t rational numbers with nominators and denominators consisting of $\mathrm{O}(\mathrm{poly}(t) \log n)$, thus can be represented as a rational number with nominator and denominator consisting also of $\mathrm{O}(\mathrm{poly}(t) \log n)$ bits. This takes $\mathrm{O}(\mathrm{poly}(t) N)$ time in our model. Next, we sort $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_N$ with radix sort in $\mathrm{O}(\mathrm{poly}(t) N)$ time as described earlier to obtain a sorted sequence $\tilde{x}_{i_1} \leq \tilde{x}_{i_2} \leq \ldots \leq \tilde{x}_{i_N}$. Because of the inequalities of the form $\langle \tilde{g}, h_{x_{y_{i+1}}} - h_{x_{y_i}} \rangle \leq -1$, for $i = 1, 2, \ldots, m-1$, we have $\tilde{x}_{y_1} < \tilde{x}_{y_2} < \ldots < \tilde{x}_{y_m}$. That is, the order on the elements of Y is the same when sorted using the original x_is and their approximations $\tilde{x}_i$. We partition all elements of X into sets $\tilde{Z}_j$ as follows:

$$\tilde{Z}_j := \{i \in [1, N] : \tilde{x}_{y_{j \cdot \log^{c+3} N}} \leq \tilde{x}_i \text{ and } \tilde{x}_i < \tilde{x}_{y_{(j+1) \log^{c+3} N}}\}.$$

This takes only $\mathrm{O}(N)$ time as the elements are already sorted by the $\tilde{x}_i$s. We claim that $\tilde{Z}_j$ is a reasonably good approximation of Z_j in the following sense. Consider $i \in \tilde{Z}_j$. We can check in

[3] The formulation of the first part of the theorem is from [16, Theorem 6.4.1]. The bound on the length of the binary encoding of x follows by either carefully analysing the algorithm or a black-box rounding procedure from [16, Theorem 6.2.13].

constant time if $i \in Z_{j-1}, Z_j, Z_{j+1}$, and, if so, insert i into the appropriate $Z_{j'}$, for $j' \in \{j-1, j, j+1\}$. Assume otherwise, by symmetry it is enough to consider the case where $x_i <_{\mathbb{G}} x_{y_{(j-1)\log^{c+3} N}}$. In such a case, for every $i' \in [(j-1)\log^{c+3} N, j \cdot \log^{c+3} N)$ we have $\langle \tilde{g}, h_{y_{i'}} - h_i \rangle < 0$ but $\langle g, h_{y_{i'}} - h_i \rangle >_{\mathbb{G}} 0_{\mathbb{G}}$. Hence, such an i contributes $\Omega(\log^{c+3} N)$ elements to the set $\{h \in H : h \notin \mathsf{infer}(H', g)\}$. However, we have conditioned on the size of this set being $O(N \log^{c+2} N)$, so such a situation can happen only for $O(N/\log N)$ elements x_i. For each such x_i, we binary search over the elements $x_{j \cdot \log^{c+3} N}$, for $j = 1, 2, \ldots, m/\log^{c+3} N$, to assign i to the appropriate Z_j in $O(tN)$ total time.

To summarise the first phase of the algorithm, in $O(\mathrm{poly}(t)N)$ time we have partitioned X into Z_js such that $|Z_j| = O(\log^{2c+3} N)$ holds for every j and the elements assigned to Z_j are all strictly smaller than the elements assigned to Z_{j+1}. Hence, it remains to sort every Z_j.

The second phase. The second phase proceeds similarly as the first phase.

§ *Summary.* After the second phase, we have correctly sorted the set X. Each phase works in $O(tN)$ time, conditioned on some events ($m = \Theta(N/\log^c N)$, $|Z_j| = O(\log^{2c+3} N)$ for every j, $|H \setminus \mathsf{infer}(H', g)| = O(N \log^{c+2} N)$, $|H_1 \setminus \mathsf{infer}(H_1', g)| = O(N \log^{3c+5} N)$) that happen with high probability in N. To obtain the statement of Lemma 5.3, we observe that we can verify the correctness of the output (by checking if $x_1 \leq_{\mathbb{G}} x_2 \leq_{\mathbb{G}} \ldots \leq_{\mathbb{G}} x_N$) in $O(tN)$ time, and terminate the procedure if its running time exceeds $O(tN)$ time. Then, with high probability in N we obtain the correct output, and otherwise we restart the procedure.

Acknowledgments

Paweł Gawrychowski is partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505.
Florin Manea is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) in the framework of the Heisenberg Programme – project number 466789228 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 466789228).
Markus L. Schmid is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – project number 522576760 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 522576760).

References

[1] Antoine Amarilli, Pierre Bourhis, Florent Capelli, and Mikaël Monet. 2024. Ranked Enumeration for MSO on Trees via Knowledge Compilation. In *27th International Conference on Database Theory, ICDT 2024, Proceedings*. 25:1–25:18. https://doi.org/10.4230/LIPICS.ICDT.2024.25
[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2020. Constant-Delay Enumeration for Nondeterministic Document Spanners. *SIGMOD Rec.* 49, 1 (2020), 25–32.
[3] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-Delay Enumeration for Nondeterministic Document Spanners. *ACM Trans. Database Syst.* 46, 1 (2021), 2:1–2:30. https://doi.org/10.1145/3436487
[4] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. 1998. Sorting in Linear Time? *J. Comput. Syst. Sci.* 57, 1 (1998), 74–93. https://doi.org/10.1006/JCSS.1998.1580
[5] Arne Andersson and Mikkel Thorup. 2007. Dynamic ordered sets with exponential search trees. *J. ACM* 54, 3 (2007), 13. https://doi.org/10.1145/1236457.1236460
[6] Guillaume Bagan. 2006. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *Computer Science Logic CSL 2006, 15th Annual Conference of the EACSL, Proceedings*. 167–181.
[7] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. 2021. Ranked Enumeration of MSO Logic on Words. In *24th International Conference on Database Theory, ICDT 2021*. 20:1–20:19. https://doi.org/10.4230/LIPIcs.ICDT.2021.20
[8] Johannes Doleschal, Benny Kimelfeld, and Wim Martens. 2023. The Complexity of Aggregates over Extractions by Regular Expressions. *Log. Methods Comput. Sci.* 19, 3 (2023). https://doi.org/10.46298/LMCS-19(3:12)2023

[9] Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. 2020. Weight Annotation in Information Extraction. In *23rd International Conference on Database Theory, ICDT 2020, Proceedings*. 8:1–8:18. https://doi.org/10.4230/LIPIcs.ICDT.2020.8

[10] David Eppstein. 1998. Finding the k Shortest Paths. *SIAM J. Comput.* 28, 2 (1998), 652–673. https://doi.org/10.1137/S0097539795290477

[11] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51.

[12] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* 45, 1 (2020), 3:1–3:42. https://doi.org/10.1145/3351451

[13] Greg N. Frederickson. 1993. An Optimal Algorithm for Selection in a Min-Heap. *Inf. Comput.* 104, 2 (1993), 197–214.

[14] Michael L. Fredman and Dan E. Willard. 1990. BLASTING through the Information Theoretic Barrier with FUSION TREES. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, Harriet Ortiz (Ed.). ACM, 1–7. https://doi.org/10.1145/100216.100217

[15] Dominik D. Freydenberger and Sam M. Thompson. 2022. Splitting Spanner Atoms: A Tool for Acyclic Core Spanners. In *25th International Conference on Database Theory, ICDT 2022, Proceedings*. 10:1–10:18. https://doi.org/10.4230/LIPIcs.ICDT.2022.10

[16] Martin Grötschel, László Lovász, and Alexander Schrijver. 1988. *Geometric Algorithms and Combinatorial Optimization*. Algorithms and Combinatorics, Vol. 2. Springer.

[17] Yijie Han and Mikkel Thorup. 2002. Integer Sorting in 0(n sqrt (log log n)) Expected Time and Linear Space. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), Proceedings*. IEEE Computer Society, 135–144. https://doi.org/10.1109/SFCS.2002.1181890

[18] Daniel M. Kane, Shachar Lovett, and Shay Moran. 2019. Near-optimal Linear Decision Trees for k-SUM and Related Problems. *J. ACM* 66, 3 (2019), 16:1–16:18.

[19] L.G. Khachiyan. 1980. Polynomial algorithms in linear programming. *U. S. S. R. Comput. Math. and Math. Phys.* 20, 1 (1980), 53–72. https://doi.org/10.1016/0041-5553(80)90061-0

[20] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[21] Serge Lang. 1987. *Linear Algebra, 3rd Edition*. Springer. https://link.springer.com/book/10.1007/978-1-4757-1949-9

[22] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 125–136. https://doi.org/10.1145/3196959.3196968

[23] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press. https://doi.org/10.1017/CBO9780511813603

[24] Martin Muñoz and Cristian Riveros. 2023. Constant-Delay Enumeration for SLP-Compressed Documents. In *26th International Conference on Database Theory, ICDT 2023, Proceedings*. 7:1–7:17. https://doi.org/10.4230/LIPICS.ICDT.2023.7

[25] Markus L. Schmid. 2024. The Information Extraction Framework of Document Spanners - A Very Informal Survey. In *SOFSEM 2024: Theory and Practice of Computer Science - 49th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2024, Proceedings*. 3–22. https://doi.org/10.1007/978-3-031-52113-3_1

[26] Markus L. Schmid and Nicole Schweikardt. 2022. Document Spanners - A Brief Overview of Concepts, Results, and Recent Developments. In *PODS '22: International Conference on Management of Data, Proceedings*. 139–150. https://doi.org/10.1145/3517804.3526069

[27] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2022. Any-k Algorithms for Enumerating Ranked Answers to Conjunctive Queries. *CoRR* abs/2205.05649 (2022). https://doi.org/10.48550/arXiv.2205.05649

Received May 2024; revised August 2024; accepted September 2024