# Constant-Delay Enumeration for SLP-Compressed Documents

**Martín Muñoz**
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

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

## Abstract

We study the problem of enumerating results from a query over a compressed document. The model we use for compression are straight-line programs (SLPs), which are defined by a context-free grammar that produces a single string. For our queries we use a model called Annotated Automata, an extension of regular automata that allows annotations on letters. This model extends the notion of Regular Spanners as it allows arbitrarily long outputs. Our main result is an algorithm which evaluates such a query by enumerating all results with output-linear delay after a preprocessing phase which takes linear time on the size of the SLP, and cubic time over the size of the automaton. This is an improvement over Schmid and Schweikardt's result [25], which, with the same preprocessing time, enumerates with a delay which is logarithmic on the size of the uncompressed document. We achieve this through a persistent data structure named Enumerable Compact Sets with Shifts which guarantees output-linear delay under certain restrictions. These results imply constant-delay enumeration algorithms in the context of regular spanners. Further, we use an extension of annotated automata which utilizes succinctly encoded annotations to save an exponential factor from previous results that dealt with constant-delay enumeration over vset automata. Lastly, we extend our results in the same fashion Schmid and Schweikardt did [26] to allow complex document editing while maintaining the constant-delay guarantee.

**2012 ACM Subject Classification** Theory of computation → Database theory

**Keywords and phrases** SLP compression, query evaluation, enumeration algorithms

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2023.7

**Funding** This work was funded by ANID – Millennium Science Initiative Program – Code `ICN17_002`.

## 1 Introduction

A *constant-delay enumeration algorithm* is an efficient solution to an enumeration problem: given an instance of the problem, the algorithm performs a preprocessing phase to build some indices, to then continue with an enumeration phase where it retrieves each output, one by one, taking constant-delay between consecutive outcomes. These algorithms provide a strong guarantee of efficiency since a user knows that, after the preprocessing phase, she will access the output as if we have already computed them. For these reasons, constant-delay algorithms have attracted researchers' attention, finding sophisticated solutions to several query evaluation problems. Starting with Durand and Grandjean's work [14], researchers have found constant-delay algorithms for various classes of conjunctive queries [6, 10], FO queries over sparse structures [19, 27], and MSO queries over words and trees [5, 2].

The enumeration problem over documents (i.e., strings) has been studied extensively under the framework of document spanners [15]. A constant-delay algorithm for evaluating deterministic regular spanners was first presented in [16] and extended to non-deterministic in [3]. After these results, people have studied the enumeration problem of document spanners in the context of ranked enumeration [12, 8], nested documents [22], or grammars [23, 4].


Recently, Schmid and Schweikardt [25, 26] studied the evaluation problem for regular spanners over a document compressed by a Straight-line Program (SLP). In this setting, one encodes a document through a context-free grammar that produces a single string (i.e., the document itself). This mechanism allows highly compressible documents, in some instances allowing logarithmic space compared to the uncompressed copy. The enumeration problem consists now of evaluating a regular spanner over an SLP-compressed document. In [25], the authors provided a logarithmic-delay (over the uncompressed document) algorithm for the problem, and in [26], they extended this setting to edit operations over SLP documents, maintaining the delay. In particular, these works left open whether one can solve the enumeration problem of regular spanners over SLP-compressed documents with a constant-delay guarantee.

This paper aims to extend the understanding of the evaluation problem over SLP-compressed documents in several directions. First, we study the evaluation problem of *annotated automata* (AnnA) over SLP-compressed documents. These automata are a general model for defining regular enumeration problems, which strictly generalizes the model of extended variable-set automaton used in [25]. Second, we provide an output-linear delay enumeration algorithm for the problem of evaluating an unambiguous AnnA over an SLP-compressed document. In particular, this result implies a constant-delay enumeration algorithm for evaluating extended variable-set automaton, giving a positive answer to the open problem left in [25]. Third, we can show that we can extend this result to what we call a *succinctly* annotated automaton, a generalization of AnnA whose annotations are succinctly encoded by an enumeration scheme. We can develop an output-linear delay enumeration algorithm for this model, showing a constant-delay algorithm for sequential (non-extended) vset automata, strictly generalizing the work in [25]. Finally, we show that one can maintain these algorithmic results when dealing with complex document editing as in [26].

The main technical result is to show that the data structure presented in [22], called Enumerable Compact Set (ECS), can be extended to deal with shift operators (called Shift-ECS). This extension allows us to compactly represent the outputs and "shift" the results in constant time, which is to add or substract a common value to all elements in a set. Then, by using matrices with Shift-ECS nodes, we can follow a bottom-up evaluation of the annotated automaton over the grammar (similar to [25]) to enumerate all outputs with output-linear delay. The combination of annotated automata and Shift-ECSs considerably simplifies the algorithm presentation, reaching a better delay bound.

**Organization of the paper.** In Section 2 we introduce the setting and its corresponding enumeration problem. In Section 3, we present our data structure for storing and enumerating the outputs, and in Section 4 we show the evaluation algorithm. Section 5 offers the application of the algorithmic results to document spanners and Section 6 shows how to extend these results to deal with complex document editing. We finish the paper with future work in Section 7.

## 2 Setting and main problem

In this section, we present the setting and state the main result. First, we define straight-line programs, which we will use for the compressed representation of input documents. Then we introduce the definition of annotated automaton, an extension of regular automata to produce outputs. We use annotated automata as our computational model to represent queries over documents. By combining both formalisms, we state the main enumeration problem and main technical result.

**Documents.** Given a finite alphabet $\Sigma$, a document $d$ over $\Sigma$ (or just a document) is a string $d = a_1 a_2 \dots a_n \in \Sigma^*$. Given documents $d_1$ and $d_2$, we write $d_1 \cdot d_2$ (or just $d_1 d_2$) for the concatenation of $d_1$ and $d_2$. We denote by $|d| = n$ the length of the document $d = a_1 \dots a_n$ (i.e., the number of symbols) and by $\varepsilon$ the document of length 0. We use $\Sigma^*$ to denote the set of all documents, and $\Sigma^+$ for all documents with one or more symbols. To simplify the notation, in the sequel we will use $d$ for a document, and $a$ or $b$ for a symbol in $\Sigma$.

**SLP compression.** A context-free grammar is a tuple $G = (N, \Sigma, R, S_0)$, where $N$ is a non-empty set of non-terminals, $\Sigma$ is finite alphabet, $S_0 \in N$ is the start symbol and $R \subseteq N \times (N \cup \Sigma)^+$ is the set of rules. As a convention, the rule $(A, w) \in R$ is commonly written as $A \to w$, and $\Sigma$ and $N$ are called terminal and non-terminal symbols, respectively. A context-free grammar $S = (N, \Sigma, R, S_0)$ is a *straight-line program* (SLP) if $R$ is a total function from $N$ to $(N \cup \Sigma)^+$ and the directed graph $(N, \{(A, B) \mid (A, w) \in R \text{ and } B \text{ appears in } w\})$ is acyclic. For every $A \in N$, let $R(A)$ be the unique $w \in (N \cup \Sigma)^+$ such that $(A, w) \in R$, and for every $a \in \Sigma$ let $R(a) = a$. We extend $R$ to a morphism $R^* : (N \cup \Sigma)^* \to \Sigma^*$ recursively such that $R^*(d) = d$ when $d$ is a document, and $R^*(\alpha_1 \dots \alpha_n) = R^*\big(R(\alpha_1) \cdot \ldots \cdot R(\alpha_n)\big)$, where $\alpha_i \in (N \cup \Sigma)$, for $i \leq n$. By our definition of SLP, $R^*(A)$ is in $\Sigma^+$, and uniquely defined for each $A \in N$. Then we define the document encoded by $S$ as $\mathrm{doc}(S) = R^*(S_0)$.

▶ **Example 1.** Let $S = (N, \Sigma, R, S_0)$ be a SLP with $N = \{S_0, A, B\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, and $R = \{S_0 \to A\mathsf{r}BABA, A \to \mathsf{ba}, B \to A\mathsf{ra}\}$. We then have that $\mathrm{doc}(A) = \mathsf{ba}$, $\mathrm{doc}(B) = \mathsf{bara}$ and $\mathrm{doc}(S) = \mathrm{doc}(S_0) = \mathsf{barbarababaraba}$, namely, the string represented by $S$.

We define the size of an SLP $S = (N, \Sigma, R, S_0)$ as $|S| = \sum_{A \in N} |R(A)|$, namely, the sum of the lengths of the right-hand sides of all rules. It is important to note that an SLP $S$ can encode a document $\mathrm{doc}(S)$ such that $|\mathrm{doc}(S)|$ is exponentially larger with respect to $|S|$. For this reason, SLPs stay as a commonly used data compression scheme [28, 20, 24, 11], and they are often studied in particular because of their algorithmic properties; see [21] for a survey. In this paper, we consider SLP compression to represent documents and use the formalism of annotated automata for extracting relevant information from the document.

**Annotated automata.** An *annotated automaton* (AnnA for short) is a finite state automaton where we label transitions with annotations. Formally, it is a tuple $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where $Q$ is a state set, $\Sigma$ is an input alphabet, $\Omega$ is an output alphabet, $q_0 \in Q$ and $F \subseteq Q$ are the initial state and final set of states, respectively, and: $\Delta \subseteq Q \times \Sigma \times Q \cup Q \times (\Sigma \times \Omega) \times Q$ is the transition relation, which contains *read transitions* of the form $(p, a, q) \in Q \times \Sigma \times Q$, and *read-write transitions* of the form $(p, (a, \circledcirc), q) \in Q \times (\Sigma \times \Omega) \times Q$.

Similarly to transducers [7], a symbol $a \in \Sigma$ is an input symbol that the machine reads and $\circledcirc \in \Omega$ is a symbol that indicates what the machine prints in an output tape. A run $\rho$ of $\mathcal{A}$ over a document $d = a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence of the form $\rho = q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$ such that for each $i \in [1, n]$ one of the following holds: (1) $b_i = a_i$ and there is a transition $(q_{i-1}, a_i, q_i) \in \Delta$ or (2) $b_i = (a_i, \circledcirc)$ and there is a transition $(q_{i-1}, (a_i, \circledcirc), q_i) \in \Delta$. We say that $\rho$ is accepting if $q_n \in F$.

We define the *annotation* of $\rho$ as $\mathsf{ann}(\rho) = \mathsf{ann}(b_1) \cdot \ldots \cdot \mathsf{ann}(b_n)$ such that $\mathsf{ann}(b_i) = (\circledcirc, i)$ if $b_i = (a, \circledcirc)$, and $\mathsf{ann}(b_i) = \varepsilon$ otherwise, for each $i \in [1, n]$. Given an annotated automaton $\mathcal{A}$ and a document $d \in \Sigma^*$, we define the set $[\![\mathcal{A}]\!](d)$ of all outputs of $\mathcal{A}$ over $d$ as: $[\![\mathcal{A}]\!](d) = \{\mathsf{ann}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ over } d\}$. Note that each output in $[\![\mathcal{A}]\!](d)$ is a sequence of the form $(\circledcirc_1, i_1) \dots (\circledcirc_k, i_k)$ for some $k \leq n$ where $i_1 < i_2 < \dots < i_k$ and each $(\circledcirc_j, i_j)$ means that position $i_j$ is annotated with the symbol $\circledcirc_j$.

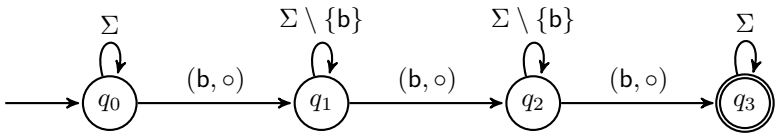


**Figure 1** Example of an annotated automaton.

▶ **Example 2.** Consider an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, $\Omega = \{\circ\}$, and $F = \{q_3\}$. We define $\Delta$ as the set of transitions that are depicted in Figure 1. For the document $d = \mathsf{barbarababaraba}$ the set $[\![A]\!](d)$ contains the results $(\circ, 1)(\circ, 4)(\circ, 8)$, $(\circ, 4)(\circ, 8)(\circ, 10)$ and $(\circ, 8)(\circ, 10)(\circ, 14)$. Intuitively, $\mathcal{A}$ selects triples of $\mathsf{b}$ which are separated by characters other than $\mathsf{b}$.

Annotated automata are the natural regular counterpart of annotated grammars introduced in [4]. Moreover, it is the generalization and simplification of similar automaton formalisms introduced in the context of information extraction [15, 23], complex event processing [18, 17], and enumeration in general [8, 22]. In Section 5, we show how we can reduce the automaton model of document spanners, called a variable-set automaton, into a (succinctly) annotated automaton, generalizing the setting in [25].

As for other automata models, the notion of an unambiguous automaton is crucial for removing duplicate outputs. We say that an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ is *unambiguous* if for every $d \in \Sigma^*$ and every $\nu \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run $\rho$ of $\mathcal{A}$ over $d$ such that $\nu = \mathsf{ann}(\rho)$. On the other hand, we say that $\mathcal{A}$ is *deterministic* if $\Delta$ is a partial function of the form $\Delta : (Q \times \Sigma \cup Q \times (\Sigma \times \Omega)) \to Q$. Note that a deterministic AnnA is also unambiguous. The definition of unambiguous is in line with the notion of unambiguous annotated grammar [4] (see also [22]), and determinism with the idea of I/O-determinism used in [16, 8, 18]. As usual, one can easily show that for every AnnA $\mathcal{A}$ there exists an equivalent deterministic AnnA and, therefore, an equivalent unambiguous AnnA.

▶ **Lemma 3.** *For every annotated automaton $\mathcal{A}$ there exists a deterministic annotated automaton $\mathcal{A}'$ such that $[\![\mathcal{A}]\!](d) = [\![\mathcal{A}']\!](d)$ for every $d \in \Sigma^*$.*

Regarding the expressiveness of annotated automata, one can note that they have the same expressive power as MSO formulas with monadic second-order free variables. We refer the reader to [22] for an equivalent result in the context of nested documents.

**Main problem and main result.** We are interested in the problem of evaluating annotated automata over an SLP-compressed document, namely, to enumerate all the annotations over the document represented by an SLP. Formally, we define the main evaluation problem of this paper as follows. Let $\mathcal{C}$ be a class of AnnA (e.g. unambiguous AnnA).

| | |
|---|---|
| **Problem:** | SLPEnum[$\mathcal{C}$] |
| **Input:** | an AnnA $\mathcal{A} \in \mathcal{C}$ and an SLP $S$ |
| **Output:** | Enumerate $[\![\mathcal{A}]\!](\mathrm{doc}(S))$. |

As it is common for enumeration problems, we want to impose an efficiency guarantee regarding the preprocessing of the input (e.g., $\mathcal{A}$ and $S$) and the delay between two consecutive outputs. For this purpose, one often divides the work of the enumeration algorithm into two phases: first, a *preprocessing phase* in which it receives the input and produces some object $D$ (e.g., a collection of indices) which encodes the expected output and, second, an

*enumeration phase* which extracts the results from $D$. We say that such an algorithm has *$f$-preprocessing time* if there exists a constant $c$ such that, for every input $\mathcal{I}$, the time for the preprocessing phase of $\mathcal{I}$ is bounded by $c \cdot f(|\mathcal{I}|)$. Instead, we say that the algorithm has *output-linear delay* if there exists a constant $d$ such that whenever the enumeration phase delivers the sequence of outputs $O_1, \ldots, O_\ell$, the time for producing the next output $O_i$ is bounded by $c \cdot |O_i|$ for every $i \leq \ell$. As is expected, we assume here the computational model of Random Access Machines (RAM) with uniform cost measure and addition and subtraction as basic operations [1]. For a formal presentation of the output-linear delay guarantee, we refer the reader to [16]. As it is commonly done on algorithms over SLPs and other compression schemes, we assume that the registers in the underlying RAM-model allow for constant-time arithmetical operations over positions in the *uncompressed* document (i.e., they have $\mathcal{O}(\log |\operatorname{doc}(S)|)$ size).

The notion of output-linear delay is a refinement of the better-known constant-delay bound, which requires that each output has a constant size (i.e., concerning the input). Since even the document encoded by an SLP can be of exponential length, it is more reasonable in our setting to use the output-linear delay guarantee.

The following is the main technical result of this work.

▶ **Theorem 4.** *Let $\mathcal{C}$ be the class of all unambiguous AnnAs. Then one can solve the problem SLPEnum[$\mathcal{C}$] with linear preprocessing time and output-linear delay. Specifically, there exists an enumeration algorithm that runs in $|\mathcal{A}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating $[\![\mathcal{A}]\!](\operatorname{doc}(S))$ given an unambiguous AnnA $\mathcal{A}$ and a SLP $S$.*

We dedicate the rest of the paper to presenting the enumeration algorithm of Theorem 4. In Section 4 we explain the preprocessing phase of the algorithm. Before that, in the next section, we explain how Enumerable Compact Sets with Shifts work, which is the data structure used in the preprocessing phase to store outputs.

## 3 Enumerable compact sets with shifts

We present here the data structure, called Enumerable Compact Sets with Shifts, to compactly store the outputs of evaluating an annotated automaton over a straight-line program. This structure extends Enumerable Compact Sets (ECS) introduced in [22] (we note that a similar data structure for constant-delay enumeration was previously proposed in [2]). Indeed, people have also used ECS extensions in [4, 9]. This new version extends ECS by introducing a shift operator, which helps compactly move all outputs' positions with a single call. Although the shift nodes require a revision of the complete ECS model, it simplifies the evaluation algorithm in Section 4 and achieves output-linear delay for enumerating all outputs. For completeness of presentation, this section goes through all main details as in [22].

**The structure.** Let $\Omega$ be an output alphabet such that $\Omega$ has no elements in common with $\mathbb{Z}$ or $\{\cup, \odot\}$ (i.e., $\Omega \cap \mathbb{Z} = \emptyset$ and $\Omega \cap \{\cup, \odot\} = \emptyset$). We define an *Enumerable Compact Set with Shifts* (Shift-ECS) as a structure $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ such that $V$ is a finite sets of nodes, $\ell \colon V \to V$ and $r \colon V \to V$ are the *left* and *right* partial functions, and $\lambda \colon V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot\}$ is a labeling function. We assume that $\mathcal{D}$ forms an acyclic graph (i.e., $(V, \{(v, \ell(v)), (v, r(v)) \mid v \in V\})$ is acyclic). Further, for every node $v \in V$, $\ell(v)$ is defined iff $\lambda(v) \in \mathbb{Z} \cup \{\cup, \odot\}$, and $r(v)$ is defined iff $\lambda(v) \in \{\cup, \odot\}$. Notice that, by definition, nodes labeled by $\Omega$ are bottom nodes in the acyclic structure formed by $\mathcal{D}$, and nodes labeled by $\mathbb{Z}$ or $\{\cup, \odot\}$ are inner

nodes. Here, $\mathbb{Z}$-nodes are unary operators (i.e., $r(\cdot)$ is not defined over them), and $\cup$-nodes or $\odot$-nodes are binary operators. Indeed, we say that $v \in V$ is a *bottom node* if $\lambda(v) \in \Omega$, a *product node* if $\lambda(v) = \odot$, a *union node* if $\lambda(v) = \cup$, and a *shift node* if $\lambda(v) \in \mathbb{Z}$. Finally, we define $|\mathcal{D}| = |V|$.

The outputs retrieved from a Shift-ECS are strings of the form $(\mathfrak{o}_1, i_1)(\mathfrak{o}_2, i_2)\dots(\mathfrak{o}_\ell, i_\ell)$, where $\mathfrak{o}_j \in \Omega$ and $i_j \in \mathbb{Z}$. To build them, we use the *shifting function* $\mathsf{sh} : (\Omega\times\mathbb{Z})\times\mathbb{Z} \to (\Omega\times\mathbb{Z})$ such that $\mathsf{sh}((\mathfrak{o}, i), k) = (\mathfrak{o}, i + k)$. We extend this function to strings over $\Omega \times \mathbb{Z}$ such that $\mathsf{sh}((\mathfrak{o}_1, i_1)\dots(\mathfrak{o}_\ell, i_\ell), k) = (\mathfrak{o}_1, i_1 + k)\dots(\mathfrak{o}_\ell, i_\ell + k)$ and to set of strings such that $\mathsf{sh}(L, k) = \{\mathsf{sh}(\nu, k) \mid \nu \in L\}$ for every $L \subseteq (\Omega \times \mathbb{Z})^*$.

Each node $v \in V$ of a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ defines a set of output strings. Specifically, we associate a set of strings $[\![\mathcal{D}]\!](v)$ recursively as follows: (1) $[\![\mathcal{D}]\!](v) = \{(\mathfrak{o}, 1)\}$ whenever $\lambda(v) = \mathfrak{o} \in \Omega$, (2) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cup [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \cup$, (3) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cdot [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \odot$, where $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$, and (4) $[\![\mathcal{D}]\!](v) = \mathsf{sh}([\![\mathcal{D}]\!](\ell(v)), \lambda(v))$ whenever $\lambda(v) \in \mathbb{Z}$.

▶ **Example 5.** Suppose $\Omega = \{x, y\}$. Consider the Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ where $V = \{v_1, v_2, v_3, v_4, v_5\}$, $\ell(v_1) = v_4$, $r(v_1) = v_2$, $\ell(v_2) = v_3$, $\ell(v_3) = v_4$, $r(v_3) = v_5$, $\lambda(v_1) = \odot$, $\lambda(v_2) = +2$, $\lambda(v_3) = \cup$, $\lambda(v_4) = x$ and $\lambda(v_5) = y$. We show this Shift-ECS in Figure 2 (a). The sets of words $[\![\mathcal{D}]\!]$ associated to each node are thus: $[\![\mathcal{D}]\!](v_4) = \{(x, 1)\}$, $[\![\mathcal{D}]\!](v_5) = \{(y, 1)\}$, $[\![\mathcal{D}]\!](v_3) = \{(x, 1), (y, 1)\}$, $[\![\mathcal{D}]\!](v_2) = \{(x, 3), (y, 3)\}$ and $[\![\mathcal{D}]\!](v_1) = \{(x, 1)(x, 3), (x, 1)(y, 3)\}$.

**Enumeration.** Given that every node of a Shift-ECS represents a set of strings, we are interested in enumerating them with output-linear delay. Specifically, we focus on the following problem. Let $\mathcal{C}$ be a class of Enumerable Compact Sets with Shifts.

| | |
|---|---|
| **Problem:** | ShiftECSEnum[$\mathcal{C}$] |
| **Input:** | a Shift-ECS $\mathcal{D} \in \mathcal{C}$ and a node $v$ of $\mathcal{D}$ |
| **Output:** | Enumerate $[\![\mathcal{D}]\!](v)$. |

The plan then is to provide an enumeration algorithm with output-linear delay for ShiftECSEnum[$\mathcal{C}$] and some helpful class $\mathcal{C}$. A reasonable strategy to enumerate the set $[\![\mathcal{D}]\!](v)$ is to do a traversal on the structure while accumulating the shift values in the path to each leaf. However, to be able to do this without repetitions and output-linear delay, we need to guarantee two conditions: first, that one can obtain every output from $\mathcal{D}$ in only one way and, second, union and shift nodes are *close* to an output node (i.e., a bottom node or a product node), in the sense that we can always reach them in a bounded number of steps. To ensure that these conditions hold, we impose two restrictions for an ECS.

For the first restriction, we say that an ECS $\mathcal{D}$ is *duplicate-free* if the following hold: (1) for every union node $v$ in $\mathcal{D}$ it holds that $[\![\mathcal{D}]\!](\ell(v))$ and $[\![\mathcal{D}]\!](r(v))$ are disjoint and (2) for every product node $v$ and for every $w \in [\![\mathcal{D}]\!](v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](\ell(v))$ and $w_2 \in [\![\mathcal{D}]\!](r(v))$.

For the second restriction, we define *k-bounded* Shift-ECS. Given a Shift-ECS $\mathcal{D}$, define the (left) output-depth of a node $v \in V$, denoted by $\mathsf{odepth}_\mathcal{D}(v)$, recursively as follows: $\mathsf{odepth}_\mathcal{D}(v) = 0$ whenever $\lambda(v) \in \{\odot\} \cup \Omega$, and $\mathsf{odepth}_\mathcal{D}(v) = \mathsf{odepth}_\mathcal{D}(\ell(v)) + 1$ whenever $\lambda(v) \in \{\cup\} \cup \mathbb{Z}$. Then, for $k \in \mathbb{N}$ we say that $\mathcal{D}$ is $k$-bounded if $\mathsf{odepth}_\mathcal{D}(v) \leq k$ for all $v \in V$.

▶ **Proposition 6.** *Fix $k \in \mathbb{N}$. Let $\mathcal{C}_k$ be the class of all duplicate-free and $k$-bounded Shift-ECSs. Then one can solve the problem ShiftECSEnum[$\mathcal{C}_k$] with output-linear delay and without preprocessing (i.e. constant preprocessing time).*

**Operations.** The next step is to provide a set of operations that allow extending a Shift-ECS $\mathcal{D}$ in a way that maintains $k$-boundedness. Fix a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$. Then for any $\circledcirc \in \Omega$, $v_1, \ldots, v_4, v \in V$ and $k \in \mathbb{Z}$, we define the operations:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{add}(\circledcirc) & \to & v' & \mathsf{prod}(v_1, v_2) & \to & v' \\ \mathsf{union}(v_3, v_4) & \to & v' & \mathsf{shift}(v, k) & \to & v' \end{array}$$

such that $[\![\mathcal{D}]\!](v') := \{(\circledcirc, 1)\}$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$; and $[\![\mathcal{D}]\!](v') := \mathsf{sh}([\![\mathcal{D}]\!](v), k)$, respectively. Here we assume that the union and prod respect properties (1) and (2) of a duplicate-free Shift-ECS, namely, $[\![\mathcal{D}]\!](v_3)$ and $[\![\mathcal{D}]\!](v_4)$ are disjoint and, for every $w \in [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](v_1)$ and $w_2 \in [\![\mathcal{D}]\!](v_2)$.

Strictly speaking, each operation above should receive as input the data structure $\mathcal{D}$, and output a fresh node $v'$ plus a new data structure $\mathcal{D}' = (\Omega, V', \ell', r', \lambda')$ such that $\mathcal{D}'$ is an extension of $\mathcal{D}$, namely, $\mathsf{obj} \subseteq \mathsf{obj}'$ for every $\mathsf{obj} \in \{V, \ell, r, \lambda\}$ and $v' \in V' \setminus V$. Note that we assume that each operation can only extend the data structure with new nodes and that old nodes are immutable after each operation. For simplification, we will not explicitly refer to $\mathcal{D}$ on the operations above, although they modify $\mathcal{D}$ directly by adding new nodes.

To define the above operations, we impose further restrictions on the structure below the operations' input nodes to ensure $k$-boundedness. Towards this goal, we introduce the notion of safe nodes. We say that a node $v \in V$ is *safe* if $v$ is a shift node and either $\ell(v)$ is an output node (i.e., a bottom or product node), or $u = \ell(v)$ is an union node, $\mathsf{odepth}_{\mathcal{D}}(u) = 1$, and $r(u)$ is a shift node with $\mathsf{odepth}_{\mathcal{D}}(r(u)) \leq 2$. In other words, $v$ is safe if it is a shift node over an output node or over a union node with an output on the left and a shift node on the right, whose output depth is less or equal to 2. The trick then is to show that all operations over Shift-ECSs receive only safe nodes and always output safe nodes. As we will see, safeness will be enough to provide a light structural restriction on the operations' input nodes in order to maintain $k$-boundedness after each operation.

Next, we show how to implement each operation assuming that every input node is safe. In fact, the cases of add and shift are straightforward. For $\mathsf{add}(\circledcirc) \to v'$ we extend $\mathcal{D}$ with two fresh nodes $v'$ and $u$ such that $\lambda(u) = \circledcirc$, $\lambda(v') = 0$, and $\ell(v') = u$. In other words, we hang a fresh 0-shift node $v'$ over a fresh $\circledcirc$-node $u$, and output $v'$. For $\mathsf{shift}(v, k) \to v'$, add the fresh node $v'$ to $\mathcal{D}$, and set $\ell(v') = \ell(v)$ and $\lambda(v') = \lambda(v) + k$. One can easily check that in both cases the node $v'$ represents the desired set, is safe, and $k$-boundedness is preserved.

To show how to implement $\mathsf{prod}(v_1, v_2) \to v'$, recall that $v_1$ and $v_2$ are safe and, in particular, both are shift nodes. Then we need to extend $\mathcal{D}$ with fresh nodes $v'$, $v''$, and $v'''$ such that $\ell(v') = v''$, $\ell(v'') = \ell(v_1)$, $r(v'') = v'''$, $\ell(v''') = \ell(v_2)$, $\lambda(v') = \lambda(v_1)$, $\lambda(v'') = \odot$ and $\lambda(v''') = \lambda(v_2) - \lambda(v_1)$. Figure 2(b) shows a diagram of this gadget. One can easily check that $v'$ represents the product of $v_1$ and $v_2$, $v'$ is safe, and the new version of $\mathcal{D}$ is $k$-bounded whenever $\mathcal{D}$ is also $k$-bounded.

The last operation is $\mathsf{union}(v_3, v_4) \to v'$. For the sake of presentation, we only provide the construction for the most involved case, which is when both $u_3 = \ell(v_3)$ and $u_4 = \ell(v_4)$ are union nodes. We show the gadget for this case in Figure 2(c). This construction has several interesting properties. First, one can check that $[\![\mathcal{D}]\!](v') = [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$ since each shift value is carefully constructed so that the accumulated shift value from $v'$ to each node remains unchanged. Thus, the semantics is well-defined. Second, union can be computed in constant time in $|\mathcal{D}|$ given that we only need to add a fixed number of fresh nodes. Furthermore, the produced node $v'$ is safe, although some of the new nodes are not necessarily safe. Finally, the new $\mathcal{D}$ is 3-bounded whenever $\mathcal{D}$ is 3-bounded. To see this, we

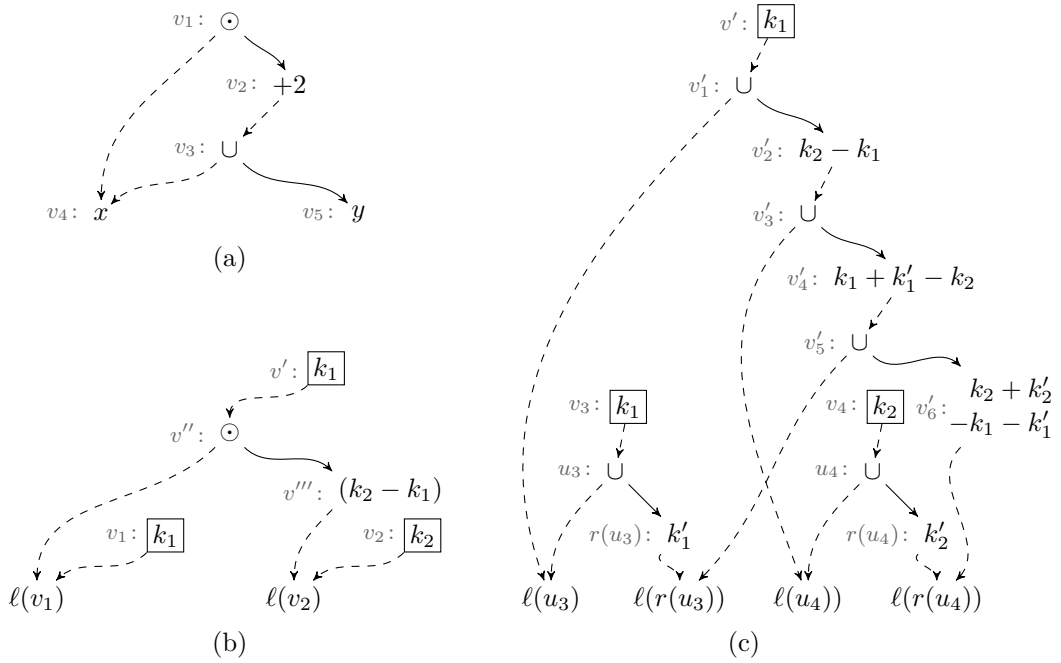


**Figure 2** (a) An example of a Shift-ECS with output alphabet $\{x, y\}$. (b) Gadget for $\mathsf{prod}(\mathcal{D}, v_1, v_2, k)$. (c) Gadget for $\mathsf{union}(\mathcal{D}, v_3, v_4)$. We use dashed and solid edges for the left and right mappings, respectively. Node names are in grey at the left of each node. In (b) and (c), square nodes are the input and output nodes of each operation.

first have to notice that $\ell(u_3)$ and $\ell(u_4)$ are output nodes, and that $\mathsf{odepth}(\ell(r(u_3))) \leq 1$ and $\mathsf{odepth}(\ell(r(u_4))) \leq 1$. We can check the depth of each node going from the bottom to the top: $\mathsf{odepth}(v'_6) \leq 2$, $\mathsf{odepth}(v'_5) \leq 2$, $\mathsf{odepth}(v'_4) \leq 3$, $\mathsf{odepth}(v'_3) \leq 1$, $\mathsf{odepth}(v'_2) \leq 2$, $\mathsf{odepth}(v'_1) \leq 1$ and $\mathsf{odepth}(v') \leq 2$.

By the previous discussion, if we start with a Shift-ECS $\mathcal{D}$ which is 3-bounded (in particular, empty) and we apply the add, prod, union and shift operators between safe nodes (which also produce safe nodes), then the result is 3-bounded as well. Furthermore, the data structure is fully-persistent [13]: for every node $v$ in $\mathcal{D}$, $[\![\mathcal{D}]\!](v)$ is immutable after each operation. Finally, by Proposition 6, the result can be enumerated with output-linear delay.

▶ **Theorem 7.** *The operations* add, prod, union *and* shift *take constant time and are fully persistent. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply these operations over safe nodes, the result node $v'$ is always a safe node and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay (without preprocessing) for every node $v$.*

**The empty- and $\varepsilon$-nodes.** The last step of constructing our model of Shift-ECS is the inclusion of two special nodes that produce the empty set and the empty string, called empty- and $\varepsilon$-nodes, respectively.

We start with the empty node, which is easier to incorporate into a Shift-ECS. Consider a special node $\perp$ and include it on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\perp) = \emptyset$. Then extend the operations prod, union, and shift accordingly to the empty set, namely, $\mathsf{prod}(v_1, v_2) \to \perp$ whenever $v_1$ or $v_2$ is equal to $\perp$, $\mathsf{union}(v, \perp) = \mathsf{union}(\perp, v) \to v$, and $\mathsf{shift}(\perp, k) \to \perp$ for every nodes $v_1, v_2, v$, and $k \in \mathbb{Z}$. It is easy to check that one can include the $\perp$-node into Shift-ECSs without affecting the guarantees of Theorem 7.

The other special node is the $\varepsilon$-node. Let $\varepsilon$ denote a special node, included on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\varepsilon) = \{\varepsilon\}$. With these new nodes in a Shift-ECS, we need to revise our notions of output-depth, duplicate-free, and $k$-boundedness to change the enumeration algorithm, and to extend the operations add, prod, union, and shift over so-called $\varepsilon$-safe nodes (i.e., the extension of safe nodes with $\varepsilon$). Given space restrictions, we omit the details on how to implement these $\varepsilon$-nodes and how we can preserve Proposition 6 and Theorem 7.

For the rest of the paper, we assume that a Shift-ECS is a tuple $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \bot, \varepsilon)$ where we define $\Omega, V, \ell, r, \lambda$ as before, and $\bot, \varepsilon \in V$ are the empty and $\varepsilon$ nodes, respectively. Further, we assume that $\ell$, $r$, and $\lambda$ are extended accordingly, namely, $\ell(v)$ and $r(v)$ are not defined whenever $v \in \{\bot, \varepsilon\}$, and $\lambda : V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot, \bot, \varepsilon\}$ such that $\lambda(v) = \bot$ ($\lambda(v) = \varepsilon$) iff $v = \bot$ ($v = \varepsilon$, resp.). Finally, we can extend Theorem 7 for the Shift-ECS extension as follows.

▶ **Theorem 8.** *The operations* add, prod, union *and* shift *over Shift-ECS extended with bot- and $\varepsilon$-nodes take constant time. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply* add, prod, union, *and* shift *over $\varepsilon$-safe nodes, the resulting node $v'$ is always an $\varepsilon$-safe node, and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay without preprocessing for every node $v$.*

## 4 Evaluation of annotatated automata over SLP-compressed strings

This section shows our algorithm for evaluating an annotated automaton over an SLP-compressed document. This evaluation is heavily inspired by the preprocessing phase in [25], as it primarily adapts the algorithm to the Shift-ECS data structure. In a nutshell, we keep matrices of Shift-ECS nodes, where each matrix represents the outputs of all partial runs of the annotated automaton over fragments of the compressed strings. We extend the operations of Shift-ECS over matrices of nodes, which will allow us to compose matrices, and thus compute sequences of compressed strings. Then the algorithm proceeds in a dynamic programming fashion, where matrices are computed bottom-up for each non-terminal symbol. Finally, the start symbol of the SLP will contain all the outputs. The result of this process is that each matrix entry succinctly represents an output set, can be operated in constant time, and can be enumerated with output-linear delay.

**Matrices of nodes.** The main ingredient for the evaluation algorithm are matrices of nodes for encoding partial runs of annotated automata. To formalize this notion, fix an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ and a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \bot, \varepsilon)$. We define a *partial run* $\rho$ of $\mathcal{A}$ over a document $d = a_1 a_2 \dots a_n \in \Sigma^*$ as a sequence $\rho = p_0 \xrightarrow{b_1} \dots \xrightarrow{b_n} p_n$ such that either $b_i = a_i$ and $(q_{i-1}, a_i, q_i) \in \Delta$, or $b_i = (a_i, \circledcirc)$ and $(q_{i-1}, (a_i, \circledcirc), q_i) \in \Delta$. Additionally, we say that the partial run $\rho$ is from state $p$ to state $q$ if $p_0 = p$ and $p_n = q$. In other words, partial runs are almost equal to runs, except they can start at any state $p$.

For the algorithm, we use the set of all $Q \times Q$ matrices where entry $M[p, q]$ is a node in $V$ for every $p, q \in Q$. Intuitively, each node $M[p, q]$ represents all outputs $[\![\mathcal{D}]\!](M[p, q])$ of partial runs from state $p$ to state $q$, which can be enumerated with output-linear delay by Theorem 8. Intuitively, $M[p, q] = \bot$ represents that there is no run from $p$ to $q$, and $M[p, q] = \varepsilon$ represents that there is a single run without outputs.

To combine matrices over $\mathcal{D}$-nodes, we define two operations. The first operation is the *matrix multiplication* over the semiring $(2^{(\Omega \times \mathbb{Z})^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ but represented over $\mathcal{D}$. Formally, let $Q = \{q_0, \dots, q_{m-1}\}$ with $m = |Q|$. Then, for two $m \times m$ matrices $M_1$ and $M_2$, we define $M_1 \otimes M_2$ such that for every $p, q \in Q$:

**Algorithm 1** The enumeration algorithm of an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ over an SLP $S = (N, \Sigma, R, S_0)$.

```
1:  procedure Evaluation(A, S)
2:      Initialize D as an empty Shift-ECS
3:      NonTerminal(S_0)
4:      v ← ⊥
5:      for each q ∈ F do
6:          v ← union(v, M_{S_0}[q_0, q])
7:      Enumerate(v, D)
8:  procedure Terminal(a)
9:      M_a ← {[p, q] → ⊥ | p, q ∈ Q}
10:     for each (p, (a, ω), q) ∈ Δ do
11:         M_a[p, q] ← union(M_a[p, q], add(ω))
12:     for each (p, a, q) ∈ Δ do
13:         M_a[p, q] ← union(M_a[p, q], ε)
14:     len_a ← 1
15: procedure NonTerminal(X)
16:     M_X ← {[p, q] → ⊥ | p, q ∈ Q, p ≠ q} ∪
              {[p, q] → ε | p, q ∈ Q, p = q}
17:     len_X ← 0
18:     for i = 1 to |R(X)| do
19:         Y ← R(X)[i]
20:         if M_Y is not defined then
21:             if Y ∈ Σ then
22:                 Terminal(Y)
23:             else
24:                 NonTerminal(Y)
25:         M_X ← M_X ⊗ shift(M_Y, len_X)
26:         len_X ← len_X + len_Y
```

$$(M_1 \otimes M_2)[p,q] := \mathsf{union}_{i=0}^{m-1}\Big(\mathsf{prod}\big(M_1[p,q_i], M_2[q_i,q]\big)\Big)$$

where $\mathsf{union}_{i=0}^{m-1} E_i := \mathsf{union}(\ldots \mathsf{union}(\mathsf{union}(E_1, E_2), E_3) \ldots, E_m)$. In other words, the node $(M_1 \otimes M_2)[p,q]$ represents the set $\bigcup_{i=0}^{m-1} \big([\![\mathcal{D}]\!](M_1[p,q_i]) \cdot [\![\mathcal{D}]\!](M_2[q_i,q])\big)$.

The second operation for matrices is the extension of the *shift operation*. Formally, $\mathsf{shift}(M,k)[p,q] := \mathsf{shift}(M[p,q],k)$ for a matrix $M$, $k \in \mathbb{Z}$, and $p, q \in Q$. Since each operation over $\mathcal{D}$ takes constant time, overall multiplying $M_1$ with $M_2$ takes time $\mathcal{O}(|Q|^3)$ and shifting $M$ with $k$ takes time $\mathcal{O}(|Q|^2)$.

**The algorithm.** We present the evaluation algorithm for the SLPEnum problem in Algorithm 1. As expected, the main procedure Evaluation receives as input an unambiguous annotated automaton $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ and an SLP $S = (N, \Sigma, R, S_0)$, and enumerates all outputs in $[\![\mathcal{A}]\!](\mathrm{doc}(S))$. To simplify the notation, in Algorithm 1 we assume that $\mathcal{A}$ and $S$ are globally defined, and we can access them in any subprocedure. Similarly, we use a Shift-ECS $\mathcal{D}$, and matrix $M_X$ and integer $\mathrm{len}_X$ for every $X \in N \cup \Sigma$, which can globally be accessed at any place as well.

The main purpose of the algorithm is to compute $M_X$ and $\mathrm{len}_X$. On one hand, $M_X$ is a $Q \times Q$ matrix where each node entry $M_X[p,q]$ represents all outputs of partial runs from $p$ to $q$. On the other hand, $\mathrm{len}_X$ is the length of the string $R^*(X)$ (i.e., the string produced from $X$). Both $M_X$ and $\mathrm{len}_X$ start undefined, and we compute them recursively, beginning from the non-terminal symbol $S_0$ and by calling the method NonTerminal$(S_0)$ (line 3). After $M_{S_0}$ was computed, we can retrieve the set $[\![\mathcal{A}]\!](S)$ by taking the union of all partial run's outputs from the initial state $q_0$ to a state $q \in F$, and storing it in node $v$ (lines 4–6). Finally, we can enumerate $[\![\mathcal{A}]\!](S)$ by enumerating all outputs represented by $v$ (line 7).

The workhorses of the evaluation algorithm are procedures NonTerminal and Terminal in Algorithm 1. The former computes matrices $M_X$ recursively whereas the latter is in charge of the base case $M_a$ for a terminal $a \in \Sigma$. For computing this base case, we can start with $M_a$ with all entries equal to the empty node $\bot$ (line 9). Then if there exists a read-write transition $(p, (a, \omega), q) \in \Delta$, we add an output node $\omega$ to $M_a[p,q]$, by making the

union between the current node at $M_a[p,q]$ with the node $\mathsf{add}(\circ)$ (line 11). Also, if a read transition $(p,a,q) \in \Delta$ exists, we do the same but with the $\varepsilon$-node (line 13). Finally, we set the length of $\text{len}_a$ to 1, and we have covered the base case.

For the recursive case (i.e., procedure NonTerminal$(X)$), we start with a sort of "identity matrix" $M_X$ where all entries are set up to the empty-node except the ones where $p = q$ that are set up to the $\varepsilon$-node, and the value $\text{len}_X = 0$ (lines 16–17). Then we iterate sequentially over each symbol $Y$ of $R(X)$, where we use $R(X)[i]$ to denote the $i$-th symbol of $R(X)$ (lines 18–19). If $M_Y$ is not defined, then we recursively compute Terminal$(Y)$ or NonTerminal$(Y)$ depending on whether $Y$ is in $\Sigma$ or not, respectively (lines 20–24). The matrix $M_Y$ is memoized (by having the check in line 20 to see if it is defined or not) so we need to compute it at most once. After we have retrieved $M_Y$, we can compute all outputs for $R(X)[1] \dots R(X)[i]$ by multiplying the current version of $M_X$ (i.e., the outputs of $R(X)[1] \dots R(X)[i-1]$), with the matrix $M_Y$ shifted by the current length $\text{len}_X$ (line 25). Finally, we update the current length of $X$ by adding $\text{len}_Y$ (line 26).

Regarding correctness, the algorithm follows a direct matrix evaluation over the SLP grammar, where its correctness depends on the Shift-ECS $\mathcal{D}$. Notice that, although all operations over nodes are not necessarily duplicate-free, we know that the runs from the initial state $q_0$ to the final states are unambiguous. Then the operations used for the final output are duplicate-free. Regarding performance, the main procedure calls NonTerminal or Terminal at most once for every symbol. Note that after making all calls to Terminal, each transition in $\Delta$ is seen exactly once, and NonTerminal takes time at most $\mathcal{O}(|R(X)| \times |Q|^3)$ not taking into account the calls inside. Overall, the preprocessing time is $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$.

▶ **Theorem 9.** *Algorithm 1 enumerates the set $[\![\mathcal{A}]\!](S)$ correctly for every unambiguous AnnA $\mathcal{A}$ and every SLP-compressed document $S$, with output-linear delay and after a preprocessing phase that takes time $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$.*

We want to finish by noticing that, contrary to [25], our evaluation algorithm does not need to modify the grammar $S$ into Chomsky's normal form (CNF) since we can evaluate $\mathcal{A}$ over $S$ directly. Although passing $S$ into CNF can be done in linear time over $S$ [25], this step can incur an extra cost, which we can avoid in our approach.

## 5 Applications in regular spanners

It was already shown in [4] that working with annotations directly and then providing a reduction from a spanner query to an annotation query is sometimes more manageable. In this section we will do just that: starting from a document-regular spanner pair $(d, \mathcal{M})$, we will show how to build a document-annotated automaton pair $(d', \mathcal{A})$ such that $\mathcal{M}(d) = [\![\mathcal{A}]\!](d')$. Although people have studied various models of regular spanners in the literature, we will focus here on sequential variable-set automata (VA) [15] and sequential extended VA [16]. The latter, which we handle first, is essentially the model that the work of Schmid and Schweikardt used in their results. In the second half of the section we reduce the former to *succinctly* annotated automata, an extension of AnnA that allows output symbols to be stored concisely. These reductions imply constant-delay enumeration for the spanner tasks.

**Variable-set automata.** Consider a document $d = a_1 \dots a_n$ over an input alphabet $\Sigma$. A *span* of $d$ is a pair $[i,j\rangle$ with $1 \leq i \leq j \leq n+1$. We define the substring of $[i,j\rangle$ by $d[i,j\rangle = a_i \dots a_{j-1}$. We also consider a finite set of variables $\mathcal{X}$ and we define a *mapping* as a partial function that maps some of these variables to spans. We define a *document spanner* as a function assigning every input document $d$ to a set of mappings [15].

A *variable-set automaton* (VA for short) is a tuple $\mathcal{A} = (Q, \Sigma, \mathcal{X}, \Delta, q_0, F)$ where $Q$ is a set of states, $q_0 \in Q$, $F \subseteq Q$, and $\Delta$ consists of *read transitions* $(p, a, q) \in Q \times \Sigma \times Q$ and *variable transitions* $(p, \vdash^x, q)$ or $(p, \dashv^x, q)$ where $p, q \in Q$ and $x \in \mathcal{X}$. The symbols $\vdash^x$ and $\dashv^x$ are referred to as *variable markers* of $x$, where $\vdash^x$ is *opening* and $\dashv^x$ is *closing*. Given a document $d = a_1 \dots a_n \in \Sigma^*$ a configuration of $\mathcal{A}$ is a pair $(q, i)$ where $q \in Q$ and $i \in [1, n+1]$. A run $\rho$ of $\mathcal{A}$ over $d$ is a sequence $\rho = (q_0, i_0) \xrightarrow{\sigma_1} (q_1, i_1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_m} (q_m, i_m)$ where $i_0 = 1$, $i_m = n+1$, and for each $j \in [0, m-1]$, $(q_j, \sigma_{j+1}, q_{j+1}) \in \Delta$ and either (1) $\sigma_{j+1} = a_{i_j}$ and $i_{j+1} = i_j + 1$, or (2) $\sigma_{j+1} \in \{\vdash^x, \dashv^x | \ x \in \mathcal{X}\}$ and $i_{j+1} = i_j$. We say that $\rho$ is *accepting* if $q_m \in F$ and that it is *valid* if variables are non-repeating, and they are opened and closed correctly. If $\rho$ is accepting and valid, we define the mapping $\mu^\rho$ which maps $x \in \mathcal{X}$ to the span $[i_j, i_k\rangle$ iff $\sigma_j = \vdash^x$ and $\sigma_k = \dashv^x$. We say that $\mathcal{A}$ is *sequential* if every accepting run is also valid. Finally, define the document spanner $[\![\mathcal{A}]\!]$ as the function $[\![\mathcal{A}]\!](d) = \{\mu^\rho \mid \rho \text{ is an accepting and valid run of } \mathcal{A} \text{ over } d\}$. Like in AnnAs, we say $\mathcal{A}$ is *unambiguous* if for each mapping $\mu \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run $\rho$ of $\mathcal{T}$ over $d$ such that $\mu^\rho = \mu$.

**Extended VA.** For the sake of presentation, we will skip a formal definition for extended VA, and we refer the reader to [16]. These are automata in which the transitions read either letters in $\Sigma$ or sets of markers from $\{\vdash^x, \dashv^x | \ x \in \mathcal{X}\}$. Runs are defined as sequences which alternate between transitions that read letters and sets, and a mapping associated to a run is defined as one would expect, where $x \in \mathcal{X}$ is mapped to the span $[i, j\rangle$ iff $\vdash^x$ is in the $i$-th set of the run, and $\dashv^x$ is in the $j$-th set in the run. We define sequential and unambiguous extended VA analogously to VA.

To illustrate the reduction from sequential extended VA to annotated automata, consider a document $d = \texttt{aab}$, and a run of some extended VA with variable set $\mathcal{X} = \{x, y\}$ over $d$:

$$\rho = q_0 \xrightarrow{\emptyset} q_0 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\{\vdash^x, \dashv^x, \vdash^y\}} p_1 \xrightarrow{\texttt{a}} q_2 \xrightarrow{\emptyset} q_2 \xrightarrow{\texttt{b}} q_3 \xrightarrow{\{\dashv^y\}} p_3$$

This run defines the mapping $\mu$ which assigns $\mu(x) = [2, 2\rangle$ and $\mu(y) = [2, 4\rangle$. To translate this run to the annotated automata model, first we append an end-of-document character to $d$, and then we "push" the marker sets one transition to the right. We then obtain a run of some annotated automaton with output set $\Omega = 2^{\{\vdash^x, \dashv^x | x \in \mathcal{X}\}}$ over the document $d' = \texttt{aab\#}$:

$$\rho' = q'_0 \xrightarrow{\texttt{a}} q'_1 \xrightarrow{(\texttt{a}, \{\vdash^x, \dashv^x, \vdash^y\})} q'_2 \xrightarrow{\texttt{b}} q'_3 \xrightarrow{(\#, \{\dashv^y\})} q'_4$$

The annotation of this run would then be $\nu = (2, \{\vdash^x, \dashv^x, \vdash^y\})(4, \{\dashv^y\})$, from where the mapping $\mu$ can be extracted directly. The reduction from extended VA into annotated automata operates in a similar fashion: the read transitions are kept, and for each pair of transitions $(p, S, q), (q, a, r)$ in the former, a transition $(p, (a, S), r)$ is added to the latter.

The equivalence between mappings and annotations is formally defined as follows: For some document $d$ of size $n$, a mapping $\mu$ from $\mathcal{X}$ to spans in $d$ is equivalent to an annotation $\nu = (S_1, i_1) \dots (S_m, i_m)$ iff $S_j = \{\vdash^x | \ \mu(x) = [i_j, k\rangle\} \cup \{\dashv^x | \ \mu(x) = [k, i_j\rangle\}$ for every $j \leq m$.

▶ **Proposition 10.** *For any unambiguous sequential extended VA $\mathcal{A}$ with state set $Q$ and transition set $\Delta$, there exists an AnnA $\mathcal{A}'$ of size $\mathcal{O}(|Q| \times |\Delta|)$ such that for every document $d$, each mapping $\mu \in [\![A]\!](d)$ is equivalent to some unique $\nu \in [\![\mathcal{A}']\!](d\#)$ and vice versa.*

Combining Proposition 10 and Theorem 4, we get a constant-delay algorithm for evaluating an unambiguous sequential extended VA over a document, proving the extension of the result in [25]. Notice that the result in [25] is for *deterministic* VA, where here we generalize this result for the unambiguous case plus constant-delay.

**Succinctly annotated automata.** For the next result, we need an extension to annotated automata which features succinct representations of sets of annotations.

A *succinct enumerable representation scheme* (SERS) is a tuple $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$ made of an infinite set of representations $\mathcal{R}$, and an infinite set of annotations $\Omega$. It includes a function $|\cdot|$ that indicates, for each $r \in \mathcal{R}$ and $\text{ø} \in \Omega$, the sizes $|r|$ and $|\text{ø}|$, i.e., the number of units needed to store $r$ and ø in the underlying computational model (e.g. the RAM model). The function $\mathcal{L}$ maps each element $r \in \mathcal{R}$ to some finite non-empty set $\mathcal{L}(r) \subseteq \Omega$. Lastly, there is an algorithm $\mathcal{E}$ which enumerates the set $\mathcal{L}(r)$ with output-linear delay for every $r \in \mathcal{R}$. Intuitively, a SERS provides us with representations to encode sets of annotations. Moreover, there is the promise of the enumeration algorithm $\mathcal{E}$ where we can recover all the annotations with output-linear delay. This representation scheme allows us to generalize the notion of annotated automaton for encoding an extensive set of annotations in the transitions.

Fix a SERS $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$. A Succinctly Annotated Automaton over $\mathcal{S}$ (sAnnA for short) is a tuple $\mathcal{T} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where all sets are defined like in AnnA, except that in $\Delta$ read-write transitions are of the form $(p, (a, r), q) \in Q \times (\Sigma \times \mathcal{R}) \times Q$. That is, transitions are now annotated by a representation $r$ which encodes *sets* of annotations in $\Omega$. For a read-write transition $t = (p, (a, r), q)$, we define its size as $|t| = |r| + 1$ and for a read transition $t = (p, a, q)$ we define its size as $|t| = 1$. A run $\rho$ over a document $d = a_1 \dots a_n$ is also defined as a sequence $\rho = q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$ with the same specifications as in AnnA with the difference that it either holds that $b_i = a_i$, or $b_i = (a_i, r)$ for some representation $r$. We now define the *set of annotations* of $\rho$ as: $\mathsf{ann}(\rho) = \mathsf{ann}(b_1, 1) \cdot \ldots \cdot \mathsf{ann}(b_n, n)$ such that $\mathsf{ann}(b_i, i) = \{(\text{ø}, i) \mid \text{ø} \in \mathcal{L}(r)\}$, if $b_i = (a, r)$, and $\mathsf{ann}(b_i, i) = \{\varepsilon\}$ otherwise. The set $[\![\mathcal{T}]\!](d)$ is defined as the union of sets $\mathsf{ann}(\rho)$ for all accepting runs $\rho$ of $\mathcal{T}$ over $d$. In this model, we say that $\mathcal{T}$ is unambiguous if for every document $d$ and every annotation $\nu \in [\![\mathcal{T}]\!](d)$ there exists only one accepting run $\rho$ of $\mathcal{T}$ over $d$ such that $\nu \in \mathsf{ann}(\rho)$. Finally, we define the size of $\Delta$ as $|\Delta| = \sum_{t \in \Delta} |t|$, and the size of $\mathcal{T}$ as $|\mathcal{T}| = |Q| + |\Delta|$.

This annotated automata extension allows for representing output sets more compactly. Moreover, given that we can enumerate the set of annotations with output-linear delay, we can compose it with Theorem 4 to get an output-linear delay algorithm for the whole set.

▶ **Theorem 11.** *Fix a SERS $\mathcal{S}$. There exists an enumeration algorithm that, given an unambiguous sAnnA $\mathcal{T}$ over $\mathcal{S}$ and an SLP $S$, it runs in $|\mathcal{T}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating $[\![\mathcal{T}]\!](\text{doc}(S))$.*

The purpose of sAnnA is to encode sequential VA succinctly. Indeed, as shown in [16], representing sequential VA with extended VA has an exponential blow-up in the number of variables that cannot be avoided. Therefore, the reduction from Proposition 10 cannot work directly. Instead, we can use a Succinctly Annotated Automaton over some specific SERS to translate every sequential VA into the annotation world efficiently.

▶ **Proposition 12.** *There exists an SERS $\mathcal{S}$ such that for any unambiguous sequential VA $\mathcal{A}$ with state set $Q$ and transition set $\Delta$ there exists a sAnnA $\mathcal{T}$ over $\mathcal{S}$ of size $\mathcal{O}(|Q| \times |\Delta|)$ such that for every document $d$, each mapping $\mu \in [\![\mathcal{A}]\!](d)$ is equivalent to some unique $\nu \in [\![\mathcal{A}]\!](d\#)$ and vice versa. Furthermore, the number of states in $\mathcal{T}$ is in $\mathcal{O}(|Q|)$.*

By Proposition 12 and Theorem 11 we prove the extension of the output-linear delay algorithm for unambiguous sequential VA.

## 6 Constant delay-preserving complex document editing

In this section, we show that the results obtained by Schmid and Schweikardt [26] regarding enumeration over document databases and complex document editing still hold, maintaining the same time bounds in doing these edits, but allowing output-linear delay. We also include a refinement of the result for whenever the edits needed are limited to the concatenation of two documents. To be precise, we will give an overview of the following theorem.

▶ **Theorem 13.** *Let $D = \{d_1, \ldots, d_m\}$ be a document database that is represented by an SLP $S$ in normal form. Let $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be unambiguous sequential variable-set automata. When given the query data structures for $S$ and $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and a CDE-expression $\varphi$ over $D$, we can construct an extension $S'$ of $S$ and new query data structures for $S'$ and $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and a new non-terminal $\tilde{A}$ of $S'$, such that* $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$.

- *If $\varphi$ contains operations other than* concat*, we require $S$ to be strongly balanced. Then, $S'$ is also strongly balanced, and this construction can be done in time $\mathcal{O}(k \cdot |\varphi| \cdot \log |d^*|)$ with data complexity where $|d^*| = |\max_\varphi(D)|$.*
- *If $\varphi$ only contains* concat*, then this can be done in $\mathcal{O}(k \cdot |\varphi|)$ with data-complexity.*

*Afterwards, upon input of any $d \in \mathsf{docs}(S')$ (represented by a non-terminal of $S'$) and any $i \in [1, m]$, the set $[\![\mathcal{A}_i]\!](d)$ can be enumerated with constant-delay.*

Note that a similar result was proved in [26] but with extended VA instead of VA, and with logarithmic delay instead of constant-delay. The rest of this section will be dedicated to define the concepts we have not yet introduced, and show how the techniques presented in [26] allow us to obtain this result.

**Normal form, balanced and rootless SLPs.** We define a *rootless SLP* as a triple $S = (N, \Sigma, R)$, where $N$ is a set of non-terminals, $\Sigma$ is the set of terminals, and $R$ is a set of rules. Rootless SLPs are defined as SLPs with the difference that there is no starting symbol, and thus $\mathrm{doc}(S)$ is not defined. Instead, we define $\mathrm{doc}(A)$ for each $A \in N$ as $\mathrm{doc}(A) = R^*(A)$. We say that $S$ is in *Chomsky normal form* if every rule in $R$ has the form $A \to a$ or $A \to BC$, where $a \in \Sigma$ and $A, B, C \in N$. Also, we say that $S$ is *strongly balanced* if for each rule $A \to BC$, the value $\mathsf{ord}(B) - \mathsf{ord}(C)$ is either -1, 0 or 1, where $\mathsf{ord}(X)$ is the maximum distance from $X$ to any terminal in the derivation tree.

**Document Databases.** A *document database* over $\Sigma$ is a finite collection $D = \{d_1, \ldots, d_m\}$ of documents over $\Sigma$. Document databases are represented by a rootless SLP as follows. For an SLP $S = (N, \Sigma, R)$, let $\mathsf{docs}(S) = \{\mathrm{doc}(A) \mid A \in N\}$ be the set of documents represented by $S$. The rootless SLP $S$ is a representation for a document database $D$ if $D \subseteq \mathsf{docs}(S)$.

For a document database $D$, it is assumed that a rootless SLP $S$ that represents $D$ is in normal form and strongly balanced. It is also assumed that for each nonterminal $A$ for which its rule has the form $A \to BC$, the values $|\mathrm{doc}(A)|$, $\mathsf{ord}(A)$ and nonterminals $B$ and $C$ are accessible in constant time. All these values can be precomputed with a linear-time pass over $S$. We call $S$ along with constant-time access to these values *the basic data structure for* $S$.

**Complex Document Editing.** As in [26], given a document database $D = \{d_1, \ldots, d_m\}$ our goal is to create new documents by a sequence of text-editing operations. Here we introduce the notion of a CDE-expression over $D$, which is defined by the following syntax:

$$\varphi := d_\ell, \ell \in [1, m] \mid \mathsf{concat}(\varphi, \varphi) \mid \mathsf{extract}(\varphi, i, j) \mid \mathsf{delete}(\varphi, i, j) \mid \mathsf{insert}(\varphi, \varphi, k) \mid \mathsf{copy}(\varphi, i, j, k)$$

where the values $i, j$ are valid *positions*, and $k$ is a valid *gap*. The semantics of these operations, called *basic operations*, works as follows:

$$\begin{array}{rclrcl} \mathsf{concat}(d,d') & = & d \cdot d' & \mathsf{insert}(d,d',k) & = & d[1,k\rangle \cdot d' \cdot d[k,|d|+1\rangle \\ \mathsf{extract}(d,i,j) & = & d[i,j+1\rangle & \mathsf{delete}(d,i,j) & = & d[1,i\rangle \cdot d[j+1,|d|+1\rangle \\ \mathsf{copy}(d,i,j,k) & = & \mathsf{insert}(d,d[i,j+1\rangle,k) & & & \end{array}$$

We write $\mathsf{eval}(\varphi)$ for the document obtained by evaluating $\varphi$ on $D$ according to these semantics. For an operation $\mathsf{extract}(\varphi,i,j)$, $\mathsf{delete}(\varphi,i,j)$, $\mathsf{insert}(\varphi,\psi,k)$, or $\mathsf{copy}(\varphi,i,j,k)$, $i,j$ are valid positions if $i,j \in [1,|\mathsf{eval}(\varphi)|]$, and $k$ is a valid gap if $k \in [1,|\mathsf{eval}(\varphi)|+1]$. We define $|\varphi|$ as the number of basic operations in $\varphi$. To adding these new documents in the database we will use the notion of extending a rootless SLP. A rootless SLP $S' = (N', \Sigma, R')$ is called an extension of $S$ if $S'$ is in normal form, $N \subseteq N'$, and $R'(A) = R(A)$ for every $A \in N$. In this context, we call $N' \setminus N$ the *set of new non-terminals*. We define the *maximum intermediate document size* $|\max_\varphi(D)|$ induced by a CDE-expression $\varphi$ on a document database $D$ as the maximum size of $\mathsf{eval}(\psi)$ for any sub-expression $\psi$ of $\varphi$ (i.e., any substring $\psi$ of $\varphi$ that matches the CDE syntax).

Having defined most of the concepts mentioned in Theorem 13, we can introduce the following Theorem in [26], which will be instrumental in the final proof.

▶ **Theorem 14** ([26], Theorem 4.3)**.** *Let $D$ be a document database represented by a strongly balanced rootless SLP $S$ in normal form. When given the basic data structure for $S$ and a CDE-expression $\varphi$ over $D$, we can construct a strongly balanced extension $S'$ of $S$, along with its basic data structure, and a non-terminal $\tilde{A}$ of $S'$ such that* $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$*. This construction takes time $\mathcal{O}(|\varphi| \cdot \log|\max_\varphi(D)|)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}(|\varphi| \cdot \log|\max_\varphi(D)|)$.*

For the second bullet point in Theorem 13, we use the following fact, given without proof:

▶ **Observation 15.** *Let $D$ be a document database that is represented by a rootless SLP $S$ in normal form. Given the basic data structure for $S$ and a CDE-expression $\varphi$ over $D$ which only mentions* concat*, we can construct an extension $S'$ of $S$, along with its basic data structure, and a nonterminal $\tilde{A}$ of $S'$ such that* $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$*. This construction takes time $\mathcal{O}(|\varphi|)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}(|\varphi|)$.*

**The query data structure.** The structure we will use is the one produced in Theorem 11. This structure is built by an algorithm that receives an SLP $S$, an unambiguous sAnnA $\mathcal{T}$, and produces a (succinct) Shift-ECS $\mathcal{D}$ indexed by the matrices $M_A$, for each non-terminal $A$ in $S$. These matrices store nodes $v = M_A[p,q]$ such that $[\![\mathcal{D}]\!](v)$ contains all partial annotations from a path of $\mathcal{T}$ which starts $p$, ends in $q$, and reads the string $\mathrm{doc}(A)$. Note that, although the algorithm receives a "rooted" SLP, it can be adapted quite easily to rootless SLPs by adding a node $v_A$ for each non-terminal $A$ in $S$, built as $v_A = \mathsf{union}_{q \in F}(M_A[q_0,q])$ (the same construction that was done for $S_0$ in the algorithm).

We define *the query data structure for $S$ and $\mathcal{T}$* as the mentioned succinct Shift-ECS along with constant-time access to every index $M_A[p,q]$ for states $p$ and $q$ and non-terminal $A$. Note that for each $A$ it holds that $[\![\mathcal{D}]\!](v_A) = [\![\mathcal{A}]\!](\mathrm{doc}(A))$. In particular, if $S$ represents a document database $D$, then for each $d \in D$ there is a $v$ in $\mathcal{D}$ for which $[\![\mathcal{D}]\!](v) = [\![\mathcal{A}]\!](d)$. Recall that for every node $v \in \mathcal{D}$, the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay.

▶ **Lemma 16.** *Let $S$ be an SLP in normal form and an extension $S'$ of $S$ with new non-terminals $\tilde{N} = N' \setminus N$. Also, let $\mathcal{T}$ be an unambiguous sAnnA and assume we are given the query data structure for $S$ and $\mathcal{A}$, and the basic data structure for $S'$. We can construct the query data structure for $S'$ and $\mathcal{T}$ in $\mathcal{O}(|\mathcal{A}|^3 \cdot |\tilde{N}|)$ time.*

To prove Theorem 13, we first reduce the variable-set automata $\mathcal{A}_1, \dots \mathcal{A}_k$ to sAnnAs $\mathcal{T}_1, \dots, \mathcal{T}_k$ using the construction of Proposition 12. Note, however, that this reduction requires the input document to be modified as well. This can be solved by adding a non-terminal $A_\#$ for each $A \in N$, and a rule $A_\# \to AH$, where $H$ is a new non-terminal with the rule $H \to \#$. Then, in the query data structure for $S$ and $\mathcal{T}$, the nodes $v_A$ are defined over the matrices $M_{A_\#}$ instead. That way, when the user chooses a document $d \in \mathsf{docs}(S)$ and a variable set automata $\mathcal{A}_i$, she can be given the set $[\![\mathcal{T}_i]\!](d\#)$ as output. Note that this has no influence in the time bounds given so far for the edit, except for a factor that is linear in $|\tilde{N}|$.

It can now be seen that the result follows from Theorem 14, Observation 15, and Lemma 16. The fact that for each $d \in \mathsf{docs}(S')$ the set $[\![\mathcal{A}]\!](d)$ can be enumerated with output-linear delay follows from the definition of the query data structure for $S'$ and $\mathcal{A}$.

## 7 Future work

One natural direction for future work is to study which other compression schemes allow output-linear delay enumeration for evaluating annotated automata. To the best of our knowledge, the only model for compressed data in which spanner evaluation has been studied is SLPs. However, other models (such as some based on run-length encoding) allow better compression rates and might be more desirable results in practice.

Regarding the Shift-ECS data structure, it would be interesting to see how further one could extend the data structure while still allowing output-linear delay enumeration. Another aspect worth studying is whether there are enumeration results in other areas that one can improve using Shift-ECS. Lastly, it would be interesting to study whether one can apply fast matrix multiplication techniques to Algorithm 1 to improve the running time to sub-cubic time in the number of states.

## 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, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, volume 80, pages 111:1–111:15, 2017.

3 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. *ACM Trans. Database Syst.*, 46(1):2:1–2:30, 2021.

4 Antoine Amarilli, Louis Jachiet, Martin Muñoz, and Cristian Riveros. Efficient enumeration for annotated grammars. In *PODS*, pages 291–300, 2022.

5 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, pages 167–181, 2006.

6 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

7 Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

8 Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked enumeration of MSO logic on words. In *ICDT*, volume 186, pages 20:1–20:19, 2021.

9 Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a complex event recognition engine. *VLDB*, 15(9):1951–1964, 2022.

10 Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *PODS*, pages 393–409, 2020.

11 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundam. Informaticae*, 111(3):313–337, 2011.

12 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. *Log. Methods Comput. Sci.*, 18(1), 2022.

13 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.

14 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.

15 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.

16 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.

17 Alejandro Grez and Cristian Riveros. Towards streaming evaluation of queries with correlation in complex event processing. In *ICDT*, volume 155, pages 14:1–14:17, 2020.

18 Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. A formal framework for complex event recognition. *ACM Trans. Database Syst.*, 46(4):1–49, 2021.

19 Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Log. Methods Comput. Sci.*, 7(2), 2011.

20 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

21 Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complex. Cryptol.*, 4(2):241–299, 2012.

22 Martin Muñoz and Cristian Riveros. Streaming enumeration on nested documents. In *ICDT*, volume 220, pages 19:1–19:18, 2022.

23 Liat Peterfreund. Grammars for document spanners. In *ICDT*, volume 186, pages 7:1–7:18, 2021.

24 Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. In *CPM*, volume 2373, pages 20–31, 2002.

25 Markus L. Schmid and Nicole Schweikardt. Spanner evaluation over slp-compressed documents. In *PODS*, pages 153–165, 2021.

26 Markus L. Schmid and Nicole Schweikardt. Query evaluation over slp-represented document databases with complex document editing. In *PODS*, pages 79–89, 2022.

27 Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. In *PODS*, pages 151–163, 2018.

28 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.