Enumerating Prefix-Closed Regular Languages with Constant Delay

Duncan Adamson
Leverhulme Research Centre for Functional Materials Design, University of Liverpool, UK

Florin Manea
Department of Computer Science, University of Göttingen, Germany

Paweł Gawrychowski
University of Wrocław

Abstract

In this paper, we propose a novel enumeration algorithm for the set of strings of a given length from a given prefix-closed regular language (i.e., a language accepted by a finite automaton which has final and failure states only). Our algorithm has constant delay in the worst case, after a preprocessing that requires linear time in the size of the deterministic finite automaton accepting the respective language. We extend our results to obtain efficient ranking and unranking algorithms for the enumerated sets of strings.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Formal languages and automata theory

Keywords and phrases Prefix-Closed Regular Languages, Constant-Delay Enumeration Algorithms

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Enumerating all members of a given class of combinatorial objects is one of the most fundamental problems in computer science. At a high level, the goal of enumeration problems is to take a description of the class of objects and produce every object satisfying this description. Often, the number of objects in each class is of exponential size relative to the size of the description: for example, the set of strings of length n over an alphabet Σ of size σ contains σ^n objects. Thus, due to the large size of these classes, the usual goal of enumeration algorithms is to reduce either the delay between outputting consecutive objects or the average time required to output each object. Various instances of enumeration problems appear in very diverse contexts and have a wide range of applications. Comprehensive surveys of enumeration problems and their connections to various areas of computer science and mathematics have been provided by Segoufin [42] (with a focus on logic), Wasa [47] (which provides a list of enumeration problems from multiple areas, ranging from graph theory to computational geometry or to combinatorics on words and automata), Uno [45] (focused on the amortized analysis of enumeration algorithms), as well as Savage [37] and Mütze [31] (both covering a wide variety of combinatorial objects, such as bitstrings, combinations, permutations, partitions). Interesting applications of enumeration algorithms include, among others, database theory [7, 8, 9, 40, 41], combinatorics and algorithms on words and the study of formal languages [1, 2, 9, 19, 23, 43], or bioinformatics [11, 24, 36].

This paper is primarily motivated by the problem of enumerating strings of a given length, which do not contain forbidden factors from a given set. This problem originates from chemistry, with the problem of *crystal structure prediction*. In one dimension, the crystal structure prediction problem asks, given an alphabet of "blocks" (3-dimensional structures), what is the optimal way to arrange these objects to minimise some pairwise objective function [18]. Currently, this problem is solved via heuristic techniques [28, 30, 33, 46], leaving the possibility of missing the optimal solution in numerous instances. At the same time, existing knowledge from chemistry allows certain solutions to be ruled out without the costly process of simulating the predicted structure [32]. In fact, it seems that crystal structure prediction algorithms would benefit significantly from having tools which allow for the complete search of the space of potential structures of a given size, while avoiding known bad combinations. Indeed, as the interaction between blocks in this model is highly localised [4], such bad combinations can be identified and represented as strings of blocks, which then have to be avoided in the construction of longer strings of blocks. Further, as the process of taking a sequence of blocks and determining the value of the objective function is computationally expensive, the length of the sequences is restricted by a given upper bound.

With this motivation in mind, our paper provides new enumeration algorithms for *strings of a given length in prefix-closed regular languages.* A regular language is *prefix-closed* if, given any string s in the language, every prefix of s is also in the language [16, 26]. Equivalently, given a deterministic finite state automaton A (for short, DFA A), A recognises a prefix-closed regular language if and only if every state in A is either a final state or a failure state (i.e., a state from which no final state is accessible). The class of prefix-closed regular languages has nice language theoretic properties [16, 26] and, interestingly w.r.t. our motivation, includes the class of languages of strings that avoid a set of forbidden factors. By focusing on a broader class of languages, our novel results and tools for the enumeration of formal languages could become of interest to a wider community. Going now more into details, in this work, we consider two variants of the problem of enumerating all words from a prefix-closed regular language L.

► **Problem 1.** *Given a DFA A recognising a prefix-closed regular L, and a positive integer n (respectively, two positive integers ℓ and n), enumerate efficiently all strings of length n recognised by A (respectively, all strings of length* at least ℓ and at most n *recognised by A, in increasing order of their length).*

To completely specify this problem, one needs to define exactly what is the output produced in the enumeration. One possibility would be to output each string of length n explicitly, letter by letter; this would inherently lead to $O(n)$-time delay between the output of two consecutive strings (as we first need to finish outputting one string, before starting the next one). To achieve a smaller delay, an implicit representation of the output is needed. However, such a representation has to be meaningful: one should be able to canonically and efficiently retrieve the list of enumerated strings, explicitly written, from the output list of implicitly represented strings. Ideally, an algorithm solving Problem 1 would also permit outputting, on demand, the current word of the enumeration explicitly (at any step of this enumeration). Worth noting, straightforward implicit representations of the enumerated strings (e.g., outputting in step i of the enumeration the implicit description "the i^{th} string of L in lexicographical order") are usually not meaningful, as obtaining the enumerated strings explicitly from such representations would usually require non-trivial work.

Our Contributions. We solve both cases of Problem 1 by providing enumeration algorithms with worst-case constant delay and linear time (in the total size of the DFA A) preprocessing.

Our *first contribution*, essential in achieving these results, is to introduce the notion of *default paths* in the DFA A. More precisely, each state of A is associated with a *default transition*, corresponding to the first edge on the longest path in the automaton, starting with that state. These *default transitions* (or *default edges*) allow the definition of *default paths* in the automaton (those paths consisting only of default edges). In this framework, strings accepted by the automaton can be represented implicitly as the concatenation of multiple default paths (specified by the starting state and their length) and the single letters labelling the non-default edges which connect these paths.

As our *second contribution*, we propose an enumeration algorithm for the strings of length n contained in a prefix-closed language L, which crucially uses the notions of default edges and paths both for the sake of efficiency of enumeration and as basis for the way the enumerated strings are output. The main idea behind this algorithm is to maintain internally, while going through the words we want to enumerate, the implicit representation of the current word, as list of default paths in the automaton accepting the given language and the non-default transitions connecting these paths. By augmenting the graph of default edges with a series of non-trivial data structures (tailored to the rather particular structure of this graph), the implicit representation of the enumerated words, maintained in our algorithm, allows us, on the one hand, to efficiently compute the representation of the next word in the enumeration and present it in a succinct (yet, meaningful, in the sense mentioned above) way, by simply referencing (by length) a prefix of the current word and extending it by appending a non-default transition and a single, succinctly represented default path. On the other hand, this implicit representation also allows us to output explicitly, on demand, at every point during the computation, the current word in linear time w.r.t. its length. In the end, we show that only constant time is needed in our enumeration algorithms to move between consecutive strings and output their representations. Our results are, as such, efficient solutions to Problem 1.

▶ **Result 1** (Theorems 6 and 7). *Let A be a DFA, with m states over an alphabet with σ letters, recognising a prefix-closed regular L, and let n be a positive integer (respectively, let $\ell \leq n$ be two positive integers). We can enumerate all strings of length n (respectively, all strings of length* at least ℓ and at most n, *in increasing order of their length) recognised by A, with $O(1)$-delay, after an $O(m\sigma)$-time preprocessing.*

This result can be immediately applied for languages of strings avoiding a given set of forbidden factors $\mathcal{F}$, and the preprocessing time remains $O(m\sigma)$ for m being the sum of the lengths of the factors in $\mathcal{F}$. An interesting byproduct of our approach is that, in the case when $\mathcal{F}$ consists of a single string f, the preprocessing time can be lowered to $O(|f|)$ (Theorem 8).

Our first two contributions show that a significant class of regular languages can be enumerated with constant delay after a preprocessing phase taking linear time in the total size of the given automaton. As a *third contribution*, we extend our results on the enumeration of strings of given length in prefix-closed regular languages to obtain polynomial time algorithms solving the natural problems of ranking and unranking strings with respect to the order in which they are output by the enumeration algorithm.

▶ **Result 2** (Theorems 9 and 10). *Let A be a DFA with m states, recognising a prefix-closed regular language over the alphabet Σ with σ letters, and let w be a string recognised by A, with $|w| = n$. Further, let $2 \leq \omega \leq 3$ be the exponent for matrix multiplication. We can compute the number of strings of length n output in our enumeration of the language recognised by A before outputting w in $O(m\sigma + n(m^\omega + n + \sigma))$ time. Moreover, let i be an integer. We can compute the i^{th} string of length n output in our enumeration algorithm of the strings recognised by A in $O(m\sigma + n(m^\omega + n + \sigma))$ time.*

Related Work. The enumeration of (regular) languages over some alphabet is a well-studied problem. Wasa lists a series of such enumeration tasks and the complexity of their solutions in [47]. Other interesting surveys of this area are [23, 43].

The most fundamental string enumeration algorithm operates by seeing each string in Σ^n as an n-digit integer in base σ (potentially with leading 0s, for $\Sigma = \{0, 1, \ldots, \sigma - 1\}$). One way to enumerate the strings of Σ^n would hence be to output the base σ representation of each integer between 0 and $\sigma^n - 1$, but this is not particularly efficient. More efficient solutions as well as other closely related fundamental enumeration problems and corresponding algorithms are discussed in the surveys [31, 47]. Several recent works are focused on some more meaningful classes of strings, which also seem closer to applications. In particular, some of these works have provided techniques for enumeration with constant delay. Examples in this direction cover theoretically relevant classes of strings, such as fixed-density binary strings (i.e., strings where the number of occurrences of each symbol is fixed) [14], or cyclic strings [22], but also the practically relevant class of document spanners [8]. On the other hand, several works have focused on more general classes of languages, at the cost of allowing enumeration with logarithmic or linear delay, relative to the length of the strings [1, 2, 29, 40, 41].

Quite close to our work are the results of Ruskey and Sawada from [35], where they give a relatively straightforward algorithm for enumerating the strings of length n, over an alphabet with σ letters, which do not contain a given factor f of length m with constant amortized delay, after a preprocessing taking $O(m\sigma)$ time. Although also starting from the construction of the DFA accepting all words which do not contain f, our solution outperforms the one from [35] by reducing the preprocessing time to $O(m)$, and the delay to constant in the worst-case. It is also worth saying that the authors of [35] do not describe the way the enumerated strings are output, and this could potentially lead to additional delays, as discussed previously.

The recent work by Amarilli and Monet [10] is of particular interest to this paper. In [10], the authors look at the enumeration of general regular languages with bounded delay. This is achieved by partitioning the input language into *orderable regular languages*, i.e., a language whose words can be ordered (in a potentially infinite sequence) in such a way that the edit distance between the i^{th} and the $i+1^{th}$ members of the sequence is bounded. The authors show that regular languages can be partitioned into a finite number of orderable languages. Further, for any orderable language L, an algorithm for enumerating the set of strings of that language with bounded delay (where the bound is given by an exponential function in the size of the DFA accepting L) is provided. A nice feature of the algorithm from [10] is that the output is given implicitly, but meaningfully (again, in the sense discussed above), as short edit scripts between consecutive strings; an internal explicit representation of the current string is maintained, and can be output on demand. Our enumeration algorithms go in a similar direction: we output a succinct representation of the way in which two consecutive strings differ (although this representation is based on completely other notions than the edit distance) and we maintain internally a representation of the current string, from which it can be explicitly output. There are however some important differences between our enumeration algorithm and the one from [10]. Firstly, not all prefix-closed regular languages are orderable (e.g., the language defined by the regular expression $a^* \cup b^*$ is not orderable), so our results address a different (incomparable) class of languages. Secondly, the delay of our algorithm does not depend on the size of the input DFA. Thirdly, the algorithm of [10] cannot be adapted to solve ranking and unrakning problems, nor to enumerate the strings in increasing order of their length.

A large amount of literature has focused on providing constant delay enumeration algorithms for classes of cyclic strings. The connections between cyclic strings and crystal structure prediction are well established [3, 4, 5], making this a particularly relevant class of strings. The general class of fixed-length cyclic strings, also known as necklaces, were first enumerated by Fredricksen and Maiorana [22] for the purpose of outputting de Bruijn sequences. This algorithm was later proven to run in linear time by Ruskey et al. [34]. A constant amortised delay algorithm for the enumeration of both cyclic strings and aperiodic cyclic strings was more recently provided by Cattell et al. [15]. Constant amortised delay enumeration algorithms have since been provided for a large number of classes of cyclic words including bracelets [38], fixed-content necklaces [39] (cyclic words where the number of occurrences of every symbol over an arbitrary alphabet is fixed), and necklaces or bracelets with a forbidden factor [35, 38]. It should be noted that not all these works discuss thoroughly the way the enumerated objects are output. As such, and taking into account that our work is partly motivated by crystal structure prediction, a natural extension of this paper would be to see how our results and techniques can be adapted to the setting of circular strings.

2 Preliminaries

Let $\mathbb{N} = \{1, 2, \ldots\}$ be the set of strictly positive integers and let $[n] = \{1, \ldots, n\}$ for $n \in \mathbb{N}$. Let $\Sigma = [\sigma]$ be an alphabet with σ letters, which is strictly ordered (i.e., $1 < 2 < \ldots < \sigma$). By Σ^+ we denote the set of non-empty words (strings) over Σ and $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ (where ε is the empty word). For a string $w \in \Sigma^*$, we denote by $|w|$ its length (with $|\,\varepsilon\,| = 0$) and by Σ^n the set of strings of length n over Σ; by $\Sigma^{\leq n}$ we denote the words of length at most n over Σ. The notation $w[i]$ is used to denote the symbol at position i of w. A subset L of Σ^* is called language. For a finite language $\mathcal{F}$, let $||\mathcal{F}|| = \sum_{w\in\mathcal{F}} |w|$ the total length of the words in $\mathcal{F}$.

Let $n, m \in \mathbb{N}$ be a pair of positive integers such that $n \geq m$. The string $u \in \Sigma^m$ is a *factor* of $w \in \Sigma^n$ if and only if there exists an index $i \in [n - m]$ such that $w[i]w[i+1]\ldots w[i+m-1] = u$; in that case, we denote $u = w[i, i+m-1]$. A factor $w[i, j]$ of $w \in \Sigma^n$ is a prefix (respectively, a suffix) of w if $i = 1$ (respectively, $j = n$).

Let $G = (V, U)$ be a directed graph with the set of vertices V and the set of edges $U \subseteq V \times V$. The directed graph G is labelled, with labels over an alphabet Σ, if there exists a function $\mathcal{L} : U \to \Sigma$ which labels each edge of G with a letter from Σ. A path of length k in G is a sequence of vertices $p = (v_1, v_2, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in U$, for all $i \in [k-1]$; if G is labelled, then the label of the path p is the word $\mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3)\cdots\mathcal{L}(v_{k-1}, v_k)$.

A deterministic finite automaton (DFA) is a construct $A = (Q, \Sigma, q_0, F, \delta)$, where Q is a finite set of states, Σ is the input alphabet, q_0 is the initial state, F is the set of final states, and $\delta : Q \times \Sigma \to Q$ is the transition function. A DFA A can be seen canonically as a labelled directed graph (also denoted A, by an abuse of notation) with the set of vertices Q and having an edge (q_1, q_2) if and only if there exists a symbol $a \in \Sigma$ such that $\delta(q_1, a) = q_2$; in this case, the edge (q_1, q_2) has the label a. Using this interpretation of DFAs as graphs, the language accepted by the DFA A, denoted $L(A)$, is the set of labels of paths between q_0 and final states. The class of languages accepted by DFAs is exactly the class of regular languages. Note that viewing DFAs as graphs allows for a very intuitive way of treating words like paths in the DFA/graph A and, vice versa, it also allows treating paths in the respective graph as words.

A state $q \in Q$ of a DFA $A = (Q, \Sigma, q_0, F, \delta)$ is *co-accessible* if there exists a path starting in q which leads to a final state. As the paths going through non-co-accessible states cannot lead to a final state, such paths cannot correspond to accepted words; thus, the non-co-accessible states are called also failure states. One can detect the failure states of an automaton in linear time w.r.t. the size of the automaton. If these failure states (and the edges incident to such a state) are eliminated from A, we obtain an *incomplete* DFA (meaning that the transition function becomes partial). Moreover, an incomplete DFA $A = (Q, \Sigma, q_0, F, \delta)$ can be cannonically completed by adding a single failure state q and setting all the undefined transitions $\delta(r, a) \leftarrow q$, for $r \in Q$ and $a \in \Sigma$, as well as $\delta(q, a) \leftarrow q$, for all $a \in \Sigma$.

For more details on DFAs and the languages accepted by them, see [25].

Before introducing the notions which are in the main focus of this paper, we note that the computational model we use is the RAM with logarithmic word size (see Appendix A).

In this paper, we are interested in *prefix-closed regular languages* (denoted, for short, PCL-languages). A regular language L is prefix-closed if $s[1,i] \in L$, for any string $s \in L$ and $i \in [|s|]$; that is, all prefixes of the strings in L are also in L. As an alternative characterization, a regular language L, recognised by the complete DFA $A = (Q, \Sigma, q_0, F, \delta)$, is a PCL if and only if every state in Q is either a final state or a failure state. Alternatively, a regular language L, recognised by the incomplete DFA $A = (Q, \Sigma, q_0, F, \delta)$, is a PCL if and only if every state in Q is final. For the rest of this paper, we assume that a PCL-language L is always given as an incomplete DFA $A = (Q, \Sigma, q_0, Q, \delta)$ with all states final, called *prefix-closed finite automata* (PCA, for short). Clearly, all the paths in the graph corresponding to the PCA A go through final states only, and those paths starting in the initial state of a PCA correspond one-to-one to words of the language accepted by A. This property is fundamental for our algorithms: enumerating the words of length n accepted by a PCA A is equivalent to enumerating the labelled paths of length n in the graph corresponding to the automaton A.

An interesting example of PCL is the following: Let $\mathcal{F}$ be a finite set of words over an alphabet Σ, called *forbidden strings*. The language $L_{\mathcal{F}}$ of the words over Σ that do not contain any forbidden string (i.e., word of $\mathcal{F}$) as factor is a PCL-language. We can efficiently construct a PCA accepting it, based on the standard Aho-Corasick automaton recognising all words ending with elements of $\mathcal{F}$ (see Appendix B).

▶ **Lemma 1.** *Let $\mathcal{F} \subseteq \Sigma^*$ be a finite set of forbidden strings. A PCA accepting $L_{\mathcal{F}}$ can be built in $O(||\mathcal{F}|||\Sigma|)$ time.*

In the case of a single forbidden string (i.e., $\mathcal{F} = \{f\})$), a succinct representation of the PCA accepting $L_{\mathcal{F}}$ can be constructed more efficiently, in $O(|f|)$ time (see Appendix B).

For completeness, let us note here that there are PCLs which cannot be defined as $L_{\mathcal{F}}$ for some finite set $\mathcal{F}$. Indeed, the set of words $L = \{w \mid w$ is a prefix of the infinite word $(abc)^\infty\}$ is clearly a PCL, but it cannot be defined as $L_{\mathcal{F}}$ for some set $\mathcal{F}$, as any language $L_{\mathcal{F}}$ is also, e.g., suffix closed, and this is not the case of L.

3 Data Structures and Automata

Setup. For the rest of this section we consider the PCA $A = (Q, \Sigma, q_0, Q, \delta)$, with $|Q| = m$ and $\Sigma = \{1, 2, \ldots, \sigma\}$, and introduce the main data structures that are used in the enumeration algorithms provided in Section 4. We also show several useful technical lemmas. Full proofs of the technical lemmas not provided here can be found in Appendix C.

For ease of presentation, we see A as a labelled directed graph, as described in the previous section; so, the states of A are also called nodes and its transitions edges, without any danger of confusion. In this setting, we assume that for each node $q \in Q$ we store a list of all outgoing and incoming edges; these can be easily computed in $O(m\sigma)$ time.

An initial simple observation is that the longest path starting in a state q of A either has a length of at most m or has infinite length. As such, we want to compute, for each state q, the length $\pi(q) \in [m] \cup \{\infty\}$ of the longest path in A, starting with state q. Moreover, for each state q, we also want to compute and store an ordered list L_q of pairs $(a, \ell) \in \Sigma \times ([m] \cup \{\infty\})$ such that $\delta(q, a)$ is defined in A and ℓ is the length of the longest path in A starting with the edge $(q, \delta(q, a))$. The list L_q is ordered decreasingly according to the second component of the pairs stored in it (i.e., according to the lengths of the paths corresponding to the respective pairs) and the ties are decided according to the order on the letters of Σ (i.e., (a, ℓ) comes before (b, ℓ) if $a < b$). The next lemma follows (see Appendix C).

▶ **Lemma 2.** *Given the PCA A, the lists L_q, for all $q \in Q$, can be computed in $O(m\sigma)$ time.*

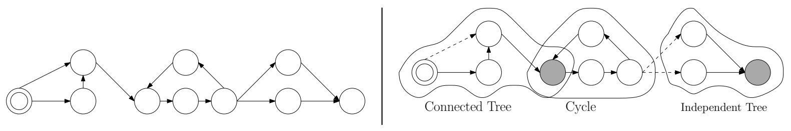

Figure 1 Overview of the different classes of default components. On the left we have a PCA (the labels of the transitions are omitted); on the right we highlighted the default paths and components of that PCA: default edges are shown as solid lines, non-default edges are dashed. From left to right, we have a tree that is connected to a cycle, a cycle, and an independent tree. The tree-roots are grey.

Using the lists L_q computed by Lemma 2, one can retrieve in $O(1)$ time the first edge $(q, \delta(q, a))$ on the longest path leaving a state q. It is indicated by (ℓ, a), the first element of L_q. Similarly, one can obtain the first edge $(q, \delta(q, b))$, with $b \neq a$, on the longest path leaving q by any edge other than $(q, \delta(q, a))$. It is given by (ℓ', b), the second element of L_q.

Default edges, default paths: definitions and basic properties. Using Lemma 2, a directed graph $D(A)$ of so-called *default edges* for the automaton A can be built. The directed graph $D(A)$ has exactly the same nodes as A and, for each node $q \in A$, a single edge e_q leaving node q, called *default edge*; the edge e_q is defined as the edge $(q, \delta(q, a))$ of A, where (ℓ, a) is the first element of L_q. That is, the default edge of q is the edge leaving q on the longest path starting from q. A path in $D(A)$ is called *default path* and $D(A)$ *default graph of* A.

Note that the out-degree of the nodes of the graph $D(A)$ (that is, the number of edges leaving each node of the graph) is at most 1. Therefore, the default graph consists of a collection of *default components*: disjoint cycles, trees whose roots are on cycles, and independent trees, which are not connected to any cycle; in the trees of this collection, the orientation of the edges is induced by the orientation of the edges in A, i.e, from children to parents, so from the leaves towards the root. For an example, see Figure 1.

The graph of default edges is fundamental for the way we present the output of our enumeration algorithm, described in Section 4. The main idea is that a pair (q, ℓ), formed by a node q in $D(A)$ (i.e., state of A) and a length $0 \leq \ell \leq \pi(q)$, uniquely defines a path of length ℓ of default edges in $D(A)$, whose labels form a word of length ℓ. This allows us to represent succinctly, by pairs (q, ℓ), strings of arbitrary length (including strings of length 0, by pairs $(q, 0)$) simply by referencing $D(A)$, as long as those strings are labels of paths of $D(A)$. Moreover, this forms a basis for the way we will represent all strings which are labels of paths in A: these can be presented as a concatenation of default paths from $D(A)$ (so pairs (q, ℓ)), connected between them by single letters (corresponding to non-default edges which connect the end of a default path to the start of another such default path).

Indeed, a word w of length n can be represented as follows. We take $w[1, n_1]$ to be the longest prefix of w which is the label of a default path starting with q_0 in $D(A)$, and let q_0' be the ending node/state of this path and $q_1 = \delta(q_0', w[n_1 + 1])$. The representation of w starts, therefore, with $(q_0, n_1)w[n_1 + 1]$. Then we take $w[n_1 + 2, n_2]$ to be the longest prefix of $w[n_1 + 2, n]$ which is the label of a default path starting with q_1 in $D(A)$, and let q_1' be the ending node/state of this path and $q_2 = \delta(q, w[n_2 + 1])$. We append to the representation of w the default path $(q_1, n_2 - n_1)$ and the letter $w[n_2 + 1]$. And we continue then in the same way from q_2 until we have traversed the entire word w.

Efficient algorithms and data structures for the default graph. To work efficiently with these representations, we need to be able to process efficiently the default graph. Thus, we need a series of lemmas, tools, and data structures, providing a more profound understanding and better structuring of this graph. We begin with the following result, which shows that the structure of $D(A)$ can be efficiently computed. A full proof can be found in Appendix C.

▶ **Lemma 3.** *Given the default graph $D(A)$, we can compute in $O(m)$ time all its default components, and store for each node q in $D(A)$ the component in which it is contained.*

In general, one can represent the trees appearing as default components of $D(A)$ (independent or attached to a cycle) as the root, and then a list of children for each node; additionally, the edges of the automaton give us the parent of each node in a tree. Each cycle α is represented by its length $|\alpha|$ and an array with $2|\alpha|$ elements. For each cycle α, we have also chosen an initial node r, which is the first element in the array corresponding to that cycle, and then we go twice around α, while writing in the array its nodes of α in the order we meet them in this traversal. So, each element of α appears exactly twice in the array associated with the cycle, with exactly $|\alpha|-1$ positions between its two occurrences. For each node q of α, it is enough to store its first occurrence i_q in the corresponding array.

We have noted that our representation of paths in A (or, alternatively, words accepted by the DFA A) relies on the usage of pairs (q,ℓ) denoting the default path of length ℓ starting in a node q. In the following, we provide a deeper analysis of these paths.

If q is in a cycle α of length $|\alpha|$, the default path (q,ℓ) simply goes around the cycle, over ℓ edges. The ending node of the path is easy to retrieve: if q is the i^{th} node on the cycle (the initial node being the first), then the ending node of the default path (q,ℓ) is the node found on the $((i+\ell) \mod |\alpha|)^{th}$ position of the cycle.

If q is in an independent tree of root r, the default path (q,ℓ) goes towards the root of the tree, traversing ℓ edges. In other words, the path goes through ℓ levels of the tree towards the root; recall that the levels of the tree are defined as follows: the root is on level 0, and the children of a node on level i are on level $i+1$. This means, that, if q is on level h in its tree, the ending node of the default path (q,ℓ) is the ancestor of q in that tree on level $h-\ell$. Thus, to be able to retrieve the ending nodes of default paths in trees quickly, we will build for all our trees (independent and attached to cycles) *level ancestor* data structures [13]. For a tree of size τ, these data structures can be computed in $O(\tau)$ time and enable us to answer in $O(1)$ queries $LA(q,j)$: return the ancestor of q which is on level j of the tree.

If q is in a tree whose root r is on a cycle, the path (q,ℓ) goes towards the root of the tree and potentially also goes around the cycle, traversing ℓ edges in total. Taking into account the ideas already described above, to retrieve the ending node of a default path (q,ℓ) in a tree of root r attached to a cycle, we will first check if the path ends inside the tree or enters the cycle. This can be simply done by verifying if the level h of q inside its tree is greater or equal to ℓ. If yes, we can compute again the level ancestor of q, which is on level $h-\ell$. If not, then the ending node of the path (q,ℓ) is on the cycle. The number of edges traversed in the cycle by this path is $\ell-h$, and to find the ending node of (q,ℓ) it is enough to find the ending node of the default path $(r,\ell-h)$, which is a path on a cycle, and can be treated as above.

Following this discussion, it seems important to store, for the nodes of the tree-components of $D(A)$, their level. To allow a uniform treatment of all the nodes, we define the *depth of node q in its default component*, denoted d^t_q, if q is in a tree, or, respectively, d^c_q, if q is in a cycle. This is defined in one of three ways, depending on which kind of component contains q. Before giving the definition, we note that the nodes which are contained in trees whose root is on a cycle will have two such depths, one w.r.t. the tree and one w.r.t. the cycle.

- If q is in a tree, then d^t_q is simply the level of q in the respective tree.
- For a cycle α, we define the depth of the positions of the array associated to α: for $i \leq 2|\alpha|$, we defined $d^c_i = 2m - i + 1$. If q is a node on the cycle α, we define $d^c_q = d^c_{i_q}$, where i_q is the first (i.e, leftmost) occurrence of q in the array associated to α.

- If q is in a tree of root r connected to a cycle α, then we associate to q a second value d_q^c (the depth of q w.r.t. the cycle α) which is defined as $d_q^c = d_q^t + d_r^c$.

The algorithm presented in Section 4 enumerates all paths of length n in A starting in the node q_0, where n is given as input. An important part of this algorithm, allowing us to move in our enumeration from one path π_1 to the next one π_2, is to check whether there exists a node q on some default path (s, ℓ) (which is part of the representation of the first path π_1) from which we can follow a non-default edge, instead of the default edge we have followed in the path π_1, and obtain, in this way, a new path starting in q_0 of length n. Such a node q is called, for simplicity of exposure, *branching node* with respect to the path π_1 (notice, though, that some nodes might be branching w.r.t. some paths π_1 and not branching w.r.t. others, depending on the position in which they appear on these paths; however, we will only use this name when there is no danger of confusion). Thus, we need to check the existence of a node q on (s, ℓ), such that, if the path from q_0 to such a state q along π_1 has length ℓ_q, then there is a path starting in q with a non-default edge, and has length at least $n - \ell_q$.

To achieve this, we should find the node q for which the sum between the length of the path from q_0 to q along π_1 and the length of the longest path starting in q with a non-default transition is maximum, and see if this sum is greater or equal to n. If this sum is not greater or equal to n, then we know that there is no node with the desired properties on our path (s, ℓ). If the sum is greater or equal to n, the node q we found has the desired properties, and we can use it next in our enumeration. Now, to actually identify this node q, we note that q is exactly that node for which the sum of the length of the path from s to q and the length of the longest path starting in q with a non-default transition is maximum (as the path from q_0 to any node on the default path (s, ℓ), along π_1, goes through s, the starting node of that default path (s, ℓ)).

In this context, we define for each node q a weight w_q, corresponding to the length of the longest path from q which starts with a non-default transition; as explained before, this is precisely the path starting with the second transition stored in L_q, and then continuing with the longest default path starting with the target-node of that edge. After running the preprocessing of Lemma 2, the value w_q can be retrieved in $O(1)$ time, for each q.

Let PathMaxNode(s, ℓ) denote, for each default path (s, ℓ) of $D(A)$, the pair (q, d) where q is the node of this path such that the sum of w_q and the distance d between s and q along the default path (s, ℓ) is maximum. In Lemma 4 (proven in Appendix C), we show that PathMaxNode queries can be answered in $O(1)$-time, after linear time preprocessing. Indeed, building on the data structures introduced above and taking into account the particular structure of the default graph, these queries reduce to either computing path minimum queries in trees, which can be handled efficiently [21], or to range minimum queries in arrays corresponding to the cycles, which again can be computed efficiently [12].

▶ **Lemma 4.** *We can build in $O(m)$ time data structures, allowing us to retrieve the node* PathMaxNode(s, ℓ) *in $O(1)$ time, for each default path (s, ℓ) of $D(A)$.*

4 Enumeration

Based on the data structures introduced in Section 3, in particular the notion of default edges, paths, and components, we present here our approach for enumerating the strings of length n of $L(A)$, for a PCA $A = (Q, \Sigma, q_0, Q, \delta)$ with $|Q| = m$ and $|\Sigma| = \sigma$.

The main algorithm. The basic idea of our enumeration algorithm is to represent and compute the words of $L(A) \cap \Sigma^n$ (or, equivalently, the paths of length n in A, starting with q_0) as sequences of default paths and the single non-default transitions connecting them.

This is achieved using a recursive procedure, Enumerate. In each call, Enumerate takes three parameters: a letter $a \in \Sigma \cup \{\uparrow\}$, a state q, and a number ℓ. In the call Enumerate(a, q, ℓ) we go through all paths of length ℓ starting from q; each such path is the suffix of a path of length n starting in q_0, going through q, which will be output.

Before making any recursive call, we preprocess the PCA A as described in Section 3: we build the default components and augment them with the data structures facilitating their efficient processing. In particular, we construct the data structures needed to answer in $O(1)$-time PathMaxNode-queries. This takes $O(m\sigma)$ time (and does not depend on n).

During the actual enumeration, done via calls to the recursive procedure Enumerate, we maintain as global variables two stacks $\mathcal{C}$ and $\mathcal{S}$, containing tuples (a, q, ℓ), with $a \in \Sigma, q \in Q, \ell \in [n]$. Intuitively, if the content of the stack $\mathcal{S}$ is, at some step of the computation the sequence $\langle(\uparrow, q_0, \ell_0), (a_0, q_1, \ell_1), \ldots, (a_{t-1}, q_t, \ell_t)\rangle$ (where the top of the stack is to the right of this sequence), then the currently enumerated string is $w_0 a_0 w_1 a_1 \cdots w_{t-1} a_{t-1} w_t$, where, for $i \geq 0$, w_i is the label of the default path of length ℓ_i starting with the state q_i and ending with some state q'_i, and $q_{i+1} = \delta(q'_i, a_i)$, for $0 \leq i \leq t-1$. Clearly, the string w can be retrieved explicitly from its representation on the stack $\mathcal{S}$ in linear time. With respect to the execution of our algorithm, $\mathcal{S}$ corresponds to the stack of currently active recursive calls. The usage of $\mathcal{C}$ is more subtle. Intuitively, at every step of the computation, $\mathcal{C}$ contains (bottom to top, in the same order as in $\mathcal{S}$) exactly those triples (a, q, ℓ) of $\mathcal{S}$ for which the default path of length ℓ starting in q still contains branching nodes leading to paths of length n which were not enumerated yet. As such, $\mathcal{C}$ facilitates the quick identification of the next path which we need to output in our enumeration: such a path should go through one of the branching points of the default path found on top of this stack $\mathcal{C}$. From the point of view of the execution of our algorithm, $\mathcal{C}$ corresponds to the currently active recursive calls which were not tail calls. Very importantly, we assume that in the computational model we use, tail calls are implemented so that no new stack frame is added to the call-stack. Hence, $\mathcal{C}$ actually corresponds, at each moment of our algorithm's execution, to the current call-stack.

The enumeration starts with the call Enumerate$(\uparrow, q_0, n)$, which outputs first the tuple $(-1, \uparrow, q_0, n)$, corresponding to the default path starting in q_0 and having length n. It is important to note that this first object in our enumeration is output in $O(1)$, after a preprocessing whose time-complexity does not depend on n. In a general step, we call Enumerate(a, q, ℓ). The pseudocode of this procedure is given in Algorithm 1 in Appendix D; a detailed description of its steps is given in Appendix E. We give in the following only a high-level, simplified description of our algorithm.

The first thing we do in the call Enumerate(a, q, ℓ) is output the tuple $(n - \ell - 1, a, q, \ell)$. That is, the current string in the enumeration, denoted by v in the following, is obtained from the previous one by keeping the prefix of length $n - \ell - 1$ (which ends in some state q'), to which we append the letter a, corresponding to the transition labelled with a from q' to q, and the default path of length ℓ starting in q. We also store the triple (a, q, ℓ) in the two stacks.

Next, we need to see if there are more paths of length ℓ starting with q, other than the default path. If no, we simply remove the triple (a, q, ℓ) from both stacks and return. If yes, we need to discover each of them, and this will be done by further recursive calls to Enumerate; all the words produced in this way will share a prefix of length at least $n - \ell$ with v. The efficient identification of these paths relies on the data structures constructed in the preprocessing phase for the default components of A. However, to see how exactly they are identified, we need to understand what we are looking for. Basically, each such path follows a (potentially empty) prefix of the default path of length ℓ starting with q, which ends with a branching node w.r.t. the path labelled by v starting in q_0. So, we only need to discover the respective branching nodes, and process them one by one, by recursive calls. However, we can find the branching nodes by PathMaxNode-queries and, by Lemma 4, such queries can be answered in $O(1)$-time due to the good structure of the default graph (composed of trees and cycles).

This is done as follows. Let r be the ending node of the default path (q, ℓ). We first retrieve $(p, d) = \text{PathMaxNode}(q, \ell)$. If $d + w_p \geq \ell$, then this node is one of the branching nodes we were looking for. Indeed, the above inequality means that the path starting from p with the transition given by the second element of L_p is long enough (at least as long as the path from p to r along the default path (q, ℓ)). Thus, we can go through the non-default transitions leaving p (as given by L_p), as long as they start paths which are at least as long as the default path from p to r, and start a corresponding recursive call of Enumerate for each of them. Once we are done with all these transitions, and, accordingly, with the node p, we might still have other branching nodes on the default path (q, ℓ). Such a node is either on the segment of this path found between q and the predecessor of p or on the segment of this path found between the successor of p and r, and they can be identified using PathMaxNode-queries, just as above. In general, if the PathMaxNode-query on such a segment of the default path (q, ℓ) (delimited by nodes q_1 and q_2) returns a node p' from which we can follow a long-enough path starting with a non-default transition, we process p' exactly as we processed node p above, and then search for more branching nodes in the two segments delimited, respectively, by q_1 and the predecessor of p' and the successor of p' and q_2. If the PathMaxNode-query on a segment of the default path (q, ℓ) returns a node p' from which no long-enough path starting with a non-default transition can be followed, we simply stop processing that entire segment: there are no more relevant branching nodes to be found there.

According to the above, we can guarantee that we only call Enumerate exactly when we are certain that it will lead to the discovery of at least one path which was not yet enumerated. Also, the total number of PathMaxNode-queries which are answered during the call $\text{Enumerate}(a, q, \ell)$ is proportional to the number of branching nodes found on the default path (q, ℓ). So, the overall time complexity of the execution of $\text{Enumerate}(a, q, \ell)$ is proportional to the number of all paths starting from q and having length ℓ (each of them corresponding to a path of length n we need to enumerate).

To obtain enumeration with constant delay in the worst case, there is one more additional subtlety. Namely, we need to make sure that as soon as a call to Enumerate is finished, we will directly consider a default path on which we certainly find branching nodes. This is ensured by the usage of tail calls and of the stack $\mathcal{C}$. Indeed, we need to make sure that the last step of the procedure Enumerate is a recursive call (and this is not hard to do); before executing this call, we remove the triple (a, q, ℓ) from $\mathcal{C}$ (as it will also be removed from the call-stack). Thus, in our computational model, after the respective tail recursive call is finished, we will return to a previous call which still has at least one path to explore (so, one branching node that was not completely processed); the respective previous call corresponds to the triple found now on top of $\mathcal{C}$, and, as such, we can check where the respective triple is found in $\mathcal{S}$ and set it to be the current top of the respective stack too (what came above it is no longer important for the rest of the computation, as it does not lead to new paths).

To show the correctness of our approach (whose details are given in Appendix E), one can show, by induction, that there is a bijection between the words $w \in L(A)$ and the path from the root to leaves in the tree of recursive calls whose root is $\text{Enumerate}(\uparrow, q_0, n)$ (in this tree, the nodes are instances of Enumerate called in our algorithm, and their children are the calls initiated in the respective instances). The full proof can also be found in Appendix E.

▶ **Lemma 5.** *The call* $\text{Enumerate}(\uparrow, q_0, n)$ *outputs a representation of every string of length* n *accepted by the PCA* A *exactly once.*

Building on the result of Lemma 5, which shows the correctness of our approach, the following theorem states the main result of this paper.

▶ **Theorem 6.** *Given a number n and a PCA A, with m states and input alphabet of size σ, we can enumerate, without repetitions, the strings of length n recognised by A with worst-case delay of $O(1)$, after a preprocessing taking $O(m\sigma)$ time.*

Variants and Generalizations. The following result follows immediately (see Appendix F).

▶ **Theorem 7.** *Given two numbers $\ell \leq n$ and a PCA A, with m states and input alphabet of size σ, we can enumerate, without repetitions, the strings of length at least ℓ and at most n recognised by A, in increasing order of their length, with worst-case delay of $O(1)$, after a preprocessing taking $O(m\sigma)$ time.*

The result in Theorem 7 can be extended to show that, after a preprocessing taking $O(m\sigma)$ time, we can enumerate, without repetitions, in increasing order of their length, the strings recognised by the PCA A with delay $O(\log i/\mathtt{w})$ between consecutive strings of length i as well as between the last string of length i and the first of length $i+1$, for $i \geq 1$. In this statement, we denote by $\mathtt{w}$ the size of the memory word used by the RAM on which our algorithm works.

Among PCLs, the class of languages $L_{\mathcal{F}}$, of the words over Σ that do not contain any forbidden factor from a finite set of words $\mathcal{F}$, is of particular interest. It remains open whether the results of Theorem 6 and 7 can be improved for such languages. However, when $\mathcal{F} = \{f\}$ the following holds (according to the construction showed in Appendix B).

▶ **Theorem 8.** *Given a number n and a word f of length m, over an alphabet Σ, we can enumerate, without repetitions, the strings of length n over Σ which do not contain f as factor, with a worst-case delay of $O(1)$, after a preprocessing taking $O(m)$ time.*

The final problems this paper considers are those of *ranking* and *unranking* strings in prefix-closed regular languages. The *rank* of a string w in a language is the number of strings smaller than w in the language under some ordering. The ranking problem requires computing the rank of a given word w. The unranking operation (problem) is the reverse of the ranking operation (problem), taking a number i as the input and asking for the string of rank i. In both cases, we refer to the ordering induced by the enumeration algorithm of Section 4, and we want to show that these two problems can be solved in polynomial time.

At a high level, referring to our main enumeration algorithm, both ranking and unranking require identifying a path in the tree of recursive calls with root Enumerate$(\uparrow, q_0, n)$. For ranking, we identify the path corresponding to w, and the branching nodes occurring on it, and then count the total number of paths of length n corresponding to the leaves of subtrees of recursive calls occurring to the left of this path (assuming that the recursive calls made by an instance are ordered in the tree left to right according to their call-order). This can be done by running Enumerate$(\uparrow, q_0, n)$ and simply performing only the recursive calls that correspond to branching nodes on the path labelled with w, and retrieving the number of induced paths for those that should have been called before them. For unranking, we run again Enumerate$(\uparrow, q_0, n)$ and we make only those recursive calls which lead to the i^{th} path of length n, in the order of our enumeration. The following two results are obtained (Appendix G).

▶ **Theorem 9.** *Given a PCA A and a string $w \in \Sigma^n \cap L(A)$, we can compute the number of strings that are recognised by A and are output before w in our enumeration algorithm in $O(m\sigma + n(m^\omega + n + \sigma))$ time, where $2 \leq \omega \leq 3$ is the exponent for matrix multiplication.*

▶ **Theorem 10.** *Given an integer i, a PCA A, and an integer n, the i^{th} string w of length n output by enumerating A can be determined in $O(m\sigma + n(m^\omega + n + \sigma))$.*

References

1 M. Ackerman and E. Mäkinen. Three new algorithms for regular language enumeration. In *Computing and Combinatorics: 15th Annual International Conference, COCOON 2009 Niagara Falls, NY, USA, July 13-15, 2009 Proceedings 15*, pages 178–191. Springer, 2009.

2 M. Ackerman and J. Shallit. Efficient enumeration of words in regular languages. *Theoretical Computer Science*, 410(37):3461–3470, 2009.

3 D. Adamson. Ranking binary unlabelled necklaces in polynomial time. In *Descriptional Complexity of Formal Systems - 24th IFIP WG 1.02 International Conference, DCFS 2022, Proceedings*, volume 13439 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2022.

4 D. Adamson, A. Deligkas, V. V. Gusev, and I. Potapov. The k-centre problem for classes of cyclic words. In *SOFSEM 2023: Theory and Practice of Computer Science - 48th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2023, Proceedings*, volume 13878 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2023.

5 Duncan Adamson, Vladimir V. Gusev, Igor Potapov, and Argyrios Deligkas. Ranking bracelets in polynomial time. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, volume 191 of *LIPIcs*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

6 A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

7 A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 89–103, 2019.

8 A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. *ACM Transactions on Database Systems (TODS)*, 46(1):1–30, 2021.

9 A. Amarilli, L. Jachiet, M. Muñoz, and C. Riveros. Efficient enumeration for annotated grammars. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 291–300, 2022.

10 Antoine Amarilli and Mikaël Monet. Enumerating regular languages with bounded delay. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023, March 7-9, 2023, Hamburg, Germany*, volume 254 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

11 S. Angelov, B. Harb, S. Kannan, S. Khanna, and J. Kim. Efficient enumeration of phylogenetically informative substrings. In *Research in Computational Molecular Biology, 10th Annual International Conference, RECOMB 2006*, volume 3909 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2006.

12 M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

13 M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

14 J. R. Bitner, G. Ehrlich, and E. M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, 1976.

15 K. Cattell, F. Ruskey, J. Sawada, M. Serra, and C.R. Miers. Fast Algorithms to Generate Necklaces, Unlabeled Necklaces, and Irreducible Polynomials over GF(2). *Journal of Algorithms*, 37(2):267–282, 2000.

16 K. Cevorová, G. Jirásková, P. Mlynárcik, M. Palmovský, and J. Sebej. Operations on automata with all states final. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014*, volume 151 of *EPTCS*, pages 201–215, 2014.

17 R. Clifford, M. Jalsenius, E. Porat, and B. Sach. Pattern matching in multiple streams. In *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012*, volume 7354 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2012.

18 C. Collins, M.S. Dyer, M.J. Pitcher, G.F.S. Whitehead, M. Zanella, P. Mandal, J.B. Claridge, G.R. Darling, and M.J. Rosseinsky. Accelerated discovery of two crystal structure types in a complex inorganic phase field. *Nature*, 546(7657):280, 2017.

19 A. Conte, R. Grossi, G. Punzi, and T. Uno. Enumeration of maximal common subsequences between two strings. *Algorithmica*, 84(3):757–783, 2022.

20 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.

21 E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. In *International Colloquium on Automata, Languages, and Programming*, pages 341–353. Springer, 2009.

22 H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978.

23 Hermann Gruber, Jonathan Lee, and Jeffrey O. Shallit. Enumerating regular expressions and their languages. In Jean-Éric Pin, editor, *Handbook of Automata Theory*, pages 459–491. European Mathematical Society Publishing House, Zürich, Switzerland, 2021.

24 W. Grüner, R. Giegerich, D. Strothmann, C. Reidys, J. Weber, I. L. Hofacker, P. F. Stadler, and P. Schuster. Analysis of rna sequence structure maps by exhaustive enumeration. i. *Monatshefte für Chemie*, 127:355–389, 1996.

25 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

26 J.-Y. Kao, N. Rampersad, and J. O. Shallit. On nfas where all states are final, initial, or both. *Theor. Comput. Sci.*, 410(47-49):5010–5021, 2009.

27 D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

28 A. O. Lyakhov, A. R. Oganov, and M. Valle. How to predict very large and complex crystal structures. *Computer Physics Communications*, 181(9):1623 – 1632, 2010.

29 E. Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–61, 1997.

30 C. Mellot-Draznieks, S. Girard, G. Férey, J. C. Schön, Z. Cancarevic, and M. Jansen. Computational design and prediction of interesting not-yet-synthesized structures of inorganic materials by using building unit concepts. *Chemistry – A European Journal*, 8(18):4102–4113, 2002.

31 T. Mütze. Combinatorial Gray codes - an updated survey. *arXiv preprint arXiv:2202.01280*, 2022.

32 A. R. Oganov. Crystal structure prediction: reflections on present status and challenges. *Faraday Discuss.*, 211:643–660, 2018.

33 A. R. Oganov and C. W. Glass. Crystal structure prediction using ab initio evolutionary techniques: Principles and applications. *The Journal of chemical physics*, 124(24), 2006.

34 F. Ruskey, C. Savage, and T. Min Yih Wang. Generating necklaces. *Journal of Algorithms*, 13(3):414–430, 1992.

35 F. Ruskey and J. Sawada. Generating necklaces and strings with forbidden substrings. In *Computing and Combinatorics, 6th Annual International Conference, COCOON 2000*, volume 1858 of *Lecture Notes in Computer Science*, pages 330–339. Springer, 2000.

36 G. Sacomoto, V. Lacroix, and M.-F. Sagot. A polynomial delay algorithm for the enumeration of bubbles with length constraints in directed graphs. *Algorithms Mol. Biol.*, 10:20, 2015.

37 C. D. Savage. A survey of combinatorial gray codes. *SIAM Rev.*, 39(4):605–629, 1997.

38 J. Sawada. Generating bracelets in constant amortized time. *SIAM Journal on Computing*, 31(1):259–268, jan 2001.

39 J. Sawada. A fast algorithm to generate necklaces with fixed content. *Theoretical Computer Science*, 301(1-3):477–489, may 2003.

40 M. L. Schmid and N. Schweikardt. Spanner evaluation over slp-compressed documents. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 153–165. ACM, 2021.

41 M. L. Schmid and N. Schweikardt. Query evaluation over slp-represented document databases with complex document editing. In *PODS '22: International Conference on Management of Data*, pages 79–89. ACM, 2022.

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

43 J. O. Shallit. Decidability and enumeration for automatic sequences: A survey. In *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013*, volume 7913 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2013.

44 R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

45 T. Uno. *Amortized Analysis on Enumeration Algorithms*, pages 72–76. Springer New York, New York, NY, 2016.

46 Y. Wang, J. Lv, L. Zhu, S. Lu, K. Yin, Q. Li, H. Wang, L. Zhang, and Y. Ma. Materials discovery via calypso methodology. *Journal of physics. Condensed matter : an Institute of Physics journal*, 27:203203, 04 2015.

47 K. Wasa. Enumeration of enumeration algorithms. *arXiv e-prints*, pages arXiv–1605, 2016.

A Computational Model

In the problems we consider, the input consists in a natural number n, a DFA with m states, over an ordered alphabet $\{1, \ldots, \sigma\}$ with σ letters; let $N = \max\{n, m, \sigma\}$. The computational model we use is the RAM with logarithmic word size. More precisely, we assume that each memory word can hold $\log N$ bits and arithmetic operations with numbers in $[N]$ take $O(1)$ time. Numbers larger than N, with ℓ bits, are represented in $O(\ell/\log n)$ memory words, and working with them takes time proportional to the number of memory words on which they are represented. The words processed in our algorithms are seen as sequences of integers, each fitting in one memory word. This model is common in string algorithms, see [20], as well as in enumeration algorithms, see [47] and the references therein, especially those addressing problems where the input is a number n and enumerating, generating, ranking, or unranking structures (such as strings, graphs, trees) of size n fulfilling certain properties is required.

We can assume that in our computational model, as in many modern programming languages, tail calls to a recursive function are implemented without adding a new stack frame to the call stack.

B Languages of Strings with Forbidden Factors

We first show Lemma 1.

▶ **Lemma 1.** *Let $\mathcal{F} \subseteq \Sigma^*$ be a finite set of forbidden strings. A PCA accepting $L_{\mathcal{F}}$ can be built in $O(||\mathcal{F}|||\Sigma|)$ time.*

Proof. Let A be the DFA constructed from the Aho-Corasick machine [6] accepting all strings over Σ ending with strings from $\mathcal{F}$. The DFA A can be constructed in $O(||\mathcal{F}|||\Sigma|)$ time. We can convert A into an incomplete DFA A' that recognizes strings that do not contain any string from $\mathcal{F}$ as factor by first removing all final states of A and then making all other states final. Every transition from A that did not involve a final state is then copied to A'. ◀

Further, let us consider the language $L_{\mathcal{F}}$ in the case of a single forbidden factor (i.e., $\mathcal{F} = \{f\}$). In that case, a succinct representation of the incomplete DFA accepting $L_{\mathcal{F}}$ can be constructed more efficiently, in $O(m)$ time only, where $m = |f|$. The key observation (see, e.g., [17]) is that the DFA $A = (Q, \Sigma, q_0, F, \delta)$ accepting all strings over Σ ending with f has the following structure:

- $Q = \{q_0, q_1, \ldots, q_m\}$ and q_m is the single final state;
- $\delta(q_{i-1}, f_i) = q_i$ for $i \in \{0, 1, \ldots, m-1\}$ and for all $i \in \{0, 1, \ldots, m\}$ and $a \neq f_i$ we have that $\delta(q_{i-1}, a) = q_j$ for some $j \leq i-1$;
- the number of transitions connecting non-initial states of this automaton is in $O(m)$.

Therefore, to construct A, it is enough to compute the transitions connecting its states to other non-initial states (as all other transitions lead to q_0). This can be done in $O(m)$ time using, e.g., the Knuth-Morris-Pratt algorithm [27]. So, indeed, a succinct representation of the PCA accepting $L_{\mathcal{F}}$ can be constructed in $O(m)$ time (again, by removing the final state of A, making all other states final, and noting that all undefined transitions lead to q_0, except for the transition $\delta(q_{m-1}, f_m)$ which leads to a failure state).

C Technical Details for Data Structures and Automata

In this appendix, we provide full proofs of the technical lemmas used by our data structures.

▶ **Lemma 2.** *Given the PCA A, the lists L_q, for all $q \in Q$, can be computed in $O(m\sigma)$ time.*

Proof. The following algorithm is rather standard (and could be considered folklore), so we go quickly over it.

The algorithm works on the graph A (corresponding to the PCA A). We firstly detect in $O(m\sigma)$ the strongly connected components of this graph, using, e.g., Tarjan's algorithm [44]. Then, we detect all the other nodes of A from which there is a path to a node from one of the strongly connected components. This, e.g., can be done as follows:

- Initially, we color all the nodes of A white.
- We put all the nodes of the strongly connected components in a queue R and also color them red.
- We repeat the following step until R is empty: we extract the first node p in the queue, and put in R all the white nodes s such that there is an edge (s, p) in A, and color each such node s red.

After this, we set $\pi(q) = \infty$ for all the red nodes, and remove them from A, as well as all the edges leaving or entering them. This whole process can be implemented in $O(m\sigma)$ time.

The remaining white nodes and edges form a directed acyclic graph A'. Now, we run the following folklore procedure:

- Find a topological ordering of the nodes of A'.
- Consider the nodes in inverse order of the topological ordering. When q is considered in this order, set $\pi(q)$ to be 0 if there is no edge leaving q or, otherwise, 1 plus the maximum of the values $\pi(p)$ where (q, p) is an edge in A'.

It is immediate that, at this point, we have computed the length of the longest path leaving each node q and stored it in $\pi(q)$. Again, this whole process runs in $O(m\sigma)$ up to this point.

Further, we explain how the lists L_q are computed, for all states q. Initially, all these lists are empty. Then, for each transition $\delta(q, a) = q'$, we store the triple $(q, \pi(q'), a)$ in a set S. Clearly, $|S| \in O(m\sigma)$. We sort (in $O(m\sigma)$ time, using radix sort) the triples of S decreasingly according to the second component, breaking ties according to their third component. Now, we go through the elements of the decreasingly sorted list S and when the element $(q, \pi(q'), a)$ is reached we add the pair $(a, \pi(q') + 1)$ as the last element of the list L_q. This process computes correctly the lists L_q, and requires $O(m\sigma)$ time.

This concludes the proof of this lemma. ◀

▶ **Lemma 3.** *Given the default graph $D(A)$, we can compute in $O(m)$ time all its default components, and store for each node q in $D(A)$ the component in which it is contained.*

Proof. We first identify in $D(A)$ the nodes q with out-degree 0 (that is, the nodes with $\pi(q) = 0$). These are the roots of the independent trees. For each such node q we perform a depth-first search in $D(A)^{-1}$ (the graph $D(A)$ with the orientation of the edges inverted) and this allows us to discover the independent tree rooted in q. Clearly, this process takes time proportional to the number of edges in these trees, so $O(m)$ overall, and we can associate to each node in the trees a pointer to the default component (e.g., a pointer to the root of the independent tree) which contains it.

Now, to discover the cycles, we consider a node q which does not belong to any independent tree. This is either on a cycle or in a tree whose root is on a cycle. In all cases, we can start a traversal of the graph following the only edge leaving q (and, further, the single edges leaving the nodes we meet in this traversal). As at some point we will reach one node a second time, and this allows us to identify a cycle of $D(A)$, denoted α. This can either be precisely the cycle containing q, or the cycle on which the root of the tree containing q is found. We store this cycle α, arbitrarily choose an initial node for α, and, for each of its nodes, we store pointers to α (e.g., to its initial node) as well as their position on α (w.r.t. the initial point). Then, for each node r of α we discover the tree rooted in r, exactly as we did in the case of independent trees, with the single difference being that we do not explore in our traversal the node of the cycle α from which there is an incoming edge towards r. In this way, we find the trees rooted in each of the nodes r on the cycle, and compute all the required information for their nodes. The time needed to do this is $O(t_\alpha)$ where t is the total number of nodes of α and of the trees rooted in nodes of α (as the time needed to complete the traversals we performed in our algorithm is proportional to the number of edges in the cycle and the attached trees). After we are done, if there are still nodes that do not belong to the default components discovered so far, we select one of them and repeat the process.

Note that, for simplicity, we will assume that if a node is both on a cycle and the root of a tree, then it will have pointers both to the cycle and to the tree structure.

The algorithm we have described is clearly correct, and it runs in time proportional to the number of edges of $D(A)$, so $O(m)$. ◀

▶ **Lemma 4.** *We can build in $O(m)$ time data structures, allowing us to retrieve the node* $\mathrm{PathMaxNode}(s, \ell)$ *in $O(1)$ time, for each default path (s, ℓ) of $D(A)$.*

Proof. Firstly, we will try to obtain a clearer image of which node we want to retrieve from each default path.

So, let us assume that we are given the default path (s, ℓ). We will have several cases, according to the type of the default component containing the respective path. In each case, once we clarify what node we need to compute, we also explain what data structures are needed, and how they can be used to return this node and its distance from s.

The main idea is to define for each node of the graph $D(A)$ a value $f(q)$ which is independent of the paths we are given as queries, such that finding for some default path (s, ℓ) of $D(A)$ the node $q = \mathrm{PathMaxNode}(s, \ell)$ of this path, which maximises the sum of w_q and the distance between s and q along the default path (s, ℓ), becomes equivalent to finding the node q of that path, for which the value $f(q)$ is maximum.

Case 1. Assume that s is in an independent tree τ. Then, the respective default path (s, ℓ) is a path going in the respective tree from s towards its root r, and as explained before, we can retrieve its end point s' in $O(1)$ time using one level ancestor query in τ. We want to retrieve the node q of this path such that the sum of w_q and the distance between s and q along the default path (s, ℓ) is maximum. But this also means that the sum of w_q and the distance between s and r in the tree, from which we subtract d^t_q, is maximum. In other words, we look for the node $q \neq s'$ on the unique path from s to s', for which the value $f(q) = w_q - d^t_q$ is maximum; note that this value is independent on (s, ℓ).

So, the setting is that we have the tree τ and associate to each of its nodes q the value $w_q - d^t_q$. Now, we simply need to be able to retrieve the node of maximum value on paths starting from a node, having a given length, and going towards the root r. Hence, we need to answer *path maximum queries in the tree τ*. A data structure can be constructed in $O(n_\tau + T_\tau)$ time, allowing us to answer such queries in constant time [21]. Here T_τ is the time needed to sort the values associated to the nodes of the tree. However, as these values can be sorted for all independent trees simultaneously, and the absolute values associated to the nodes of the trees are either ∞ or $O(m)$, then the total time needed to sort them is $O(m)$ (again, radix sort can be used, while keeping track of the tree from which each value comes). So, we can retrieve the node q on the first component of a query $\mathrm{PathMaxNode}(s, \ell)$

in $O(1)$ time, after an $O(m)$-time processing; the distance d between the start of the path and the node q is simply the difference between d^t_s and d^t_q.

Case 2. Assume now that s is on a cycle α. We can compute the end-node s' of the path (s, ℓ), as explained before. Now, let i_s be the leftmost (first) occurrence of s in the array associated to α and $i_{s'}$ (respectively, $j_{s'}$) be the first (respectively, second) occurrence of s' in that array. If $\ell < |\alpha|$, then the path from s to s' goes exactly once through the nodes contained between i_s and $j_{s'}$ in the array associated to α. Similarly to the case of trees, discussed above, it is again enough to identify that node q for which $w_q - d^c_q$ is maximum (this value does not depend on (s, ℓ), just like above). If $\ell \geq |\alpha|$, then the path from s to s' goes through all the nodes of the cycle one or more times. Now, if we consider a node r on this cycle, it is not hard to see that the sum of w_r and the distance between s and r along the default path (s, ℓ) is maximum for the last occurrence of this node r on the respective path. Therefore, it is enough to select the node q for which the sum of w_q and the distance between s and q along the default path (s, ℓ) is maximum by considering only the last occurrence of the nodes of α on the path (s, ℓ). But, once more, just like in the case of trees, this means retrieving the node q for which the value $w_q - d^c_q$ is maximum from the nodes appearing in the range $[i_{s'}, j_{s'} - 1]$.

So, in both cases, we need to associate with each node q in the array corresponding to α the value $f(q) = w_q - d^c_q$. Then, we build in $O(|\alpha|)$ time data structures allowing us to answer *range maximum queries for this array* in $O(1)$ time [12]. That is, we can retrieve in $O(1)$ time the node q of maximum value from a range $[g : h]$ of that respective array. The distance between s and q can be trivially computed in $O(1)$ time. This is precisely what we needed.

Case 3. Assume, finally, that s is in a tree τ whose root r is on a cycle α. We produce the same data structures as in case 1 for the tree τ and we already have the data structures mentioned in case 2 for the cycle α. Now, we explain how to answer the queries for a path (s, ℓ).

If the default path (s, ℓ) is completely contained in τ, we can proceed as in case 1. This can be checked by simply looking whether the end node of the path (s, ℓ) is a node of τ. If $\ell > d^t_s$, then the path ends inside the cycle α. So, we split the default path in two sub-paths (s, d^t_s) and $(r, \ell - d^t_s)$ and obtain the node q_1 on the path (s, d^t_s) such that the sum of w_{q_1} and the distance between s and q_1 along the default path (s, d^t_s) is maximum (as in case 1) and the node q_2 on the path $(r, \ell - d^t_s)$ such that the sum of w_{q_2} and the distance between r and q_2 along the default path $(r, \ell - d^t_s)$ is maximum (as in case 2). Note that q_2 also has the property the sum of w_{q_2} and the distance between s and q_2 along the default path (s, ℓ) is maximum compared to the respective sum for all the nodes on the path $(r, \ell - d^t_s)$. Now, as the distance between s and q_1 can be computed easily (it is $d^t_s - d^t_{q_1}$) as well as the distance between s and q_2 (it is simply ℓ from which we subtract the distance between q_2 and the end of the path $(r, \ell - d^t_s)$), we can compare which is greater: the sum of w_{q_1} and the distance between s and q_1 along the default path (s, ℓ) or the sum of w_{q_2} and the distance between s and q_2 along the default path (s, ℓ). The node for which the respective sum is greater is the node we wanted to retrieve. It is important to notice that obtaining the respective node can be done in $O(1)$ (as well as the distance from s to the respective node), after the linear preprocessing described at the beginning of the analysis for this case is performed.

This ends our case analysis, and concludes the proof of our claim. ◀

D Pseudo-Code

See Algorithm 1.

Algorithm 1 The main recursive procedure of our enumeration algorithm.

```
1:  procedure Enumerate(Letter $a$, State $q$ of $A$, length $\ell$)
2:      Push $(a, q, \ell)$ in $\mathcal{S}$.
3:      Push (stack frame of this call, pointer to top element of $\mathcal{S}$) as the top element of $\mathcal{C}$.
4:      Output $(n - \ell - 1, a, q, \ell)$, which describes the current path w.r.t. the previous one.
5:      Return, if $\ell = 0$.
6:      Let $U$ be an empty list.
7:      Let $(p, d) = \text{PathMaxNode}(q, \ell)$.
8:      if $(d + w_p) \geq \ell$ then
9:          Add $((q, \ell), q, 0)$ to $U$.
10:         Let $(b, e)$ be the second element of $L_p$;
11:         Let $last = (b, p', \ell - d - 1)$.
12:     else
13:         Remove the elements of $\mathcal{S}$ and $\mathcal{C}$ corresponding to this call.
14:         Return.
15:     end if
16:     while $U$ is not empty do
17:         Extract the first tuple $((s, j), q, h)$ from $U$. Let $(q', f) = \text{PathMaxNode}(s, j)$.
18:         Add $((s, f), q, h)$ to $U$, if $f > 0$, $(r, e) = \text{PathMaxNode}(s, f)$ and $h+e+1+w_r \geq \ell$.
19:         Add $((q'', j - f - 1), q, h + f + 1)$ to $U$, where $q''$ is the successor of $q'$ on $(s, j)$,
              if $j - f - 1 > 0$, $(r, e) = \text{PathMaxNode}(q'', f)$ and $h + f + 1 + e + 1 + w_r \geq \ell$.
20:         if $((s, j), q, h) \neq ((q, \ell), q, 0)$ then
21:             $x = 2$
22:         else
23:             $x = 3$
24:         end if
25:         Let $(b, g)$ be the $x^{th}$ element of $L_{q'}$ ($(b, g)$ is undefined if $|L_{q'}| < x$)
26:         while $(b, g)$ is defined and $h + f + g + 1 \geq \ell$ do
27:             Enumerate$(b, q', \ell - h - f - 1)$.
28:             Set the top element of $\mathcal{S}$ to the element of $\mathcal{S}$ pointed by the top element of $\mathcal{C}$.
29:             Let $(b, g)$ be the next element of $L_{q'}$ ($(b, g)$ is undefined if $|L_{q'}| < x$)
30:         end while
31:     end while
32:     Pop the stack frame of Enumerate$(a, q, \ell)$, and the attached pointer, from $\mathcal{C}$.
33:     Enumerate$(last)$, where $last = (b, p', \ell - d - 1)$
34:     Return.
35: end procedure
```

E Technical Details for the Enumeration Algorithm

We can now formally describe our algorithm. Its input is a positive integer n and the PCA $A = (Q, \Sigma, q_0, Q, \delta)$, with $|Q| = m$ and $|\Sigma| = \sigma$. We want to enumerate the elements of the language $L(A) \cap \Sigma^n$.

We first preprocess the PCA A using the algorithms from Section 3, including those from Lemmas 2, 3, 4. This takes $O(m\sigma)$ and we obtain all data structures defined in Section 3, including, in particular, data structures allowing us to retrieve in $O(1)$ time the answer to PathMaxNode queries. Note that this preprocessing time is does not depend on n.

We also define two empty stacks $\mathcal{S}$ and $\mathcal{C}$. These are maintained as global variables during the execution of our algorithm. The stack $\mathcal{S}$ contains tuples (a, q, ℓ), with $a \in \Sigma, q \in Q, \ell \in [n]$. Intuitively, if the content of the stack $\mathcal{S}$ is, at some step of the computation the sequence $\langle(\uparrow, q_0, \ell_0), (a_0, q_1, \ell_1), \ldots, (a_{t-1}, q_t, \ell_t)\rangle$ (where the top of the stack is to the right of this sequence), then the currently enumerated string is $w_0 a_0 w_1 a_1 \cdots w_{t-1} a_{t-1} w_t$, where, for $i \geq 0$, w_i is the label of the default path of length ℓ_i starting with the state q_i and ending with some state q_i', and $q_{i+1} = \delta(q_i', a_i)$, for $0 \leq i \leq t-1$. Clearly, the string w can be retrieved explicitly from its representation on the stack $\mathcal{S}$ in linear time. With respect to the execution of our algorithm, $\mathcal{S}$ corresponds to the stack of currently active recursive calls.

The usage of $\mathcal{C}$ is more subtle. The stack $\mathcal{C}$ contains tuples (a, q, ℓ), with $a \in \Sigma, q \in Q, \ell \in [n]$, with the property that each such tuple also occurs in $\mathcal{S}$; therefore, for each tuple, we will also store, together with it, in the stack $\mathcal{C}$ a pointer to the corresponding record of $\mathcal{S}$. Intuitively, at every step of the computation, $\mathcal{C}$ contains (bottom to top, in the same order as in $\mathcal{S}$) exactly those triples (a, q, ℓ) of $\mathcal{S}$ for which the default path of length ℓ starting in q still contains branching nodes leading to paths of length n which were not enumerated yet. As such, $\mathcal{C}$ allows for the quick identification of the next path which we need to output in our enumeration: such a path should go through one of the branching points of the default path found on top of this stack $\mathcal{C}$. From the point of view of the execution of our algorithm, $\mathcal{C}$ corresponds to the currently active recursive calls, which were not tail calls. Recall that we assume that, in the computational model we use, tail calls are implemented so that no new stack frame is added to the call-stack. Hence, $\mathcal{C}$ actually corresponds, at each moment of our algorithm's execution, to the current call-stack.

As mentioned in the informal description of our approach in Section 4, a recursive procedure, named Enumerate, is central to our approach. At each call, Enumerate takes three parameters, a letter $a \in \Sigma \cup \{\uparrow\}$, a node/state q, and a length ℓ. Referring to our intuitive explanation, we now have to go through all paths starting from q and having length ℓ. Each will lead to a path of length n that we need to output.

Now we will describe how the call Enumerate(a, q, ℓ) works. The pseudo-code of this procedure is given in Algorithm 1 in Appendix D.

We begin by pushing (a, q, ℓ) in the stack $\mathcal{S}$. Furthermore, an element containing a pointer to the element just pushed in $\mathcal{S}$ as well as the stack frame of the call Enumerate(a, q, ℓ) is pushed in $\mathcal{C}$.

Then, we output the tuple $(n - \ell - 1, a, q, \ell)$, encoding path of length n consisting in the prefix of length $n - \ell - 1$ of the previously output string from $L(A) \cap \Sigma^n$, followed by a, and the default path (q, ℓ).

At this point, we need to see whether there are more paths of length ℓ starting with q that we need to go through. Each of these paths will be obtained via a recursive call of Enumerate, and it is important to make sure that the last path we discover in this way is obtained via a tail call. To facilitate the discovery of paths, we will maintain a list U, whose elements are of the form $((s, j), q, d)$, where (s, j) is a default path on which there is at least one branching node that needs to be explored, and (s, j) is part of the default path starting from q such that one needs to traverse d edges to get from q to s on this path. U is initially empty.

Firstly, we detect if there is at least one path of length ℓ from q, that we did not enumerate yet. For this, we retrieve $(p, d) = \text{PathMaxNode}(q, \ell)$.

If $d + w_p \geq \ell$, we add the tuple $((q, \ell), q, 0)$ to U; this means that there is a branching node on the default path (q, ℓ) from which we can obtain at least one new path. One of these new paths will be certainly obtained by calling $\text{Enumerate}(b, p', \ell - d - 1)$, where (b, e) is the second element of L_p and $\delta(p, b) = p'$, i.e., the longest path starting with a non-default edge from p starts with the edge going from p to p' and continues with the default path from p'. Note that the fact that this path is long enough is guaranteed by the truth of the inequality $d + w_p \geq \ell$. Now, we have discovered at least one recursive call that we need to do in the current instance of Enumerate. However, we will not execute this call immediately. Instead, we will postpone this call until the end of the procedure, and this will be the tail call (we avoid in this way doing a series of tedious checks which would allow us to manage the tail call otherwise). We save in a variable *last* the tuple $(b, p', \ell - d - 1)$, the parameters of the future tail call.

If $d + w_p < \ell$, then there are no more paths to be explored (the default path from q was the only one), so we simply return (and remove the elements corresponding to the call $\text{Enumerate}(a, q, \ell)$ from both stacks $\mathcal{S}$ and $\mathcal{C}$).

In the case we added something to U, we continue as follows.

While $U \neq \emptyset$, we extract the first tuple $((s, j), q, h)$ from U and process it as follows.

Assume first that $((s, j), q, h) \neq ((q, \ell), q, 0)$. We retrieve $(q', f) = \text{PathMaxNode}(s, j)$. This is a branching node, so we will need to do some recursive calls for it. But first we will update U as follows. We add to U the element $((s, f), q, h)$, if $f > 0$ and $(r, e) = \text{PathMaxNode}(s, f)$ and $h+e+1+w_r \geq \ell$. We add to U the element $((q'', j-f-1), q, h+f+1)$, where q'' is the successor of q' on the default path (s, j), if $j - f - 1 > 0$ and $(r, e) = \text{PathMaxNode}(q'', f)$ and $h + f + 1 + e + 1 + w_r \geq \ell$. The idea is that we will now explore the possible paths originating in q' but we still need to explore the possible paths originating in branching nodes from (s, f), if that path is non-empty (meaning that $s \neq q'$ or, in other words, $f > 0$), or from $(q'', j - f - 1)$, if that path is non-empty (meaning that $j - f - 1 > 0$). Once U is updated, we traverse the list $L_{q'}$ starting with its second element. Let the current element be $(b, g) \in L_{q'}$. We check if $h + f + g + 1 \geq \ell$ (i.e., we check if there is a long enough path going on the default path (q, ℓ) through q, then s, until it reaches q' and then following the transition $(q', \delta(q', b))$, which can lead to a new path that we need to enumerate). If $h + f + g + 1 \geq \ell$, we call $\text{Enumerate}(b, q', \ell - h - f - 1)$; when this call is completed, we update the pointer to the top element of $\mathcal{S}$ to correspond to the element of $\mathcal{S}$ pointed by the top element of $\mathcal{C}$ (i.e., eliminate tail calls from $\mathcal{S}$, as now they also became useless in our representations of words) and continue our traversal of $L_{q'}$. If If $h + f + g + 1 < \ell$, we simply stop the traversal of $L_{q'}$.

If $((s, j), q, h) = ((q, \ell), q, 0)$, note that $\text{PathMaxNode}(q, \ell) = (p, d)$. We do exactly the same thing as above, except for the fact that we start the traversal of L_p with its third node. This is because the second node of L_p is p', and we have already saved the call $\text{Enumerate}(b, p', \ell - d - 1)$ to be the tail call of our procedure.

As soon as U is empty, we will simply run the tail call $\text{Enumerate}(b, p', \ell - d - 1)$, and then return. Just before doing that, we remove the stack frame of $\text{Enumerate}(a, q, \ell)$ from the stack $\mathcal{C}$, to simulate the management of tail calls in our model.

We can now show the correctness of our approach. Intuitively, one can show, by induction, that there is a bijection between the words $w \in L(A)$ and the path from the root to leaves in the tree of recursive calls whose root is $\text{Enumerate}(\uparrow, q_0, n)$ (in this tree, the nodes are instances of Enumerate called in our algorithm, and their children are the calls initiated in the respective instances).

▶ **Lemma 5.** *The call* Enumerate$(\uparrow, q_0, n)$ *outputs a representation of every string of length* n *accepted by the PCA* A *exactly once.*

Proof. The proof has several steps.

Firstly, let us note that if we call Enumerate(a, q, ℓ) at some point during the execution of our algorithm, then there is a path of length $n - \ell$ ending with the letter a (or the path representing the empty word if $a = \uparrow$) that leads from q_0 to q. This can be shown by induction on $n - \ell$. If $n - \ell = 0$, the conclusion follows by the fact that the only time we call Enumerate with the third parameter being equal to n is in the first call Enumerate$(\uparrow, q_0, n)$. Also note that no other call is made during the execution of Enumerate$(\uparrow, q_0, n)$ which has the first parameter equal to $\uparrow$. Now, assume that the claim holds for $n - \ell \leq k$, and we will show it for $n - \ell = k + 1$. In this case, the call Enumerate(a, q, ℓ) was initiated by a previous recursive call Enumerate$(b, q', \ell + t)$, for some $t > 0$. This means that there is a node q'' on the default path $(q', \ell + t)$, at distance $t - 1$ from q, such that $\delta(q'', b) = q'$ and the edge from q'' to q is non-default and labelled with b. The conclusion follows immediately.

By this claim, we immediately get that every word that is output during the execution of the call Enumerate$(\uparrow, q_0, n)$ represents a word of length n of $L(A)$.

Further, let us show first that during the call Enumerate(a, q, ℓ) we correctly identify all branching nodes on the path (q, ℓ); that is, all nodes q' from which we can follow a non-default edge and extend the path leading from q_0 to q and then to q' to a path of length n. Firstly, for such a node to exist, then $d + w_p \geq \ell$ must hold for $(p, d) =$ PathMaxNode(q, ℓ). If (p, d) does not fulfil this requirement, there is no other path of length ℓ starting in a node of the default path (q, ℓ), and our algorithm reports that correctly. So, our algorithm considers p as a first branching node. If the path (q, ℓ) had length 1, then we are done. If our path is longer, after correctly identifying the branching node p, such that $(p, d) =$ PathMaxNode(q, ℓ), our algorithm looks for branching nodes on the default paths (q, d) (from q to p) and $(p', \ell - d - 1)$, which starts with the successor of p on the default path (q, ℓ). Basically, we have partitioned the path (q, ℓ) in three parts: the branching node and two shorter paths, on which we keep looking for branching nodes. The search on each of these paths is done exactly like the search on the whole path (q, ℓ), with PathMaxNode queries, which is correct. Ultimately, we either rule out entire paths because not even the node returned by PathMaxNode leads to a long enough path, or we split them further. Eventually, in at most ℓ splitting steps, each node of the default path (q, ℓ) is either correctly ruled out or discovered as a branching node.

Now, let w be a word of length n from $L(A)$. We will show that a representation of w is output by the call Enumerate$(\uparrow, q_0, n)$. Basically, we can write $w = w_0 a_0 w_1 a_1 \dots w_{t-1} a_{t-1} w_t$, where w_i is the label of a default path between the states q_i and q'_i, for $i \in \{0\} \cup [t]$, and a_{i-1} labels a non-default edge between q'_{i-1} and q_i, for $i \in [t]$. Note that this representation of a word as a decomposition of labels of default paths (of length ≥ 0) and their connecting edges is unique. In the call Enumerate$(\uparrow, q_0, n)$ we identify all the branching nodes on the path (q_0, n). Therefore, we also identify the branching node q'_0 (as the sum of the length of the path from q_0 to q'_0 along the default path (q_0, n) and the longest path starting with a non default edge from q'_0 is at least n, as exhibited by w). Given that there is a non default edge labelled with a from q'_0 to q_1, we will call Enumerate$(a, q_1, n - |w_0| - 1)$. Let us assume that at some point we have called Enumerate$(a_k, q_{k+1}, n - \sum_{i=0,k} |w_i| - (k + 1))$ and the path from q_0 to q_{k+1} is labelled with $w_0 a_0 w_1 a_1 \dots w_{k-1} a_{k-1} w_k a_k$. We will discover the branching node q'_{k+1} on the default path starting in q_{k+1}, after traversing the prefix labelled with w_{k+1} of this path. There must also be a non-default edge between q'_{k+1} and q_{k+2} labelled with a_{k+1}. So, during the execution of Enumerate$(a_k, q_{k+1}, n - \sum_{i=0,k} |w_i| - (k + 1))$ we will call, at some point, Enumerate$(a_{k+2}, q_{k+2}, n - \sum_{i=0,k+1} |w_i| - (k + 2))$. By this inductive argument, it follows that we will output the representation of w when Enumerate$(a_{t-1}, q_t, n - \sum_{i=0,t-1} |w_i| - t)$ is executed.

It only remains to show that each word of length n from $L(A)$ is only enumerated once, so we do not output two triples representing the same word of $L(A) \cap \Sigma^n$. This follows from the fact that a word/path w of $L(A)$ has a unique decomposition as a concatenation of default paths and connecting non-default edges. As above, let $w = w_0 a_0 w_1 a_1 \ldots w_{t-1} a_{t-1} w_t$, where w_i is the label of a default path between the states q_i and q'_i, for $i \in \{0\} \cup [t]$, and a_{i-1} labels a non-default edge between q'_{i-1} and q_i, for $i \in [t]$. Basically, to obtain the word w, we first call Enumerate$(\uparrow, q_0, n)$. Then, the next call of Enumerate must be Enumerate$(a, q_1, n - |w_0| - 1)$ (there is no other way to choose a branching node than choosing the one at the end of the longest common prefix of w and the label of the default path (q_0, n)). By an inductive process, we see that actually the calls of Enumerate that lead to the output of the representation of w are uniquely determined. Therefore, a representation of this word is only output once.

This concludes our proof. ◀

According to Lemma 5, the call Enumerate$(\uparrow, q_0, n)$ outputs for each string of $L(A) \cap \Sigma^n$ exactly one representation. This leaves the question of the time complexity of the algorithm, specifically the delay. The following theorem states the main result of this paper.

▶ **Theorem 6.** *Given a number n and a PCA A, with m states and input alphabet of size σ, we can enumerate, without repetitions, the strings of length n recognised by A with worst-case delay of $O(1)$, after a preprocessing taking $O(m\sigma)$ time.*

Proof. We build, for the PCA A, the data structures from Section 3, as a preprocessing, and then run Enumerate$(\uparrow, q_0, n)$, if $\pi(q_0) \geq n$.

To show that the enumeration performed by the algorithm is done with constant delay, it is enough to show that the operations performed between two consecutive output-operations can be executed in constant time.

Each output operation is associated to a call of Enumerate. So, let us consider two such output operations, caused by two distinct calls to Enumerate. There are several cases we need to check.

In the first case, the first output operation is done in a (parent-)instance of Enumerate, which then directly calls the instance of Enumerate in which the second output is performed. Clearly, this means that the operations performed between these calls are those from lines 5–15 of Algorithm 1 followed by exactly one execution of the lines 17–25 from the while-loop of line 16; all these are done in the parent-instance. With the data structures defined in Section 3, these can all be executed in $O(1)$ time.

In the second case, both output operations are done in instances of Enumerate originating (maybe not directly, but in a sequence of recursive calls) from the same parent-instance of Enumerate. The idea is that once the first output operation is performed, the instance that executed this operation will not make any other recursive call (as this would cause an intermediate output operation) before it ends. Once it ends (and, note at this point that getting from the output in line 4 to any return instruction requires only $O(1)$ time, with our data structures, if no further recursive calls are made), exactly the next call of Enumerate must be the one that causes the second output-operation (or, again, we would have intermediate output operations). So, the algorithm directly returns to the parent-instance (the origin of the two calls leading to the two consecutive output operations), and this means that every intermediate call that led from the parent-instance to the one causing the first output was a tail call. So, this return to the parent-instance is done in $O(1)$ time in our setting. Now, looking at the recursive call that led to the instance of Enumerate which produced the first output and at the one that leads to the second output, it must hold that they happen in two consecutive iterations of the while-loop from line 26 of the algorithm, or the first happens in the last iteration of that loop, and the second is either the tail call from line 33 or the first call made in a new iteration of the while-loop from line 16 (and in the first iteration of the while loop from line 26 in this new iteration). In both cases, the time required to complete the computations between two such calls is constant (with the help of our data structures).

To conclude, while the first case is straightforward, the second case is a bit more involved and can be seen like this. The first output is done by an instance of Enumerate, which then returns after $O(1)$ time. This then takes us in $O(1)$ time (due to the management of tail calls) to the parent-instance that originated the sequence of calls leading to the first output. Then, we can reach the call of Enumerate that makes the second output in $O(1)$ time. This is basically the next call to Enumerate made by the parent instance, after the one that led to the first output; the time needed to execute the instructions between two such calls is $O(1)$.

There are no other cases to be considered, so our claim follows. ◀

F Technical Details for the Extensions of the Enumeration Algorithm

▶ **Theorem 7.** *Given two numbers $\ell \leq n$ and a PCA A, with m states and input alphabet of size σ, we can enumerate, without repetitions, the strings of length at least ℓ and at most n recognised by A, in increasing order of their length, with worst-case delay of $O(1)$, after a preprocessing taking $O(m\sigma)$ time.*

Proof. We build, for the PCA A, the data structures from Section 3, as a preprocessing. This takes $O(m\sigma)$ time. Again, note that this phase does not depend on ℓ or n. We then run, for i from ℓ to n, the procedure Enumerate$(\uparrow, q_0, i)$ (of course, if $L(A)$ contains strings of length i, which can be checked in $O(1)$ time by looking at the length of the default path leaving q_0, and seeing if this length is at least i). The delay when moving from one length to another is, clearly, constant, as we always end the enumeration of the strings of some length with a tail call and start the enumeration for the next length with the default path of that length, starting in q_0. ◀

G Ranking and Unraking

We recall the setting of the problem. The *rank* of a string w in a language is the number of strings smaller than w in the language under some ordering. The ranking problem requires computing the rank of a given word w.

The unranking operation is the reverse of the ranking operation, taking a number i as the input and asking for the string of rank i. The unranking problem requires computing the word of rank i.

In both cases, we consider the order induced by the enumeration algorithm of Section 4, and we want to show that these two problems can be solved in polynomial time.

As before, we assume that we have a PCA $A = (Q, \Sigma, q_0, Q, \delta)$, where $|Q| = m$ and $\Sigma = \{1, 2, \ldots, \sigma\}$.

We will keep the presentation in this section rather informal, as the technicalities are straightforward.

Recall the description from the main part of the paper, which is based on the enumeration algorithm. The main idea is that both ranking and unranking require identifying a path in the tree of recursive calls with root Enumerate$(\uparrow, q_0, n)$.

In the case of ranking, we identify the path corresponding to w, and the branching nodes occurring on it, and then count the total number of paths of length n corresponding to the leaves of subtrees of recursive calls occurring to the left of this path (assuming that the recursive calls made by an instance are ordered in the tree left to right according to their call-order). This can be done by running Enumerate$(\uparrow, q_0, n)$ and simply performing only the recursive calls that correspond to branching nodes on the path labelled with w, and retrieving the number of induced paths for those that should have been called before them.

In the case of unranking, one standard approach is to use the ranking procedure to determine the letters of the searched word one by one, or, alternatively, more efficiently, and closer to what we have done here so far, one can again run Enumerate$(\uparrow, q_0, n)$ and we make only those recursive calls which lead to the i^{th} path of length n, in the order of our enumeration.

Let us go now into more details.

We will use the following lemma.

▶ **Lemma 11** (Folklore). *Let $2 \leq \omega \leq 3$ be the exponent for matrix multiplication. Given the PCA A and a number n, we can compute the number of strings of length ℓ starting in a state q, for all $q \in Q$ and $\ell \leq n$, in $O(nm^{\omega})$ time.*

Since A is deterministic, the number of paths of length ℓ between two states q and q' can be retrieved from the matrix M^{ℓ}, where M is the $m \times m$ matrix which contains on the entry corresponding to q and q' the number of transitions between q and q', for all states q, q'. Then, using M^{ℓ}, with $\ell \leq n$, for each state q, we can compute the number of paths of length ℓ starting in q in $O(m)$ time.

Now, the ranking procedure works as follows. We run the PCA A on the input word w and obtain its decomposition $w = w_0 a_0 w_1 a_1 \ldots w_{t-1} a_{t-1} w_t$, where w_i is the label of a default path between the states q_i and q'_i, for $i \in \{0\} \cup [t]$, and a_{i-1} labels a non-default edge between q'_{i-1} and q_i, for $i \in [t]$. We store this decomposition, as well as the states q_i and q'_i, for $i \in \{0\} \cup [t]$. If w is the label of the default path of length n starting in q_0, then the rank of w is 1. Otherwise, we run Enumerate$(\uparrow, q_0, n)$ (without making any outputs) and use two integer variables *count*, set to 1 initially, and *total*, set to 0 initially; *count* keeps track of how many of the non-default transitions from the path with label w were met in our sequence of recursive calls (the non-default edges actually correspond one-to-one to these calls), and *total* keeps track of how many paths we have identified and counted already, that come before the one labelled with w in the enumeration. In this process, each time a call Enumerate(a, q, ℓ) should be made, we check first if it corresponds to the $count^{th}$ non-default edge of the path labelled with w (i.e., the transition from $q'_{count-1}$ to q_{count}, labelled with $a_{count-1}$). If yes, we perform that call. If not, we increase *total* by the number of paths of length ℓ originating in q. After the t^{th} call of Enumerate, we simply return $total + 1$ as the rank of w.

As our procedure Enumerate(a, q, ℓ) might end up going through every branching node on (q, ℓ) before making the next recursive call, the overall complexity of this algorithm (once the preprocessing is done) is $O(n^2\sigma)$. However, one can also reduce this factor $n^2\sigma$ to $n(n+\sigma)$ by noting that there is exactly one branching node for which we need to explore all the transitions leaving it. Indeed, we can check first for each branching node the total number of paths of desired length leaving from it (instead of going through each transition individually) and only go through the transitions of that node individually if this check indicates that our recursive call should be done for one of the non-default edges leaving the respective node.

The soundness of this approach follows from the fact that we basically use the non-default edges on the path labelled by w to traverse a root-to-leaf path of the tree of recursive calls started by Enumerate$(\uparrow, q_0, n)$, keeping track of the number of paths of length n of A which are discovered by calls occurring in the subtrees of the tree of recursive calls, which should have been done in the enumeration process before the calls we actually execute.

▶ **Theorem 9.** *Given a PCA A and a string $w \in \Sigma^n \cap L(A)$, we can compute the number of strings that are recognised by A and are output before w in our enumeration algorithm in $O(m\sigma + n(m^\omega + n + \sigma))$ time, where $2 \leq \omega \leq 3$ is the exponent for matrix multiplication.*

For unranking, one can use essentially the same approach. This time, we are given as input a number i. If $i = 1$, we simply return the default path of length n, starting in q_0. Otherwise, we run Enumerate$(\uparrow, q_0, n)$ (without making any outputs) and use one integer variable *total*, set to 0 initially. In this process, each time a call Enumerate(a, q, ℓ) should be made, we first sum up *total* and the number of paths of length ℓ originating in q. If this sum is strictly smaller than i, then we increase *total* by the number of paths of length ℓ originating in q and skip that call. Otherwise, if the sum is greater or equal to i, we make the recursive call. Each time a recursive call is made, we check if $i = \mathit{total} + 1$; if yes, we output the string represented on the stack $\mathcal{S}$, and then stop the process: we have identified the word of rank i. The correctness follows immediately, just as in the case of ranking: all is required to identify the word of rank i is a guided root-to-leaf traversal of the tree of recursive calls. The complexity of the algorithm is the same as in the case of the ranking algorithm (by the same arguments).

▶ **Theorem 10.** *Given an integer i, a PCA A, and an integer n, the i^{th} string w of length n output by enumerating A can be determined in $O(m\sigma + n(m^\omega + n + \sigma))$.*

The complexities listed in these two theorems do not take into account the time needed to do arithmetical operations on the numbers we work with (in particular, operations involving the variable *total*). To cover this, in the worst case, the final complexity is obtained by multiplying our complexities with $\frac{n \log \sigma}{\mathtt{w}}$, where $\mathtt{w}$ is the size of the memory word in our model. However, the overall complexity of both the ranking and the unranking algorithm stays polynomial.