Enumerating Regular Languages with Bounded Delay

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

Mikaël Monet
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

We study the task, for a given language L, of enumerating the (generally infinite) sequence of its words, without repetitions, while bounding the *delay* between two consecutive words. To allow for delay bounds that do not depend on the current word length, we assume a model where we produce each word by editing the preceding word with a small edit script, rather than writing out the word from scratch. In particular, this witnesses that the language is *orderable*, i.e., we can write its words as an infinite sequence such that the Levenshtein edit distance between any two consecutive words is bounded by a value that depends only on the language. For instance, $(a + b)^*$ is orderable (with a variant of the Gray code), but $a^* + b^*$ is not.

We characterize which regular languages are enumerable in this sense, and show that this can be decided in PTIME in an input deterministic finite automaton (DFA) for the language. In fact, we show that, given a DFA A, we can compute in PTIME automata $A_1, \ldots, A_t$ such that $\mathrm{L}(A)$ is partitioned as $\mathrm{L}(A_1) \sqcup \ldots \sqcup \mathrm{L}(A_t)$ and every $\mathrm{L}(A_i)$ is orderable in this sense. Further, we show that the value of t obtained is optimal, i.e., we cannot partition $\mathrm{L}(A)$ into less than t orderable languages.

In the case where $\mathrm{L}(A)$ is orderable (i.e., $t = 1$), we show that the ordering can be produced by a bounded-delay algorithm: specifically, the algorithm runs in a suitable pointer machine model, and produces a sequence of bounded-length edit scripts to visit the words of $\mathrm{L}(A)$ without repetitions, with bounded delay – exponential in $|A|$ – between each script. In fact, we show that we can achieve this while only allowing the edit operations *push* and *pop* at the beginning and end of the word, which implies that the word can in fact be maintained in a double-ended queue.

By contrast, when fixing the distance bound d between consecutive words and the number of classes of the partition, it is NP-hard in the input DFA A to decide if $\mathrm{L}(A)$ is orderable in this sense, already for finite languages.

Last, we study the model where push-pop edits are only allowed at the end of the word, corresponding to a case where the word is maintained on a stack. We show that these operations are strictly weaker and that the *slender languages* are precisely those that can be partitioned into finitely many languages that are orderable in this sense. For the slender languages, we can again characterize the minimal number of languages in the partition, and achieve bounded-delay enumeration.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Regular language, constant-delay enumeration, edit distance

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.8

Related Version *Full Version*: `https://arxiv.org/abs/2209.14878` [5]

Funding *Antoine Amarilli*: Partially supported by the ANR project EQUUS ANR-19-CE48-0019 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758.

Acknowledgements We thank Florent Capelli and Charles Paperman for their insights during initial discussions about this problem. We thank user pcpthm on the Theoretical Computer Science Stack Exchange forum for giving the argument for Proposition 6.1 in [18]. We thank Jeffrey Shallit for pointing us to related work. We thank Torsten Mütze and Arturo Merino for other helpful pointers. We thank the anonymous reviewers for their valuable feedback. Finally, we are grateful to Louis Jachiet and Lê Thành Dũng (Tito) Nguyễn for feedback on the draft.

© Antoine Amarilli and Mikaël Monet;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Enumeration algorithms [23, 26] are a way to study the complexity of problems beyond decision or function problems, where we must produce a large number of outputs without repetitions. In such algorithms, the goal is usually to minimize the worst-case *delay* between any two consecutive outputs. The best possible bound is to make the delay *constant*, i.e., independent from the size of the input. This is the case, for example, when enumerating the results of acyclic free-connex conjunctive queries [7] or of MSO queries over trees [6, 14].

Unfortunately, constant-delay is an unrealistic requirement when the objects to enumerate can have unbounded size, simply because of the time needed to write them out. Faced by this problem, one option is to neglect this part of the running time, e.g., following Ruskey's "Do not count the output principle" [21, p. 8]. In this work, we address this challenge in a different way: we study enumeration where each new object is not written from scratch but produced by *editing the previous object*, by a small sequence of edit operations called an *edit script*. This further allows us to study the enumeration of *infinite* collections of objects, with an algorithm that runs indefinitely and ensures that each object is produced after some finite number of steps, and exactly once. The size of the edit scripts must be *bounded*, i.e., it only depends on the collection of objects to enumerate, but not on the size of the current object. The algorithm thus outputs an infinite series of edit scripts such that applying them successively yields the infinite collection of all objects. In particular, the algorithm witnesses that the collection admits a so-called *ordering*: it can be ordered as an infinite sequence with a bound on the *edit distance* between any two consecutive objects, namely, the number of edit operations.

In this paper, we study enumeration for regular languages in this sense, with the Levenshtein edit distance and variants thereof. One first question is to determine if a given regular language L admits an ordering, i.e., can we order its words such that the Levenshtein distance of any two consecutive words only depends on L and not on the word lengths? For instance, the language a^* is easily orderable in this sense. The language a^*b^* is orderable, e.g., following any Hamiltonian path on the infinite $\mathbb{N} \times \mathbb{N}$ grid. More interestingly, the language $(a + b)^*$ is orderable, for instance by considering words by increasing length and using a *Gray code* [17], which enumerates all n-bit words by changing only one bit at each step. More complex languages such as $a(a + bc)^* + b(cb)^*ddd^*$ can also be shown to be orderable (as our results will imply). However, one can see that some languages are not orderable, e.g., $a^* + b^*$. We can nevertheless generalize orderability by allowing multiple "threads": then we can partition $a^* + b^*$ as a^* and b^*, both of which are orderable. This leads to several questions: Can we characterize the orderable regular languages? Can every regular language be partitioned as a finite union of orderable languages? And does this lead to a (possibly multi-threaded) enumeration algorithm with bounded delay (i.e., depending only on the language but not on the current word length)?

Contributions. The present paper gives an affirmative answer to these questions. Specifically, we show that, given a DFA A, we can decide in PTIME if $\mathrm{L}(A)$ is orderable. If it is not, we can compute in PTIME DFAs $A_1, \ldots, A_t$ partitioning the language as $\mathrm{L}(A) = \mathrm{L}(A_1) \sqcup \ldots \sqcup \mathrm{L}(A_t)$ such that each $\mathrm{L}(A_i)$ is orderable; and we show that the t given in this construction is optimal, i.e., no smaller such partition exists. If the language *is* orderable (i.e., if $t = 1$), we show in fact that the same holds for a much more restricted notion of distance, the *push-pop distance*, which only allows edit operations at the beginning and end of the word. The reason we are interested in this restricted edit distance is that edit scripts featuring push and pop can be

easily applied in constant-time to a word represented in a double-ended queue; by contrast, Levenshtein edit operations are more difficult to implement, because they refer to integer word positions that change whenever characters are inserted or deleted.[1]

And indeed, this result on the push-pop distance then allows us to design a bounded-delay algorithm for $\mathrm{L}(A)$, which produces a sequence of bounded edit scripts of push or pop operations that enumerates $\mathrm{L}(A)$. The length of the edit scripts is polynomial in $|A|$ and the delay of our algorithm is exponential in $|A|$, but crucially it remains bounded throughout the (generally infinite) execution of the algorithm, and does not depend on the size of the words that are achieved. Formally, we show:

▶ **Result 1.** *Given a DFA A, one can compute in PTIME automata $A_1, \ldots, A_t$ for some $t \leqslant |A|$ such that $\mathrm{L}(A)$ is the disjoint union of the $\mathrm{L}(A_i)$, and we can enumerate each $\mathrm{L}(A_i)$ with bounded delay for the push-pop distance with distance bound $48|A|^2$ and exponential delay in $|A|$. Further, $\mathrm{L}(A)$ has no partition of cardinality $t-1$ into orderable languages, even for the Levenshtein distance.*

Thus, we show that orderability and enumerability, for the push-pop or Levenshtein edit distance, are in fact all logically equivalent on regular languages, and we characterize them (and find the optimal partition cardinality) in PTIME. By contrast, as was pointed out in [18], testing orderability for a fixed distance d is NP-hard in the input DFA, even for finite languages.

Last, we study the *push-pop-right distance*, which only allows edits at the end of the word. The motivation for studying this distance is that it corresponds to enumeration algorithms in which the word is maintained on a stack. We show that, among the regular languages, the *slender languages* [19] are then precisely those that can be partitioned into finitely many orderable languages, and that these languages are themselves enumerable. Further, the optimal cardinality of the partition can again be computed in PTIME:

▶ **Result 2.** *Given a DFA A, then $\mathrm{L}(A)$ is partitionable into finitely many orderable languages for the push-pop-right distance if and only if $\mathrm{L}(A)$ is slender (which we can test in PTIME in A). Further, in this case, we can compute in PTIME the smallest partition cardinality, and each language in the partition is enumerable with bounded delay with distance bound $2|A|$ and linear delay in $|A|$.*

In terms of proof techniques, our PTIME characterization of Result 1 relies on a notion of *interchangeability* of automaton states, defined via paths between states and via states having common loops. We then show orderability by establishing *stratum-connectivity*, i.e., for any *stratum* of words of the language within some length interval, there are finite sequences obeying the distance bound that connect any two words in that stratum. We show stratum-connectivity by pumping and de-pumping loops close to the word endpoints. We then deduce an ordering from this by adapting a standard technique [25] of visiting a spanning tree and enumerating even and odd levels in alternation (see also [22, 13]). The bounded-delay enumeration algorithm then proceeds by iteratively enlarging a custom data structure called a *word DAG*, where the construction of the structure for a stratum is amortized by enumerating the edit scripts to achieve the words of the previous stratum.

[1] There is, in fact, an $\Omega(\log |w| / \log \log |w|)$ lower bound on the complexity of applying Levenshtein edit operations and querying which letter occurs at a given position: crucially, this bound depends on the size of the word. See `https://cstheory.stackexchange.com/q/46746` for details. This is in contrast to the application of push-pop-right edit operations, which can be performed in constant time (independent from the word length) when the word is stored in a double-ended queue.

Related work. As we explained, enumeration has been extensively studied for many structures [26]. For regular languages specifically, some authors have studied the problem of enumerating their words in *radix order* [15, 1, 2, 10]. For instance, the authors of [1, 2] provide an algorithm that enumerates all words of a regular language in that order, with a delay of $O(|w|)$ for w the next word to enumerate. Thus, this delay is not bounded, and the requirement to enumerate in radix order makes it challenging to guarantee a bounded distance between consecutive words (either in the Levenshtein or push-pop distance), which is necessary for bounded-delay enumeration in our model. Indeed, our results show that not all regular languages are orderable in our sense, whereas their linear-delay techniques apply to all regular languages.

We have explained that enumeration for $(a + b)^*$ relates to Gray codes, of which there exist several variants [17]. Some variants, e.g., the so-called *middle levels problem* [16], aim at enumerating binary words of a restricted form; but these languages are typically finite (i.e., words of length n), and their generalization is typically not regular. While Gray codes typically allow arbitrary substitutions, one work has studied a variant that only allows restricted operations on the endpoints [9], implying the push-pop orderability of the specific language $(a + b)^*$.

Independently, some enumeration problems on automata have been studied recently in the database theory literature, in particular for *document spanners* [8], which can be defined by finite automata with capture variables. It was recently shown [11, 3] that we can enumerate in constant delay all possible assignments of the capture variables of a fixed spanner on an input word. In these works, the delay is constant in *data complexity*, which means that it only depends on the (fixed) automaton, and does not depend on the word; this matches what we call *bounded delay* in our work (where there is no input word and the automaton is given as input). However, our results do not follow from these works, which focus on the enumeration of results of constant size. Some works allow second-order variables and results of non-constant size [4] but the delay would then be linear in each output, hence unbounded.

Paper structure. We give preliminaries in Section 2. In Section 3 we present our PTIME construction of a partition of a regular language into finitely many orderable languages, and prove that the cardinality of the obtained partition is minimal for orderability. We then show in Section 4 that each term of the union is orderable, and then that it is enumerable in Section 5. We present the NP-hardness result on testing orderability for a fixed distance and our results on push-pop-right operations in Section 6. We conclude and mention some open problems in Section 7. Due to space constraints, we mostly present the general structure of the proofs and give the main ideas; detailed proofs of all statements can be found in the extended version of this work [5].

2 Preliminaries

We fix a finite non-empty *alphabet* Σ of *letters*. A *word* is a finite sequence $w = a_1 \cdots a_n$ of letters. We write $|w| = n$, and write ϵ for the empty word. We write Σ^* the infinite set of words over Σ. A *language* L is a subset of Σ^*. For $k \in \mathbb{N}$, we denote $L^{<k}$ the language $\{w \in L \mid |w| < k\}$. In particular we have $L^{<0} = \varnothing$.

In this paper we study *regular languages*. Recall that such a language can be described by a *deterministic finite automaton* (DFA) $A = (Q, \Sigma, q_0, F, \delta)$, which consists of a finite set Q of *states*, an *initial state* $q_0 \in Q$, a set $F \subseteq Q$ of *final states*, and a partial *transition function* $\delta \colon Q \times \Sigma \to Q$. We write $|A|$ the size of representing A, which is $O(|Q| \times |\Sigma|)$. A *(directed)*

path in A from a state $q \in Q$ to a state $q' \in Q$ is a sequence of states $q = q_0, \ldots, q_n = q'$ where for each $0 \leqslant i < n$ we have $q_{i+1} = \delta(q_i, a_i)$ for some a_i. For a suitable choice $a_0, \ldots, a_{n-1}$, we call the word $a_0 \cdots a_{n-1} \in \Sigma^*$ a *label* of the path. In particular, there is an empty path with label ϵ from every state to itself. The *language* $\mathrm{L}(A)$ accepted by A consists of the words w that label a path from q_0 to some final state. We assume without loss of generality that all automata are *trimmed*, i.e., every state of Q has a path from q_0 and has a path to some final state; this can be enforced in linear time.

Edit distances. For an alphabet Σ, we denote by $\delta_{\mathrm{Lev}} \colon \Sigma^* \times \Sigma^* \to \mathbb{N}$ the *Levenshtein edit distance*: given $u, v \in \Sigma^*$, the value $\delta_{\mathrm{Lev}}(u, v)$ is the minimum number of *edits* needed to transform u into v, where the edit operations are single-letter *insertions*, *deletions* or *substitutions* (we omit their formal definitions).

While our lower bounds hold for the Levenshtein distance, our positive results already hold with a restricted set of $2|\Sigma| + 2$ edit operations called the *push-pop edit operations*: $\mathrm{pushL}(a)$ and $\mathrm{pushR}(a)$ for $a \in \Sigma$, which respectively insert a at the beginning and at the end of the word, and $\mathrm{popL}()$ and $\mathrm{popR}()$, which respectively remove the first and last character of the word (and cannot be applied if the word is empty). Thus, we define the *push-pop edit distance*, denoted δ_{pp}, like δ_{Lev} but allowing only these edit operations.

Orderability. Fixing a distance function $\delta \colon \Sigma^* \times \Sigma^* \to \mathbb{N}$ over Σ^*, for a language $L \subseteq \Sigma^*$ and $d \in \mathbb{N}$, a *d-sequence in L* is a (generally infinite) sequence $\boldsymbol{s}$ of words $w_1, \ldots, w_n, \ldots$ of L *without repetition*, such that for every two consecutive words w_i, w_{i+1} in $\boldsymbol{s}$ we have $\delta(w_i, w_{i+1}) \leqslant d$. We say that $\boldsymbol{s}$ *starts at* w_1 and, in case $\boldsymbol{s}$ is finite and has n elements, that $\boldsymbol{s}$ *ends at* w_n (or that $\boldsymbol{s}$ is *between* w_1 *and* w_n). A *d-ordering of L* is a d-sequence $\boldsymbol{s}$ in L such that every word of L occurs in $\boldsymbol{s}$; equivalently, it is a permutation of L such that any two consecutive words are at distance at most d. An *ordering* is a d-ordering for some $d \in \mathbb{N}$. If these exist, we call the language L, respectively, *d-orderable* and *orderable*. We call L *(t, d)-partition-orderable* if it can be partitioned into t languages that each are d-orderable:

▶ **Definition 2.1.** *Let L be a language and $t, d \in \mathbb{N}$. We call L* (t, d)-partition-orderable *if L has a partition $L = \bigsqcup_{1 \leqslant i \leqslant t} L_i$ such that each L_i is d-orderable.*[2]

Note that, if we allowed repetitions in d-orderings, then the language of any DFA A would be $O(|A|)$-orderable: indeed, any word w can be transformed into a word w' of length $O(|A|)$ by iteratively removing simple loops in the run of w. By contrast, we will see in Section 3 that allowing a *constant* number of repetitions of each word makes no difference.

▶ **Example 2.2.** We consider the Levenshtein distance in this example. The language $(aa)^*$ is $(1, 2)$-partition-orderable (i.e., 2-orderable) and not $(k, 1)$-partition-orderable for any $k \in \mathbb{N}$. The language $a^* + b^*$ is $(2, 1)$-partition-orderable and not orderable, i.e., not d-orderable for any $d \in \mathbb{N}$. Any finite language is d-orderable with d the maximal length of a word in L. The non-regular language $\{a^{n^2} \mid n \in \mathbb{N}\}$ is not (t, d)-partition-orderable for any $t, d \in \mathbb{N}$.

Enumeration algorithms. We study *enumeration algorithms*, which output a (generally infinite) sequence of *edit scripts* $\sigma_1, \sigma_2, \ldots$. We only study enumeration algorithms where each *edit script* σ_i is a finite sequence of push-pop edit operations. The algorithm enumerates a language L if the sequence satisfies the following condition: letting w_1 be the result of

[2] We use $\bigsqcup$ for disjoint unions.

applying σ_1 on the empty word, w_2 be the result of applying σ_2 to w_1, and so on, then all w_i are distinct and $L = \{w_1, w_2, \ldots\}$. If L is infinite then the algorithm does not terminate, but the infinite sequence ensures that every $w \in L$ is produced as the result of applying (to ϵ) some finite prefix $\sigma_1, \ldots, \sigma_n$ of the output.

We aim for *bounded-delay* algorithms, i.e., each edit script must be output in time that only depends on the language L that is enumerated, but not on the current length of the words. Formally, the algorithm can emit any push-pop edit operation and a delimiter Output, it must successively emit the edit operations of σ_i followed by Output, and there is a bound $T > 0$ (the delay) depending only on L such that the first Output is emitted at most T operations after the beginning of the algorithm, and for each $i > 1$ the i-th Output is emitted at most T operations after the $(i-1)$-th Output. Note that our notion of delay also accounts for what is usually called the preprocessing phase in the literature, i.e., the phase before the first result is produced. Crucially the words w_i obtained by applying the edit scripts σ_i are not written, and T does not depend on their length.

We say that a bounded-delay algorithm *d-enumerates* a language L if it produces a d-ordering of L (for the push-pop distance). Thus, if L is d-enumerable (by an algorithm), then L is in particular d-orderable, and we will show that for regular languages, the converse also holds.

▶ **Example 2.3.** Consider the regular language $L := a^*b^* + b^*a^*$. This language is 2-orderable for the push-pop distance. Indeed, we can order it by increasing word length, finishing for word length i by the word a^i as follows. We start by length zero with the empty word ϵ (so the first edit script is empty), then, assuming we have ordered all words of L of size $\leqslant i$ while finishing with a^i, we continue with words of L of size $i+1$ in the following manner: we push-right the letter b to obtain a^ib, and then we "shift" with edit scripts of the form $(\mathrm{pushR}(b); \mathrm{popL}())$ until we obtain b^{i+1}, and then we shift again with edit scripts of the form $(\mathrm{pushR}(a); \mathrm{popL}())$ until we obtain a^{i+1} as promised. This gives us an enumeration algorithm for L, shown in Algorithm 1. As such, the delay of Algorithm 1 is not bounded, because of the time needed to increment the integer variable *size*: this variable becomes arbitrarily large throughout the enumeration, so it is not realistic to assume that we can increment it in constant time. This can however be fixed by working in a suitable *pointer machine model*, as explained next.

Note that our enumeration algorithms run indefinitely, and thus use unbounded memory: this is unavoidable because their output would necessarily be ultimately periodic otherwise, which is not suitable in general. To avoid specifying the size of memory cells or the complexity of arithmetic computations (e.g., incrementing the integer *size* in Algorithm 1), we consider a different model called *pointer machines* [24] which only allows arithmetic on a bounded domain. We use this model for our enumeration algorithms (but not, e.g., our other complexity results such as PTIME bounds).

Intuitively, a pointer machine works with *records* consisting of a constant number of labeled *fields* holding either *data values* (in our case of constant size, i.e., constantly many possible values) or *pointers* (whose representation is not specified). The machine has *memory* consisting of a finite but unbounded collection of records, a constant number of which are designated as *registers* and are always accessible. The machine can allocate records in constant time, retrieving a pointer to the memory location of the new record. We can access the fields of records, read or write pointers, dereference them, and test them for equality, all in constant time, but we cannot perform any other manipulation on pointers or other arithmetic operations. (We can, however, count in unary with a linked list, or perform arbitrary operations on the constant-sized data values.) See the full version [5] for details.

Algorithm 1 Push-pop enumeration algorithm for the language $a^*b^* + b^*a^*$ from Example 2.3.

```
// The first edit script is empty, corresponding to the empty word.
Output;
int size = 0;
while true do
    size++;
    pushR(b) ; Output;
    for int j = 0; j < size − 1; j++ do
        pushR(b) ; popL() ; Output;
    end
    for int j = 0; j < size; j++ do
        pushR(a) ; popL() ; Output;
    end
end
```

▶ **Example 2.4.** Continuing Example 2.3, Algorithm 1 can easily be adapted to a pointer-machine algorithm that 2-enumerates L, maintaining the word in a double-ended queue (deque) and keeping pointers to the first and last positions in order to know when to stop the **for** loops. Deques can indeed be simulated in this machine model, e.g., with linked lists.

3 Interchangeability partition and orderability lower bound

In this section, we start the proof of our main result, Result 1. Let A be the DFA and let Q be its set of states. The result is trivial if the language $\mathrm{L}(A)$ is finite, as we can always enumerate it naively with distance $O(|A|)$ and some arbitrary delay bound, so in the rest of the proof we assume that $\mathrm{L}(A)$ is infinite.

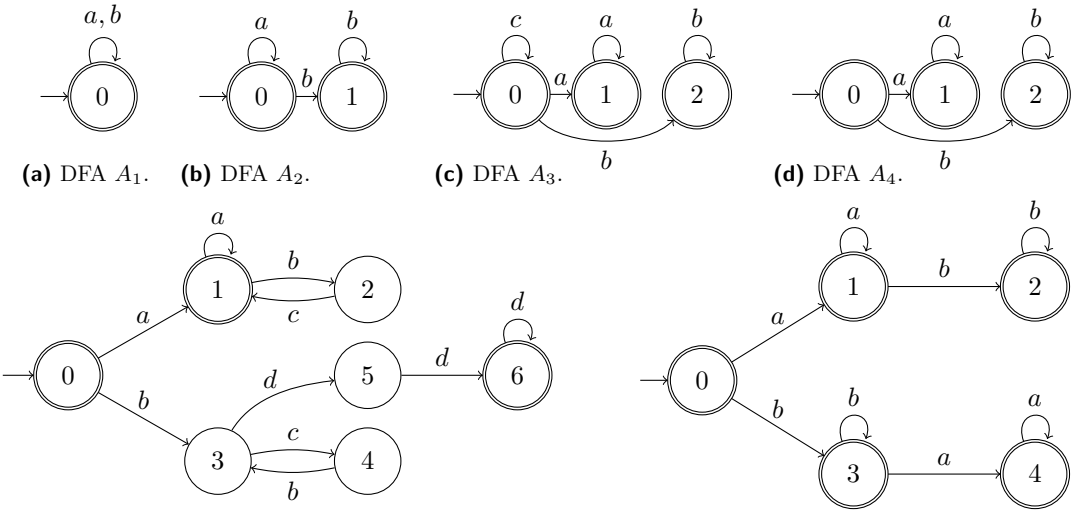
We will first define a notion of *interchangeability* on DFAs by introducing the notions of *connectivity* and *compatibility* on DFA states (this notion will be used in the next section to characterize orderability). We then partition $\mathrm{L}(A)$ into languages $\mathrm{L}(A_1) \sqcup \cdots \sqcup \mathrm{L}(A_t)$ following a so-called *interchangeability partition*, with each A_i having this interchangeability property. Last, we show in the section our lower bound establishing that t is optimal.

Interchangeability. To define our notion of interchangeability, we first define the *loopable* states of the DFA as those that are part of a non-empty cycle (possibly a self-loop):

▶ **Definition 3.1.** *For a state $q \in Q$, we let A_q be the DFA obtained from A by setting q as the only initial and final state. We call q* loopable *if $\mathrm{L}(A_q) \neq \{\epsilon\}$, and* non-loopable *otherwise.*

We then define the *interchangeability* relation on loopable states as the transitive closure of the union of two relations, called *connectivity* and *compatibility*:

▶ **Definition 3.2.** *We say that two loopable states q and q' are* connected *if there is a directed path from q to q', or from q' to q. We say that two loopable states q, q' are* compatible *if $\mathrm{L}(A_q) \cap \mathrm{L}(A_{q'}) \neq \{\epsilon\}$. These two relations are symmetric and reflexive on loopable states. We then say that two loopable states q and q' are* interchangeable *if they are in the transitive closure of the union of the connectivity and compatibility relations. In other words, q and q'*



(a) DFA A_1. **(b)** DFA A_2. **(c)** DFA A_3. **(d)** DFA A_4.

(e) DFA A_5. **(f)** DFA A_6.

Figure 1 Example DFAs from Example 3.4.

are interchangeable if there is a sequence $q = q_0, \dots, q_n = q'$ of loopable states such that for any $0 \leqslant i < n$, the states q_i and q_{i+1} are either connected or compatible. Interchangeability is then an equivalence relation over loopable states.

Note that if two loopable states q, q' are in the same strongly connected component (SCC) of A then they are connected, hence interchangeable. Thus, we can equivalently see the interchangeability relation at the level of SCCs (excluding those that do not contain a loopable state, i.e., excluding the trivial SCCs containing only one state having no self-loop).

▶ **Definition 3.3.** *We call* classes of interchangeable states*, or simply* classes*, the equivalence classes of the interchangeability relation. Recall that, as* $\mathrm{L}(A)$ *is infinite, there is at least one class. We say that the DFA A is* interchangeable *if the partition has only one class, in other words, if all loopable states of A are interchangeable.*

▶ **Example 3.4.** The DFA A_1 shown in Figure 1a for the language $(a + b)^*$ has only one loopable state, so A_1 is interchangeable.

The DFA A_2 shown in Figure 1b for the language a^*b^* has two loopable states 0 and 1 which are connected, hence interchangeable. Thus, A_2 is interchangeable.

The DFA A_3 shown in Figure 1c for the language $c^*(a^* + b^*)$ has three loopable states: 0, 1 and 2. The states 0 and 1 are connected, and 0 and 2 are also connected, so all loopable states are interchangeable and A_3 is interchangeable.

The DFA A_4 shown in Figure 1d for the language $a^* + b^*$ has two loopable states 1 and 2 which are neither connected nor compatible. So A_4 is not interchangeable.

The DFA A_5 shown in Figure 1e for the language $a(a + bc)^* + b(cb)^*ddd^*$ mentioned in the introduction has five loopable states: 1, 2, 3, 4, and 6. Then 1 and 2 are connected, 3 and 4 are connected, 3 and 6 are connected, and 1 and 4 are compatible (with the word bc). Hence, all loopable states are interchangeable and A_5 is interchangeable.

The DFA A_6 shown in Figure 1f for the language $a^*b^* + b^*a^*$ from Example 2.3 has four loopable states: 1, 2, 3, and 4. Then 1 and 2 are connected, 3 and 4 are connected, and (for instance) 1 and 4 are compatible (with the word a). Hence all loopable states are interchangeable and A_6 is interchangeable.

Interchangeability partition. We now partition $\mathrm{L}(A)$ using interchangeable DFAs:

▶ **Definition 3.5.** *An* interchangeability partition *of A is a sequence $A_1, \ldots, A_t$ of DFAs such that $\mathrm{L}(A)$ is the disjoint union of the $\mathrm{L}(A_i)$ and every A_i is interchangeable. Its* cardinality *is the number t of DFAs.*

Let us show how to compute an interchangeability partition whose cardinality is the number of classes. We will later show that this cardinality is optimal. Here is the statement:

▶ **Proposition 3.6.** *We can compute in polynomial time in A an interchangeability partition $A_1, \ldots, A_t$ of A, with $t \leqslant |A|$ the number of classes of interchangeable states.*

Intuitively, the partition is defined following the classes of A. Indeed, considering any word $w \in \mathrm{L}(A)$ and its accepting run ρ in A, for any loopable state q and q' traversed in ρ, the word w witnesses that q and q' are connected, hence interchangeable. Thus, we would like to partition the words of $\mathrm{L}(A)$ based on the common class of the loopable states traversed in their accepting run. The only subtlety is that $\mathrm{L}(A)$ may also contain words whose accepting run does not traverse any loopable state, called *non-loopable words*. For instance, ϵ is a non-loopable word of $\mathrm{L}(A_5)$ for A_5 given in Figure 1e. Let us formally define the non-loopable words, and our partition of the loopable words based on the interchangeability classes:

▶ **Definition 3.7.** *A word $w = a_1 \cdots a_n$ of $\mathrm{L}(A)$ is* loopable *if, considering its accepting run $q_0, \ldots, q_n$ with q_0 the initial state and $q_i = \delta(q_{i-1}, a_i)$ for $1 \leqslant i \leqslant n$, one of the q_i is loopable. Otherwise, w is* non-loopable*. We write $\mathrm{NL}(A)$ the set of the non-loopable words of $\mathrm{L}(A)$.*

Letting $\mathcal{C}$ be a class of interchangeable states, we write $\mathrm{L}(A, \mathcal{C})$ the set of (loopable) words of $\mathrm{L}(A)$ whose accepting run traverses a state of $\mathcal{C}$.

We then have the following, with finiteness of $\mathrm{NL}(A)$ shown by the pigeonhole principle:

▷ Claim 3.8. The language $\mathrm{L}(A)$ can be partitioned as $\mathrm{NL}(A)$ and $\mathrm{L}(A, \mathcal{C}_1), \ldots, \mathrm{L}(A, \mathcal{C}_t)$ over the classes $\mathcal{C}_1, \ldots, \mathcal{C}_t$ of interchangeable states, and further $\mathrm{NL}(A)$ is finite.

We now construct an interchangeability partition of A of the right cardinality by defining one DFA A_i for each class of interchangeable states, where we simply remove the loopable states of the other classes. These DFAs are interchangeable by construction. We modify the DFAs to ensure that the non-loopable words are only captured by A_1. This construction (explained in the full version [5]) is doable in PTIME, in particular the connectivity and compatibility relations can be computed in PTIME, testing compatibility by checking the nonemptiness of product automata. This establishes Proposition 3.6.

Lower bound. We have shown how to compute an interchangeability partition of a DFA A with cardinality the number t of classes. Let us now show that this value of t is optimal, in the sense that $\mathrm{L}(A)$ cannot be partitioned into less than t orderable (even non-regular) languages. This lower bound holds even when allowing Levenshtein edits. Formally:

▶ **Theorem 3.9.** *For any partition of the language $\mathrm{L}(A)$ as $\mathrm{L}(A) = L_1 \sqcup \cdots \sqcup L_{t'}$ if for each $1 \leqslant i \leqslant t'$ the language L_i is orderable for the Levenshtein distance, then we have $t' \geqslant t$ for t the number of classes of A.*

This establishes the negative part of Result 1. Incidentally, this lower bound can also be shown even if the unions are not disjoint, indeed even if we allow repetitions, provided that there is some constant bound on the number of repetitions of each word.

Theorem 3.9 can be shown from the following claim which establishes that sufficiently long words from different classes are arbitrarily far away for the Levenshtein distance:

▶ **Proposition 3.10.** *Letting $\mathcal{C}_1, \ldots, \mathcal{C}_t$ be the classes of A, for any distance $d \in \mathbb{N}$, there is a threshold $l \in \mathbb{N}$ such that for any two words $u \in \mathrm{L}(A, \mathcal{C}_i)$ and $v \in \mathrm{L}(A, \mathcal{C}_j)$ with $i \neq j$ and $|u| \geqslant l$ and $|v| \geqslant l$, we have $\delta_{\mathrm{Lev}}(u, v) > d$.*

This proposition implies Theorem 3.9 because, if we could partition $\mathrm{L}(A)$ into less than t orderable languages, then some ordering must include infinitely many words from two different classes $\mathrm{L}(A, \mathcal{C}_i)$ and $\mathrm{L}(A, \mathcal{C}_j)$, hence alternate infinitely often between the two. Fix the distance d, and consider a point when all words of L of length $\leqslant \max(l, \max_{w \in \mathrm{NL}(A)} |w|)$ have been enumerated, for l the threshold of the proposition: then it is no longer possible for any ordering to move from one class to another, yielding a contradiction. As for the proof of Proposition 3.10, we give a sketch below (the complete proofs are in the full version [5]):

Proof sketch. Given a sufficiently long word $u \in \mathrm{L}(A, \mathcal{C}_i)$, by the pigeonhole principle its run must contain a large number of loops over some state $q \in \mathcal{C}_i$. Assume that we can edit u into $v \in \mathrm{L}(A, \mathcal{C}_j)$ with d edit operations: this changes at most d of these loops. Now, considering the accepting run of v and using the pigeonhole principle again on the sequence of endpoints of contiguous unmodified loops, we deduce that some state q' occurs twice; then $q' \in \mathcal{C}_j$ by definition of $\mathrm{L}(A, \mathcal{C}_j)$. The label of the resulting loop on q' is then also the label of a loop on q, so q and q' are compatible, hence $\mathcal{C}_i = \mathcal{C}_j$. ◀

4 Orderability upper bound

We have shown in the previous section that we could find an interchangeability partition of any regular language $\mathrm{L}(A)$ into languages $\mathrm{L}(A_1), \ldots, \mathrm{L}(A_t)$ of interchangeable DFAs, for t the number of classes. We know by our lower bound (Theorem 3.9) that we cannot hope to order $\mathrm{L}(A)$ with less than t sequences. Thus, in this section, we focus on each interchangeable A_i separately, and show how to order $\mathrm{L}(A_i)$ as one sequence. Hence, we fix for this section a DFA A that is interchangeable, write k its number of states, and show that $\mathrm{L}(A)$ is orderable. We will in fact show that this is the case for the push-pop distance:

▶ **Theorem 4.1.** *For any interchangeable DFA A, the language $\mathrm{L}(A)$ is $48k^2$-orderable for the push-pop distance.*

We show this result in the rest of this section, and strengthen it in the next section to a bounded-delay algorithm. Before starting, we give an overview of the structure of the proof. The proof works by first introducing *d-connectivity* of a language (not to be confused with the connectivity relation on loopable automaton states). This weaker notion is necessary for d-orderability, but for finite languages we will show a kind of converse: d-connectivity implies $3d$-orderability. We will then show that $\mathrm{L}(A)$ is *stratum-connected*, i.e., the finite *strata* of words of $\mathrm{L}(A)$ in some length interval are each d-connected for some common d. Last, we will show show that this implies orderability, using the result on finite languages.

Connectivity implies orderability on finite languages. We now define *d-connectivity*:

▶ **Definition 4.2.** *A language L is* d-connected *if for every pair of words $u, v \in L$, there exists a d-sequence in L between u and v.*

Clearly d-connectivity is a necessary condition for d-orderability: indeed if $w_1, w_2, \ldots$ is a d-ordering of L, and $u = w_i$, $v = w_j$ are two words of L with $i \leqslant j$ (without loss of generality), then $w_i, w_{i+1}, \ldots, w_j$ is indeed a d-sequence in L between u and v. What is more, for finite languages, the converse holds, up to multiplying the distance by a constant factor:

▶ **Lemma 4.3.** *Let L be a finite language that is d-connected and $s \neq e$ be words of L. Then there exists a $3d$-ordering of L starting at s and ending at e.*

Proof sketch. We use the fact, independently proved by Sekanina and by Karaganis [22, 13], that the cube of every connected graph G has a Hamiltonian path between any pair of vertices (see also [17]). One algorithmic way to see this is by traversing a spanning tree of G and handling odd-depth and even-depth nodes in prefix and postfix fashion (see, e.g., [25]). Applying this to the graph G whose vertices are the words of L and where two words w, w' are connected by an edge when $\delta(w, w') \leqslant d$ yields the result. ◀

The constant 3 in this lemma is optimal, as follows from [20]; see the full version [5] for more details. Note that the result does not hold for infinite languages: $a^* + b^*$ is 1-connected (via ϵ) but not d-orderable for any d.

Stratum-connectivity. To show orderability for infinite languages, we will decompose them into *strata*, which simply contain the words in a certain length range. Formally:

▶ **Definition 4.4.** *Let L be a language, let $\ell > 0$ be an integer, and let $i > 0$. The* i-th stratum of width ℓ *(or* ℓ-stratum*) of L, written* $\mathrm{strat}_\ell(L, i)$*, is $L^{<i\ell} \backslash L^{<(i-1)\ell}$.*

We will show that, for the language $\mathrm{L}(A)$ of our interchangeable DFA A, we can pick ℓ and d such that every ℓ-stratum of $\mathrm{L}(A)$ is d-connected, i.e., $\mathrm{L}(A)$ is (ℓ, d)-*stratum-connected*:

▶ **Definition 4.5.** *Let L be a regular language and fix $\ell, d > 0$. We say that L is* (ℓ, d)-stratum-connected *if every ℓ-stratum* $\mathrm{strat}_\ell(L, i)$ *is d-connected.*

Note that our example language $a^* + b^*$, while 1-connected, is not (ℓ, d)-stratum-connected for any ℓ, d, because any i-th ℓ-stratum for $i > d$ is not d-connected. We easily show that stratum-connectivity implies orderability:

▶ **Lemma 4.6.** *Let L be an infinite language recognized by a DFA with k' states, and assume that L is (ℓ, d)-stratum-connected for some $\ell \geqslant 2k'$ and some $d \geqslant 3k'$. Then L is $3d$-orderable.*

Proof sketch. We show by pumping that we can move across contiguous strata. Thus, we combine orderings on each stratum obtained by Lemma 4.3 with well-chosen endpoints. ◀

We can then show using several pumping and de-pumping arguments that the language of our interchangeable DFA A is (ℓ, d)-stratum-connected for $\ell := 8k^2$ and $d := 16k^2$.

▶ **Proposition 4.7.** *The language $\mathrm{L}(A)$ is $(8k^2, 16k^2)$-stratum-connected.*

Proof sketch. As there are only a finite number of non-loopable words, we focus on loopable words. Consider a stratum S and two loopable words u and v of S. Their accepting runs involve loopable states, respectively q and q', that are interchangeable because A is. We first show that u is d-connected (in S) to a *normal form*: a repeated loop on q plus a prefix and suffix whose length is bounded, i.e., only depends on the language. We impose this in two steps: first we move the last occurrence of q in u near the end of the word by pumping at the left end and de-pumping at the right end, second we pump the loop on q at the right end while de-pumping the left end. This can be done while remaining in the stratum S. We obtain similarly a normal form consisting of a repeated loop on q' with bounded-length prefix and suffix that is d-connected to v in S.

Then we do an induction on the number of connectivity and compatibility relations needed to witness that q and q' are interchangeable. If $q = q'$, we conclude using the normal forms of u and v. If q is connected to q', we impose the normal form on u, then we modify

it to a word whose accepting run also visits q', and we apply the previous case. If q is compatible with q', we conclude using the normal form with some loop label z in $A_q \cap A_{q'}$ (of length $\leqslant k^2$) that witnesses their compatibility. The induction case is then easy. ◀

From this, we deduce with Lemma 4.6 that $\mathrm{L}(A)$ is $48k^2$-orderable, so Theorem 4.1 holds. Note that the construction ensures that the words are ordered stratum after stratum, so "almost" by increasing length: in the ordering that we obtain, after producing some word w, we will never produce words of length less than $|w| - \ell$.

5 Bounded-delay enumeration

In this section, we show how the orderability result of the previous section yields a bounded-delay algorithm. We use the pointer-machine model from Section 2, which we modify for convenience to allow data values and the number of fields of records to be exponential in the automaton (but fixed throughout the enumeration, and independent on the size of words): see the full version [5] for more explanations. We show:

▶ **Theorem 5.1.** *There is an algorithm which, given an interchangeable DFA A with k states, enumerates the language $\mathrm{L}(A)$ with push-pop distance bound $48k^2$ and exponential delay in $|A|$.*

Let us accordingly fix the interchangeable DFA A with k states. Following Proposition 4.7, we let $d := 16k^2$ and $\ell := 8k^2$.

Overall amortized scheme. The algorithm will run two processes in parallel: the first process simply enumerates a previously prepared sequence of edit scripts that gives a $3d$-ordering of some stratum, while the second process computes the sequences for subsequent strata (and of course imposing that the endpoints of the sequences for contiguous strata are sufficiently close). We initialize this by computing in an arbitrary way a $3d$-ordering for the first stratum.

The challenging part is to prepare efficiently the sequences for all strata, and in particular to build a data structure that represents the strata. We will require of our algorithm that it processes each stratum in *amortized linear time* in its size. Formally, letting $N_j := |\mathrm{strat}_\ell(L, j)|$ be the number of words of the j-th stratum for all $j \geqslant 1$, there is a value $C \in \mathbb{N}$ that is exponential in $|A|$ such that, after having run for $C \sum_{j=1}^{i} N_j$ steps, the algorithm is done processing the i-th stratum. Note that this is weaker than processing each separate stratum in linear time: the algorithm can go faster to process some strata and spend this spared time later so that some later strata are processed arbitrarily slowly relative to their size.

If we can achieve amortized linear time, then the overall algorithm runs with bounded delay. To see why, notice that the prepared sequence for the i-th stratum has length at least its size N_i, and we can show that the size N_{i+1} of the next stratum is within a factor of N_i that only depends on L (this actually holds for any infinite regular language and does not use interchangeability):

▶ **Lemma 5.2.** *Letting $C_A := (k+1)|\Sigma|^{\ell+k+1}$, for all $i \geqslant 1$ we have $N_i/C_A \leqslant N_{i+1} \leqslant C_A N_i$.*

Proof. Each word in the $(i+1)$-th stratum of L can be transformed into a word in the i-th stratum as follows: letting k be the number of DFA states, first remove a prefix of length at most $\ell + k$ to get a word (not necessarily in L) of length $i\ell - k - 1$, and then add back a prefix corresponding to some path of length $\leqslant k$ from the initial state to get a word in the

i-th stratum of L as desired. Now, for any word w of the i-th stratum, the number of words of the $(i+1)$-th stratum that lead to w in this way is bounded by C_A, by considering the reverse of this rewriting, i.e., all possible ways to rewrite w by removing a prefix of length at most k and then adding a prefix of length at most $\ell + k$. A simple union bound gives $N_{i+1} \leqslant C_A N_i$. Now, a similar argument in the other direction gives $N_i / C_A \leqslant N_{i+1}$. ◀

Thanks to this lemma, it suffices to argue that we can process the strata in amortized linear time, preparing $3d$-orderings for each stratum: enumerating these orderings in parallel with the first process thus guarantees (non-amortized) bounded delay.

Preparing the enumeration sequence. We now explain in more detail the working of the amortized linear time algorithm. The algorithm consists of two components. The first component runs in amortized linear time over the successive strata, and prepares a sequence $\Gamma_1, \Gamma_2, \ldots$ of concise graph representations of each stratum, called *stratum graphs*; for each $i \geqslant 1$, after $C \sum_{j=1}^{i} N_j$ computation steps, it has finished preparing the i-th stratum graph Γ_i in the sequence. The second component will run as soon as some stratum graph Γ_i is finished: it reads the graph Γ_i and computes a $3d$-ordering for $\mathrm{strat}_\ell(L, i)$ in (non-amortized) linear-time, using Lemma 4.3. Let us formalize the notion of a stratum graph:

▶ **Definition 5.3.** *Let Δ be the set of all push-pop edit scripts of length at most d; note that $|\Delta| \leqslant (2|\Sigma| + 2)^{d+1}$, and this bound depends on the alphabet and on d. For $i \geqslant 1$, the* i-th stratum graph *is the edge-labeled directed graph $\Gamma_i = (V_i, \eta_i)$ where the nodes $V_i = \{v_w \mid w \in \mathrm{strat}_\ell(L, i)\}$ correspond to words of the i-th stratum, and the directed (labeled) edges are given by the function $\eta_i \colon V_i \times \Delta \to V_i \cup \{\bot\}$ and describe the possible scripts: for each $v_w \in V_i$ and each $s \in \Delta$, if the script s is applicable to w and the resulting word w' is in $\mathrm{strat}_\ell(L, i)$ then $\eta(v_w, s) = v_{w'}$, otherwise $\eta(v_w, s) = \bot$.*

In our machine model, each node v_w of Γ_i is a record with $|\Delta|$ pointers, i.e., we do not store the word w. Hence, Γ_i has linear size in N_i.

A stratum graph sequence *is an infinite sequence $(\Gamma_1, v_{s_1}, v_{e_1}), (\Gamma_2, v_{s_2}, v_{e_2}), \ldots$ consisting of the successive stratum graphs together with couples of nodes of these graphs such that, for all $i \geqslant 1$, s_i and e_i are distinct words of the i-th stratum, and we have $\delta_{\mathrm{pp}}(e_i, s_{i+1}) \leqslant d$.*

We can now present the second component of our algorithm. Note that the algorithm runs on the in-memory representations of the stratum graphs, in which, e.g., the subscripts are not stored.

▶ **Proposition 5.4.** *For $i \geqslant 1$, given the stratum graph Γ_i and starting and ending nodes $v_{s_i} \neq v_{e_i}$ of Γ_i, we can compute in time $O(|\Gamma_i|)$ a sequence of edit scripts $\sigma_1, \ldots, \sigma_{N_i - 1}$ such that, letting $s_i = u_1, \ldots, u_{N_i}$ be the successive results of applying $\sigma_1, \ldots, \sigma_{N_i - 1}$ starting with s_i, then $u_1, \ldots, u_{N_i}$ is a $3d$-ordering of $\mathrm{strat}_\ell(L, i)$ starting at s_i and ending at e_i.*

Proof sketch. We apply the spanning tree enumeration technique from Lemma 4.3 (in $O(|\Gamma_i|)$) on Γ_i, starting with v_{s_i} and ending with v_{e_i}, and read the scripts from the edge labels. ◀

In the rest of the section we present the first component of our enumeration algorithm:

▶ **Proposition 5.5.** *There is an integer $C \in \mathbb{N}$ exponential in $|A|$ such that we can produce a stratum graph sequence $(\Gamma_1, v_{s_1}, v_{e_1}), (\Gamma_2, v_{s_2}, v_{e_2}), \ldots$ for L in amortized linear time, i.e., for each $i \geqslant 1$, after having run $C \sum_{j=1}^{i} N_j$ steps, the algorithm is done preparing $(\Gamma_i, v_{s_i}, v_{e_i})$.*

Word DAGs. The algorithm to prove Proposition 5.5 will grow a large structure in memory, common to all strata, from which we can easily compute the $(\Gamma_i, v_{s_i}, v_{e_i})$. We call this structure a *word DAG*. A word DAG is informally a representation of a collection of words, each of which has outgoing edges corresponding to the possible left and right push operations.

▶ **Definition 5.6.** *Let* $\Lambda := \{\texttt{pushR}(a) \mid a \in \Sigma\} \cup \{\text{pushL}(a) \mid a \in \Sigma\}$ *be the set of labels corresponding to the possible left and right push operations. A* pre-word DAG *is an edge-labeled directed acyclic graph (DAG)* $G = (V, \eta, \mathsf{root})$ *where* V *is a set of anonymous vertices,* $\mathsf{root} \in V$ *is the* root*, and* $\eta \colon V \times \Lambda \to V \cup \{\bot\}$ *represents the labeled edges in the following way: for each node* $v \in V$ *and label* $s \in \Lambda$*, if* $\eta(v, s) \neq \bot$ *then* v *has one successor* $\eta(v, s)$ *for label* s*, and none otherwise. We impose:*

- *The root has no incoming edges. All other nodes have exactly two incoming edges: one labeled* $\texttt{pushR}(a)$ *for some* $a \in \Sigma$*, the other labeled* $\text{pushL}(b)$ *for some* $b \in \Sigma$*. Each node stores two pointers leading to these two parents, which may be identical.*
- *All nodes can be reached from the root via at least one directed path.*
- *The root has one outgoing edge for each child, i.e., for all* $s \in \Lambda$*, we have* $\eta(\mathsf{root}, s) \neq \bot$*.*

The word *represented by a directed path from the root to a node* n *is defined inductively:*

- *the word represented by the empty path is* ϵ*,*
- *the word represented by a path* $P, \texttt{pushR}(a)$ *is* wa *where* w *is the word represented by* P*,*
- *the word represented by a path* $P, \text{pushL}(a)$ *is* aw *where* w *is the word represented by* P*.*

The pre-word DAG G *is called a* word DAG *if for each node* n*, all paths from* root *to* n *represent the same word. This word is then called* the word represented by n*.*

Example pre-word DAGs and word DAGs are shown on Figures 2 and 3 in the appendix. In our machine model, each node is represented by a record; crucially, like for stratum graphs, the word that the node represents is not explicitly written.

Crucially, word DAGs do not us allow not to create two different nodes that represent the same word – these would be problematic since we have to enumerate without repetition.

▶ **Fact 5.7.** *There are no two different nodes in a word DAG that represent the same word.*

We can then show the following theorem, intuitively saying that we can discover all the words of the language by only visiting words that are not too far from it:

▶ **Proposition 5.8.** *We can build a word DAG* G *representing the words of* L *in amortized linear time: specifically, for some value* C *that is exponential in* $|A|$*, for all* i*, after* $C \times \sum_{j=1}^{i} N_j$ *computation steps, for each word* w *of* Σ^* *whose push-pop distance to a word of* $\bigcup_{j=1}^{i} \text{strat}_\ell(L, j)$ *is no greater than* d*, then* G *contains a node that represents* w*. Moreover, there is also a value* D *exponential in* $|A|$ *such that any node that is eventually created in the word DAG represents a word that is at push-pop distance at most* D *from a word of* L*.*

Proof sketch. We progressively add nodes to a word DAG while efficiently preserving its properties, and thus avoid creating duplicate nodes. By labeling each node with the element of $Q \cup \{\bot\}$ achieved by the word represented by that node, and also by the distance to the closest known word of L, we can restrict the exploration to nodes corresponding to words that are close to the words of L, which ensures the amortized linear time bound. ◀

This is enough to prove Proposition 5.5: we run the algorithm of Proposition 5.8 and, whenever it has built a stratum, construct the stratum graph Γ_i and nodes v_{s_i}, v_{e_i} by exploring the relevant nodes of the word DAG. Full proofs are deferred to the full version [5].

6 Extensions

Complexity of determining the optimal distance. We have shown in Result 1 that, given a DFA A, we can compute in PTIME a minimal cardinality partition of $\mathrm{L}(A)$ into languages that are each d-orderable, for $d = 48|A|^2$. However, we may achieve a smaller distance d if we increase the cardinality, e.g., $a^* + bbba^*$ is $(1, 3)$-partition-orderable and not $(1, d)$-partition-orderable for $d < 3$, but is $(2, 1)$-partition-orderable. This tradeoff between t and d seems difficult to characterize, and in fact it is NP-hard to determine if an input DFA is (t, d)-partition-orderable, already for fixed t, d and for finite languages. Indeed, there is a simple reduction pointed out in [18] from the Hamiltonian path problem on grid graphs [12]:

▶ **Proposition 6.1** ([18]). *For any fixed $t, d \geqslant 1$, it is NP-complete, given a DFA A with $\mathrm{L}(A)$ finite, to decide if $\mathrm{L}(A)$ is (t, d)-partition-orderable (with the push-pop or Levenshtein distance).*

Push-pop-right distance. A natural restriction of the push-pop distance would be to only allow editions at the right endpoint of the word, called the *push-pop-right* distance. A d-ordering for this distance witnesses that the words of the language can be produced successively while being stored in a stack, each word being produced after at most d edits.

Unlike the push-pop distance, one can show that some regular languages are not even partition-orderable for this distance, e.g., a^*b^* is not (t, d)-partition-orderable with any $t, d \in \mathbb{N}$. The enumerable regular languages for this distance in fact correspond to the well-known notion of *slender languages*. Recall that a regular language L is *slender* [19] if there is a bound $C \in \mathbb{N}$ such that, for each $n \geqslant 0$, we have $|L \cap \Sigma^n| \leqslant C$. It is known [19] that we can test in PTIME if an input DFA represents a slender language. Rephrasing Result 2 from the introduction, we can show that a regular language is enumerable for the push-pop-right distance if and only if it is slender; further, if it is, then we can tractably compute the optimal number t of sequences (by counting the number of different paths to loops in the automaton), and we can do the enumeration with bounded delay:

▶ **Theorem 6.2.** *Given a DFA A, the language $\mathrm{L}(A)$ is (t, d)-partition-orderable for the push-pop-right distance for some $t, d \in \mathbb{N}$ if and only if $\mathrm{L}(A)$ is slender. Further, if $\mathrm{L}(A)$ is slender, we can compute in PTIME the smallest t such that $\mathrm{L}(A)$ is (t, d)-partition-orderable for some $d \in \mathbb{N}$ for the push-pop-right distance.*

In addition, there is an algorithm which, given a DFA A for which $\mathrm{L}(A)$ is slender and $t = 1$, enumerates the language $\mathrm{L}(A)$ with push-pop-right distance bound $2k$ and linear delay in $|A|$. Further, the sequence of edit scripts produced by the algorithm is ultimately periodic.

Of course, our results for the push-pop-right distance extend to the push-pop-left distance up to reversing the language, except for the complexity results because the reversal of the input DFA is generally no longer deterministic.

7 Conclusion and future work

We have introduced the problem of ordering languages as sequences while bounding the maximal distance between successive words, and of enumerating these sequences with small edit scripts to achieve bounded delay. Our main result is a PTIME characterization of the regular languages that can be ordered in this sense for the push-pop distance (or equivalently the Levenshtein distance), for any specific number of sequences; and a bounded-delay enumeration algorithm for the orderable regular languages. Our characterization uses the

number of classes of interchangeable states of a DFA A for the language, which, as our results imply, is an intrinsic parameter of $\mathrm{L}(A)$, shared by all (trimmed) DFAs recognizing the same language. We do not know if this parameter can be of independent interest.

Our work opens several questions for future research. The questions of orderability and enumerability can be studied for more general languages (e.g., context-free languages), other distances (in particular substitutions plus push-right operations, corresponding to the Hamming distance on a right-infinite tape), or other enumeration models (e.g., reusing factors of previous words). We also do not know the computational complexity, e.g., of optimizing the distance while allowing any finite number of threads, in particular for slender languages. Another complexity question is to understand if the bounded delay of our enumeration algorithm could be made polynomial in the input DFA rather than exponential, or what delay can be achieved if the input automaton is nondeterministic.

References

1 Margareta Ackerman and Erkki Mäkinen. Three new algorithms for regular language enumeration. In *ICCC*, 2009. URL: https://maya-ackerman.com/wp-content/uploads/2018/09/ThreeNewAlgorithmsForRegularLanEnum.pdf.

2 Margareta Ackerman and Jeffrey Shallit. Efficient enumeration of words in regular languages. *Theoretical Computer Science*, 410(37), 2009. URL: https://maya-ackerman.com/wp-content/uploads/2018/09/Enumeration_AckermanShallit_TCS.pdf.

3 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019. arXiv:1807.09320.

4 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, 2019. arXiv:1812.09519.

5 Antoine Amarilli and Mikaël Monet. Enumerating regular languages with bounded delay, 2023. Full version with proofs. arXiv:2209.14878.

6 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

7 Guillaume Bagan, Arnaud Durand, and Étienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007. URL: https://grandjean.users.greyc.fr/Recherche/PublisGrandjean/EnumAcyclicCSL07.pdf.

8 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2), 2015. URL: https://pdfs.semanticscholar.org/8df0/ad1c6aa0df93e58071b8afe3371a16a3182f.pdf, doi:10.1145/2699442.

9 Rainer Feldmann and Peter Mysliwietz. The shuffle exchange network has a Hamiltonian path. *Mathematical systems theory*, 29(5), 1996.

10 Lukas Fleischer and Jeffrey Shallit. Recognizing lexicographically smallest words and computing successors in regular languages. *International Journal of Foundations of Computer Science*, 32(06), 2021.

11 Fernando Florenzano, Cristian Riveros, Martin Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, 2018. arXiv:1803.05277.

12 Alon Itai, Christos H Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982. URL: http://www.cs.technion.ac.il/~itai/publications/Algorithms/Hamilton-paths.pdf.

13 Jerome J. Karaganis. On the cube of a graph. *Canadian Mathematical Bulletin*, 11(2), 1968.

14 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013. URL: https://hal.archives-ouvertes.fr/docs/00/90/70/85/PDF/cdlin-survey.pdf.

15 Erkki Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–61, 1997. URL: `http://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3479/3464`.

16 Torsten Mütze. Proof of the middle levels conjecture. *Proceedings of the London Mathematical Society*, 112(4):677–713, 2016. `arXiv:1404.4442`.

17 Torsten Mütze. Combinatorial Gray codes—An updated survey, 2022. `arXiv:2202.01280`.

18 pcpthm (https://cstheory.stackexchange.com/users/65605/pcpthm). Enumerating finite set of words with Hamming distance 1. Theoretical Computer Science Stack Exchange. Version: 2022-07-02. URL: `https://cstheory.stackexchange.com/q/51653`.

19 Jean-Éric Pin. Mathematical foundations of automata theory, 2019. URL: `https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf`.

20 Jakub Radoszewski and Wojciech Rytter. Hamiltonian paths in the square of a tree. In *ISAAC*, 2011. URL: `https://www.mimuw.edu.pl/~rytter/MYPAPERS/isaac2011_rytter.pdf`.

21 Frank Ruskey. Combinatorial generation. Preliminary working draft, 2003. URL: `https://page.math.tu-berlin.de/~felsner/SemWS17-18/Ruskey-Comb-Gen.pdf`.

22 Milan Sekanina. On an ordering of the set of vertices of a connected graph. *Publ. Fac. Sci. Univ. Brno*, 412, 1960.

23 Yann Strozecki et al. Enumeration complexity. *Bulletin of EATCS*, 3(129), 2019. URL: `http://eatcs.org/beatcs/index.php/beatcs/article/view/596`.

24 Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

25 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. Technical report, National Institute of Informatics, 2003. URL: `https://www.nii.ac.jp/TechReports/public_html/03-004E.pdf`.

26 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, 2016. `arXiv:1605.05102`.

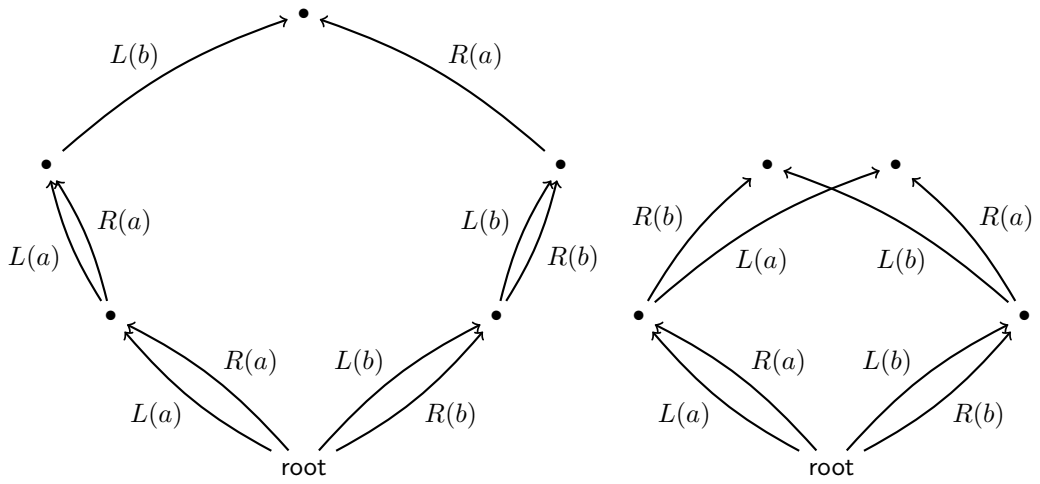


Figure 2 Two example pre-word DAGs which are not word DAGs. The labels pushL and `pushR` are abbreviated for legibility. In the left pre-word DAG, the four paths to the top node that start to the left of the root all represent the word *baa*, whereas the four paths to that same node that start to the right of the root all represent the word *bba*. In the right pre-word DAG, the left topmost node represents *ab* and *bb* and the right topmost node represents *aa* and *ba*. The criteria of word DAGs, and our construction to enlarge them, are designed to prevent these problems.

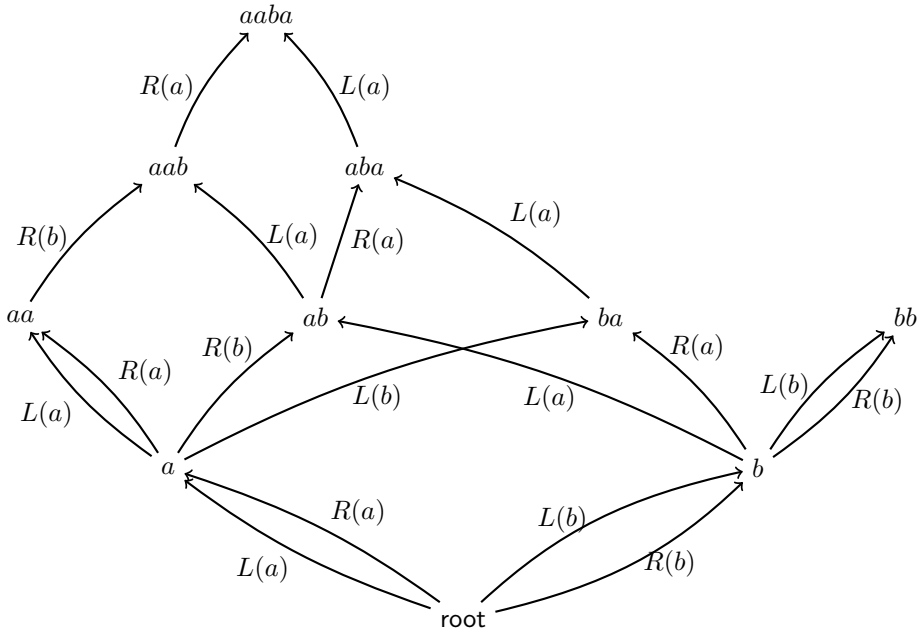


Figure 3 An example word DAG. We annotate the nodes with the word that they represent, even though in the memory representation the nodes are anonymous and the words are not represented. The labels pushL and `pushR` are abbreviated for legibility.