Constant-Delay Enumeration for Nondeterministic Document Spanners

ANTOINE AMARILLI, LTCI and Télécom Paris and Institut polytechnique de Paris
PIERRE BOURHIS, CRIStAL and CNRS UMR 9189 and Inria Lille
STEFAN MENGEL, CRIL, CNRS & Univ Artois
MATTHIAS NIEWERTH, University of Bayreuth

We consider the information extraction framework known as *document spanners* and study the problem of efficiently computing the results of the extraction from an input document, where the extraction task is described as a sequential *variable-set automaton* (VA). We pose this problem in the setting of enumeration algorithms, where we can first run a preprocessing phase and must then produce the results with a small delay between any two consecutive results. Our goal is to have an algorithm that is tractable in combined complexity, i.e., in the sizes of the input document and the VA, while ensuring the best possible data complexity bounds in the input document size, i.e., constant delay in the document size. Several recent works at PODS'18 proposed such algorithms but with linear delay in the document size or with an exponential dependency in size of the (generally nondeterministic) input VA. In particular, Florenzano et al. suggest that our desired runtime guarantees cannot be met for general sequential VAs. We refute this and show that, given a nondeterministic sequential VA and an input document, we can enumerate the mappings of the VA on the document with the following bounds: the preprocessing is linear in the document size and polynomial in the size of the VA, and the delay is independent of the document and polynomial in the size of the VA. The resulting algorithm thus achieves tractability in combined complexity and the best possible data complexity bounds. Moreover, it is rather easy to describe, particularly for the restricted case of so-called extended VAs. Finally, we evaluate our algorithm empirically using a prototype implementation.

CCS Concepts: • **Information systems → Information extraction**; • **Theory of computation → Regular languages**; **Design and analysis of algorithms**;

Additional Key Words and Phrases: Documents spanners, information extraction, constant delay enumeration

1 INTRODUCTION

Information extraction from text documents is an important problem in data management. One approach to this task has recently attracted a lot of attention: it uses *document spanners*, a declarative logic-based approach first implemented by IBM in their tool SystemT [27] and whose core semantics has then been formalized by Fagin et al. [11]. The spanner approach uses variants of regular expressions (e.g., *regex-formulas* with variables), compiles them to variants of finite automata (e.g., variable-set automata (VAs)), and evaluates them on the input document to extract the data of interest. After this extraction phase, algebraic operations like joins, unions, and projections can be performed. The formalization of the spanner framework by Fagin et al. [11] has led to a thorough investigation of its properties by the theoretical database community [12, 13, 15, 16, 22].

Here we consider the basic task in the spanner framework of efficiently computing the results of the extraction, i.e., computing without duplicates all tuples of ranges of the input document (called *mappings*) that satisfy the conditions described by a VA. As many algebraic operations can also be compiled into VAs [16], this task actually solves the whole data extraction problem for so-called *regular spanners* [11]. Although the extraction task is intractable for general VAs [13], it is known to be tractable if we impose that the VA is *sequential* [12, 16], which requires that all accepting runs describe a well-formed mapping; we will make this assumption throughout our work. Even then, however, it may still be unreasonable in practice to materialize all mappings: if there are k variables to extract, then mappings are k-tuples and there may be up to n^k mappings on an input document of size n, which is unrealistic if n is large. For this reason, recent works [12, 16, 22] have studied the extraction task in the setting of *enumeration algorithms*: instead of materializing all mappings, we enumerate them one by one while ensuring that the *delay* between two results is always small. Specifically, Freydenberger et al. [16, Theorem 3.3] have shown how to enumerate the mappings with delay linear in the input document and quadratic in the VA, i.e., given a document d and a functional VA A (a subclass of sequential VAs), the delay is $O(|A|^2 \times |d|)$.

Although this result ensures tractability in both the size of the input document and the automaton, the delay may still be long as $|d|$ is generally very large. By contrast, enumeration algorithms for database tasks often enforce stronger tractability guarantees in data complexity [29, 32], particularly *linear preprocessing* and *constant delay* (when measuring complexity in the RAM model with uniform cost measure [1]). Such algorithms consist of two phases: a *preprocessing phase* that precomputes an index data structure in linear data complexity, and an *enumeration phase* that produces all results so that the delay between any two consecutive results is always *constant*, i.e., independent from the input data. It was recently shown by Florenzano et al. [12] that this strong guarantee could be achieved when enumerating the mappings of VAs if we only focus on data complexity, i.e., for any *fixed* VA, we can enumerate its mappings with linear preprocessing and constant delay in the input document. However, the preprocessing and delay in the work of Florenzano et al. [12] are exponential in the VA because they first determinize it [12, Propositions 4.1 and 4.3]. This is problematic because the VAs constructed from regex-formulas [11] are generally nondeterministic.

Thus, to efficiently enumerate the results of the extraction, we would ideally want to have the best of both worlds: ensure that the *combined complexity* (in the sequential VA and in the document) remains polynomial while ensuring that the *data complexity* (in the document) is as small as possible, i.e., linear time for the preprocessing phase and constant time for the delay of the enumeration phase. However, up to now, there was no known algorithm to satisfy these requirements while working on nondeterministic sequential VAs. Further, it was conjectured that such an algorithm is unlikely to exist [12] because the related task of *counting* the number of mappings is SpanL-hard for such VAs.

The question of nondeterminism is also unsolved for the related problem of enumerating the results of monadic second-order (MSO) queries on words and trees: there are several approaches for this task where the query is given as an automaton, but they require the automaton to be deterministic [2, 6] or their delay is not constant in the input document [20]. Hence, also in the context of MSO enumeration, it is not known whether we can achieve linear preprocessing and constant delay in data complexity while remaining tractable in the (generally nondeterministic) automaton. The result that we show in the present work implies that we can achieve this for MSO queries on words when all free variables are first order, with the query being represented as a generally nondeterministic sequential VA, or as a sequential regex-formula with capture variables: note that an extension to trees is investigated in our follow-up work [5].

Contributions. In this work, we show that nondeterminism is in fact not an obstacle to enumerating the results of document spanners: we present an algorithm that enumerates the mappings of a nondeterministic sequential VA in polynomial combined complexity while ensuring linear preprocessing and constant delay in the input document. This answers the open question of Florenzano et al. [12] and improves on the bounds of Freydenberger et al. [16]. More precisely, we show the following theorem.

THEOREM 1.1. *Let $2 \leq \omega \leq 3$ be an exponent for Boolean matrix multiplication. Let $\mathcal{A}$ be a sequential VA with variable set $\mathcal{V}$ and with state set Q, and let d be an input document. We can enumerate the mappings of $\mathcal{A}$ on d with preprocessing time in $O((|Q|^{\omega+1} + |\mathcal{A}|) \times |d|)$ and with delay $O(|\mathcal{V}| \times (|Q|^2 + |\mathcal{A}| \times |\mathcal{V}|^2))$, i.e., linear preprocessing and constant delay in the input document, and polynomial preprocessing and delay in the input VA.*

The existence of such an algorithm is surprising but in hindsight not entirely unexpected: remember that in formal language theory, when we are given a word and a nondeterministic finite automaton, we can evaluate the automaton on the word with tractable combined complexity by determinizing the automaton "on the fly," i.e., computing at each position of the word the set of states where the automaton can be. Our algorithm generalizes this intuition and extends it to the task of enumerating mappings without duplicates: we first present it for so-called *extended sequential VAs*,[1] a variant of sequential VAs introduced by Florenzano et al. [12], before generalizing it to sequential VAs. Our overall approach is to construct a kind of product of the input document with the extended VA, similarly to Florenzano et al. [12]. We then use several tricks to ensure the constant delay bound despite nondeterminism; in particular, we precompute a *jump function* that allows us to skip quickly the parts of the document where no variable can be assigned. The resulting algorithm is rather simple and has no large hidden constants. Note that our enumeration algorithm does not contradict the counting hardness results of Florenzano et al. [12, Theorem 5.2]: although our algorithm *enumerates* mappings with constant delay and without duplicates, we do not see a way to adapt it to *count* the mappings efficiently. This is similar to the enumeration and counting problems for maximal cliques: one can enumerate maximal cliques with polynomial delay [30], but counting them is #P-hard [31].

To extend our result to sequential VAs that are not extended, one possibility would be to convert them to extended VAs, but this necessarily entails an exponential blowup [12, Proposition 4.2]. We avoid this by adapting our algorithm to work with nonextended sequential VAs directly. Our idea for this is to efficiently enumerate at each position the possible sets of markers that can be assigned by the VA: we do so by enumerating paths in the VA, relying on the fact that the VA is sequential

---

[1]Note that, contrary to what the terminology suggests, VAs are not special cases of extended VAs. Further, although extended VAs can be converted in PTIME to VAs, the converse is not true, as there are extended VAs for which the smallest equivalent VA has exponential size [12].

so these paths are acyclic. The challenge is that the same set of markers can be captured by many different paths, but we explain how we can explore efficiently the set of distinct paths with a technique known as *flashlight search* [21, 26]: the key idea is that we can efficiently determine which partial sets of markers can be extended to the label of a path (Lemma 6.4).

Of course, our main theorem (Theorem 1.1) implies analogous results for all spanner formalisms that can be translated to sequential VAs. In particular, spanners are not usually written as automata by users but instead are given in a form of regular expressions called *regex-formulas*, see the work of Fagin et al. [11] for exact definitions. As we can translate sequential regex-formulas to sequential VAs in linear time [11, 16, 22], our results imply that we can also evaluate them.

COROLLARY 1.2. *Let $2 \leq \omega \leq 3$ be an exponent for Boolean matrix multiplication. Let φ be a sequential regex-formula with variable set $\mathcal{V}$, and let d be an input document. We can enumerate the mappings of φ on d with preprocessing time in $O(|\varphi|^{\omega+1} \times |d|)$ and with delay $O(|\mathcal{V}| \times (|\varphi|^2 + |\varphi| \times |\mathcal{V}|^2))$, i.e., linear preprocessing and constant delay in the input document, and polynomial preprocessing and delay in the input regex-formula.*

Another direct application of our result is for so-called *regular spanners*, which are unions of conjunctive queries (UCQs) posed on regex-formulas, i.e., the closure of regex-formulas under union, projection, and joins. We again point the reader to other works [11, 16] for the full definitions. Considering that UCQs can in fact be evaluated by VAs, our result also implies tractability for such representations, as long as we only perform a bounded number of joins:

COROLLARY 1.3. *For every fixed $k \in \mathbb{N}$, let k-UCQ denote the class of document spanners represented by UCQs over functional regex-formulas with at most k applications of the join operator. Then the mappings of a spanner in k-UCQ can be enumerated with linear preprocessing and constant delay in the document size, and with polynomial preprocessing and delay in the size of the spanner representation.*

One last contribution of this work is to present a prototype implementation of the enumeration algorithm presented here, which is available online as open source software.[2] We evaluate this software experimentally for different types of queries. The results show that our approach can be implemented in practice and run efficiently.

Article structure. In Section 2, we formally define spanners, VAs, and the enumeration problem that we want to solve on them. In Sections 3 through 5, we prove our main result (Theorem 1.1) for *extended* VAs, where the sets of variables that can be assigned at each position are specified explicitly. We first describe in Section 3 the main part of our preprocessing phase, which converts the extended VA and input document to a *mapping* DAG (directed acyclic graph) whose paths describe the mappings that we wish to enumerate. We then describe in Section 4 how to enumerate these paths, up to having precomputed a so-called *jump function* whose computation is explained in Section 5. Last, we adapt our scheme in Section 6 for sequential VAs that are not extended. We present our experimental results in Section 7 and conclude in Section 8.

This article is an extended version of our earlier work [4]. Compared to that work [4], in this work we provide complete proofs of the results and present the new experimental analysis of Section 7.

2 PRELIMINARIES

Document spanners. We fix a finite alphabet Σ. A *document* $d = d_0 \cdots d_{n-1}$ is just a word over Σ. A *span* of d is a pair $[i, j\rangle$ with $0 \leq i \leq j \leq |d|$ that represents a substring (contiguous subsequence)

---

[2]https://github.com/PoDMR/enum-spanner-rs.

of d starting at position i and ending at position $j - 1$. To describe the possible results of an information extraction task, we will use a finite set $\mathcal{V}$ of variables and define a result as a *mapping* from these variables to spans of the input document. Following other works [12, 22] but in contrast to the work of Fagin et al. [11], we will not require mappings to assign all variables: formally, a *mapping* of $\mathcal{V}$ on d is a function μ from some domain $\mathcal{V}' \subseteq \mathcal{V}$ to spans of d. We define a *document spanner* to be a function assigning to every input document d a set of mappings, which denotes the set of results of the extraction task on the document d.

Variable-set automata. We will represent document spanners using VAs. The transitions of a VA can carry letters of Σ or *variable markers*, which are either of the form $x \vdash$ for a variable $x \in \mathcal{V}$ (denoting the start of the span assigned to x) or $\dashv x$ (denoting its end). Formally, a *variable-set automaton* $\mathcal{A}$ (or VA) is then defined to be an automaton $\mathcal{A} = (Q, q_0, F, \delta)$, where the transition relation δ consists of *letter transitions* of the form (q, a, q') for $q, q' \in Q$ and $a \in \Sigma$, and of *variable transitions* of the form $(q, x \vdash, q')$ or $(q, \dashv x, q')$ for $q, q' \in Q$ and $x \in \mathcal{V}$. A *configuration* of a VA is a pair (q, i), where $q \in Q$ and i is a position of the input document d. A *run* σ of $\mathcal{A}$ on d is then a sequence of configurations

$$(q_0, i_0) \xrightarrow{\sigma_1} (q_1, i_1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_m} (q_m, i_m),$$

where $i_0 = 0$, $i_m = |d|$, and where for every $1 \le j \le m$, one of the following holds:

- The label σ_j is a letter of Σ, we have $i_j = i_{j-1} + 1$, we have $d_{i_{j-1}} = \sigma_j$, and (q_{j-1}, σ_j, q_j) is a letter transition of $\mathcal{A}$;
- The label σ_j is a variable marker, we have $i_j = i_{j-1}$, and (q_{j-1}, σ_j, q_j) is a variable transition of $\mathcal{A}$. In this case, we say that the variable marker σ_j is *read* at position i_j.

As usual, we say that a run is *accepting* if $q_m \in F$. A run is *valid* if it is accepting, every variable marker is read at most once, if an open marker $x \vdash$ is read at a position i then the corresponding close marker $\dashv x$ is read at a position i' with $i \le i'$, and if $x \vdash$ is not read then $\dashv x$ is not read either. Each valid run defines a mapping on the domain $\mathcal{V}'$ of the variables for which the run has read some markers: specifically, each variable $x \in \mathcal{V}'$ is mapped to the span $[i, i'\rangle$ such that $x \vdash$ is read at position i and $\dashv x$ is read at position i'. The *document spanner* of the VA $\mathcal{A}$ is then the function that assigns to every document d the set of mappings defined by the valid runs of $\mathcal{A}$ on d: note that the same mapping can be defined by multiple different runs, and note that the different runs may have different domains. The task studied in this work is the following: given a VA $\mathcal{A}$ and a document d, enumerate *without duplicates* the mappings that are assigned to d by the document spanner of $\mathcal{A}$. The enumeration must write each mapping as a set of pairs (m, i), where m is a variable marker and i is a position of d, each set being written as a sequence in some arbitrary order. We will say that a set of pairs of markers and positions is *valid* when every marker occurs at most once in the set, if an open marker $x \vdash$ occurs in the set as $(x \vdash, i)$ then the set also contains $(\dashv x, i')$ with $i < i'$, and if $x \vdash$ does not occur in the set then neither does $\dashv x$. Thus, the results of the enumeration are always valid in this sense. Note that we will often abuse notation and identify the function representation of mappings defined earlier with this representation as a set of pairs that is valid.

Sequential VAs. We cannot hope to efficiently enumerate the mappings of arbitrary VAs because it is already NP-complete to decide if, given a VA $\mathcal{A}$ and a document d, there are any valid runs of $\mathcal{A}$ on d [13]. For this reason, we will restrict ourselves to so-called *sequential* VAs [22]. A VA $\mathcal{A}$ is *sequential* if for every document d, every accepting run of $\mathcal{A}$ of d is also valid: this implies that the document spanner of $\mathcal{A}$ can simply be defined following the accepting runs of $\mathcal{A}$. If we are given a VA, then we can test in NL whether it is sequential [22, Proposition 5.5], and otherwise we

can convert it to an equivalent sequential VA (i.e., that defines the same document spanner) with an unavoidable exponential blowup in the number of variables (not in the number of states), using existing results.

PROPOSITION 2.1. *Given a VA $\mathcal{A}$ on variable set $\mathcal{V}$, letting $k := |\mathcal{V}|$ and r be the number of states of $\mathcal{A}$, we can compute an equivalent sequential VA $\mathcal{A}'$ with $3^k r$ states. Conversely, for any $k \in \mathbb{N}$, there exists a VA $\mathcal{A}_k$ with 1 state on a variable set with k variables such that any sequential VA equivalent to $\mathcal{A}_k$ has at least 3^k states.*

PROOF. This can be shown exactly like in the work of Freydenberger (Proposition 12 [13] and Proposition 3.9 [14]). In short, the upper bound is shown by modifying $\mathcal{A}$ to remember in the automaton state which variables have been opened or closed, and by rewiring the transitions to ensure that the run is valid: this creates 3^k copies of every state because each variable can be either unseen, opened, or closed. For the lower bound, Freydenberger [14, Proposition 3.9] gives a VA for which any equivalent sequential VA must remember the status of all variables in this way. □

All VAs studied in this work will be sequential, and we will further assume that they are *trimmed* in the sense that for every state q there is a document d and an accepting run of the VA where the state q appears. This condition can be enforced in linear time on any sequential VA: we do a graph traversal to identify the accessible states (the ones that are reachable from the initial state), we do another graph traversal to identify the co-accessible states (the ones from which we can reach a final state), and we remove all states that are not accessible or not co-accessible. We will implicitly assume that all sequential VAs have been trimmed, which implies that they cannot contain any cycle of variable transitions (as such a cycle would otherwise appear in a run, which would not be valid).

Extended VAs. We will first prove our results for a variant of sequential VAs introduced by Florenzano et al. [12], called sequential *extended VAs*. An extended VA on alphabet Σ and variable set $\mathcal{V}$ is an automaton $\mathcal{A} = (Q, q_0, F, \delta)$, where the transition relation δ consists of *letter transitions* as before, and of *extended variable transitions* (or *ev-transitions*) of the form (q, M, q'), where M is a possibly empty set of variable markers. Intuitively, on ev-transitions, the automaton reads multiple markers at once. Formally, a *run* σ of $\mathcal{A}$ on $d = d_0 \cdots d_{n-1}$ is a sequence of configurations (defined like before) where letter transitions and ev-transitions alternate:

$$(q_0, 0) \xrightarrow{M_0} (q'_0, 0) \xrightarrow{d_0} (q_1, 1) \xrightarrow{M_1} (q'_1, 1) \xrightarrow{d_1} \cdots \xrightarrow{d_{n-1}} (q_n, n) \xrightarrow{M_n} (q'_n, n),$$

where (q'_i, d_i, q_{i+1}) is a letter transition of $\mathcal{A}$ for all $0 \leq i < n$, and (q_i, M_i, q'_i) is an ev-transition of $\mathcal{A}$ for all $0 \leq i \leq n$, where M_i is the set of variable markers *read* at position i. Accepting and valid runs are defined like before, and the extended VA is sequential if all accepting runs are valid, in which case its document spanner is defined like before.

Our definition of extended VAs is slightly different from that of Florenzano et al. [12] because we allow ev-transitions that read the empty set to change the automaton state. This allows us to make a small additional assumption to simplify our proofs: we require that the states of extended VAs are partitioned between *ev-states*, from which only ev-transitions originate (i.e., the q_i above), and *letter-states*, from which only letter transitions originate (i.e., the q'_i above); and we impose that the initial state is an ev-state and the final states are all letter-states. Note that transitions reading the empty set move from an ev-state to a letter-state, like all other ev-transitions. Our requirement can be imposed in linear time on any extended VA, by rewriting each state to one letter-state and one ev-state, and rewiring the transitions and changing the initial/final status of states appropriately. This rewriting preserves sequentiality and guarantees that any path in the
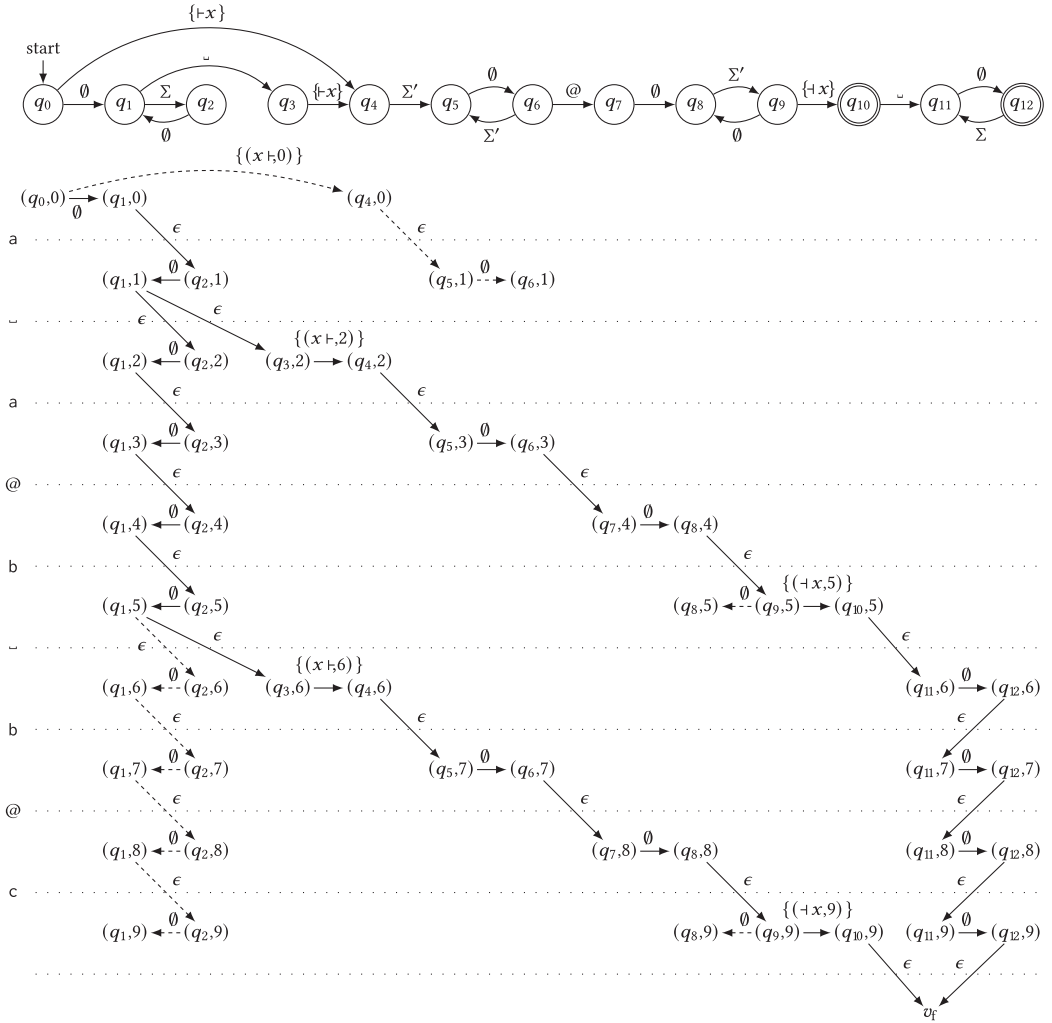
Fig. 1. Example sequential extended VA $\mathcal{A}_0$ to extract email addresses (see Example 2.2) and example mapping DAG on an example document (see Examples 3.3, 3.6, 3.7, and 3.10).

rewritten extended VA must alternate between letter transitions and ev-transitions. Hence, we implicitly make this assumption on all extended VAs from now on.

Example 2.2. The top of Figure 1 represents a sequential extended VA $\mathcal{A}_0$ to extract email addresses. To keep the example readable, we simply define them as words (delimited by a space or by the beginning or end of document) that contain one at-sign "@" preceded and followed by a nonempty sequence of non-"@" characters. In the drawing of $\mathcal{A}_0$, the initial state q_0 is at the left, and the states q_{10} and q_{12} are final. The transitions labeled by Σ represent a set of transitions for each letter of Σ, and the same holds for Σ', which we define as $\Sigma' := \Sigma \setminus \{@, \_\}$.

It is easy to see that, on any input document d, there is one mapping of $\mathcal{A}_0$ on d per email address contained in d, which assigns the markers $x \vdash$ and $\dashv x$ to the beginning and end of the email address, respectively. In particular, $\mathcal{A}_0$ is sequential, because any accepting run is valid. Note that

$\mathcal{A}_0$ happens to have the property that each mapping is produced by exactly one accepting run, but our results in this work do not rely on this property.

Matrix multiplication. The complexity bottleneck for some of our results will be the complexity of multiplying two Boolean matrices, which is a long-standing open problem; see, e.g., the work of Gall [17] for a recent discussion. When stating our results, we will often denote by $2 \leq \omega \leq 3$ an exponent for Boolean matrix multiplication: this is a constant such that the product of two r-by-r Boolean matrices can be computed in time $O(r^\omega)$. For instance, we can take $\omega := 3$ if we use the naive algorithm for Boolean matrix multiplication, and it is obvious that we must have $\omega \geq 2$. The best known upper bound is currently $\omega < 2.3728639$; see another work by Gall [18].

3   COMPUTING MAPPING DAGS FOR EXTENDED VAS

We start our work by studying extended VAs, which are easier to work with because the set of markers that can be assigned at every position is explicitly written as the label of a single transition. We accordingly show Theorem 1.1 for the case of extended VAs in Sections 3 through 5. We will then cover the case of nonextended VAs in Section 6.

Mapping DAGs. To show Theorem 1.1 for extended VAs, we will reduce the problem of enumerating the mappings captured by $\mathcal{A}$ to that of enumerating path labels in a special kind of DAG called a *mapping DAG*. This DAG is intuitively a variant of the product of $\mathcal{A}$ and of the document d, where we represent simultaneously the position in the document and the corresponding state of $\mathcal{A}$. We will no longer care in the mapping DAG about the labels of letter transitions, so we will erase these labels and call these transitions ϵ-*transitions*. As for the ev-transitions, we will extend their labels to indicate the position in the document in addition to the variable markers. We first give the general definition of a mapping DAG.

Definition 3.1. A *mapping DAG* consists of a set V of *vertices*, an *initial vertex* $v_0 \in V$, a *final vertex* $v_f \in V$, and a set of *edges* E where each edge (s, x, t) has a *source vertex* $s \in V$, a *target vertex* $t \in V$, and a *label* x that may be ϵ (in which case we call the edge an ϵ-*edge*) or a finite (possibly empty) set of pairs (m, i), where m is a variable marker and i is a position. These edges are called *marker edges*. We require that the graph (V, E) is acyclic. We say that a mapping DAG is *normalized* if every path in the mapping DAG alternates between marker edges and ϵ-edges, every path starting at the initial vertex starts with a marker edge, and every path ending at the final vertex ends with an ϵ-edge.

The *premapping* $\mu(\pi)$ of a path π in the mapping DAG is the union of labels of the marker edges of π: we require of any mapping DAG that, for every path π, this union is disjoint, and that for every path π from v_0 to v_f, the premapping $\mu(\pi)$ is valid, i.e., it corresponds to a mapping. Given a set U of vertices of G, we write $\mathcal{M}(U)$ for the set of premappings of paths from a vertex of U to the final vertex; note that the same premapping may be captured by multiple different paths. The set of premappings *captured* by G is then $\mathcal{M}(G) := \mathcal{M}(\{v_0\})$; all of these are mappings, i.e., they are valid.

Intuitively, the ϵ-edges will correspond to letter transitions of $\mathcal{A}$ (with the letter being erased, i.e., replaced by ϵ), and marker edges will correspond to ev-transitions: their labels are a possibly empty finite set of pairs of a variable marker and position, describing which variables have been assigned during the transition. We now explain how we construct a DAG from $\mathcal{A}$ and from a document d, which we call the *product DAG* of $\mathcal{A}$ and d, and which we will show to be a mapping DAG.

Definition 3.2. Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a sequential extended VA, and let $d = d_0 \cdots d_{n-1}$ be an input document. The *product DAG* of $\mathcal{A}$ and d is the DAG whose vertex set is $Q \times \{0, \ldots, n\} \cup \{v_f\}$ with $v_f := (\bullet, n + 1)$ for some fresh value $\bullet$. Its edges are as follows:

- For every letter-transition (q, a, q') in δ, for every $0 \le i < |d|$ such that $d_i = a$, there is an ϵ-edge from (q, i) to $(q', i + 1)$.
- For every ev-transition (q, M, q') in δ, for every $0 \le i \le |d|$, there is a marker edge from (q, i) to (q', i) labeled with the (possibly empty) set $\{(m, i) \mid m \in M\}$.
- For every final state $q \in F$, an ϵ-edge from (q, n) to v_f.

The initial vertex of the product DAG is $(q_0, 0)$ and the final vertex is v_f.

Note that, contrary to Florenzano et al. [12], we do not contract the ϵ-edges but keep them throughout our algorithm.

Example 3.3. The product DAG of our example sequential extended VA $\mathcal{A}_0$ and of the example document a␣a@b␣b@c is shown on Figure 1, with the document being written at the left from top to bottom. The initial vertex of the DAG is $(q_0, 0)$ at the top left, and its final vertex is v_f at the bottom. We draw marker edges horizontally and ϵ-edges diagonally. To simplify the example, we only draw the parts of the DAG that are reachable from the initial vertex. Edges are dashed when they cannot be used to reach the final vertex.

It is easy to see that this construction satisfies the following definition.

CLAIM 3.4. *The product DAG of $\mathcal{A}$ and d is a normalized mapping DAG.*

PROOF. It is immediate that the product DAG is indeed acyclic, because the second component is always nondecreasing, and an edge where the second component does not increase (corresponding to an ev-transition of the VA) must be followed by an edge where it does (corresponding to a letter-transition of the VA). What is more, we claim that no path in the product DAG can include two edges whose labels contain the same pair (m, i) so that the unions used to define the mappings of the mapping DAG are indeed disjoint. To see this, consider a path from an edge $((q_1, i_1), M_1, (q'_1, i_1))$ to an edge $((q_2, i_2), M_2, (q'_2, i_2))$ where $M_1 \ne \epsilon$ and $M_2 \ne \epsilon$, we have $i_1 < i_2$ and M_1 and M_2 are disjoint because all elements of M_1 have i_1 as their first component, and all elements of M_2 have i_2 as their first component. Further, the product DAG is also normalized because $\mathcal{A}$ is an extended VA that we have preprocessed to distinguish letter-states and ev-states. □

Further, the product DAG clearly captures what we want to enumerate. Formally, we have the following claim.

CLAIM 3.5. *The set of mappings of $\mathcal{A}$ on d is exactly the set of mappings $\mathcal{M}(G)$ captured by the product DAG G.*

PROOF. This is immediate as there is a clear bijection between accepting runs of $\mathcal{A}$ on d and paths from the initial vertex of G to its final vertex, and this bijection ensures that the label of the path in G is the mapping corresponding to that accepting run. □

Example 3.6. The set of mappings captured by the example product DAG on Figure 1 is

$$\{ \{(x \vdash, 2), (\dashv x, 5)\}, \{(x \vdash, 6), (\dashv x, 9)\} \},$$

and this is indeed the set of mappings of the example extended VA $\mathcal{A}_0$ on the example document.

Connection to circuits. We remark that our mapping DAG can be seen as a kind of Boolean circuit, and our enumeration algorithm on mapping DAGs can be connected to earlier work by some of

the present authors on enumeration for Boolean circuits [2, 5]. Specifically, a mapping DAG can be understood as describing a kind of binary decision diagram: these are a special kind of Boolean circuits where each conjunction always involves a literal. This class is more restricted than the circuits obtained for tree automata in other works [2, 5], intuitively because trees feature branching that requires the conjunction of multiple subruns. Our enumeration algorithm on mapping DAGs in the present work could then be phrased as a generic algorithm on a class of bounded-width, nondeterministic binary decision diagrams. However, in this work, we chose to eschew the circuit terminology, as we believe that our definitions and algorithms are simpler to present on an ad hoc mapping DAG data structure.

Trimming, levels, and level sets. Our task is to enumerate $\mathcal{M}(G)$ *without duplicates*, and this is still nonobvious: because of nondeterminism, the same mapping in the product DAG may be witnessed by exponentially many paths, corresponding to exponentially many runs of the nondeterministic extended VA $\mathcal{A}$. We will present in the next section our algorithm to perform this task on the product DAG G. To do this, we will need to preprocess G by *trimming* it and introduce the notion of *levels* to reason about its structure.

First, we present how to *trim* G. We say that G is *trimmed* if every vertex v is both *accessible* (there is a path from the initial vertex to v) and *co-accessible* (there is a path from v to the final vertex). Given a mapping DAG, we can clearly trim in linear time by two linear-time graph traversals. Hence, we will always implicitly assume that the mapping DAG is trimmed. If the mapping DAG may be empty once trimmed, then there are no mappings to enumerate, so our task is trivial. Hence, we assume in the sequel that the mapping DAG is nonempty after trimming. Further, if $\mathcal{V} = \emptyset$, then the only possible mapping is the empty mapping and we can produce it at that stage, so in the sequel we assume that $\mathcal{V}$ is nonempty.

Example 3.7. For the mapping DAG of Figure 1, trimming eliminates the nonaccessible vertices (which are not depicted) and the non-co-accessible vertices, i.e., those with incoming dashed edges. Note that trimming the mapping DAG has an effect even though the example sequential extended VA $\mathcal{A}_0$ was already trimmed.

Second, we next present an invariant on the structure of G by introducing the notion of *levels*.

Definition 3.8. A mapping DAG G is *leveled* if its vertices $v = (q, i)$ are pairs whose second component i is a nonnegative integer called the *level* of the vertex and written $\mathrm{level}(v)$, and where the following conditions hold:

- For the initial vertex v_0 (which has no incoming edges), the level is 0;
- For every ϵ-edge from u to v, we have $\mathrm{level}(v) = \mathrm{level}(u) + 1$;
- For every marker edge from u to v, we have $\mathrm{level}(v) = \mathrm{level}(u)$, and furthermore, all pairs (m, i) in the label of the edge have $i = \mathrm{level}(v)$.

The *depth* D of G is the maximal level. The *width* W of G is the maximal number of vertices that have the same level.

The following claim is then immediate by construction.

CLAIM 3.9. *The product DAG of $\mathcal{A}$ and d is leveled, and we have $W \leq |Q|$ and $D = |d| + 1$.*

PROOF. It is clear by construction that the product DAG satisfies the first three points in the definition of a leveled mapping DAG. To see why the last point holds, observe that for every edge of the product DAG, for every pair (m, i) that occurs in the label of that edge, the second component i of the pair indicates how many letters of d have been read so far, so the source vertex must have level i.

To see why the width and depth bounds hold, observe that each level of the product DAG corresponds to a copy of $\mathcal{A}$, so it has at most $|Q|$ vertices, and that the number of levels corresponds to the number of letters of the document, plus one level for the final vertex. □

Example 3.10. The example mapping DAG on Figure 1 is leveled, and the levels are represented as horizontal layers separated by dotted lines: the topmost level is level 0, and the bottommost level is level 10.

In addition to levels, we will need the notion of a *level set*.

Definition 3.11. A *level set* Λ is a nonempty set of vertices in a leveled normalized mapping DAG that all have the same level (written level(Λ)), and that are all the source of some marker edge. The singleton $\{v_f\}$ of the final vertex is also considered as a level set.

In particular, letting v_0 be the initial vertex, the singleton $\{v_0\}$ is a level set. Further, if we consider a level set Λ that is not the final vertex, then we can follow marker edges from all vertices of Λ (and only such edges) to get to other vertices, and follow ϵ-edges from these vertices (and only such edges) to get to a new level set Λ' with level(Λ') = level(Λ) + 1.

4 ENUMERATION FOR MAPPING DAGS

In the previous section, we reduced our enumeration problem for extended VAs on documents to an enumeration problem on normalized leveled mapping DAGs. In this section, we describe our main enumeration algorithm on such DAGs and show the following.

THEOREM 4.1. *Let* $2 \leq \omega \leq 3$ *be an exponent for Boolean matrix multiplication. Given a normalized leveled mapping DAG G of depth D and width W, we can enumerate $\mathcal{M}(G)$ (without duplicates) with preprocessing $O(|G| + D \times W^{\omega+1})$ and delay $O(W^2 \times (r + 1))$, where r is the size of each produced mapping.*

Remember that as part of our preprocessing, we have ensured that the leveled normalized mapping DAG G has been trimmed. We will also preprocess G to ensure that, given any vertex, we can access its adjacency list (i.e., the list of its outgoing edges) in some sorted order on the labels, where we assume that $\emptyset$-edges come last. This sorting can be done in linear time on the RAM model [19, Theorem 3.1], so the preprocessing is in $O(|G|)$.

Our general enumeration algorithm is then presented as Algorithm 1. We explain the missing pieces next. The function ENUM is initially called with $\Lambda = \{v_0\}$, the level set containing only the initial vertex, and with Mapping being the empty set.

ALGORITHM 1: Main enumeration algorithm

1: **procedure** ENUM(Λ, Mapping)
2:     $\Lambda' :=$ JUMP(Λ)
3:     **if** Λ' is the singleton $\{v_f\}$ of the final vertex **then**
4:         OUTPUT(Mapping)
5:     **else**
6:         **for** (LocMark, Λ'') in NEXTLEVEL(Λ') **do**
7:             ENUM(Λ'', LocMark ∪ Mapping)

For simplicity, let us assume for now that the JUMP function just computes the identity, i.e., $\Lambda' := \Lambda$. As for the call NEXTLEVEL(Λ'), it returns the pairs (LocMark, Λ''), where

- The label set LocMark is an edge label such that there is a marker edge e labeled with LocMark that starts at some vertex of Λ';

- The level set Λ'' is formed of all vertices w at level $\text{level}(\Lambda') + 1$ that can be reached by first following a marker edge e like in the preceding bullet point, and then following some ϵ-edge. Formally, a vertex w is in Λ'' if and only if there is an edge labeled LocMark from some vertex $v \in \Lambda'$ to some vertex v', and there is an ϵ-edge from v' to w.

Remember that as the mapping DAG is normalized, we know that all edges starting at vertices of the level set Λ' are marker edges (several of which may have the same label), and for any target v' of these edges, all edges that leave v' are ϵ-edges whose targets w are at the level $\text{level}(\Lambda') + 1$.

It is easy to see that the NEXTLEVEL function can be computed efficiently.

ALGORITHM 2: Enumeration algorithm for Proposition 4.2

1: **input:** Level set $\Lambda' = \{v_1, \ldots, v_n\}$
2: **for** $j \in \{1, \ldots, n\}$ **do**
3:      $E_j \leftarrow$ outgoing edges of v_j
4:      $p_j \leftarrow 0$
5: **while** there is $1 \le j \le n$ such that $p_j < |E_j|$ **do**
6:      LocMark $\leftarrow \min_{(j : p_j < |E_j|)} E_j[p_j]$.label
7:      $\Lambda'_2 \leftarrow \emptyset$
8:      **for** $j \in \{1, \ldots, n\}$ **do**
9:          **while** $p_j < |E_j|$ and $E_j[p_j]$.label = LocMark **do**
10:              $\Lambda'_2 \leftarrow \Lambda'_2 \cup \{E_j[p_j]$.target$\}$
11:             $p_j \leftarrow p_j + 1$
12:     $\Lambda'' \leftarrow \emptyset$
13:     **for** $v' \in \Lambda'_2$ **do**
14:         **for** e outgoing edge of v' **do**
15:             $\Lambda'' \leftarrow \Lambda'' \cup \{e$.target$\}$
16:     OUTPUT(LocMark, Λ'')

PROPOSITION 4.2. *Given a leveled trimmed normalized mapping DAG G with width W, and a level set Λ', we can enumerate without duplicates all of the pairs* (LocMark, Λ'') $\in$ *NEXTLEVEL(Λ') with delay $O(W^2 \times |\text{LocMark}|)$ in an order such that* LocMark $= \emptyset$ *comes last if it is returned.*

PROOF. The algorithm is outlined as Algorithm 2. Intuitively, we simultaneously go over the sorted lists of the outgoing edges of each vertex of Λ', of which there are at most W, and we merge them. Specifically, as long as we are not done traversing all lists, we consider the smallest value of LocMark (according to the order) that occurs at the current position of one of the lists. Then, we move forward in each list until the list is empty or the edge label at the current position is no longer equal to LocMark, and we consider the set Λ'_2 of all vertices v' that are the targets of the edges that we have seen. This considers at most W^2 edges and reaches at most W vertices (which are at the same level as Λ'), and the total time spent reading edge labels is in $O(|\text{LocMark}|)$, so the process is in $O(W^2 \times |\text{LocMark}|)$ so far. Now, we consider the outgoing edges of all vertices $v' \in \Lambda'_2$ (all are ϵ-edges) and return the set Λ'' of the vertices w to which they lead: this only adds $O(W^2)$ to the running time because we consider at most W vertices v' with at most W outgoing edges each. Last, LocMark $= \emptyset$ comes last because of our assumption on the order of adjacency lists.   □

The design of Algorithm 1 is justified by the fact that for any level set Λ', the set $\mathcal{M}(\Lambda')$ can be partitioned based on the value of LocMark. Formally, we have the following claim.

CLAIM 4.3. *For any level set Λ of G that is not the final vertex, we have*

$$\mathcal{M}(\Lambda) \quad = \bigcup_{(\text{LocMark}, \Lambda'') \in \textsc{NextLevel}(\Lambda)} \{\text{LocMark} \cup \alpha \mid \alpha \in \mathcal{M}(\Lambda'')\}. \qquad (1)$$

Furthermore, this union is disjoint and nonempty, and none of its terms is empty.

PROOF. The definition of a level set and of a normalized mapping DAG ensures that we can decompose any path π from Λ to v_{f} as a marker edge e from Λ to some vertex v', an ϵ-edge from v' to some vertex w, and a path π' from w to v_{f}. Further, the set of such w is clearly a level set. Hence, the left-hand side of Equation (1) is included in the right-hand side. Conversely, given such v, v', w, and π', we can combine them into a path π, so the right-hand side is included in the left-hand side. This proves Equation (1).

We show that the union is disjoint. Recall that the definition of a leveled mapping DAG (Definition 3.8) implies that LocMark is a set of pairs whose second component is level(Λ), and that each mapping in $\mathcal{M}(\Lambda'')$ is a set of pairs whose second components are values strictly greater than level(Λ). Thus, each mapping in $\mathcal{M}(\Lambda)$ can only be obtained for the value of LocMark that is equal to the subset of the pairs of the mapping whose second component is level(Λ).

We show that the union is nonempty. This is because Λ is nonempty and its vertices must be co-accessible so they must have some outgoing marker edge, which implies that $\textsc{NextLevel}(\Lambda)$ is nonempty.

We last show that none of the terms of the union is empty. This is because, for each $(\text{LocMark}, \Lambda'') \in \textsc{NextLevel}(\Lambda)$, we know that Λ'' is nonempty because the mapping DAG is trimmed so all vertices are co-accessible. □

Thanks to this claim, we could easily prove by induction that Algorithm 1 correctly enumerates $\mathcal{M}(G)$ when JUMP is the identity function. However, this algorithm would not achieve the desired delay bounds: indeed, it may be the case that $\textsc{NextLevel}(\Lambda')$ only contains LocMark $= \emptyset$, and then the recursive call to ENUM would not make progress in constructing the mapping, so the delay would not generally be linear in the size of the mapping. To avoid this issue, we use the JUMP function to directly "jump" to a place in the mapping DAG where we can read a label different from $\emptyset$. Let us first give the relevant definitions.

Definition 4.4. Given a level set Λ in a leveled mapping DAG G, the *jump level* JL(Λ) of Λ is the first level $j \geq$ level(Λ) containing a vertex v' such that some $v \in \Lambda$ has a path to v' and such that v' is either the final vertex or has an outgoing edge with a label that is $\neq \epsilon$ and $\neq \emptyset$. In particular, we have JL$(\Lambda) =$ level(Λ) if some vertex in Λ already has an outgoing edge with such a label, or if Λ is the singleton set containing only the final vertex.

The *jump set* of Λ is then JUMP$(\Lambda) := \Lambda$ if JL$(\Lambda) =$ level(Λ), and otherwise JUMP(Λ) is formed of all vertices at level JL(Λ) to which some $v \in \Lambda$ have a directed path whose last edge is labeled ϵ. This ensures that JUMP(Λ) is always a level set.

Example 4.5. In the mapping DAG in Figure 1, we have JL$(\{(q_2, 3), (q_5, 3)\}) = 5$, as the reachable node $(q_9, 5)$ has an outgoing edge labeled $\{(\dashv x, 5)\}$. The set JUMP$(\{(q_2, 3), q_5, 3)\})$ is $\{(q_2, 5), (q_9, 5)\}$, as $(q_2, 5)$ is reachable from $(q_2, 3)$ and $(q_9, 5)$ is reachable from $(q_5, 3)$.

The definition of JUMP ensures that we can jump from Λ to JUMP(Λ) when enumerating mappings, and it will not change the result because we only jump over ϵ-edges and $\emptyset$-edges.

CLAIM 4.6. *For any level set Λ of G, we have $\mathcal{M}(\Lambda) = \mathcal{M}(\textsc{Jump}(\Lambda))$.*

PROOF. As JUMP(Λ) contains all vertices from level JL(Λ) that can be reached from Λ, any path π from a vertex $u \in \Lambda$ to the final vertex can be decomposed into a path π_{uw} from u to a vertex

$w \in \text{Jump}(\Lambda)$ and a path π_{wv} from w to v. By definition of $\text{Jump}(\Lambda)$, we know that all edges in π_{uw} are labeled with ϵ or $\emptyset$, so $\mu(\pi) = \mu(\pi_{wv})$. Hence, we have $\mathcal{M}(\Lambda) \subseteq \mathcal{M}(\text{Jump}(\Lambda))$.

Conversely, given a path π_{wv} from a vertex $w \in \text{Jump}(\Lambda)$ to the final vertex, the definition of $\text{Jump}(\Lambda)$ ensures that there is a vertex $u \in \Lambda$ and a path π_{uw} from u to w, which again consists only of ϵ-edges or $\emptyset$-edges. Hence, letting π be the concatenation of π_{uw} and π_{wv}, we have $\mu(\pi_{wv}) = \mu(\pi)$ and π is a path from Λ to the final vertex. Thus, we have $\mathcal{M}(\text{Jump}(\Lambda)) \subseteq \mathcal{M}(\Lambda)$, concluding the proof.   □

Claims 4.3 and 4.6 imply that Algorithm 1 is correct with this implementation of Jump.

PROPOSITION 4.7. $\text{Enum}(\{v_0\}, \emptyset)$ *correctly enumerates* $\mathcal{M}(G)$ *(without duplicates).*

PROOF. We show the stronger claim that for every level set Λ, and for every set of labels Mapping, we have that $\text{Enum}(\Lambda, \text{Mapping})$ enumerates (without duplicates) the set Mapping $\uplus$ $\mathcal{M}(\Lambda) := \{\text{Mapping} \cup \alpha \mid \alpha \in \mathcal{M}(\Lambda)\}$. The base case is when Λ is the final vertex, and then $\mathcal{M}(\Lambda) = \{\{\}\}$ and the algorithm correctly returns $\{\text{Mapping}\}$.

For the induction case, let us consider a level set Λ that is not the final vertex, and some set of labels Mapping. We let $\Lambda' := \text{Jump}(\Lambda)$, and by Claim 4.6 we have that $\mathcal{M}(\Lambda') = \mathcal{M}(\Lambda)$. Now we know by Claim 4.3 that $\mathcal{M}(\Lambda')$ can be written as in Equation (1) and that the union is disjoint; the algorithm evaluates this union. So it suffices to show that, for each $(\text{LocMark}, \Lambda'') \in \text{NextLevel}(\Lambda')$, the corresponding iteration of the **for** loop enumerates (without duplicates) the set $(\text{Mapping} \cup \text{LocMark}) \uplus \mathcal{M}(\Lambda'')$. By the induction hypothesis, the call $\text{Enum}(\text{Jump}(\Lambda'), \text{Mapping} \cup \text{LocMark})$ enumerates (without duplicates) the set $(\text{Mapping} \cup \text{LocMark}) \uplus \mathcal{M}(\text{Jump}(\Lambda''))$. So this establishes that the algorithm is correct.   □

What is more, Algorithm 1 now achieves the desired delay bounds, as we will show. Of course, this relies on the fact that the Jump function can be efficiently precomputed and evaluated. We only state this fact for now, and prove it in the next section.

PROPOSITION 4.8. *Given a leveled mapping DAG G with width W and depth D, we can preprocess G in time $O(D \times W^{\omega+1})$ such that, given any level set Λ of G, we can compute the jump set $\text{Jump}(\Lambda)$ of Λ in time $O(W^2)$.*

We can now conclude the proof of Theorem 4.1 by showing that the preprocessing and delay bounds are as claimed. For the preprocessing, this is clear: we do the preprocessing in $O(|G|)$ presented at the beginning of the section (i.e., trimming, and computing the sorted adjacency lists), followed by that of Proposition 4.8. For the delay, we claim the following.

CLAIM 4.9. *Algorithm 1 has delay $O(W^2 \times (r + 1))$, where r is the size of the mapping of each produced path. In particular, the delay is independent of the size of G.*

PROOF. Let us first bound the delay to produce the first solution. When we enter the Enum function, we call the Jump function to produce Λ' in time $O(W^2)$ by Proposition 4.8, and either Λ' is the final vertex or some vertex in Λ' must have an outgoing edge with a label different from $\emptyset$. Then we enumerate $\text{NextLevel}(\Lambda')$ with delay $O(W^2 \times |\text{LocMark}|)$ for each LocMark using Proposition 4.2. Remember that Proposition 4.2 ensures that the label $\emptyset$ comes last; so by definition of Jump, the first value of LocMark that we consider is different from $\emptyset$. At each round of the **for** loop, we recurse in constant time: in particular, we do not copy Mapping when writing LocMark $\cup$ Mapping, as we can represent the set simply as a linked list. Eventually, after $r + 1$ calls, by definition of a leveled mapping DAG, Λ must be the final vertex, and then we output a mapping of size r in time $O(r)$: the delay is indeed in $O(W^2 \times (r + 1))$ because the sizes of the values of LocMark seen along the path sum up to r, and the unions of LocMark and Mapping are always disjoint by definition of a mapping DAG.

Let us now bound the delay to produce the next solution. To do so, we will first observe that when enumerating a mapping of cardinality r, then the size of the recursion stack is always $\leq r + 1$. This is because Proposition 4.2 ensures that the value LocMark $= \emptyset$ is always considered last in the **for** loop on NextLevel(Λ'). Thanks to this, every call to Enum where LocMark $= \emptyset$ is actually a tail recursion, and we can avoid putting another call frame on the call stack using tail recursion elimination. This ensures that each call frame on the stack (except possibly the last one) contributes to the size of the currently produced mapping so that indeed when we reach the final vertex of G the call stack is no greater than the size of the mapping that we produce.

Now, let us use this fact to bound the delay between consecutive solutions. When we move from one solution to another, it means that some **for** loop has moved to the next iteration somewhere in the call stack. To identify this, we must unwind the stack: when we produce a mapping of size r, we unwind the stack until we find the next **for** loop that can move forward. By our observation on the size of the stack, the unwinding takes time $O(r)$ with r is the size of the previously produced mapping; so we simply account for this unwinding time as part of the computation of the previous mapping. Now, to move to the next iteration of the **for** loop and do the computations inside the loop, we spend a delay $O(W^2 \times |\text{LocMark}|)$ by Proposition 4.2. Let r' be the current size of Mapping, including the current LocMark. The **for** loop iteration finishes with a recursive call to Enum, and we can reapply our argument about the preceding first solution to argue that this call identifies a mapping of some size r'' in delay $O(W^2 \times (r'' + 1))$. However, because the argument Mapping to the recursive call had size r', the mapping that is enumerated actually has size $r' + r''$ and it is produced in delay $O(W^2 \times (r'' + 1) + r')$. This means that the overall delay to produce the next solution is indeed in $O(W^2 \times (r + 1))$, where r is the size of the mapping that is produced, which concludes the proof. □

Memory usage. We briefly discuss the *memory usage* of the enumeration phase, i.e., the maximal amount of working memory that we need to keep throughout the enumeration phase, not counting the precomputation phase. Indeed, in enumeration algorithms, the memory usage can generally grow to be very large even if one adds only a constant amount of information at every step. We will show that this does not happen here, and that the memory usage throughout the enumeration remains polynomial in $\mathcal{A}$ and constant in the input document size.

All of our memory usage during enumeration is in the call stack, and thanks to tail recursion elimination (see the proof of Claim 4.9), we know that the stack depth is at most $r + 1$, where r is the size of the produced mapping as in the statement of Theorem 4.1. The local space in each stack frame must store Λ' and Λ'', which have size $O(W)$, and the status of the enumeration of NextLevel in Proposition 4.2, i.e., for every vertex $v \in \Lambda'$, the current position in its adjacency list: this also has total size $O(W)$, so the total memory usage of these structures over the whole stack is in $O((r + 1) \times W)$. Last, we must also store the variables Mapping and LocMark, but their total size of the variables LocMark across the stack is clearly r, and the same holds of Mapping because each occurrence is stored as a linked list (with a pointer to the previous stack frame). Hence, the total memory usage is $O((r + 1) \times W)$, i.e., $O((|\mathcal{V}| + 1) \times |Q|)$ in terms of the extended VA.

5 JUMP FUNCTION

The only missing piece in the enumeration scheme of Section 4 is the proof of Proposition 4.8. We first explain the preprocessing for the Jump function and then the computation scheme.

Preprocessing scheme. Recall the definition of the jump level JL(Λ) and jump set Jump(Λ) of a level set Λ (Definition 4.4). We assume that we have precomputed in $O(|G|)$ the mapping level associating each vertex v to its level level(v), as well as, for each level i, the list of the vertices v such that level(v) = i.

The first part of the preprocessing is then to compute, for every individual vertex v, the jump level $\mathsf{JL}(v) := \mathsf{JL}(\{v\})$, i.e., the minimal level containing a vertex v' such that v' is reachable from v and v' is either the final vertex or has an outgoing edge that is neither an ϵ-edge nor an $\emptyset$-edge. We claim the following.

CLAIM 5.1. *We can precompute in $O(D \times W^2)$ the jump level $\mathsf{JL}(v)$ of all vertices v of G.*

PROOF. This construction can be performed iteratively from the final vertex v_f to the initial vertex v_0: we have $\mathsf{JL}(v_\mathrm{f}) := \mathrm{level}(v_\mathrm{f})$ for the final vertex v_f, we have $\mathsf{JL}(v) := \mathrm{level}(v)$ if v has an outgoing edge that is not an ϵ-edge or an $\emptyset$-edge, and otherwise we have $\mathsf{JL}(v) := \min_{v \to w} \mathsf{JL}(w)$.

This computation can be performed along a reverse topological order, which by Cormen et al. [10, Section 22.4] takes linear time in G. However, note that G has at most $D \times W$ vertices, and we only traverse ϵ-edges and $\emptyset$-edges: we just check the existence of edges with other labels but do not traverse them. Now, as each vertex has at most W outgoing edges labeled $\emptyset$ and at most W outgoing edges labeled ϵ, the number of edges in the DAG that we actually traverse is only $O(D \times W^2)$, which shows our complexity bound and concludes the proof.   □

The second part of the preprocessing is to compute, for each level i of G, the *reachable levels* $\mathrm{Rlevel}(i) := \{\mathsf{JL}(v) \mid \mathrm{level}(v) = i\}$, which we can clearly do in linear time in the number of vertices of G, i.e., in $O(D \times W)$. Note that the definition clearly ensures that we have $|\mathrm{Rlevel}(i)| \leq W$.

Example 5.2. In Figure 1, the jumping level for nodes $(q_1, 3)$ and $(q_2, 3)$ is 6 and the jumping level for nodes $(q_5, 3)$ and $(q_6, 3)$ is 5. Hence, the set of reachable levels $\mathrm{Rlevel}(3)$ for level 3 is $\{5, 6\}$.

Last, the third step of the preprocessing is to compute a reachability matrix from each level to its reachable levels. Specifically, for any two levels $i < j$ of G, let $\mathrm{Reach}(i, j)$ be the Boolean matrix of size at most $W \times W$ that describes, for each (u, v) with $\mathrm{level}(u) = i$ and $\mathrm{level}(v) = j$, whether there is a path from u to v whose last edge is labeled ϵ. We cannot afford to compute all of these matrices, but we claim that we can efficiently compute a subset of them, which will be enough for our purposes.

CLAIM 5.3. *We can precompute in time $O(D \times W^{\omega+1})$ the matrices $\mathrm{Reach}(i, j)$ for all pairs of levels $i < j$ such that $j \in \mathrm{Rlevel}(i)$.*

PROOF. We compute the matrices in decreasing order on i, then for each fixed i in arbitrary order on j:

- If $j = i$, then $\mathrm{Reach}(i, j)$ is the identity matrix;
- If $j = i + 1$, then $\mathrm{Reach}(i, j)$ can be computed from the edge relation of G in time $O(W \times W)$, because it suffices to consider the edges labeled $\emptyset$ and ϵ between levels i and j;
- If $j > i + 1$, then $\mathrm{Reach}(i, j)$ is the product of $\mathrm{Reach}(i, i + 1)$ and $\mathrm{Reach}(i + 1, j)$, which can be computed in time $O(W^\omega)$.

In the last case, the crucial point is that $\mathrm{Reach}(i + 1, j)$ has already been precomputed, because we are computing Reach in decreasing order on i, and because we must have $j \in \mathrm{Rlevel}(i + 1)$. Indeed, if $j \in \mathrm{Rlevel}(i)$, then there is a vertex v with $\mathrm{level}(v) = i$ such that $\mathsf{JL}(v) = j$, and the inductive definition of JL implies that v has an edge to a vertex w such that $\mathrm{level}(w) = i + 1$ and $\mathsf{JL}(v) = \mathsf{JL}(w) = j$, which witnesses that $j \in \mathrm{Rlevel}(i + 1)$.

The total running time of this scheme is in $O(D \times W^{\omega+1})$: indeed, we consider each of the D levels of G, we compute at most W matrices for each level of G because we have $|\mathrm{Rlevel}(i)| \leq W$ for any i, and each matrix is computed in time at most $O(W^\omega)$.   □

Evaluation scheme. We can now describe our evaluation scheme for the jump function. Given a level set Λ, we wish to compute $\textsc{Jump}(\Lambda)$. Let i be the level of Λ, and let j be $\mathsf{JL}(\Lambda)$ that we compute

as $\min_{v \in \Lambda}$ JL(v) in $O(W)$ time. If $j = i$, then JUMP$(\Lambda) = \Lambda$ and there is nothing to do. Otherwise, by definition, there must be $v \in \Lambda$ such that JL$(v) = j$, so v witnesses that $j \in$ Rlevel(i), and we know that we have precomputed the matrix Reach(i, j). Now JUMP(Λ) are the vertices at level j to which the vertices of Λ (at level i) have a directed path whose last edge is labeled ϵ, which we can simply compute in time $O(W^2)$ by unioning the lines that correspond to the vertices of Λ in the matrix Reach(i, j).

This concludes the proof of Proposition 4.8 and completes the presentation of our scheme to enumerate the set captured by mapping DAGs (Theorem 4.1). Together with Section 3, this proves Theorem 1.1 in the case of extended sequential VAs.

6 FROM EXTENDED SEQUENTIAL VAS TO GENERAL SEQUENTIAL VAS

In this section, we adapt our main result (Theorem 1.1) to work with sequential nonextended VAs rather than sequential extended VAs. Remember that we cannot tractably convert nonextended VAs into extended VAs [12, Proposition 4.2], so we must modify our construction in Sections 3 throug 5 to work with sequential nonextended VAs directly. Our general approach will be the same: compute the mapping DAG and trim it like in Section 3, then precompute the jump level and jump set information as in Section 5, and apply the enumeration scheme of Section 4. The difficulty is that nonextended VAs may assign multiple markers at the same word position by taking multiple variable transitions instead of one single ev-transition. Hence, when enumerating all possible values for LocMark in Algorithm 1, we need to consider all possible sequences of variable transitions. The challenge is that there may be many different transition sequences that assign the same set of markers, which could lead to duplicates in the enumeration. Thus, our goal will be to design a replacement to Proposition 4.2 for nonextended VAs, i.e., enumerate possible values for LocMark at each level without duplicates.

We start as in Section 3 by computing the product DAG G of $\mathcal{A}$ and of the input document $d = d_0 \cdots d_{n-1}$ with vertex set $Q \times \{0, \ldots, n\} \cup \{v_f\}$ with $v_f := (\bullet, n+1)$ for some fresh value $\bullet$, and with the following edge set:

- For every letter-transition (q, a, q') of $\mathcal{A}$, for every $0 \le i < |d|$ such that $d_i = a$, there is an ϵ-edge from (q, i) to $(q', i+1)$;
- For every variable-transition (q, m, q') of $\mathcal{A}$ (where m is a marker), for every $0 \le i \le |d|$, there is an edge from (q, i) to (q', i) labeled with $\{(m, i)\}$.
- For every final state $q \in F$, an ϵ-edge from (q, n) to v_f.

The initial vertex of G is $(q_0, 0)$, and the final vertex is v_f. Note that the edge labels are now always singleton sets or ϵ; in particular, there are no longer any $\emptyset$-edges.

We can then adapt most of Claim 3.4: the product DAG is acyclic because all letter-transitions make the second component increase, and because we know that there cannot be a cycle of variable transitions in the input sequential VA $\mathcal{A}$ (remember that we assume VAs to be trimmed). We can also trim the mapping DAG in linear time as before, and Claim 3.5 also adapts to show that the resulting mapping DAG correctly captures the mappings that we wish to enumerate. Last, as in Claim 3.9, the resulting mapping DAG is still leveled, the depth D (number of levels) is still $|d| + 1$, and the width W (maximal size of a level) is still $\le |Q|$; we will also define the *complete width* W_c of G in this section as the maximum, over all levels i, of the sum of the number of vertices in level i, and of the number of *edges* with a source vertex in level i. Formally, writing $G = (V, v_0, v_f, E)$, and writing D the depth of G, we have $W_c := \max_{1 \le i \le D} |\{v \in V \mid \text{level}(v) = i\}| + |\{(s, x, t) \in E \mid \text{level}(s) = i\}|$. Notice that we have $W_c \le |\mathcal{A}|$. The main change in Section 3 is that the mapping DAG is no longer normalized, i.e., we may follow several marker edges in succession (staying at the same level) or follow several ϵ-edges in succession (moving to the next level each

time). Because of this, we change Definition 3.11 and redefine *level sets* to mean any nonempty set of vertices that are at the same level.

We then reuse the enumeration approach of Sections 4 and 5. Even though the mapping DAG is no longer normalized, it is not hard to see that with our new definition of level sets we can reuse the jump function from Section 5 as-is, and we can also reuse the general approach of Algorithm 1. However, to accommodate for the different structure of the mapping DAG, we will need a new definition for NextLevel: instead of following exactly one marker edge before an ϵ-edge, we want to be able to follow any (possibly empty) path of marker edges before an ϵ-edge. We formalize this notion as an S^+-*path*.

Definition 6.1. For S^+ a set of labels, an S^+-*path* in the mapping DAG G is a path of $|S^+|$ edges that includes no ϵ-edges and where the labels of the path are exactly the elements of S^+ in some arbitrary order. Recall that the definition of a mapping DAG (Definition 3.1) ensures that there can be no duplicate labels on the path, and that the start and end vertices of an S^+-path must have the same level because no ϵ-edge is traversed in the path.

For Λ a level set, NextLevel(Λ) is the set of all pairs (S^+, Λ''), where

- S^+ is a set of labels such that there is an S^+-path that goes from some vertex v of Λ to some vertex v' that has an outgoing ϵ-edge;
- Λ'' is the level set containing exactly the vertices w that are targets of these ϵ-edges, i.e., there is an S^+-path from some vertex $v \in \Lambda$ to some vertex v', and there is an ϵ-edge from v' to w.

Note that these definitions are exactly equivalent to what we would obtain if we converted $\mathcal{A}$ to an extended VA and then used our original construction. This directly implies that the modified enumeration algorithm is correct (i.e., Proposition 4.7 extends). In particular, the modified algorithm still uses the jump pointers as computed in Section 5 to jump over positions where the only possibility is $S^+ = \emptyset$, i.e., positions where the sequential VA make no variable transitions. The only thing that remains is to establish the delay bounds, for which we need to enumerate NextLevel efficiently without duplicates (and replace Proposition 4.2). To present our method for this, we will introduce the *alphabet size B* as the maximal number, over all levels j of the mapping DAG G, of the different labels that can occur in marker edges between vertices at level j; in our construction, this value is bounded by the number of different markers, i.e., $B \leq 2|\mathcal{V}|$. We can now state the claim that we will prove later in the section.

THEOREM 6.2. *Given a leveled trimmed mapping DAG G with complete width W_c and alphabet size B, and a level set Λ', we can enumerate without duplicates all of the pairs $(S^+, \Lambda'') \in$ NextLevel(Λ') with delay $O(W_c \times B^2)$ in an order such that $S^+ = \emptyset$ comes last if it is returned.*

With this runtime, the delay of Theorem 4.1 becomes $O((r + 1) \times (W^2 + W_c \times B^2))$, and we know that $W_c \leq |\mathcal{A}|$, that $W \leq |Q|$, that $r \leq |\mathcal{V}|$, and that $B \leq 2|\mathcal{V}|$; so this leads to the overall delay of $O(|\mathcal{V}| \times (|Q|^2 + |\mathcal{A}| \times |\mathcal{V}|^2))$ in Theorem 1.1.

The idea to prove Theorem 6.2 is to use a general approach called *flashlight search* [21, 26]: we will use a search tree on the possible sets of labels on $\mathcal{V}$ to iteratively construct the set S^+ that can be assigned at the current position, and we will avoid useless parts of the search tree by using a lemma to efficiently check if a partial set of labels can be extended to a solution. To formalize the notion of extending a partial set, we will need the notion of S^+/S^--*paths*.

Definition 6.3. For S^- and S^+ two disjoint sets of labels, an S^+/S^--*path* in the mapping DAG G is a path of edges that includes no ϵ-edges, that includes no edges with a label in S^-, and where every label of S^+ is seen exactly once along the path.

Note that when $S^+ \cup S^-$ contains all labels used in G, the notions of S^+/S^--path and S^+-path coincide, but if G contains some labels not in $S^+ \cup S^-$, then an S^+/S^--path is free to use them or not, whereas an S^+-path cannot use them. The key to prove Theorem 6.2 is to efficiently determine if S^+/S^--paths exist: we formalize this as a lemma that we will apply to the mapping DAG G restricted to the current level (in particular removing ϵ-edges).

LEMMA 6.4. *Let G be a mapping DAG with no ϵ-edges, and let V be its vertex set. Given a nonempty set $\Lambda' \subseteq V$ of vertices of G and given two disjoint sets of labels S^+ and S^-, we can compute in time $O(|G| \times (|S^+| + |S^-|))$ the set $\Lambda'_2 \subseteq V$ of vertices v such that there is an S^+/S^--path from one vertex of Λ' to v.*

PROOF. In a first step, we delete from G all edges with a label that is in S^-. This can be done in time $O(|G| \times |S^-|)$ and ensures that no path that we consider contains any label from S^-. Hence, we can completely ignore S^- in what follows.

In a second step, we add a fresh source vertex s_0 and edges with a fresh label l_0 from s_0 to each vertex in Λ', we add l_0 to S^+, and we set $\Lambda' := \{s\}$. This allows us to assume that the set Λ' is a singleton $\{s\}$.

In a third step, we traverse G in linear time from s_0 with a breadth-first search to remove all vertices that are not reachable from s_0. Hence, we can now assume that every vertex in G is reachable from s_0; in particular, every vertex except s_0 has at least one predecessor.

Now, we follow a topological order on G to give a label $\chi(w) \subseteq S^+$ to each vertex $w \in V$ with predecessors $w_1, \ldots, w_p$ and to give a label $\chi(w_i, w) \subseteq S^+$ to each edge (w_i, w) of G, as follows:

$$
\begin{aligned}
\chi(s) \quad &:= \quad \emptyset \\
\chi(w_i, w) \quad &:= \quad (\chi(w_i) \cup \{\mu(w_i, w)\}) \cap S^+ \\
\chi(w) \quad &:= \quad \begin{cases} \chi(w_i, w) & \text{if } w \text{ has a predecessor } w_i \text{ with } \chi(w_i, w) = \bigcup_{1 \leq j \leq p} \chi(w_j, w) \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}
$$

The topological order can be computed in time $O(|G|)$ by Cormen et al. [10, Section 22.4], and computing χ along this order takes time $O(|G| \times |S^+|)$.

Intuitively, the labels assigned to a vertex w or an edge (w_i, w) correspond to the subset of labels from S^+ that are read on a path starting at s_0 and using w as the last vertex (respectively, (w_i, w) as last edge). However, we explicitly label a vertex w with $\emptyset$ if there are two paths starting at s_0 that have seen a different subset of S^+ to reach w. Indeed, as we know that any label can occur at most once on each path, such vertices and edges can never be part of a path that contains all labels from S^+. We will formalize this intuition in the following.

The key claim is that, for every vertex v, there is an S^+/S^--path from s_0 to v if and only if $\chi(v) = S^+$. We first show the forward direction of the key claim. Assume that $\chi(v) = S^+$. We construct a path P by going backward starting from v. We initialize the current vertex w to be v. Now, as long as $\chi(w)$ is nonempty, we pick a predecessor w_i with $\chi(w_i, w) = \chi(w)$, and we know that either $\chi(w_i, w) = \chi(w_i)$ or $\chi(w_i, w) = \chi(w_i) \cup \{\mu(w_i, w)\}$ with $\mu(w_i, w) \in S^+$, and then we assign w_i as our current vertex w. We repeat this process until we reach a current vertex w_0 with $\chi(w) = \emptyset$, which must eventually happen: the DAG is acyclic, and all vertices except s_0 must have a predecessor, and we know by definition that $\chi(s) = \emptyset$. As all elements of S^+ were in $\chi(w)$, they were all witnessed on P, so we know that P is an S^+/S^--path from w_0 to v. Now, we know that there is a path P' from s_0 to w_0 thanks to our third preprocessing step, and we know that P' uses no elements from S^- by our assumption on the DAG; so the concatenation of P' and P is an S^+/S^--path from s_0 to v.

We now show the converse direction of the key claim. Assume that there is an S^+/S^--path $P = v_1, \ldots, v_r$ from $v_1 = s_0$ to a vertex $v_r = v$. We show by induction that $\chi(v_i)$ is exactly the set of the labels of S^+ that have been seen so far on the path from s_0 to v_i. For $v_1 = s_0$, this is true by definition. For $i > 1$, we claim that $\chi(v_i) = \chi(v_{i-1}, v_i)$. By way of contradiction, assume this were not the case. This means that $\chi(v_{i-1}, v_i)$ is not the union of the $\chi(v'_{i-1}, v_i)$, where v'_{i-1} ranges over the predecessors of v_i. Hence, there is a specific choice of a predecessor $v'_{i-1} \neq v_{i-1}$ such that $\chi(v'_{i-1}, v_i)$ contains an $x \in S^+$ that does not appear in $\chi(v_{i-1}, v_i)$. By the induction hypothesis, $\chi(v_{i-1})$ contains exactly the labels of S^+ that were seen on the path from s_0 to v_{i-1}, and as x is not in $\chi(v_{i-1}, v_i)$, we know that x does not appear on the path $v_1 \ldots v_i$.

Now, we know that x cannot appear on the path $v_i \ldots v_r$ either. Indeed, by the definition of χ, the fact that $x \in \chi(v'_{i-1}, v_i)$ must mean that there is a path from s_0 to v_i (via v'_{i-1}) where the label x appears. Now, the definition of a mapping DAG ensures that x can occur only once on every path of G. Thus, it cannot also appear on the path $v_i \ldots v_r$ that starts at v_i. Hence, x does not appear in the path P at all, and this contradicts the fact that P is an S^+/S^--path.

Thus, we have shown by contradiction that we have indeed $\chi(v_i) = \chi(v_{i-1}, v_i)$. This means that $\chi(v_i)$ is exactly the set of labels of S^+ that have been seen so far on the path from s_0 to v_i, so we have established the claim made in the inductive step. But then, since all elements of S^+ appear on edges in P and are thus added iteratively in the construction of the $\chi(v_i)$, we have $S^+ = \chi(v_r) = \chi(v)$ as desired. This establishes the converse direction of the key claim, and so the key claim is proven.

Hence, thanks to the key claim, once we have computed the labeling χ, we can compute in time $O(|G| \times |S^+|)$ the set Λ'_2 by simply finding all vertices v with $\chi(v) = S^+$. This concludes the proof.   □

We can now use Lemma 6.4 to prove Theorem 6.2.

Proof. Clearly if Λ' is the singleton level set consisting only of the final vertex, then the set to enumerate is empty and there is nothing to do. Hence, in the sequel, we assume that this is not the case.

Let p be the level of Λ'. We call $\mathcal{K}$ the set of possible labels at level p, with $|\mathcal{K}|$ being no greater than the alphabet size B of G. We fix an arbitrary order $m_1 < m_2 < \cdots < m_r$ on the elements of $\mathcal{K}$. Remember that we want to enumerate NextLevel(Λ'), i.e., all pairs (S^+, Λ'') of a subset S^+ of $\mathcal{K}$ such that there is an S^+-path in G from a vertex in Λ' to a vertex v' (which will be at level p) with an outgoing ϵ-edge, and the set Λ'' of the targets of these ϵ-edges (at level $p + 1$). Let us consider the complete decision tree $T_\mathcal{K}$ on $m_1, \ldots, m_r$: it is a complete binary tree of height $r + 1$, where, for all $1 \leq i \leq r$, every edge at height i is labeled with $+m_i$ if it is a right child edge and with $-m_i$ otherwise. For every node n in the tree, we consider the path from the root of $T_\mathcal{K}$ to n, and call the *positive set* P_n of n the labels m such that $+m$ appears in the path, and the *negative set* N_n of n the labels m such that $-m$ appears in the path: it is immediate that for every node n of $T_\mathcal{K}$ the sets P_n and N_n are a partition of $\{m_1, \ldots, m_j\}$ where $0 \leq j \leq r$ is the depth of n in $T_\mathcal{K}$.

We say that a node n of $T_\mathcal{K}$ is *good* if there is some P_n/N_n-path in G starting at a vertex of Λ' and leading to a vertex that has an outgoing ϵ-edge. Our goal of determining NextLevel(Λ') can then be rephrased as finding the set of all positive sets P_n for all good leaves n of $T_\mathcal{K}$ (and the corresponding level set Λ''), because there is a clear one-to-one correspondence that sends each subset $S \subseteq \mathcal{K}$ to a leaf n of $T_\mathcal{K}$ such that $P_n = S$.

Observe now that we can use Lemma 6.4 to determine in time $O(|W_c| \times |\mathcal{K}|)$, given a node n of $T_\mathcal{K}$, whether it is good or bad: call the procedure on the subgraph of G that is induced by level p (it has size $\leq W_c$) and with the sets $S^+ := P_n$ and $S^- := N_n$ (their union has cardinality $\leq |\mathcal{K}|$), then check in G whether one of the vertices returned by the procedure has an outgoing ϵ-edge.

A naive solution to find the good leaves would then be to test them one by one using Lemma 6.4; however, a more efficient idea is to use the structure of $T_\mathcal{K}$ and the following facts:

- *The root of $T_\mathcal{K}$ is always good.* Indeed, G is trimmed, so we know that any $v \in \Lambda'$ has a path to some ϵ-edge.
- *If a node is good then all of its ancestors are good.* Indeed, if n' is an ancestor of n, and there is a P_n/N_n-path in G starting at a vertex of Λ', then this path is also a $P_{n'}/N_{n'}$ path, because $P_{n'} \subseteq P_n$ and $N_{n'} \subseteq N_n$.
- *If a node n' is good, then it must have at least one good descendant leaf n.* Indeed, taking any $P_{n'}/N_{n'}$-path that witnesses that n' is good, we can take the leaf n to be such that $P_n \supseteq P_{n'}$ is exactly the set of labels that occur on the path so that the same path witnesses that n is indeed good.

Our flashlight search algorithm will rely on these facts. We explore $T_\mathcal{K}$ depth-first, constructing it on-the-fly as we visit it, and we use Lemma 6.4 to guide our search: at a node n of $T_\mathcal{K}$ (inductively assumed to be good), we call Lemma 6.4 on its two children to determine which of them are good (from the preceding facts, at least one of them must be), and we explore recursively the first good child, and then the second good child if there is one. When the two children are good, we first explore the child labeled $+m$ before exploring the child labeled $-m$: this ensures that if the empty set is produced as a label set in NextLevel(Λ'), then we always enumerate it last, as we should. Once we reach a leaf n (inductively assumed to be good), we then output its positive set of labels P_n.

It is clear that the algorithm only enumerates label sets that occur in NextLevel(Λ'). What is more, as the set of good nodes is upward-closed in $T_\mathcal{K}$, the depth-first exploration visits all good nodes of $T_\mathcal{K}$, so it visits all good leaves and produces all label sets that should occur in NextLevel(Λ'). Now, the delay is bounded by $O(|W_c| \times |\mathcal{K}|^2)$: indeed, whenever we are exploring at any node n, we know that the next good leaf will be reached in at most $2|\mathcal{K}|$ calls to the procedure of Lemma 6.4, and we know that the subgraph of G induced by level p has size bounded by the complete width W_c of G so each call takes time $O(|W_c| \times |\mathcal{K}|)$, including the time needed to verify if any of the reachable vertices v' has an outgoing ϵ-edge: this establishes the delay bound of $O(|W_c| \times B^2)$ that we claimed. Last, while doing this verification, we can produce the set Λ'' of the targets of these edges in the same time bound. This set Λ'' is correct because any such vertex v' has an outgoing ϵ-edge and there is a P_n/N_n-path from some vertex $v \in \Lambda'$ to v'. Now, as $P_n \cup N_n = \mathcal{K}$ and the path cannot traverse an ϵ-edge, these paths are actually P_n-paths (i.e., they exactly use the labels in P_n), so Λ'' is indeed the set that we wanted to produce according to Definition 6.1. This concludes the proof. □

Memory usage. The recursion depth of Algorithm 1 on general sequential VAs is unchanged, and we can still eliminate tail recursion for the case LocMark $= \emptyset$ as we did in Section 4.

The local space must now include the local space used by the enumeration scheme of NextLevel, of which there is an instance running at every level on the stack. We need to remember our current position in the binary search tree: assuming that the order of labels is fixed, it suffices to remember the current positive set P_n plus the last label in the order on $\mathcal{K}$ that we use, with all other labels being implicitly in N_n. This means that we store one label per level (the last label), plus the positive labels, so their total number in the stack is at most the total number of markers, i.e., $O(|\mathcal{V}|)$. Hence, the structure of Theorem 6.2 has no effect on the memory usage.

The space usage must also include the space used for one call to the construction of Lemma 6.4, only one instance of which is running at every given time. This space usage is clearly in $O(|Q| \times |V|)$, so this additive term has again no impact on the memory usage. Hence, the memory usage of

our enumeration algorithm is the same as in Section 4, i.e., $O((r + 1) \times W)$, or $O((|\mathcal{V}| + 1) \times |Q|)$ in terms of the VA.

7 EXPERIMENTAL VALIDATION

Having concluded the proof of our main result, we move on in this section to an experimental study of a prototype implementation of our method. A first direct implementation of our algorithm was developed by Rémi Dupré during his master thesis, which we further optimized to achieve better results, particularly to improve the handling of the reachability matrices and the space usage. In this section, we present this optimized implementation and show how it performs on several benchmarks. Our software is written in Rust and is freely available online[3] under the BSD 3-clause license.

Overall design. Our implementation enumerates the solutions of the evaluation of a nondeterministic sequential VA over a word. The nondeterministic sequential VA is given in the input as a regex-formula. This regex-formula is translated into a nondeterministic sequential VA using a variant of Glushkov's algorithm. Note that our implementation uses VAs so the underlying algorithm could work with any regular spanner, and not only with hierarchical regular spanners [11, Theorem 4.6]. As for the input document, it is provided as a text file.

Our implementation follows the different parts of the algorithm presented in the article. The preprocessing phase comprises (i) the construction of the mapping DAG as described in Section 3 and modified for nonextended VAs in Section 6, and (ii) the construction of the jump function described in Section 5 and all necessary matrices. The enumeration phase explores the DAG as described in Section 4 and modified for nonextended VAs in Section 6. In particular, we use the flashlight search approach described in Section 6.

7.1 Optimizations

Our optimizations focus on three main problems: efficiently managing the mapping DAG during the preprocessing phase, managing the reachability matrices that we build at the end of the preprocessing phase, and optimizing the enumeration phase.

Efficient representation of the mapping DAG and efficient exploration. The first stage of the preprocessing phase is to compute the mapping DAG. This DAG is efficiently represented as a bitmap[4] in which we store which states are reachable at each position of the input document. To save space, the implementation does not actually store any edges of the DAG, as the edges can be reconstructed on-the-fly from the automaton and input string.

The second stage is to make this DAG trimmed by exploring it to remove the vertices that are not co-accessible, i.e., those that have no path to the final vertex.

Implementation of the matrices. The third stage of the preprocessing is to compute the reachability matrices that are necessary for the jump function, which requires many Boolean matrix multiplications. We considered using optimized implementations of matrix multiplication, but these are generally designed for floating-point numbers rather than Boolean values, so using them would significantly increase the memory usage. As memory space tends to be an important bottleneck in our implementation, we instead implemented our own matrix multiplication code: it uses the naive matrix multiplication algorithm with three nested loops, but we optimized it for Boolean matrices as follows. We store matrices as bitvectors and pad their width to 8, 16, 32, or a multiple

---

[3]https://github.com/PoDMR/enum-spanner-rs.
[4]The bitmap contains a single bit for each pair $(q, i) \in Q \times \{0, \ldots, |d|\}$ that says whether the node is part of the trimmed mapping DAG or not. Padding is applied to ensure that each level starts at a machine word boundary.

of 64, which reduces their memory usage. Further, we use fast bitwise operations in the inner loop of the matrix multiplication algorithm, which speeds up the multiplication of large matrices by a factor of up to 64. With this vectorized implementation, the multiplication time grows roughly like n^2 for matrices with width up to 64.

Enumeration phase. After these optimizations to the three stages of the preprocessing phase, our implementation performs the enumeration phase by traversing the mapping DAG *in reverse*, i.e., we explore it backward from the final vertex to the initial vertices. Following this reverse order, we then enumerate the mappings seen along the traversed paths as we previously described in the article. One advantage of doing enumeration backward is that it allows us to skip the trimming step (second stage of the preprocessing phase): if some vertices of the mapping DAG are not co-accessible, the enumeration phase will never reach them and the delay bounds are not affected. However, as we will later show, in practice the time spent on trimming (second preprocessing stage) is often recouped during the third preprocessing stage (because it runs faster when the mapping DAG is smaller).

A more distant benefit of processing the DAG backward is to later extend our implementation to support *updates*, i.e., modifications to the underlying document. A common case of updates is appending characters at the end, which we believe would be easier to handle when enumeration starts at the end. Nevertheless, the question of extending the algorithm and implementation to handle updates is left for future work (see also the discussion in Section 8).

7.2 Experiments

Experimental setup and delay measurement. The tests were run in a virtual machine that had exclusive access to two Xeon E5-2630 CPU cores. The algorithm is single threaded, but the additional core was added to minimize the effects of background activity of the operating system.

Measuring the delays of the algorithm is challenging, because the timescale for the delays is so tiny that unavoidable hardware interrupts can make a big difference. To eliminate outliers resulting from such interrupts, we exploited the fact that our enumeration algorithm is fully deterministic. We ran the algorithm 10 times and recorded all delays. Afterward, for each produced result, we took the median of the 10 delays we collected. All measurements related to delays use this approach, e.g., if we compute the maximum delay for a query, it is actually the maximum over these medians.

We benchmarked our implementation on two datasets: one based on genetic data and another one based on blog posts using the corpus from Schler et al. [28] and comparing against that of Morciano [23]. We first describe the experiments on DNA data and then the experiments on blog posts.

DNA data. For our experiments on DNA data, the input document is the first chromosome of the human genome reference sequence GRCh38. It contains roughly 250 million base pairs,[5] where each base pair is encoded as a single character. We also use prefixes of this data in the experiments, when we need to benchmark against input documents of various sizes.

In most queries, there are no named capture variables, but there is an implicit capture variable that captures each possible match of the regex as a subword of the input document. Formally, when we write a query in the sequel as a regular expression e without capture variables, the corresponding spanner is the one described by the regex-formula $\Sigma^* x\{e\}\Sigma^*$, where Σ is the alphabet and x is the implicit capture variable.
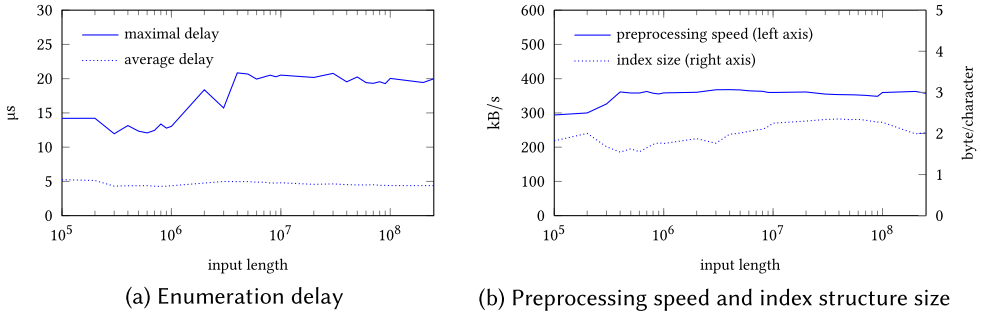
---

[5]https://www.ncbi.nlm.nih.gov/genome/guide/human/.

(a) Enumeration delay

(b) Preprocessing speed and index structure size

Fig. 2. Enumerating the query `TTAC.{0,1000}CACC` on inputs of different lengths.

(a) Preprocessing time

(b) Enumeration delay

(c) Size of index structure

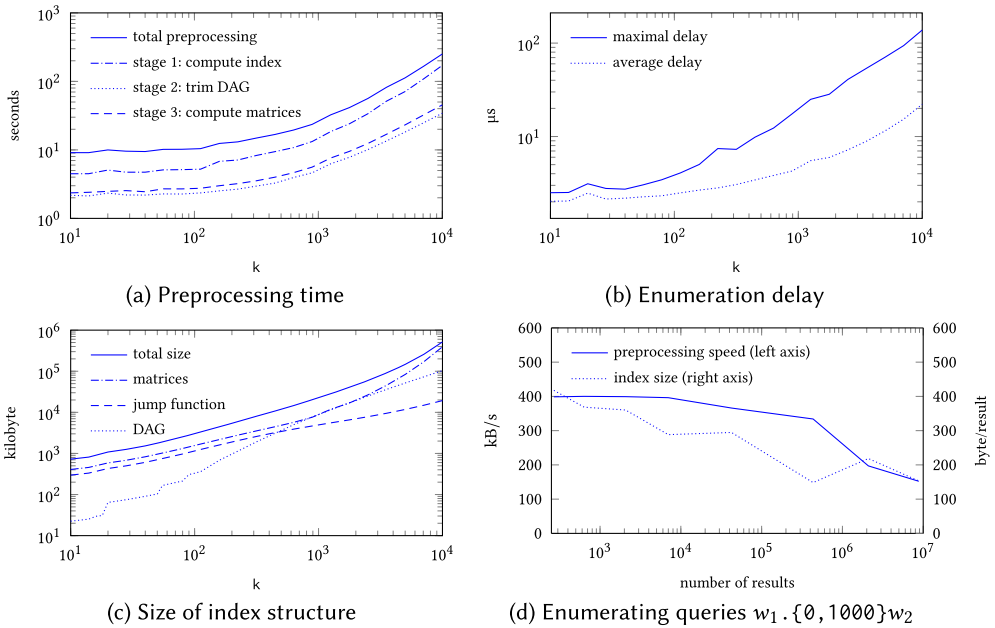(d) Enumerating queries $w_1.\{0,1000\}w_2$

Fig. 3. Enumerating the query `TTAC.{0,k}CACC` on an input document of 10 MB.

Close-fragments queries. Our experiments on DNA data use so-called *close-fragment queries*, where we search for two DNA fragments w_1 and w_2 that occur close to each other. Specifically, we used the query `TTAC.{0,k}CACC`, with various values of k, for several different tests that we list in the following and then present in more detail:

(1) We first verified that the delay is independent from the document length, whereas the preprocessing time and memory usage depends linearly on the document length. This is presented in Figure 2.

(2) We then tested how the preprocessing time, the index structure size, and the delay depends on the automaton size. This is depicted in Figure 3, where we used a 10-MB prefix of the DNA string and used values of k between 10 and 10,000.

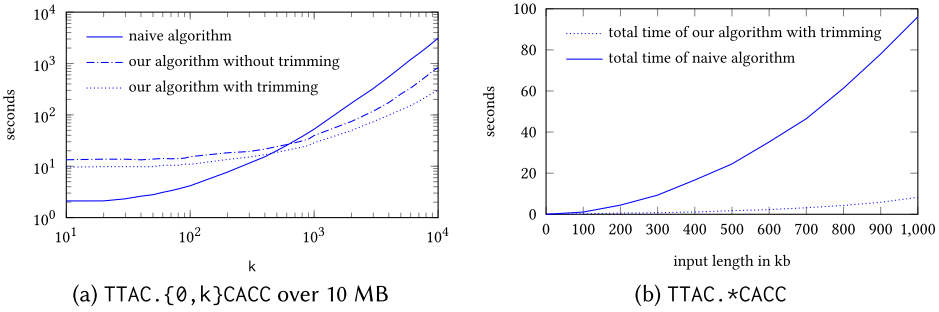(a) TTAC.{0,k}CACC over 10 MB   (b) TTAC.*CACC

Fig. 4. Comparing the total enumeration time with a simple algorithm.

(3) Last, we compared the total enumeration time with the naive approach that starts one run of the NFA at every position of the document.[6] We also investigated the effect of skipping the second stage of the preprocessing. The results are depicted in Figure 4.

For (1), we fixed $k = 1,000$ and used prefixes of different length of the DNA string. The results are depicted in Figure 2, where in Figure 2(a), we depicted the maximal and average delay encountered during enumeration, whereas in Figure 2(b), we depicted the preprocessing time and size of the index structure divided by the input length. One can see that the average delay is constant (around 5 μs allowing to enumerate 200,000 results per second), whereas the maximum delay is roughly four times larger. The preprocessing speed is roughly 300 to 350 kB per second and the index structure twice as large as the input document.

Toward (2), we fixed the length of the input to 10 MB and made k vary between 10 and 10,000. The results are shown in Figure 3(a) and (b). The most interesting outcome is that the preprocessing is much faster than the worst case bound of $O(k^4)$. Analyzing the numbers from Figure 3(a) shows that the preprocessing time grows roughly like $\Theta(k^2)$. A closer look into the index structure used in the algorithm suggests an explanation: the width of the mapping DAG seems to grow sublinearly as a function of k for this query.

As for the delay in Figure 3(b), remember that our experiment is about changing the query, so the delay bound is not supposed to be constant with respect k. The theoretical bounds suggest that the delay should be $O(k^2)$, which matches what we obtain experimentally for the maximum delay. The average delay is much lower.

We also measured the size of our index structure for the queries TTAC.{0,k}CACC after completing the preprocessing and depicted the results in Figure 3(c). The index structure consists of three parts, with the total size being the sum of these three parts:

- *DAG*: The bitmap storing the states that exist in the (trimmed) DAG. We remove the levels to which the algorithm will never jump.
- *Jump function*: The jump function, as explained in Section 5.
- *Matrices*: All necessary reachability matrices, as explained in the same section.

For small automata, the size is dominated by the administrative overhead of the vectors used to store the jump functions and matrices, whereas the DAG is represented in a very compact way as a bitmap. For larger automata, one can see that the DAG representation uses more space, but the

---

[6]Note that this naive approach only works for the special case where there is exactly one capture variable that surrounds the whole expression. Our implementation has the added advantage of handling regular spanners with arbitrarily many capture variables.

memory footprint is still dominated by the matrices. Notice that the size of each level of the DAG is padded to a multiple of 32, and hence the bumps of the DAG curve around the sizes 32 and 64.

A question related to the close-fragment queries TTAC.{0,k}CACC is to understand if the change in performance across different values of k is only caused by the change in the number of results. To experiment with this, we fixed $k = 1,000$ and benchmarked queries w_1.{0,1 000}w_2, where w_1 was a prefix of TTACGG and w_2 was a prefix of CACCTG, so as to make the number of results vary without changing the size of the automaton too much. The results are depicted in Figure 3(d). The resulting index structure size for these queries indeed depends a lot on the number of results. This is expected as the index structure only contains information for levels that are used as the boundary of at least one span in the results. Specifically, the size grows slightly sublinearly. The preprocessing speed (and thus the preprocessing time) is almost constant until the number of results becomes sufficiently large to be comparable to the input size. This is because, before that point, the dominating term in the preprocessing time is the processing of the input and not the computations performed on the DAG.

For (3), we implemented a naive enumeration algorithm that works without any preprocessing, to serve as a baseline. It evaluates the NFA starting from each position i in the input document and outputs a pair (i, j) for each position j where the NFA reaches an accepting state using the standard algorithm that computes for each position the set of possible states. We do the easy optimization of stopping the run for a starting position i if we reach an ending position j with no more reachable states. We depicted the total time used for enumeration of our approach and the naive algorithm in Figure 4, where we ran the query TTAC.{0,k}CACC for various sizes of k on the 10-MB prefix of the DNA sequence (Figure 4(a)) and additionally the query TTAC.*CACC for various prefixes of the input DNA sequence (Figure 4(b)). For small k in Figure 4(a), the naive algorithm has a clear advantage, as it does not need to compute any index structure. In addition, for these queries, the runtime is bounded by $O(nk)$, as all runs of the NFA have a length bounded by at most $k + 8$ because we optimized the baseline algorithms to stop the run early. For larger k, the naive algorithm is much slower than our approach. For the query TTAC.*CACC in Figure 4(b), the naive approach exhibits its $\Theta(n^2)$ worst-case behavior, and is much slower than our approach, even for small input documents.

In Figure 4(a), we also have a look on the effect of trimming the DAG (second stage of the preprocessing). Indeed, although skipping this trimming stage saves a small amount of time, this is usually overcompensated by the third preprocessing stage, where we need to compute more and larger matrices because the unpruned DAG is larger. This can be seen for the query TTAC.{0,k}CACC even for small values for k. Trimming saves more time for larger values of k, as more nodes of the DAG can be pruned. For the query TTAC.*CACC in Figure 4(b), where trimming can only remove a few nodes from the DAG, the runtime effect of disabling trimming was negligible, i.e., the time savings from the second stage were almost exactly compensated by the additional work in the third stage.

Querying blog posts. We also evaluated our algorithm on roughly 800 MB of blog posts using the corpus from Schler et al. [28]. To apply our implementation, we concatenated all blog posts to get a single file and stripped all characters that did not have a valid UTF-8 encoding. We ran the same queries used in the master thesis of Morciano [23, Chapter 6]. These queries try to extract reviews for movies from blog posts. They are built over simple dictionaries that contain, e.g., synonyms for "movie," synonyms for "actor," or a list of genres. These basic spanners are combined to more complex queries using the union operator and joins of the following form: "spanner B matches at most k characters after spanner A matches." For instance, the queries Q_1 to Q_4 are of the following form: find a word in the dictionary d_1, and then a word in the dictionary d_2 matching at most k characters after the first word.

Table 1. Querying Blog Data

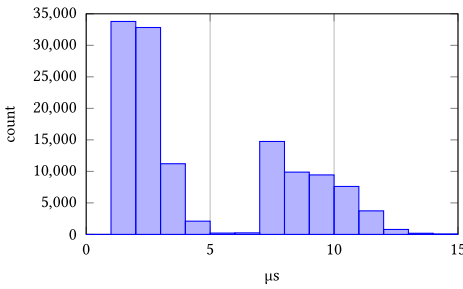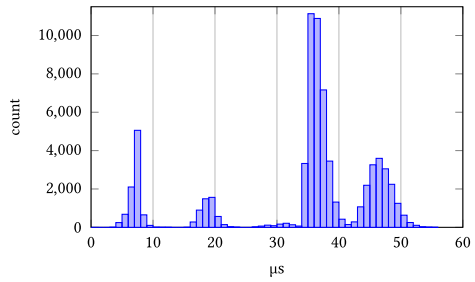| Query	#States	#Variables	#Results	Preprocess (s)	Memory (MB)	Time of [23] (s)
Q_1	40	2	4,975	772	2.72	≈780
Q_2	211	2	6,099	1,057	3.70	≈1,100
Q_3	246	2	5,915	1,090	3.63	≈ 1,200
Q_4	52	2	2,232	771	1.22	≈810
Q_5	343	6	12,020	1,254	8.04	≈2,780
Q'_6	661	8	19,561	1,704	16.00	≈4,330
Q'_7	805	10	62,103	1,948	53.36	≈5,100
Q'_8	813	10	70,509	1,956	60.02	≈6,000

(a) Query TTAC.{0,1000}CACC

(b) Query Q'_8 from Marciano [23]

Fig. 5. Histogram for delays between two outputs.

In Table 1, we give some statistical data over these queries, and give the running time of our algorithm, its memory usage, and the approximate times of the implementation of Morciano [23]. We only report the time for the preprocessing phase of our algorithm, because the time taken by the enumeration phase is always less than 1 second. We stress that the running times of Morciano [23] and our running times are not comparable, because the experimental setup is very different, the hardware in use is not the same, and the algorithm of Marciano [23] is not an enumeration algorithm but simply produces all results. The point of our comparison is not to claim an improvement in running times relative to Marciano [23], but to show that, on this existing dataset, the total running time of our approach is comparable to that of their implementation.

Looking into our running times, we notice that the dependency of the preprocessing time on the automaton size is again much less than the $O(|\mathcal{A}|^4)$ worst-case bound. Again, this is probably because the matches are sparse, i.e., there are only very few nodes per level and therefore the matrices are of almost constant size. Similarly to our experiments on DNA data, the preprocessing time and index structure size show a dependency on the number of matches, as we need to compute matrices for all levels where a variable is opened or closed for some match. Of course, as our preprocessing is linear in the input document, this dependency can only hold when the number of results is at most linear in the document.

Detailed analysis of delay. We did a more detailed analysis of the various delays that we obtain while running the enumeration phase of our algorithm. We show a histogram of the delays for the query TTAC.{0,1000}CACC on the first 10 MB of the DNA data in Figure 5(a), and for the query Q'_8 from Marciano [23] on the blog post corpus in Figure 5(b).

One can see that the delay varies, which is expected: our algorithm is constant delay in the sense of enforcing a constant upper bound on the delay, but the effective delay can vary from one output to the next. Specifically, the number of jumps that need to be performed between two outputs can be any number between one and the maximal number of variable markers encountered in a single match. In addition, the time needed for the flashlight search can vary within given limits.

In Figure 5(a), as the DNA query has only one implicit capture variable covering the whole match and thus two variable markers, we have two spikes in the histogram. The first spike corresponds to the case where the next matching is found by just changing the end marker, whereas the second spike corresponds to the case where the two markers are changed, so that the flashlight search and jump functions have to be executed twice. In Figure 5(b), as the query Q_8' has 10 variables, we notice that the maximal delay is larger and there are more spikes in the histogram.

8 CONCLUSION

We have shown that we can efficiently enumerate the mappings of sequential VAs on input documents, achieving linear-time preprocessing and constant delay in data complexity, while ensuring that preprocessing and delay are polynomial in the input VA even if it is not deterministic. This result was previously considered as unlikely by Florenzano et al. [12], and it improves on the algorithms in the work of Freydenberger et al. [16]: with our algorithm, the delay between outputs does not depend on the input document, whereas it had a linear dependency on the size of the input document in the work of Freydenberger et al. [16].

In Section 7, we did a thorough practical evaluation of our approach. The most encouraging result is that for several classes of queries, the algorithm runs much faster than the theoretical worst-case analysis would suggest. An interesting open question raised by the experimental validation is whether it is possible to adapt our algorithm to NFAs with counters. We believe that queries that use a join condition of the form pattern A should be matched near pattern B are important in practice. These kind of queries intrinsically depend on the use of counters. As the efficiency of our algorithm crucially depends on the size of the underlying automata, a more efficient representation of counters that does not depend on encoding the counter value in the state of the automaton could allow for big improvements in the runtime.

We will consider different directions for future work. A first question is how to cope with changes to the input document without recomputing our enumeration index structure from scratch. This question has been recently studied for other enumeration algorithms, see, e.g., [3, 7–9, 20, 24, 25], but for atomic update operations: insertion, deletion, and relabelings of single nodes. However, as spanners operate on text, we would like to use bulk update operations that modify large parts of the text at once: cut and paste operations, splitting or joining strings, or appending at the end of a file and removing from the beginning, e.g., in the case of log files with rotation. It may be possible to show better bounds for these operations than the ones obtained by modifying each individual letter [20, 25], and we believe that our implementation could be modified to do so, at least when appending new content at the end of the document.

Another question is to generalize our result from words to trees, but this is challenging: the run of a tree automaton is no longer linear in just one direction, so it is not easy to skip parts of the input similarly to the jump function of Section 5, or to combine computation that occurs in different branches. We already explored this direction of work in our follow-up work [5].

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[2] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. 2017. A circuit-based approach to efficient enumeration. In *Proceedings of ICALP 2017*. https://arxiv.org/abs/1702.05589.

[3] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. 2018. Enumeration on trees under relabelings. In *Proceedings of ICDT 2018*. https://arxiv.org/abs/1709.06185.

[4] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Constant-delay enumeration for nondeterministic document spanners. In *Proceedings of ICDT 2019*. https://drops.dagstuhl.de/opus/frontdoor.php?source_opus=10324.

[5] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Enumeration on trees with tractable combined complexity and efficient updates. In *Proceedings of PODS 2019*. https://arxiv.org/abs/1812.09519.

[6] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of CSL 2006*.

[7] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering conjunctive queries under updates. In *Proceedings of PODS 2017*. https://arxiv.org/abs/1702.06370.

[8] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering FO+MOD queries under updates on bounded degree databases. In *Proceedings of ICDT 2017*. https://arxiv.org/abs/1702.08764.

[9] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2018. Answering UCQs under updates and in the presence of integrity constraints. In *Proceedings of ICDT 2017*. https://arxiv.org/abs/1709.10039.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd ed.)*. MIT Press, Cambridge, MA.

[11] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document spanners: A formal approach to information extraction. *Journal of the ACM* 62, 2 (2015), Article 12. https://pdfs.semanticscholar.org/8df0/ad1c6aa0df93e58071b8afe3371a16a3182f.pdf.

[12] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2018. Constant delay algorithms for regular document spanners. In *Proceedings of PODS 2018*. https://arxiv.org/abs/1803.05277.

[13] Dominik D. Freydenberger. 2017. A logic for document spanners. In *Proceedings of ICDT 2017*. http://drops.dagstuhl.de/opus/volltexte/2017/7049/.

[14] Dominik D. Freydenberger. 2019. A logic for document spanners. *Theory of Computing Systems* 63, 7 (2019), 1679–1754. https://link.springer.com/article/10.1007%2Fs00224-018-9874-1.

[15] Dominik D. Freydenberger and Mario Holldack. 2018. Document spanners: From expressive power to decision problems. *Theory of Computing Systems* 62, 4 (2018), 854–898. https://doi.org/10.1007/s00224-017-9770-0.

[16] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining extractions of regular expressions. In *Proceedings of PODS 2018*. https://arxiv.org/abs/1703.10350.

[17] François Le Gall. 2012. Improved output-sensitive quantum algorithms for Boolean matrix multiplication. In *Proceedings of SODA 2012*. https://pdfs.semanticscholar.org/91a5/dd90ed43a6e8f55f8ec18ceead7dd0a6e988.pdf.

[18] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of ISSAC 2014*. https://arxiv.org/abs/1401.7714.

[19] Étienne Grandjean. 1996. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence* 16, 1 (1996), 183–236.

[20] Katja Losemann and Wim Martens. 2014. MSO queries on trees: Enumerating answers under updates. In *Proceedings of CSL-LICS 2014*. http://www.theoinf.uni-bayreuth.de/download/lics14-preprint.pdf.

[21] Arnaud Mary and Yann Strozecki. 2016. Efficient enumeration of solutions produced by closure operations. In *Proceedings of STACS 2016*. http://drops.dagstuhl.de/opus/volltexte/2016/5753/.

[22] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of PODS 2018*. https://arxiv.org/abs/1707.00827.

[23] Andrea Morciano. 2017. *Engineering a Runtime System for AQL*. Master's Thesis. Politecnico di Milano. https://www.politesi.polimi.it/bitstream/10589/135034/1/2017_07_Morciano.pdf.

[24] Matthias Niewerth. 2018. MSO queries on trees: Enumerating answers under updates using forest algebras. In *Proceedings of LICS 2018*.

[25] Matthias Niewerth and Luc Segoufin. 2018. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *Proceedings of PODS 2018*. http://www.di.ens.fr/~segoufin/Papers/Mypapers/enum-update-words.pdf.

[26] Ronald C. Read and Robert E. Tarjan. 1975. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks* 5, 3 (1975), 237–252.

[27] IBM Research. 2018. SystemT. Retrieved February 7, 2021 from https://researcher.watson.ibm.com/researcher/view_
      group.php?id=1264.
[28] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James W. Pennebaker. 2006. Effects of age and gender on
      blogging. In *AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs*, Vol. 6. 199–205. http://u.cs.
      biu.ac.il/~koppel/BlogCorpus.htm.
[29] Luc Segoufin. 2014. A glimpse on constant delay enumeration (Invited talk). In *Proceedings of STACS 2014*. https://
      hal.inria.fr/hal-01070893/document.
[30] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and I. Shirakawa. 1977. A new algorithm for generating all the maximal
      independent sets. *SIAM Journal on Computing* 6, 3 (1997), 505–517. DOI : https://doi.org/10.1137/0206036.
[31] L. G. Valiant. 1979. The complexity of computing the permanent. *Theoretical Computer Science* 8, 2 (1979), 189–201.
      https://www.sciencedirect.com/science/article/pii/0304397579900446.
[32] Kunihiro Wasa. 2016. Enumeration of enumeration algorithms. arXiv:1605.05012. https://arxiv.org/abs/1605.05102.