Constant-Delay Enumeration for Nondeterministic Document Spanners

Antoine Amarilli
LTCI Paris, France
Télécom ParisTech, France
Université Paris-Saclay, France

Pierre Bourhis
CNRS Lille, CRIStAL UMR 9189, France
Inria Lille, France

Stefan Mengel
CNRS, CRIL UMR 8188, Lens, France

Matthias Niewerth
University of Bayreuth, Germany

Abstract

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 which 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, in particular for the restricted case of so-called extended VAs.

2012 ACM Subject Classification Information systems → Information extraction; Theory of computation → Formal languages and automata theory; Theory of computation → Database query processing and optimization (theory)

Keywords and phrases enumeration, spanners, automata

Digital Object Identifier 10.4230/LIPIcs.ICDT.2019.22

Related Version A full version of the paper is available at `https://arxiv.org/abs/1807.09320`.

Funding *Pierre Bourhis*: Supported by the DeLTA ANR project (ANR-16-CE40-0007).
Matthias Niewerth: Supported by grant number MA 4938/4–1 from the Deutsche Forschungsgemeinschaft.

22nd International Conference on Database Theory (ICDT 2019).
Editors: Pablo Barcelo and Marco Calautti; Article No. 22; pp. 22:1–22:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [26] and whose core semantics have then been formalized in [10]. 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*, for short *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 in [10] has led to a thorough investigation of its properties by the theoretical database community [13, 15, 21, 14, 11].

We here 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 [15], this task actually solves the whole data extraction problem for so-called *regular spanners* [10]. While the extraction task is intractable for general VAs [13], it is known to be tractable if we impose that the VA is *sequential* [15, 11], which requires that all accepting runs actually 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 [21, 11, 15] 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, [15, Theorem 3.3] has 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 [27, 30], in particular *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* which precomputes an index data structure in linear data complexity, and an *enumeration phase* which 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 in [11] 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 [11] are exponential in the VA because they first determinize it [11, Propositions 4.1 and 4.3]. This is problematic because the VAs constructed from regex formulas [10] 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 [11] 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 [6, 2] or their delay is not constant in the input document [19]. 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 non-deterministic) automaton. The result that we will show in the present paper will imply 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 non-deterministic sequential VA, or as a sequential regex-formula with capture variables: note that an extension to trees is investigated in our follow-up work [4].

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 [11], and improves on the bounds of [15]. More precisely, we show:

▶ **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, then 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 in [11], 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 [11]. 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 [11, Theorem 5.2]: while 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: we can enumerate maximal cliques with polynomial delay [28], but counting them is #P-hard [29].

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 [11, Proposition 4.2]. We avoid this by adapting our algorithm to work with non-extended 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

[1] Note that, contrary to what the terminology suggests, VAs are not special cases of extended VAs. Further, while 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 [11].

in the VA, relying on the fact that the VA is sequential 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* [20, 25]: 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 given in a form of regular expressions called *regex-formulas*, see [10] for exact definitions. As we can translate sequential regex-formulas to sequential VAs in linear time [10, 15, 21], 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 [10, 15] for the full definitions. As such 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.*

Paper structure. In Section 2, we formally define spanners, VAs, and the enumeration problem that we want to solve on them. In Sections 3–5, 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* 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 conclude in 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 $0 \leq i \leq j \leq |d|$ which represents a substring (contiguous subsequence) 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 [11, 21] but in contrast to [10], 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 *variable-set automata* (or *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 $1 \leq j \leq m$:

- Either 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 $\mathcal{A}$;
- Or 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, and whenever 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 $i \leq i'$. From each valid run, we define a mapping where 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'; if these markers are not read then x is not assigned by the mapping (i.e., it is not in the domain $\mathcal{V}'$). 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. The task studied in this paper 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.

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 [21]. 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 [21, 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 [13, Proposition 12] and [12, Proposition 3.9]. 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 re-wiring 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, [12, 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 [11], 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 [11] 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 input extended VA by rewriting each state to one letter-state and one ev-state, and re-wiring the transitions and changing the initial/final status of states appropriately. This rewriting preserves sequentiality and guarantees that any path in the 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) which contain one at-sign "@" preceded and followed by a non-empty 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 $\Sigma' := \Sigma \setminus \{@, \sqcup\}$.

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 paper do not rely on this property.

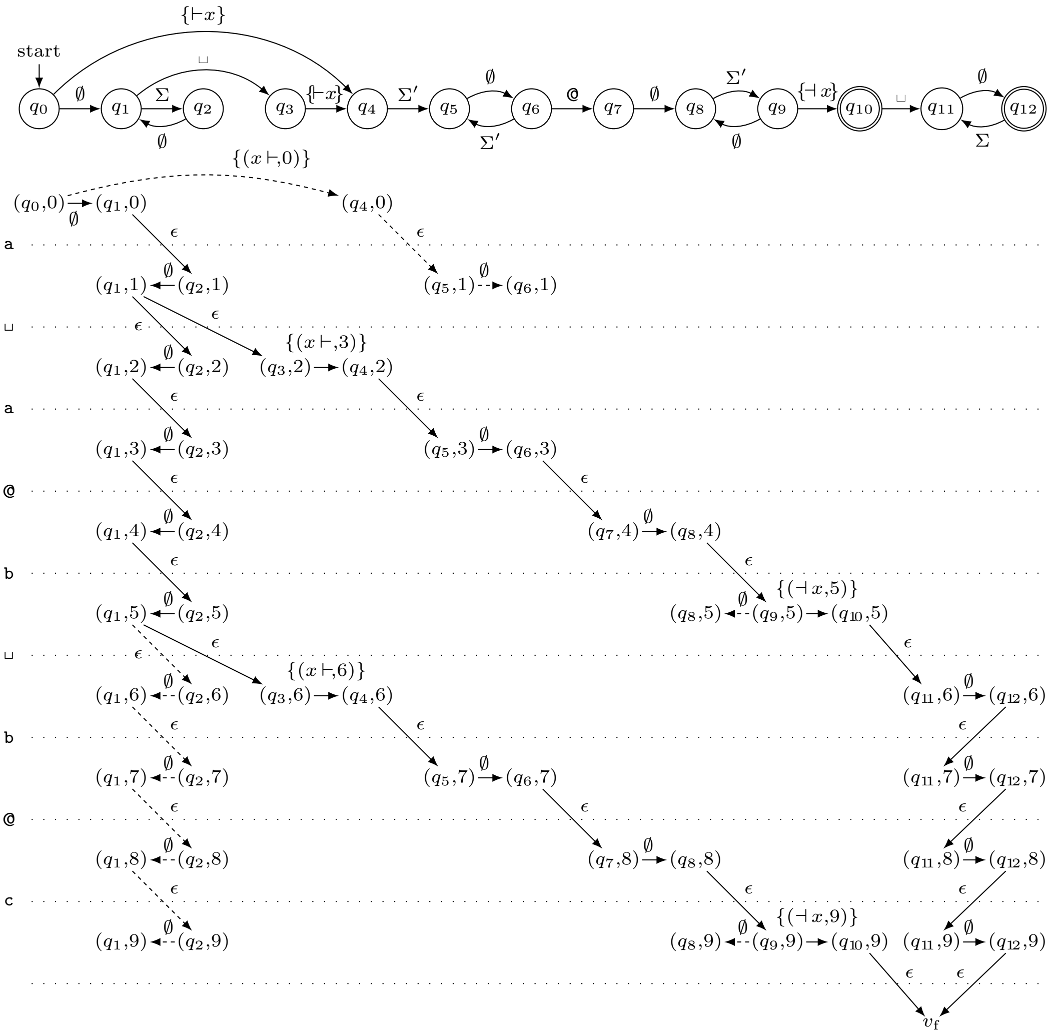

Figure 1 Example sequential extended VA $\mathcal{A}_0$ to extract e-mail addresses (see Example 2.2) and example mapping DAG on an example document (see Examples 3.3, 3.6, 3.7, and 3.10).

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. [16] 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 [17].

3 Computing Mapping DAGs for Extended VAs

We start our paper 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 3–5. We will then cover the case of non-extended VAs in Section 6.

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 directed acyclic graph (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 $v_\mathrm{f} \in V$*, 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 from the initial vertex to the final vertex starts with a marker edge, ends with an ϵ-edge, and alternates between marker edges and ϵ-edges.*

The mapping $\mu(\pi)$ *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. Given a set U of vertices of G, we write $\mathcal{M}(U)$ for the set of mappings of paths from a vertex of U to the final vertex; note that the same mapping may be captured by multiple different paths. The set of mappings* captured *by G is then $\mathcal{M}(G) := \mathcal{M}(\{v_0\})$.*

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 mapping DAG from $\mathcal{A}$ and from a document d, which we call the *product DAG* of $\mathcal{A}$ and d:

▶ **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 normalized mapping DAG whose vertex set is $Q \times \{0, \ldots, n\} \cup \{v_\mathrm{f}\}$ with $v_\mathrm{f} := (\bullet, n+1)$ for some fresh value $\bullet$. Its edges are:*

- *For every letter-transition (q, a, q') in δ, for every $0 \leq i < |d|$ 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 $0 \leq i \leq |d|$, 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 [11], we do not contract the ϵ-edges but keep them throughout our algorithm.

▶ **Example 3.3.** The mapping DAG for our example sequential extended VA $\mathcal{A}_0$ on 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 mapping 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 mapping 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 definition:

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

Proof sketch. The mapping DAG is acyclic and normalized because its edges follow the transitions of the extended VA, which we had preprocessed to distinguish letter-states and ev-states. Paths in the mapping DAG cannot contain multiple occurrences of the same label, because the labels in the mapping DAG include the position in the document. ◁

Further, the product DAG clearly captures what we want to enumerate. Formally:

▷ 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.

▶ **Example 3.6.** The set of mappings captured by the example product DAG on Figure 1 is $\{\{(x\vdash, 3), (\dashv x, 5)\}, \{(x\vdash, 6), (\dashv x, 9)\}\}$, and this is indeed the set of mappings of the example extended VA $\mathcal{A}_0$ on the example document.

Our task is to enumerate $\mathcal{M}(G)$ *without duplicates*, and this is still non-obvious: 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 non-empty 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 non-empty.

▶ **Example 3.7.** For the mapping DAG of Figure 1, trimming eliminates the non-accessible vertices (which are not depicted) and the non-co-accessible vertices (i.e., those with incoming dashed edges).

Second, we 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* $\mathsf{level}(v)$*, 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 $\mathsf{level}(v) = \mathsf{level}(u) + 1$;*
- *For every marker edge from u to v, we have $\mathsf{level}(v) = \mathsf{level}(u)$. Furthermore, all pairs (m, i) in the label of the edge have $i = \mathsf{level}(v)$.*

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 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 $D = |d| + 2$.

▶ **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 non-empty set of vertices in a leveled normalized mapping DAG that all have the same level (written* $\mathsf{level}(\Lambda)$*) and which are all the source of some marker edge. The singleton* $\{v_{\mathrm{f}}\}$ *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 which 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 $\mathsf{level}(\Lambda') = \mathsf{level}(\Lambda) + 1$.

4 Enumeration for Mapping DAGs

In the previous section, we have 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 [18, 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 $\mathsf{mapping}$ being the empty set.

Algorithm 1 Main enumeration algorithm.

```
procedure ENUM($G, \Lambda, \mathsf{mapping}$)
    $\Lambda' := $ JUMP($\Lambda$)
    if $\Lambda'$ is the singleton $\{v_{\mathrm{f}}\}$ of the final vertex then
        OUTPUT($\mathsf{mapping}$)
    else
        for ($\mathsf{locmark}, \Lambda''$) in NEXTLEVEL($\Lambda'$) do
            ENUM($G, \Lambda'', \mathsf{locmark} \cup \mathsf{mapping}$)
```

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 $(\mathsf{locmark}, \Lambda'')$ where:

- The label set $\mathsf{locmark}$ is an edge label such that there is a marker edge labeled with $\mathsf{locmark}$ that starts at some vertex of Λ'
- The level set Λ'' is formed of all the vertices w at level $\mathsf{level}(\Lambda') + 1$ that can be reached from such an edge followed by an ϵ-edge. Formally, a vertex w is in Λ'' if and only if there is an edge labeled $\mathsf{locmark}$ 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 $\mathsf{level}(\Lambda') + 1$.

It is easy to see that the NextLevel function can be computed efficiently:

▶ **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 the pairs $(\mathsf{locmark}, \Lambda'') \in \textsc{NextLevel}(\Lambda')$ with delay $O(W^2 \times |\mathsf{locmark}|)$ in an order such that $\mathsf{locmark} = \emptyset$ comes last if it is returned.*

Proof. 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 $O(|\mathsf{locmark}|)$, so the process is in $O(W^2 \times |\mathsf{locmark}|)$ 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, $\mathsf{locmark} = \emptyset$ 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:

▷ Claim 4.3. For any level set Λ of G which is not the final vertex, we have:

$$\mathcal{M}(\Lambda) \quad = \bigcup_{(\mathsf{locmark},\Lambda'') \in \textsc{NextLevel}(\Lambda)} \mathsf{locmark} \cup \mathcal{M}(\Lambda'') \; . \tag{1}$$

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

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 $\mathsf{locmark} = \emptyset$, 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 $\mathsf{JL}(\Lambda)$ *of Λ is the first level $j \geq \mathsf{level}(\Lambda)$ 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 which is $\neq \epsilon$ and $\neq \emptyset$. In particular we have $\mathsf{JL}(\Lambda) = \mathsf{level}(\Lambda)$ 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 $\textsc{Jump}(\Lambda) := \Lambda$ if $\mathsf{JL}(\Lambda) = \mathsf{level}(\Lambda)$, and otherwise $\textsc{Jump}(\Lambda)$ is formed of all vertices at level $\mathsf{JL}(\Lambda)$ to which some $v \in \Lambda$ have a directed path whose last edge is labeled ϵ. This ensures that $\textsc{Jump}(\Lambda)$ is always a level set.*

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

▷ Claim 4.5. For any level set Λ of G, we have $\mathcal{M}(\Lambda) = \mathcal{M}(\textsc{Jump}(\Lambda))$.

Claims 4.3 and 4.5 imply that Algorithm 1 is correct with this implementation of Jump:

▶ **Proposition 4.6.** Enum($\{v_0\}, \epsilon$) *correctly enumerates $\mathcal{M}(G)$ (without duplicates).*

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 4.7.** *Given a leveled mapping DAG G with width W, 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* Jump(Λ) *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 4.7. For the delay, we claim:

▷ Claim 4.8. 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 sketch. The time to call Jump is in $O(W^2)$ by Proposition 4.7, and the time spent to move to the next iteration of the **for** loop with a label set locmark is in time $O(W^2 \times |\text{locmark}|)$ using Proposition 4.2: now the operations in the loop body run in constant time if we represent mapping as a linked list so that we do not have to copy it when making the recursive call. As Proposition 4.2 ensures that $\emptyset$ comes last, when producing the first solution, we make at most $r+1$ calls to produce a solution of size r, and the time is in $O(W^2 \times (r+1))$. We adapt this argument to show that each successive solution is also produced within that bound: note that when we use $\emptyset$ in the **for** loop (which does not contribute to r) then the next call to Enum either reaches the final vertex or uses a non-empty set which contributes to r. What is more, as $\emptyset$ is considered last, the corresponding call to Enum is tail-recursive, so we can ensure that the size of the stack (and hence the time to unwind it) stays $\leq r+1$. ◁

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 our memory usage during enumeration is in the call stack, and thanks to tail recursion elimination (see the proof of Claim 4.8) 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 4.7. We first explain the preprocessing for the JUMP function, and then the computation scheme.

Preprocessing scheme. Recall the definition of the jump level $\mathsf{JL}(\Lambda)$ and jump set $\textsc{Jump}(\Lambda)$ 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 $\mathsf{level}(v)$, as well as, for each level i, the list of the vertices v such that $\mathsf{level}(v) = i$.

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 which is neither an ϵ-edge nor an $\emptyset$-edge. We claim:

▷ 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 sketch. We do the computation along a reverse topological order: we have $\mathsf{JL}(v_\mathrm{f}) := \mathsf{level}(v_\mathrm{f})$ for the final vertex v_f, we have $\mathsf{JL}(v) := \mathsf{level}(v)$ if v has an outgoing edge which is not an ϵ-edge or an $\emptyset$-edge, and otherwise we have $\mathsf{JL}(v) := \min_{v \to w} \mathsf{JL}(w)$. ◁

The second part of the preprocessing is to compute, for each level i of G, the *reachable levels* $\mathsf{Rlevel}(i) := \{\mathsf{JL}(v) \mid \mathsf{level}(v) = i\}$, 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 $|\mathsf{Rlevel}(i)| \leq W$.

▶ **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 $\mathsf{Rlevel}(3)$ 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 $\mathsf{Reach}(i, j)$ be the Boolean matrix of size at most $W \times W$ which describes, for each (u, v) with $\mathsf{level}(u) = i$ and $\mathsf{level}(v) = j$, whether there is a path from u to v whose last edge is labeled ϵ. We can't afford to compute all 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 $\mathsf{Reach}(i, j)$ for all pairs of levels $i < j$ such that $j \in \mathsf{Rlevel}(i)$.

Proof sketch. We compute them in decreasing order on i: the matrix $\mathsf{Reach}(i, i+1)$ can be computed in time $O(W \times W)$ from the edge relation, and matrices $\mathsf{Reach}(i, j)$ with $j > i + 1$ can be computed in time $O(W^\omega)$ as the product of $\mathsf{Reach}(i, i+1)$ and $\mathsf{Reach}(i+1, j)$: note that $\mathsf{Reach}(i+1, j)$ has been precomputed because $j \in \mathsf{Rlevel}(i)$ easily implies that $j \in \mathsf{Rlevel}(i+1)$. ◁

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 $\mathsf{JL}(\Lambda)$ which we compute as $\min_{v \in \Lambda} \mathsf{JL}(v)$. If $j = i$, then $\textsc{Jump}(\Lambda) = \Lambda$ and there is nothing to do. Otherwise, by definition there must be $v \in \Lambda$ such that $\mathsf{JL}(v) = j$, so v witnesses that $j \in \mathsf{Rlevel}(i)$, and we know that we have precomputed the matrix $\mathsf{Reach}(i, j)$. Now $\textsc{Jump}(\Lambda)$ 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 $\mathsf{Reach}(i, j)$.

This concludes the proof of Proposition 4.7 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 non-extended VAs rather than sequential extended VAs. Remember that we cannot tractably convert non-extended VAs into extended VAs [11, Proposition 4.2], so we must modify our construction in Sections 3–5 to work with sequential non-extended 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 non-extended 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 transitions 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 non-extended 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 $Q \times \{0, \ldots, n\} \cup \{v_\mathrm{f}\}$ with $v_\mathrm{f} := (\bullet, n+1)$ for some fresh value $\bullet$, and with the following edge set:

- For every letter-transition (q, a, q') of $\mathcal{A}$, for every $0 \leq i < |d|$ 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 $0 \leq i \leq |d|$, 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 $|d| + 2$, and the width W (maximal size of a level) is still $\leq |Q|$; we will also define the *complete width* W_c of G in this section as the maximal size, over all levels i, of the sum of the number of vertices with level i and of the number of *edges* with a source vertex having level i: clearly we have $W_\mathrm{c} \leq |\mathcal{A}|$. 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 non-empty set of vertices that are at the same level.

We then reuse the enumeration approach of Section 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 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' which 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 4.6 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:

▶ **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 the pairs $(S^+, \Lambda'') \in$* NextLevel(Λ') *with delay $O(W_{\mathrm{c}} \times B^2)$ in an order such that $S^+ = \emptyset$ comes last if it is returned.*

With this runtime, the delay of Theorem 4.1 becomes $O((r+1) \times (W^2 + W_{\mathrm{c}} \times B^2))$, and we know that $W_{\mathrm{c}} \leq |\mathcal{A}|$, 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* [20, 25]: 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, then 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 which 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 non-empty set $\Lambda' \subseteq V$ of vertices of G and given two disjoint sets of labels S^+ and S^-, we can compute in time $O(|G| \times |S^+|)$ 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 sketch. We first delete all edges from G with a label in S^-, add a fresh source vertex s_0, and remove all vertices that are not reachable from s_0. We then follow a topological sort of G to annotate each vertex v with the maximal set of labels of S^+ that can be seen along paths from s_0 to v: and we use a failure annotation $\emptyset$ when there are two such paths that can see two incomparable sets of labels of S^+. Indeed, as we argue, when this happens the vertex v can never be part of an S^+/S^--path because the definition of G imposes that each edge label occurs at most once on any path, so the partial paths from s_0 to v can never be completed with all missing labels from S^+. Hence, we can compute our set Λ'_2 simply by returning all the vertices annotated by the whole set S^+. ◀

We can now use Lemma 6.4 to prove Theorem 6.2:

Proof sketch of Theorem 6.2. We restrict our attention to the level $\mathsf{level}(\Lambda')$ of the mapping DAG G that contains the input level set Λ': in particular we remove all ϵ-edges. The resulting mapping DAG has size at most W_c, and we call $\mathcal{K}$ the set of labels that it uses, whose cardinality is at most the alphabet size B of G. We fix some arbitrary order on $\mathcal{K}$. Now, let us consider the full decision tree $T_{\mathcal{K}}$ on $\mathcal{K}$ following this order: it is a complete binary tree of height $|\mathcal{K}|$, each internal node at depth $0 \leq r < |\mathcal{K}|$ has two children reflecting on whether we take the r-th label of $\mathcal{K}$ or not, and each leaf n corresponds to a subset of $\mathcal{K}$ built according to the choices described on the path from the root of $T_{\mathcal{K}}$ to n. Our algorithm will explore $T_{\mathcal{K}}$ to find the sets S^+ of labels that we must enumerate for Λ' and G.

More precisely, we wish to determine the leaves of $T_{\mathcal{K}}$ that correspond to a set S^+ such that there is an S^+-path in G from a vertex of Λ' to a vertex with an outgoing ϵ-edge: we call this a *good* leaf. The naive way to find the good leaves would be to test them one after the other, but this would not ensure a good delay bound. Instead, we use the notion of S^+/S^--paths to only explore the relevant parts of $T_{\mathcal{K}}$. Following this idea, we say that an internal node n at depth $0 \leq r < |\mathcal{K}|$ of $T_{\mathcal{K}}$ is *good* if there is an S^+/S^- path from a vertex of Λ' to a vertex with an outgoing ϵ-edge, where S^+ and S^- respectively contain the labels of $\mathcal{K}$ that we decided to take and those that we decided not to take when going from the root of $T_{\mathcal{K}}$ to n. Note that S^+, S^- is a partition of the r first labels of $\mathcal{K}$ that uniquely defines n.

We can now use Lemma 6.4 as an oracle to determine, given any node n of the tree, whether n is good in this sense or not. This oracle makes it possible to find the good leaves of $T_{\mathcal{K}}$ efficiently, by starting at the root of $T_{\mathcal{K}}$ and doing a depth-first exploration of good nodes of the tree. We build $T_{\mathcal{K}}$ on-the-fly while doing so, to avoid materializing irrelevant parts of the tree. The exploration is guaranteed to find all good leaves, because the root of the tree is always good, and because the ancestors of a good leaf are always good. Further, it ensures that we always find one new good leaf after at most $O(|\mathcal{K}|)$ invocations of Lemma 6.4, because whenever we are at a good node then it must have a good child and therefore, by induction, a good descendant that is a leaf. We will find this leaf in our depth-first search with a number of oracle calls that is at most linear in the height of $T_{\mathcal{K}}$. Together with the delay bound of Lemma 6.4, this yields the claimed delay bound of $O(|W_c| \times B^2)$.

Last, it is clear that whenever we have found a good leaf corresponding to a set S^+, then we can compute the new level set Λ'' that we must return together with S^+, with the same delay bound. Indeed, we can simply do this by post-processing the set of vertices returned by the corresponding invocation of Lemma 6.4. ◀

Memory usage. The recursion depth of Algorithm 1 on general sequential VAs is unchanged, and we can still eliminate tail recursion for the case $\mathsf{locmark} = \emptyset$ 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 Conclusion

We have shown that we can efficiently enumerate the mappings of sequential variable-set automata 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 [11], and it improves on the algorithms in [15]: 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 [15].

We will consider different directions for future works. 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, 8, 9, 19, 23, 24], 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 [24, 19].

A second 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 believe that these difficulties can be solved and that a similar result can be shown for trees, but that the resulting algorithm is far more complex: this point, and the question of updates, are explored in our follow-up work [4].

Finally, it would be interesting to implement our algorithms and evaluate them on real-world data similarly to the work in [5, 22]. We believe that our techniques are rather simple and easily implementable, at least in the case of extended VAs. Moreover, since there are no large hidden constants in any of our constructions, we feel that they might be feasible in practice. Nevertheless, an efficient implementation would of course have to optimize implementation details that we could gloss over in our theoretical analysis since they make no difference in theory but might change practical behavior substantially.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

2 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017. arXiv:1702.05589.

3 Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. In *ICDT*, 2018. arXiv:1709.06185.

4 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on Trees with Tractable Combined Complexity and Efficient Updates. Under review, 2019. arXiv:1812.09519.

5 Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. A framework for annotating CSV-like data. *PVLDB*, 9(11), 2016. URL: http://www.vldb.org/pvldb/vol9/p876-arenas.pdf, doi:10.14778/2983200.2983204.

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

7 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries under Updates. In *PODS*, 2017. arXiv:1702.06370.

8 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD Queries Under Updates on Bounded Degree Databases. In *ICDT*, 2017. arXiv:1702.08764.

9 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *ICDT*, 2018. arXiv:1709.10039.

10 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document Spanners: A Formal Approach to Information Extraction. *J. ACM*, 62(2), 2015. URL: https://pdfs.semanticscholar.org/8df0/ad1c6aa0df93e58071b8afe3371a16a3182f.pdf, doi:10.1145/2699442.

11 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant Delay Algorithms for Regular Document Spanners. In *PODS*, 2018. arXiv:1803.05277.

12 Dominik D. Freydenberger. A Logic for Document Spanners. Unpublished extended version. URL: http://ddfy.de/sci/splog.pdf.

13 Dominik D. Freydenberger. A Logic for Document Spanners. In *ICDT*, 2017. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7049/, doi:10.4230/LIPIcs.ICDT.2017.13.

14 Dominik D. Freydenberger and Mario Holldack. Document Spanners: From Expressive Power to Decision Problems. *Theory Comput. Syst.*, 62(4), 2018. doi:10.1007/s00224-017-9770-0.

15 Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining Extractions of Regular Expressions. In *PODS*, 2018. arXiv:1703.10350.

16 François Le Gall. Improved output-sensitive quantum algorithms for Boolean matrix multiplication. In *SODA*, 2012. URL: https://pdfs.semanticscholar.org/91a5/dd90ed43a6e8f55f8ec18ceead7dd0a6e988.pdf.

17 François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, 2014. arXiv:1401.7714.

18 Étienne Grandjean. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence*, 16(1), 1996.

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

20 Arnaud Mary and Yann Strozecki. Efficient Enumeration of Solutions Produced by Closure Operations. In *STACS*, 2016. URL: http://drops.dagstuhl.de/opus/volltexte/2016/5753/.

21 Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *PODS*, 2018. arXiv:1707.00827.

22 Andrea Morciano. Engineering a runtime system for AQL. Master's thesis, Politecnico di Milano, 2017. URL: https://www.politesi.polimi.it/bitstream/10589/135034/1/2017_07_Morciano.pdf.

23 Matthias Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, 2018. doi:10.1145/3209108.3209144.

24 Matthias Niewerth and Luc Segoufin. Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates. In *PODS*, 2018. URL: http://www.di.ens.fr/~segoufin/Papers/Mypapers/enum-update-words.pdf.

25 Ronald C. Read and Robert E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3), 1975.

26 IBM Research. SystemT, 2018. URL: https://researcher.watson.ibm.com/researcher/view_group.php?id=1264.

27 Luc Segoufin. A glimpse on constant delay enumeration (Invited talk). In *STACS*, 2014. URL: https://hal.inria.fr/hal-01070893/document.

28 Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and I Shirakawa. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM J. Comput.*, 6, September 1977. doi:10.1137/0206036.

29 L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2), 1979. URL: https://www.sciencedirect.com/science/article/pii/0304397579900446.

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