Constant-Delay Enumeration for Nondeterministic Document Spanners

Antoine Amarilli
LTCI, Télécom Paris, Institut Polytechnique de Paris
antoine.amarilli@telecom-paris.fr

Pierre Bourhis
CNRS, CRIStAL, UMR 9189 & Inria Lille
pierre.bourhis@univ-lille.fr

Stefan Mengel
CRIL, CNRS & Univ Artois
mengel@cril.fr

Matthias Niewerth
University of Bayreuth
matthias.niewerth@uni-bayreuth.de

ABSTRACT

One of the classical tasks in information extraction is to extract subparts of texts through regular expressions. In the database theory literature, this approach has been generalized and formalized as *document spanners*. In this model, extraction is performed by evaluating a particular kind of automata, called a sequential *variable-set automaton* (VA). The efficiency of this task is then measured in the context of enumeration algorithms: we first run a preprocessing phase computing a compact representation of the answers, and second we produce the results one after the other with a short time between consecutive answers, called the *delay* of the enumeration. 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., a constant delay that does not depend on the document. We present such an algorithm for a variant of VAs called *extended sequential VAs* and give an experimental evaluation of this algorithm.

This article is a shortened version of the conference article [4] published at ICDT'19, incorporating experimental results from the journal version [6] currently under review.

1. INTRODUCTION

Information extraction from text documents is an important task in data management. One of the classical approaches is to use regular expressions (*regexes*) with variables to extract subwords satisfying a pattern. For example, to extract the emails addresses in a text, we could extract substrings that contain an @ character, contain no blank character, but are preceded and followed by a blank character. A more general, declarative way to define this task is the framework of *document spanners*, which was first implemented by IBM in their tool SystemT [16], and whose core semantics have then been formalized in [8]. The spanner approach uses variants of regular expressions (namely, *regex formulas* with variables) to extract substrings, and a relational query over these extraction results to combine them. To perform evaluation, the first step is to evaluate the regular expressions, which is done by compiling them to variants of finite automata, the so-called *variable-set automata*, or *VAs* for short. Second, we compute a plan for the relational query, using relational algebra operators like joins, unions and projections. Last, we evaluate this plan over the results of the extraction. The formalization of the spanner framework in [8] has led to a thorough investigation of its properties by the theoretical database community, see [10, 12, 19, 11, 9, 22].

This paper focuses on the first task 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 in fact be compiled directly into VAs [12], this task actually covers the whole data extraction problem for so-called *regular spanners* [8]. While the extraction task is intractable for general VAs [10], it is known to be tractable if we impose that the VA is *sequential* [12, 9], i.e., if we impose that all accepting runs actually describe a well-formed mapping; we make this assumption throughout our work. Even with this restriction, 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 can be $\Theta(n^{2k})$ mappings on an input document of size n, which is unreasonable if n is large. For this reason, recent works [19, 9, 12] 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 time spent between two consecutive results, called *delay*, is always small. Specifically, [12, Theorem 3.3] has shown how to enumerate

The original version of this paper is entitled "Constant-Delay Enumeration for Nondeterministic Document Spanners" and was published in (22nd International Conference on Database Theory 2019, 2019, Schloss Dagstuhl - Leibniz-Zentrum für Informatik). The authors have been partially supported by the ANR project EQUUS ANR-19-CE48-0019. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758.

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 as long as $|d|$, which is generally very large. By contrast, enumeration algorithms for other database tasks often enforce stronger tractability guarantees in data complexity [23, 26], 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 such that the delay between any two consecutive results is always *constant*, i.e., independent from the input data. It was recently shown in [9] 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 [9] are exponential in the VA because they first determinize it [9, Propositions 4.1 and 4.3]. This is problematic because the VAs constructed from regex formulas [8] are generally nondeterministic and determinization can blow up the size of the automaton exponentially.

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 size of the sequential VA and the document) remains polynomial, while ensuring that the *data complexity* (in the document size only) 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 that satisfies both these requirements while working on nondeterministic sequential VAs. Further, it was conjectured that such an algorithm is unlikely to exist [9] because the related task of *counting* the number of mappings is SpanL-hard and thus intractable 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 [7, 2] or their delay is not constant in the input document [18].

Contributions. We show that nondeterminism is in fact not an obstacle to enumerating the results of document spanners efficiently: 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 size. This answers the open question of [9], and improves on the bounds of [12].

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. Here, we present it for so-called *extended sequential VAs*, a variant of sequential VAs introduced in [9]. Note that, despite the name, extended VAs are actually more restrictive than VAs: they can be converted in PTIME to VAs, but the converse is not true as there are VAs for which the smallest equivalent extended VA has exponential size [9]. This being said, our approach also generalizes from sequential extended VAs to sequential VAs: we do not include this extension in this paper for lack of space, but the result can be found in the original paper [4].

Our overall approach is to construct a kind of product of the input document with the extended VA, similarly to [9]. We then use several tricks to ensure the constant delay bound despite nondeterminism; in particular, we precompute a *jump function* that allows us to quickly skip 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 [9, 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 [24], but counting them is #P-hard [25].

We have also implemented our algorithm and present a short experimental evaluation using this implementation. The implementation can be found at `https://github.com/PoDMR/enum-spanner-rs` and is under the BSD 3-clause license.

Paper structure. In Section 2, we formally define spanners, VAs, and the enumeration problem that we want to solve on them. We then describe our main result in Section 3, and prove it in Sections 4 and 5. Last, we present the experimental performance of our algorithm in Section 6 and conclude in Section 7.

2. PRELIMINARIES

Document spanners. 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 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 [9, 19] but in contrast to [8], we do 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.

Extended VAs. Document spanners are often repre-

sented as *variable-set automata* (or *VAs*). We present our results on a variant of VAs introduced by [9], 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* of the form (q, a, q') for $q, q' \in Q$ and $a \in \Sigma$, and of *extended variable transitions* (or *ev-transitions*) of the form (q, M, q') where M is a possibly empty set of variable markers ($x\vdash$ or $\dashv x$, $x \in \mathcal{V}$). Intuitively, on ev-transitions, the automaton reads multiple markers at once. A *configuration* of an extended VA is a pair (q, i) where $q \in Q$ and i is a position of the input document d Formally, a *run* σ of $\mathcal{A}$ on $d = d_0 \cdots d_{n-1}$ is a sequence of configurations 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.

An extended VAs is called *sequential* if all its accepting runs are *valid* in the following sense: 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 accepting run of an extended sequential VA, we can then 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 $\mathcal{V}'$). Throughout this work, we always assume that extended VAs are *sequential*.

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 accepting 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 sequential extended 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.

In the rest of the paper, we further assume that all extended VAs 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 implicitly assume that all sequential VAs have been trimmed, which implies that they cannot contain any cycle of variable transitions.

Last, we assume that the states of our 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. This requirement can be imposed in linear time on any input extended VA; because we allow transitions labeled with the empty set, unlike the definition of [9].

EXAMPLE 2.1. *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 $\Sigma' := \Sigma \setminus \{@, \textvisiblespace\}$.*

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.

Matrix multiplication. The complexity bottleneck for some of our results is the complexity of multiplying two Boolean matrices, which is a long-standing open problem, see e.g. [13] for a recent discussion. When stating our results, we 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 $O(r^\omega)$. The best known upper bound is currently $\omega < 2.3728639$, see [14].

3. ENUMERATION RESULT

Our main result is the following.

THEOREM 3.1. *Let $2 \leq \omega \leq 3$ be an exponent for Boolean matrix multiplication. Let $\mathcal{A}$ be a extended 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.*

This result is extended to sequential VAs in [4]. Our result 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 [8] for exact definitions. As we can translate sequential regex-formulas to sequential VAs in linear time [8, 12, 19], our results imply that we can also evaluate them.

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 [8, 12] 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.

4. COMPUTING A MAPPING DAG

To show Theorem 3.1, we reduce the problem of enumerating the mappings captured by an extended sequential VA $\mathcal{A}$ 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 $\mathcal{A}$. In the mapping DAG, we no longer care about the labels of letter transitions, so we erase these labels and call these transitions *ϵ-transitions*. As for the ev-transitions, we 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 4.1. *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*. There are two kinds of edges:* ϵ-edge*, whose label x is ϵ, and* marker edges*, whose label x is a finite (possibly empty) set of pairs (m, i), where m is a variable marker and i is a position. 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 correspond to letter transitions of $\mathcal{A}$ (with the letter being erased, i.e., replaced by ϵ), and marker edges 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 4.2. *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 $Q \times \{0, \ldots, n\} \cup \{v_\mathrm{f}\}$. 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 $q \in F$, there is 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 [9], we do not contract the ϵ-edges but keep them throughout our algorithm.

Example 4.3. *The mapping DAG for our example sequential extended VA $\mathcal{A}_0$ on the 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 clear that the notion of product DAG is a mapping DAG and captures the mappings that we want to enumerate.

Example 4.4. *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 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 it 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 is 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 4.5. *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).*

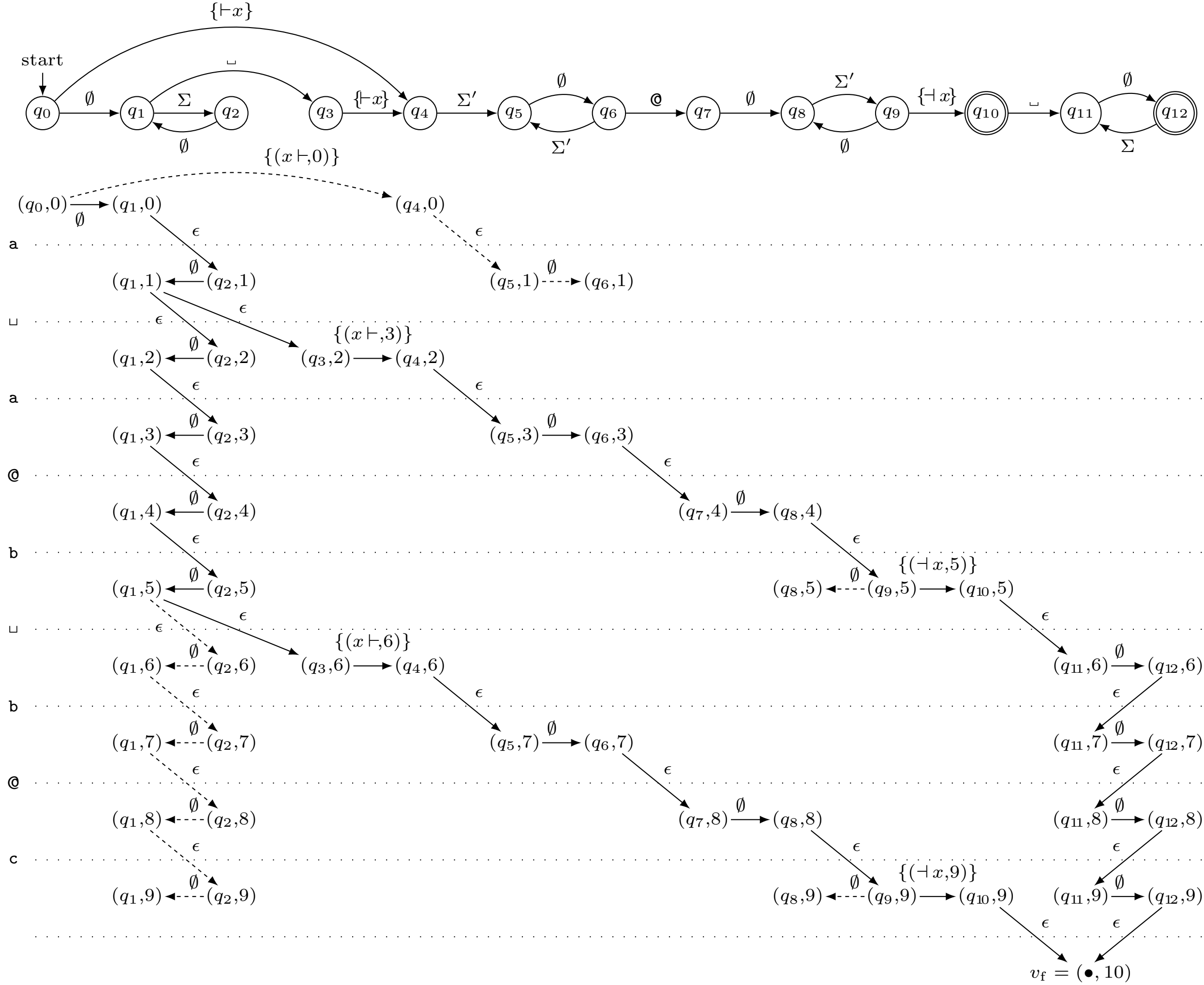

Figure 1: Example sequential extended VA $\mathcal{A}_0$ to extract e-mail addresses (see Example 2.1) and example mapping DAG on an example document (see Examples 4.3, 4.4, 4.5, and 4.7).

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

Definition 4.6. *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, it holds that* $\mathsf{level}(v) = \mathsf{level}(u) + 1$*;*
- *For every marker edge from u to v, it holds that* $\mathsf{level}(v) = \mathsf{level}(u)$*. 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 product DAG of $\mathcal{A}$ and d is leveled, W is less than $|Q|$, and D is equal to $|d| + 1$.

Example 4.7. *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 need the notion of a *level set*:

Definition 4.8. *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 $\{v_f\}$ 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 $\mathsf{level}(\Lambda') = \mathsf{level}(\Lambda) + 1$.

5. ENUMERATION ON 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 5.1. *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 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 [15, Theorem 3.1], so the preprocessing is in $O(|G|)$.

Our general enumeration algorithm is 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(G, Λ, mapping)
2:     Λ' := 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(G, Λ'', locmark ∪ mapping)
```

For simplicity, let us assume for now that the JUMP function is just 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 labeled with locmark that starts at some vertex of Λ'
- The level set Λ'' is formed of all the vertices w at level level(Λ')$+1$ 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 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 level(Λ')$+1$.

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

PROPOSITION 5.2. *Given a leveled trimmed normalized mapping DAG G with width W, and given a level set Λ', we can enumerate without duplicates all the pairs* (locmark, Λ'') $\in$ NEXTLEVEL(Λ') *with delay $O(W^2 \times$ |locmark|$)$ in an order such that* locmark $= \emptyset$ *comes last if it is returned.*

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.

It can easily be proven by induction that Algorithm 1 correctly enumerates $\mathcal{M}(G)$ when JUMP is the identity function. However, the algorithm then does not achieve the desired delay bounds: indeed, it may be the case that NEXTLEVEL(Λ') 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 5.3. *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 which is $\neq \epsilon$ and $\neq \emptyset$. In particular, we have* JL(Λ) $=$ 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(Λ) $:= \Lambda$ *if* JL(Λ) $=$ 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.*

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.

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 here, and give the proof and more details in [4]. Intuitively, the jump function relies on the multiplication of matrices of size $W \times W$, hence the time bound.

PROPOSITION 5.4. *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 5.1 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 5.4. For the delay, we can show that 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.

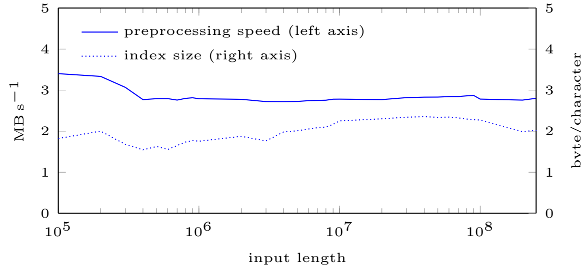

Figure 2: Preprocessing time and index structure size for the query TTAC.{0,1000}CACC on inputs of different lengths.

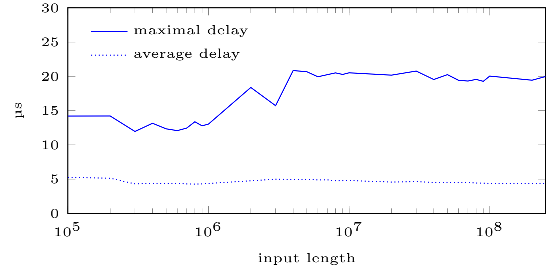

Figure 3: Enumeration delay for the query TTAC.{0,1000}CACC on inputs of different lengths.

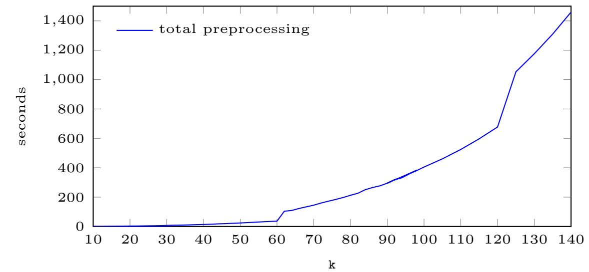

Figure 4: Preprocessing time for the query .{0,k} on an input document of 100 kB, as a function of k.

6. EXPERIMENTS

In this section, we present a very short experimental evaluation of our implementation of the enumeration algorithm. More results can be found in [6]. Our implementation enumerates the mappings assigned to a document by a nondeterministic sequential VA.

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 between outputs of the algorithm is challenging, because the timescale for these 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 twenty times and recorded all delays. Afterwards, for each produced result, we took the median of the twenty delays that we collected. All delay measurements 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 a genetic dataset: the first chromosome of the human genome reference sequence GRCh38, available at https://www.ncbi.nlm.nih.gov/genome/guide/human/. It contains roughly 250 million base pairs, 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.

We consider the query extracting factors defined by the regex TTAC.{0,100}CACC to illustrate the data complexity of our algorithm, and consider the set of queries extracting all substrings up to a given length k (i.e., the regex .{0,k}) to illustrate its combined complexity.

For the first query, we give in Figure 2 the preprocessing time and size of the index structure divided by the input length, and give in Figure 3 the delay. We see that the preprocessing speed is roughly 3 megabytes per second and the index structure is twice as large as the input document. The average delay is constant (around five microseconds, amounting to 200,000 results per second), while the maximum delay is roughly four times larger.

For the queries of the form .{0,k}, we used as input the first 100,000 characters of the genomic data from the previous experiment. This query does not look interesting and indeed, all resulting mappings can be computed trivially given the length of the string. However, this query triggers the worst case behavior of our algorithm, as almost all levels have width $k+1$. We give the preprocessing time in Figure 4. As our implementation uses the naive $O(n^3)$ matrix multiplication algorithm, its running time is supposed to be $\Theta(k^4)$ in this case. This is consistent with what we observe experimentally. The jumps in preprocessing time that can be seen in the figure result from the fact that our implementation pads the matrix widths to a multiple of 64.

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 [9], and it improves on the algorithms in [12]: 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 [12].

Since the publication of our original paper [4], we have extended our results in several ways. First, our algorithm has been implemented and we have evaluated its performance experimentally; we summarized these results in Section 6, with the full results being given in [6]. Secondly, we have studied the problem of efficient enumeration on dynamic documents, i.e., maintaining the index structures that we use for enumeration when the input document is updated. Our results in this

direction are presented in [5], in the more general setting of enumerating queries over trees. Specifically, relative to [4], we study enumeration for nondeterministic tree automata (rather than word automata), and achieve the same theoretical complexity bounds. Moreover, we can update our index structure in logarithmic time in the size of the tree when performing atomic updates on the input tree, i.e., relabeling a node, deleting or adding a leaf. Our results in [5] thus achieve the same data complexity bounds as the previously proposed algorithms for efficient enumeration of such queries on trees, e.g., those of [3, 18, 17, 20, 21], while supporting a more expressive update language, and while additionally ensuring tractability in the nondeterministic tree automaton.

One remaining open problem for efficient enumeration on dynamic data is to have an efficient support for more general updates. Specifically, in the context of words, our update language from [5] only allows single letter changes in the input documents. We do not know how to deal efficiently with more complex update operators, e.g., bulk update operations that modify large parts of the text at once like cutting and pasting parts of the text, splitting or joining strings, etc. We also do not know how to handle the complexity of updates to avoid the logarithmic dependency in the input document: while we show a lower bound in [5] on the update time, it may be possible to achieve constant-time updates for the case of strings for specific updates, e.g., at the beginning or end of the word, as in the case of rotating a log file, or for more restricted queries than the class of regular spanners. Last, an interesting open question is whether our methods allow for efficient support for other operations, e.g., testing if an input mapping is an answer to the query: such testing queries are efficiently supported in [17] (which has no support for updates), and we do not know if we can handle such queries with our methods (and especially in combination with updates).

8. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017.

[3] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *ICDT*, 2018.

[4] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019.

[5] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, 2019.

[6] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners, 2020. `https://arxiv.org/abs/2003.02576`.

[7] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

[8] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2), 2015.

[9] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, 2018.

[10] D. D. Freydenberger. A logic for document spanners. In *ICDT*, 2017.

[11] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. *Theory Comput. Syst.*, 62(4), 2018.

[12] D. D. Freydenberger, B. Kimelfeld, and L. Peterfreund. Joining extractions of regular expressions. In *PODS*, 2018.

[13] F. L. Gall. Improved output-sensitive quantum algorithms for Boolean matrix multiplication. In *SODA*, 2012.

[14] F. L. Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, 2014.

[15] E. Grandjean. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence*, 16(1), 1996.

[16] IBM Research. SystemT, 2018. `https://researcher.watson.ibm.com/researcher/view_group.php?id=1264`.

[17] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013.

[18] K. Losemann and W. Martens. MSO queries on trees: Enumerating answers under updates. In *CSL-LICS*, 2014.

[19] F. Maturana, C. Riveros, and D. Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, 2018.

[20] M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, 2018.

[21] M. Niewerth and L. Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, 2018.

[22] L. Peterfreund. *The Complexity of Relational Queries over Extractions from Text*. PhD thesis, Technion, 2019. `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2019/PHD/PHD-2019-10.pdf`.

[23] L. Segoufin. A glimpse on constant delay enumeration (Invited talk). In *STACS*, 2014.

[24] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6, 09 1977.

[25] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2), 1979.

[26] K. Wasa. Enumeration of enumeration algorithms. *CoRR*, 2016.