Spanner Evaluation over SLP-Compressed Documents

Markus L. Schmid
MLSchmid@MLSchmid.de
Humboldt-Universität zu Berlin
Germany

Nicole Schweikardt
schweikn@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

ABSTRACT
We consider the problem of evaluating regular spanners over compressed documents, i.e., we wish to solve evaluation tasks directly on the compressed data, without decompression. As compressed forms of the documents we use straight-line programs (SLPs) — a lossless compression scheme for textual data widely used in different areas of theoretical computer science and particularly well-suited for algorithms on compressed data.

In data complexity, our results are as follows. For a regular spanner $M$ and an SLP $S$ of size $s$ that represents a document $D$, we can solve the task of model checking and of checking non-emptiness in time $O(s)$. Computing the set $[M](D)$ of all span-tuples extracted from $D$ can be done in time $O(s \cdot |[M](D)|)$, and enumeration of $[M](D)$ can be done with linear preprocessing $O(s)$ and a delay of $O(\text{depth}(S))$, where $\text{depth}(S)$ is the depth of $S$’s derivation tree.

Note that $s$ can be exponentially smaller than the document’s size $|D|$, and, due to known balancing results for SLPs, we can always assume that depth($S$) = $O(\log(|D|))$ independent of $D$’s compressibility. Hence, our enumeration algorithm has a delay logarithmic in the size of the non-compressed data and a preprocessing time that is at best (i.e., in the case of highly compressible documents) also logarithmic, but at worst still linear. Therefore, in a big-data perspective, our enumeration algorithm for SLP-compressed documents may nevertheless beat the known linear preprocessing and constant delay algorithms for non-compressed documents.

CCS CONCEPTS
• Information systems → Information retrieval; • Theory of computation → Database query languages (principles); Data structures and algorithms for data management; Design and analysis of algorithms; Automata extensions; Regular languages.

1 INTRODUCTION
The information extraction framework of document spanners has been introduced in [8] as a formalisation of the query language AQ, which is used in IBM’s information extraction engine SystemT. A document spanner performs information extraction by mapping a document $D$ (i.e., a string) over a finite alphabet $\Sigma$, to a relation over so-called spans of $D$, which are intervals $[i,j]$ with $0 \leq i \leq j \leq |D|+1$. For example, a spanner may map documents $D = d_1d_2 \ldots d_n$ over $\Sigma = \{a,b,c\}$ to the binary relation that contains all pairs $([i,i+1],[j,\ell])$ such that $d_i$ is the first occurrence of symbol $a$ and $d_jd_{j+1} \ldots d_{\ell-1}$ with $i < j$ is some factor over $\{c\}$ (i.e., it consists only of the symbol $c$). Thus, $D = \text{abcca}$ would be mapped to the relation 
\[
\{(1,2), (3,4), (1,2), (4,5), (1,2), (3,5)\}.
\]
It is common to let the attributes of the extracted relations be given by a set $X$ of variables (i.e., span-tuples are mappings from $X$ to the set of spans) and associate a pair of parentheses $\langle \rangle$ with each $x \in X$. These parentheses can be used as markers that mark subwords directly in a document (therefore they mark spans), e.g.,
the subword-marked words
\[
\langle a \rangle \langle x \rangle \langle b \rangle \langle y \rangle \langle c \rangle \langle y \rangle \langle a \rangle, \quad \langle a \rangle \langle x \rangle \langle b \rangle \langle y \rangle \langle c \rangle \langle y \rangle \langle a \rangle \langle c \rangle \langle y \rangle \langle a \rangle
\]
represent $D$ from above with the three mentioned span-tuples encoded by the marker symbols. In this way, spanners can be represented by sets (or languages) $L$ of subword-marked words, i.e., $L$ represents the spanner $\langle L \rangle$ that maps any document $D$ to the set $\langle L \rangle(D)$ of all span-tuples $t$ with the property that marking $D$ with $t$’s spans in the way explained above yields a word from $L$. In this sense, the subword-marked language given by the regular expression $\langle b \cup c \rangle^* \langle a \rangle \langle \Sigma \rangle^* \langle y \rangle \langle c \rangle \langle y \rangle \langle a \rangle \langle \Sigma \rangle^*$ describes the spanner mentioned above. Spanners that can be expressed by regular languages in this way are called regular spanners and have been studied extensively since the introduction of spanners in [8]; we discuss the respective related work in detail below. An example of a regular spanner represented by an automaton can be found in Figure 3.

For regular spanners, typical evaluation tasks can be solved in linear time in data complexity, including the enumeration of all span-tuples of $\langle L \rangle(D)$ with linear preprocessing and constant
delay [2, 9]. Under the assumption that we have to fully process the document at least once, this can be considered optimal.

As a new angle to the evaluation of regular spanners, we consider the setting where the input documents are given in a compressed form, and we want to evaluate spanners directly on the compressed documents without decompressing them. This is especially of interest in big-data scenarios, where the documents are huge, but it is also in general reasonable to assume that textual data is managed in compressed form, simply because the state of the art in algorithms allows for it. Due to redundancies, textual data (especially over natural languages) is often highly compressible by practical compression schemes, and, maybe even more importantly (and in contrast to relational data), many basic algorithmic tasks can be efficiently solved directly on compressed textual data (see, e.g., [6, 17]).

As our underlying compression scheme, we use so-called straight-line programs (SLPs), which compress a document D by a context-free grammar that represents the singleton language \( \{D\} \).

### 1.1 Algorithmics on SLP-Compressed Strings

See Example 4.1 for an SLP of size 16 that represents a document of size 25. An illustrative way to represent SLPs is in form of their derivation trees (see Figure 1). While the full derivation tree is an uncompressed representation, it nevertheless reveals in an intuitive way the structural redundancies exploited by the SLP: for every node label (i.e., non-terminal) we have to store only one subtree rooted by this label. In this regard, Figure 1 only shows the actual SLP in bold, while the redundancies are shown in grey.

The task investigated in this work is to evaluate a spanner, e.g., the one represented by the automaton of Figure 3, on a document given as an SLP, e.g., the one represented by the bold parts of Figure 1. However, we want to avoid to completely construct the document (or the full derivation tree).

SLPs play a prominent role in the context of string algorithms and other areas of theoretical computer science. They are mathematically easy to handle and therefore very appealing for theoretical considerations. Independent of their data-compression applications, they have been used in many different contexts as a natural tool for representing (and reasoning about) hierarchical structure in sequential data (see, e.g., [15, 17, 18, 21, 22, 29]).

SLPs are also of high practical relevance, mainly because many practically applied dictionary-based compression schemes can be converted efficiently into SLPs of similar size, i.e., with size blow-ups by only moderate constants or log-factors (see [1, 6, 14, 17, 26]). Hence, algorithms for SLP-compressed data carry over to these practical formats.

While in the early days of computer science fast compression and decompression was an important factor, it is nowadays common to also rate compression schemes according to how suitable they are for solving problems directly on the compressed data without prior decompression (also called algorithmics on compressed strings). In this regard, SLPs have very good properties: many basic problems on strings like comparison, pattern matching, membership in a regular language, retrieving subwords, etc. can all be efficiently solved directly on SLPs [17]. As demonstrated by our results, this is even true for spanner evaluation.

A possible drawback of SLPs is that computing a minimal size SLP for a given document is intractable (even for fixed alphabets) [4]. However, this has never been an issue for the application of SLPs, since many approximations and heuristics are known that efficiently (i.e., in (near) linear time) compute SLPs that are only a log-factor larger than minimal ones (see [4, 5, 16]).

Since we cannot discuss all relevant papers in the context of algorithmics on SLP-compressed data here, we refer for further reading to the survey [17], the PhD-thesis [6] and the comprehensive introductions of the papers [1, 4].

### 1.2 Regular Spanner Evaluation

The original framework of [8] uses regular spanners to extract relations directly from documents, which can then be further manipulated by relational algebra. Since the string-compression aspect applies only to the first stage of this approach, we are only concerned with regular spanners (for non-regular aspects of spanners see [10, 11, 24, 27]). We note that [24] is also concerned with grammars in the context of spanners, but in a different way: while in our case the documents are represented by grammars (i.e., SLPs), but the spanners are classical regular spanners, [24] considers spanners that are represented by grammars.

We follow the conceptional approach of [27] and consider spanners as regular languages of subword-marked words, as sketched above. In this way, we can abstract from specialised machine models and represent our spanners as classical finite automata (we discuss this aspect in some more detail in Section 3). In order to avoid that the same span-tuple can be represented by different markings, we represent sequences of consecutive marker symbols by sets of marker symbols (e.g., \( a \Rightarrow b \cdot x \Rightarrow y \cdot cc \cdot \overline{0} \) is represented as \( \Rightarrow b (\overline{x}, \overline{y}) cc \cdot \)). This is a common approach and is analogous to the extended sequential VAs introduced in [9] (also used in [2, 3]).

Our spanners can be non-functional, i.e., we allow span-tuples with undefined variables (also called the schemaless semantics in [20]).

Regular spanners can be evaluated very efficiently since they inherit the good algorithmic properties of regular languages (e.g.,

---

1E.g., run-length encoding, and – most notably – the Lemppel-Ziv-family LZ77, LZ78, LZW, etc. which is relevant for practical tools like the built-in Unix utility compress or data formats like GIF, PNG, PDF and some ZIP archive file formats

2Note that some spanner description models can be substantially more concise than others; this is discussed in more detail in Section 3.3.
model checking for regular spanners is a special variant of the membership problem for regular languages; see [2, 3, 8, 20, 23] for further details. A new aspect that has not been considered in formal language theory is that of enumerating all query results (i.e., span-tuples). This has been considered in [2, 9, 12] and it is a major result that constant delay enumeration is possible after linear preprocessing (even if the spanners are given by non-deterministic automata); see especially the survey [3]. The algorithmic approach is to construct the product graph of the automaton that represents the spanner (e.g., the one of Figure 3) and the input document (treated as a path). This yields a directed acyclic graph that fully represents the solution set and which can be used for enumeration (Figure 1 of [3] illustrates this construction in a single picture).

The main challenge of the present paper is that the above described construction is not possible in our setting, since it requires the input document to be decompressed. We aim to represent all runs of the automaton on the decompressed document, while respecting the document’s compressed form given by the SLP.

### 1.3 Our Contribution

We investigate the following tasks, for which we get as input an SLP $S$ (of size $s$) for a document $D$ (of size $d$) and a spanner represented by an automaton $M$:

- **non-emptiness**: check if $\lfloor M \rfloor(D) \neq \emptyset$
- **model checking**: check if $t \in \lfloor M \rfloor(D)$ for a given span-tuple $t$
- **computation**: compute the whole set $\lfloor M \rfloor(D)$
- **enumeration**: enumerate the elements of $\lfloor M \rfloor(D)$

Let $r$ denote the number of result tuples (i.e., span-tuples) in $\lfloor M \rfloor(D)$. In terms of data complexity, our main results solve

1. non-emptiness and model checking in time $O(s)$,
2. computation in time $O(s \cdot r)$,
3. enumeration with delay $O(\log d)$ after $O(s)$ preprocessing.

Note that (3) also implies a solution for computation in time $O(s + r \cdot \log d)$ (however, our direct algorithm for computing $\lfloor M \rfloor(D)$ is much simpler and better in combined complexity).

These runtimes are incomparable to the known runtimes on uncompressed documents, which solve non-emptiness and model checking in time $O(d)$, computation in time $O(d + r)$, and enumeration with delay $O(1)$ after $O(d)$ preprocessing. But note that, for highly compressible documents (due to structural redundancies like repetitions), $s$ might be exponentially smaller than $d$, and in these cases our algorithms will outperform the approach of first decompressing the entire document and then applying an efficient algorithm on uncompressed documents. In the case of highly compressible documents, our setting can also be considered as spanner evaluation with sublinear data complexity.

In terms of combined complexity, the $O$-notation in our runtime guarantees hides some (low degree) polynomial factors in $|M|$ (the total size of the automaton), $|Q|$ (the number of $M$’s states), and $|X|$ (the number of span variables); the precise bounds in combined complexity are stated in Theorems 5.1, 7.1 and 8.10. We wish to point out that the aspect of conciseness of different spanner representations is hidden in the factor $|M|$. The automata we use are, in terms of conciseness, like (non-deterministic) extended VAs (see [2, 3, 9]); and for enumeration (but only for enumeration) we additionally need the automata to be deterministic.

### 1.4 Technical approach

Model checking and checking non-emptiness can be done in a rather straightforward way by a reduction to the problem of checking membership of an SLP-compressed document to a regular language. For computing or enumerating the solution set, we have to come up with new ideas.

Intuitively speaking, the compression of SLPs is done by representing several occurrences of the same factor of a document by just a single non-terminal, e.g., the three occurrences of factor $\texttt{aa}$ are represented by $E$ in the SLP $S$ of Figure 1. However, the span-tuples to be extracted may treat different occurrences of the same factor compressed by the same non-terminal in different ways. For example, the spanner $M$ of Figure 3 may extract the span-tuple that corresponds to $\texttt{aabc} \Rightarrow \texttt{aba} \Rightarrow \texttt{a}$. This messes up the compression, since the three occurrences of $\texttt{aa}$ have now become three different factors: $\texttt{aa} \Rightarrow \texttt{a}$ and $\texttt{a} \Rightarrow \texttt{a}$. So it seems that extracting a span-tuple enforces at least a partial decompression of $S$ (since different occurrences of the same factor need to be treated differently).

The technical challenge that we face also becomes clear by a comparison to the approach of [2] (for spanner evaluation in the uncompressed case), which first computes in the preprocessing data structure that represents the whole solution set (i.e., the product graph of spanner and document), and then the enumeration is done by systematically searching this data structure (with the help of additional, pre-computed information). Since each position of the document might be the start or end position of some extracted span, it is difficult to imagine such a data structure that is not at least as large as the whole document. Therefore, this approach seems impossible in our setting.

In our approach, we enumerate SLPs that represent marked variants of the document. As illustrated above, these SLPs must be at least partially decompressed. However, since we must only accommodate the at most $2|X|$ positions of the document that are start or end positions of the spans of a fixed span tuple, the required decompression is still bounded in terms of the spanner. We show that the breadth of these partially decompressed SLPs is bounded by $O(|X|)$. Their depth, however, can be as large as the depth of the input SLP representing the document. By a well-known balancing theorem [13], this depth can be assumed to be logarithmic in the size of the (uncompressed) document.

### 1.5 Organisation

Section 2 fixes basic notation, Sections 3 and 4 provide background on document spanners and SLPs, respectively. Section 5 is devoted to model checking and checking non-emptiness. Section 6 develops a tool box that is used in Sections 7 and 8 for computing and for enumerating the result set. We conclude in Section 9. Due to space restrictions, we only provide proof sketches for some results (full proofs for all results are available in the paper’s preliminary full version [28]).

### 2 BASIC DEFINITIONS

Let $\mathbb{N} = \{1, 2, 3, \ldots \}$ and $|n| = \{1, 2, \ldots, n\}$ for $n \in \mathbb{N}$. For a (partial) mapping $f : X \rightarrow Y$, we write $f(x) = \bot$ for some $x \in X$ to denote that $f(x)$ is not defined; and we set $\text{dom}(f) = \{x : f(x) \neq \bot\}$. By $\mathcal{P}(A)$ we denote the power set of a set $A$, and $A^*$ denotes the
set of non-empty words over $A$, and $A^* = A^* \cup \{\varepsilon\}$, where $\varepsilon$ is the empty word. For a word $w \in A^*$, $|w|$ denotes its length (in particular, $|\varepsilon| = 0$), and for every $b \in A$, $|w|_b$ denotes the number of occurrences of $b$ in $w$. A word $w \in A^*$ is a factor or a subword of a word $w \in \Sigma^*$ if there are $u_1, u_2 \in \Sigma^*$ with $w = u_1 w u_2$.

For all our algorithmic considerations, we assume the RAM-model with logarithmic word-size as our computational model.

A nondeterministic finite automaton (NFA for short) is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ with a finite set of states, a finite alphabet $\Sigma$, a start state $q_0 \in Q$, a set $F \subseteq Q$ of accepting states and a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$. We also interpret NFA as directed, edge-labelled graphs in the obvious way.

We extend the transition function to $\delta : Q \times \Sigma^* \rightarrow P(Q)$ in the usual way, i.e., for $w \in \Sigma^*$, $x \in \Sigma \cup \{\varepsilon\}$ and $p \in Q$, we set $\delta(p, wx) = \bigcup_{q \in \delta(p, w)} \delta(q, x)$. If $M$ and its transition function $\delta$ is clear from the context, we also write $p \xrightarrow{w} q \xrightarrow{v} r$ instead of $p \xrightarrow{w} q$ and $q \xrightarrow{v} r$, and we write $p \xrightarrow{F} q$ to denote that there is some $q \in F$ with $p \xrightarrow{w} q$. A word $w \in \Sigma^*$ is accepted by $M$ if $q_0 \xrightarrow{w} F$; and $L(M) = \{w : q_0 \xrightarrow{w} F\}$ is the language accepted by $M$.

An NFA $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite automaton (DFA for short) if, for every $p \in Q$ and $x \in \Sigma \cup \{\varepsilon\}$, $\delta(p, x) = \emptyset$ if $x = \varepsilon$, and $|\delta(p, x)| \leq 1$ if $x \in \Sigma$. In this case we view $\delta$ as a function from $Q \times \Sigma$ to $Q$, and we extend it to $\delta : Q \times \Sigma^* \rightarrow Q$ by setting $\delta(p, wx) = \delta(p, w, x)$ and we write $p \xrightarrow{w} q$ to denote $\delta(p, w) = q$.

The size $|M|$ of an NFA is the number of its transitions. As a convention for the rest of the paper, we always assume for NFA that $Q = \{1, 2, \ldots, q\}$, for some $q \in \mathbb{N}$, and $q_0 = 1$. In particular, this means that $q = |Q|$ throughout the rest of this paper.

### 3 DOCUMENT SPANNERS

Let $\Sigma$ be a terminal alphabet of constant size, and in the following we call words $D \in \Sigma^*$ document. For a document $D \in \Sigma^*$, we denote by $d$ its length and for every $i, j \in [d+1]$ with $i \leq j$, $(i, j)$ is a span of $D$ and its value, denoted by $D(i, j)$, is the subword of symbol $i$ to symbol $j-1$. The special case $D[i, i+1]$ is denoted by $D[i]$. Spans(D) denotes the set of spans of $D$, and by Spans we denote the set of spans for any document, i.e., $\{[i, j] : i, j \in \mathbb{N}, i \leq j\}$ (elements from Spans are simply called spans).

For a finite set of variables $X$, an $(X, D)$-tuple is a partial function $X \rightarrow \text{Spans}(D)$. For simplicity, we usually denote $(X, D)$-tuples in tuple-writing, for which we assume an order on $X$ and use the symbol $\cdot$ for undefined variables, e.g., $\{(x_1, x_2, x_3) \in D\}$-tuple that maps $x_1 \rightarrow [1, 5], x_3 \rightarrow [5, 7)$, and is undefined for $x_2$. Since the dependency on the document $D$ is often negligible, we also use the term $X$-tuple (or span-tuple over $X$)) to denote an $(X, D)$-tuple.

We also define an obvious set-representation of span-tuples that will be convenient in the context of this work. For any set $X$ of variables, we use a special alphabet $\Gamma_X = \{\varepsilon, \sigma^i : x \in X\}$. This alphabet shall play an important role in the remainder of this work; its elements are also called markers. For any $(X, D)$-tuple $t$, its marker set $\hat{t} \subseteq X \times [d+1]$ is defined as $\hat{t} = \{(\varepsilon, i), (\sigma^j, i) : (x) = [i, j), x \in \text{dom}(t)\}$. It is obvious that there is a one-to-one correspondence between span-tuples and their marker sets.

An $(X, D)$-relation (or $X$-relation if the dependency on $D$ is negligible) is a set of $(X, D)$-tuples. As a measure of the size of a reasonable representation of an $(X, D)$-relation we use size($R$) = $|X| \cdot |R|$. A spanner (over terminal alphabet $\Sigma$ and variables $X$) is a function that maps every document $D \in \Sigma^*$ to an $(X, D)$-relation (note that the empty relation $\emptyset$ is also a valid image of a spanner).

We next introduce some terminology that will be crucial for reasoning about spanners and span-tuples. We follow the common approach in the literature to represent a pair of document $D$ and span-tuple $t$ as a single word (which will be called subword-marked word) by means of special marker symbols that are inserted into the document (for which we use the symbols of $\Gamma_X$). For example $D = abab$ and span-tuple $t$ with $t(x) = (2, 4)$ and $t(y) = (3, 5)$ can be represented by the subword-marked word $ab_s \sigma^2 \sigma^6 b_\sigma^3 b_\sigma^9$.

### 3.1 Subword-Marked Words

For any set $X$ of variables, we shall use the set $\Gamma_X = \{\varepsilon, \sigma^i : x \in X\}$ and its powerset as alphabets. The intuitive meaning of an occurrence of symbol $\varepsilon$ (or $\sigma^i$) at position $i$ is that the span of variable $x$ starts at position $i$ (or ends at position $i$, respectively). If spans of several variables start or end at the same position, we encode this by using a subset of $\Gamma_X$ as a single symbol.

**Definition 3.1**. A subword-marked word (over $\Sigma$ and $X$) is a word $w = A_1b_1A_2b_2 \ldots A_nb_nA_{n+1}$ with $b_i \in \Sigma$ for every $i \in [n]$, and $A'_i \in \mathcal{P}(\Gamma_X)$ for every $i' \in [n+1]$, that satisfies the properties:

- for all distinct $i, j \in [n+1], A_i \cap A_j = \emptyset$,
- if $\varepsilon \in A_i$ and $\sigma^i \in A_j$ for $x \in X$, then $i \leq j$,
- for all $x \in X$, $\{\varepsilon, \sigma^i\}$ is contained in or disjoint from $\bigcup_{i=1}^{n+1} A_i$.

We define the document-length of $w$ as $|w|_{\text{doc}} = n$ (note that the actual length of $w$ is $|w| = 2|w|_{\text{doc}} + 1$; the document-length will be the more relevant size measure for us). For convenience, we also omit symbols $A_i$ if they are the empty set.

We claimed above that subword-marked words represent a document and a span-tuple as a single word. We shall now substantiate this interpretation of subword-marked words by defining the function $\varepsilon(\cdot)$ that retrieves the document and the function $\psi(\cdot)$ that retrieves the span-tuple (as marker set) encoded by a subword-marked word. To this end, let $w = A_1b_1A_2b_2 \ldots A_nb_nA_{n+1}$ be a subword-marked word over $\Sigma$ and $X$. By $\varepsilon(w)$, we denote the document over $\Sigma$ obtained by erasing all occurrences of symbols from $\mathcal{P}(\Gamma_X)$ from $w$, i.e., $\varepsilon(w) = b_1b_2 \ldots b_n$ (note that $|w|_{\text{doc}} = |\varepsilon(w)|$). Furthermore, let $\psi$ be the set $\{\sigma(i) : i \in A_i \in [n + 1]\}$. It can be easily seen that $\psi(w)$ is the marker set $T$ of an $(X, \varepsilon(w))$-tuple $t$.

For given document $D$ and an $(X, D)$-tuple $t$, it is obvious how to construct a subword-marked word $w$ with $\varepsilon(w) = D$ and $\psi(w) = T$. We will nevertheless formally define this. For any $(X, D)$-tuple $t$, we denote by $\psi(D, t)$ the word $A_1b_1A_2b_2 \ldots A_nb_nA_{n+1}$, where $b_1 = D[i]$ for every $i \in [d]$, and, for every $i' \in [d+1], A'_i = \{\sigma : (\sigma, i') \in T\}$. It can be easily seen that $\psi(D, t)$ is in fact a subword-marked word with $\varepsilon(w) = D$ and $\psi(w) = T$.

Let us illustrate these definitions with a brief example (see also Figure 2 for an illustration of the mappings $\varepsilon(\cdot), \psi(\cdot)$ and $\psi(\cdot, \cdot)$ that translate between the different representations).
We now formally describe the concept of straight-line programs (SLPs, for short), that has already been discussed in the introduction.

### 4.1 Straight-Line Programs

A context-free grammar \( G = (N, \Sigma, R, S_0) \), where \( N \) is the set of non-terminals, \( \Sigma \) is the terminal alphabet, \( S_0 \in N \) is the start symbol and \( R \subseteq N \times (N \cup \Sigma)^* \) is the set of rules (as a convention, we write rules \((A, w) \in R \) also in the form \( A \rightarrow w \)). A context-free grammar \( S = (N, \Sigma, R, S_0) \) is a straight-line program (SLP) if \( R \) is a total function \( N \rightarrow (N \cup \Sigma)^* \) and the relation \((A, B) : (A, w) \in R, |w|_\Sigma \geq 1 \) is acyclic. In this case, for every \( A \in N \), let \( D_S(A) \) be the unique \( w \in (N \cup \Sigma)^* \) such that \((A, w) \in R \) and let \( D_S(A) = a \) for every \( a \in \Sigma \); we also call \( A \) to \( D_S(A) \) the rule for \( A \). For an SLP, \( S = (N, \Sigma, R, S_0) \), we extend \( D_S \) to a morphism \((N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^* \) by setting \( D_S(\alpha_1 \ldots \alpha_n) = D_S(\alpha_1) \ldots D_S(\alpha_n) \), for \( \alpha_i \in (N \cup \Sigma), 1 \leq i \leq n \). Furthermore, for

3.3 Representations of Regular Spanners

In the initial paper [8], regular spanners were represented by so-called variable-set automata (VA, for short). In our terminology, VAs are NFAs that accept subword-marked languages with the difference that consecutive marker symbols are explicitly represented as sequences and not merged into sets. As a result, a document and a span-tuple do not describe a subword-marked word in a unique way (i.e., the function \( m(\cdot, \cdot) \) is not well-defined), which means that for solving model checking according to Proposition 3.3, we potentially need to consider an exponential number of subword-marked words. This is a well-known problem and can be dealt with by restricting spanners to be functional (i.e., span-tuples are total functions) [9, 11], by imposing a fixed order on sequences of marker symbols in the subword-marked words [7, 27], or by using sets of marker symbols as symbols, as done for extended VAs [2, 9] and also in this paper.

It is well-known that the VAs of [8] can be transformed into extended VAs, or into VAs with an order on the marker symbols, or into NFAs for subword-marked languages (in the way defined here); see, e.g., [2, 9]. However, these translations cause an exponential size blow-up in the worst-case (this is formally proven in [9]), except for functional VAs (on the other hand, functionality is a proper restriction compared to non-functional regular spanners).

We present our results in a way that abstracts from these well-documented issues of conversions between different representations of regular spanners, since they would distract from the actual story of this paper, which is spanner evaluation on compressed documents. In order to extend our results to other spanner formalisms, one has to keep in mind the overhead of translations between formalisms (which affects the combined complexity, but not the data complexity). More precisely, translating a compact spanner representations like (non-extended) VA to the NFA spanner-representation considered in this paper may impose an exponential size blow-up in the worst case, and determining the NFA (as required in our result about enumeration; see Theorem 8.10) may even cause another exponential size blow-up. However, these blow-ups are exponential only in the worst case, exhibited only by special instances.
every \( \alpha \in (N \cup \Sigma)^* \), we set \( D^1_S(\alpha) = D_S(\alpha), D^k_S(\alpha) = D_S(D^{k-1}_S(\alpha)) \), for every \( k \geq 2 \); and \( \hat{D}_S(\alpha) = D^{|N|}_S(\alpha) \) is the derivative of \( \alpha \). By definition, \( \hat{D}_S(\alpha) \in \Sigma^+ \) for every \( \alpha \in (N \cup \Sigma)^+ \).

The depth of a non-terminal \( A \in N \) is defined by \( \text{depth}(A) = \min \{ k : D^k_S(A) = \hat{D}_S(A) \} \), and the depth of \( S \) is \( \text{depth}(S) = \text{depth}(S_0) \). The size of \( S \) is defined by \( \text{size}(S) = |N| + \sum_{A \in N} |D_S(A)| \).

If the SLP under consideration is clear from the context, we also drop the subscript \( S \). Moreover, we set \( \Xi(S) = \hat{D}(S) \) and say that \( S \) is an SLP for \( (\text{the word or document}) \) \( \Xi(S) \). We view \( S \) as a compressed representation of the document \( \Xi(S) \).

The derivation tree of an SLP \( S = (N, \Sigma, R, S_0) \) is a ranked ordered tree with node-labels from \( \Sigma \cup N \), inductively defined as follows. The root is labelled by \( S_0 \) and every node labelled by \( A \in N \) with \( D_S(A) = \alpha_1 \alpha_2 \ldots \alpha_n \) has \( n \) children labelled by \( \alpha_1, \alpha_2, \ldots, \alpha_n \) in exactly this order. We note that all leaves of the derivation tree are from \( \Sigma \), and spreading them out from left to right yields exactly \( \hat{D}(S) \); moreover, the derivation of the derivation tree is exactly \( \text{depth}(S) \). See Figure 1 for an example of a derivation tree. We stress the fact that the derivation tree of an SLP \( S \) is a non-compressed representation of \( \Xi(S) \). In particular, algorithms on SLP-compressed strings cannot afford to explicitly build the full derivation tree.

Example 4.1. Let \( S = (N, \Sigma, R, S_0) \) be an SLP with \( N = \{ S_0, A, B \} \), \( \Sigma = \{ a, b \} \), and \( R = \{ S_0 \rightarrow ABaBBa, A \rightarrow BaB, B \rightarrow ba b \} \). By definition, \( \Xi(A) = \hat{X}(A) = \hat{X}(S_0) = baababaaabaaabbaababbaababbaaabbaababbaabab \). Thus, \( S \) is an SLP for \( baababaaabaaabbaababbaababbaababbaabab \). In particular, we note that \( \text{size}(S) = 16 < 25 = |\Xi(S)| \).

From now on, we shall always denote the document compressed by the SLP by \( D \) (i.e., \( \Xi(S) = D \) for the SLPs \( S \) that we consider). Recall that we denote by \( d \) the size of \( D \).

An SLP \( S = (N, \Sigma, R, S_0) \) is in Chomsky normal form if, for every \( A \in N \), \( D_S(A) \in (\Sigma \cup N^2) \), and \( S \) is c-balanced for some \( c \in \mathbb{N} \) if \( \text{depth}(S) \leq c \log(d) \). We note that if \( S \) is in Chomsky normal form, then \( \text{size}(S) \leq 3|N| \). We say that an SLP in normal form if it is in Chomsky normal form and, for every \( x \in \Sigma \), \( T_x \) is the unique non-terminal with rule \( T_x \rightarrow x \). We call the \( T_x \) leaf non-terminals and all other \( A \in N \setminus \{ T_x : x \in \Sigma \} \) inner non-terminals. For SLPs in normal form, we let the leaf non-terminals be the leaves of derivation trees. From now on, we assume that all SLPs are in normal form.

Example 4.2. Let \( S = (N, \Sigma, R, S_0) \) be a normal form SLP with \( N = \{ S_0, A, B, C, D, E, T_0, T_1, T_2 \} \), \( \Sigma = \{ a, b, c \} \), and \( R = \{ S_0 \rightarrow AB, A \rightarrow CD, B \rightarrow CE, C \rightarrow ET_0, D \rightarrow T_2T_1, E \rightarrow T_0T_1 \} \). Figure 1 shows the derivation tree of \( S \). It can be easily verified that \( \Xi(S) = \text{aabccaaabaa} \).

4.2 Further Properties of SLPs

The size of an SLP can be logarithmic in the size of the document, e.g., strings \( a^m \) can be represented by \( n + 1 \) rules of the form \( S \rightarrow A_1A_2 \ldots A_n \). On the other hand, it can be shown that \( \log(d) \) is also an asymptotic lower bound for \( \text{size}(S) \) (see [5, Lemma 1]). Another important parameter is \( \text{depth}(S) \). E.g., finding in an SLP the \( i \)-th symbol \( D[i] \) of the document represented by \( S \) can be achieved by a top-down traversal of the derivation tree, which depends on \( \text{depth}(S) \). For SLPs with a constant branching factor (like SLPs in normal form), \( \text{depth}(S) \) is also lower bounded by \( \log(d) \). This optimum is achieved by balanced SLPs and the following theorem shows that it is in fact without loss of generality to assume SLPs to be balanced:

Theorem 4.3 (SLP Balancing Theorem, Ganardi, Jeż and Lohrey [13]). There is a \( c \in \mathbb{N} \) such that any given SLP \( S \) for document \( D \) can be transformed in time \( O(\text{size}(S)) \) into a c-balanced SLP \( S' \) for \( D \) in Chomsky normal form with size \( |\Xi(S')| = O(\text{size}(S)) \).

Theorem 4.3 means that whenever a factor depth\((S)\) occurs in the running time, which, in the general case, can only be upper bounded by \( \text{size}(S) \), it can be replaced by \( \log(d) \), which corresponds to \( \text{size}(S) \) in the best-case compression scenario. For clarity, we nevertheless mention any dependency on \( \text{depth}(S) \) in our results.

In the field of algorithmics on (SLP-)compressed strings, it is common to assume the word-size of the underlying RAM-model to be logarithmic in \( d \), where \( d \) is the size of the non-compressed input. This means that we can perform arithmetical operations on the positions of \( D \) in constant time. In particular, we state the following fact, which is easy to show and well-known in the context of SLPs.

Lemma 4.4. Given an SLP \( S \), we can compute all the numbers \( |\Xi(A)| \) for all non-terminals \( A \) within time \( O(\text{size}(S)) \).

4.3 SLPs and Finite Automata

A classical task in the context of algorithms on SLP-compressed strings is to check membership of an SLP-compressed document \( D \) to a given regular language \( L \). It is intuitively clear that algorithms for our spanner evaluation tasks (see Section 1) will necessarily also implicitly solve this task in some way. For example, given an SLP for \( D \) and an NFA \( M \) over \( \Sigma \), checking if \( D \in L(M) \) reduces to the model checking task \( \varphi \in [M](D) \). Hence, we discuss checking membership of SLPs to regular languages in a bit more detail.

Let \( S \) be an SLP for \( D \) and let \( M \) be an NFA with \( q \) states. The general idea is to compute, for each \( A \in N \), a Boolean \( (q \times q) \) matrix \( M_A \) whose entries indicate from which state we can reach which state by reading \( \Xi(A) \). This can be done recursively along the structure of \( S \): the matrices \( M_{T_x} \) for the leaf non-terminals are directly given by \( M \)'s transition function, and for every inner non-terminal \( A \in N \) with a rule \( A \rightarrow BC \), we have \( M_A = M_B \cdot M_C \) (where \( \cdot \) denotes the usual Boolean matrix multiplication). This yields the following well-known result, that has been formally stated at several places in the literature (see, e.g., [17, 19, 25]):

Lemma 4.5. Let \( S \) be an SLP for \( D \) and let \( M \) be an NFA with \( q \) states. Then we can check whether \( D \in L(M) \) in time \( O(\text{size}(S) q^2) \).

With a fast Boolean matrix multiplication algorithm that runs in time \( O(q^\omega) \), Lemma 4.5 can be improved to \( O(\text{size}(S) q^\omega) \). In fact, the best known upper bound is \( O(\min\{\text{size}(S) q^{\omega}, |D| q^2\}) \) (the latter running-time is achieved by explicitly constructing \( D \)). However, for “combinatorial algorithms”, this bound simplifies to \( O(\min\{\text{size}(S) q^3, |D| q^2\}) \) and it is shown in [1] that, conditional to the so-called combinatorial k-Clique conjecture, this is optimal in the sense that there is no “combinatorial algorithm” with running-time \( O(\min\{\text{size}(S) q^3, |D| q^2(1-\epsilon)\}) \) for any \( \epsilon > 0 \).

5 NON-EMPTINESS AND MODEL CHECKING

In this section, we consider the non-emptiness and the model checking problem (see Section 1), which can be reduced to the problem
of checking membership of an SLP-compressed document to a regular language. Here, we provide a sketch of how this can be done (for the following explanations, recall the definitions of the functions $\epsilon(\cdot)$, $p(\cdot)$ and $m(\cdot \epsilon)$ (Section 3.1), which are also illustrated by Example 3.2 and Figure 2).

For checking if $[M](D) \neq \emptyset$, it suffices to check whether $M$ can accept a subword-marked word $w$ with $\epsilon(w) = D$. This can be easily done by treating all $P(\Gamma_X)$-transitions of $M$ as $\epsilon$-transitions and then simply check membership of $D$ by using Lemma 4.5.

For checking if $t \in [M](D)$ for a given span-tuple $t$, we proceed as follows. We transform the SLP $S$ for $D$ into an SLP $S'$ for the subword-marked word $w = m(D, t)$ (recall from Section 3 that $\epsilon(w) = D$ and $p(w) = t$). Since $t \in [M](D)$ if and only if $w \in L(M)$ (see Proposition 3.3), it suffices to check whether $\Sigma(S') \subseteq L(M)$ (for which we can rely again on Lemma 4.5). The only question left is how to construct $S'$, and this can be done as follows. For every $i \in [d]$ such that there is at least one $(\sigma, i) \in \tilde{t}$, we compute the set $\Lambda_i = \{\sigma : (\sigma, i) \in \tilde{t}\}$. Note that there are at most $2|X|$ such sets, and these can be easily obtained from $\tilde{t}$ in time $O(|X|)$. Then, for each such set $\Lambda_i$, we traverse the derivation tree of $S$ top-down in order to find the leaf corresponding to position $i$ (for this, the numbers $|\Sigma(A)|$ are essential, which we can compute according to Lemma 4.4). Then we add the symbol $\Lambda_i$ at this position, but, since this changes the meaning of all the non-terminals of this root-to-leaf path, we have to introduce depth($S$) new non-terminals. Overall, we only add $O(|X| \cdot \text{depth}(S))$ new non-terminals to $S'$; in particular, we never have to construct the whole derivation tree, but at most $2|X|$ paths of length depth($S$). This leads to:

Theorem 5.1. Let $S$ be an SLP for $D$, let $M$ be an NFA with $q$ states that represents a $(\Sigma, X)$-spanner, and let $t$ be an $(X, D)$-tuple. Checking whether

1. $[M](D) \neq \emptyset$ can be done in time $O(|M| + \text{size}(S)q^3)$.
2. $t \in [M](D)$ can be done in time $O(|\text{size}(S)| + |X| \cdot \text{depth}(S))q^3$.

6 ALGORITHMIC PRELIMINARIES

In this section, we develop a tool box for spanner evaluation over SLPs. On the conceptual side, we first extend our definitions from Section 3 to the case of incomplete (or partial) span-tuples (which is necessary to reason about the subwords of the document compressed by single non-terminals of the SLP). Then, we present a sequence of lemmas that allow us to regard the solution set $[M](D)$ as being decomposed according to the recursive structure of the SLP. This point of view will be crucial both for the task of computing (Section 7) and of enumerating (Section 8) the set $[M](D)$.

6.1 Representations of Partial Span-Tuples

Recall Example 3.2 for document $D = abbcab$:

$$w = \{b\}ab\{b\}, 3, \{s\}bc\{s\}ab\{s\}ac,$$

$$p(w) = \{(s, 1), (s, 3), (s, 3), (s, 3), (s, 3), (s, 5)\}.$$ 

If we consider the factorisation $D = D_1 D_2$ with $D_1 = abbc$ and $D_2 = cab$, then this corresponds to the factorisation $w = w_1 w_2$ with $w_1 = \{b\}ab\{b\}, 3, \{s\}bc\{s\}ab\{s\}ac$. Technically, neither $w_1$ nor $w_2$ are sub-word-marked words. However, it can be easily seen that the functions $\epsilon(\cdot)$ and $p(\cdot)$ are still well-defined and

$\epsilon(w_1) = D_1, p(w_1) = \{(s, 1), (s, 3), (s, 3), (s, 3), (s, 3)\},$

$\epsilon(w_2) = D_2, p(w_2) = \{(s, 2), (s, 4)\}$. The sets $p(w_1)$ and $p(w_2)$ are not valid marker sets that describe valid span-tuples, but we can interpret them as representing partial span-tuples. Moreover, we can also combine $p(w_1)$ and $p(w_2)$ in order to obtain the marker set of the whole span-tuple, but we have to keep in mind that $p(w_2)$ corresponds to a factor of $D$ that is not a prefix and therefore the elements from $p(w_2)$ have to be shifted to the right by $|D_1| = 3$ positions. We now formalise these observations.

As a factor of a subword-marked word is called a marked word. Since marked words are words $w = A_1 b_1 \ldots A_n b_n A_{n+1}$ with $b_i \in \Sigma$ and $A_i \in P(\Gamma_X)$ (except for the possibility that $A_1$ or $A_{n+1}$ are missing, which we can simply interpret as $A_1 = \emptyset$ or $A_{n+1} = \emptyset$, respectively), the functions $\epsilon(\cdot)$ and $p(\cdot)$ can be defined in the same way as for subword-marked words, i.e., $\epsilon(w) = b_1 b_2 \ldots b_n$ and $p(w) = \{(\sigma, i) : \sigma \in A_i, i \in [n + 1]\}$.

For any marked word $w$, we call the set $p(w)$ a partial marker set, and we shall denote partial marker sets by $\Lambda$ in order to distinguish them from span-tuples and from (non-partial) marker sets.

As long as a partial marker set $\Lambda$ is compatible with a document $D$, i.e., $\max|\ell| : (\sigma, \ell) \in \Lambda| \leq \ell + 1$, we can also define $m(\Lambda, \Lambda)$ analogously as for non-partial marker sets, i.e., $m(\Lambda, \Lambda) = A_1 b_1 \ldots A_n b_n A_{n+1}$, where $b_i = D[\ell]$ for every $i \in [d]$ and, for every $i' \in [d + 1], A_{i'} = \{\sigma, i' \in \Lambda\}$. Note that the diagram of Figure 2 still serves as an illustration (we just have to keep in mind that $t$ is now a partial span-tuple).

For any partial marker set $\Lambda$ and any $\ell \in \mathbb{N}$, the $\ell$-rightshift of $\Lambda$, denoted by $\Lambda_{\ell}$, is the partial marker set $\{(\sigma, k + \ell) : (\sigma, k) \in \Lambda\}$.

Example 6.1. Let $\Sigma = \{a, b, c\}, X = \{x, y, z\}$. The partial marker sets $\Lambda_1 = \{(2, y), (4, z), (6, x), 7\}$ and $\Lambda_2 = \{(2, s), (4, x)\}$ are compatible with $D_1 = abbc$ and $D_2 = cab$, respectively, but are both not marker sets of some span-tuple. Moreover, $m(D_1, \Lambda_1) = a\{y\}ba\{z\}bc\{c\}c, \Lambda(D_2, \Lambda_2) = c\{a\}ba\{s\}a$. We observe that

$\Lambda = \Lambda_1 \cup \Lambda_2 | \Lambda(D_2, \Lambda_2) = \{(2, y), (4, z), (6, x), (8, x), (10, x)\}$

is a marker set for $D = D_1 D_2$, and $m(D, \Lambda) = m(D_1, \Lambda_1) m(D_2, \Lambda_2)$.

For any subword-marked word $w$ with $\epsilon(w) = D$ and any factorisation $D = D_1 D_2$, there might be two ways of factorising $w = w_1 w_2$ such that $\epsilon(w_1) = D_1$ and $\epsilon(w_2) = D_2$ (i.e., depending on whether the symbol from $P(\Gamma_X)$ at the cut point belongs to $w_1$ or to $w_2$).

We will deal with this issue, we will only consider marked words that end on a symbol from $\Sigma$. This is only possible, if all our subword-marked words are non tail-spanning, which means that the final symbol $A_{|w|+1}$ from $P(\Gamma_X)$ is empty (and therefore, can be ignored). We say that a subword-marked language $L$ (i.e., a spanner) is non tail-spanning if every $w \in L$ is non tail-spanning.

We assume all regular spanners to be non-tail spanning in the remainder of this paper. Note that this is a very minor restriction: any NFA $M$ that represents a $(\Sigma, X)$-spanner can be easily transformed into an NFA $M'$ with $L(M') = \{w : w \in L(M)\}$ for some $\# \notin \Sigma$. In particular, this means that $[M']$ is non-tail spanning and, for every document $D$, we have $[M](D) = [M'](D \#)$. 
6.2 Technical Lemmas

In the following, let $S = (N, \Sigma, \delta, S_0)$ be an SLP for $D$, and let $M = (Q, \Sigma, 1, \delta, F)$ be an NFA with $Q = \{1, 2, \ldots, q\}$ that represents a $(2, X)$-spanner.

The following definition is central for our evaluation algorithms (recall that for $i, j \in [q]$ we denote by $i \xrightarrow{w} j$ that $w$ takes $M$ from state $i$ to state $j$, i.e., $j \in \delta(i, w)$).

**Definition 6.2.** For any non-terminal $A \in N$, we define a $(q \times q)$-matrix $M_A$ as follows. For every $i, j \in [q]$, $M_A[i, j]$ is a set that contains exactly the partial marker sets $\Delta$ such that

- $\Delta$ is compatible with $\Sigma(A)$,
- $m(\Sigma(A), \Delta)$ is non-tail-spanning, and
- $i \xrightarrow{w} j$.

Intuitively speaking, $M_A$ contains all the information of how the spanner represented by $M$ operates on the word $\Sigma(A)$; thus, $M_{S_0}$ can be interpreted as a representation of $[M](D)$. This is formalised by the next lemma. Recall that $F$ denotes $M$’s set of accepting states and $1$ is $M$’s start state.

**Lemma 6.3.** $[M](D) = \bigcup_{i \in F} M_{S_0}[1, i]$.

This means that computing or enumerating the set $[M](D)$ reduces to the computation or enumeration of the sets $M_{S_0}[1, j]$ with $j \in F$. The purpose of the remaining notions and lemmas of this section is to show how we can recursively construct the entries of the matrices $M_A$ along the structure of the SLP.

Note that for each $A \in N$ and $i, j \in [q]$, there are three possible (mutually exclusive) cases of how the set $M_A[i, j]$ looks like:

- There is no marked word $w$ with $i \xrightarrow{w} j$ and $\epsilon(w) = \Sigma(A)$.
- The only possible marked word $w$ with $i \xrightarrow{w} j$ and $\epsilon(w) = \Sigma(A)$ is the word $w = \Sigma(A)$ (i.e., $\delta(w) = \emptyset$).
- There is at least one marked word $w$ with $i \xrightarrow{w} j$ and $\epsilon(w) = \Sigma(A)$ that actually contains markers (i.e., $\delta(w) \neq \emptyset$).

This means that $M_A[i, j]$ is neither $\emptyset$ nor $\emptyset$. For the computation (and enumeration) of the sets $M_{S_0}[1, j]$ with $j \in F$ (and therefore the set $[M](D)$) it will be a crucial preprocessing step to compute for every $A \in N$ and $i, j \in [q]$, which of the three cases mentioned above apply.

Moreover, for any rule $A \rightarrow BC$ of $S$, for every marked word $w$ with $i \xrightarrow{w} j$ and $\epsilon(w) = \Sigma(A)$, there must be some state $k$ that we enter after having read exactly the (non-tail spanning) portion of $w$ that corresponds to $\Sigma(B)$, i.e., $w = w_B w_C$, where $\epsilon(w_B) = \Sigma(B)$, $\epsilon(w_C) = \Sigma(C)$ and $i \xrightarrow{w_B} k \xrightarrow{w_C} j$. We also want to compute these intermediate states for every inner non-terminal $A \in N$ and $i, j \in [q]$. We now formally define these data structures and then show how to compute them efficiently.

**Definition 6.4.** For any non-terminal $A \in N$, we define a $(q \times q)$-matrix $R_A$ as follows. For every $i, j \in [q]$, let $R_A[i, j] = \perp$ if $M_A[i, j] = \emptyset$, let $R_A[i, j] = e$ if $M_A[i, j] = \emptyset$, and let $R_A[i, j] = I$ otherwise. For any inner non-terminal $A \in N$ with rule $A \rightarrow BC$, we define a $(q \times q)$-matrix $I_A$ as follows. For every $i, j \in [q]$, $I_A[i, j] = \{k : R_B[i, k] \neq \perp \text{ and } R_C[k, j] \neq \perp\}$.

The next lemma will be crucial for the precomputation phase of our algorithms for computing and enumerating $[M](D)$.

**Lemma 6.5.** All the matrices $R_A$ for every $A \in N$, $I_A$ for every inner non-terminal $A' \in N$, and $M_{T_{X'}}$ for every $x \in \Sigma$ can be computed in total time $O(q^3 |M| + \text{size}(S))$.

**Proof Sketch.** For computing all $M_{T_{X'}}$, with $x \in \Sigma$, it is helpful to observe that, for every $x \in \Sigma$ and every $i, j \in [q]$, we have $M_{T_{X'}}[i, j] = \{p(A, x) : A \in \mathcal{P}(\Gamma_X), i \xrightarrow{A} x \rightarrow j\}$. Consequently, we can compute the matrices $M_{T_{X'}}$ by iterating through all $i, j, k \in [q]$ and compute all $Y \in \mathcal{P}(\Gamma_X)$ and $x \in \Sigma$ such that $i \xrightarrow{k} \overrightarrow{x} j$ and then add $p(Y, x)$ to $M_{T_{X'}}[i, j]$. This can be done in time $O(q^3 |M|)$.

We now have all $M_{T_{X'}}$ with $x \in \Sigma$, and we can directly obtain $R_{T_{X'}}$ from $M_{T_{X'}}$ in time $O(|M| q^2)$. Finally, the matrices $R_A$ and $I_A$ for inner non-terminals with $A \rightarrow BC$ can be computed recursively in a bottom-up fashion using time $O(q^3 |N|)$.

The next lemma states how for inner non-terminals $A$ with rule $A \rightarrow BC$, and $i, j \in [q]$, the set $M_A[i, j]$ is composed from sets $M_B[i, k]$ and $M_C[k, j]$ with $k \in \Lambda_{i,j}$. For formulating the lemma, we need the following notation. For partial marker sets $\Delta, \Delta'$ and some $s \in \mathbb{N}$, let $\Delta \otimes_s \Delta' = \Delta \cup \\mathcal{R}_s(\Delta')$.

**Lemma 6.6.** Let $A \rightarrow BC$ be a rule of $S$, let $i, j \in [q]$ and let $\Lambda_{A}$ be a partial marker set. Then following are equivalent:

1. $\Lambda_{A} \subseteq M_A[i, j]$.
2. There are a $k \in \Lambda_{i,j}$ and partial marker sets $\Lambda_{B} \subseteq M_B[i,k]$ and $\Lambda_{C} \subseteq M_C[k,j]$, such that $\Lambda_{A} = \Lambda_{B} \otimes_{|M|} \Lambda_{C}$.

We extend the operator $\otimes$ to sets $\Delta, \Delta'$ of partial marker sets by $\Delta \otimes_s \Delta' = \{ (\delta \otimes_s \delta') : \Delta, \Delta' \in \Delta, \Delta' \in \Delta' \}$. With this terminology, we can now conclude from Lemma 6.6 that $M_A[i, j]$ actually decomposes into the $[I_A][i, j]$ (not necessarily disjoint) sets $\mathcal{R}_s(\Lambda_{i,j})$ with $k \in \Lambda_{i,j}$.

**Lemma 6.7.** For every inner non-terminal $A \in N$ with rule $A \rightarrow BC$, for every $i, j \in [q]$ and $k \in \Lambda_{i,j}$, we define $\mathcal{R}_s^{k}(\Lambda_{i,j}) = \mathcal{R}_s(\Lambda_{i,j}) \setminus \mathcal{R}_s^{k-1}(\Lambda_{i,j})$.

**Lemma 6.8.** Let $A \in N$ be an inner non-terminal and let $i, j \in [q]$.

Then $M_A[i, j] = \bigcup_{k \in \Lambda_{i,j}} \mathcal{R}_s^{k}(\Lambda_{i,j})$.

For $k, k' \in \Lambda_{i,j}$ with $k \neq k'$, $\mathcal{R}_s^{k}(\Lambda_{i,j}) \cap \mathcal{R}_s^{k'}(\Lambda_{i,j}) = \emptyset$ is possible. But for every fixed $k$, every element from $\mathcal{R}_s^{k}(\Lambda_{i,j})$ can only be obtained from elements of $M_B[i, k]$ and $M_C[k, j]$ in a unique way:

**Lemma 6.9.** Let $A \in N$ with rule $A \rightarrow BC$, let $i, j, k \in [q]$, let $\Lambda_{B}, \Lambda_{C}' \subseteq M_B[i,k]$ and $\Lambda_{C}, \Lambda_{C}' \subseteq M_C[k,j]$. Then $\Lambda_{B} \otimes_{|M|} \Lambda_{C} = \Lambda_{B}' \otimes_{|M|} \Lambda_{C}' \iff \Lambda_{B} = \Lambda_{B}'$ and $\Lambda_{C} = \Lambda_{C}'$.

7 COMPUTATION OF THE SOLUTION SET

We now consider the problem of computing the full set $[M](D)$. In contrast to non-emptiness and model-checking, this task, as well as enumerating $[M](D)$, are not decision problems anymore and, to the best of our knowledge, they do not reduce to any existing algorithm on SLP-compressed documents. By utilising the technical machinery of Section 6 we obtain this section’s main result.
8 ENUMERATION OF THE SOLUTION SET

In this section, we consider the problem of enumerating the set \([M](D)\). In the following, let \(S = (N, \Sigma, R, S_0)\) be an SLP for \(D\), and let \(M = (Q, \Sigma, 1, \delta, F)\) be an NFA with \(Q = \{1, 2, \ldots, q\}\) that represents a \((\Sigma, X)\)-spanner.

The matrices \(R_A\) (Definition 6.4) shall play an important role in the following. In particular, recall the meaning of the three possible entries \(\perp\) (\(M_A[i,j] = \emptyset\)), \(\perp\) (\(M_A[i,j] = \{q\}\)) and \(\perp\) (\(M_A[i,j] = \{q\}\)) after Lemma 6.3.

The high-level idea of our enumeration procedure will be to enumerate special trees (formally defined in Section 8.1) that uniquely represent a subset of the marker sets to be enumerated. Moreover, the marker set represented by such a tree can be retrieved in time proportional to the size of the tree, which can be shown to be bounded by the depth of the SLP. Consequently, the whole enumeration task reduces to enumerating those trees (and the delay is dominated by constructing the next tree, i.e., by its size).

We write \(\text{sort}(n)\) for the time it takes to sort a set of size \(O(n)\); depending on the underlying machine model this might be interpreted as \(O(n)\) or as \(O(n \log n)\).

**Theorem 7.1.** Let \(S\) be an SLP for \(D\) and let \(M\) be an NFA with \(q\) states that represents a \((\Sigma, X)\)-spanner. The set \([M](D)\) can be computed in time \(O(\text{sort}(|M|) \cdot q^7 + |M| \cdot \text{size}(\Sigma) \cdot q^7 \cdot \text{size}(\{M\}(D)))\).

**Proof.** We first perform the preprocessing described by Lemma 6.5. For any given \(A \in N\) and \(i, j \in [q]\), we can inductively compute \(M_A[i,j]\) as follows. If \(A = T_x\) is a leaf non-terminal, then we already have computed \(M_A[i,j]\); this serves as the basis of the induction. If \(A \to BC\) is a rule, then, according to Lemma 6.8, the set \(M_A[i,j]\) is given by \(\bigcup_k k \in \mathcal{L}_A[k,j] \mathcal{R}_A[k,j]\). Therefore, for every \(k \in \mathcal{I}_A[i,j]\), we compute the set \(\mathcal{R}_A[k,j]\). By Definition 6.7, \(\mathcal{R}_A[k,j] = \mathcal{M}_B[i,k] \otimes \mathcal{T}(B) \mathcal{M}_C[k,j]\). By induction, we can assume that the sets \(\mathcal{M}_B[i,k]\) and \(\mathcal{M}_C[k,j]\) have already been computed for every \(k \in \mathcal{I}_A[i,j]\). Finally, according to Lemma 6.3, \([M](D) = \bigcup_{j \in [q]} M_S[1,j]\), where \(F = \{j \in [q] : \mathcal{R}_S[1,j] \neq \perp\}\), so it is sufficient to recursively compute all \(M_{S}[1,j]\) with \(j \in F\). There are, however, two difficulties to be dealt with.

To avoid doing duplicates when constructing unions of sets of marker sets, we define an order on marker sets and handle all sets of marker sets as sorted lists according to this order. More precisely, we initially construct sorted lists of the sets \(M_{T_x}[i,j]\) for every \(x \in \Sigma\) and \(i, j \in [q]\) (which is responsible for the additive term \(\text{sort}(M)\) in the running time). Then, we can create sorted lists of unions of sets of marker sets by merging sorted lists and directly discarding the duplicates.

To obtain the claimed running time, we have to show that the computed intermediate sets \(M_A[i,j]\) cannot get larger than the final set \([M](D)\). In fact, this is not necessarily the case for every \(A \in N\) and \(i, j \in [q]\). However, if in the recursion we need to compute some set \(M_A[i,j]\), then for every \(A \in M_A[i,j]\) there is a subword-marked word \(v \in L(M)\) with \(e(D) = D\) and \(v = v_1 \in \mathcal{T}(A) \mathcal{L}(A)v_2\) such that \(v_1 = (i \rightarrow j \rightarrow k \rightarrow F)\). This directly implies that, if \(M_A[i,j]\) is computed in the recursion, then for each \(A \in M_A[i,j]\) there is a unique element in \([M](D)\). Thus \(|M_A[i,j]| \leq |[M](D)|\). □

8.1 \((M, S)\)-Trees

We define certain ordered binary trees with node- and arc-labels. All arc-labels will be non-negative integers, namely numbers 0 or \(\mathcal{T}(A)\) for \(A \in N\). The available node-labels are given as follows. For every \(A \in N\) and all \(i, j \in [q]\),

- if \(R_A[i,j] = \{\emptyset\}\), then there is a node-label \(A(i \cdot j, e)\).
- if \(R_A[i,j] = \{1\}\), then
  - if \(A\) is a leaf non-terminal, then there is a node-label \(A(i \cdot j, 1)\),
  - if \(A\) is an inner non-terminal, then for every \(k \in \mathcal{I}_A[i,j]\) there is a node-label \(A(k \cdot j, k)\).

For \((A, i, j)\) with \(R_A[i,j] = \perp\), we do not define any node-label(s). In an \((M, S)\)-tree, nodes labelled with \(A(i \cdot j, e)\) or \(A(i \cdot j, 1)\) are leaves. Each node \(v\) labelled with \(A(i \cdot k \cdot j)\) has a left child \(v_l\) and a right child \(v_r\). Let \(A \to BC\) be the rule for \(A\). Then the arc from \(v\) to \(v_l\) is labelled 0 and the arc from \(v\) to \(v_r\) is labelled \(\mathcal{T}(B)\). The node \(v_l\) is labelled as follows:

- if \(R_B[i,k] = \{\emptyset\}\), then \(v_l\) is labelled \(B(i \cdot k, e)\).
- if \(R_B[i,k] = \{1\}\), then
  - if \(B\) is a leaf non-terminal, then \(v_l\) is labelled \(B(i \cdot k, 1)\),
  - if \(B\) is an inner non-terminal, then \(v_l\) is labelled \(B(i \cdot k', k)\) for a \(k' \in I_B[i,k]\).
- \(R_B[i,k] = \perp\) cannot occur because we know that \(k \in \mathcal{I}_A[i,j]\).

The node \(v_r\) is labelled analogously:

- if \(R_C[k,j] = \{\emptyset\}\), then \(v_r\) is labelled \(C(k \cdot j, e)\).
- if \(R_C[k,j] = \{1\}\), then
  - if \(C\) is a leaf non-terminal, then \(v_r\) is labelled \(C(k \cdot j, 1)\),
  - if \(C\) is an inner non-terminal, then \(v_r\) is labelled \(C(k \cdot k', j)\) for a \(k' \in I_C[k,j]\).
- \(R_C[k,j] = \perp\) cannot occur because we know that \(k \in \mathcal{I}_A[i,j]\).

The idea underlying this notion is that a subtree rooted by \(A(i \cdot k \cdot j)\) represents some partial marker sets \(A \in M_A[i,j]\) that correspond to marked words that can be read via intermediate state \(k\), i.e., the subset \(\Delta_k \subseteq \Delta\) corresponding to \(T(B)\) is from \(M_B[i,k]\) and the subset \(\Delta_C \subseteq \Delta\) corresponding to \(T(C)\) is from \(M_C[k,j]\). Hence, at \(A(i \cdot k \cdot j)\), it can be interpreted as representing some elements of \(\mathcal{R}_A[k,j] \subseteq M_A[i,j]\). Moreover, all possible subtrees rooted by \(A(i \cdot k \cdot j)\) will represent the full set \(\mathcal{R}_A[k,j]\). Then, by Lemma 6.8, the set of all subtrees rooted by \(\mathcal{A}(i \cdot k_A \cdot j)\) for a \(k_A \in I_A[i,j]\) represents the complete set \(M_A[i,j]\).
In the case that $R_A[i,j] = e$, we know that $M_A[i,j] = \{\emptyset\}$, i.e., the empty set is the only partial marker set in $M_A[i,j]$. If $A$ is a leaf non-terminal $T_x$ with $R_{T_x}[i,j] = 1$, then the set $M_A[i,j]$ can be easily computed in a preprocessing step (see Lemma 6.5). Therefore, we treat these cases as leaves in our trees (i.e., as the base cases where the recursive branches represented by these trees terminate).

In this way, such a tree rooted by $A(i\oplus k \oplus j)$ for some $k \in I_A[i,j]$ is a concise representation of some runs of the recursive procedure implicitly given by Lemma 6.8, i.e.,

$$M_A[i,j] \supseteq A^+_{\emptyset}[i,j] \cup \{ M_B[i,k] \cup \{ M_C[k,j] \} \}$$

This also explains why we store the shift $|\Sigma(B)|$, which is necessary for the operation $\otimes_{\Sigma(B)}$, on the arc from a node $v$ labelled $A(i\oplus k \oplus j)$ to its right child $v_j$, labelled $C(k \oplus j)$.

For any $(M,S)$-tree $T$, we denote its leaves labelled by $T_x(i\oplus j,1)$ (for $x \in \Sigma$) as terminal-leaves and all the other leaves, i.e., leaves labelled by $A(i\oplus j,e)$, as empty-leaves. Note that leaves $A(i\oplus j,e)$ with $A = T_x$ are considered empty-leaves. Obviously, different nodes of $(M,S)$-trees can have the same label. As indicated before, the purpose of $(M,S)$-trees is to represent sets of partial marker sets. We shall now define this formally by first defining the yield of single $(M,S)$-trees. From here on, the following notation will be convenient. For trees $T_1,T_2$, arc-labels $s_1,s_2$, and a node-label $P$ we write $P((T_1,s_1),(T_2,s_2))$ to denote the tree whose root is labelled $P$ and has the roots of $T_1$ and $T_2$ as its left and right child, respectively, with arcs labelled by $s_1$ and $s_2$, respectively.

**Definition 8.1.** The yield of an $(M,S)$-tree $T$ is inductively defined as follows. If $T$ is a single node labelled $A(i\oplus j,e)$, then $\text{yield}(T) = \emptyset$. If $T$ is a single node labelled $T_x(i\oplus j,1)$, then $\text{yield}(T) = M_{T_x}[i,j]$. If $T = P((T_1,0),(T_2,s))$, then $\text{yield}(T) = \text{yield}(T_1) \otimes_{\Sigma} \text{yield}(T_2)$.

For every node $u$ of a fixed $(M,S)$-tree $T$, we shall denote by $\text{yield}_T(u)$ the yield of the subtree of $T$ rooted by $u$. An $(M,S)$-tree whose root node has a label including the non-terminal $A$ will sometimes be called $(M,A)$-tree.

**Example 8.2.** We recall the SLP $S$ from Example 4.2 for $D = \text{aabccaaabbaaAAA}$ and the DFA $M$ from Figure 3. It can be verified that the tree $T$ depicted in Figure 4 is an $(M,S_0)$-tree. As an example, note that according to the definition of $(M,S)$-trees the root can have a left child labelled by $A(1\oplus 0 \oplus 5)$, since $M$ can go from state 1 to state 5 by reading the marked word $\text{aab} \cc \text{cc}$ (corresponding to $\Sigma(A)$), while reading the prefix $\text{aab}$ (corresponding to $\Sigma(C)$) between state 1 and state 1, and reading the suffix $\text{cc}$ (corresponding to $\Sigma(D)$) between state 1 and state 5. Then, the node labelled by $C(1\oplus 1,e)$ is an empty-leaf, since $w = \Sigma(C) = \text{aab}$ is the only marked word with $e(w) = \Sigma(C)$ that can read going from state 1 to state 1.

The yield of all leaves of the $(M,S_0)$-tree depicted in Figure 4 is $\emptyset$, except for the terminal-leaves labelled by $T_x(1\oplus 5,1)$ and by $T_x(5\oplus 6,1)$, whose yields are $\text{yield}(T_x(1\oplus 5,1)) = \{(\emptyset^4,1)\}$ and $\text{yield}(T_x(5\oplus 6,1)) = \{(\emptyset^6,1)\}$. These yields are shown in Figure 4 below the corresponding leaves. By the recursive definition of yield(), we get $\text{yield}(A(1\oplus 0 \oplus 5)) = \{(\emptyset^4,4)\}$ and $\text{yield}(B(5\oplus 6 \oplus 6)) = \{(\emptyset^6,6)\}$. Since the arc from the root to the node labelled by $B(5\oplus 6 \oplus 6)$ is labelled by $5$, we get $\text{yield}(T) = \{(\emptyset^4,4), (\emptyset^6,6)\}$.

**Figure 4:** The $(M,S_0)$-tree discussed in Example 8.2.

Note that $\Lambda = \{(\Sigma^4,4),(\Sigma^6,6)\}$ corresponds to the $(x,y,D)$-tuple $t$ with $t(x) = \perp$ and $t(y) = \{4,6\}$, and $m(D,\Lambda) = \text{aabccaaabbaa\text{AAA}}$.

As an immediate consequence of Definition 8.1 we obtain:

**Lemma 8.3.** Let $T$ be an $(M,S)$-tree and let $u$ be a node of $T$ labelled $B(i\oplus k \oplus j)$, $B(i\oplus j,1)$ or $B(i\oplus j,1)$ for some $B \in N$, $i,j \in [q]$. Then every element from $\text{yield}_T(u)$ is a partial marker set over $X$ compatible with $\Sigma(B)$.

We measure the size of $|T|$ of a tree $T$ as the number of its nodes. Next, we estimate the size of $(M,A)$-trees. Recall that the depth of non-terminals has been defined in Section 4.

**Lemma 8.4.** Let $A \in N$ and let $T$ be an $(M,A)$-tree. Then $|T| \leq 4|X|\cdot \text{depth}(A)$, and $T$ has at most $2|X|$ terminal-leaves.

**Proof Sketch.** The following can be shown by induction. If the subtree rooted by an inner node $u$ contains $\ell$ terminal-leaves, then, since the yield of each terminal-leaf contains at least one non-empty partial marker set, there must be partial marker sets in $\text{yield}_T(u)$ with a size of at least $\ell$ (i.e., $\text{yield}_T(u)$ must contain a partial marker set that is constructed from $\ell$ many non-empty marker sets from the terminal-leaves). Since partial marker sets have size at most $2|X|$, this means that $T$ has at most $2|X|$ terminal-leaves.

Furthermore, all inner nodes and all terminal-leaves lie on paths (of length $\leq \text{depth}(A)$) from some terminal-leaf to the root. Thus, there are at most $2|X|\cdot \text{depth}(A)$ inner nodes and terminal-leaves. Moreover, each of these nodes can be adjacent to at most one empty-leaf, thus, the total number of nodes is at most $4|X|\cdot \text{depth}(A)$.

We next consider the algorithmic problem of enumerating the yield of a given $(M,A)$-tree. An $(M,A)$-tree with leaf-pointers is an $(M,A)$-tree where, additionally, every terminal-label by $T_x(i\oplus j,1)$ stores a pointer to the first element of a list that contains the elements of $M_{T_x}[i,j]$ (for all $x \in \Sigma, i,j \in [q]$). This enables us to obtain the following.

**Lemma 8.5.** Given an $(M,A)$-tree $T$ with leaf-pointers, the set $\text{yield}(T)$ can be enumerated with preprocessing $O(\text{depth}(A)|X|)$ and delay $O(|X|)$.

So far, we have established that $(M,A)$-trees represent partial marker sets, that they have moderate size and that their yield can be easily enumerated. However, we still need to show that the yields of
all \((M, A)\)-trees rooted by \(A(i \otimes k \otimes j)\) for some \(k \in I_A[i, j]\), represent the complete set \(M_A[i, j]\). Moreover, in order to reduce the problem of enumerating elements from \(M_A[i, j]\) to enumerating \((M, A)\)-trees, we have to establish some kind of one-to-one correspondence between \((M, A)\)-trees and partial marker sets from \(M_A[i, j]\). These issues will be settled next.

### 8.2 A Unique Representation by \((M, S)\)-Trees

For \(A \in N, i, j \in [q]\) and \(k \in I_A[i, j] \cup \{b\}\), we define the set 

\[
\text{Trees}(A, i, k, j) = \{ \text{single tree with a single node labelled } A(i \otimes j, e) \text{ if } R_A[i, j] = e \text{ and it contains a single tree with a single node labelled } A(i \otimes j, 1) \text{ if } R_A[i, j] \neq e \}.
\]

Lemma 8.3. The yield of any set of \((M, A)\)-trees is a set of partial marker sets. The next lemma can be concluded in a straightforward way from Definition 6.7 and Lemma 6.8.

Lemma 8.6. \(\text{yield(Trees}(A, i, k, j)) = \{ R_A[i, j] \} \) for all inner non-terminals \(A, all i, j \in [q] with R_A[i, j] \neq \psi, and all k \in I_A[i, j].\)

By Lemma 8.6, we can consider the \((M, A)\)-trees of Trees\((A, i, k, j)\) as a representation of \(R_A[i, j]\). Hence, \(\bigcup_{k \in I_A[i, j]} \text{Trees}(A, i, k, j)\) serves as a representation of \(M_A[i, j]\). We could thus enumerate the trees of \(\bigcup_{k \in I_A[i, j]} \text{Trees}(A, i, k, j)\) and, for each individual \((M, A)\)-tree, use Lemma 8.5 to enumerate its yield. However, in addition to the question of how to enumerate all these trees (which shall be taken care of later on), we also have to deal with the possibility that the yields of different \((M, A)\)-trees are not disjoint, which would lead to duplicates in the enumeration. With respect to this latter issue, we have already observed in Section 6, that \(R_A[i, j] \cap R_A[i, j] \neq \emptyset\) for some \(k, k' \in I_A[i, j]\) with \(k \neq k'\), is possible. However, if \(M\) is a DFA, the sets \(R_A[i, j]\) with \(k \in I_A[i, j]\) are in fact pairwise disjoint:

Lemma 8.7. Let \(A\) be a non-terminal, let \(A'\) be an inner non-terminal, let \(i, j, j' \in [q]\) with \(j \neq j'\), and let \(k, k' \in I_A[i, j]\) with \(k \neq k'\). If \(M\) is a DFA, then \(M_A[i, j] \cap M_A[i, j'] = \emptyset\) and \(R_A[i, j] \cap R_A[i, j'] = \emptyset\).

Using this lemma we can show that, as long as \(M\) is deterministic, the yields of different \((M, A)\)-trees are necessarily disjoint. We define equality of \((M, A)\)-trees \(T_1\) and \(T_2\), denoted by \(T_1 = T_2\), as follows. The roots are called corresponding if they have the same label; and any other node \(P_1\) of \(T_1\) corresponds to a node \(P_2\) of \(T_2\) if they have the same label and are both the left (or both the right) child of corresponding parent nodes. Now \(T_1 = T_2\) if and only if this correspondence is a bijection between the nodes of \(T_1\) and \(T_2\).

This means that non-equal \((M, A)\)-trees have either differently labelled roots or they are extensions of the same tree (i.e., the tree of all the corresponding nodes) and differ in the way that a leaf of this common tree (possibly the root) has differently labelled left children or differently labelled right children in \(T_1\) and \(T_2\), respectively. Note, however, that for different nodes \(P_1\) and \(P_2\) of non-equal \(T_1\) and \(T_2\) it is nevertheless possible that \(\text{yield}_{T_1}(P) \neq \text{yield}_{T_2}(P)\).

Lemma 8.8. Let \(A \in N\) be an inner non-terminal, let \(i, j, j' \in [q]\) with \(R_A[i, j] = R_A[i, j'] = 1\), let \(k_1 \in I_A[i, j]\) and \(k_2 \in I_A[i, j]\). Let \(T_1, T_2\) be non-equal \((M, A)\)-trees with roots labelled by \(A(i \otimes k_1 \otimes j)\) and \(A(i \otimes k_2 \otimes j')\). If \(M\) is a DFA, then \(\text{yield}_{T_1}(P) \not\supseteq \text{yield}_{T_2}(P)\).

Proof Sketch. For contradiction, assume \(\text{yield}_{T_1}(P) \not\supseteq \text{yield}_{T_2}(P)\). By Lemma 8.6, \(\text{yield}_{T_1} \subseteq R_A[i, j] \subseteq R_A[i, j] \not\supseteq \text{yield}_{T_2}\). Thus, Lemma 8.7 implies \(j_1 = j_2\) and \(k_1 = k_2\). This means that \(T_1\) and \(T_2\) have corresponding roots labelled by \(A(i \otimes k \otimes j)\), for \(j = j_1 = j_2\) and \(k = k_1 = k_2\). Let \(T\) be the tree of the nodes of \(T_1\) and \(T_2\) that are corresponding. For any node \(P\) of \(T\) with a left child \(L_1\) and a right child \(R_1\) in \(T_1\) and \(L_2\) and a right child \(R_2\) in \(T_2\), we can show that if \(\text{yield}_{T_1}(P) \setminus \text{yield}_{T_2}(P) \neq \emptyset\), then \(\text{yield}_{T_1}(L_1) \cap \text{yield}_{T_1}(R_1) \neq \emptyset\) and \(\text{yield}_{T_2}(L_1) \cap \text{yield}_{T_2}(R_2) \neq \emptyset\) (for this we use Lemmas 8.6 and 6.9).

Hence, since \(T_1 \neq T_2\), there must be some node of \(T\) with \(\text{yield}_{T_1}(P) \setminus \text{yield}_{T_2}(P) \neq \emptyset\), such that \(P\)'s left children \(L_1\) and \(L_2\) in \(T_1\) and \(T_2\), respectively, are not corresponding (or this is the case with respect to \(P\)’s right children, which can be handled analogously). By our above observation, \(\text{yield}_{T_1}(L_1) \cap \text{yield}_{T_1}(R_1) = \emptyset\), but \(L_1\) is labelled by \(B(i' \otimes k_{B,1} \otimes k')\), \(L_2\) is labelled by \(B(i' \otimes k_{B,2} \otimes k')\) with \(k_{B,1} = k_{B,2}\). Since \(\text{yield}_{T_1}(L_1) \subseteq R_B[k_{B,1}][i', k']\) and \(\text{yield}_{T_2}(L_2) \subseteq R_B[k_{B,2}][i', k']\), this means that \(R_B[k_{B,1}][i', k'] \cap R_B[k_{B,2}][i', k'] = \emptyset\), which is a contradiction to Lemma 8.7.

### 8.3 The Enumeration Algorithm

An enumeration algorithm \(A\) produces, on some input \(I\), an output sequence \((s_1, s_2, \ldots, s_n, \text{EOE})\), where EOE is the end-of-enumeration marker. We say that \(A\) on input \(I\) enumerates a set \(S\) if and only if the output sequence is \((s_1, s_2, \ldots, s_n, \text{EOE})\), \(|S| = n = S = \{s_1, s_2, \ldots, s_n\}\). The preprocesing time of \(A\) on input \(I\) is the time that elapses between starting \(A(I)\) and the output of the first element, and the delay is the time that elapsed between any two elements of the output sequence. The preprocessing time and the delay of \(A\) is the maximum preprocessing time and maximum delay, respectively, over all possible inputs (measured as function of the input size).

We present an enumeration algorithm EnumAll (given in Algorithm 1), that receives as input some \(A \in N, i, j \in [q]\) and \(k \in I_A[i, j]\). We treat recursive calls to EnumAll as sets of the elements of the output sequence, which allows to use for-loops to iterate through the output sequences returned by the recursive calls (see Lines 8 and 10). For this, we assume that any recursive call of EnumAll writes its output element in a buffer and then produces the next element only when it is requested by the for-loop. Consequently, the time used for starting the next iteration of the for-loop is bounded by the preprocessing time (if it is the first iteration) or the delay (for all other iterations) of the recursive call of EnumAll (this includes checking that there is no iteration left, since we can only check this by receiving EOE from the recursive call).
The algorithm requires the data-structures $R_A$ and $I_A$, which, for now, we assume to be at our disposal. We further assume that, for all $A \in N$ and all $i, j \in [q]$, we have the sets $I_A[i, j]$ at our disposal, which are defined as follows. If $A = T_x$ or $R_A[i, j] = e$ then $I_A[i, j] = \{b\}$ (here, the symbol $b$ serves as a marker for the “base case”), and $I_A[i, j] = I_{A[i, j]}$ otherwise.

**Algorithm 1: EnumAll($A, i, k, j$)**

1. Input: Non-terminal $A \in N$, $i, j \in [q], k \in I_A[i, j] \cup \{b\}$.
2. Output: A sequence of the trees in $\text{Trees}(A, i, k, j)$, followed by EOE.
3. if $k = b$ then
4.   if $R_A[i, j] = e$ then
5.     output $\leftarrow$ single node with label $A(i \odot j, e)$, output $\leftarrow$ EOE;
6.   else
7.     output $\leftarrow$ single node with label $A(i \odot j, 1)$, output $\leftarrow$ EOE;
8. else if $A$ is an inner non-terminal with $A \rightarrow BC$ then
9.   for $(k_B, k_C) \in (I_B[i, k] \times I_C[k, j])$ do
10.      for $T_B \in \text{EnumAll}(B, i, k_B, k)$ do
11.         if $T_B \neq \text{EOE}$ then
12.            for $T_C \in \text{EnumAll}(C, k, k_C, j)$ do
13.                if $T_C \neq \text{EOE}$ then
14.                   output $\leftarrow A(i \odot j \odot k)(T_B, T_C)$;
15.         output $\leftarrow$ EOE;
16.   output $\leftarrow$ EOE;

For $A \in N, i, j \in [q]$, and $k \in I_A[i, j] \cup \{b\}$ we let $\max(A, i, k, j)$ be the maximum number of nodes of a tree in $\text{Trees}(A, i, k, j)$.

**Lemma 8.9.** Whenever it receives as input an inner non-terminal $A$, states $i, j \in [q]$ such that $R_A[i, j] = 1$, and a $k \in I_A[i, j]$, the algorithm EnumAll($A, i, k, j$) enumerates the elements of the set $\text{Trees}(A, i, k, j)$ with preprocessing and delay $O(\max(A, i, k, j))$.

**Proof Sketch.** We first observe that if $R_A[i, j] = e$ or if $A$ is a leaf non-terminal, then EnumAll($A, i, b, j$) enumerates the set $\text{Trees}(A, i, b, j)$ with constant preprocessing and constant delay.

This can be used as the base of an induction to show that there is a constant $c$ such that for all inputs $A \in N, i, j \in [q], k \in I_A[i, j] \cup \{b\}$, such that $k = b$ or $R_A[i, j] = 1$, the algorithm EnumAll($A, i, k, j$) enumerates (without duplicates) the set $\text{Trees}(A, i, k, j)$ such that it takes time at most

1. $c \cdot \max(A, i, k, j)$ before the first output is created,
2. $2c \cdot \max(A, i, k, j)$ between any two consecutive output trees,
3. $c \cdot \max(A, i, k, j)$ between outputting the last tree and EOE.

Let $A \rightarrow BC$ be a rule. To see that EnumAll($A, i, k, j$) does in fact enumerate $\text{Trees}(A, i, k, j)$, we observe that, for all $(k_B, k_C) \in (I_B[i, k] \times I_C[k, j])$, all $T_B \in \text{Trees}(B, i, k_B, k)$ and all $T_C \in \text{Trees}(C, k, k_C, j)$, the algorithm will produce the tree with a root labelled by $A(i \odot j \odot k)$, and with the roots of $T_B$ and $T_C$ as left and right child, respectively. By definition of $(M, A)$-trees, the algorithm produces exactly all $(M, A)$-trees with a root labelled by $A(i \odot j \odot k)$. Note that duplicate output trees can neither be produced during the same iteration of the loop of Line 7 nor during the executions of different iterations of the loop of Line 7.

In order to prove the claimed runtime bounds, we assume as induction hypothesis that these bounds hold with respect to every $k_B \in I_B[i, k]$ and every $k_C \in I_C[k, j]$ (with $\max(A, i, k, j)$ replaced by $\max(B, i, k_B, k)$ and by $\max(C, k, k_C, j)$, respectively). Then we can show that the first element of $A(i \odot j \odot k \odot j)$ is produced in time $c \cdot \max(A, i, k, j)$. We also have to show that after having produced some (but not the last) element of $A(i \odot j \odot k \odot j)$, we only need time $2c \cdot \max(A, i, k, j)$ to produce the next element, and that after having produced the last element of $A(i \odot j \odot k \odot j)$, we need at most time $c \cdot \max(A, i, k, j)$ to produce EOE. There are 4 individual cases to consider (for convenience, we call the loops of Lines 7, 8 and 10 by states-loop, B-loop and C-loop, respectively): (1) we are not in the last iteration of the C-loop, (2) we are in the last iteration of the C-loop (but not the B-loop), (3) we are in the last iterations of the C-loop and the B-loop (but not the states-loop), (4) we are in the last iterations of the C-loop, the B-loop and the states-loop. By using our induction hypothesis, we can show that the first three cases yield in fact a delay of at most $2c \cdot \max(A, i, k, j)$, while the fourth case yields a delay of $c \cdot \max(A, i, k, j)$. We emphasise that for obtaining these bounds, it is absolutely vital that the delay for getting the first element and the element EOE is better than the delay between two consecutive elements.

**Theorem 8.10.** Let $S$ be an SLP for $D$ and let $M$ be a DFA with $q$ states that represents a $(\Sigma, X)$-spanner. The set $[M](D)$ can be enumerated with preprocessing time $O(q^3(|M| + \text{size}(S)))$ and delay $O(\text{depth}(S) \cdot |X|)$.

**Proof Sketch.** In the preprocessing phase, we compute all the matrices $R_A$ for every $A \in N$, $I_A$, for every inner non-terminal $A' \in N$, and $M_{x_A}$ for every $x \in \Sigma$. We also compute the set $F' = \{j \in F : R_{S_0}(1, j) \neq \perp\}$ and, for every $A \in N$ and for every $i, j \in [q]$, the sets $I_A[i, j]$. According to Lemma 6.5, all this can be done in time $O(q^3(|M| + \text{size}(S)))$. Next, we present an enumeration procedure that receives an $(M, A)$-tree $T$ as input.

**EnumSingleTree($T$):**

1. Add the correct leaf-pointers to $T$
2. Enumerate yield($T$) according to Lemma 8.5.

The following can be concluded from Lemmas 8.4 and 8.5.

Claim 1: The procedure EnumSingleTree($T$) enumerates yield($T$) with preprocessing time $O(\text{depth}(A) |X|)$ and delay $O(|X|)$.

For all $j \in F'$ and $k \in I_{S_0}[1, j]$, we use the enumeration procedure EnumSingleRoot($j, k$):

1. By calling EnumAll($S_0, 1, k, j$), we produce a sequence $(T_{1f}, T_{2f}, \ldots, T_{n_{lf}, k})$ of $(M, S_0)$-trees followed by EOE.
2. In this enumeration, whenever we receive $T_{\ell f}$ for some $\ell \in [n_{lf, k}]$, we carry out EnumSingleTree($T_{\ell f}$) and produce its output sequence as output.

Claim 2: The procedure EnumSingleRoot($j, k$) enumerates $R_{S_0}$ with preprocessing time and delay $O(\text{depth}(S_0) |X|)$.

This claim is mainly a consequence of Lemma 8.9 and Claim 1; but we also need Lemma 8.4 to bound the preprocessing time and delay, Lemma 8.6 to argue that exactly the set $R_{S_0}$ is enumerated, and Lemma 8.8 to show that the enumeration is without duplicates.

The complete enumeration phase now consists of performing EnumSingleRoot($j, k$) for every $j \in F'$ and every $k \in I_{S_0}[1, j]$. By Claims 1 and 2, and Lemmas 6.8, 6.3, 8.7, this enumeration produces the correct output within the claimed time bounds.
Note that we need $M$ to be a DFA to apply Lemma 8.8, i.e., to argue that the yields of different $(M, S_0)$-trees are disjoint. Observe that running the algorithm of Theorem 8.10 directly on an NFA yields a correct enumeration with the same complexity bounds, but with possible duplicates. But since we can transform NFAs into DFAs (at the cost of an exponential blow-up in automata size), Theorem 8.10, without producing duplicates, holds also for NFAs, but $|M|$ and $q$ in the preprocessing become $2^{|M|}$ and $2^q$. However, this affects only the preprocessing time, and it does not change the data complexity.

9 CONCLUSION

We showed that regular spanners can be efficiently evaluated directly on SLP-compressed documents. In the best-case scenario where the SLPs have a size logarithmic in the size $d$ of the uncompressed document, our approach solves all the considered evaluation tasks with only a logarithmic dependency on $d$. Our enumeration algorithm’s delay is $O(\log d)$, and the most important question left open is whether this can be improved to a constant delay — we believe this to be difficult.

In terms of combined complexity, it might be interesting to know whether fast Boolean matrix multiplication can lower the degree of the polynomial with respect to the number of states, as it is the case for checking membership of an SLP-compressed document in a regular language (see Section 4). Another intriguing question is whether spanner evaluation on compressed documents can handle updates of the document.

REFERENCES

[1] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann. 2017. Fine-Grained Complexity of Analyzing Compressed Data: Quantifying Improvements over Decompress-and-Solve. In Proc. FOCS’17. 192–203. Extended version available at http://arxiv.org/abs/1803.00796.

[2] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. 2019. Constant-Delay Enumeration for Nondeterministic Document Spanners. In Proc. ICDT’19.

[3] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. 2020. Constant-Delay Enumeration for Nondeterministic Document Spanners. SIGMOD Record 49, 1 (2020), 25–32. https://doi.org/10.1145/3422648.3422655

[4] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M.L. Schmid. 2020. On the Complexity of the Smallest Grammar Problem over Fixed Alphabets. Theory of Computing Systems (2020). https://doi.org/10.1007/s00224-020-10013-w

[5] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shielat. 2005. The smallest grammar problem. IEEE Transactions on Information Theory 51, 7 (2005), 2554–2576.

[6] Patrick Hagge Cording. 2015. Algorithms and data structures for grammar-compressed strings. Ph.D. Dissertation.

[7] J. Doleschal, B. Kimelfeld, W. Martens, Y. Naehouf, and F. Neven. 2019. Split-Completeness in Information Extraction. In Proc. PODS’19. 149–163.

[8] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. J. ACM 62, 2 (2015), 12:1–12:51.

[9] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. 2018. Constant-Delay Algorithms for Regular Document Spanners. In Proc. PODS’18.

[10] D. Freydenberger. 2019. A Logic for Document Spanners. Theory Comput. Syst. 63, 7 (2019), 1679–1754. https://doi.org/10.1007/s00224-018-9874-1

[11] D. Freydenberger and M. Hoddack. 2018. Document Spanners: From Expressive Power to Decision Problems. Theory Comput. Syst. 62, 4 (2018), 854–898.

[12] D. Freydenberger, B. Kimelfeld, and L. Peterfreund. 2018. Joining Extractions of Regular Expressions. In Proc. PODS’18. 137–149.

[13] M. Ganardi, A. Jez, and M. Lohrey. 2019. Balancing Straight-Line Programs. In Proc. FOCS’19. 1169–1183.

[14] K. Goto, S. Matsuyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda. 2011. Restructuring Compressed Texts without Explicit Decompression. CoRR abs/1107.2729 (2011). http://arxiv.org/abs/1107.2729

[15] J. C. Kieffer and E.-H. Yang. 2000. Grammar-based codes: A new class of universal lossless source codes. IEEE Trans. on Information Theory 46, 3 (2000), 737–754.

[16] F. Lehman. 2002. Approximation Algorithms for Grammar-Based Data Compression. Ph.D. Dissertation. Massachusetts Institute of Technology.

[17] M. Lohrey. 2012. Algorithms on SLP-compressed strings: A survey. Groups Complex. Cryptol. 4, 2 (2012), 241–299. https://doi.org/10.1515/gcc-2012-0016

[18] M. Lohrey. 2014. The Compressed Word Problem for Groups (Springer Briefs in Mathematics ed.). Springer.

[19] N. Markey and P. Schroelken. 2004. A PTIME-complete matching problem for SLP-compressed words. Inf. Process. Lett. 90, 1 (2004), 3–6.

[20] F. Maturana, C. Riveros, and D. Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In Proc. PODS’18.

[21] C. Nevill-Manning and I. Witten. 1997. Identifying Hierarchical Structure in Text. Ph.D. Dissertation. University of Waikato, NZ.

[22] L. Peterfreund. 2019. The Complexity of Relational Queries over Extractions from Text. Ph.D. Dissertation.

[23] L. Peterfreund. 2021. Grammars for Document Spanners. In Proc. ICDT’21. Extended version available at https://arxiv.org/abs/2003.06880.

[24] W. Plandowski and W. Rytt. 1999. Complexity of Language Recognition Problems for Compressed Words. In Jewels Are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa. 262–272.

[25] W. Rytt. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoret. Comput. Sci. 302, 1-3 (2003), 211–222.

[26] M.L. Schmid and N. Schweikardt. 2021. Spanner evaluation over SLP-Compressed strings. In Proc. PODS’21.

[27] M.L. Schmid and N. Schweikardt. 2021. A Purely Regular Approach to Non-Regural Core Spanners. In Proc. ICDT’21.

[28] M.L. Schmid and N. Schweikardt. 2021. Decompress-and-Solve. In Proc. FOCS’17.

[29] J. A. Storer and T. G. Szymanski. 1982. Data compression via textual substitution. J. ACM 29, 4 (1982), 928–951.