Regular Expression Matching on Compressed Text

Pierre Ganty[1] and Pedro Valero[2]

1 **IMDEA Software Institute, Spain**
`pierre.ganty@imdea.org`
2 **IMDEA Software Institute, Spain & Universidad Politécnica de Madrid, Spain**
`pedro.valero@imdea.org`

Abstract

Companies like Facebook, Google or Amazon store and process huge amounts of data. Lossless compression algorithms such as *brotli* and *zstd* are used to store textual data in a cost-effective way. On the other hand, it is common to process textual data using regular expression engines as evidenced by the number of highly-performant engines under development such as *Hyperscan* or *RE2*. For the scenario in which the input to the regular expression engine is only available in compressed form, the state of the art approach is to feed the output of the decompressor into the regular expression engine. We challenge this approach by searching directly on the compressed data. The experiments show that our purely sequential implementation outperforms the state of the art even when decompressor and search engine each have a distinct computing unit.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Straight Line Program, Grammar-based Compression, Regular Expression Matching

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.

1 Introduction

Big data demands efficient techniques both for storing and processing it. Over the last years, such demand has lead to the development of a variety of algorithms both for compression, to improve the storage cost, and for regular expression matching, to improve the performance of the processing. However these two problems have generally been considered independently and consequently the state of the art practice for searching with a regular expression in a compressed text is to feed the output of the decompressor into the search engine. We challenge this approach and propose to use the information about the text, present in its compressed version, to enhance the search.

Lossless compression of textual data is achieved by finding repetitions in the input text and replacing them by references. Each of these references points to either a sequence of text symbols or a sequence combining text symbols and previously defined references. We focus on grammar-based compression schemes in which each tuple "reference $\rightarrow$ repeated text" is considered as a rule of a context-free grammar. The resulting grammar, produced as the output of the compression, generates a language containing a single word: the uncompressed text. Figure 1 depicts the output of a grammar-based compression algorithm.

Our approach iterates thorough the grammar rules and computes for each of them how they update the state of the search. For instance, when processing rule $S_2 \rightarrow S_1\$$ of Fig. 1 our algorithm composes the update previously computed for S_1 with the update for \$. The resulting information will be reused every time the variable S_2 appears in the right hand side of a rule. We implement this approach in a tool called *zearch* which, given a grammar-compressed text and a regular expression, reports all lines in the uncompressed


© Pedro Valero and Pierre Ganty;
licensed under Creative Commons License CC-BY

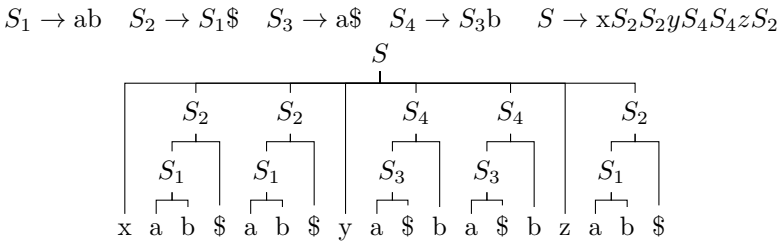

Figure 1 *List of grammar rules (on top) generating the string "$xab\$ab\$ya\$ba\$bzab\$$" (and no other) as evidenced by the parse tree (bottom).*

text containing a substring that matches the expression. Our experiments show that *zearch*, in its current sequential form, outperforms the state of the art even when decompression and search are done in parallel.

The rest of the paper is organized as follows: Section 2 introduces some basic notions required to prove, in Section 3, the correctness of the algorithms used by *zearch*; Section 4 describes the implementation of our tool; Section 5 discusses the experiments carried out and the obtained results and Section 6 analyzes the complexity of *zearch*. Finally, Section 7 describes some related works before concluding with Section 8.

2 Preliminaries

An *alphabet* Σ is a nonempty finite set of *symbols*. A *string* w is a finite sequence of symbols of Σ where the empty sequence is denoted ε. A *language* is a set of strings and the set of all strings over Σ is denoted Σ^*. We denote by $|w|$ the *length* of w and denote it by $\dagger$ when w is clear from the context. Further define $(w)_i$ as the i-th symbol of w if $1 \leq i \leq \dagger$ and ε otherwise. Similarly, $(w)_{i,j}$ denotes the substring, also called *factor*, of w between the i-th and the j-th symbols, both inclusive.

A *finite state automaton* (FSA or automaton for short) is a tuple $A = (Q, \Sigma, q_0, F, \delta)$ where Q is a finite set of *states* including the *initial state* q_0; Σ is an alphabet; $F \subseteq Q$ are the *final states*; and $\delta \subseteq Q \times \Sigma \times Q$ are the *transitions*. Note that with this definition ε-transitions are not allowed. A *configuration* of an FSA is a pair (v, q) where $v \in \Sigma^*$ represents the rest of the input and $q \in Q$, the current state. Define the *step* relation $\to$ over configurations given by $(v, q) \to (v', q')$ if $v = a\,v'$ for some $a \in \Sigma$ and $(q, a, q') \in \delta$. Given $(v, q), (v', q')$ such that $(v, q) \to^* (v', q')$ (the reflexive-transitive closure of $\to$), there exists a witnessing sequence (possibly empty) of steps between (v, q) and (v', q') called a *run*. A *run* $(v, q) \to^* (\varepsilon, q')$ is usually denoted $q \xrightarrow{v} q'$. The *language* of A, denoted $\mathcal{L}(A)$, is the set $\{w \mid q_0 \xrightarrow{w} q, q \in F\}$. A string w *matches* against the automaton, and we say it is a *match*, if $w \in \mathcal{L}(A)$. A language L is said to be *regular*, if there exists an FSA A such that $L = \mathcal{L}(A)$.

A *context-free grammar* (or grammar for short) is a tuple $G = (V, \Sigma, S, \mathcal{R})$ where V is a finite set of *variables* (or *non-terminals*) including the *start variable* S; Σ is an alphabet (or set of *terminals*), $\mathcal{R} \subseteq V \times (\Sigma \cup V)^*$ is a finite set of *rules*. Given a rule $(X, \xi) \in \mathcal{R}$, we often write it as $X \to \xi$ for convenience. Define the *step* relation $\Rightarrow$ as a binary relation over $(V \cup \Sigma)^*$ given by $\rho \Rightarrow \sigma$ if there exists a rule $X \to \xi$ of G and a position i such that $(\rho)_i = X$ and $\sigma = (\rho)_{1,i-1}\xi(\rho)_{i+1,\dagger}$. Given ρ, σ such that $\rho \Rightarrow^* \sigma$ (the reflexo-transitive closure of $\Rightarrow$), there exists a witnessing sequence (possibly empty) of steps between ρ and σ. Incidentally when $\rho \in V$ and $\sigma \in \Sigma^*$ such a step sequence is called a *derivation* (of variable ρ). The *language* of G, denoted $\mathcal{L}(G)$, is the set $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$. A language L is said to be *context-free*, or *CFL*, if there exists a grammar G such that $L = \mathcal{L}(G)$.

A *Straight Line Program* $P = (V, \Sigma, \mathcal{R})$, hereafter SLP, is a grammar $G = (V, \Sigma, S, \mathcal{R})$ such that $\mathcal{R}$ is an indexed set of $|V|$ rules (i.e. $\mathcal{R} = (r_i)_{i \in 1,\ldots,|V|}$) where each indexed rule has the form $r_i = (X_i, \alpha_i\beta_i)$. That is, variables are also indexed, there is exactly one rule per variable and their index coincide. Furthermore $r_i = (X_i, \alpha_i\beta_i)$, usually denoted as $X_i \to \alpha_i$, satisfies $\alpha_i, \beta_i \in (\Sigma \cup \{X_j \in V \mid j < i\})$ and $S = X_{|V|}$. It is worth pointing that variables appearing in the right hand side of r_i are left hand side of rule with a lower index. We define the *size* of an SLP P, denoted $|P|$, as the number of rules, $|\mathcal{R}|$. It is easy to see that the language generated by an SLP consists of a single word $w \in \Sigma^*$ and, by definition, $|w| > 1$. When $\mathcal{L}(P) = \{w\}$ we abuse of notation and identify w with $\mathcal{L}(P)$.

An *Extended Straight Line Program*, hereafter ESLP, is an SLP $(V, \Sigma, \mathcal{R})$ in which the axiom rule $(X_{|V|} \to \alpha) \in \mathcal{R}$ has $|\alpha| \geq 2$. We define the size of an ESLP P as $|P| = |\mathcal{R}| + |\alpha|$.

3 Regular Expression Matching on Grammar-compressed Texts

Esparza et al. [4] defined the *saturation construction* as an algorithm to solve a number of decision problems involving automata and context-free grammars. In particular, given an automaton built from a regular expression and an SLP produced by a grammar-based compressor, the *saturation construction* can be used to decide whether or not the uncompressed text matches against the expression.

We consider this algorithm as a starting point for searching with regular expressions in grammar-compressed texts. By restricting the input grammar to SLPs, the *saturation construction* can be simplified since all rules have the same format and are listed in an orderly manner so that rule $X \to \alpha\beta$ is always processed after α and β. As a result we obtain Algorithm 0 which returns ***True*** iff the string generated by the SLP belongs to the language generated by the automaton, i.e. there exists a run $q_0 \xrightarrow{w} q_f$ with $\mathcal{L}(P) = \{w\}$ and $q_f \in F$.

Algorithm 0 Saturation construction for SLPs

Input: An SLP $P = (V, \Sigma, \mathcal{R})$ and an FSA $A = (Q, \Sigma, q_0, F, \delta)$.

Output: $\mathcal{L}(P) \stackrel{?}{\in} \mathcal{L}(A)$

```
1: function MAIN
2:   for each ℓ = 1, 2, ..., |V| do
3:     let (X_ℓ → α_ℓ β_ℓ) ∈ R
4:     for each q_1, q' ∈ Q s.t. (q_1, α_ℓ, q') ∈ δ do
5:       for each q_2 ∈ Q s.t. (q', β_ℓ, q_2) ∈ δ do
6:         δ := δ ∪ {(q_1, X_ℓ, q_2)}
7:   return ((({q_0} × {X_|V|} × F) ∩ δ) ≠ ∅)
```

3.1 Finding all variables that generate a string containing a match

Standard search tools go beyond Algorithm 0 and check the existence of factors in w that belong to the language generated by the automaton. This is equivalent to answering the question $\mathcal{L}(P) \stackrel{?}{\in} \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$ which can be done by adding $\{(q, a, q) \mid q \in (\{q_0\} \cup F), a \in \Sigma\}$ to the set of transitions of the automaton before applying Algorithm 0.

However, the presence of these extra transitions in δ carries no relevant information and forces the algorithm to add $\{(q, X, q) \mid q \in (\{q_0\} \cup F)\}$ to δ, which is also meaningless, for each variable of the grammar. We improve this situation by developing Algorithm 1, which assumes the existence of all these transitions without adding any of them.

Algorithm 1 Extension of Algorithm 0 to find factor matching the expression.

Input: An SLP $P = (V, \Sigma, \mathcal{R})$ and an FSA $A = (Q, \Sigma, q_0, F, \delta)$.

Output: $\mathcal{L}(P) \stackrel{?}{\in} \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$

1: **function** MAIN
2: **for each** $\ell = 1, 2, \ldots, |V|$ **do**
3: **let** $(X_\ell \to \alpha_\ell \beta_\ell) \in \mathcal{R}$
4: **for each** $q_1, q' \in Q$ s.t. $(q_1, \alpha_\ell, q') \in \delta$ or $q_1{=}q'{=}q_0$ **do**
5: **for each** $q_2 \in Q$ s.t. $(q', \beta_\ell, q_2) \in \delta$ or $q'{=}q_2 \in F$ **do**
6: $\delta := \delta \cup \{(q_1, X_\ell, q_2)\}$
7: **return** $\left(\left(\{q_0\} \times \{X_{|V|}\} \times F\right) \cap \delta\right) \neq \emptyset$

The idea behind the loops in lines 4 and 5 of Algorithm 1 is to assume the existence of the sets of transitions $\{(q_0, X, q_0) \mid X \in V\}$ and $\{(q_f, X, q_f) \mid X \in V, q_f \in F\}$, respectively. The following definitions allow us to prove the correctness of Algorithm 1 in Theorem 5.

▶ **Definition 1** (Extended label)**.** Let $A = (Q, \Sigma, q_0, F, \delta)$ be an FSA, $w \in \Sigma^+$ and $q_1, q_2 \in Q$. We say w is an *extended label* of a path from q_1 to q_2, and write $q_1 \overset{w}{\rightsquigarrow} q_2$, iff there exists a run $q_1 \xrightarrow{(w)_{i,j}} q_2$ with $1 \leq i \leq j \leq |w|$ and $\left(i = 1 \vee q_1 = q_0\right) \wedge \left(j = |w| \vee q_2 \in F\right)$. We refer to factor $(w)_{i,j}$ as the *witness* of the extended label.

▶ **Definition 2** (Minimal witness)**.** Let $A = (Q, \Sigma, q_0, F, \delta)$ be an FSA, $w \in \Sigma^+$ and $q_1, q_2 \in Q$. Let $q_1 \overset{w}{\rightsquigarrow} q_2$ with witness $w' \in \Sigma^+$. We say witness w' is a *minimal witness* if no factor of w' is a witness of $q_1 \overset{w}{\rightsquigarrow} q_2$.

Intuitively, $q_1 \overset{w}{\rightsquigarrow} q_2$ indicates that if transitions $\{(q, a, q) \mid q \in (\{q_0\} \cup F), a \in \Sigma\}$ were added to δ then there would be a run $q_1 \xrightarrow{w} q_2$. Lemmas 3 and 4 rely on this concept to describe the set of transitions that Algorithm 1 adds to the automaton.

▶ **Lemma 3.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, q_0, F, \delta)$ an FSA with $q_0 \notin F$. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$ and $q_1, q_2 \in Q$. After ℓ iterations of loop in line 2 of Algorithm 1, if $X \in \{X_1, \ldots, X_\ell\} \subseteq V$ then*

$$(q_1, X, q_2) \in \delta \implies q_1 \overset{w}{\rightsquigarrow} q_2 \ .$$

Proof. See Appendix A.1. ◀

▶ **Lemma 4.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, q_0, F, \delta)$ an FSA with $q_0 \notin F$. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$ and $q_1, q_2 \in Q$. After ℓ iterations of loop in line 2 of Algorithm 1, if $X \in \{X_1, \ldots, X_\ell\} \subseteq V$ then*

$$q_1 \overset{w}{\rightsquigarrow} q_2 \implies (q_1, X, q_2) \in \delta \ .$$

Proof. See Appendix A.2. ◀

▶ **Theorem 5.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, q_0, F, \delta)$ an FSA with $q_0 \notin F$. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$ and $q_f \in F$. After ℓ iterations of loop in line 2 of Algorithm 1, if $X \in \{X_1, \ldots, X_\ell\} \subseteq V$ then*

$$(q_0, X, q_f) \in \delta \iff w \in \Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^* \ .$$

Proof. Assume $(q_0, X, q_f) \in \delta$ with $q_f \in F$. Then by Lemma 3, $q_0 \overset{w}{\rightsquigarrow} q_f$ and from Definition 1 it follows $w \in \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$.

Now assume $w \in \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$. Then there is a factor w' such that $q_0 \xrightarrow{w'} q_f$ with $q_f \in F$ so, by Definition 1, $q_0 \overset{w}{\rightsquigarrow} q_f$. By Lemma 4, (q_0, X, q_f) is added to δ by Algorithm 1. ◀

▶ **Corollary 6.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, q_0, F, \delta)$ an FSA with $q_0 \notin F$. After applying Algorithm 1, it returns **True** iff $\mathcal{L}(P) \in \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$.*

3.2 Counting Matching Lines

State of the art tools for regex search are equipped with a number of features[1] that allow them to perform different operations beyond deciding the existence of a match in the text. Among the most relevant of these features we find *counting*. Tools like *grep*[2], *ripgrep*[3], *sift*[4] of *ack*[5] report the number of lines containing a match, ignoring matches across lines.

As shown in the previous section, Algorithm 1 finds all variables of the SLP that generate a factor of the uncompressed text containing a match. This information, together with the set of rules of the SLP and some extra data about the end-of-line delimiters generated by each variable, can be used to implement the counting feature. Indeed, Algorithm 2 extends Algorithm 1 to report the number of lines in the uncompressed text containing a match of the expression. As the tools mentioned before, we do not consider matches across lines.

Next we prove the correctness of Algorithm 2 using $\hookleftarrow$ to denote the end-of-line delimiter, $\widehat{\Sigma}{=}\Sigma\backslash\{\hookleftarrow\}$ for any alphabet Σ and defining a *line* as a factor of a string delimited by $\hookleftarrow$ symbols or the ends of the string.

▶ **Definition 7** (Matching lines)**.** Let $P{=}(V, \Sigma, \mathcal{R})$ be an SLP with $\tau \in (V \cup \Sigma)$, $w \in \Sigma^+$ and $\tau{\Rightarrow^*}w$. Let $A{=}(Q, \widehat{\Sigma}, q_0, F, \delta)$ an FSA. The *matching lines generated by* τ is the set

$$\mathcal{ML}_\tau{=}\left\{(i,j) \mid (w)_{i,j} \in \widehat{\mathcal{L}}(A), \left(i{=}1 \vee (w)_{i-1}{=}\hookleftarrow\right), \left(j{=}|w| \vee (w)_{j+1}{=}\hookleftarrow\right)\right\}$$

where $\widehat{\mathcal{L}}(A){=}\widehat{\Sigma}^*{\cdot}\mathcal{L}(A){\cdot}\widehat{\Sigma}^*$.

In order to count the number of matching lines generated by an SLP with axiom $X_{|V|}$, it suffices to compute $|\mathcal{ML}_{X_{|V|}}|$. A naive approach would consist on computing $\mathcal{ML}_X$ for each symbol of the grammar, which, given rule $X \to \alpha\beta$, can be done by combining $\mathcal{ML}_\alpha$, $\mathcal{ML}_\beta$ and the information implicitly generated by Algorithm 1, as shown by Lemmas 8 and 9.

▶ **Lemma 8.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ an FSA. Let $u, v, w \in \Sigma^+$ and $X \Rightarrow \alpha\beta \Rightarrow^* u{\cdot}\beta \Rightarrow^* u{\cdot}v = w$.*

1. $\forall\, 1 \le i \le j < |u|,\ \left((i,j) \in \mathcal{ML}_\alpha \iff (i,j) \in \mathcal{ML}_X\right)$.
2. $\forall\, 1 < i \le j \le |v|,\ \left((i,j) \in \mathcal{ML}_\beta \iff (i{+}|u|, j{+}|u|) \in \mathcal{ML}_X\right)$.

Proof. See Appendix A.3. ◀

▶ **Lemma 9.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ an FSA with $q_0 \notin F$. Let $w{,}u{,}v \in \Sigma^+$ and $X \Rightarrow \alpha\beta \Rightarrow^* u{\cdot}\beta \Rightarrow^* u{\cdot}v = w$. Then there exists $(\ell_X, r_X) \in \mathcal{ML}_X$ such that $\ell_X \le |u| < r_X$* if and only if *one of the following holds:*

[1] https://beyondgrep.com/feature-comparison/
[2] https://www.gnu.org/software/grep
[3] https://github.com/BurntSushi/ripgrep
[4] https://github.com/svent/sift
[5] https://github.com/beyondgrep/ack2

1. $(\ell_X, |u|) \in \mathcal{ML}_\alpha$.
2. $(1, r_X - |u|) \in \mathcal{ML}_\beta$.
3. *Algorithm 1 adds* $(q_0, \alpha, q'), (q', \beta, q_f)$ *to* δ *with* $q' \notin (\{q_0\} \cup F)$ *and* $q_f \in F$.

Furthermore, when the pair (ℓ_X, r_X) *exists it is unique and*

$$\ell_X = \max\{0, i \mid (u)_i = \hookleftarrow\} + 1 \text{ and } r_X = |u| + \min\{|v| + 1, i \mid (v)_i = \hookleftarrow\} - 1 \ .$$

Proof. See Appendix A.4. ◀

Since all pairs $(i, j) \in \mathcal{ML}_X$ with $(w)_{i-1} = (w)_{j+1} = \hookleftarrow$ belong to every set $\mathcal{ML}_Y$ such that $Y \Rightarrow^* \sigma X \rho$ with $\sigma, \rho \in (\Sigma \cup V)^*$, storing sets $\mathcal{ML}_X$ for each variable of the grammar causes a lot of redundancies. We improve this situation by defining the *counting information* of a variable, represented as $\mathcal{C}_X$. Intuitively, $\mathcal{C}_X$ is an abstraction of $\mathcal{ML}_X$ which has constant size and allow us to a) compute n_X from $\mathcal{C}_X$ and b) compute $\mathcal{C}_X$ from $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$ for a given rule $X \to \alpha\beta$.

▶ **Definition 10** (Counting information)**.** Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ an FSA. Let $\tau \in (V \cup \Sigma)$ with $\tau \Rightarrow^* w$ and

$$\leftarrow\!\!|_\tau := |\{(i,j) \in \mathcal{ML}_\tau \mid i = 1\}| \in \{0,1\} \qquad |\!\!\rightarrow_\tau := |\{(i,j) \in \mathcal{ML}_\tau \mid j = |w|\}| \in \{0,1\}$$
$$\hookleftarrow_\tau := \min\{1, |\{k \mid (w)_k = \hookleftarrow\}|\} \in \{0,1\} \qquad \sharp_\tau := |\{(i,j) \in \mathcal{ML}_\tau \mid 1 < i \leq j < |w|\}|$$

We define the *counting information of* X as $\mathcal{C}_X = \{\hookleftarrow_X, \leftarrow\!\!|_X, |\!\!\rightarrow_X, \sharp_X\}$.

▶ Remark 11. Although the set $\mathcal{ML}_X$ itself cannot be recovered from $\mathcal{C}_X$ its size can be computed as the result of $\sharp_X + \leftarrow\!\!|_X + |\!\!\rightarrow_X$ if $\hookleftarrow_X = 1$ and $\leftarrow\!\!|_X + |\!\!\rightarrow_X$ otherwise.

Algorithm 1 can be extended to compute the counting information of each variable obtaining Algorithm 2. Note that Algorithm 2 preserves the functionality of Algorithm 1, whose legacy is highlighted in blue, and therefore Lemmas 3, 4, 8 and 9 as well as Theorem 5 remain valid when applied to Algorithm 2. Next we prove that the output of the algorithm corresponds with the number of matching lines generated by the axiom, i.e. the number of lines in the uncompressed text containing a match of the expression.

▶ **Lemma 12.** *Let* $P = (V, \Sigma, \mathcal{R})$ *be an SLP and* $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ *an FSA. For each rule* $(X \to \alpha\beta) \in \mathcal{R}$ *let* $m = 1$ *if* $(q_0, \alpha, q')(q', \beta, q_f) \in \delta$ *with* $q_f \in F$, $q' \notin (\{q_0\} \cup F)$ *and* $m = 0$ *otherwise. The following equalities hold:*

1. $\hookleftarrow_X = \min\{1, \hookleftarrow_\alpha + \hookleftarrow_\beta\}$
2. $\leftarrow\!\!|_X = \begin{cases} \max\{\leftarrow\!\!|_\alpha, \leftarrow\!\!|_\beta, m\} & \text{if } \hookleftarrow_\alpha = 0 \\ \leftarrow\!\!|_\alpha & \text{otherwise} \end{cases}$; $|\!\!\rightarrow_X = \begin{cases} \max\{|\!\!\rightarrow_\alpha, |\!\!\rightarrow_\beta, m\} & \text{if } \hookleftarrow_\beta = 0 \\ |\!\!\rightarrow_\beta & \text{otherwise} \end{cases}$
3. $\sharp_X = \begin{cases} \sharp_\alpha + \sharp_\beta & \text{if } \hookleftarrow_\alpha = 0 \vee \hookleftarrow_\beta = 0 \\ \sharp_\alpha + \sharp_\beta + \max\{|\!\!\rightarrow_\alpha, \leftarrow\!\!|_\beta, m\} & \text{otherwise} \end{cases}$

Proof. See Appendix A. ◀

▶ **Theorem 13.** *Let* $P = (V, \Sigma, \mathcal{R})$ *be an SLP and* $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ *an FSA. After applying Algorithm 2,* $\mathcal{C}_X$ *has been computed for each* $X \in V$.

Proof. Note that definitions 7 and 10 applied to terminals $a \in \Sigma$ result in $\mathcal{ML}_a$ either empty or containing a single pair $(1, 1)$ and:

$$\leftarrow\!\!|_a = |\!\!\rightarrow_a = \begin{cases} 1 & \text{if } (q_0, a, q_f) \in \delta,\ q_f \in F \\ 0 & \text{otherwise} \end{cases} \qquad \hookleftarrow_a = \begin{cases} 1 & \text{if } a = \hookleftarrow \\ 0 & \text{otherwise} \end{cases} \ .$$

Since no transition labeled with a terminal is added by Algorithm 2, all values $\mathcal{C}_a$ with $a \in \Sigma$ are computed by function INIT_AUTOMATON.

It is straightforward to observe that Algorithm 2 implements both the operations just described to initialize the terminals and the ones from Lemma 12 to propagate the counting information from the symbols on the right hand side of a rule to the one on the left. Therefore after Algorithm 2 has processed the axiom rule, according to Remark 11:

$$|\mathcal{ML}_{X_{|V|}}| = \begin{cases} \leftarrow\!\!|_{X_{|V|}} + \mapsto_{X_{|V|}} & \text{if } \hookleftarrow_{X_{|V|}} = 1 \\ \sharp_{X_{|V|}} + \leftarrow\!\!|_{X_{|V|}} + \mapsto_{X_{|V|}} & \text{otherwise} \end{cases} .$$

which coincides with the value output by Algorithm 2. ◀

4 Implementation

In this section, we describe an implementation based on Algorithm 2 to perform regular expression matching on grammar compressed text: *zearch*. This tool works on texts compressed with *repair*[6], which implements the Recursive Pairing algorithm defined by Larsson et al. [8]. The choice of a particular compressor simplifies the implementation task since *zearch* has to read the grammar rules from the compressed data. However *zearch* can handle any grammar based scheme by modifying the way in which rules are read.

The regular expression received as input is translated into an ε-free FSA by using Thompson's algorithm [12] with on-the-fly ε-removal. This operation is done by using the automata library *libfa*[7].

4.1 Data structures

A straightforward implementation of Algorithm 2 uses the data structure suggested by Esparza et al. [4], which consists on a bit array where each position represents a possible transition (q_i, X_ℓ, q_j) and the value thereof indicates whether the transition is present in the automaton. This data structure offers constant time membership test and addition to the set of transitions using $\mathcal{O}(ps^2)$ space, where p and s are the sizes of the SLP and the automaton, respectively. However, this representation of the automaton forces loop in line 18 of Algorithm 2 to perform s^2 iterations since it has to consider all pairs of states and check whether they are connected by a transition labeled with α. Similarly, loop in line 19 requires s iterations regardless of the number of transitions in the automaton. As a result Algorithm 2 operates in $\mathcal{O}(ps^3)$.

The current implementation of *zearch* improves this situation by keeping a list of labeled transitions for each symbol of the grammar. The counting information of each symbol, together with a pointer to the list of labeled transitions, is stored in an array where the i-th position contains the information related to the i-th symbol of the grammar. This representation of the automaton allow loops in lines 18 and 19 of Algorithm 2 to iterate directly over the transitions labeled by α and β respectively. However, to have constant time addition to the list of transitions labeled by a variable while avoiding repetitions, *zearch* requires an auxiliary matrix with s^2 elements of size $\log(p)$ bits. When transition (q_i, X_ℓ, q_j) is added to δ *zearch* first checks if position (i, j) of the matrix contains the value ℓ. If not,

[6] `https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/re-pair/repair110811.tar.gz`
[7] `http://augeas.net/libfa/index.html`

then it adds the pair (q_i, q_j) to the list of transitions labeled by X_ℓ and stores ℓ in the matrix at position (i, j). Note that the matrix can be reused for all rules since they are processed one at a time in an orderly manner. Finally, each variable of the grammar might label up to s^2 transitions in the automaton, so Algorithm 2 exhibits a worst case $\mathcal{O}(ps^4)$ time complexity using $\mathcal{O}\left(ps^2\right)$ space. On the other hand, the lists of transitions labeled by each variable of the grammar could be empty, resulting in a best case $\mathcal{O}(p)$ time complexity using $\mathcal{O}\left(p+s\log p\right)$ space. Section 6 provides a more detailed analysis of the complexity of Algorithm 2.

4.2 Processing the axiom rule

Grammar-based compression algorithms as *Recursive Pairing* [8], *LZ78* [15] or *LZW* [14] produce extended straight line programs. This happens because the axiom rule is built with the remains of the input text after extracting all repeated factors and replacing them by variables of the grammar. Once the algorithm stops there is no repeated factor in the remaining text, otherwise it would result in a new rule. The remaining text is then folded into an axiom rule. As shown by column *axiom length* of Table 2 there are typically more symbols in the axiom than rules in the grammar so the way in which the axiom is handled has a great impact in the performance.

An ESLPs can be transformed into an SLP by rewriting the axiom rule $X_{|V|}{\to}\sigma$ as $P_a{=}\{S_0{\to}(\sigma)_0(\sigma)_1\} \cup \{S_i{\to}S_{i-1}(\sigma)_{i+1} \mid i = 1\ldots|\sigma|{-}2\} \cup \{X_{|V|}{\to}S_{|\sigma|-2}(\sigma)_\dagger\}$ so, in theory, Algorithm 2 could process the axiom rule by translating it into an SLP. It is worth pointing that the transitions labelled S_i can be discarded after processing the grammar rule with S_{i+1} on the left hand side. Hence we can reuse the array entry for S_i when processing S_{i+1} and so on. Similarly, since for any S_i there is exactly one derivation $X_{|V|} \Rightarrow^* \alpha S_i \beta$ and $\alpha = \varepsilon$, it suffices to store the transitions labeled by S_i leaving from the initial state. The resulting algorithm, defined as Algorithm 3, exhibits a worst case complexity of $\mathcal{O}\left((p - |\sigma|)s^4 + |\sigma|s^2\right)$ where p is the size of the ESLP and $X_{|V|}{\to}\sigma$ its axiom.

4.3 Reporting matching lines

Algorithm 3 always processes rule $X{\to}\alpha\beta$ after rules for α and β so the matching lines generated by each of them, and therefore generated also by X, have already been reported. Therefore it suffices to report the new matching line generated by variable X, which corresponds to $(\ell_X, r_X) \in \mathcal{ML}_X$ as explained in Lemma 9.

To report the new matching line generated by X we compute the derivation from X using the counting information to guide the process and prune symbols occurring before or after the new matching line. For instance, let $X{\to}\alpha\beta$ with X generating a new matching line and $\alpha \to \sigma\rho$ with $\hookleftarrow_\rho = 1$. Then the string generated by symbol σ appears before the new matching line so the derivation is $X \Rightarrow \alpha\beta \Rightarrow \sigma\rho\beta \Rightarrow \rho\beta \Rightarrow \ldots$.

Note that the above process does not print all matching lines of the uncompressed text. However, it comes with the following guarantee: ignoring duplicated lines in the output of *zearch* and ignoring duplicated matching lines in the uncompressed text, we find that the two sets of lines coincide.

5 Results

5.1 Design of the experiments

In this section we compare the performance[8] of *zearch* against decompressing the file with *zstd*[9] and searching on its output, as it is generated, with *grep* or *ripgrep* (*rg*). These tools are top of the class for lossless compression[10] and regular expression matching[11], respectively. Therefore they are a representative implementation of the state of the art approach for searching with regular expressions in compressed texts.

The benchmark designed for this comparison combines ideas from those for regular expression engines and compression tools. As such, we consider different sizes and formats for the input files since both factors have an impact in the result of the compression. In particular our benchmark includes files containing English subtitles, JSON and CSV formatted data and a log file.

Furthermore, we have crafted some experiments to highlight the strengths of our approach: a file where all the lines are the same and a file with a single line on a binary alphabet on which we search with expressions of the form "`[0-1]*1[0-1]{n}2`". These experiments correspond to rows "Contrived text" and "Contrived regex", respectively, in Table 1.

Finally, note that our implementation is purely sequential while the state of the art allows for a trivial parallelization by having distinct processes for the decompressor and the regular expression engine. Table 1 summarizes the obtained results, grouped by type of file and size of the uncompressed text. The details of these experiments, including the list of regular expressions considered for each file and the running times for each expression as well as the number of matches reported, can be found on *zearch*'s page[12].

File	Uncompressed size 1MB					Uncompressed size 100MB				
	zearch	*zstd+grep*		*zstd+rg*		*zearch*	*zstd+grep*		*zstd+rg*	
Subtitles	**6**	8	7	12	10	**354**	459	380	678	563
JSON	**4**	6	6	9	7	**177**	279	252	385	301
CSV	9	9	**8**	12	10	625	529	**487**	739	611
Log	**4**	7	6	11	9	**161**	316	244	546	431
Contrived regex	**17**	T.O.	T.O.	300	298	**1266**	T.O.	T.O.	2605	2553
Contrived text	**2**	8	7	12	10	**2**	446	404	683	590

Table 1 *Each tool is fed with data compressed in the unique format it accepts. The uncompressed data is 1MB (left) or 100MB (right). Each cell contains the time (in milliseconds) obtained as follows: compute the average running times of the last 10 runs (out of 11) for computing the number of matching lines for a given regular expression; then compute the average of those running times obtained for different regular expressions. Tools with two columns have the left column report the time when both processes share a single core and the right when they don't. The assignment of processes to core is done using* `taskset`*. T.O. means no result returned after 60,000ms.*

[8] We run our experiments in a an Intel Xeon E5640 CPU 2.67 GHz with 20 GB RAM
[9] `https://github.com/facebook/zstd`
[10] `https://quixdb.github.io/squash-benchmark/`
[11] `https://rust-leipzig.github.io/regex/2017/03/28/comparison-of-regex-engines/`
[12] `https://pevalme.github.io/zearch/graphs/index.html`

5.2 Analysis of the results

Table 1 shows that *zearch*, in most cases, outperforms the state of the art approach. Therefore we consider these results a strong evidence of the benefits of using the information about repetitions, computed by the compressor, during the search.

Note that the performance of *zearch* (uncompressed size / time) increases with the compression ratio (uncompressed size / compressed size). Indeed, considering the experiments on files with uncompressed size 100MB and sorting the four rows according to compressed size we obtain Log file (7 MB), JSON (9 MB), English subtitles (14 MB) and CSV (27 MB); and average running time Log file (161ms), JSON (177ms), English subtitles (354ms) and CSV (625ms). This relation is to be expected since *zearch* processes exactly once each rule of the grammar that compresses the text. Better compression ratio means less rules for the same uncompressed size which plays in favor of *zearch*.

The "Contrived text" experiment best illustrates this phenomenon. A 100 MB file where all lines are the same is compressed down to 52 bytes, producing a grammar with 50 rules which is processed in no time by *zearch*. On the contrary, the state of the art approach has to decompress the file and search through 100 MB of repeated lines.

Finally, in contrast to *grep* and *rg*, *zearch* computes on the non-deterministic automaton obtained from the regular expression without ever attempting to determinize it even partially. Because of determinization, *grep* and *rg* can face scalability issues under some symptomatic inputs. This phenomenon is illustrated by the "Contrived regex" experiment which is based on a regular expression of the form "`[0-1]*1[0-1]{n}2`", for which no deterministic automaton has less than 2^n states. *zearch* is the fastest for this experiment while *grep* suffers a timeout (probably caused by determinization), and *ripgrep*, to the best of our knowledge, falls back onto simulating the non-deterministic automata giving up on determinizing it. The data used in that experiment is a random string of 0's and 1's. Note that for this experiments the 100 MB long input is compressed to 18 MB.

We conclude our reporting by bringing to the reader's attention the following statistics: upon termination of the algorithm we very often find that the ratio between the number of grammar rules and the number of transitions in the resulting automaton is approximately 1. Formally $|\delta|/p \simeq 1$. We claim that a ratio of 1 corresponds to a runtime of the algorithm linear in p, which coincide with the performance observed in the experiments.

6 Complexity Analysis

As described in Section 4 Algorithm 3 exhibits $\mathcal{O}\left((p-|\sigma|)s^4+|\sigma|s^2\right)$ worst case complexity using $\mathcal{O}(ps^2)$ space where p is the size of the ESLP and $X_{|V|}\to\sigma$ is the axiom rule. However the experiments show that *zearch*'s runtime typically does not match that upper bound.

The $\mathcal{O}\left((p-|\sigma|)s^4+|\sigma|s^2\right)$ worst case time complexity corresponds to the scenario in which each variable of the grammar generates a string that labels s^2 paths, one for each pair of states. In such scenario each variable labels s^2 transitions which, with the data structure described in Section 4, results in Algorithm 3 iterating through $s^2 \times s^2$ transitions for each rule in the grammar. We refine this bound by defining the maximum number of paths in the automaton that can be labeled by a single string: $\pi = \max_{w\in\Sigma^*} |\{(q_1,q_2) \mid q_1 \overset{w}{\rightsquigarrow} q_2\}|$. Since each variable of the grammar generates a single string then it labels, at most, π transitions in the automaton. Hence the list of transitions labeled by any variable of the grammar can be iterated in less than π steps, reducing the worst case time complexity of Algorithm 3 to $\mathcal{O}\left((p-|\sigma|)\pi^2+|\sigma|\pi\right)$. Although π is bounded by s^2 we observe in our experiments that it tends to be bounded $c\cdot s$ for a small value of c.

Furthermore, the $\mathcal{O}((p-|\sigma|)\pi^2+|\sigma|\pi)$ complexity can be refined by distinguishing among the different types of rules. For instance, a rule $X \to \alpha\beta$ in which β (or α) labels no edge in the automaton is processed in $\mathcal{O}(\pi)$ since Algorithm 3 has to iterate through a single list of transitions. Therefore, the complexity of Algorithm 3 is given by $\mathcal{O}\left(r_0 + r_1{\cdot}\pi + r_2{\cdot}\pi^2 + |\sigma|\pi\right)$, where r_0, r_1 and r_2 are the number of rules $X \to \alpha\beta$ such that neither α nor β, only one of them and both of them, respectively, label any edge in the automaton. Note that $r_0{+}r_1{+}r_2$ equals the number of rules of the grammar minus one, because every rule (but the axiom) falls into exactly one of the three cases. Table 2 shows the values of these parameters in the experiments reported on Table 1.

File	Uncompressed size 1MB					Uncompressed size 100MB				
	$\lvert\sigma\rvert$	$r_0{+}r_1{+}r_2$	r_0	r_1	r_2	$\lvert\sigma\rvert$	$r_0{+}r_1{+}r_2$	r_0	r_1	r_2
Subtitles	34	32	13	9	10	2646	2367	913	668	786
JSON	35	27	18	5	4	1929	1428	1059	277	92
CSV	142	24	15	3	6	9518	803	500	142	161
Log	28	12	4	3	5	2345	455	147	169	139
Con. regex	90	9	0.7	0.01	8.29	6694	499	39	15	445
Con. text	0.002	0.2	0.01	0	0.19	0.03	0.27	0.13	0	0.14

Table 2 *Statistical data corresponding to experiments in Table 1. Values given in thousands. The number of rules of each kind as well as the number of transitions added is the average of the values computed for each regular expression used to search in the file.*

We conclude this section with an insight on the relation between the complexities of Algorithm 3 and the state of the art approach for regular expression matching on compressed text. To simplify the comparison we ignore the decompression step for the state of the art approach and assume the grammar received by Algorithm 3 is an SLP.

Thomson [12] developed an algorithm for regular expression matching on plain text that operates in $\mathcal{O}(ns)$, where n is the size of the text and s is the number of states of the non-deterministic automaton. This algorithm, as well as Algorithm 3, consists on iterating through a list of "units" (characters for Thompson's algorithm and grammar rules for Algorithm 3) and performing some operations for each of them. However, the operations carried out for each grammar rule are computationally more complex than the ones done for each character: $\mathcal{O}(s^4)$ against $\mathcal{O}(s)$ time complexity, respectively. On the other hand, the lists iterated by each of these algorithms are not equally long. According to Lehman [9], any text of length n over an alphabet Σ can be compressed into a grammar with $p = \mathcal{O}(n/\log_{|\Sigma|} n)$ rules. Moreover, for highly redundant inputs grammar-based compression can achieve $p = \log_{|\Sigma|} n$. Thus, the worst case complexity of our algorithm lies between $\mathcal{O}(s^4 \log_{|\Sigma|} n)$ and $\mathcal{O}(s^4 n/\log_{|\Sigma|} n)$, depending on the compression achieved.

7 Related Work

The pattern matching problem on compressed text was first formally defined by Amir et al. [1] as the problem of finding all occurrences of a given pattern in a compressed text without decompressing it. An algorithm solving this problem is said to be optimal when it runs in $\mathcal{O}(N+m)$ time: the runtime of the algorithm that decompresses and searches, where m is the length of the pattern and N the length of the uncompressed text. Later Amir et al. [2] defined an algorithm that finds the first, but not all, occurrence of the

pattern in the compressed text in $\mathcal{O}(n + m^2)$ space and time, where n is the size of the compressed text. Since then, several algorithms were developed to perform pattern matching on grammar compressed text [6, 5, 7]. To the best of our knowledge and to our surprise, none of these was implemented. The only existing implementation to perform pattern matching on grammar compressed text is due to Navarro et al. [11] who modified the Boyer-Moore algorithm to search on LZ78/LZW compressed text. We omitted this implementation from our experiments since its performance is not competitive with the state of the art tools searching on the uncompressed text as it is generated by the decompressor.

The first algorithm for regular expression matching on compressed text is due to Navarro [10] who achieved complexities of $\mathcal{O}(2^s + s \cdot N + \text{occ} \cdot s \log s)$ and $\mathcal{O}(s^2 + (n + \text{occ} \cdot s) \log s)$ for worst and average cases respectively, where "occ" is the number of occurrences of the expression in the uncompressed text. Later, Bille et al. [3] found a relation between the time and space required to perform regular expression matching on compressed text, which improved the previous bounds. Indeed Bille defined a data structure of size $o(n)$ to represent LZ78 compressed texts and an algorithm that, given a parameter τ, finds all occurrences of a pattern in a LZ78/LZW compressed file in $\mathcal{O}(n \cdot s(s + \tau) + \text{occ} \cdot s \cdot \log s)$ time and $\mathcal{O}(n \cdot s^2/\tau + ns)$ space. To the best of our knowledge, no implementation of this algorithm was carried out. Finally, remark that the algorithms of Navarro and Bille report and count the positions of the uncompressed text at which a match ends, not the matching lines.

8 Conclusions and Future Work

We have shown that the performance of regular expression matching on compressed text can be improved by working directly on the compressed file rather than decompressing it and searching on the output as it is generated. Furthermore we provide a tool, *zearch*, that outperforms the state of the art even when decompression and search are done in parallel.

It is yet to be considered how the performance of *zearch* is affected by the choice of the grammar-based compression algorithm. By using different heuristics to build the grammar, the resulting SLP will have different properties, such as depth, width or length of the axiom rule, which affects *zearch*'s performance. Furthermore, there are grammar-based compression algorithms such as *sequitur*[13] that produce SLPs in which rules might have more than two symbols on the right hand side. It is worth considering whether adapting *zearch* to work on such SLPs will have a positive impact on its performance.

On the other hand, the current implementation is purely sequential. However, the algorithms involved allow for a conceptually simple parallelization since any set of rules such that the sets of symbols on the left and on the right hand side are disjoint can be processed simultaneously. Indeed, a theoretical result by Ullman et al. [13] on the parallelization of Datalog queries can be interpreted in our setting to show that the regular expression search on grammar-compressed text is in $\mathcal{NC}^2$ when the automaton built from the expression is acyclic. Therefore it is worth considering the development of a parallel version of *zearch*.

References

1 Amihood Amir and C Benson. Efficient two-dimensional compressed matching. In *Data Compression Conference, 1992. DCC'92.*, pages 279–288. IEEE, 1992.

[13] `http://www.sequitur.info/`

2 Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

3 Philip Bille, Rolf Fagerberg, and Inge Li Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms (TALG)*, 6(1):3, 2009.

4 Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. In *EATCS BULLETIN*. Citeseer, 2000.

5 Martin Farach and Mikkel Thorup. String matching in lempel—ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

6 Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for lempel-ziv encoding. *Algorithm Theory—SWAT'96*, pages 392–403, 1996.

7 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

8 N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

9 Eric Lehman. *Approximation algorithms for grammar-based data compression*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2002. URL: `http://hdl.handle.net/1721.1/87172`.

10 Gonzalo Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 1(5):423–443, 2003.

11 Gonzalo Navarro and Jorma Tarhio. Lzgrep: a boyer–moore string matching tool for ziv–lempel compressed text. *Software: Practice and Experience*, 35(12):1107–1130, 2005.

12 Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

13 Jeffrey D Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3(1-4):5–42, 1988.

14 Terry A. Welch. A technique for high-performance data compression. *Computer*, 6(17):8–19, 1984.

15 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

A Deferred Proofs

A.1 Proof of Lemma 3

Base case: $\ell = 1$.

By definition of SLP, $X_1 \to ab$ with $ab \in \Sigma$. Let $w = ab$ so $X_1 \Rightarrow^* w$. Since $q_0 \notin F$, there are three possible scenarios for which transition (q_1, X_1, q_2) is added:

- $(q_1, a, q'), (q', b, q_2) \in \delta$. Then $q_1 \xrightarrow{ab} q_2$ so ab is a witness of $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q' = q_2 \in F$ and $(q_1, a, q') \in \delta$. Then $q_1 \xrightarrow{a} q'$ so a is a witness of $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q_1 = q' = q_0$ and $(q', b, q_2) \in \delta$. Then $q' \xrightarrow{b} q_2$ so b is a witness of $q_1 \overset{w}{\rightsquigarrow} q_2$.

Inductive step: Assume the lemma holds for some $\ell < |\mathcal{R}|$ and consider rule $X_{\ell+1} \to \alpha\beta$ with $X_{\ell+1} \Rightarrow^* w$ for some $w \in \Sigma^*$. By definition of SLP $\alpha, \beta \in (\Sigma \cup \{X_1, X_2, \ldots, X_\ell\})$ and $X \Rightarrow \alpha\beta \Rightarrow^* u\beta \Rightarrow^* uv = w$. Again, there are three possible scenarios for which transition $(q_1, X_{\ell+1}, q_2)$ is added:

- $(q_1, \alpha, q'), (q', \beta, q_2) \in \delta$. By hypothesis, $q_1 \overset{u}{\rightsquigarrow} q'$ and $q' \overset{v}{\rightsquigarrow} q_2$ so, by Definition 1, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q' = q_2 \in F$ and $(q_1, \alpha, q') \in \delta$. By hypothesis, $q_1 \overset{u}{\rightsquigarrow} q_2$ so, by Definition 1, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q_1 = q' = q_0$ and $(q', \beta, q_2) \in \delta$. By hypothesis, $q_1 \overset{v}{\rightsquigarrow} q_2$ so, by Definition 1, $q_1 \overset{w}{\rightsquigarrow} q_2$. ◀

A.2 Proof of Lemma 4

Base case: $\ell = 1$.

By definition of SLP, $X_1 \to ab$ with $a, b \in \Sigma$ and, since $q_0 \notin F$, there are three possibilities for $w' \in \Sigma^+$, the *minimal witness* of $q_1 \overset{ab}{\rightsquigarrow} q_2$:

- $w' = ab$. Then $(q_1, a, q_3), (q_3, b, q_2) \in \delta$ so (q_1, X_1, q_2) is added to δ with $q' = q_3$.
- $w' = a$. Then $(q_1, a, q_2) \in \delta$ and $q_2 \in F$ so (q_1, X_1, q_2) is added to δ with $q' = q_2 \in F$.
- $w' = b$. Then $(q_1, b, q_2) \in \delta$ and $q_1 = q_0$ so (q_1, X_1, q_2) is added to δ with $q_1 = q' = q_0$.

Inductive step: Assume the lemma holds for some $\ell < |\mathcal{R}|$ and consider rule $X_{\ell+1} \to \alpha\beta$ with $X_{\ell+1} \Rightarrow^* w$ for some $w \in \Sigma^*$. By definition of SLP $\alpha, \beta \in (\Sigma \cup \{X_1, X_2, \ldots, X_\ell\})$ and $\alpha \Rightarrow^* u$, $\beta \Rightarrow^* v$ with $w = u{\cdot}v$. Again, there are three possibilities for $w' \in \Sigma^+$, the *minimal witness* of $q_1 \overset{w}{\rightsquigarrow} q_2$.

- w' is the concatenation of a nonempty suffix of u and a nonempty prefix of v. By Definition 1 $q_1 \overset{u}{\rightsquigarrow} q_3$ and $q_3 \overset{v}{\rightsquigarrow} q_2$ for some $q_3 \in Q$. By hypothesis $(q_1, \alpha, q_3), (q_3, \beta, q_2) \in \delta$ and therefore $(q_1, X_{\ell+1}, q_2)$ is added to δ when $q'{=}q_3$.
- w' is a factor of u. By Definition 1 $q_2 \in F$ and $q_1 \overset{u}{\rightsquigarrow} q_2$. By hypothesis $(q_1, \alpha, q_2) \in \delta$ so transition $(q_1, X_{\ell+1}, q_2)$ is added to δ when $q'{=}q_2$.
- w' is a factor of v. Definition 1 $q_1{=}q_0$ and $q_1 \overset{v}{\rightsquigarrow} q_2$. By hypothesis $(q_1, \beta, q_2) \in \delta$ so transition $(q_1, X_{\ell+1}, q_2)$ is added to δ with $q_1{=}q'$. ◀

A.3 Proof of Lemma 8

1. It follows from Definition 7 since $(w)_{i,j} = (u)_{i,j}$ for all $1 \le i \le j < |u|$.
2. It follows from Definition 7 since $(w)_{i+|u|,j+|u|} = (v)_{i,j}$ for all $1 < i \le j \le |v|$. ◀

A.4 Proof of Lemma 9

Note that $(w)_i = (u)_i$ for $1 \leq i \leq |u|$ and $(w)_{i+|u|}{=}(v)_i$ for $1 \leq i \leq |v|$. Therefore, it follows from Definition 7 that if there exists $(i,j) \in \mathcal{ML}_X$ with $i \leq |u| < j$ then $(i,j) = (\ell_X, r_X)$. Next we prove that if any of the properties holds then $(\ell_X, r_X) \in \mathcal{ML}_X$.

1. $(\ell_X, |u|) \in \mathcal{ML}_\alpha$ means that $(w)_{\ell_X,|u|} \in \widehat{\mathcal{L}}(A)$. Since $(w)_{\ell_X,|u|}$ is a factor of $(w)_{\ell_X,r_X} \in \widehat{\Sigma}^+$ it follows that $(w)_{\ell_X,r_X} \in \widehat{\mathcal{L}}(A)$ and, therefore, $(\ell_X, r_X) \in \mathcal{ML}_X$.
2. $(1, r_X{-}|u|) \in \mathcal{ML}_\beta$ means that $(w)_{1+|u|,r_X} \in \widehat{\mathcal{L}}(A)$. Since $(w)_{1+|u|,r}$ is a factor of $(w)_{\ell_X,r_X} \in \widehat{\Sigma}^+$ it follows that $(w)_{\ell_X,r_X} \in \widehat{\mathcal{L}}(A)$ and, therefore, $(\ell_X, r_X) \in \mathcal{ML}_X$.
3. By Lemma 3 $q_0 \overset{u}{\rightsquigarrow} q'$ and $q' \overset{v}{\rightsquigarrow} q_f$ with *witnesses* u' and v' respectively. Since $\hookleftarrow \notin \widehat{\Sigma}$ and $q' \notin (\{q_0\} \cup F)$ then $u'{\cdot}v'$ is a factor of $(w)_{\ell_X,r_X} \in \widehat{\mathcal{L}}(A)$ and, therefore, $(\ell_X, r_X) \in \mathcal{ML}_X$.

We conclude by proving the other side of the implication: if $(\ell_X, r_X) \in \mathcal{ML}_X$ then at least one of the properties *a)-c)* holds. By Definition 7, $(\ell_X, r_X) \in \mathcal{ML}_X$ means $(w)_{\ell_X,r_X} \in \widehat{\mathcal{L}}(A)$ and, therefore, $q_0 \overset{w}{\rightsquigarrow} q_f$ with $q_f \in F$. There are three possibilities for w', the *minimal witness* of $q_0 \overset{w}{\rightsquigarrow} q_f$:

- w' is a factor of u. Then $q_0 \overset{u}{\rightsquigarrow} q_f$ and, therefore $(w)_{\ell_X,|u|} \in \widehat{\mathcal{L}}(A)$ so property *a)* holds.
- w' is a factor of v. Then $q_0 \overset{v}{\rightsquigarrow} q_f$ and, therefore $(w)_{|u|,r_X} \in \widehat{\mathcal{L}}(A)$ so property *b)* holds.
- Otherwise w' is the concatenation of a nonempty suffix of u and a nonempty prefix of v. Then $q_0 \overset{u}{\rightsquigarrow} q'$, $q' \overset{v}{\rightsquigarrow} q_f$ for some $q' \in (Q \setminus (\{q_0\} \cup F))$ and, by Lemma 4, property *c)* holds. ◀

A.5 Proof of Lemma 12

Let $X \Rightarrow^* w$, $\alpha \Rightarrow^* u$, $\beta \Rightarrow^* v$. Next we show that, for each of the variables in $\mathcal{C}_X$, the value given by Definition 10 coincides with the one computed using the formulas given by the lemma.

1. $\hookleftarrow_X = \min\{1, |\{k \mid (w)_k = \hookleftarrow\}|\} = \min\{1, |\{k \mid (u)_k = \hookleftarrow\}| + |\{k \mid (v)_k = \hookleftarrow\}|\}$. Therefore $\hookleftarrow_X = \min\left\{1, \hookleftarrow_\alpha + \hookleftarrow_\beta\right\}$.
2. We provide the proof for $\leftarrowtail_X$ but not for $\rightarrowtail_X$ since they are conceptually identical. $\leftarrowtail_X = |\{(i,j) \in \mathcal{ML}_X \mid i = 1\}|$. If $(1,j) \in \mathcal{ML}_X$ with $j < |u|$ then $\hookleftarrow_\alpha = 1$ and, by Lemma 8, $(1,j) \in \mathcal{ML}_X$ iff $(1,j) \in \mathcal{ML}_\alpha$, i.e. $\leftarrowtail_\alpha = 1$. Otherwise $j = r_X$, $\hookleftarrow_\alpha = 0$ and, by Lemma 9, $(1, r_X) \in \mathcal{ML}_X$ iff one of the following holds:
   - **a.** $(1, |u|) \in \mathcal{ML}_\alpha$ i.e. $\leftarrowtail_\alpha = 1$.
   - **b.** $(1, r_X{-}|u|) \in \mathcal{ML}_\beta$ i.e. $\leftarrowtail_\beta = 1$.
   - **c.** Algorithm 2 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to δ with $q' \notin (\{q_0\} \cup F)$ and $q_f \in F$ i.e. function COUNT is invoked with $m{=}1$.

   Therefore $\leftarrowtail_X = \begin{cases} \max\{\leftarrowtail_\alpha, \leftarrowtail_\beta, m\} & \text{if } \hookleftarrow_\alpha = 0 \\ \leftarrowtail_\alpha & \text{otherwise} \end{cases}$
3. $\sharp_X = |\{(i,j) \in \mathcal{ML}_X \mid 1{<}i{\leq}j{<}|w|\}|$. Let $\mathcal{ML}_{\mathcal{C}_X} = \{(i,j) \in \mathcal{ML}_X \mid 1{<}i{\leq}j{<}|w|\}$. Then $\mathcal{ML}_{\mathcal{C}_X} = \{(i,j) {\in} \mathcal{ML}_{\mathcal{C}_X} \mid i,j{<}|u|\} \bigcup \{(i,j) {\in} \mathcal{ML}_{\mathcal{C}_X} \mid |u|{<}i,j\} \bigcup \{(i,j) {\in} \mathcal{ML}_{\mathcal{C}_X} \mid i{\leq}|u|{\leq}j\}$. From Lemma 8 and Definition 10 it follows $\sharp_X = \sharp_\alpha + \sharp_\beta + |\{(i,j) {\in} \mathcal{ML}_{\mathcal{C}_X} \mid i{\leq}|u|{<}j\}|$. Finally, $\{(i,j) {\in} \mathcal{ML}_{\mathcal{C}_X} \mid i{\leq}|u|{\leq}j\} \neq \emptyset$ iff $(\ell_X, r_X) \in \mathcal{ML}_X$ with $\ell_X \neq 1$ and $r_X \neq |w|$ which, by Lemma 9, occurs iff one of the following holds and $\hookleftarrow_\alpha {=} \hookleftarrow_\beta {=} 1$:
   - **a.** $(\ell_X, |u|) \in \mathcal{ML}_\alpha$ i.e. $\leftarrowtail_\alpha = 1$.
   - **b.** $(\ell_X, r_X - |v|) \in \mathcal{ML}_\beta$ i.e. $\rightarrowtail_\beta = 1$.
   - **c.** Algorithm 2 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to δ with $q' \notin (\{q_0\} \cup F)$ and $q_f \in F$ i.e. function COUNT is invoked with $m{=}1$.

Therefore $\sharp_X = \begin{cases} \sharp_\alpha + \sharp_\beta & \text{if } \hookleftarrow_\alpha = 0 \vee \hookleftarrow_\beta = 0 \\ \sharp_\alpha + \sharp_\beta + \max\{\mapsto_\alpha, \leftarrow\!\!\!\shortmid_\beta, m\} & \text{otherwise} \end{cases}$ ◀

B Algorithms

Algorithm 2 Extension of Algorithm 1 to perform the counting

Input: An SLP $P = (V, \Sigma, \mathcal{R})$ and an FSA $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$
Output: Number of lines of $\mathcal{L}(P)$ containing a factor matching A.

procedure COUNT($\mathcal{C}_\alpha$, $\mathcal{C}_\beta$, m)
  $\hookleftarrow_X := \min\{1, \hookleftarrow_\alpha + \hookleftarrow_\beta\}$
  if $\hookleftarrow_\alpha = 0 \wedge \hookleftarrow_\beta = 0$ **then** $\begin{cases} \leftarrow\!\!\!\shortmid_X := 1, \ \mapsto_X := 1 \end{cases}$
  else if $\hookleftarrow_\alpha = 0 \wedge \hookleftarrow_\beta \neq 0$ **then** $\begin{cases} \mapsto_X := \mapsto_\beta, \ \sharp_X := \sharp_\beta \\ \leftarrow\!\!\!\shortmid_X := \max\{\leftarrow\!\!\!\shortmid_\alpha, \leftarrow\!\!\!\shortmid_\beta, m\} \end{cases}$
  else if $\hookleftarrow_\alpha \neq 0 \wedge \hookleftarrow_\beta = 0$ **then** $\begin{cases} \leftarrow\!\!\!\shortmid_X := \leftarrow\!\!\!\shortmid_\alpha, \ \sharp_X := \sharp_\alpha \\ \mapsto_X := \max\{\mapsto_\beta, \mapsto_\alpha, m\} \end{cases}$
  else if $\hookleftarrow_\alpha \neq 0 \wedge \hookleftarrow_\beta \neq 0$ **then** $\begin{cases} \leftarrow\!\!\!\shortmid_X := \leftarrow\!\!\!\shortmid_\alpha, \ \mapsto_X := \mapsto_\beta \\ \sharp_X := \sharp_\alpha + \sharp_\beta + \max\{\mapsto_\alpha, \leftarrow\!\!\!\shortmid_\beta, m\} \end{cases}$
procedure INIT_AUTOMATON( )
  for $a \in \Sigma$ **do**
    if $a = \hookleftarrow$ **then** $\hookleftarrow_a := 1$
    for $q_f \in F$ **do**
      if $(q_0, a, q_f) \in \delta$ **then**
        $\leftarrow\!\!\!\shortmid_a := 1$; $\mapsto_a := 1$
function MAIN
  INIT_AUTOMATON( )
  for each $\ell = 1, 2, \ldots, |V|$ **do**
    let $(X_\ell \to \alpha_\ell \beta_\ell) \in \mathcal{R}$
    $new\_match := -1$
    for each $q_1, q' \in Q$ s.t. $(q_1, \alpha_\ell, q') \in \delta$ or $q_1 = q' = q_0$ **do**
      for each $q_2 \in Q$ s.t. $(q', \beta_\ell, q_2) \in \delta$ or $q' = q_2 \in F$ **do**
        $\delta := \delta \cup \{(q_1, X_\ell, q_2)\}$
        if $(q_1, q_2) \in (\{q_0\} \times F)$ **then**
          $new\_match := \max\{new\_match, 0\}$
          if $q' \notin (\{q_0\} \cup F)$ **then** $new\_match := 1$
    if $new\_match \neq -1$ **then** $\mathcal{C}_{X_\ell}$:=COUNT($\mathcal{C}_{\alpha_\ell}, \mathcal{C}_{\beta_\ell}, new\_match$)
    else $\mathcal{C}_{X_\ell}$:=0
  if $\hookleftarrow_S = 0$ **then return** $\leftarrow\!\!\!\shortmid_S$
  else return $\sharp_S + \leftarrow\!\!\!\shortmid_S + \mapsto_S$

Algorithm 3 Extension of Algorithm 3 to improve its performance

Input: An ESLP $P = (V, \Sigma, \mathcal{R})$ and an FSA $A = (Q, \Sigma, q_0, F, \delta)$
Output: Number of lines in $\mathcal{L}(P)$ containing a factor matching A.

1: **procedure** COUNT($\mathcal{C}_\alpha$, $\mathcal{C}_\beta$, m) ▷ Inherited from Algorithm 2
2: **procedure** INIT_AUTOMATON( ) ▷ Inherited from Algorithm 2
3: **function** MAIN
4: INIT_AUTOMATON( )
5: **for each** $\ell = 1, 2, \ldots, |V| - 1$ **do** [...] ▷ Inherited from Algorithm 2
6: active_states := $\emptyset$
7: $\mathcal{C}_X := 0$
8: **let** $(X_{|V|} \to \sigma) \in \mathcal{R}$
9: **for each** $\lambda \in (\sigma)_0 \ldots (\sigma)_\dagger$ **do**
10: $new\_match := -1$
11: aux := $\emptyset$
12: **for each** $(q_1, \lambda, q_2) \in \delta$ s.t. $q_1 \in$ active_states **do**
13: aux := aux $\cup\{q_2\}$
14: **if** $q_2 \in F$ **then**
15: $new\_match := \max\{new\_match, 0\}$
16: **if** $q_1 \notin (\{q_0\} \cup F)$ **then** $new\_match := 1$
17: **if** $new\_match \neq -1$ **then** $\mathcal{C}_X :=$COUNT$(\mathcal{C}_X, \mathcal{C}_\lambda, new_match)$
18: **else** $\mathcal{C}_X$:=0
19: active_states := aux
20: **if** $\hookleftarrow_{X_{|V|}} = 0$ **then return** $\leftarrow\!\!\!\shortmid_{X_{|V|}}$
21: **else return** $\sharp_{X_{|V|}} + \leftarrow\!\!\!\shortmid_{X_{|V|}} + \shortmid\!\!\!\rightarrow_{X_{|V|}}$