# Regular Expression Matching on Compressed Text

**Pierre Ganty**
IMDEA Software Institute, Spain
pierre.ganty@imdea.org

**Pedro Valero**
IMDEA Software Institute, Spain & Universidad Politécnica de Madrid, Spain
pedro.valero@imdea.org

## Abstract

The state of the art approach to perform regular expression matching on compressed text is to feed the uncompressed text, as it is recovered by the decompressor, into the regular expression engine. We present `zearch`, a tool for searching *directly* on the compressed text, requiring linear time with respect to the size of the compressed data which might be exponentially smaller than its uncompressed version. The experiments show that `zearch`, in its current sequential implementation, outperforms some *highly optimized* implementations of the decompress and search approach even when decompressor and search engine each have a distinct computing unit.

**2012 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.CVIT.2016.23

## 1 Introduction

The growing amount of information handled by modern systems demands efficient techniques both for compression, to reduce the storage cost, and for regular expression matching, to speed up processing. Despite the efforts invested in designing algorithms for searching with a regular expression *directly* in compressed text, there is no algorithm that nowadays outperforms feeding the output of the decompressor into the search engine.

Lossless compression of textual data is achieved by finding repetitions in the input text and replacing them by references. We focus on grammar-based compression schemes in which each tuple "reference → repeated text" corresponds to a rule of a context-free grammar. The resulting grammar, produced as the output of the compression, generates a language consisting of a single word: the uncompressed text. Figure 1 depicts the output of a grammar-based compression algorithm.

$S \to \mathrm{x}S_2S_2yS_4S_4zS_2$
$S_4 \to S_3\mathrm{b}$
$S_3 \to \mathrm{a}\$$
$S_2 \to S_1\$$
$S_1 \to \mathrm{ab}$

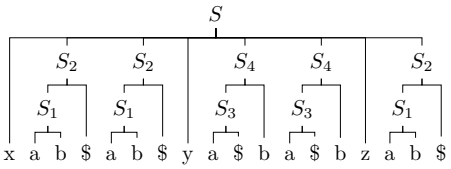


***Figure 1*** *List of grammar rules (left) generating the string "$xab\$ab\$ya\$ba\$bzab\$$" (and no other) as evidenced by the parse tree (right).*

Our approach is briefly described as follows: given a grammar-compressed text and a regular expression compute, for each variable of the grammar, the update on the state of the search that would result from processing the string generated from that variable (assuming the search is performed with an automata-based algorithm). The algorithm iterates through the grammar rules and composes, for each of them, the updates previously computed for the

variables on the right hand side. For example, when processing rule $S_2 \to S_1\$$ of Figure 1 our algorithm composes the update for $S_1$ with the one for \$. The resulting information, i.e. the update that results from processing the string "ab\$", 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` that reports the number[1] of lines in the uncompressed text containing a substring that matches the expression. Our experiments show that `zearch` significantly outperforms previous tools for regular expression matching on compressed text and offers better results than some of the most efficient implementations of the decompress and search approach. The later holds also when decompression and search are done in parallel, even though our implementation is purely sequential.

**Related work.** Results in regular expression matching on grammar-compressed text fall into two main groups: a) characterization of the problem's complexity from a theoretical point of view [1, 2, 3] and b) development of algorithms and data structures to efficiently solve different instances of the problem (pattern matching [4, 5, 6], approximate pattern matching [7, 8], multi-pattern matching [9, 10], regular expression matching [11, 7]...).

Abboud et al. [3] proved that, under the Strong Exponential Time Hypothesis, there is no combinatorial algorithm deciding whether a grammar-compressed text contains a match of a deterministic FSA running on $\mathcal{O}((p \cdot s)^{1-\varepsilon})$ time [Thm. 3.2] with $\varepsilon > 0$ where $p$ is the size of the compressed text and $s$ is the number of states of the FSA. Similarly, for the case of non deterministic FSA they proved that, under the $k$-Clique Conjecture, there is no such algorithm running on $\mathcal{O}((p \cdot s^3)^{1-\varepsilon})$ time [Thm. 4.2]. As we show in this paper, our algorithm is optimal for both scenarios since it exhibits $\mathcal{O}(p \cdot s^3)$ and $\mathcal{O}(p \cdot s)$ time complexities for the deterministic and non deterministic cases respectively.

On the other hand, algorithms for regular expression matching on grammar-compressed text are, traditionally, designed for a particular compression scheme [4, 11, 7]. The first algorithm to solve this problem is due to Navarro [11] and it is defined for LZ78/LZW compressed text. His algorithm reports all positions in the uncompressed text at which a substring that matches the expression ends and exhibits $\mathcal{O}(2^s + s \cdot N + \mathrm{occ} \cdot s \log s)$ worst case time complexity using $\mathcal{O}(2^s + p \cdot s)$ space, where $N$ is the size of the uncompressed text, *occ* is the number of occurrences, and $p$ and $s$ are defined as above. To the best of our knowledge this is the only algorithm for regular expression matching on compressed text that has been implemented and evaluated in practice. Bille et al. [7] improved the result of Navarro defining a data structure of size $o(p)$ to represent LZ78 compressed texts and an algorithm that, given a parameter $\tau$, finds all occurrences of a regular expression in a LZ78 compressed text in $\mathcal{O}(p \cdot s(s + \tau) + \mathrm{occ} \cdot s \cdot \log s)$ time using $\mathcal{O}(p \cdot s^2/\tau + p \cdot s)$ space. Recall that *occ* might be as large as $N$.

We define an *occurrence* as a line of uncompressed text containing an a match of the expression and design an algorithm to report the number of occurrences in $\mathcal{O}(p \cdot s^3 \cdot \log(N))$ time using $\mathcal{O}(p \cdot s^2 \cdot \log(N))$ space. Hence, `zearch` reports the number of occurrences of a *fixed* regular expression in a compressed text in $\mathcal{O}(p \cdot \log N)$ time while previous algorithms require $\mathcal{O}(N)$ since occ might be as large as $N$. Note that when we are interested in printing the matches, the dependence with $N$ in the running time cannot be avoided.

Finally, note that finding the smallest grammar – $m^*$ – that generates a given string is an intractable problem and even approximating its size up to a small constant factor is NP-hard [12]. However there are several polynomial approximations such as LZ78 [13], LZW [14], RePair [15] or Sequitur [16] among others. While the algorithms of Navarro

[1] Optionally `zearch` also reports the matching lines.

and Bille et al. tie themselves to a particular approximation (LZ78/LZW) our approach is defined over straight line programs (grammars generating a single string), hence it applies to any grammar-based compression scheme, even for future ones yet to be defined. This generality of our algorithm has practical implications since, as shown by Hucke et al. [17], the LZ78 representation of a text of length $N$ has size $\Omega\big(|m^*| \times (N/\log N)^{2/3}\big)$, while its RePair representation has size $\Omega\big(|m^*| \times \log N/\log\log N\big)$. Therefore, even though the complexity of our algorithm and the one of Bille et al. depend on $p$, for the same uncompressed text we deal with much smaller values of $p$.

**Paper structure.** Section 2 introduces the notation used to prove[2], in Section 3, the correctness of our algorithm; Section 4 describes the implementation of our tool while Section 5 discusses the experiments carried out and the obtained results; Section 5 analyzes the complexity of `zearch` and Section 6 concludes with a brief discussion on some possible directions for future work.

## 2 Preliminaries

An *alphabet* $\Sigma$ is a nonempty finite set of *symbols*. A *string* $w$ is a finite sequence of symbols of $\Sigma$ where the empty sequence is denoted $\varepsilon$. A *language* is a set of strings and the set of all strings over $\Sigma$ is denoted $\Sigma^*$. 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 $\varepsilon$ otherwise. Similarly, $(w)_{i,j}$ denotes the substring, also called *factor*, of $w$ between the $i$-th and the $j$-th symbols, both included.

A *finite state automaton* (FSA or automaton for short) is a tuple $A = (Q, \Sigma, I, F, \delta)$ where $Q$ is a finite set of *states*; $I \subseteq Q$ are the *initial states*; $\Sigma$ 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 $\varepsilon$-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 $\{w \mid q_0 \xrightarrow{w} q, q_0 \in I, q \in F\}$. A language $L$ is said to be *regular*, if there exists an FSA $A$ such that $L = \mathcal{L}(A)$. A string $w$ *matches* against the automaton, and we say it is a *match*, if $w \in \mathcal{L}(A)$. For clarity, we assume through the paper that $I$ and $F$ are disjoint. Otherwise $\varepsilon \in \mathcal{L}(A)$ therefore for every string there is a factor (in this case $\varepsilon$) that matches the automaton.

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 *axiom* $S$; $\Sigma$ 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 $\rho, \sigma$ such that $\rho \Rightarrow^* \sigma$ (the reflexo-transitive closure of $\Rightarrow$), there exists a witnessing sequence (possibly empty) of steps between $\rho$ and $\sigma$. Incidentally when $\rho \in V$ and $\sigma \in \Sigma^*$ such a step sequence is called a *derivation* (of variable $\rho$). 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 $L = \mathcal{L}(G)$.

[2] Due to the lack of space some proofs are deferred to the Appendix.

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. $\mathcal{R} = (r_i)_{i \in 1,\dots,|V|}$) where each indexed rule has the form $r_i = (X_i, \alpha_i \beta_i)$ with $\alpha_i, \beta_i \in (\Sigma \cup \{X_j \in V \mid j < i\})$ and $S = X_{|V|}$. That is, variables are also indexed, there is exactly one rule per variable and their index coincide. Note that variables in the right hand side of $r_i$ are left hand side of a rule with a lower index. We refer to $X_{|V|} \to \alpha_{|V|}\beta_{|V|}$ as the *axiom rule*. 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 SLP*, hereafter ESLP, is a grammar $(V \cup \{X_{|V|+1}\}, \Sigma, X_{|V|+1}, \mathcal{R} \cup \{r_{|V|+1}\})$ such that $(V, \Sigma, \mathcal{R})$ is an SLP, and the *axiom rule*, $r_{|V|+1} = (X_{|V|+1} \to \sigma)$, has $\sigma \in (\Sigma \cup V)^*$ with $|\sigma| > 2$. We define the size of an ESLP $P$ as $|P| = |\mathcal{R}| + |\sigma|$.

## 3 Regular Expression Matching on Grammar-compressed Texts

Esparza et al. [18] 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 $\alpha$ and $\beta$. 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 is a run $q_0 \xrightarrow{w} q_f$ with $\mathcal{L}(P) = \{w\}$, $q_0 \in I$ and $q_f \in F$.

**Algorithm 0** Saturation construction for SLPs

**Input:** SLP $P = (V, \Sigma, \mathcal{R})$, FSA $A = (Q, \Sigma, I, F, \delta)$
**Output:** $\mathcal{L}(P) \overset{?}{\in} \mathcal{L}(A)$

```
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 ((I × {X_|V|} × F) ∩ δ) ≠ ∅
```

However, standard search tools go beyond Algorithm 0 and check the existence of factors in $w$ that match the regular expression. In our setup, this is equivalent to answering the question $\mathcal{L}(P) \overset{?}{\in} (\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*)$ which can be done by adding $\{(q, a, q) \mid q \in (I \cup F), a \in \Sigma\}$ to the set of transitions of the automaton before applying Algorithm 0. This will increase the number of operations carried out by the algorithm which will end up adding $\{(q, X, q) \mid q \in (I \cup F), X \in V\}$ to $\delta$. To prevent the performance issues derived from handling these transitions, we consider them implicitly as shown by Algorithm 1.

**Algorithm 1** Regex matching on SLP-compressed text

**Input:** SLP $P = (V, \Sigma, \mathcal{R})$, FSA $A = (Q, \Sigma, I, F, \delta)$
**Output:** $\mathcal{L}(P) \overset{?}{\in} (\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*)$

```
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 or q_1 = q' ∈ I do
5:       for each q_2 ∈ Q s.t. (q', β_ℓ, q_2) ∈ δ or q' = q_2 ∈ F do
6:         δ := δ ∪ {(q_1, X_ℓ, q_2)}
7:   return ((I × {X_|V|} × F) ∩ δ) ≠ ∅
```

▶ **Theorem 1.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, I, F, \delta)$ an FSA. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$. After $\ell$ iterations of the outermost "for" loop of Algorithm 1, if $X \in \{X_1, \dots, X_\ell\} \subseteq V$ then $((I \times \{X\} \times F) \cap \delta) \neq \emptyset \iff w \in \Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*$ .*

▶ **Corollary 2.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, I, F, \delta)$ an FSA. Algorithm 1 returns* true *iff* $\mathcal{L}(P) \in \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$.

## 3.1 Counting Matching Lines

State of the art tools for regular expression search are equipped with a number of features[3] 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`[4], `rg`[5], or `ack`[6] report the number of lines containing a match, ignoring matches across lines. Next we modify Algorithm 1 to count matching lines, denoting the new-line delimiter by symbol $\hookleftarrow$ and $\widehat{\Sigma} = \Sigma \setminus \{\hookleftarrow\}$. Given $w \in \Sigma^*$, a *line* is a maximal factor of $w$ each symbol of which belongs to $\widehat{\Sigma}$. A *closed line* is a line which is not a prefix nor a suffix of $w$.

▶ **Definition 3** (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 $A{=}(Q, \widehat{\Sigma}, I, F, \delta)$ be an FSA. The *matching lines generated by* $\tau$ is the set

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

▶ **Example 4.** Let $X$ be a variable with $X \Rightarrow^* ba\hookleftarrow ab\hookleftarrow aba$ and let $A$ be an automaton with $\mathcal{L}(A) = \{ab, ba\}$. Then $\mathcal{ML}_X{=}\{(1,2),(4,5),(7,9)\}$ ◊

In order to count the matching lines generated by an SLP with axiom $X_{|V|}$, it suffices to compute $|\mathcal{ML}_{X_{|V|}}|$. Lemmas 5 and 6 show that given a rule $X{\to}\alpha\beta$ the set $\mathcal{ML}_X$ can be computed using $\mathcal{ML}_\alpha$, $\mathcal{ML}_\beta$ and information about the transitions added by Algorithm 1.

▶ **Lemma 5.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, I, F, \delta)$ an FSA. Let $u, v \in \Sigma^+$ such that $X \Rightarrow \alpha\beta \Rightarrow^* u{\cdot}\beta \Rightarrow^* u{\cdot}v$.*

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

▶ **Lemma 6.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, I, F, \delta)$ an FSA. Let $u, v \in \Sigma^+$ such that $X \Rightarrow \alpha\beta \Rightarrow^* u{\cdot}\beta \Rightarrow^* u{\cdot}v$. 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. $(\ell_X, |u|) \in \mathcal{ML}_\alpha$.
2. $(1, r_X{-}|u|) \in \mathcal{ML}_\beta$.
3. *Algorithm 1 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to $\delta$ with $q_0 \in I$, $q' \notin (I \cup F)$ and $q_f \in F$.*

*Furthermore, when the pair $(\ell_X, r_X)$ exists it is unique and*

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

▶ **Example 7.** Let $A$ be an automaton with $\mathcal{L}(A) = \{ab, ba\}$. Consider the grammar rule $X{\to}\alpha\beta$ with $\alpha \Rightarrow^* ba\hookleftarrow a$ and $\beta \Rightarrow^* b\hookleftarrow aba$. Then $X \Rightarrow^* ba\hookleftarrow ab\hookleftarrow aba$. By Definition 3 we have $\mathcal{ML}_\alpha{=}\{(1,2)\}$ and $\mathcal{ML}_\beta{=}\{(3,5)\}$.

Due to properties 1 and 2 of Lemma 5 we have $(1,2),(7,9) \in \mathcal{ML}_X$ and due to property 3 of Lemma 6 we have $(4,5) \in \mathcal{ML}_X$. Therefore $\mathcal{ML}_X = \{(1,2),(4,5),(7,9)\}$. ◊

Observe that, for an uncompressed text of size $N$, $\mathcal{ML}_\tau$ has up to $N$ elements yet its size can be represented in space $\log N$. Next we define the *counting information* of a symbol $\tau$, denoted $\mathcal{C}_\tau$, of size logarithmic in $\mathcal{ML}_\tau$. For a variable $X$ generating the string $w$, $\mathcal{C}_X$

[3] `https://beyondgrep.com/feature-comparison/`
[4] `https://www.gnu.org/software/grep`
[5] `https://github.com/BurntSushi/ripgrep`
[6] `https://github.com/beyondgrep/ack2`

contains: the existence of end-of-line delimiters in $w$ ($\mathtt{NL}(X)$); the existence of a prefix (resp. suffix) of $w$ containing a match ($\mathtt{L}(X)$, resp. $\mathtt{R}(X)$) and the number of closed lines in $w$ containing a match ($\mathtt{M}(X)$). Below Lemma 10 shows that, given a rule $X \to \alpha\beta$, $\mathcal{C}_X$ can be computed using $\mathcal{C}_\alpha$, $\mathcal{C}_\beta$ and the information about the transitions added by Algorithm 1.

▶ **Definition 8** (Counting information)**.** Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, I, F, \delta)$ an FSA. Let $\tau \in (V \cup \Sigma)$ and $w \in \Sigma^+$ with $\tau \Rightarrow^* w$ and

$$\mathtt{L}(\tau) := \exists i \text{ s.t. } (0,i) \in \mathcal{ML}_\tau \qquad \mathtt{R}(\tau) := \exists j \text{ s.t. } (j,\dagger) \in \mathcal{ML}_\tau$$
$$\mathtt{NL}(\tau) := \exists k \text{ s.t. } (w)_k = \hookleftarrow\supset \qquad \mathtt{M}(\tau) := |\{(i,j) \in \mathcal{ML}_\tau \mid 1 < i \leq j < |w|\}|$$

We define the *counting information of* $\tau$ as $\mathcal{C}_\tau = \langle \mathtt{NL}(\tau), \mathtt{L}(\tau), \mathtt{R}(\tau), \mathtt{M}(\tau)\rangle$.

▶ Remark 9. Since $\hookleftarrow\supset \notin \widehat{\Sigma}$, pairs in $\mathcal{ML}_\tau$ do not overlap. Hence

$$|\mathcal{ML}_\tau| = \mathtt{M}(\tau) + \begin{cases} 1 & \text{if } \mathtt{L}(\tau) \vee \mathtt{R}(\tau) \\ 0 & \textit{otherwise} \end{cases} + \begin{cases} 1 & \text{if } \mathtt{NL}(\tau) \wedge \mathtt{L}(\tau) \wedge \mathtt{R}(\tau) \\ 0 & \textit{otherwise} \end{cases}$$

As mentioned before, to count the number of matching lines generated by an SLP with axiom $X_{|V|}$, it suffices to compute $|\mathcal{ML}_{X_{|V|}}|$. By Remark 9, this amounts to compute $\mathcal{C}_{X_{|V|}}$.

▶ **Lemma 10.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A{=}(Q, \widehat{\Sigma}, I, F, \delta)$ an FSA. For each rule $(X{\to}\alpha\beta) \in \mathcal{R}$ let $m =$ true iff Algorithm 1 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to $\delta$ with $q_0 \in I$, $q' \notin (I \cup F)$ and $q_f \in F$. The following equalities hold:*

1. $\mathtt{NL}(X) = \mathtt{NL}(\alpha) \vee \mathtt{NL}(\beta)$
2. $\mathtt{L}(X) = \begin{cases} \mathtt{L}(\alpha) \vee \mathtt{L}(\beta) \vee m & \textit{if } \neg\mathtt{NL}(\alpha) \\ \mathtt{L}(\alpha) & \textit{otherwise} \end{cases}$ ; $\mathtt{R}(X) = \begin{cases} \mathtt{R}(\alpha) \vee \mathtt{R}(\beta) \vee m & \textit{if } \neg\mathtt{NL}(\beta) \\ \mathtt{R}(\beta) & \textit{otherwise} \end{cases}$
3. $\mathtt{M}(X) = \mathtt{M}(\alpha){+}\mathtt{M}(\beta) + \begin{cases} 1 & \textit{if } \mathtt{NL}(\alpha) \wedge \mathtt{NL}(\beta) \wedge (\mathtt{R}(\alpha) \vee \mathtt{L}(\beta) \vee m) \\ 0 & \textit{otherwise} \end{cases}$

▶ **Example 11.** Let $A$ be an automaton with $\mathcal{L}(A) = \{ab, ba\}$. Consider the grammar rule $X{\to}\alpha\beta$ with $\alpha \Rightarrow^* ba{\hookleftarrow\supset}a$ and $\beta \Rightarrow^* b{\hookleftarrow\supset}aba$. Then $X \Rightarrow^* ba{\hookleftarrow\supset}ab{\hookleftarrow\supset}aba$. By Definition 3 we have $\mathcal{C}_\alpha{=}\langle 1,1,0,0\rangle$ and $\mathcal{C}_\beta{=}\langle 1,0,1,0\rangle$.

Following Lemma 10 we find that $\mathcal{C}_X = \langle 1,1,1,1\rangle$, hence $|\mathcal{ML}_X| = 1+1+1 = 3$, by Remark 9, which coincides with the length of the set $\mathcal{ML}_X$ computed in Example 7. ◊

Algorithm 2 extends Algorithm 1 using the counting information to report the number of lines of uncompressed text containing a match of the regular expression. We use the conditional operator `(cond ? op1 : op2)` that returns `op1` if `cond` is *true* and `op2` otherwise. For each symbol $\tau$, $\langle \tau_{\mathtt{NL}}, \tau_{\mathtt{L}}, \tau_{\mathtt{R}}, \tau_{\mathtt{M}}\rangle$ denotes the information computed and stored by Algorithm 2 for $\tau$. As Theorem 12 shows, this equals to the *counting information* of each symbol $\tau$.

▶ **Theorem 12.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \widehat{\Sigma}, q_0, F, \delta)$ an FSA. Algorithm 2 returns $|\mathcal{ML}_{X_{|V|}}|$.*

**Proof.** Note that Definitions 3 and 8 applied to terminals $a \in \Sigma$ result in $\mathcal{ML}_a$ either empty or containing a single pair $(1,1)$ and:

$$\mathtt{L}(a) = \mathtt{R}(a) = \begin{cases} 1 & \text{if } (q_0, a, q_f) \in \delta,\ q_f \in F \\ 0 & \text{otherwise} \end{cases} \quad \mathtt{NL}(a) = \begin{cases} 1 & \text{if } a = \hookleftarrow\supset \\ 0 & \text{otherwise} \end{cases}.$$

It is straightforward to observe that a) after running function INIT_AUTOMATON of Algorithm 2, $a_{\mathtt{NL}} = \mathtt{NL}(a)$, $a_{\mathtt{L}} = a_{\mathtt{R}} = \mathtt{L}(a) = \mathtt{R}(a)$ and $a_{\mathtt{M}} = \mathtt{M}(a)$ for each $a \in \Sigma$ and b) if $\alpha_{\mathtt{NL}} = \mathtt{NL}(\alpha)$, $\alpha_{\mathtt{L}} = \mathtt{L}(\alpha)$, $\alpha_{\mathtt{R}} = \mathtt{R}(\alpha)$ and $\alpha_{\mathtt{M}} = \mathtt{M}(\alpha)$ (and the same for $\beta$) then after running function COUNT of Algorithm 2 equalities from Lemma 10 hold when replacing $\langle \mathtt{NL}(\tau), \mathtt{L}(\tau), \mathtt{R}(\tau), \mathtt{M}(\tau)\rangle$ by $\langle \tau_{\mathtt{NL}}, \tau_{\mathtt{L}}, \tau_{\mathtt{R}}, \tau_{\mathtt{M}}\rangle$. Furthermore, Theorem 1 and Lemmas 5, 6 and 10

**Algorithm 2** Extension of Algorithm 1 to report the number of matching lines

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

```
1:  procedure COUNT(X, α, β, m)
2:    X_NL := α_NL ∨ β_NL
3:    X_L := (¬α_NL ? α_L ∨ β_L ∨ m : α_L);     X_R := (¬β_NL ? α_R ∨ β_R ∨ m : β_R)
4:    X_M := α_M + β_M + (α_NL ∧ β_NL ∧ (α_R ∨ β_L ∨ m) ? 1 : 0)
5:  procedure INIT_AUTOMATON( )
6:    for each a ∈ Σ do
7:      a_NL := (a = ↩)
8:      a_L := ((q0, α, qf) ∈ δ, q0 ∈ I, qf ∈ F);   a_R := a_L
9:      a_M := 0
10: function MAIN
11:   INIT_AUTOMATON( )
12:   for each ℓ = 1, 2, ..., |V| do
13:     let (X_ℓ → α_ℓ β_ℓ) ∈ R
14:     new_match := false
15:     for each q1, q' ∈ Q s.t. (q1, α_ℓ, q') ∈ δ or q1 = q' ∈ I do
16:       for each q2 ∈ Q s.t. (q', β_ℓ, q2) ∈ δ or q' = q2 ∈ F do
17:         δ := δ ∪ {(q1, X_ℓ, q2)}
18:         new_match := new_match ∨ (q1 ∈ I ∧ q' ∉ (I ∪ F) ∧ q2 ∈ F)
19:     COUNT(X_ℓ, α_ℓ, β_ℓ, new_match)
20:   return (X_|V|)_M + ((X_|V|)_NL ? (X_|V|)_L + (X_|V|)_R : (X_|V|)_L)
```

also apply to Algorithm 2 since it performs *exactly* the same operations on $\delta$ as Algorithm 1. Therefore Algorithm 2 propagates the counting information from the terminals to the axiom,

Finally, by Remark 9 the value returned by the algorithm is $|\mathcal{ML}_{X_{|V|}}|$. ◀

# 4 Implementation

In this section we describe `zearch`, an implementation based on Algorithm 2 to perform regular expression matching on grammar-compressed text. This tool works on texts compressed with `repair`[7], which implements the Recursive Pairing algorithm defined by Larsson et al. [15]. The choice of a particular compressor simplifies the implementation task since `zearch` has to read the grammar rules from the compressed file. However `zearch` can handle any grammar-based scheme by providing a way to recover the SLP from the input file. We use the automata library `libfa`[8] to translate the input regular expression into an $\varepsilon$-free FSA. Internally it uses Thompson's algorithm [19] with on-the-fly $\varepsilon$-removal.

## 4.1 Data structures

We index the transitions of the automaton by their label, i.e. we store a pointer to the list of labeled transitions, together with the counting information of each symbol, 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 allows the loops in lines 15 and 16 to iterate directly over the transitions labeled by $\alpha$ and $\beta$ respectively.

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

This data structure causes the loop in line 16 of Algorithm 2 to iterate through all transitions labeled with $\beta_\ell$, checking whether they leave from $q'$ or not. We improve this situation by using an auxiliary matrix with $s^2$ elements of size $\log s$ bits. Before entering the loop in line 15, `zearch` stores the elements of the set $\beta_\ell(i) = \{q_j \mid (q_i, \beta_\ell, q_j) \in \delta\} \subseteq Q$ at positions $(i,1)\dots(i,|\beta_\ell(i)|)$ of the matrix. Whenever $|\beta_\ell(i)| < s$, it also stores $-1$ at position $(i, \beta_\ell(i)+1)$ to mark the end of the set. This allows the loop in line 16 of Algorithm 2 to iterate directly over the transitions labeled with $\beta_\ell$ leaving from state $q'$.

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 each of which of size $\log p$ bits. When transition $(q_i, X_\ell, q_j)$ is added to $\delta$, `zearch` first checks if position $(i,j)$ of the matrix contains the value $\ell$. If not, then it adds the pair $(q_i, q_j)$ to the list of transitions labeled by $X_\ell$ and stores $\ell$ 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, the *counting information* of each variable $X$ can be stored using one bit for each $X_{\mathtt{NL}}$, $X_{\mathtt{L}}$ and $X_{\mathtt{R}}$ and $\log N$ bits for $X_{\mathtt{M}}$, where $N$ is the size of the uncompressed text Hence, both function COUNT and INIT_AUTOMATON are executed in $\mathcal{O}(\log N)$ time.

These data structures allow Algorithm 2 to exhibit $\mathcal{O}(p \cdot s^3 \cdot \log N)$ time complexity using $\mathcal{O}(p \cdot s^2 \cdot \log N)$ space for the worst case scenario and $\mathcal{O}(p \cdot \log N + s)$ time complexity using $\mathcal{O}(p \cdot \log N + s)$ space for the best one[9]. Further details on the complexity of `zearch` are given in Section 5.

**Previously proposed data structures.** The data structure suggested by Esparza et al. [18] for the *saturation construction*, in which Algorithm 0 is inspired, consists of a bit array where each position represents a possible transition $(q_i, X_\ell, q_j)$ 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 which makes it a good candidate for implementing Algorithm 2. However, this representation of the automaton forces the loop in line 15 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 $\alpha$. Similarly, the loop in line 16 requires $s$ iterations regardless of the number of transitions labeled with $\beta$. Furthermore, in order to perform the counting, this data structure should be extended with an array of size $\mathcal{O}(p \cdot \log N)$ for keeping the counting information, which should also be updated each time a rule is processed. As a result, with this data structure, Algorithm 2 operates in $\Theta(p \cdot s^3 \cdot \log N)$ time and $\Theta(p \cdot s^2 \cdot \log N)$ space.

## 4.2 Processing the axiom rule.

Grammar-based compression algorithms typically 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 which typically contains more symbols than rules are in the grammar so the way in which the axiom is handled has a great impact in the performance.

An ESLP can be transformed into an SLP by rewriting the axiom rule $X_{|V|} \to \sigma$ as $\{S_1 \to (\sigma)_1 (\sigma)_2\} \cup \{S_i \to S_{i-1}(\sigma)_{i+1} \mid i = 2 \dots |\sigma|-2\} \cup \{X_{|V|} \to S_{|\sigma|-2}(\sigma)_\dagger\}$ so Algorithm 2 could process the axiom rule by translating it into an SLP. It is worth pointing that the transitions labeled $S_i$ can be discarded after processing the grammar rule with $S_{i+1}$ on the

[9] The best case scenario corresponds to the loops in lines 15 and 16 iterating over empty sets.

left hand side. Hence we can reuse the array entry for $S_i$ when processing $S_{i+1}$. Similarly, since for any $S_i$ there is exactly one derivation $X_{|V|} \Rightarrow^* \gamma S_i \rho$ and $\gamma = \varepsilon$, it suffices to store the transitions labeled by $S_i$ leaving from the initial state. As a result we obtain Algorithm 3 (see Appendix B) which exhibits $\mathcal{O}\left(\log N \left((p - |\sigma|) \cdot s^3 + |\sigma| \cdot s^2\right)\right)$ worst case time complexity using $\mathcal{O}\left((p - |\sigma|) \cdot s^2 \cdot \log N\right)$ space where $p$ is the size of the ESLP and $X_{|V|} \to \sigma$ its axiom.

## 4.3 Printing matching lines

The counting information computed by Algorithm 2 can be used to *partially* decompress the input file, *fully decompressing* the matching lines and *skipping* symbols generating no matching lines. This partial decompression of the input file requires `zearch` to store the grammar rules, indexed by the symbol on the left hand side, which can be done in $\mathcal{O}(p)$ space. Therefore, `zearch` has the same space complexity regardless of whether the matching lines are being reported or simply counted.

On the other hand, the time complexity does increase when `zearch` is asked to report the matching lines. In particular, the partial decompression described above requires $\mathcal{O}(v \cdot d)$ extra time, where $v$ is the size of the output generated and $d$ is the depth of the grammar, i.e. the length of the longest sequence $X_{i_1} X_{i_2} \dots X_{i_d}$ of variables of the SLP such that $(X_{i_j} \to \alpha X_{i_{j+1}} \beta) \in \mathcal{R}$ with $\alpha, \beta \in (\Sigma \cup V \cup \{\varepsilon\})$. Note that this is the running time required to recover the uncompressed matching lines from the compressed file.

# 5 Empirical Evaluation

We measure the performance of our implementation by comparing it against `LZgrep` [4] and `GNgrep` [11], described in the *Related Work* section. Both tools operate on LZW-compressed text and, while `LZgrep` is restricted to pattern matching, `GNgrep` performs regular expression matching. These are, to the best of our knowledge, the only two existing tools for regular expression matching on compressed text.

As shown by Table 2, `zearch` significantly outperforms these two tools even by an order of magnitude in some cases. Therefore, we also compared its performance against the state of the art technique for regular expression matching on compressed text: feeding the output of the decompressor into the search engine. We implement this approach by decompressing the file with `zstd`[10] or `lz4`[11] and searching with `grep` or `rg`. These tools are top of the class for lossless compression[12] and regular expression matching[13], respectively. We use `zstd` and `lz4` with the highest compression level since it has negligible impact in the time required for decompression. We also consider `gzip` (with highest compression level) as a reference.

For the experiments we consider versions `grep v3.1`, `rg v0.7.1`, `lz4 v1.8.2`, `zstd v1.3.2`, and `gzip v1.9` running in an Intel Xeon E5640 CPU 2.67 GHz with 20 GB RAM.

**Design of the experiments.** The benchmark designed to assess the performance of `zearch` 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* extracted from the OpenSubtitles2016 dataset [20], a sample of the public timeline recorded by GitHub Archive[14] in *JSON* format, a sample of The Google Books Ngram

[10] `https://github.com/facebook/zstd`
[11] `https://github.com/lz4/lz4`
[12] `https://quixdb.github.io/squash-benchmark/`
[13] `https://rust-leipzig.github.io/regex/2017/03/28/comparison-of-regex-engines/`
[14] https://www.gharchive.org/

Viewer dataset[15] considering n-grams from 5 to 20 characters long in *CSV* format and a *Log* file containing two months worth of all HTTP requests to the NASA Kennedy Space Center WWW server in Florida[16]. Table 1 shows how each compressor behaves on these files[17].

| Uncompressed size | File | Compressed size | | | | | Compression time | | | | | Decompression time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LZW | gzip | repair | zstd | lz4 | LZW | gzip | repair | zstd | lz4 | LZW | gzip | repair | zstd | lz4 |
| 1 MB | *Logs* | **0.19** | 0.1 | **0.08** | **0.07** | 0.12 | **0.04** | 0.04 | 0.19 | **0.56** | **0.03** | **0.02** | 0.01 | 0.01 | **0.004** | **0.003** |
| | *Subtitles* | **0.36** | 0.33 | **0.13** | **0.11** | 0.15 | **0.04** | 0.1 | **0.25** | 0.21 | **0.03** | **0.02** | 0.01 | 0.01 | **0.004** | **0.003** |
| | *JSON* | **0.27** | 0.14 | **0.12** | **0.1** | 0.17 | 0.05 | **0.03** | 0.22 | **0.4** | **0.02** | **0.02** | 0.01 | 0.01 | **0.004** | **0.004** |
| | *CSV* | 0.37 | 0.38 | **0.33** | **0.3** | **0.41** | **0.04** | 0.21 | 0.26 | **0.43** | **0.11** | **0.02** | 0.01 | 0.02 | **0.01** | **0.004** |
| 100 MB | *Logs* | **19** | 11 | **6.8** | **6.3** | 12 | **3.4** | 3.7 | 25.9 | **116.4** | **2.8** | **1.4** | 0.5 | 1.0 | **0.18** | **0.08** |
| | *Subtitles* | **39** | 31 | **14** | **11** | 23 | **4.0** | 9.5 | 31.6 | **62.0** | **4.8** | **1.6** | 0.83 | 1.5 | **0.2** | **0.08** |
| | *JSON* | **26** | 14 | **8.8** | **6.8** | 16 | 3.7 | **3.2** | 25.7 | **59.5** | **2.0** | **1.4** | 0.58 | 1.1 | **0.17** | **0.08** |
| | *CSV* | 37 | 37 | **27** | **24** | **40** | **3.9** | 21.6 | 31.2 | **111.7** | **10.6** | **1.6** | 0.93 | 1.6 | **0.35** | **0.1** |

***Table 1*** *Sizes (in MB) of the compressed files and (de)compression times (in seconds). Values measured with maximum compression levels enabled. (Blue = best tool; bold black text = second best; red = worst one).*

We first run each experiment 3 times as warm up so that the files are loaded in memory, reducing the impact of the I/O bound. Then we measure the running time 30 times and compute the *confidence interval* (with 95% confidence) for the average running time required to count the number of lines matching a regular expression in a certain file using a certain tool. Table 2 summarizes the obtained results when considering, for all files, the regular expressions[18]: "`qwerty`", "`Hello`", "`.`", "`I .* you`", " `[a-z]{4}` " " `[a-z]*[a-z]{6}` ", " `[0-9]7[0-9]4[0-9]9[0-9]0[0-9]` ", "`([a-z]{5} +){4}`". A more extensive evaluation considering a bigger range of file sizes and regular expressions is available online[19].

| Tool | Logs | Subtitles | JSON | CSV | Logs | Subtitles | JSON | CSV |
|---|---|---|---|---|---|---|---|---|
| `grep` | 1.02 | 0.91 | 0.67 | 0.51 | 1.57 | 0.98 | 0.70 | 0.49 |
| `rg` | 1.59 | 1.57 | 0.82 | 1.00 | 2.38 | 1.56 | 0.45 | 1.16 |
| `zearch` | **4.08** | **6.41** | **5.36** | 9.47 | **118.77** | **350.97** | **180.11** | 509.54 |
| `GNgrep` | 13.55 | 9.70 | 10.88 | 6.54 | 6.83 | 4.41 | 5.73 | 2.69 |
| `LZgrep` | 3.30 | 1.86 | 2.54 | 1.24 | 9.12 | 2.41 | 5.93 | 1.67 |
| `gzip+grep` | 2.60 (1.90) | 2.48 (1.76) | 1.98 (1.59) | 1.62 (1.26) | 6.09 (4.43) | 3.41 (2.41) | 4.02 (3.28) | 2.32 (1.82) |
| `lz4+grep` | 1.66 (1.34) | 1.29 (1.10) | 1.17 (0.91) | **0.82** (0.68) | 2.55 (2.19) | 1.31 (1.18) | 1.29 (1.11) | **0.74** (0.66) |
| `lz4+rg` | 2.30 (1.77) | 2.01 (1.68) | 1.48 (1.03) | 1.37 (1.10) | 3.80 (3.34) | 2.05 (1.88) | 1.51 (1.23) | 1.50 (1.40) |
| `zstd+grep` | 1.80 (1.33) | 1.41 (1.08) | 1.30 (1.01) | 1.02 (0.78) | 3.41 (2.36) | 1.66 (1.20) | 1.84 (1.49) | 1.24 (0.97) |
| `zstd+rg` | 2.45 (1.94) | 2.12 (1.75) | 1.61 (1.24) | 1.57 (1.23) | 4.62 (3.45) | 2.40 (1.88) | 1.97 (1.57) | 2.00 (1.55) |

***Table 2*** *Running times (in milliseconds) required to search for a regular expression in a compressed file of uncompressed size 1 MB (left) and 100 MB (right). Note that the first two rows correspond to* searching on the uncompressed text. *We report the running time required by* `zearch` *and the relative time used by the other tools, i.e. the average running time used by* `grep` *when searching on 1 MB of JSON data is* $0.64 \cdot 5.33 = 3.4ms$. *For tools implementing the decompress and search approach we report the results obtained a) using the Linux command* `taskset -c` *to force decompressor and search to use a single CPU (without parenthesis) and b) when no restriction is imposed (in parenthesis) .*

**Analysis of the results.** It follows from Table 2 that `zearch` outperforms `LZgrep` and `GNgrep`, the only two tools that perform the search on the compressed text without decompression. Besides, the compression algorithm on which `LZgrep` and `GNgrep` perform the search, `LZW`, exhibits much worse compression ratio than `repair`, as evidenced in Table 1.

[15] `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`
[16] `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`
[17] The files used for our experiments are available at `TODO`
[18] For `LZgrep` Table 2 shows the average running time required to search for the patterns in the list.
[19] `https://pevalme.github.io/zearch/graphs/index.html`

Comparing `zearch`'s running time on different files we observe that its performance is related to the compression ratio achieved for the given file. This is to be expected since bigger compression ratio means there are less rules in the resulting grammar, each of which is processed by `zearch` exactly once. Hence, `zearch` appears as the fastest tool for searching on a compressed log file, being more than twice as fast as any other tool searching in compressed text and even outperforming `grep` and `rg`, which operate on the uncompressed text. Similarly, `zearch` yields its worst results when working on *CSV*.

Finally remark that, except for *CSV* files, `zearch` offers better performance than any other tool searching on compressed text using a single CPU and even when the decompress and search approach uses two processing units, it is outperformed by `zearch` in most of the experiments. We believe these results evidence the virtues of our approach, which allows us to keep our files effectively compressed (see Table 1) and yet be able to efficiently perform regular expression matching (see Figure 1) on them.

**Theoretical Complexity and Behavior in Practice** In the experiments `zearch` does not run into the worst case scenario, which corresponds to Algorithm 2 adding a transition for each variable and each pair of states (maintaining a complete graph at each step of the algorithm). Such situation implies that each string generated by a grammar variable, i.e. each repeated factor in the uncompressed text, labels a path in the automaton between each pair of states. However, the upper bound for the worst case time complexity of `zearch` can be reduced to $\mathcal{O}\left((p-|\sigma|)\pi\pi_q + |\sigma|\pi\right)$ where $\pi = \max_{w\in\Sigma^*} |\{\langle q_1, q_2\rangle \in Q^2 \mid q_1 \xrightarrow{w} q_2\}|$ and $\pi_q = \max_{w\in\Sigma^*, q_1\in Q} |\{q_2 \in Q \mid q_1 \xrightarrow{w} q_2\}|$. In our experiments we observe that $\pi$ and $\pi_q$ are much smaller than $s^2$ and $s$ respectively.

Let $s_\tau$ be the number of transitions in the automaton labeled with $\tau$. With the data structures described in Section 4, the runtime complexity of processing the rule $X \to \alpha\beta$ with Algorithm 3 is $\mathcal{O}(s_\alpha \times s_\beta)$. Similarly, processing each symbol $(\sigma)_i$ of the axiom rule $X \to \sigma$ requires $\mathcal{O}(s_{(\sigma)_i})$ time. Table 3 shows, for the set of experiments used to generate Table 2, the average value of $s_\alpha \times s_\beta$ and $s_{(\sigma)_i}$ for each rule and axiom symbol of the grammar. Note that the given numbers are much smaller than the theoretical $s^3$.

| File | $p-\lvert\sigma\rvert$ | $s_\alpha \times s_\beta$ | $\lvert\sigma\rvert$ | $s_{(\sigma)_i}$ |
|---|---|---|---|---|
| Subtitles | 34,056 | 8.86 | 32,426 | 1.55 |
| CSV | 142,012 | 12.18 | 24,992 | 0.83 |
| JSON | 35,282 | 8.36 | 27,013 | 1.06 |
| Logs | 28,476 | 6.41 | 12,329 | 0.54 |

| File | $p-\lvert\sigma\rvert$ | $s_\alpha \times s_\beta$ | $\lvert\sigma\rvert$ | $s_{(\sigma)_i}$ |
|---|---|---|---|---|
| Subtitles | 2,646,418 | 3.37 | 2,367,739 | 1.32 |
| CSV | 9,518,105 | 1.91 | 803,548 | 0.51 |
| JSON | 1,929,180 | 3.19 | 1,428,333 | 0.77 |
| Logs | 2,345,505 | 2.58 | 455,132 | 0.46 |

***Table 3*** *Average number of operatiosn performed by Algorithm 2 per grammar rule. The table on the left (resp. right) corresponds to input files of uncompresssed size 1 MB (resp. 100MB).*

## 6 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 despite being purely sequential outperforms the state of the art even when decompression and search are done simultaneously.

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 affect `zearch`'s performance. Furthermore, there are grammar-based compression algorithms such as `Sequitur` 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.

Finally, remark that 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. [21] 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. W. Plandowski and W. Rytter. Complexity of language recognition problems for compressed words. In *Jewels are forever*, 1999.
2. N. Markey and Ph. Schnoebelen. A ptime-complete matching problem for slp-compressed words. *Information Processing Letters*, 2004.
3. A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. *FOCS*, 2018.
4. G. Navarro and J. Tarhio. Lzgrep: a boyer–moore string matching tool for ziv–lempel compressed text. *Software: Practice and Experience*, 2005.
5. E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Direct pattern matching on compressed text. In *SPIRE*, 1998.
6. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2007.
7. P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms*, 2009.
8. J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching on ziv–lempel compressed text. *Journal of Discrete Algorithms*, 2003.
9. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in lzw compressed text. In *DCC*, 1998.
10. P. Gawrychowski. Simple and efficient lzw-compressed multiple pattern matching. *Journal of Discrete Algorithms*, 2014.
11. G. Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 2003.
12. M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 2005.
13. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 1978.
14. Terry A. Welch. A technique for high-performance data compression. *Computer*, 1984.
15. N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *DCC*, 1999.
16. C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 1997.
17. D. Hucke, M. Lohrey, and C. Philipp Reh. The smallest grammar problem revisited. In *SPIRE*, 2016.
18. J. Esparza, P. Rossmanith, and S. Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 2000.

**19** K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 1968.

**20** P. Lison and J. Tiedemann. Opensubtitles2016: Extracting large parallel corpora from movie and TV subtitles. In *LREC*, 2016.

**21** J. D. Ullman and A. Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 1988.

# A Deferred Proofs

## A.1 Proof of Theorem 1

The following definitions allow us to prove Theorem 1.

▶ **Definition 13** (Extended label). Let $A = (Q, \Sigma, I, F, \delta)$ 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 $\big(i = 1 \vee q_1 \in I\big) \wedge \big(j = |w| \vee q_2 \in F\big)$. We refer to factor $(w)_{i,j}$ as the *witness* of the extended label.

▶ **Definition 14** (Minimal witness). Let $A = (Q, \Sigma, I, F, \delta)$ 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 $\{(q, a, q) \mid q \in (I \cup F), a \in \Sigma\}$ were added to $\delta$ then there would be a run $q_1 \xrightarrow{w} q_2$. Proofs of Lemmas 15 and 16 rely on this concept to describe the set of transitions that Algorithm 2 adds to the automaton.

▶ **Lemma 15.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, I, F, \delta)$ an FSA with $I \cap F = \emptyset$. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$ and $q_1, q_2 \in Q$. After $\ell$ iterations of the outermost "for" loop 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.**

**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 $I \cap F = \emptyset$, 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 $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q_1 = q' \in I$ 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 13, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q' = q_2 \in F$ and $(q_1, \alpha, q') \in \delta$. By hypothesis, $q_1 \overset{u}{\rightsquigarrow} q_2$ so, by Definition 13, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q_1 = q' \in I$ and $(q', \beta, q_2) \in \delta$. By hypothesis, $q_1 \overset{v}{\rightsquigarrow} q_2$ so, by Definition 13, $q_1 \overset{w}{\rightsquigarrow} q_2$. ◀

▶ **Lemma 16.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, I, F, \delta)$ an FSA with $I \cap F = \emptyset$. Let $X \in V$, $w \in \Sigma^*$ with $X \Rightarrow^* w$ and $q_1, q_2 \in Q$. After $\ell$ iterations of the outermost "for" loop 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.**

**Base case:** $\ell = 1$.

By definition of SLP, $X_1 \to ab$ with $a, b \in \Sigma$ and, since $I \cap F = \emptyset$, 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 $\delta$ 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 $\delta$ with $q' = q_2 \in F$.
- $w' = b$. Then $(q_1, b, q_2) \in \delta$ and $q_1 \in I$ so $(q_1, X_1, q_2)$ is added to $\delta$ with $q_1 = q' \in I$.

**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 13 $q_1 \overset{u}{\rightsquigarrow} q_3$ 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 $\delta$ when $q'{=}q_3$.
- $w'$ is a factor of $u$. By Definition 13 $q_2 \in F$ 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 $\delta$ when $q'{=}q_2$.
- $w'$ is a factor of $v$. Definition 13 $q_1 \in I$ 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 $\delta$ with $q_1{=}q'$. ◀

**Proof of Theorem 1.** Assume $(q_0, X, q_f) \in \delta$ with $q_0 \in I$ and $q_f \in F$. Then by Lemma 15, $q_0 \overset{w}{\rightsquigarrow} q_f$ and from Definition 13 it follows $w \in \left(\Sigma^* \cdot \mathcal{L}(A) \cdot \Sigma^*\right)$.

Now assume $w{\in}\left(\Sigma^*{\cdot}\mathcal{L}(A){\cdot}\Sigma^*\right)$. There is a factor $w'$ for which $q_0 \xrightarrow{w'} q_f$ with $q_0{\in}I$ and $q_f{\in}F$ so, by Definition 13, $q_0 \overset{w}{\rightsquigarrow} q_f$. By Lemma 16, $(q_0, X, q_f)$ is added to $\delta$ by Algorithm 1. ◀

## A.2 Proof of Lemma 5

1. It follows from Definition 3 since $(w)_{i,j} = (u)_{i,j}$ for all $1 \le i \le j < |u|$.
2. It follows from Definition 3 since $(w)_{i+|u|,j+|u|} = (v)_{i,j}$ for all $1 < i \le j \le |v|$. ◀

## A.3 Proof of Lemma 6

Note that $(w)_i = (u)_i$ for $1 \le i \le |u|$ and $(w)_{i+|u|}{=}(v)_i$ for $1 \le i \le |v|$. Therefore, it follows from Definition 3 that if there exists $(i, j) \in \mathcal{ML}_X$ with $i \le |u| < j$ then $(i, j) = (\ell_X, r_X)$. Next we prove that if any of the properties holds then $(\ell_X, r_X) \in \mathcal{ML}_X$. Let $\widehat{\mathcal{L}}(A) = \widehat{\Sigma}^*{\cdot}\mathcal{L}(A){\cdot}\widehat{\Sigma}^*$.

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 15 $q_0 \overset{u}{\rightsquigarrow} q'$ and $q' \overset{v}{\rightsquigarrow} q_f$ with *witnesses* $u'$ and $v'$ respectively. Since $\hookleftarrow \notin \widehat{\Sigma}$ and $q' \notin (I \cup F)$ 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 3, $(\ell_X, r_X) \in \mathcal{ML}_X$ means $(w)_{\ell_X,r_X} \in \widehat{\mathcal{L}}(A)$ and, therefore, $q_0 \overset{w}{\rightsquigarrow} q_f$ with $q_0 \in I$ and $q_f \in F$. 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 $q'{\in}(Q \setminus (I{\cup}F))$ and, by Lemma 16, property *c)* holds. ◀

## A.4 Proof of Lemma 10

Let $X \Rightarrow^* w$, $\alpha \Rightarrow^* u$, $\beta \Rightarrow^* v$. Next we show that, for each element in $\mathcal{C}_X$, the value given by Definition 8 coincides with the one computed using the formulas given by Lemma 10.

1. $\exists k\ (w)_k = \hookleftarrow$ iff $(\exists k\ (u)_k = \hookleftarrow) \vee (\exists k\ (v)_k = \hookleftarrow)$. Therefore $\mathtt{NL}(X) = \mathtt{NL}(\alpha) \vee \mathtt{NL}(\beta)$.
2. We provide the proof for $\mathtt{L}(X)$ but not for $\mathtt{R}(X)$ since they are conceptually identical. $\mathtt{L}(X) := \exists i$ s.t. $(0,i) \in \mathcal{ML}_\tau$. If $(1,j) \in \mathcal{ML}_X$ with $j < |u|$ then $\mathtt{NL}(\alpha) = \mathit{true}$ and, by Lemma 5, $(1,j) \in \mathcal{ML}_X$ iff $(1,j) \in \mathcal{ML}_\alpha$, i.e. $\mathtt{L}(\alpha) = \mathit{true}$. Otherwise $j = r_X$, $\mathtt{NL}(\alpha) = \mathit{false}$ and, by Lemma 6, $(1, r_X) \in \mathcal{ML}_X$ iff one of the following holds:
   a. $(1,|u|) \in \mathcal{ML}_\alpha$ i.e. $\mathtt{L}(\alpha) = \mathit{true}$.
   b. $(1, r_X - |u|) \in \mathcal{ML}_\beta$ i.e. $\mathtt{L}(\beta) = \mathit{true}$.
   c. Algorithm 1 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to $\delta$ with $q_0 \in I$, $q' \notin (I \cup F)$ and $q_f \in F$ i.e. function COUNT is invoked with $m = \mathit{true}$.

   Therefore $\mathtt{L}(X) = \begin{cases} \mathtt{L}(\alpha) \vee \mathtt{L}(\beta) \vee m & \text{if } \neg\mathtt{NL}(\alpha) \\ \mathtt{L}(\alpha) & \text{otherwise} \end{cases}$
3. Write $\mathcal{ML}_X$ as the union of three disjoint sets as follows:
   $\mathcal{ML}_X = \{(i,j) \in \mathcal{ML}_X \mid i,j < |u|\} \bigcup \{(i,j) \in \mathcal{ML}_X \mid |u| < i,j\} \bigcup \{(i,j) \in \mathcal{ML}_X \mid i \leq |u| \leq j\}$ .
   From Lemma 5 and Definition 8 it follows $\mathtt{M}(X) = \mathtt{M}(\alpha) + \mathtt{M}(\beta) + |\{(i,j) \in \mathcal{ML}_{\mathcal{C}_X} \mid i \leq |u| < j\}|$ with $\{(i,j) \in \mathcal{ML}_{\mathcal{C}_X} \mid i \leq |u| \leq j\} \neq \emptyset$ iff $(\ell_X, r_X) \in \mathcal{ML}_X$ with $\ell_X \neq 1$ and $r_X \neq |w|$ which, by Lemma 6, occurs iff one of the following holds and $\mathtt{NL}(\alpha) = \mathtt{NL}(\beta) = \mathit{true}$:
   a. $(\ell_X, |u|) \in \mathcal{ML}_\alpha$ i.e. $\mathtt{L}(\alpha) = \mathit{true}$.
   b. $(\ell_X, r_X - |v|) \in \mathcal{ML}_\beta$ i.e. $\mathtt{R}(\beta) = \mathit{true}$.
   c. Algorithm 1 adds $(q_0, \alpha, q'), (q', \beta, q_f)$ to $\delta$ with $q_0 \in I$, $q' \notin (I \cup F)$ and $q_f \in F$ i.e. function COUNT is invoked with $m = \mathit{true}$.

   Therefore $\mathtt{M}(X) = \mathtt{M}(\alpha) + \mathtt{M}(\beta) + \begin{cases} 1 & \text{if } (\mathtt{NL}(\alpha) \vee \mathtt{NL}(\beta)) \wedge (\mathtt{R}(\alpha) \vee \mathtt{L}(\beta) \vee m) \\ 0 & \text{otherwise} \end{cases}$ ◀

# B Algorithms

**Algorithm 3** Extension of Algorithm 2 to handle ESLP-compressed text

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

```
 1: function MAIN
 2:   INIT_AUTOMATON( )
 3:   for each ℓ = 1, 2, ..., |V| − 1 do
 4:     execute lines [13–19] of Algorithm 2
 5:   let (X_|V| → σ) ∈ R
 6:   C_X|V| := (NL(X_|V|) := 0, L(X_|V|) := 0, R(X_|V|) := 0, M(X_|V|) := 0)
 7:   active_states := ∅
 8:   for each λ = 1, ..., |σ| do
 9:     C_tmp := C_X|V|
10:     new_match := false
11:     active_states := I ∪ {q2 | (q1, (σ)_λ, q2) ∈ δ, q1 ∈ active_states}
12:     for each (q1, (σ)_λ, q2) ∈ δ s.t. q1 ∈ active_states do
13:       new_match := new_match ∨ (q2 ∈ F ∧ q1 ∉ (I ∪ F))
14:     COUNT(C_X|V|, C_tmp, C_(σ)_λ, new_match)
15:   return (NL(X_|V|) ? M(X_|V|)+L(X_|V|)+R(X_|V|) : L(X_|V|))
```