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

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.

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

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 reduce the storage cost, and for regular expression matching, to speed up 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 overcome the artificial separation between decompression and search by using the information about repetitions in 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 consisting of a single word: the uncompressed text. Figure 1 depicts the output of a grammar-based compression algorithm.

Given a grammar-compressed text and a regular expression we 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. This is done by iterating thorough the grammar rules and composing, for each of them, the updates previously computed for the variables on the right hand side. For instance, when processing rule $S_2 \rightarrow S_1\$$ of Fig. 1 our algorithm composes the update for S_1 with the one for \$. The resulting information, i.e. the update

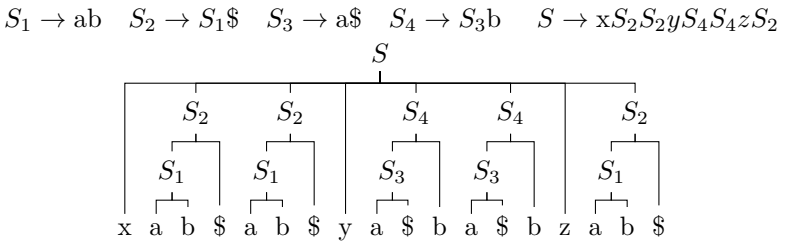

Figure 1 *List of grammar rules (on top) generating the string "$xab\$ab\$ya\$ba\$bzab\$$" (and no other) as evidenced by the parse tree (bottom).*

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 lines in the uncompressed 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.

Related work. The regular expression matching problem on grammar-compressed text has been extensively studied for the last decades. Results in this topic can be divided in two main groups: a) characterization of the problem's complexity from a theoretical point of view [19, 12, 1, 2] and b) development of algorithms and data structures to efficiently solve different instances of the problem (pattern matching [17, 5, 16, 7], approximate pattern matching [4, 14, 9], multi-pattern matching [10, 8], regular expression matching [15, 4]...).

To characterize the complexity of search problems on compressed text it is common to use of straight line programs (grammars generating a single string) to represent the result of the compression algorithm. Straight-line programs are a natural model for compression algorithms such as LZ78 [25], LZW [23], Recursive Pairing [11] or Sequitur [18] and, as proven by Rytter [20], polynomially equivalent to LZ77 [24].

However, algorithms for regular expression matching on grammar-compressed text are typically defined for a particular compression scheme [13, 17, 15, 4]. The first algorithm to solve this problem is due to Navarro [15] and 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 "occ" is the number of occurrences, s is the size of the expression and N is the size of the text compressed to size p. 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. We omitted it from our experimental evaluation since its performance is not competitive with the state of the art tools searching on the uncompressed text as it is recovered by the decompressor.

Bille et al. [4] improved the result of Navarro by defining a relation between the time and space required to perform regular expression matching on compressed text. Indeed they defined a data structure of size $o(p)$ to represent LZ78 compressed texts and an algorithm that, given a parameter τ, finds all occurrences of a regular expression in a LZ78/LZW compressed text in $\mathcal{O}(p \cdot s(s + \tau) + \mathrm{occ} \cdot s \cdot \log s)$ time using $\mathcal{O}(p \cdot s^2/\tau + ps)$ space. To the best of our knowledge, no implementation of this algorithm was carried out.

Our approach differs from the previous ones in the generality of its definition since, by working on straight line programs, our algorithm and its complexity analysis apply to any grammar-based compression scheme. Furthermore, the definition of "occurrence" used in previous works, i.e. positions in the uncompressed text from which we can read a substring matching the expression, is of limited practical interest as evidenced by the fact that tools for regular expression matching on plain text do not share this definition. Instead, these

tools define an occurrence as a line of text containing a match of the expression and so do we. Finally our algorithm reports the number occurrences of a fixed regular expression in a compressed file in $\mathcal{O}(p)$ time. This supposes a major difference with previous algorithms which perform the counting by enumerating all the occurrences. Consequently these algorithms exhibit running times that depend (even for counting) on the number of occurrences, which might be exponentially larger than the size of the compressed text.

Paper structure. Section 2 introduces the notation used to prove[1], 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 6 analyzes the complexity of *zearch* and Section 7 discusses some possible directions for future work.

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 $X_i \to \alpha_i\beta_i$, satisfies $\alpha_i, \beta_i \in (\Sigma \cup \{X_j \in V \mid j < i\})$ and $S = X_{|V|}$. We refer to $X_{|V|} \to \alpha_{|V|}\beta_{|V|}$ as the

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

axiom rule. Note that variables in the right hand side of r_i are left hand side of a 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 $(X_{|V|} \to \sigma) \in \mathcal{R}$ has $|\sigma| \geq 2$. We define the size of an ESLP P as $|P| = |\mathcal{R}| + |\sigma|$.

3 Regular Expression Matching on Grammar-compressed Texts

Esparza et al. [6] 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 is 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: SLP $P = (V, \Sigma, \mathcal{R})$, 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 match the regular expression. In our setup, 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 transitions $\{(q, X, q) \mid q \in (\{q_0\} \cup F), X \in V\}$ to δ, which is also meaningless. Algorithm 1 improves this situation by considering transitions $\{(q_0, X, q_0) \mid X \in (V \cup \Sigma)\}$ and $\{(q_f, X, q_f) \mid X \in (V \cup \Sigma), q_f \in F\}$ even if they are not in δ.

Algorithm 1 Regex matching on SLP-compressed text

Input: SLP $P = (V, \Sigma, \mathcal{R})$, 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 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' = q_0 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 (({q_0} × {X_|V|} × F) ∩ δ) ≠ ∅
```

▶ **Theorem 1.** *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 the outermost "for" loop of Algorithm 1, if $X \in \{X_1, \ldots, X_\ell\} \subseteq V$ then*

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

▶ **Corollary 2.** *Let $P = (V, \Sigma, \mathcal{R})$ be an SLP and $A = (Q, \Sigma, q_0, F, \delta)$ an FSA with $q_0 \notin F$. Algorithm 1 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 regular expression search are equipped with a number of features[2] 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*[3], *ripgrep*[4], *sift*[5] of *ack*[6] 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 matching the expression. 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.

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

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

procedure COUNT($\mathcal{C}_X$, $\mathcal{C}_\alpha$, $\mathcal{C}_\beta$, m)
  $\hookleftarrow_X := \min\{1, \hookleftarrow_\alpha + \hookleftarrow_\beta\}$
  $\leftarrow\!\!\shortmid_X := (\hookleftarrow_\alpha = 0 \,?\, \max\{\leftarrow\!\!\shortmid_\alpha, \leftarrow\!\!\shortmid_\beta, m\} : \leftarrow\!\!\shortmid_\alpha)$;
  $\shortmid\!\!\rightarrow_X := (\hookleftarrow_\beta = 0 \,?\, \max\{\shortmid\!\!\rightarrow_\alpha, \shortmid\!\!\rightarrow_\beta, m\} : \shortmid\!\!\rightarrow_\beta)$
  $\sharp_X := \left((\hookleftarrow_\alpha = 0 \vee \hookleftarrow_\beta = 0) \,?\, \sharp_\alpha + \sharp_\beta : \sharp_\alpha + \sharp_\beta + \max\{\shortmid\!\!\rightarrow_\alpha, \leftarrow\!\!\shortmid_\beta, m\}\right)$
procedure INIT_AUTOMATON( )
  for each $a \in \Sigma$ **do**
    $\hookleftarrow_a := ((a = \hookleftarrow) \,?\, 1 : 0)$
    $\leftarrow\!\!\shortmid_a := (((q_0, \alpha, q_f) \in \delta,\ q_f \in F) \,?\, 1 : 0)$; $\shortmid\!\!\rightarrow_a := \leftarrow\!\!\shortmid_a$
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 := 0
    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 $q_1 \in \{q_0\}$, $q' \notin \left(\{q_0\} \cup F\right)$ and $q_2 \in F$ **then**
          new_match := 1
    COUNT($\mathcal{C}_{X_\ell}$, $\mathcal{C}_{\alpha_\ell}$, $\mathcal{C}_{\beta_\ell}$, new_match)
  return $(\hookleftarrow_{X_{|V|}} = 1 \;?\; \sharp_{X_{|V|}} + \leftarrow\!\!\shortmid_{X_{|V|}} + \shortmid\!\!\rightarrow_{X_{|V|}} : \leftarrow\!\!\shortmid_{X_{|V|}})$

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

Next we prove the correctness of Algorithm 2 using $\hookleftarrow$ to denote the end-of-line delimiter, $\widehat{\Sigma}=\Sigma\setminus\{\hookleftarrow\}$ for any alphabet Σ and defining a *line* as a factor of a string delimited by $\hookleftarrow$ symbols or the ends of the string. Note that Algorithm 2 preserves the functionality of Algorithm 1 (lines [12-17]) and therefore Theorem 1 remains valid when applied to it.

▶ **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 $A=(Q,\widehat{\Sigma},q_0,F,\delta)$ an FSA. The *matching lines generated by* τ is the set

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

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 information about the transitions added by Algorithm 2, as shown by Lemmas 4 and 5.

▶ **Lemma 4.** *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 $u,v\in\Sigma^+$ such that $X\Rightarrow\alpha\beta\Rightarrow^* u{\cdot}\beta\Rightarrow^* u{\cdot}v$.*

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

▶ **Lemma 5.** *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 $u,v\in\Sigma^+$ 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 2 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

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

For each pair $(i,j)\in\mathcal{ML}_X$ with $(w)_{i-1}=(w)_{j+1}=\hookleftarrow$ and each variable Y such that $Y\Rightarrow^*\sigma X\rho$ with $\sigma,\rho\in(\Sigma\cup V)^*$, we have $(i{+}|\sigma|,j{+}|\sigma|)\in\mathcal{ML}_Y$. Therefore storing the set $\mathcal{ML}_X$ 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 allows 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 6** (Counting information)**.** Let $P=(V,\Sigma,\mathcal{R})$ be an SLP and $A=(Q,\widehat{\Sigma},q_0,F,\delta)$ an FSA. Let $\tau\in(V\cup\Sigma)$ and $w\in\Sigma^+$ with $\tau\Rightarrow^* w$ and

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

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

▶ Remark 7. 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 $\leftarrow\!\!|_X$ (which equals $|\!\!\rightarrow_X$) otherwise.

Next we prove that the output of Algorithm 2 corresponds to 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 8.** *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. $\leftarrow\!\!|_X = \begin{cases} \max\{\leftarrow\!\!|_\alpha, \leftarrow\!\!|_\beta, m\} & \text{if } \hookleftarrow_\alpha = 0 \\ \leftarrow\!\!|_\alpha & \text{otherwise} \end{cases}$; $\quad |\!\!\rightarrow_X = \begin{cases} \max\{|\!\!\rightarrow_\alpha, |\!\!\rightarrow_\beta, m\} & \text{if } \hookleftarrow_\beta = 0 \\ |\!\!\rightarrow_\beta & \text{otherwise} \end{cases}$
3. $\sharp_X = \begin{cases} \sharp_\alpha + \sharp_\beta & \text{if } \hookleftarrow_\alpha = 0 \vee \hookleftarrow_\beta = 0 \\ \sharp_\alpha + \sharp_\beta + \max\{|\!\!\rightarrow_\alpha, \leftarrow\!\!|_\beta, m\} & \text{otherwise} \end{cases}$

▶ **Theorem 9.** *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 3 and 6 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 8 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 7:

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

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*[7], which implements the Recursive Pairing algorithm defined by Larsson et al. [11]. 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 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 [21] with on-the-fly ε-removal. This operation is done by using the automata library *libfa*[8].

4.1 Data structures

A straightforward implementation of Algorithm 2 uses the data structure suggested by Esparza et al. [6], which consists of 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

[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`

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 the loop in line 15 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, the loop in line 16 requires s iterations regardless of the number of transitions labeled with β. As a result Algorithm 2 operates in $\Theta(ps^3)$ time and $\Theta(ps^2)$ space.

To reduce the number of iterations required by the loops in lines 15 and 16 of Algorithm 2, 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 α and β respectively. Hence, when no variable of the grammar labels any transition in the automaton, Algorithm 2 operates in $\mathcal{O}(p)$ time using $\mathcal{O}(p)$ space.

Furthermore, to prevent the loop in line 16 of Algorithm 2 from iterating through the up to s^2 transitions labeled with β_ℓ, which will cause $\mathcal{O}(ps^4)$ worst case time complexity, *zearch* uses 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 β_ℓ leaving from state q'.

Finally, 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, 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.

These data structures allow Algorithm 2 to exhibit $\mathcal{O}(ps^3)$ time complexity using $\mathcal{O}(ps^2)$ space for the worst case scenario and $\mathcal{O}(p)$ time complexity using $\mathcal{O}(p)$ space for the best one. Section 6 provides a detailed analysis of the complexity of Algorithm 2.

Processing the axiom rule. Grammar-based compression algorithms as Recursive Pairing [11], LZ78 [25] or LZW [23] 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 ESLPs can be transformed into an SLP by rewriting the axiom rule $X_{|V|}{\to}\sigma$ as $\{S_1{\to}(\sigma)_1(\sigma)_2\} \cup \{S_i{\to}S_{i-1}(\sigma)_{i+1} \mid i = 1\dots|\sigma|{-}2\} \cup \{X_{|V|}{\to}S_{|\sigma|-2}(\sigma)_\dagger\}$ so, in theory, Algorithm 2 could process the axiom rule by translating it into an SLP. It is worth pointing that the transitions labeled 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}. 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 which exhibits $\mathcal{O}\left((p-|\sigma|)s^3 + |\sigma|s^2\right)$ worst case time complexity using $\mathcal{O}\left((p-|\sigma|)s^2\right)$ space where p is the size of the ESLP and $X_{|V|}{\to}\sigma$ its axiom.

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

Input: An ESLP $P = (V, \Sigma, \mathcal{R})$ and an FSA $A = (Q, \Sigma, q_0, F, \delta)$
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 [13–20] of Algorithm 2
5:   let (X_{|V|} → σ) ∈ R
6:   C_{X_{|V|}} := (↩⊐_{X_{|V|}} := 0, ↤_{X_{|V|}} := 0, ↦_{X_{|V|}} := 0, ♯_{X_{|V|}} := 0)
7:   active_states := ∅
8:   for each λ = 1, ..., |σ| do
9:     C_tmp := C_{X_{|V|}}
10:    new_match := 0
11:    active_states := {q_0} ∪ {q_2 | (q_1, (σ)_λ, q_2) ∈ δ, q_1 ∈ active_states}
12:    for each (q_1, (σ)_λ, q_2) ∈ δ s.t. q_1 ∈ active_states do
13:      if q_2 ∈ F and q_1 ∉ ({q_0} ∪ F) then
14:        new_match := 1
15:    COUNT(C_{X_{|V|}}, C_tmp, C_{(σ)_λ}, new_match)
16:  return (↩⊐_{X_{|V|}} = 1 ? ♯_{X_{|V|}} + ↤_{X_{|V|}} + ↦_{X_{|V|}} : ↤_{X_{|V|}})
```

4.2 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 5. This is done by invoking functions P_RIGHT(α); P_LEFT(β), defined in Figure 2, which print the terminals occurring on the right (resp. left) of the rightmost (resp. leftmost) new line symbol generated by a variable.

```
1: procedure P_RIGHT(X)
2:   if X is terminal then
3:     PRINT(X)
4:   else
5:     let (X → αβ) ∈ R
6:     if ↩⊐_β = 0 then
7:       P_RIGHT(α)
8:     P_RIGHT(β)
```

```
1: procedure P_LEFT(X)
2:   if X is terminal then
3:     PRINT(X)
4:   else
5:     let (X → αβ) ∈ R
6:     P_LEFT(α)
7:     if ↩⊐_α = 0 then
8:       P_LEFT(β)
```

Intuitively, we use the counting information to derive the relevant part from X. Let $X \to \alpha\beta$ with X generating a new matching line and $\alpha \to \gamma\rho$ with $\hookleftarrow_\rho = 1$. Then the string generated by symbol γ appears before the new matching line so the derivation is $X \Rightarrow \alpha\beta \Rightarrow \gamma\rho\beta \Rightarrow \rho\beta \Rightarrow \ldots$.

Figure 2 *Functions used to report the new matching line generated by a variable.*

This 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. Finally, these functions require access to arbitrary grammar rules which *zearch* achieves in constant time by storing the rules once they have been processed. As a result, when asked to report the matching lines (besides counting them), *zearch* requires $\mathcal{O}(p)$ extra space and $\mathcal{O}(v)$ extra time, where v is the size of the output generated. Note that this is the minimum running time required to produce the output.

5 Results

In this section we compare the performance[9] of *zearch* against decompressing the file with *zstd*[10] and searching on its output, as it is generated, with *grep* or *ripgrep* (*rg*). These tools are top of the class for lossless compression[11] and regular expression matching[12], respectively. Therefore they are a representative implementation of the state of the art approach for searching with regular expressions in compressed texts. In the experiments all tools are asked to report the number of lines in the uncompressed text matching the regular expression.

Design of the experiments. The benchmark designed for asses 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, JSON and CSV formatted data and an automatically generated log file. To highlight the strengths of our approach we also consider a file where all the lines are the same and a file with a single word of the form "`[0-1]*2`" on which we search with expressions of the form "`[0-1]*1[0-1]{k}2`". These experiments correspond to rows "Contrived text" and "Contrived regex", respectively, in Table 1.

Measuring twice the time required to run a particular command typically yields different results. This is due to a number of factors, such as the CPU and RAM usage of other programs running in the system, that are beyond our reach and influence the performance of any tool. In order to minimize the impact of these factors we run each search 3+30 times. The first 3 executions serve as warm up and the last 30 are used to 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.

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[13].

Analysis of the results. Table 1 shows that, for the set of expressions considered, there is enough statistical evidence to claim that *zearch* outperforms *zstd+grep* (with two processes) working on 100MB long (uncompressed) English subtitles, JSON and Log files. This follows from the disjointness of the respective confidence intervals. Similarly, *zstd+grep* outperforms *zearch* on CSV files. 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 (145ms), JSON (168ms), English subtitles (328ms) and CSV (569ms). This relation is to be expected since *zearch* processes exactly once each rule

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

File	Uncompressed size 1MB					Uncompressed size 100MB				
	zearch	*zstd+grep*		*zstd+rg*		*zearch*	*zstd+grep*		*zstd+rg*	
Subtitles	**6**±0	8±0	6±0	12±0	10±0	**328**±9	467±16	349±12	695±32	560±29
JSON	**5**±0	7±0	5±0	9±0	7±0	**168**±3	284±6	208±3	394±15	288±10
CSV	9±0	9±0	**7**±0	12±0	9±0	569±9	535±14	**419**±7	752±47	576±37
Log	**4**±0	7±0	5±0	11±0	8±0	**145**±2	321±9	220±5	561±29	432±24
Contrived regex	**21**±1	T.O.	T.O.	657±61	656±61	**1597**±46	T.O.	T.O.	24±3 (sec)	23±3 (sec)
Contrived text	**2**±0	9±1	7±1	12±1	9±1	**2**±0	456±62	406±52	700±71	601±69

Table 1 *Confidence intervals for the average running times in milliseconds required to count the number of occurrences of a regular expression in a compressed file. 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.*

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]{k}2`", for which no deterministic automaton has less than 2^n states. *zearch* is the fastest for this experiment (with $k = 10, \ldots, 20$) 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 experiment the 100 MB long input is compressed to 18 MB.

6 Complexity Analysis

As described in Section 4, *zearch* exhibits $\mathcal{O}\left((p - |\sigma|)s^3 + |\sigma|s^2\right)$ worst case complexity using $\mathcal{O}((p - |\sigma|)s^2)$ space where p is the size of the ESLP and $X_{|V|} \to \sigma$ is the axiom rule.

Theoretical Optimality. Abboud et al. [1] proved that, under the Strong Exponential Time Hypothesis, deciding whether a grammar-compressed text contains a match of a FSA requires $\Omega(ps)$ (resp. $\Omega(ps^3)$) time when the automaton is deterministic (resp. non deterministic). It follows that *zearch* is optimal for non deterministic FSA matching since it exhibits $\mathcal{O}(ps^3)$ time complexity. Note that although *zearch* operates on ε-free FSA, this imposes no limitation since ε-transitions can be removed in $\mathcal{O}(s^2)$ time. For a deterministic FSA, since each variable of the grammar generates a single string, the list of transitions labeled by a variable has up to s elements and the data structure described in Section 4 allows *zearch* to operate in $\mathcal{O}(ps)$ time. Hence, *zearch* is optimal for deterministic FSA matching on grammar-compressed text.

Practical Efficiency. The experiments show that, in practice, the running time and space consumption exhibited by *zearch* are far from its worst case complexity.

Running time. Any rule $X \to \alpha\beta$ in which β (or α) labels no edge in the automaton is processed in $\mathcal{O}(s^2)$ time since *zearch* has to iterate through a single list of transitions. Consequently, *zearch* operates in $\mathcal{O}\left(r_0 + r_1 \cdot s^2 + r_2 \cdot s^3 + |\sigma| s^2\right)$ time, 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 some 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.

Space consumption. For any input, *zearch* first allocates an array with $p - |\sigma|$ elements of fixed size, each of them including a pointer to an empty list of transitions. After that, *zearch* allocates extra memory on demand, i.e. when a new transition is added to the automaton. As a result the amount of memory used is $\mathcal{O}(r_0 + r_1 + r_2 + |\delta^\star|)$, where $|\delta^\star|$ is the number of transitions in the automaton upon termination. As evidenced by column $|\delta^\star|$ of Table 2, the average memory usage of *zearch* is far from $\mathcal{O}(ps^2)$.

	Uncompressed size 1MB						Uncompressed size 100MB					
File	$\lvert\sigma\rvert$	$r_0+r_1+r_2$	r_0	r_1	r_2	$\lvert\delta^\star\rvert$	$\lvert\sigma\rvert$	$r_0+r_1+r_2$	r_0	r_1	r_2	$\lvert\delta^\star\rvert$
Subtitles	34	32	13	9	10	30	2646	2367	913	668	786	1585
JSON	35	27	18	5	4	12	1929	1428	1059	277	92	640
CSV	142	24	15	3	6	13	9518	803	500	142	161	330
Log	28	12	4	3	5	12	2345	455	147	169	139	448
Contrived regex	90	9	0.7	0.01	8.29	175	6694	499	39	15	445	9372
Contrived text	0.002	0.2	0.01	0	0.19	0.1	0.03	0.27	0.13	0	0.14	0.11

Table 2 *Statistical data corresponding to experiments in Table 1. Values given in thousands. Each row corresponds to the average of the values obtained for each regular expression used to search in the file.* ($\star$) *Number of transitions in the automaton upon termination of the algorithm.*

7 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*[14] 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. [22] 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*.

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

References

1. Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. *CoRR*, abs/1803.00796, 2018. URL: `http://arxiv.org/abs/1803.00796`, `arXiv:1803.00796`.
2. A. Backurs and P. Indyk. Which regular expression patterns are hard to match? In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–466, Oct 2016. `doi:10.1109/FOCS.2016.56`.
3. Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, 77(2):336–348, 2017.
4. 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.
5. Edleno Silva De Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Direct pattern matching on compressed text. In *String Processing and Information Retrieval: A South American Symposium, 1998. Proceedings*, pages 90–95. IEEE, 1998.
6. Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72:169–177, 2000.
7. Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *J. Exp. Algorithmics*, 13:12:1.12–12:1.31, February 2009. URL: `http://doi.acm.org/10.1145/1412228.1455268`, `doi:10.1145/1412228.1455268`.
8. Paweł Gawrychowski. Simple and efficient lzw-compressed multiple pattern matching. *Journal of Discrete Algorithms*, 25:34–41, 2014.
9. Juha Kärkkäinen, Gonzalo Navarro, and Esko Ukkonen. Approximate string matching on ziv–lempel compressed text. *Journal of Discrete Algorithms*, 1(3-4):313–338, 2003.
10. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in lzw compressed text. In *Data Compression Conference, 1998. DCC '98. Proceedings*, pages 103–112, Mar 1998. `doi:10.1109/DCC.1998.672136`.
11. N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
12. Nicolas Markey and Ph Schnoebelen. A ptime-complete matching problem for slp-compressed words. *Information Processing Letters*, 90(1):3–6, 2004.
13. Shuichi Mitarai, Masahiro Hirao, Tetsuya Matsumoto, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compressed pattern matching for sequitur. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 469–478. IEEE, 2001.
14. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001. URL: `http://doi.acm.org/10.1145/375360.375365`, `doi:10.1145/375360.375365`.
15. Gonzalo Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 1(5):423–443, 2003.
16. Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), April 2007. URL: `http://doi.acm.org/10.1145/1216370.1216372`, `doi:10.1145/1216370.1216372`.
17. 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.
18. Craig G Nevill-Manning and Ian H Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2_and_3):103–116, 1997.
19. Wojciech Plandowski and Wojciech Rytter. Complexity of language recognition problems for compressed words. In *Jewels are forever*, pages 262–272, 1999.

447 **20** Wojciech Rytter. Grammar compression, lz-encodings, and string algorithms with implicit
448 input. In *Automata, Languages and Programming: 31st International Colloquium, ICALP*
449 *2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 15–27, 2004. URL: `https:`
450 `//doi.org/10.1007/978-3-540-27836-8_5`, `doi:10.1007/978-3-540-27836-8_5`.
451 **21** Ken Thompson. Programming techniques: Regular expression search algorithm. *Commu-*
452 *nications of the ACM*, 11(6):419–422, 1968.
453 **22** Jeffrey D Ullman and Allen Van Gelder. Parallel complexity of logical query programs.
454 *Algorithmica*, 3(1-4):5–42, 1988.
455 **23** Terry A. Welch. A technique for high-performance data compression. *Computer*, 6(17):8–19,
456 1984.
457 **24** J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE*
458 *Transactions on Information Theory*, 23(3):337–343, May 1977. `doi:10.1109/TIT.1977.`
459 `1055714`.
460 **25** Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate
461 coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

A Deferred Proofs

A.1 Proof of Theorem 1

The following definitions allow us to prove Theorem 1.

▶ **Definition 10** (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 $\big(i = 1 \vee q_1 = q_0\big) \wedge \big(j = |w| \vee q_2 \in F\big)$. We refer to factor $(w)_{i,j}$ as the *witness* of the extended label.

▶ **Definition 11** (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 $q_1 \xrightarrow{w} q_2$. Proofs of Lemmas 12 and 13 rely on this concept to describe the set of transitions that Algorithm 2 adds to the automaton.

▶ **Lemma 12.** *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 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 $q_0 \notin F$, 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 10, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q' = q_2 \in F$ and $(q_1, \alpha, q') \in \delta$. By hypothesis, $q_1 \overset{u}{\rightsquigarrow} q_2$ so, by Definition 10, $q_1 \overset{w}{\rightsquigarrow} q_2$.
- $q_1 = q' = q_0$ and $(q', \beta, q_2) \in \delta$. By hypothesis, $q_1 \overset{v}{\rightsquigarrow} q_2$ so, by Definition 10, $q_1 \overset{w}{\rightsquigarrow} q_2$. ◀

▶ **Lemma 13.** *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 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 $q_0 \notin F$, 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 10 $q_1 \overset{u}{\rightsquigarrow} q_3$ 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 10 $q_2 \in F$ 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 10 $q_1{=}q_0$ 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 $q_1{=}q'$. ◀

Proof of Theorem 1. Assume $(q_0, X, q_f) \in \delta$ with $q_f \in F$. Then by Lemma 12, $q_0 \overset{w}{\rightsquigarrow} q_f$ and from Definition 10 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 10, $q_0 \overset{w}{\rightsquigarrow} q_f$. By Lemma 13, (q_0, X, q_f) is added to δ by Algorithm 1. ◀

A.2 Proof of Lemma 4

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 5

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 12 $q_0 \overset{u}{\rightsquigarrow} q'$ 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 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 $q_f \in F$. 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 13, property *c)* holds. ◀

A.4 Proof of Lemma 8

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 6 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 4, $(1,j) \in \mathcal{ML}_X$ iff $(1,j) \in \mathcal{ML}_\alpha$, i.e. $\leftarrowtail_\alpha = 1$. Otherwise $j = r_X$, $\hookleftarrow_\alpha = 0$ and, by Lemma 5, $(1, r_X) \in \mathcal{ML}_X$ 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. 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 4 and Definition 6 it follows $\sharp_X = \sharp_\alpha + \sharp_\beta + |\{(i,j) \in \mathcal{ML}_{\mathcal{C}_X} \mid i \leq |u| < j\}|$ 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 5, 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 $\sharp_X = \begin{cases} \sharp_\alpha + \sharp_\beta & \text{if } \hookleftarrow_\alpha = 0 \vee \hookleftarrow_\beta = 0 \\ \sharp_\alpha + \sharp_\beta + \max\{\rightarrowtail_\alpha, \leftarrowtail_\beta, m\} & \text{otherwise} \end{cases}$ ◀