

# Regular Expression Search on Compressed Text

Pierre Ganty\* and Pedro Valero\*†

\*IMDEA Software Institute  
pierre.ganty@imdea.org

†Universidad Politécnica de Madrid  
pedro.valero@imdea.org

## Abstract

We present an algorithm for searching regular expression matches in compressed text. The algorithm reports the number of matching lines in the uncompressed text in time *linear* in the size of its *compressed* version. We define efficient data structures that yield *nearly optimal* complexity bounds and provide a *sequential* implementation –`zearch`– that requires up to 25% less time than the state of the art.

## 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 searching, to speed up querying. A type of query that is supported out of the box by many tools<sup>1</sup> is *counting*: compute how many lines of the input text contain a match for the expression. When the text is given in compressed form, the fastest approach in practice is to query the uncompressed text as it is recovered by the decompressor.

We present an algorithm for counting the lines in a compressed text containing a match for a regular expression whose runtime does not depend on the size  $N$  of the uncompressed text. Instead, it runs in time *linear* in the size of its *compressed version*. Furthermore, the information computed for counting can be used to perform an *on-the-fly, lazy* decompression to recover the matching lines from the compressed text. Note that, for reporting the matching lines, the dependency on  $N$  is unavoidable.

The salient features of our approach are:

**Generality.** Our algorithm is not tied to any particular grammar-based compressor. Instead, we consider the compressed text is given by a straight line program (SLP): a context-free grammar generating the uncompressed text and nothing else.

Finding the smallest SLP  $g$  generating a text of length  $N$  is an NP-hard problem, as shown by Charikar et al. [1], for which grammar-based compressors such as LZ78 [2], LZW [3], RePair [4] and Sequitur [5] produce different approximations. For instance, Hucke et al. [6] showed that the LZ78 algorithm produces a representation of size  $\Omega(|g| \cdot (N/\log N)^{2/3})$  and the representation produced by the RePair algorithm has size  $\Omega(|g| \cdot (\log N/\log \log N))$ . Since it is defined over SLPs, our algorithm applies to all such approximations, including  $g$  itself.

**Nearly optimal data structures.** We define data structures enabling the algorithm to run in time linear in the size of the compressed text. With these data struc-

<sup>1</sup>Tools such as `grep`, `ripgrep`, `awk` and `ag`, among others, can be used to report the number of matching lines in a text.

tures our algorithm runs in  $\mathcal{O}(p \cdot s^3)$  time using  $\mathcal{O}(p \cdot s^2)$  space where  $p$  is the size of the compressed text and  $s$  is the size of the automaton built from the expression. When the automaton is deterministic, the complexity drops to  $\mathcal{O}(p \cdot s)$  time and  $\mathcal{O}(p \cdot s)$  space. Abboud et al. [7] showed that there is no combinatorial<sup>2</sup> algorithm improving these time complexity bounds beyond *polylog* factors, hence our algorithm is *nearly optimal*.

**Efficient implementation.** We present **zearch**, a purely *sequential* implementation of our algorithm which uses the above mentioned data structures.<sup>3</sup> The experiments show that **zearch** requires up to 25% less time than the state of the art: running **hyperscan** on the uncompressed text as it is recovered by **lz4** (in *parallel*). Furthermore, when the grammar-based compressor achieves high compression ratio (above 13:1), running **zearch** on the compressed text is as fast as running **hyperscan** directly on the uncompressed text. This is the case, for instance, when working with automatically generated log files.

## Notation

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  $\varepsilon$ . Let  $|w|$  denote the *length* of  $w$  that we abbreviate to  $\dagger$  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 the (finite) set of *states*;  $I \subseteq Q$  are the *initial states*;  $\Sigma$  is the alphabet;  $F \subseteq Q$  are the *final states*; and  $\delta \subseteq Q \times \Sigma \times Q$  are the *transitions*<sup>4</sup>. The notions of *accepting run*, *accepted string* and *language* of an automaton (denoted  $\mathcal{L}(A)$ ) are defined as expected. 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 ( $\varepsilon$ ) in  $\mathcal{L}(A)$ .

A *Straight Line Program*, hereafter SLP, is a tuple  $P = (V, \Sigma, \mathcal{R})$  where  $V$  is the set of *variables*  $\{X_1, X_2, \dots, X_{|V|}\}$ ,  $\Sigma$  is the alphabet and  $\mathcal{R}$  is the set of rules  $\{X_i \rightarrow \alpha_i \beta_i \mid \alpha_i, \beta_i \in (\Sigma \cup \{X_1, \dots, X_{i-1}\})\}$ . We refer to  $X_{|V|} \rightarrow \alpha_{|V|} \beta_{|V|}$  as the *axiom rule*. We write  $\rho \Rightarrow \sigma$ , with  $\rho, \sigma \in (\Sigma \cup V)^*$ , if there exists  $(\rho)_i = X$  and  $(X \rightarrow \alpha\beta) \in \mathcal{R}$  such that  $\sigma = (\rho)_{1, i-1} \alpha \beta (\rho)_{i+1, \dagger}$ . Denote the reflexo-transitive closure of  $\Rightarrow$  by  $\Rightarrow^*$ . Clearly, the language generated by an SLP consists of a single string  $w \in \Sigma^*$  and, by definition,  $|w| > 1$ . Since  $\mathcal{L}(P) = \{w\}$  we identify  $w$  with  $\mathcal{L}(P)$ .

## Counting Algorithm

Let  $\leftarrow \sqsupset$  denote the new-line delimiter and  $\widehat{\Sigma} = \Sigma \setminus \{\leftarrow \sqsupset\}$ . Given a string  $w \in \Sigma^+$  compressed as an SLP  $P = (V, \Sigma, \mathcal{R})$  and an automaton  $A = (Q, \widehat{\Sigma}, I, F, \delta)$  built from a regular expression, Algorithm 1 reports the number of lines in  $w$  containing a match for the expression. Note that we deliberately ignore matches across lines.

<sup>2</sup>Interpreted as any *practically efficient* algorithm that does not suffer from the issues of Fast Matrix Multiplication such as large constants and inefficient memory usage.

<sup>3</sup>**zearch** can optionally report the matching lines.

<sup>4</sup>Our definition prescribes  $\varepsilon$ -transitions which can be removed in  $\mathcal{O}(s^2)$  time adding no state.

As an overview, our algorithm computes some *counting information* for each alphabet symbol of the grammar (procedure INIT\_AUTOMATON) which is then propagated, in a bottom-up manner, to the axiom rule. Such propagation is achieved by iterating through the grammar rules (loop in line 13) and combining, for each rule, the information for the symbols on the right hand side to obtain the information for the variable on the left (procedure COUNT). Finally, the output of the counting is computed from the information propagated to the axiom symbol (line 20).

---

**Algorithm 1** Counting matching lines.

---

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

**Output:** The number of matching lines in  $\mathcal{L}(P)$ .

```

1: procedure COUNT( $X, \alpha, \beta, m$ )
2:    $N_X := N_\alpha \vee N_\beta$ ;
3:    $L_X := (\neg N_\alpha ? L_\alpha \vee L_\beta \vee m : L_\alpha)$ ;
4:    $R_X := (\neg N_\beta ? R_\alpha \vee R_\beta \vee m : R_\beta)$ ;
5:    $M_X := M_\alpha + M_\beta + (N_\alpha \wedge N_\beta \wedge (R_\alpha \vee L_\beta \vee m) ? 1 : 0)$ ;
6: procedure INIT_AUTOMATON( $\phantom{}$ )
7:   for each  $a \in \Sigma$  do
8:      $N_a := (a = \leftarrow) ; M_a := 0$ ;
9:      $L_a := ((q_0, a, q_f) \in \delta, q_0 \in I, q_f \in F)$ ;
10:     $R_a := L_a$ ;
11: function MAIN
12:   INIT_AUTOMATON( $\phantom{}$ )
13:   for each  $\ell = 1, 2, \dots, |V| - 1$  do
14:     let  $(X_\ell \rightarrow \alpha_\ell \beta_\ell) \in \mathcal{R}$ ;  $\text{new\_match} := \text{false}$ ;
15:     for each  $q_1, q' \in Q$  s.t.  $(q_1, \alpha_\ell, q') \in \delta$  or  $q_1 = q' \in I$  do
16:       for each  $q_2 \in Q$  s.t.  $(q', \beta_\ell, q_2) \in \delta$  or  $q' = q_2 \in F$  do
17:          $\delta := \delta \cup \{(q_1, X_\ell, q_2)\}$ ;
18:          $\text{new\_match} := \text{new\_match} \vee (q_1 \in I \wedge q' \notin (I \cup F) \wedge q_2 \in F)$ ;
19:     COUNT( $X_\ell, \alpha_\ell, \beta_\ell, \text{new\_match}$ );
20:   return  $M_{X|V|} + (N_{X|V|} ? L_{X|V|} + R_{X|V|} : L_{X|V|})$ ;

```

---

Define a *line* as a maximal factor of  $w$  each symbol of which belongs to  $\widehat{\Sigma}$ , a *closed line* as a line which is not a prefix nor a suffix of  $w$  and a *matching line* as a line in  $\widehat{\mathcal{L}(A)}$ , where  $\widehat{\mathcal{L}(A)} = \widehat{\Sigma}^* \cdot \mathcal{L}(A) \cdot \widehat{\Sigma}^*$ . The *counting information* of  $\tau \in (V \cup \Sigma)$ , with  $\tau \Rightarrow^* u$  and  $u \in \Sigma^+$ , is the tuple  $\mathcal{C}_\tau = \langle N_\tau, L_\tau, R_\tau, M_\tau \rangle$  where

$$\begin{aligned}
N_\tau &:= \exists k (u)_k = \leftarrow & L_\tau &:= \exists i (u)_{1,i} \in \widehat{\Sigma}^* \cdot \mathcal{L}(A) \\
R_\tau &:= \exists j (u)_{j,\dagger} \in \mathcal{L}(A) \cdot \widehat{\Sigma}^* & M_\tau &:= |\{(i+1, j-1) \mid (u)_{i,j} \in \leftarrow \cdot \widehat{\mathcal{L}(A)} \cdot \leftarrow\}|
\end{aligned}$$

Note that  $N_\tau, L_\tau$  and  $R_\tau$  are boolean values while  $M_\tau$  is an integer. It follows from the definition that the number of *matching lines* in  $u$ , with  $\tau \Rightarrow^* u$ , is given by the number of *closed matching lines* ( $M_\tau$ ) plus the prefix of  $u$  iff it is a *matching line* ( $L_\tau$ ) and the suffix of  $u$  iff it is a *matching line* ( $R_\tau$ ) different from the prefix ( $N_\tau$ ). Since whenever  $N_\tau = \text{false}$  we have  $L_\tau = R_\tau$ , it follows that

$$\# \text{matching lines in } u = M_\tau + \begin{cases} 1 & \text{if } L_\tau \\ 0 & \text{otherwise} \end{cases} + \begin{cases} 1 & \text{if } N_\tau \wedge R_\tau \\ 0 & \text{otherwise} \end{cases}$$

Computing the counting information of  $\tau$  requires deciding membership of certain factors of  $u$  in  $\widehat{\mathcal{L}(A)}$ . To solve such queries we adapt an algorithm of Esparza et al. [8] designed to decide whether the languages generated by a context-free grammar and an automaton intersect. The resulting algorithm iterates through the rules  $(X \rightarrow \alpha\beta) \in \mathcal{R}$  applying the following operation: add  $(q_1, X, q_2)$  to  $\delta$  iff (a)  $(q_1, \alpha, q'), (q', \beta, q_2) \in \delta$ , (b)  $(q_1, \beta, q_2) \in \delta$  with  $q_1 \in I$ , or (c)  $(q_1, \alpha, q_2) \in \delta$  with  $q_2 \in F$ . This operation

corresponds to lines 15 to 17 of Algorithm 1. As a result, after processing the rule for  $\tau$ , we have  $(q_1, \tau, q_2) \in \delta$  iff the automaton moves from  $q$  to  $q'$  reading (a)  $u$ , (b) a suffix of  $u$  and  $q_1 \in I$ , or (c) a prefix of  $u$  and  $q_2 \in F$ .

Procedures COUNT and INIT\_AUTOMATON are quite straightforward, the main difficulty being the computation of  $M_X$  which we explain next. Let  $x, y \in \Sigma^+$  be the strings generated by  $\alpha$  and  $\beta$ , respectively. Given rule  $X \rightarrow \alpha\beta$ ,  $X$  generates all the matching lines generated by  $\alpha$  and  $\beta$  plus, possibly, a “new” matching line of the form  $z = (x)_{i,\dagger}(y)_{1,j}$  with  $1 < i \leq |x|$  and  $1 \leq j < |y|$ . Such an extra matching line appears iff both  $\alpha$  and  $\beta$  generate a  $\leftarrow$  symbol and either the suffix of  $x$  or the prefix of  $y$  matches the expression or there is a new match  $m \in z$  with  $m \notin x$ ,  $m \notin y$  (line 18).

**Example 1.** Let  $A$  be an automaton with  $\mathcal{L}(A) = \{ab, ba\}$ . Consider the grammar rule  $X \rightarrow \alpha\beta$  with  $\alpha \Rightarrow^* ba\leftarrow a$  and  $\beta \Rightarrow^* b\leftarrow aba$ . Then  $X \Rightarrow^* ba\leftarrow ab\leftarrow aba$ .

The matching lines generated by  $\alpha$ ,  $\beta$  and  $X$  are, respectively,  $\{ba\}$ ,  $\{aba\}$  and  $\{ba, ab, aba\}$ . Furthermore  $\mathcal{C}_\alpha = \langle true, true, false, 0 \rangle$  and  $\mathcal{C}_\beta = \langle true, false, true, 0 \rangle$ .

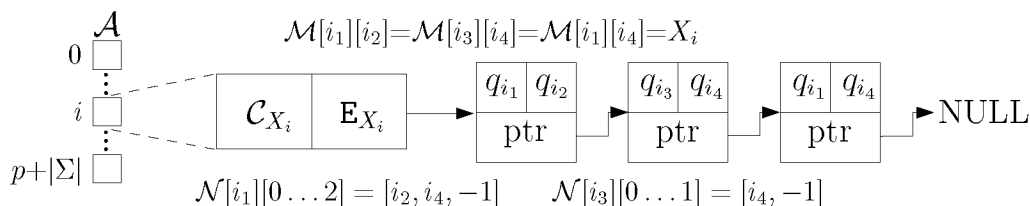
Applying function COUNT we find that  $\mathcal{C}_X = \langle true, true, true, 1 \rangle$ . Therefore the number of matching lines is  $1+1+1=3$ , as expected.  $\diamond$

Note that the counting information computed by Algorithm 1 can be used to uncompress *only* the matching lines by performing a top-down processing of the SLP. For instance, given  $X \rightarrow \alpha\beta$  with  $\mathcal{C}_X = \langle true, true, false, 0 \rangle$  and  $\mathcal{C}_\alpha = \langle true, true, false, 0 \rangle$ , there is no need to decompress the string generated by  $\beta$  since we are certain it is not part of any matching line (otherwise we should have  $M_X > 0$  or  $R_X = true$ ).

Next, we describe the data structures used to implement Algorithm 1 with *nearly optimal* complexity.

### Data Structures

We assume the alphabet symbols, variables and states are indexed and use the following data structures, illustrated in Figure 1: an array  $\mathcal{A}$  with  $p + |\Sigma|$  elements, where  $p$  is the number of rules of the SLP, and two  $s \times s$  matrices  $\mathcal{M}$  and  $\mathcal{N}$  where  $s$  is the number of states of the automaton.



**Figure 1:** Data structures enabling nearly optimal running time for Algorithm 1. The image shows the contents of  $\mathcal{M}$  after processing rule  $X_i \rightarrow \alpha_i\beta_i$  and the contents of  $\mathcal{N}$  after processing  $X_\ell \rightarrow \alpha_\ell\beta_\ell$  with  $\beta_\ell = X_i$ .

Each element  $\mathcal{A}[i]$  contains the information related to variable  $X_i$ , i.e.  $\mathcal{C}_{X_i}$  and the list of transitions labeled with  $X_i$ ,  $E_{X_i}$ . We store  $\mathcal{C}_X$  using one bit for each  $N_X$ ,  $L_X$  and  $R_X$  and an integer for  $M_X$ . For each rule  $X_\ell \rightarrow \alpha_\ell\beta_\ell$  the matrix  $\mathcal{N}$  is set so that row  $i$  contains all the states reachable from  $q_i$  with a transition labeled with  $\beta_\ell$ , i.e.  $\{q_j \mid (q_i, \beta_\ell, q_j) \in \delta\}$ . If there are less than  $s$  such states we use a sentinel value ( $-1$

in Figure 1). Finally, each element  $\mathcal{M}[i][j]$  stores the index  $\ell$  of the last variable for which  $(q_i, X_\ell, q_j)$  was added to  $\delta$ . Note that since rules are processed one at a time, matrices  $\mathcal{N}$  and  $\mathcal{M}$  can be reused for all rules.

These data structures provide  $\mathcal{O}(1)$  runtime for the following operations:

- Accessing the information corresponding to  $\alpha_\ell$  and  $\beta_\ell$  at line 14 (using  $\mathcal{A}$ ).
- Accessing the list of pairs  $(q, q')$  with  $(q, \alpha_\ell, q') \in \delta$  at line 15 (using  $\mathbf{E}_{X_i}$ ).
- Accessing the list of states  $q_2$  with  $(q', \beta_\ell, q_2) \in \delta$  at line 16 (using  $\mathcal{N}$ ).
- Inserting a pair  $(q, q')$  in  $\mathbf{E}_{X_i}$  at line 17 (using  $\mathcal{M}$ ).

As a result, Algorithm 1 runs in  $\mathcal{O}(p \cdot s^3)$  time using  $\mathcal{O}(p \cdot s^2)$  space when the FSA built from the regular expression is non deterministic and it runs in  $\mathcal{O}(p \cdot s)$  time and  $\mathcal{O}(p \cdot s)$  space when the FSA is deterministic (each row of  $\mathcal{N}$  stores up to one state).

Abboud et al. proved [7, Thm. 3.2] that, under the Strong Exponential Time Hypothesis, there is no combinatorial algorithm deciding whether a grammar-compressed text contains a match for a deterministic FSA running in  $\mathcal{O}((p \cdot s)^{1-\varepsilon})$  time with  $\varepsilon > 0$ . For non deterministic FSA, they proved [7, Thm. 4.2] that, under the  $k$ -Clique Conjecture, there is no combinatorial algorithm running in  $\mathcal{O}((p \cdot s^3)^{1-\varepsilon})$  time. Therefore, our algorithm is *nearly optimal* both for deterministic and non deterministic FSA.

## Implementation

We implemented Algorithm 1, using the data structures described in the previous section, in a tool named **zearch**<sup>5</sup>. This tool works on **repair**<sup>6</sup>-compressed text and, beyond counting the matching lines, it can also report them by partially decompressing the input file. The implementation consists of less than 2000 lines of C code.

The choice of this particular compressor, which implements the RePair algorithm [4], is due to the little effort required to adapt Algorithm 1 to the specific grammar built by **repair** and the compression it achieves (see Table 1). However **zearch** can handle any grammar-based scheme by providing a way to recover the SLP from the input file. Recall that we assume the alphabet symbols, variables and states are indexed. For **repair**-compressed text, the indexes of the alphabet symbols are  $0, 1, \dots, 255$  ( $\Sigma$  is fixed<sup>7</sup>) and the indexes of the variables are  $256 \dots p+256$ . Grammar-based compressors encode the grammar so that rule  $X \rightarrow \alpha\beta$  appears always after the rules with  $\alpha$  and  $\beta$  on the left hand side. Thus, each iteration of the loop in line 14 reads a subsequent rule from the compressed input file.

We translate the input regular expression into an  $\varepsilon$ -free FSA using the automata library **libfa**<sup>8</sup> which applies Thompson’s algorithm [9] with on-the-fly  $\varepsilon$ -removal.

## Empirical Evaluation

Next we present a *summary* of the experiments carried out to assess the performance of **zearch**. The details of the experiments, including the runtime and number of

<sup>5</sup><https://github.com/pevalme/zearch>

<sup>6</sup><https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/re-pair/repair1>

<sup>7</sup>Our algorithm also applies to larger alphabets, such as UTF8, without altering its complexity

<sup>8</sup><http://augeas.net/libfa/index.html>

matching lines reported for each expression on each file and considering more tools, file sizes and regular expressions are available on-line<sup>9</sup>.

All tools for regular expression searching considered in this benchmark are used to count the matching lines without reporting them. To simplify the terminology, we refer to counting the matching lines as *searching*, unless otherwise stated.

### *Tools*

Our benchmark compares the performance of **zearch** against the fastest implementations we found for (i) searching the compressed text without decompression, (ii) searching the uncompressed text, (iii) decompressing the text without searching and (iv) searching the uncompressed text as it is recovered by the decompressor.

For searching the compressed text we consider **GNgrep**, the tool developed by Navarro [10] for searching LZW-compressed text. To the best of our knowledge, this is the only existing tool departing from the *decompress and search* approach.

For searching uncompressed text we consider **grep** and **hyperscan**. We improve the performance of **grep** by compiling it without *perl regular expression* compatibility, which is not supported by **zearch**. We used the library **hyperscan** by means of the tool (provided with the library) **simplegrep**, which we modified<sup>10</sup> to *efficiently* read data either from stdin or an input file.

For (de)compressing the files we use **zstd** and **lz4** which are among the best lossless compressors<sup>11</sup>, being **lz4** considerably faster while **zstd** achieves better compression. We use both tools with the highest compression level, which has little impact on the time required for decompression.

We use versions **grep v3.3**, **hyperscan v5.0.0**, **lz4 v1.8.3** and **zstd v1.3.6** running in an Intel Xeon E5640 CPU 2.67 GHz with 20 GB RAM which supports SIMD instructions up to SSE4-2. We restrict to ASCII inputs and set `LC_ALL=C` for all experiments, which significantly improves the performance of **grep**. Since both **hyperscan** and **GNgrep** count positions of the text where a match ends, we extend each regular expression (when used with these tools) to match the whole line. We made this decision to ensure all tools solve the same counting problem and therefore produce the *same output*.

### *Files and Regular Expressions*

Our benchmark consists of an automatically generated *Log*<sup>12</sup> of HTTP requests, English *Subtitles* [11], and a concatenation of English *Books*<sup>13</sup>. Table 1 shows how each compressor behaves on these files.

We first run each experiment 3 times as warm up so that the files are loaded in memory. Then we measure the running time 30 times and compute the *confidence interval* (with 95% confidence) for the running time required to count the number of matching lines for a regular expression in a certain file using a certain tool. We

<sup>9</sup><https://pevalme.github.io/zearch/graphs/index.html>

<sup>10</sup><https://gist.github.com/pevalme/f94bedc9ff08373a0301b8c795063093>

<sup>11</sup><https://quixdb.github.io/squash-benchmark/>

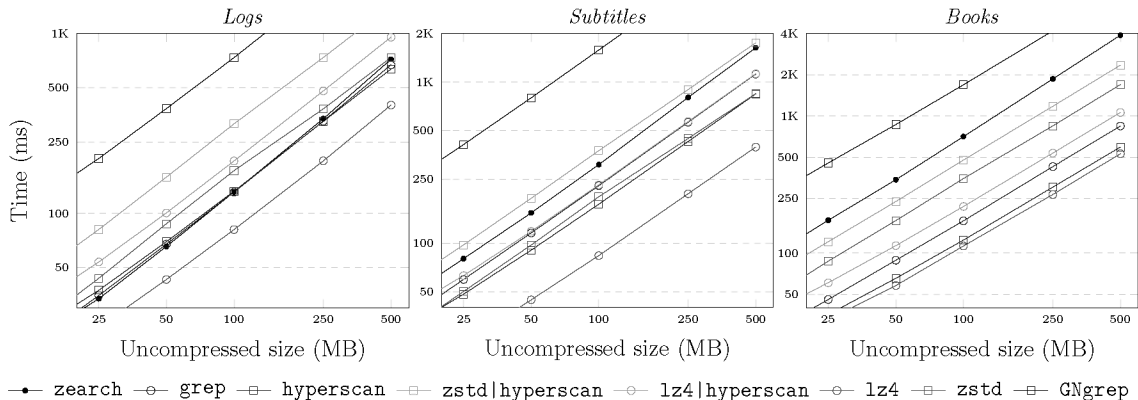
<sup>12</sup><http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html>

<sup>13</sup>[https://web.eecs.umich.edu/~lahiri/gutenberg\\_dataset.html](https://web.eecs.umich.edu/~lahiri/gutenberg_dataset.html)

	File	Compressed size				Compression time				Decompression time			
		LZW	repair	zstd	lz4	LZW	repair	zstd	lz4	LZW	repair	zstd	lz4
Uncompressed size 1 MB	<i>Logs</i>	0.19	<b>0.08</b>	<b>0.07</b>	0.12	<b>0.04</b>	0.19	0.51	<b>0.03</b>	<b>0.02</b>	0.01	<b>0.01</b>	<b>0.004</b>
	<i>Subtitles</i>	0.36	<b>0.13</b>	<b>0.11</b>	0.15	<b>0.04</b>	0.25	0.3	<b>0.03</b>	<b>0.02</b>	0.01	<b>0.01</b>	<b>0.004</b>
	<i>Books</i>	0.42	<b>0.34</b>	<b>0.27</b>	0.43	<b>0.04</b>	0.29	0.42	<b>0.08</b>	<b>0.02</b>	0.02	<b>0.01</b>	<b>0.004</b>
Uncompressed size 500 MB	<i>Logs</i>	96	<b>38</b>	<b>33</b>	65	<b>16.9</b>	123.2	819.1	<b>13.3</b>	<b>7.8</b>	5.5	<b>1.1</b>	<b>0.64</b>
	<i>Subtitles</i>	191	<b>66</b>	<b>55</b>	114	<b>19.9</b>	169.3	415.2	<b>22.8</b>	<b>8.6</b>	8.2	<b>1.2</b>	<b>0.81</b>
	<i>Books</i>	206	<b>153</b>	<b>129</b>	216	<b>20.2</b>	198.6	646.3	<b>40.6</b>	<b>8.6</b>	9.7	<b>2.0</b>	<b>0.8</b>

**Table 1:** Sizes (in MB) of the compressed files and (de)compression times (in seconds). Maximum compression levels enabled. (Blue = best; bold black = second best; red = worst).

consider the *point estimate* of the confidence interval and omit the *margin of error* which never exceeds the 9% of the point estimate for the reported experiments. Figure 2 summarizes the obtained results when considering, for all files, the regular expressions: “what”, “HTTP”, “.”, “I .\* you”, “[a-z]{4}”, “[a-z]\*[a-z]{3}”, “[0-9]{4}”, “[0-9]{2}/(Jun|Jul|Aug)/[0-9]{4}”. For clarity, we report only on the most relevant tools among the ones considered. For `lz4` and `zstd`, we report the time required to decompress the file and send the output to `/dev/null`.



**Figure 2:** Average running time required to report the number of lines matching a regular expressions in a file and time required for decompression. Colors indicate whether the tool performs the search on the uncompressed text (blue); the compressed text (black); the output of the decompressor (green); or decompresses the file without searching (red).

### Analysis of the Results.

Figure 2 and Table 1 show that the performance of `zsearch` improves with the compression ratio. This is to be expected since `zsearch` processes each grammar rule exactly once and better compression results in less rules to be processed. In consequence, `zsearch` is the fastest tool for counting matching lines in compressed *Log* files while it is the second slowest one for the *Books*.

In particular, `zsearch` is more than 25% faster than any other tool working on compressed *Log* files. Actually `zsearch` is competitive with `grep` and `hyperscan`, even though these tools operate on the uncompressed text. These results are remarkable since `hyperscan`, unlike `zsearch`, uses algorithms specifically designed to take advantage of SIMD parallelization.<sup>14</sup>

<sup>14</sup>According to the documentation, `hyperscan` requires, at least, support for SSSE3.

Finally, the fastest tool for counting matching lines in compressed *Subtitles* and *Books*, `lz4|hyperscan`, applies to files larger than the ones obtained when compressing the data with `repair` (see Table 1). However, when considering a better compressor such as `zstd`, which achieves slightly more compression than `repair`, the decompression becomes slower. As a result, `zearch` outperforms `zstd|hyperscan` by more than 7% for *Subtitles* files and 50% for *Logs*.

### Fine-Grained Complexity

The grammars produced by `repair` break the definition of SLP in behalf of the compression by allowing the axiom rule to have more than two symbols on the right hand side. This is due to the fact that the axiom rule is built with the remains of the input text after creating all grammar rules. Typically  $|\sigma| \geq |\mathcal{R}|$  so the way in which the axiom is processed heavily influences the performance of `zearch`.

Furthermore, our experiments show that the performance of `zearch` is typically far from its worst case complexity. This is because the worst case scenario assumes each string generated by a grammar variable labels a path between each pair of states of the automaton. However, we only observed such behavior in contrived examples.

#### *Processing the Axiom Rule.*

Algorithm 1 could process the axiom rule  $X_{|V|} \rightarrow \sigma$  by building an SLP with rules  $\{S_1 \rightarrow (\sigma)_1 (\sigma)_2\} \cup \{S_i \rightarrow S_{i-1} (\sigma)_{i+1} \mid i = 2 \dots |\sigma| - 2\} \cup \{X_{|V|} \rightarrow S_{|\sigma|-2} (\sigma)_\dagger\}$ . However it is more efficient to compute the set of states reachable from the initial ones when reading the string generated with  $S_1$  and update this set for each symbol  $(\sigma)_i$ . To perform the counting note that  $\mathcal{C}_{S_i}$  is only used to compute  $\mathcal{C}_{S_{i+1}}$  and can be discarded afterwards. This yields an algorithm running in  $\mathcal{O}(p \cdot s^3 + |\sigma| \cdot s^2)$  time using  $\mathcal{O}(p \cdot s^2)$  space where  $p$  is the number of rules of the input grammar and  $X_{|V|} \rightarrow \sigma$  its axiom.

#### *Complexity in Terms of Operations Performed by the Algorithm*

Define  $s_{\tau,q} = |\{q' \mid (q, \tau, q') \in \delta\}|$  and  $s_\tau = \sum_{q \in Q} s_{\tau,q}$ . Let us recall the complexity of Algorithm 1 according to the described data structures. The algorithm iterates over the  $p$  rules of the grammar and, for each of them, 1) initializes matrix  $\mathcal{N}$  with  $s_{\beta_\ell}$  elements<sup>15</sup> and 2) iterates through  $\mathcal{N}[q'] [0 \dots s_{\beta_\ell, q'}]$  for each pair  $(q_1, q') \in \mathbf{E}_{\alpha_\ell}$ . Then it processes the axiom rule iterating, for each symbol  $(\sigma_i)$ , through  $s_{(\sigma)_i}$  transitions. These are all the operations performed by the algorithm with running time dependent on the size of the input. Hence, Algorithm 1 runs in  $\mathcal{O}\left(\sum_{\ell=1}^{|V|} \tilde{s}_\ell + \sum_{i=1}^{|\sigma|} s_{(\sigma)_i}\right)$  time where  $\tilde{s}_\ell = s_{\beta_\ell} + s + \sum_{(q_1, q') \in \mathbf{E}_{\alpha_\ell}} (1 + s_{\beta_\ell, q'})$ . Note that  $\tilde{s}_\ell \leq s^3$  and  $s_{(\sigma)_i} \leq s^2$ .

In the experiments we observed that  $\tilde{s}_\ell$  and  $s_{(\sigma)_i}$  are usually much smaller than  $s^3$  and  $s^2$ , respectively, as reported in Table 2. Indeed, `zearch` exhibits almost linear behavior with respect to the size of the FSA built from the expression. Nevertheless, there are regular expressions that trigger the worst case behavior (last row in Table 2), which cannot be avoided due to the result of Abboud et al. [7] describe before.

<sup>15</sup>We need to set up to  $s$  sentinel values for the rows in  $\mathcal{N}$  not used for storing  $s_{\beta_\ell}$



Expression	$s$	$s^3$	percentiles for $\tilde{s}_\ell$					$s^2$	percentiles for $s_{(\sigma)_i}$				
			50%	75%	95%	98%	100%		50%	75%	95%	98%	100%
"I .* you"	8	512	3	13	15	17	28	64	3	3	5	5	8
".*[A-Za-z ]{5}"	7	343	14	25	48	48	48	49	11	14	14	14	14
".*[A-Za-z ]{10}"	12	1728	29	51	86	95	98	144	16	26	29	29	29
".*[A-Za-z ]{20}"	22	10648	57	87	132	153	198	484	23	38	52	58	59
"(((((.)*.)*.)*.)*.)*"	6	216	12	29	209	209	209	36	29	29	29	29	29

**Table 2:** Analysis of the values  $\tilde{s}_\ell$  and  $s_{(\sigma)_i}$  obtained when considering different regular expressions to search Subtitles (100 MB uncompressed long). The fifth column of the first row indicates that when considering the expression "I .\* you", for 75% of the grammar rules we have  $\tilde{s}_\ell \leq 5$  while  $s^3 = 512$ .

## Related Work

Regular expression matching on compressed text is a well-studied problem consisting on deciding whether a string generated by an SLP matches a regular expression (represented as an automaton). Plandowsky et al. [12] reduced the problem to a series of matrix multiplications, showing it can be solved in  $\mathcal{O}(ps^3)$  time ( $\mathcal{O}(ps)$  for deterministic automata) where  $p$  is the size of the grammar and  $s$  is the size of the automaton. Independently, Esparza et al. [8] defined an algorithm to solve a number of decision problems involving automata and context-free grammars which, when restricted to SLPs, results in a particular implementation of Plandowsky’s approach. Removing the function COUNT from Algorithm 1 and replacing the *return* statement of the algorithm by  $((I \times \{X_{|V|}\} \times F) \cap \delta : true ? false)$  we are left with an efficient implementation of Plandowsky’s approach.

The first algorithm searching grammar-compressed text for regular expression matches is due to Navarro [10], **GNgrep**, and it is defined for LZ78/LZW compressed text. His algorithm reports all  $k$  positions in the uncompressed text at which a substring matching the expression ends (but not the matches) in  $\mathcal{O}(2^s + s \cdot p + k \cdot s \cdot \log s)$  time using  $\mathcal{O}(2^s + p \cdot s)$  space. The algorithm computes the number of matches by enumerating these positions. To the best of our knowledge this is the only algorithm searching compressed text for regular expression matches that has been implemented and evaluated in practice. Bille et al. [13] improved the result of Navarro by defining a data structure of size  $o(p)$  to represent LZ78 compressed texts that offers a time-space trade off (reflected in parameter  $\tau$ ) for finding all occurrences of a regular expression in a LZ78 compressed text. Their algorithm operates in  $\mathcal{O}(p \cdot s \cdot (s + \tau) + k \cdot s \cdot \log s)$  time using  $\mathcal{O}(p \cdot s^2 / \tau + p \cdot s)$  space, where  $1 \leq \tau \leq p$ . Note that these approaches to search compressed text exhibit running time linear in the size of the uncompressed text which might be exponentially larger than its compressed version. In consequence, they are not optimal nor competitive with the state of the art, as show by Figure 2.

## Conclusions and Future Work

We have presented the first algorithm for *counting* the number of lines in a grammar-compressed text containing a match for a regular expression. The algorithm applies to any grammar-based compression scheme and is nearly optimal for regular expression

matching on compressed text. Furthermore, we described the data structures required to achieve nearly optimal complexity and used them to implement a (sequential) tool that significantly outperforms the (parallel) state of the art to solve this problem. Indeed, when the grammar-based compressor achieves high compression ratio, which is the case for automatically generated *Log* files, **zearch** uses up to 25% less time than **lz4|hyperscan**, even outperforming **grep** and being competitive with **hyperscan**.

Finally, Algorithm 1 allows for a conceptually simple parallelization since any set of rules such that no variable appearing on the left hand side of a rule appears on the right hand side of another, can be processed simultaneously. Indeed, a theoretical result by Ullman et al. [14] on the parallelization of Datalog queries can be used to show that counting the number of lines in a grammar-compressed text containing a match for a regular expression is in  $\mathcal{NC}^2$  when the automaton built from the expression is acyclic. We are optimistic about the possibilities of a parallel version of **zearch**.

## References

- [1] 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.
- [2] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, 1978.
- [3] T. A. Welch, “A technique for high-performance data compression,” *Computer*, 1984.
- [4] N. J. Larsson and A. Moffat, “Off-line dictionary-based compression,” *DCC*, 1999.
- [5] C. G. Nevill-Manning and I. H. Witten, “Compression and explanation using hierarchical grammars,” *The Computer Journal*, 1997.
- [6] D. Hucke, M. Lohrey, and C. Philipp Reh, “The smallest grammar problem revisited,” in *SPIRE*, 2016.
- [7] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann, “Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve,” *FOCS*, 2018.
- [8] J. Esparza, P. Rossmanith, and S. Schwoon, “A uniform framework for problems on context-free grammars,” *Bulletin of the EATCS*, 2000.
- [9] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Communications of the ACM*, 1968.
- [10] G. Navarro, “Regular expression searching on compressed text,” *Journal of Discrete Algorithms*, 2003.
- [11] P. Lison and J. Tiedemann, “Opensubtitles2016: Extracting large parallel corpora from movie and TV subtitles,” in *LREC*, 2016.
- [12] W. Plandowski and W. Rytter, “Complexity of language recognition problems for compressed words,” in *Jewels are forever*, 1999.
- [13] 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.
- [14] J. D. Ullman and A. Van Gelder, “Parallel complexity of logical query programs,” *Algorithmica*, 1988.