Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Grammar Compression by Induced Suffix Sorting

Daniel Saad Nogueira Nunes CIC/UnB

Document submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Advisor
Prof. Dr. rer. nat. Mauricio Ayala-Rincón

Co-advisor
Prof. Dr. Gonzalo Navarro Badino

Brasília
2022

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Grammar Compression by Induced Suffix Sorting

Daniel Saad Nogueira Nunes CIC/UnB

Document submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Prof. Dr. rer. nat. Mauricio Ayala-Rincón (Advisor)
CIC/UnB
Prof. Dr. Gonzalo Navarro Badino (Co-advisor)
DCC/Universidad de Chile

Prof.a Dr.a Alba Cristina Magalhães Alves de Melo
CIC/UnB

Dr. Francisco Claude-Faust
LinkedIn

Prof. Dr. Guilherme Pimentel Telles
UNICAMP

Prof. Dr. Hideo Bannai
Tokyo Medical and Dental University

Prof. Dr. Ricardo Pezzuol Jacobi
Coordinator of Graduate Program in Informatics

Brasília, March 11th, 2022

Dedicatória

Dedico esta tese aos meus pais e esposa. Sem a compreensão, paciência, amor e apoio incondicionais deles nos momentos difíceis, seria impossível completar este percurso sinuoso, porém imprescindível.

Agradecimentos

Agradeço à Universidade de Brasília, instituição responsável pela minha formação acadêmica desde a minha graduação. Nesta trajetória contei com a presença de vários docentes extremamente qualificados e instalações de qualidade, as quais me forneceram as condições propícias para o desenvolvimento dos trabalhos. Também não posso deixar de mencionar a Universidad de Chile (UCHILE), que sempre me acolheu, fornecendo todo o suporte necessário, nos momentos de cooperação científica com o meu coorientador e seu grupo de pesquisa.

Demostro profunda gratidão aos meus orientadores Mauricio Ayala Rincón (UnB) e Gonzalo Navarro (UCHILE) por toda paciência e apoio nesta longa jornada. Certamente levarei comigo todos os seus ensinamentos para o restante de minha carreira acadêmica.

Não posso deixar de incluir meus colegas Ariane Almeida, Felipe Louza, Jeremias Gomes e Lucas de Melo, que sempre me apoiaram nos momentos complicados e sempre estavam dispostos a conversar sobre as dificuldades. Também agradeço todos os colegas da UnB, IFB, IFG e da UCHILE que se fizeram presentes durante o percurso.

A Fundação de Apoio à Pesquisa do Distrito Federal (FAPDF) também merece todo o reconhecimento, pois sem ela, minhas visitas técnicas com a UCHILE e o estreitamento dos meus laços com o meu coorientador, Gonzalo Navarro, e seu grupo de pesquisa seriam dificultados. As ações de fomento desta fundação foram cruciais para o desenvolvimento de minha pesquisa.

Resumo

Este trabalho apresenta um novo método de compressão por gramáticas chamado GCIS. Este método é baseado na abordagem de ordenação de sufixos por indução, SAIS, apresentada por Nong *et al.* em 2009. A solução proposta utiliza os fatores produzidos pela ordenação SAIS para construir uma gramática livre de contexto que gera o texto. As regras das gramáticas são formadas substituindo cada fator encontrado pela ordenação SAIS por um símbolo não terminal. O método é aplicado recursivamente na sequência composta por não terminais que substitui o texto original até que todos os fatores produzidos sejam distintos. A gramática gerada ainda pode ser comprimida ao explorar redundâncias, tais como os prefixos comuns compartilhado pelo lado direito das regras de produção, que por construção, estão ordenadas. O método GCIS se destaca pelo seu tempo de compressão enquanto mantém a taxa de compressão competitiva. Através de experimentos sobre textos regulares, repetitivos e imensos, GCIS demonstra ser uma escolha factível quando comparado com outros compressores como: Gzip, 7-zip, RePair, a principal referência para compressores baseados em gramáticas, e as recentes alternativas; SOLCA; LZRR; e LZD. Em contrapartida, GCIS não possui uma descompressão tão rápida. Contudo, compressores baseados em gramáticas são mais convenientes do que aqueles baseados nas técnicas de compressão Lempel-Ziv haja vista que possibilitam a extração de subpalavras diretamente da informação comprimida, sem que seja necessário gerar o texto original para tal. Neste cenário, de compressores por gramática, GCIS possui pontos fortes quando comparado aos demais. Também apresentamos que, devido a sua proximidade com a abordagem SAIS, podemos usar GCIS para construir os vetores de sufixos e *longest common prefix* do texto, estruturas fundamentais no processamento de palavras, durante a descompressão da informação.

Palavras-chave: grammar-compression, compact data structures, extraction, suffix-array

Abstract

A grammar compression algorithm, called GCIS, is introduced in this work. GCIS is based on the induced suffix sorting algorithm SAIS, presented by Nong et al. in 2009. The proposed solution builds on the factorization performed by SAIS during suffix sorting to construct a context-free grammar that replaces each distinct factor with a nonterminal. The algorithm is then recursively applied on the shorter sequence of nonterminals. The resulting grammar is encoded by exploiting redundancies, such as common prefixes between right-hands of rules, sorted according to SAIS. GCIS excels for its low space and time required for compression while obtaining competitive compression ratios. Our experiments on regular, repetitive, moderate, and very large texts show that GCIS is a very convenient choice compared to well-known compressors such as Gzip, 7-Zip, RePair, the gold standard in grammar compression, and recent compressors like SOLCA, LZRR, and LZD. In exchange, GCIS is slow at decompressing. Nevertheless, grammar compressors are more convenient than Lempel-Ziv compressors in that one can access text substrings directly in compressed form without ever decompressing the text. We demonstrate that GCIS is an excellent candidate for this scenario because it shows to be competitive among its RePair based alternatives. We also show that the relation with SAIS makes GCIS a good intermediate structure to build the suffix and longest common prefix arrays during decompression of the text.

Keywords: grammar-compression, compact data structures, extraction, suffix-array

Contents

Chapter 1

Introduction

According to Hennessy and Patterson [2019], the performance gap between a single processor core and memory has grown over decades, despite slightly decreasing in recent years. Figure 1.1 plots the number of processor memory requests per second made by the processor and the number of DRAM memory accesses per second. When considering multicore architectures, the gap does not shift in favor of memories: the difference between the number of processors memory requests and memory bandwidth (number of bytes transferred within a time unit) continues to grow as the number of cores increases.

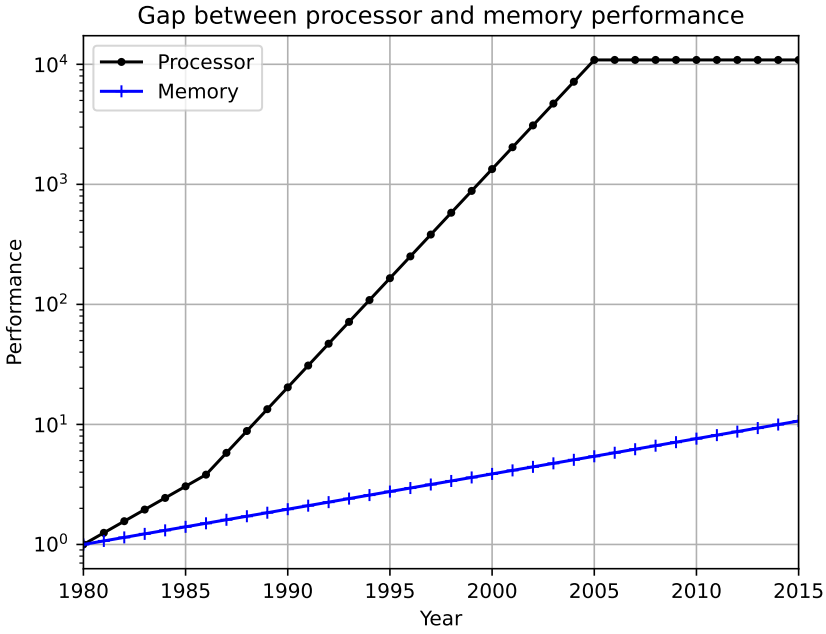


Figure 1.1: Performance gap between single core processor and memory. Adapted from [Hennessy and Patterson, 2019].

In order to mitigate this problem, access to faster memories such as cache memories that utilize SRAM technologies might be considered. However, this does not come out

without a price. Faster memories are much more expensive than cheaper ones, which makes the replacement of usual DRAM for SRAM memories unfeasible. The power factor is also an issue for this kind of memory. They represent a considerable amount of power consumption on mobile devices, and a considerable memory increase would impact battery life severely. The same problem occurs when considering primary and secondary memories, the first containing the RAM memory, whereas the latter dwelling Disk and Flash-based technologies. The design of choice to maintain performance while having plenty resources is to dispose the different types of memories in a hierarchical configuration, also known as **memory hierarchy**. In this hierarchy, slower, cheaper, higher capacity memories such as DRAM or Disk/Flash-based are demanded when the requested data is not available on the faster, more expensive, lower-capacity memories, such as cache and registers. Figure 1.2, shows the time required to access each type of memory with its usual capacity considering a commodity desktop computer. The gap between memories located near the processor, such as registers and cache, to main memory, is considerable.

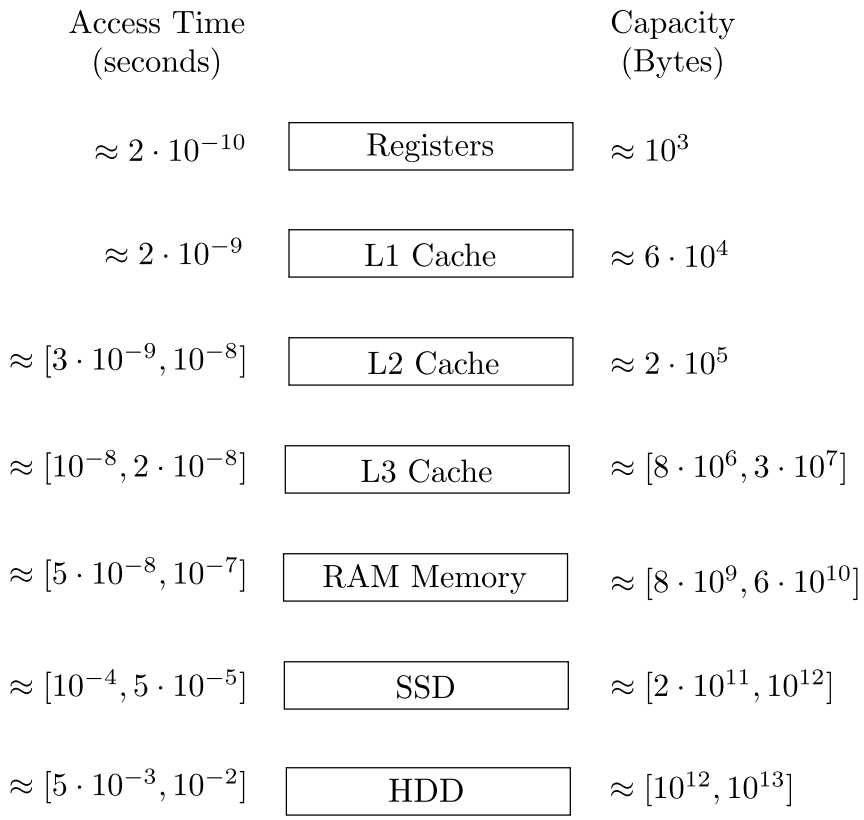


Figure 1.2: Memory Hierarchy. Typical values for access time and capacity regarding different type of memories for a commodity Desktop. Adapted from [Hennessy and Patterson, 2019].

Since high-capacity memories are cheap, it is possible now to store all the data in the main memory regarding several domains. Hence, the shift of several applications from

secondary memory to first memory arises, as well with different concerns, *e.g.* in-memory databases, as presented by [Zhang et al., 2015]. However, this is not always the case. The short read alignment problem consists of determining the *locus* of small DNA sequences (the short reads) in a collection of genomes [Li and Durbin, 2009]. A regular index, which is capable of locating such short reads in efficient time, such as the suffix tree, would require several times the collection of genomes size, thus, forbidding the application to execute main memory.

The need for manipulating data in faster but lower-capacity memories is latent, hence it is crucial to represent such data in a more space-efficient way to work on the top of the memory hierarchy.

Succinct data structures, a term coined by Jacobson [1988], represent the information close, asymptotically speaking, to the lower bound established by information theory while retaining the ability to manipulate such data in feasible time [Kao, 2016]. The term compact data structure also describes space-efficient data structures, but in a more loosened way regarding asymptotic lower terms [Navarro, 2016]. Despite being more complex and slower than the regular data structures, they might be used in cases where the data does not fit in faster memory using the traditional approach for data structure design; hence, they are a beautiful solution for reducing resources for computational problems involving a large volume of data [Navarro, 2016]. Returning to the read alignment problem, by using compact data structures, the alignment can be performed within hundreds of gigabytes [Kuhnle et al., 2020], a common resource for servers.

With statistical compressors, *i.e.*, that assigns shorter codes to more frequent symbols (or sequence of symbols), it is possible to compress and manipulate the data efficiently. The area of research of such compressors has rapidly evolved in the last decades, bringing a wide range of solutions with different time/space trade-offs [Pereira, 2016]. However, this type of compressor cannot deal appropriately with very repetitive data once they do not capture the source's repetitiveness. Such repetitive data are present in many formats, such as:

- Log files: such files describe systems runtime events in a text format for several reasons, *e.g.*, anomaly detection and system failure diagnosis, as stated by [Yao et al., 2021]. However, some log structures might occur several times on a document making general compressors an inappropriate choice for compressing the information [Yao et al., 2020].
- Version control: modern version control systems track the changes within a software repository. Consecutive changes tend to be small compared to the repository size, making the history of changes redundant and susceptible to specialized approaches for repetitive data. The authors in Boldi et al. [2020] describe a solution for this

problem based on graph compression techniques that compress a collection of 5 million source codes into 100 GB, manageable for a modest server.

- Collection of genomes: several problems consider a collection of genomes, such as the read alignment or the 100,000 Genomes Project, the latter being a project from the UK government for sequencing 100,000 genomes focusing the study on rare diseases [Genomics England, 2012]. As expected, data from different genomes from the same organism induces a highly repetitive collection [Kuhnle et al., 2020].

The dictionary-based approach of **grammar compression** is considered a powerful mechanism that addresses the compression of repetitive texts, for its basic idea relies on identifying substrings that occur in the text multiple times and representing such substrings compactly with a pointer to the first occurrence. One benefit of such mechanism is that, if compared with other repetition-aware compression schemes, it may be augmented to support more complex operations, such as extracting substrings or matching patterns.

We present in this document a novel approach for grammar compression, which relies on a consolidated method of suffix sorting by induction of suffixes. Theoretically, the proposed approach is interesting since it brings a novel strategy for grammar compression, and, in practical terms, showed to be competitive with its main competitors regarding the space/time trade-off [Nunes et al., 2018, 2022].

1.1 Contributions

We developed a novel compression approach, called GCIS, that relies on suffix sorting by induction to create a context-free grammar that generates the original text. This grammar can be encoded in a very compact way by exploring redundancies of the sorted suffixes. Besides originating a different strategy for grammar compression and compressing and decompressing the repetitive information with interesting practical results, GCIS is also capable of:

- Computing the suffix array and the LCP array during the decompression. Hence, it is not necessary to decompress all the text to compute the suffix and LCP arrays. This approach demonstrated to be superior to computing the data structures from scratch.
- Extracting substrings in competitive time/space regarding its major competitors once the grammar is augmented with additional information.

When confronted with popular compressors and modern compressors and extractors, regarding several criteria, GCIS turn out to be a practical option for compressing and extracting from repetitive sources.

1.2 Structure of the Document

- Chapter 2 describes the concepts needed to understand the underlying mechanisms of the proposed approach.
- Chapter 3 focuses on methods capable of dealing with repetitive texts. These are our objects of comparison.
- Chapter 4 presents the GCIS approach and how to augment it to support extraction of substrings and construction the suffix and LCP arrays during the decompression.
- Chapter 5 show the experimental setup, including the *copora* descriptions, and the experimental results and discussion regarding GCIS and the state-of-the-art compressors.
- Chapter 6 reports our final considerations, the improvements related to the GCIS approach of grammar compression, and possible future works.

Chapter 2

Background

2.1 Basic Concepts

Definition 1 (Alphabet)**.** *An alphabet is a finite set* $\Sigma = \{\alpha_1, \ldots, \alpha_\sigma\}$ *with* $\sigma = |\Sigma|$*. This set is totally ordered given the relation* $<$*, such that* $\alpha_1 < \alpha_2 < \ldots < \alpha_\sigma$*, with* $<$ *being called lexicographical order.*

Definition 2 (String)**.** *A string* $S = S[1] \ldots S[n] \in \Sigma^*$ *is a finite sequence of symbols of* Σ*. The empty string is denoted by* ε *.*

Definition 3 (String length)**.** *The length of a string* $S = S[1]S[2] \ldots S[n]$ *is the number of symbols is* S *and is denoted by* $|S| = n$*. Specially,* $|\varepsilon| = 0$*.*

Definition 4 (Σ^k and Σ^*)**.** Σ^k *defines the set of all strings of length* k *over the alphabet* Σ*. The set of all finite strings is given by* $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$

Definition 5 (Substring)**.** *If* $1 \leq i \leq j \leq n$*, a substring* $S[i, j]$ *is a string with the symbols* $S[i], \ldots, S[j]$ *of a string* S*. Otherwise,* $S[i, j] = \varepsilon$*.*

Definition 6 (Occurrence)**.** *A string* R *is said to occur in* S*, with* $|S| = n$*, when there are indices* $1 \leq i \leq j \leq n$*, such that* $R = S[i, j]$*.*

Definition 7 (Concatenation)**.** *The concatenation of strings* R *and* S *is the juxtaposition of symbols of* R *and* S *in order. We denote the concatenation of strings with the dot operator* $(\cdot)$*. Thus, given* $|R| = m$ *and* $|S| = n$*, we have* $R \cdot S = R[1] \ldots R[m]S[1] \ldots S[n]$*.*

Definition 8 (Restricted Alphabet)**.** *We call restricted alphabet of a string* $S \in \Sigma^*$*,* $|S| = n$*, also denoted by* $\Sigma(S)$*, the set:*

$$\Sigma(S) = \bigcup_{i=1}^{n} \{S[i]\}$$

In other words, $\Sigma(S)$ is the set of symbols that occur in S.

Notation 1 (Text)**.** *T, the text, is a string of length n, over an ordered alphabet Σ that has size σ. We assume that our alphabet Σ has an integer size, but limited to n, that is, $1 \leq \sigma \leq n$. For convenience, we assumed that T always ends with a special symbol $T[n] = \$$, which is not present elsewhere in T and lexicographically precedes every symbol in $T[1, n-1]$.*

Notation 2 (Suffix)**.** *A suffix of a string S is a substring of the form $S[i, n]$, also denoted by S_i.*

Definition 9 (Range-minimimum-query)**.** *Let V be a sequence of length n. We define the range-minimum-query over an interval $[l, r] \subseteq [1, n]$ of this sequence as:*

$$\mathsf{RMQ}_V(l, r) = \min\{V[k] \mid l \leq k \leq r\}$$

Notation 3 ($\lg x$)**.** *From now on we denote $\log_2 x$ as $\lg x$.*

2.2 Suffix Trees and Suffix Arrays

Suffix Trees are a key data-structure for solving several string processing problems in feasible time [Gusfield, 1997]. According to Gusfield [1997], a suffix tree over the text T has n leaves, one for each T_i, $1 \leq i \leq n$; each internal node, excluding the root, has at least two children; and each edge is labeled with a substring of T. There are no edges out of the same node with the same starting symbol and the concatenation of the edges labels of the path from the root to a leaf spells a unique suffix of T.

The main concern involving suffix trees is their space consumption. Despite requiring $\Theta(n \lg n)$ bits to encode the suffixes, the hidden constant is unpractical for bigger texts. Even with a cautious implementation, the suffix tree representation would require ≈ 15 times the input text size [Kurtz, 1999], making it prohibitive for large texts such as a collection of genomes.

The suffix array (SA) [Gonnet et al., 1992; Manber and Myers, 1993] of a string T is an array of distinct integers in the range $[1, n]$ that gives the lexicographic order of all suffixes of T, such that $T_{\mathsf{SA}[1]} < T_{\mathsf{SA}[2]} < \ldots < T_{\mathsf{SA}[n]}$. It can be viewed as a space-efficient alternative to a suffix tree and when enhanced with additional information, it can solve the same problems within the same time complexity of a suffix tree [Abouelhoda et al., 2004; Ohlebusch, 2013].

The suffixes starting with the same symbol $c \in \Sigma$ form a c-bucket in SA, hence, each c-bucket defines a contiguous interval in SA containing the suffixes that start with the

same symbol. The head and the tail of a c-bucket refer to the first and the last position of the c-bucket in SA.

The suffix array for the text T = AGCCTAAGCCTAAGTAAAG$ is depicted in Figure 2.1. In this example, the A-bucket is defined by the interval $[2, 9]$.

Table 2.1: Suffix Array, LCP array and BWT transform for the text T = AGCCTAAGCCTAAGTAAAG$.

i	SA$[i]$	LCP$[i]$	BWT$[i]$	$T_{SA[i]}$
1	20	0	G	$
2	16	0	T	AAAG$
3	17	2	A	AAG$
4	6	3	T	AAGCCTAAGTAAAG$
5	12	3	T	AAGTAAAG$
6	18	1	A	AG$
7	1	2	$	AGCCTAAGCCTAAGTAAAG$
8	7	8	A	AGCCTAAGTAAAG$
9	13	2	A	AGTAAAG$
10	3	0	G	CCTAAGCCTAAGTAAAG$
11	9	6	G	CCTAAGTAAAG$
12	4	1	C	CTAAGCCTAAGTAAAG$
13	10	5	C	CTAAGTAAAG$
14	19	0	A	G$
15	2	1	A	GCCTAAGCCTAAGTAAAG$
16	8	7	A	GCCTAAGTAAAG$
17	14	1	A	GTAAAG$
18	15	0	G	TAAAG$
19	5	3	C	TAAGCCTAAGTAAAG$
20	11	4	C	TAAGTAAAG$

Longest Common Prefix

The length of the longest common prefix (LCP) of two strings X and Y in Σ^* is denoted $\texttt{lcp}(X, Y)$. The LCP array of T is an array of integers that stores the lcp value between consecutive suffixes in SA, such that $\mathsf{LCP}[i] = \texttt{lcp}(T_{\mathsf{SA}[i-1]}, T_{\mathsf{SA}[i]})$, for $1 < i \leq n$, and we define $\mathsf{LCP}[1] = 0$. The LCP array can be used to enhance the suffix-array and simulate a suffix-tree traversal over the text [Abouelhoda et al., 2004].

Figure 2.1 displays the LCP values for the chosen text in the third column.

Burrows-Wheeler Transform

The Burrows-Wheeler transform, also known as BWT [Burrows and Wheeler, 1994], can be seen as the last column of a virtual matrix containing all the cyclical suffixes of a text in lexicographical order. A cyclical suffix starting at position i corresponds to $T[i, n] \cdot T[1, i-1]$. This transform has a close relation to the suffix array, which can be stated as:

$$\mathsf{BWT}[i] = \begin{cases} T[\mathsf{SA}[i]-1], & SA[i] > 1 \\ \$, & \text{otherwise} \end{cases} \tag{2.1}$$

The BWT is a permutation of the original text that favors runs of equal characters, which eases the compression. It is possible to compute the BWT and the SA from each other in linear time. Moreover, the BWT is reversible, *i.e.*, it is possible to recover the text from the BWT in linear time. The fourth column of Table 2.1 shows the BWT for the corresponding text.

2.3 Suffix Array Construction by Induced Suffix Sorting

The suffix array construction method by induced suffix sorting (SAIS) of Nong et al. [2009] builds on the induced suffix sorting technique introduced by previous algorithms [Itoh and Tanaka, 1999; Ko and Aluru, 2003]. Induced suffix sorting consists in deducing the order of unsorted suffixes from a smaller set of already ordered suffixes.

The next definitions classify suffixes and symbols of strings.

Definition 10 (S-type)**.** *For any string T, $T_n = \$$ has type S. A suffix T_i is an S-type suffix, or simply an S-suffix, if $T_i < T_{i+1}$.*

Definition 11 (L-type)**.** *A suffix T_i is an L-type suffix, or simply an L-suffix, if $T_i > T_{i+1}$.*

The suffixes can be classified in linear time by scanning T once from right to left, so that the type of each suffix is stored in a bitmap of size n.

Note that within any c-bucket, the L-suffixes precede the S-suffixes.

Further, the classification of suffixes is refined as follows:

Definition 12 (LMS-type)**.** *Let T be a string. Then T_i is an LMS-type suffix, or simply an LMS-suffix, if T_i is an S-suffix and T_{i-1} is an L-suffix.*

Definition 13 (Symbol type)**.** *A symbol $T[i]$ has the same type as the suffix T_i.*

Nong et al. [2009] showed that the order of the LMS-suffixes is enough to induce the order of all suffixes. This is the basis of the SAIS algorithm.

2.3.1 SAIS framework

1. Sort the LMS-suffixes. This step is explained later.

2. Insert the LMS-suffixes into the tail of their respective c-buckets in SA, without changing their relative order. Now SA contains the LMS-suffixes positions, in sorted order, on the end of each c-bucket. The remaining values of SA are initialized with a sentinel $\perp$ value.

3. Induce L-suffixes by scanning SA from left to right: for each suffix $\mathsf{SA}[i] \neq \perp$, if $T[\mathsf{SA}[i]-1]$ is L-type, insert $\mathsf{SA}[i]-1$ into the head of its c-bucket and increment the head pointer.

4. Induce S-suffixes by scanning SA from right to left: for each suffix $\mathsf{SA}[i] \neq \perp$, if $T[\mathsf{SA}[i]-1]$ is S-type, insert $\mathsf{SA}[i]-1$ into the tail of its c-bucket and decrement the tail pointer.

In order to sort the LMS-suffixes in Step 1, T is divided (factorized) into LMS-substrings.

Definition 14. *$T[i,j]$ is an LMS-substring if both T_i and T_j are LMS-suffixes, but no suffix between i and j has LMS-type. $T[n,n]$, the last suffix, is also an LMS-substring.*

Let $r_1^1, r_2^1, \ldots, r_{n^1}^1$ be the n^1 LMS-substrings of T read from left to right. A modified version of SAIS is applied to sort the LMS-substrings. Starting from Step 2, T is scanned (right-to-left) and each new LMS-suffix starting with c is inserted (bucket-sorted) at the tail of its c-bucket. Steps 3 and 4 work exactly the same. At the end, the LMS-substrings are sorted and the beginning positions of each LMS-substring are stored in their corresponding c-buckets in SA.

2.3.2 Naming

A *name* v_i^1 is assigned to each LMS-substring r_i^1 according to its lexicographical rank in $[1, \sigma^1]$, such that $v_i^1 \leq v_j^1$ iff $r_i^1 \leq r_j^1$, where σ^1 is the number of different LMS-substrings in T. In order to compute the names, each consecutive LMS-substrings in SA, say r_i^1 and r_{i+1}^1, are compared to determine if either $r_i^1 = r_{i+1}^1$ or $r_i^1 < r_{i+1}^1$. In the former case v_{i+1}^1 is set to v_i^1, whereas in the latter case v_{i+1}^1 is set to $v_i^1 + 1$. This procedure may be sped up by comparing the LMS-substrings first by symbol and then by type, with L-type symbols being smaller than S-type symbols in case of ties [Nong et al., 2011].

2.3.3 Recursive call

A new (reduced) string $T^1 = v_1^1 \cdot v_2^1 \cdots v_{n^1}^1$ is created, whose length n^1 is at most $n/2$, and the alphabet size σ^1 is integer. If every $v_i^1 \neq v_j^1$ for $1 \leq i < j \leq n^1$, then all LMS-suffixes are already sorted.

Otherwise, SAIS is recursively applied to sort all the suffixes of T^1. Nong et al. [2009] showed that the relative order of the LMS-suffixes in T and the order of the respective suffixes in T^1 are the same. Therefore, the order of all LMS-suffixes can be determined by the result of the recursive algorithm.

2.3.4 Running Example

Figures 2.1 to 2.9 show a running example for the suffix array construction of the text T = AGCCTAAGCCTAAGTAAAG\$. They describe the following steps:

1. The suffixes are classified according to their types (Figure 2.1).
2. Each LMS-Suffix is placed at the end of its c-bucket (Figure 2.2).
3. The L-type suffixes are induced from the LMS-type suffixes in a left-to-right fashion (Figure 2.3).
4. The S-type suffixes are induced from the L-type suffixes in a right-to-left fashion (Figure 2.4). Now the LMS-substrings are sorted.
5. We name every LMS-substring and recurse by replacing the LMS-substrings of T into its name (Figure 2.5).
6. By solving the subproblem, we have the correct order of the LMS-Substrings (Figure 2.6).
7. Each LMS-Suffix, is placed at the end of its c-bucket (Figure 2.7).
8. Steps 3 and 4 are repeated to finally compute SA (Figures 2.8 and 2.9).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	G	C	C	T	A	A	G	C	C	T	A	A	G	T	A	A	A	G	\$
	S	L	S	S	L	S	S	L	S	S	L	S	S	S	L	S	S	S	L	S
			*			*			*			*				*				*

Figure 2.1: Classification of the suffixes according to its types. The LMS suffixes are marked with a '*'.

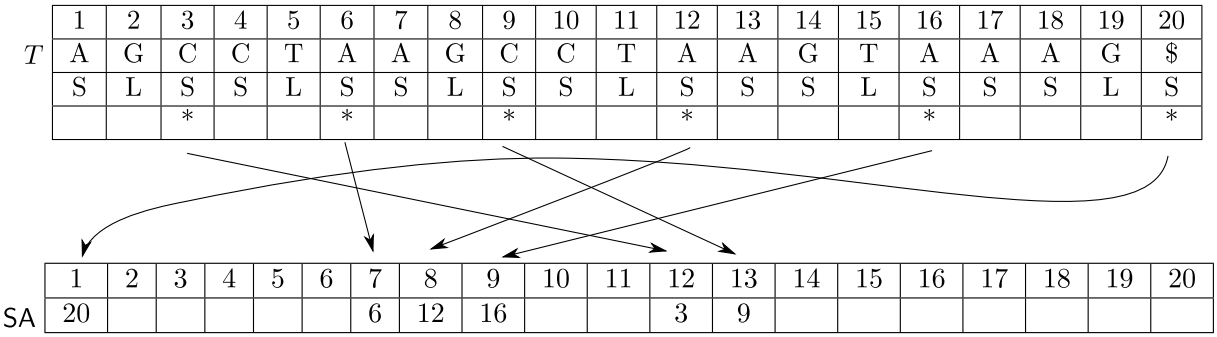


Figure 2.2: The LMS suffixes are inserted at the end of its c-bucket.

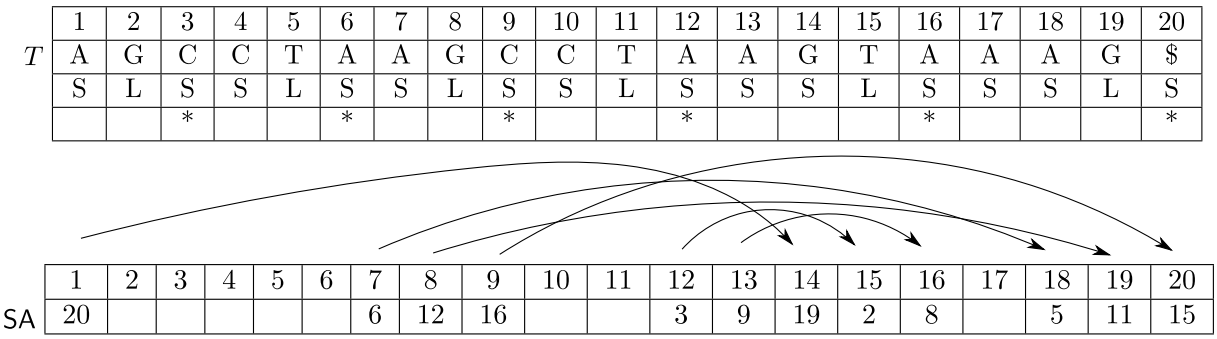


Figure 2.3: The L-type suffixes are induced from the LMS-type and L-type suffixes at the beginning of its c-bucket.

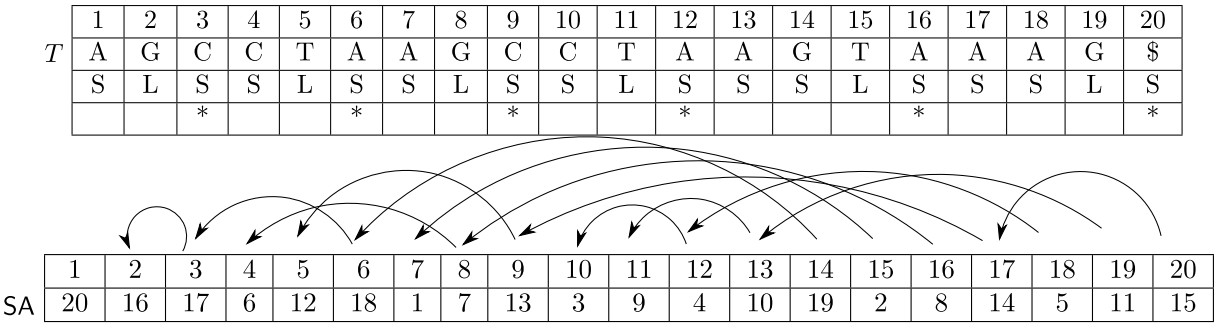


Figure 2.4: The S-type suffixes are induced from the L-type and S-type suffixes at the end of its c-bucket.

2.3.5 Time and Space Complexities

There are no consecutive LMS-type suffixes in T, hence, the subproblem must have at most size $\lfloor \frac{n}{2} \rfloor$. Since the induction steps and the bucket-sort cost $\Theta(n)$, the recurrence relation of SAIS can be modeled as:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & \text{otherwise} \end{cases} \tag{2.2}$$

Which evaluates to $\Theta(n)$ optimal time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	G	C	C	T	A	A	G	C	C	T	A	A	G	T	A	A	A	G	$
	S	L	S	S	L	S	S	L	S	S	L	S	S	S	L	S	S	S	L	S
			*			*			*			*				*				*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA	20	16	17	6	12	18	1	7	13	3	9	4	10	19	2	8	14	5	11	15
	*	*		*	*					*	*									

$T[20, 20] = \$ \mapsto 1$
$T[16, 20] = AAAG\$ \mapsto 2$
$T[6, 9] = AAGC \mapsto 3$
$T[12, 16] = AAGTA \mapsto 4$
$T[3, 6] = CCTA \mapsto 5$
$T[9, 12] = CCTA \mapsto 5$

	1	2	3	4	5	6
T^1	5	3	5	4	2	1

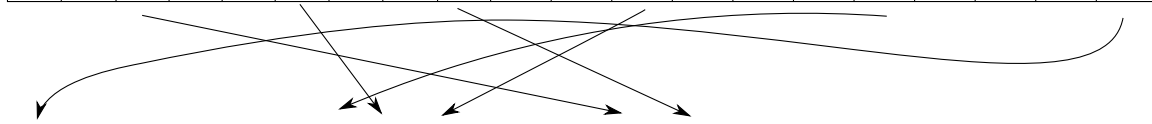
Figure 2.5: We name every LMS-substring and replace them in T for its name, generating T^1. Since $\sigma^1 < |T^1|$, we solve the problem recursively for T^1.

	1	2	3	4	5	6
T^1	5	3	5	4	2	1

	1	2	3	4	5	6
SA^1	6	5	2	4	1	3

Figure 2.6: After solving the problem recursively for T^1, we obtain the suffix array SA^1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	G	C	C	T	A	A	G	C	C	T	A	A	G	T	A	A	A	G	$
	S	L	S	S	L	S	S	L	S	S	L	S	S	S	L	S	S	S	L	S
			*			*			*			*				*				*



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA	20						16	6	12			3	9							

Figure 2.7: The LMS suffixes are inserted at the end of its c-bucket according to the SA^1 order.

For the space, since it is possible to reuse the SA array during the recursion to save space, we have a working space (excluding T and SA) of $0.5n \lg n + n + O(1)$ bits when the alphabet is constant and $n \lg n + n + O(1)$ bits when the alphabet is integer (Corollary

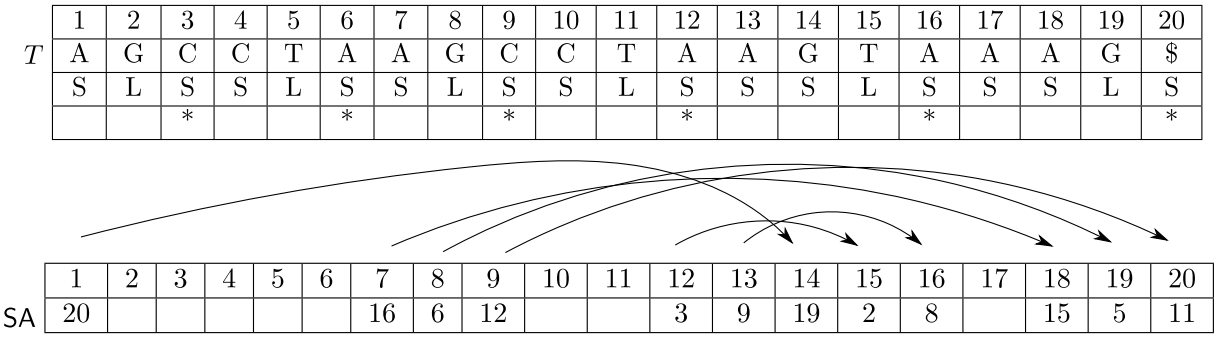


Figure 2.8: The L-type suffixes are induced from the LMS-type and L-type suffixes at the beginning of its c-bucket.

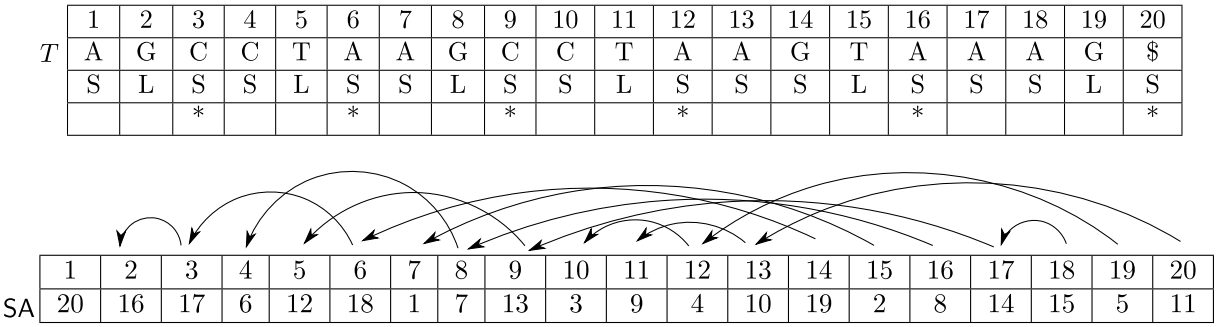


Figure 2.9: The S-type suffixes are induced from the LMS-type and L-type suffixes at the beginning of its c-bucket.

3.1 of [Nong et al., 2009]). Further improvements showed that is possible to guarantee $O(\sigma \lg n)$ of working space [Nong, 2013].

2.4 LCP Array Induction

It is possible to compute the LCP array in linear time as a byproduct of the final induction steps of the SAIS method [Fischer, 2011; Louza et al., 2021]. Whenever two suffixes, say $\mathsf{SA}[i]$ and $\mathsf{SA}[j]$, with $i < j$, induce two L-type adjacent entries $\mathsf{SA}[k-1]$ and $\mathsf{SA}[k]$, we have $\mathsf{LCP}[k] = \mathtt{lcp}(T_{\mathsf{SA}[i]}, T_{SA[j]})$. Symmetrically, whenever two S-type suffixes, say $\mathsf{SA}[i]$ and $\mathsf{SA}[j]$, with $i > j$, induce two adjacent entries $\mathsf{SA}[k]$ and $\mathsf{SA}[k+1]$, we have $\mathsf{LCP}[k] = \mathtt{lcp}(T_{\mathsf{SA}[i]}, T_{\mathsf{SA}[j]})$. Thus, the key to induce the LCP values is to compute the longest common prefix between two suffixes, which can be translated by a range-minimum query $\mathsf{RMQ}_{\mathsf{LCP}}(i+1, j)$ for the L-type adjacent suffixes or by $\mathsf{RMQ}_{\mathsf{LCP}}(j+1, i)$, for the S-type adjacent entries.

However, there is a small issue with this description: it does not specify how to compute the suffixes in a L/S-frontier, that is, the LCP between the last L-type and the first S-type

suffixes of a c-bucket. These LCP values must be computed by explicit comparisons during the induction steps. Overall, the comparisons take linear time (Lemma 1 [Fischer, 2011]).

Since we compute the LCP values of the L-type and S-type suffixes during the induction steps, we must know previously the LCP between the LMS-suffixes. This can be done by:

1. Computing the `lcp` between every consecutive LMS-substrings during the naming step.

2. Employing a variation of the Φ-algorithm, *cf.* [Kärkkäinen et al., 2009], to compute the `lcp` between every consecutive LMS-suffixes in linear time resorting to the precomputed `lcp` values in the previous step to speed up the process. Note that the `lcp` between LMS-suffixes might be greater than the `lcp` between LMS-substrings.

2.4.1 Time and Space Complexities

The operation which dominates the time complexity is the RMQ over the LCP array. By using an array $M[1, \sigma]$ that keeps track of the minimum value, $M[c]$, induced in each c-bucket it is possible to answer these queries in constant time. However, every time a $M[c]$ entry is updated to a value $\mathsf{LCP}[i]$, all other entries of M that are greater than $\mathsf{LCP}[i]$ must be updated to $\mathsf{LCP}[i]$. This process takes $\Theta(\sigma)$ time and hence, the LCP array is calculated in $\Theta(n\sigma)$ time, which is linear for constant alphabets. By using a more refined data structure to answer the RMQ queries, such as Fischer and Heun [2007]'s, it is possible to answer each query in $\Theta(1)$ time and thus obtaining a $\Theta(n)$ time algorithm.

If the M array is used an additional σ words of working space are needed.

2.4.2 Running Example

Figures 2.10 to 2.13 depict the LCP computation process. They describe the following steps:

1. The LCP values of the LMS-substrings are computed during the naming step.

2. After the recursive call, all LMS-suffixes are sorted, and by employing the Φ-algorithm, the LCP values of the LMS-suffixes are calculated in linear time. This process is sped-up by relying on the precomputed values in the last step.

3. During the induction of the L-type suffixes with a left-to-right scan, the LCP values of such suffixes are calculated. If any, the LCP values lying on a L/S-frontier must also be computed.

4. During the induction of the S-type suffixes with a right-to-left scan, the LCP values of such suffixes are calculated. If any, the LCP values lying on a L/S-frontier must also be computed.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	G	C	C	T	A	A	G	C	C	T	A	A	G	T	A	A	A	G	$
	S	L	S	S	L	S	S	L	S	S	L	S	S	S	L	S	S	S	L	S
			*			*			*			*				*				*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA	20	16	17	6	12	18	1	7	13	3	9	4	10	19	2	8	14	5	11	15
LCP	0	0		2	3					0	4									
	*	*		*	*					*	*									

$T[20, 20] = \$ \mapsto 1$
$T[16, 20] = AAAG\$ \mapsto 2$
$T[6, 9] = AAGC \mapsto 3$
$T[12, 16] = AAGTA \mapsto 4$
$T[3, 6] = CCTA \mapsto 5$
$T[9, 12] = CCTA \mapsto 5$

T^1	1	2	3	4	5	6
	5	3	5	4	2	1

Figure 2.10: The LCP values of the LMS-substrings are computed directly during the naming procedure.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	G	C	C	T	A	A	G	C	C	T	A	A	G	T	A	A	A	G	$
	S	L	S	S	L	S	S	L	S	S	L	S	S	S	L	S	S	S	L	S
			*			*			*			*				*				*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA	20						16	6	12			3	9							
LCP	0						0	2	3			0	6							

Figure 2.11: By employing the Φ-algorithm, the LCP values of the LMS-suffixes are computed.

2.5 Entropy and Repetitiveness Measures

According to Navarro [2021], the empirical entropy over a finite and particular sequence S of length n can be defined as:

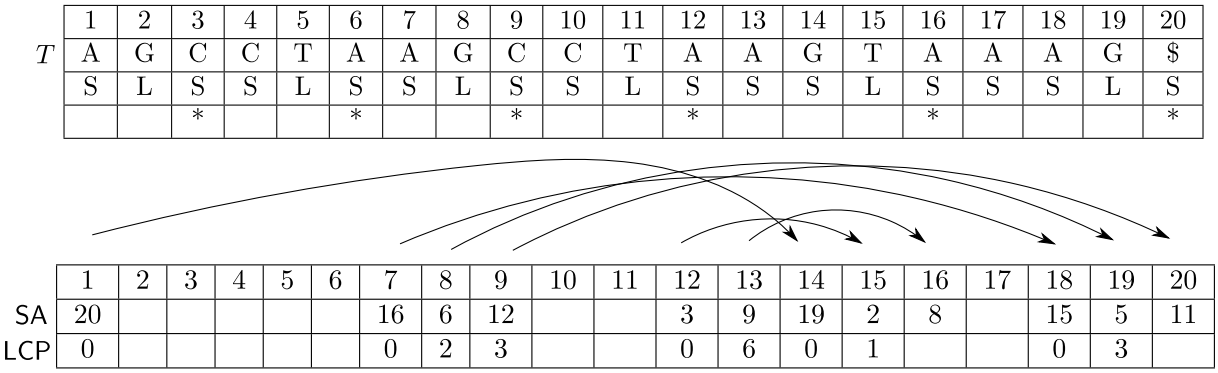


Figure 2.12: The LCP values of the L-type suffixes are computed during the induction step of those suffixes.

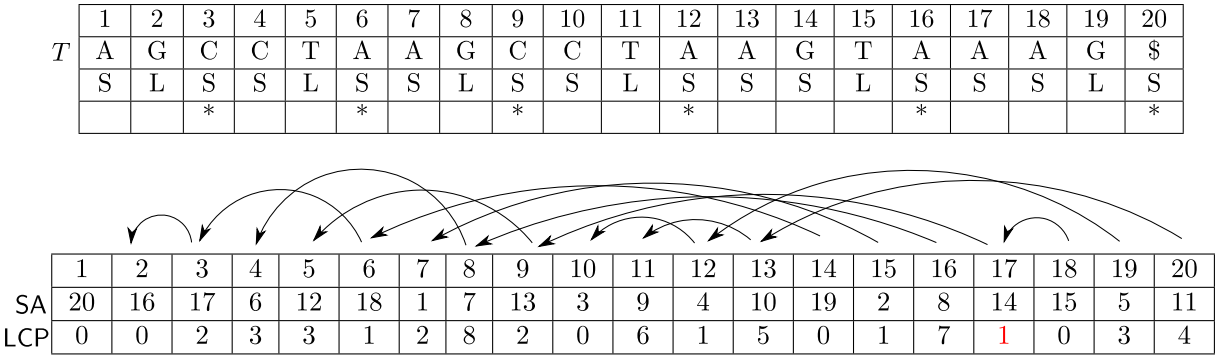


Figure 2.13: The LCP values of the S-type suffixes are computed during the induction step of those suffixes. The value in red is computed explicitly, since it is located in a L/S-frontier.

$$H(S) = \sum_{a \in \Sigma(S)} \frac{n_a}{n} \lg \frac{n}{n_a} \tag{2.3}$$

Where n_a stands for the frequency of the symbol a in S. This is also known as zeroth order empirical entropy, also denoted by $H_0(S)$, and it serves as a lower bound to any statistical compressor that exploits skewed frequencies, *i.e.* which encodes each symbol $S[i]$ with $-\log \frac{n_i}{n}$ bits [Ferragina and Manzini, 2005].

If we also consider the context of the substring of length k preceding each symbol in S, we can define the k-th order empirical entropy as:

$$H_k(S) = \sum_{C \in \Sigma^k} \frac{n_C}{n} \cdot H_0(S_C) \tag{2.4}$$

Where S_C stands for the sequence of symbols following occurrences C in S and $n_C = |S_C|$ [Navarro, 2021]. The k-th order entropy can be used as a lower bound to any statistical compressor that encodes each symbol with a code that depends on the symbol itself and the context of size k immediately preceding this symbol [Ferragina and Manzini,

2005]. It always follows that $H_k(S) \geq H_{k+1}(S)$ for any $k \geq 0$.

In spite of its properties, the empirical entropy does not capture repetitiveness, since $H_k(S \cdot S) \geq H_k(S)$. Hence, statistical compressors cannot fully explore the repetitiveness of texts. Chapter 3 encompasses other several methods capable of exploiting the redundancy over highly repetitive texts and the associated metrics.

2.6 Compact Sequence Representations

This section considers the notation established in [Belazzougui and Navarro, 2015]. Assume S a sequence of length n over an alphabet Σ. The following operations over S are defined:

- ACCESS(S, i), or simply $S[i]$: retrieve the i-th symbol of S.
- RANK$_a(S, i)$: count the number of occurrences of a $S[1, i]$.
- SELECT$_a(S, j)$: compute the position where the j-th a appears in S. This is only defined when $1 \leq j \leq n_a$, being n_a the frequency of a in S.

When the alphabet is binary, *i.e.*, $\Sigma = \{0, 1\}$, the sequence is also referred to as a bitmap and all the operations can be supported in constant time [David Clark, 1996; González et al., 2005]. There are some methods that achieve a compressed representation maintaining constant query times per operation when the frequency of symbols is skewed [Raman et al., 2007].

For integer sequences, Wavelet Trees or Wavelet Matrices can be employed and further compressed if they are given a shape of a Huffman Tree [Claude et al., 2015; Grossi et al., 2003; Mäkinen and Navarro, 2005; Navarro, 2014]. A lower bound of RANK in $\Omega(\log \log_w \sigma)$ time for any structure using $O(nw^{O(1)})$ bits, w being the width in bits of a word in a RAM machine, was obtained in [Belazzougui and Navarro, 2015] while maintaining constant time for SELECT and $\omega(1)$ time for ACCESS or *vice-versa* [Navarro, 2016].

2.7 Compressed Indices

During the last two decades, several compressed indices were developed. More than indices, they are *self-indices*, *i.e.*, they can recover the text from a compressed representation without having to store it.

Grossi and Vitter [2000, 2005] described a compressed suffix array data structure that represented its information in $O(n \lg \sigma)$ bits by compressing the increasing sequence Ψ

and employing SA sampling. With the compressed Ψ function it was possible to navigate in the virtual suffix array and compute the entries of SA that were not sampled.

Ferragina and Manzini [2000, 2005] developed what is called the FM family of indices, which compresses the BWT and uses a sampling technique on SA to represent the suffix-array in a compressed form. Similarly to the compressed suffix array, the FM-indices rely on a function LF(i) that works comparably as $\Psi(i)$ to recover entries from a virtual suffix-array. As a matter of fact $\Psi(\mathrm{LF}(i)) = \mathrm{LF}(\Psi(\mathrm{i})) = i$. In a *specimen* of this family, they obtained an index that represented the suffix array in $5nH_k(T) + o(n)$ bits. The approach of Mäkinen and Navarro [2005] works similarly as the FM-indices but compresses the BWT runs to achieve k-th order entropy.

The LZ-index (*cf.* [Navarro, 2004]) builds succinct tries of the LZ78 phrases and its reverses to achieve a representation bounded by $4nH_k(T) + o(n \lg \sigma)$ for $k = o(\log_\sigma n)$.

It is also feasible to build compressed suffix trees that, built on top of the compressed SA and LCP information, can emulate any query on a plain suffix-tree while retaining a compact representation [Abeliuk et al., 2013; Nunes and Ayala-Rincón, 2013; Sadakane, 2007]. This is made possible by representing the suffix tree topology succinctly, explicitly or implicitly.

All the aforementioned indices are bounded by the empirical entropy and hence cannot explore all the compressibility of a very repetitive text. Other approaches based on different compression schemes, such as grammar compression, were proposed to represent the information in a more space-efficient way while retaining the ability to perform queries over the compressed information.

2.8 Integer Encoding

We cover various techniques to encode sequences of integers when most of them are expected to be small. Some techniques allow us to directly access any integer in the sequence.

2.8.1 Simple8b

The `Simple8b` scheme, proposed by Anh and Moffat [2010], encodes a sequence of small integers in a 64-bit word using the number of bits required by the largest integer. It identifies a word with a 4-bit tag called *selector*, which specifies the number of integers encoded in the rest of the word and the width of such integers. `Simple8b` also has specific selectors for a run consisting of zeroes. If a run of 240 or 120 zeros is encountered, it can be represented with a single 64-bit word.

Table 2.2 describes the number of integers packed (size) and the width of such integers according to each selector value. It also can be observed in this table that the number of wasted bits due to internal fragmentation only occurs when selector values are in $\{0, 1, 8, 9\}$. For instance, suppose that we have to encode the integer sequence $(51, 17, 35, 53, 59, 20, 37, 38, 13, 9)$. Since each value uses at most 6 bits, we can encode all the 10 integers, each one with fixed width of 6 bits, under the selector value 7.

Table 2.2: `Simple8b` scheme (Adapted from [Anh and Moffat, 2010]). For each **selector value**, we can pack in a single 64-bit word **size** integers with **width** bits each while wasting **wasted bits** due to internal fragmentation.

Selector Value	Width	Size	Wasted Bits
0	0	240	60
1	0	120	60
2	1	60	0
3	2	30	0
4	3	20	0
5	4	15	0
6	5	12	0
7	6	10	0
8	7	8	4
9	8	7	4
10	10	6	0
11	12	5	0
12	15	4	0
13	20	3	0
14	30	2	0
15	60	1	0

2.8.2 Directly Addressable Codes

The Directly Addressable Codes (DAC), proposed by Brisaboa et al. [2013], allow efficient retrieval of any given value $A[i]$ from an array of integers while encoding such integers compactly. Let b be an integer parameter and l_i be the width (number of bits) of $A[i]$. This encoding splits each $A[i]$ into $k = \lceil l_i/b \rceil$ blocks $V^1[i]$, $V^2[i]$,..., $V^k[i]$ of b bits each. A bit $B^l[i] = 1$ is associated with a block $V^l[i]$ if $l < k$, that is, $V^l[i]$ is not the last block of $A[i]$. Otherwise, $B^l[i] = 0$. Then a layered data structure is constructed in such a way that the l-th layer contains two bitmaps: the first bitmap is the concatenation of blocks V^l, whereas the second bitmap is the concatenation of the bits B^l, each one in correspondence with a block of V^l.

To retrieve any given value $A[i]$, one must first recover $B^1[i]$ and check if its value is zero. If so $A[i]$ equals $V^1[i]$, otherwise, it is necessary to proceed recursively to the j-th entry of the next layer, where $j = \text{RANK}_1(B^1, i)$, and append the result of the recursive call to $V^1[i]$. RANK_1 can be computed in constant time by using auxiliary data structures on top of B, as exposed in Section 2.6. Algorithm 1 schematizes the recovery of an entry $A[i]$ under DAC encoding when called with parameters $\text{DAC}(V, B, i, 1)$.

Algorithm 1: $\text{DAC}(V, B, i, l)$.

Input: V,B,i,l
Output: $A[i]$

```
1 if B^l[i] = 1 then
2     j ← RANK_1(B^l, i);
3     return BINARY-CONCAT(V^l[i], DAC(V, B, j, l + 1))
4 else
5     return V^l[i]
```

2.8.3 Elias-Fano Encoding

This format permits the encoding of a monotonically increasing sequence of n integers over the interval $[0, m-1]$ within $2n + n\lceil \lg \frac{m}{n} \rceil$ bits and allows the retrieval of any integer of such sequence in constant time [Vigna, 2013]. Each integer a_i is divided into two parts: u_i, the $\lceil \lg n \rceil$ most significant bits of a_i and l_i, the $\lceil \lg \frac{m}{n} \rceil$ remaining bits of a_i. The l_i values are concatenated in a single array of $n\lceil \lg \frac{m}{n} \rceil$ bits and each value a_i is classified in one of the n possible buckets. Then, the number of elements of each bucket is represented in a negated unary representation and such representations are concatenated in a bitmap B of $2n$ bits, n bits for each possible bucket and further n bits for every element a_i.

To retrieve the i-th value of the sequence of integers, one just needs to search for the position $k = \text{SELECT}_1(B, i)$, and append l_i to the binary representation of $k - i$. The position k can be retrieved in constant time using auxiliary data structures on top of B (*cf.* Section 2.6).

Chapter 3

Compressors for Repetitive Data

This chapter approaches several compression schemes, ranging from classical to modern, which are used when compressing highly repetitive data. Such approaches (or variations) are compared with the GCIS proposed approach (*cf.* Chapter 4) in Chapter 5.

3.1 LZ77

The authors in [Ziv and Lempel, 1977] presented a compression scheme known as LZ77 that employs a left-to-right scan to find the substring of minimum length from a position i, also called a phrase or factor, that does not occur in $T[1, i]$. Algorithm 2 summarizes this process.

Algorithm 2: LZ77-PARSE(T)

Input: $T[1, n]$
Output: LZ77 parse of T

```
1 i ← 1;
2 while i ≤ |T| do
3     Search for the minimum j > i which makes T[i, j] not occur in T[1, i − 1];
4     OUTPUT-PHRASE(S[i, j]);
5     i ← j + 1;
```

Definition 15 (Metric for LZ77 parsing)**.** *The number of LZ77 phrases of T is defined as $z(T)$, or simply z when the context is clear.*

Several compression schemes such as LZRR, LZD and ReLZ; and practical tools, *e.g.*, `zip`, `7zip` and `gzip`; arise from this compression scheme.

3.2 LZ78

The LZ78 scheme, differently from the LZ77, is a dictionary-based technique. It creates a dictionary D while reading the text in a left-to-right fashion and maintaining a read string S. While the read string S is found in the dictionary during this linear inspection, a new symbol is appended to this S. Whenever $S = S[1, k] \notin D$ a new factor $S[1, k-1] \cdot S[k]$ is created and S resets to the empty string [Ziv and Lempel, 1978]. More formally, we have the following definition.

Definition 16 (LZ78 factorization)**.** *The LZ78 factorization for a text T is a sequence of factors $f_1 \dots f_m$ such that $f_0 = \epsilon$ and $f_i = f_j \cdot c$ where:*

- *f_j is the longest prefix of $T[k, n]$, with $f_j \in \{f_k | 0 \leq k < i\}$.*
- *c is the character $T[k + |f_j|]$.*
- $k = 1 + \sum_{j=0}^{i-1} |f_i|$

Algorithm 3 describes the LZ78 scheme.

Algorithm 3: LZ78 scheme

Input: $T[1, n]$
Output: LZ78 factorization $f_1 \dots f_m$

```
1  S ← ϵ;
2  f_0 ← ϵ;
3  m ← 0;
4  D ← {f_0};
5  for i ← 1; i ≤ n; i++ do
6      if S · T[i] ∈ D then
7          S ← S · T[i]
8      else
9          m++;
10         f_m ← S · T[i];
11         D ← D ∪ {f_m};
```

The LZ78 factorization can be obtained in $O(n)$ time by using a trie data structure or hashing for storing each factor f_i.

Running example

Figure depicts a running example for the LZ78 factorization for the input text $T =$ abaababaabaababaaababa.

$$\underbrace{\text{a}}_{f_1}\ \underbrace{\text{b}}_{f_2}\ \underbrace{\text{aa}}_{f_3}\ \underbrace{\text{ba}}_{f_4}\ \underbrace{\text{baa}}_{f_5}\ \underbrace{\text{baab}}_{f_6}\ \underbrace{\text{bab}}_{f_7}\ \underbrace{\text{aaa}}_{f_8}\ \underbrace{\text{baba}}_{f_9}$$

$$\begin{aligned} f_0 &\to \epsilon & f_5 &\to f_4 \cdot a \\ f_1 &\to f_0 \cdot a & f_6 &\to f_5 \cdot b \\ f_2 &\to f_0 \cdot b & f_7 &\to f_4 \cdot b \\ f_3 &\to f_1 \cdot a & f_8 &\to f_3 \cdot a \\ f_4 &\to f_2 \cdot a & f_9 &\to f_7 \cdot a \end{aligned}$$

Figure 3.1: LZ78 factorization for $T = \text{abaababaabaababaaababa}$

3.3 Grammar Compression

Grammars describe a set of strings, *i.e.*, a formal language, by a set of production rules [Hopcroft et al., 2007]. It is natural to suggest that one can use grammars for compressing repetitive data.

Definition 17 (Context-free Grammar)**.** *A context-free grammar can be defined as a tuple* $G = (\Sigma, \Gamma, P, X_S)$*, where:*

- Σ *is the terminal alphabet of* G*;*
- Γ *is the set of variables, or non-terminal symbols,* $\Sigma \cup \Gamma = \emptyset$*;*
- $P \subseteq \Gamma \times (\Sigma \cap \Gamma)^*$ *is a finite set of production rules;*
- *and* $X_S \in \Gamma$ *is the start symbol.*

Definition 18 (Derivations)**.** *A production rule* $(X_i, \alpha_i) \in P$ *is also denoted* $X_i \to \alpha_i$*. In this case, it is said that* α_i *is derived from* X_i*. For strings* $S, R \in (\Sigma \cup \Gamma)^*$*, if* R *can be obtained from* S *by production rules in* P*, then* R *is derived from* S*. When* R *is obtained by a (possibly empty) sequence of derivations from* S*, then* R *is generated from* S*. If* S *is a non terminal, we say that* $\mathcal{G}(S) = R$*.*

Definition 19 (Grammar size)**.** $|G|$ *is defined as the total length of the strings on the right side of all production rules, that is:*

$$|G| = \sum_{(X_i, \alpha_i) \in P} |\alpha_i|$$

Given a string $T \in \Sigma^*$, the *grammar compression problem* is to find a grammar G that generates only T, such that G can be represented in less space than the original T. Given

that G grammar-compresses T, for $(X_i, \alpha_i) \in P$, $\mathcal{G}(X_i) = S$ is defined as the single string $S \in \Sigma^*$ that is generated from X_i. The language generated by G contains the unique string $\mathcal{G}(X_S) = T$. This notion can be extended further for a string of terminals and non-terminals S, such that:

$$\mathcal{G}(S) = \begin{cases} S, & S \in \Sigma \\ \{W_1 \cdot W_2 \cdot \ldots \cdot W_{|S|} \mid W_k \in \mathcal{G}(S[k]), 1 \leq k \leq |S|\}, & \text{otherwise} \end{cases}$$

When each $S[i]$, $1 \leq i \leq |S|$, generates a single sequence, the previous definition can be replaced by the concatenation of the strings generated by $S[i]$, $1 \leq i \leq |S|$:

$$\mathcal{G}(S) = \begin{cases} S, & S \in \Sigma \\ \mathcal{G}(S[1]) \cdot \mathcal{G}(S[2]) \cdot \ldots \cdot \mathcal{G}(S[|S|]), & \text{otherwise} \end{cases}$$

3.3.1 Grammar Compression and LZ77

Definition 20 (Minimum Grammar Size ($g(T)$))**.** *We call $g(T)$, or simply g when the context is clear, the size of the minimum context-free grammar that generates only the text T.*

The problem of computing g is NP-Complete [Charikar et al., 2005], but there are grammar algorithms that run in linear time and obtain good compression ratios. Especially the RePair method (*cf.* [Larsson and Moffat, 1999]) is considered the gold standard of grammar compression and thus often used as a baseline for comparing grammar compressors.

It holds that $g = \Omega(z \lg(\frac{n}{z}))$ [Claude et al., 2021], so we can expect LZ77-based compressors to represent the information more compactly than grammar compressors. Nevertheless, grammar compression remains interesting with respect to LZ77, being one of them the capability of extracting arbitrary symbols of the compressed text, which cannot be done in LZ77 efficiently. It is also possible to build self-indices based on grammars to support fast pattern matching [Claude and Navarro, 2011, 2012; Claude et al., 2021]. Besides, in practice, several grammar compression algorithms obtain reasonable compression ratios when compared with LZ77 based compressors [Larsson and Moffat, 1999; Maruyama and Tabei, 2013; Nunes et al., 2018].

3.4 RePair

The RePair heuristic, proposed by Larsson and Moffat [1999], grammar-compresses a given text by joining the most frequent pairs to create the production rules $X \rightarrow \text{CD}$, being X a newly introduced nonterminal and $\text{CD} \in (\Gamma \cup \Sigma)^2$, and recursing in the text

with CD replaced by X, thus generating a process of **Re**cursive-**Pair**ing. The Algorithm 4 describes such process.

Algorithm 4: RePair heuristic.

```
1 T' ← T;
2 Select symbols CD of T' such that CD is the most frequent pair;
3 if CD occurs only once then
4 |   X_S ← T';
5 else
6 |   Introduce the rule X → CD with a fresh nonterminal X;
7 |   Replace every occurrence of CD for X in T' and go to Step 2;
```

This whole process is bounded by $O(n)$ time and space by employing: a symbol array, which holds each symbol number and previous and next references for each active pair; a hash table for constant-time access to the pairs; and a priority queue that tracks the most frequent pairs.

The RePair heuristic is often considered a gold standard for grammar compressors for its practical value. It is often used as a baseline for grammar compressors and resides in the core of several applications [Claude and Navarro, 2010; Furuya et al., 2019; Ganczorz and Jez, 2017; Lohrey et al., 2013].

Running example

Figure 3.2 depicts a running example for RePair considering the input text $T =$ bananadadabanana. Each pair of lines describes that the most frequent pair, highlighted in red in the first member of such pair, is replaced by a fresh nonterminal until there is no pair with two or more occurrences.

3.4.1 Extraction

It is possible to support extraction of symbols in the RePair grammar if we known $|\mathcal{G}(X_i)|$ for each nonterminal X_i. The idea is to descend into the parse tree by computing the expansion length of each child to decide if one should recursively descend into the left or to the right child [Gagie et al., 2019]. Maruyama and Tabei [2013] make a better use of space, at the cost of being slower, by succinctly encoding the grammar parse tree using a post-order fashion.

$$T' : bananadadababanana$$
$$X_1 \to na$$
$$T' : baX_1X_1dadabaX_1X_1$$
$$X_2 \to ba$$
$$T' : X_2X_1X_1dadaX_2X_1X_1$$
$$X_3 \to da$$
$$T' : X_2X_1X_1X_3X_3X_2X_1X_1$$
$$X_4 \to X_1X_1$$
$$T' : X_2X_4X_3X_3X_2X_4$$
$$X_5 \to X_2X_4$$
$$T' : X_5X_3X_3X_5$$
$$X_S \to T'$$

Figure 3.2: RePair compression for $T =$ bananadadabanana.

3.4.2 BigRePair

The main problem with RePair implementations is the memory needed to build the grammar. By using the Rabin-Karp sliding hashing method to preprocess the text, it is possible to obtain a factorization in a way that long repeated substrings are likely to be parsed in the same way [Gagie et al., 2019]. By building the grammar upon the generated factorization, it is possible to greatly reduce the needed working space.

3.5 LZD

LZD stands for LZ Double-factor factorization and was proposed by Goto et al. [2015]. It is a compression scheme that is heavily inspired by the LZ78 parsing. Informally, when reading T in a left-to-right fashion, a new factor is created by composing two maximal previous factors (or single alphabet symbols). More formally, we have the following definition.

Definition 21 (LZD factorization [Goto et al., 2015])**.** *The LZD factorization for a text T is a sequence of factors $f_1 \dots f_m$ such that $f_0 = \epsilon$ and $f_i = f_{i_1} f_{i_2}$ for $1 \leq i \leq m$, where:*

- $k = 1 + \sum_{j=1}^{i-1} |f_j|$.
- *f_{i_1} is the longest prefix of $T[k, n]$, with $f_{i_1} \in \{f_j | 1 \leq j < i\} \cup \Sigma$.*

- f_{i_2} *is the longest prefix of* $T[k + |f_{i_1}|, n]$, *with* $f_{i_2} \in \{f_j | 0 \leq j < i\} \cup \Sigma$.

This factorization can be obtained with an incremental trie construction in $O(m)$ workspace and $O(n^2/\log_\sigma n)$ time or, by applying a modified Ukkonen's Suffix Tree algorithm, in $O(n)$ time and space.

Running example

Figure 3.3 shows a running example for the LZD factorization for the text $T =$ abaababaabaababaaababa. Every factor f_i is built maximally upon two previous factors (or symbols from $\Sigma \cup \epsilon$).

$$\underbrace{a|b}_{f_1}\ \underbrace{a|a}_{f_2}\ \underbrace{b|ab}_{f_3}\ \underbrace{aa|b}_{f_4}\ \underbrace{aab|ab}_{f_5}\ \underbrace{aa|ab}_{f_6}\ \underbrace{ab|a}_{f_7}$$

$$\begin{aligned} f_0 &\rightarrow \epsilon & f_4 &\rightarrow f_2 b \\ f_1 &\rightarrow ab & f_5 &\rightarrow f_4 f_1 \\ f_2 &\rightarrow aa & f_6 &\rightarrow f_2 f_1 \\ f_3 &\rightarrow b f_1 & f_7 &\rightarrow f_1 a \end{aligned}$$

Figure 3.3: LZD factorization for $T =$ abaababaabaababaaababa.

3.6 ReLZ

The ReLZ method relies on preprocessing the text by using the Relative Lempel Ziv (RLZ) approach, which parses the text by using a reference sequence from where the LZ77 factors from the text are generated. Such factors are interpreted as new symbols inducing a new sequence, which can be later LZ77 factored to identify repetitions that are separated by considerable distances. Since the preprocessed sequence tends to be much smaller than the original text, this approach does not suffer from memory consumption problems and obtains a very close approximation to the original LZ77 parse in $O(n)$ time [Kosolobov et al., 2020].

3.7 LZRR

Nishimoto and Tabei [2019] proposed a practical bidirectional parser based on LZ77 called LZRR, *i.e.*, a given parsed substring can be referenced by a previous or by a sequent occurrence. This work shows that the number of phrases generated by such parser is always less than the number of LZ77 phrases. Even though determining the smallest bidirectional parser is an NP-complete problem, it is empirically demonstrated that LZRR has practical value since it requires five percent fewer phrases than an LZ77 parse.

3.8 Repetitiveness Measures: a Case Study

Taking the real texts from Pizza-Chili Repetitive Corpus as a case study, it is possible to compare the impact of compressors that are capable of dealing with repetitive data with statistical compressors that are bounded by empirical entropy (cf. Section 2.5). In this case study, the compression ratio, *i.e.*, division of compressed text over original text sizes, is the subject of comparison. Table 3.1 shows the theoretical compression ratio for empirical entropy H_k for each corpus' text with $k \in \{0, \ldots, 8\}$, whereas Table 3.2 depicts compression ratio for LZ77 and Grammar-based compressors choosing 7-zip and RePair compressors for these measures, respectively. It is easy to see that empirical entropy cannot capture repetitiveness, being, in some cases, e.g. `cere` file, roughly 20 times worse than the chosen measures for LZ77 and grammar-based compressors.

Table 3.1: H_k for real texts from Pizza-Chili Repetitive Corpus [Ferragina and Navarro, 2005b].

	Compression Ratio (%)								
Text	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
cere	27.38	22.63	22.63	22.50	22.50	22.50	22.50	22.38	22.25
coreutils	68.38	51.25	35.88	23.88	17.00	12.88	10.13	8.00	6.50
einstein.de.txt	63.00	44.88	32.63	20.88	13.25	9.00	6.13	4.38	3.13
einstein.en.txt	62.00	46.38	33.38	21.13	13.25	9.00	6.50	4.75	3.50
Escherichia_Coli	25.00	24.75	24.50	24.38	24.25	24.25	24.13	24.13	23.88
influenza	24.63	24.13	24.13	24.00	23.88	23.50	22.00	18.63	13.25
kernel	67.25	50.50	36.63	25.75	19.25	15.13	12.13	9.63	7.75
para	26.50	23.50	23.38	23.38	23.38	23.38	23.25	23.25	23.13
world_leaders	43.38	24.38	17.25	11.63	7.63	5.13	4.00	3.50	3.13

Table 3.2: Compression ratio for LZ77 and Grammar based compressors for real texts from Pizza-Chili Repetitive Corpus.

	Compression Ratio (%)	
Text	7-zip	RePair
cere	1.05	1.86
coreutils	1.99	2.54
einstein.de.txt	0.11	0.16
einstein.en.txt	0.07	0.1
Escherichia_Coli	4.43	9.6
influenza	1.55	3.26
kernel	0.82	1.1
para	1.24	2.74
world_leaders	1.39	1.79

Chapter 4

A New Grammar Compression Approach by Induced Suffix Sorting

This chapter describes a novel grammar compression method that relies on the SAIS framework to create the grammar rules during the naming steps. This method is called GCIS. Theoretically, the described method brings a new approach for grammar compressors, and empirically, since it is built upon a fast and optimal suffix sorting strategy, it shows competitive results in compression and decompression speeds. By employing integer encoding techniques, it is possible to further improve the compressed representation and obtain competitive results regarding compression ratio. If augmented with additional data structures, this method is capable of performing extraction of substrings and exhibits a good space/time trade-off. Another feature of grammars generated by this method is that is also possible to compute the Suffix and LCP arrays during decompression without the need to store the original text. The discussion presented in this Chapter and the experimental results presented at Chapter 5 are also available in [Nunes et al., 2018] and [Nunes et al., 2022].

4.1 Compression

For computing the context-free grammar $G = (\Sigma, \Gamma, P, X_S)$ that generates only $T[1, n]$, we must modify the SAIS framework (*cf.* Section 2.3.1) as follows.

Consider the j-th recursion level. In Step 1, after the input string $T^j[1, n^j]$ is divided into the LMS-substrings $r_1^j, r_2^j, \ldots, r_{n^{j+1}}^j$ and named $v_1^j, v_2^j, \ldots, v_{n^{j+1}}^j$, a new rule $v_i^j \rightarrow r_i^j$ is created for each different LMS-substring r_i^j. Moreover, an additional rule $v_0^j \rightarrow v_0^{j-1} T^j[1, j_1 - 1]$ if $j > 0$ or $v_0^j \rightarrow T[1, j_1 - 1]$ if $j = 0$, with j_1 standing for the index of the leftmost LMS-type suffix of T^j, is created for the prefix of T^j that is not included in any LMS-substring. In this context, when $j = 0$, $n^0 = n$ and $T^0 = T$.

The algorithm is then called recursively with the reduced string $T^{j+1} = v_1^j \cdot v_2^j \cdots v_{n^{j+1}}^j$ as input as long as $\sigma^{j+1} < n^{j+1}$, that is, the LMS-substrings are not pairwise distinct. At the end, when $\sigma^\ell = n^\ell$, the last recursion level $j = \ell$ is reached, and the start symbol of X_S of G is created so that the initial production $X_S \to v_0^\ell \cdot v_1^\ell \cdot v_2^\ell \cdots v_{n^\ell}^\ell$ generates the original string $T[1, n]$.

The algorithm stops after computing X_S, since we are not interested in constructing the suffix array; Steps 2, 3 and 4 of SAIS are not executed. The recursive calls return to the top level and a grammar G that generates only $T[1, n]$ has been computed.

Since for each LMS-substring there is a unique v_i^j, there are no cycles in any generation. Further, there is only one path of derivations that from a string S generates a string S'. The consequence of this deterministic choice, for every derivation, is that $\mathcal{G}(X_i)$, for $X_i \in \Gamma$, is a fixed string of terminals. Figure 4.1 shows the grammar construction by GCIS.

Consecutive entries in the set of productions P are likely to share a common prefix, since the LMS-substrings are given lexicographically ordered by SAIS. Therefore, each rule $X_i \to \alpha_i \in P$ is encoded using two values $(l_i, \mathrm{s}(\alpha_i))$, such that l_i encodes $\texttt{lcp}\ (\alpha_{i-1}, \alpha_i)$, and the remaining symbols of α_i are given by $\mathrm{s}(\alpha_i) = \alpha_i[l_i + 1, |\alpha_i|]$. For each starting rule v_0^j, we define $l_i = 0$. This technique is known as Front-coding [Witten et al., 1999].

The computation of $(l_i, \mathrm{s}(\alpha_i))$ is performed with no additional cost with a slight modification in the naming procedure of SAIS. Consecutive LMS-substrings in SA, say r_{i-1}^j and r_i^j, are compared first by symbol until a mismatch is found, and then compared by type, to check if either $r_{i-1}^j = r_i^j$ or $r_{i-1}^j < r_i$. The symbol-wise comparison reveals $\texttt{lcp}(r_{i-1}^j, r_i^j)$ as well, so the resulting complexity is the same with a small slowdown in the running time.

Time complexity

As SAIS, GCIS runs in $\Theta(n)$ time, since each step of the modified SAIS runs in linear time and the length of the reduced string T^j is at most $|T^{j-1}|/2$.

Grammar size

The number of rules for a given string T^{k-1} is dependent of the number of distinct LMS-Substrings (σ^k) plus one: the additional rule for the prefix that does not belong to any LMS-substring of T^{k-1}. Let ℓ be the number of grammar levels, r_s^ℓ be the right-hand side of the starting symbol X_S and r_0^k the prefix of T^{k-1} that does not belong to any LMS-Substring, hence the grammar size can be computed as:

$$|r_S^\ell| + \sum_{i=1}^{\ell} \sum_{j=0}^{\sigma^i} |r_j^i|$$

Implementation details

Each non-terminal X_i is represented by a pair $\alpha_i = (l_i, \mathrm{s}(\alpha_i))$, as explained. The l_i values tend to be small and, considering the j-th recursion value, the sum of such values cannot be greater than n^j, since no two LMS-substrings overlap by more than one symbol. This property favors integer encodings to obtain a compact representation of the sequence of l_i values.

One can encode all l_i values by using the `Simple8b` encoding in an integer array W. All strings $\mathrm{s}(\alpha_i)$ are encoded in a single fixed-width integer array Y, of cell width $\lceil \lg(\sigma^j) \rceil$ bits. The length of each $\mathrm{s}(\alpha_i)$ is also encoded using `Simple8b` into a word array Z. The same observation of the `lcp` sum can be done here: the sum of all $|\mathrm{s}(\alpha_i)|$ on the j-th recursion level is no larger than n^j.

4.2 Decompression

The decoding process is done level-wise, starting from the last recursion level $j = \ell$, by decoding the right side of each rule. At the end, T is decoded from T^1.

In the j-th recursion level, the values (w, y, z) from W, Y and Z, the data structures mentioned in the implementation details of Section 4.1, are decoded sequentially. To obtain the right-hand side of the production rules α_{k+1} from α_k, the first w symbols of α_k are copied to α_{k+1} and the z symbols from Y, which correspond to the string y, are appended to α_{k+1}. After this process, the plain representation of each rule is stored, in a single array of cells with fixed width $\lfloor \lg(\sigma^j) \rfloor + 1$ bits. An additional array of pointers D is also created to find the starting position of each rule in this fixed-width array.

With the fixed-width array and the array of pointers D, T^{j-1} now can be decoded from T^j. First, the right side of v_0^j is copied into T^{j-1}. Then, T^j is scanned in a left-to-right fashion and for each $T^j[i]$ the algorithm appends to T^{j-1} the right-hand side of the non-terminal $T^j[i]$, which can be easily found with the support of array D in constant time.

Time Complexity

The decompression process takes $\Theta(n)$ time.

4.3 Extraction

In order to support extraction of substrings from the compressed text, it is necessary to augment the dictionary with two additional data structures: P_S, a partial-sum on the lengths of the symbols in the reduced string T^ℓ of the last recursion level, and L, a data structure that for each non-terminal X_i stores $|\mathcal{G}(X_i)|$. Formally, those data structures are defined as:

$$P_S(i) = \sum_{j=1}^{i-1} |\mathcal{G}(X_{S_i})|, \quad X_S \to X_{S_1}, \ldots X_{S_k} \text{ and } \quad 1 \leq i \leq k+1$$

$$L(X) = |\mathcal{G}(X)|, \quad X \in \Sigma \cup \Gamma$$

The data structure L can also be defined recursively as:

$$L(X) = \begin{cases} 1, & X \in \Sigma \\ \sum_{i=1}^{|S|} L(S[i]), & X \to S \end{cases}$$

To obtain a substring $T[l, r]$, we then proceed as follows:

1. With a binary search, locate *indices* a and b from P_S such that:

$$a = \max\{1 \leq k \leq |T^\ell| \mid P_S(k) \leq l\}$$
$$b = \min\{1 \leq k \leq |T^\ell| + 1 \mid P_S(k) > r\} - 1$$

2. Let ℓ be the number of levels in GCIS grammar and S the string derived from X_S. Then define $E^\ell = S[a, b]$ and follow the next steps for $i = \ell$ to $i = 1$.

3. Apply a derivation step to each non-terminal $X \in E^i$ to obtain a new string E^{i-1}. Note that $\mathcal{G}(E^{i-1}) = T[l', r']$ is a superstring of $T[l, r]$.

4. Trim E^{i-1} from the left and right as much as possible as long as it generates a superstring of $T[l, r]$. This can be done efficiently because we know the length of $\mathcal{G}(X)$, for every $X \in \Sigma \cup \Gamma$.

   (a) If $i = 1$, then E^0 contains only terminal symbols and generates a superstring $T[l', r'] = E^0$ of $T[l, r]$. Thus, one simply extracts the symbols $E^0[l - l' + 1, r - l' + 1]$ to obtain $T[l, r]$.

   (b) If $i > 1$, then E^{i-1} contains only non-terminal symbols and generates a superstring $T[l', r'] = \mathcal{G}(E^{i-1})$ of $T[l, r]$. We then trim E^{i-1} by using L and finding, with a linear search, two indices a and b of E^{i-1} such that:

$$a = \max \left\{ 1 \leq k \leq |E^i| \;\middle|\; l' + \sum_{j=1}^{k-1} L(E^i[j]) \leq l \right\}$$

$$b = \max \left\{ 1 \leq k \leq |E^i| \;\middle|\; r' - \sum_{j=k+1}^{|E^i|} L(E^i[j]) \geq r \right\}$$

E^{i-1} is then trimmed to $E^{i-1}[a, b]$ before proceeding.

Figure 4.2 shows an example for extracting a text using the aforementioned procedure.

Implementation details

Since the length of the string T^ℓ is much shorter than the original text in practice, the verbatim representation of P_S as an array of integers is affordable.

The array L is represented using DACs. This representation allows efficient access while representing the data in a compact way.

To support fast extraction, we need to efficiently decompress a single rule. `Simple8b` encoding works very well when the objective is compressing or decompressing since all the rules are first expanded sequentially in the decompressing stage. However, when the aim is to extract symbols, we need to expand individual rules. Thus, instead of encoding all the `lcp` values with the `simple8b` scheme, Elias-Fano encoding is employed, allowing us to retrieve an arbitrary `lcp` value of a rule efficiently and hence the decoding of an arbitrary rule. The length of each $\mathrm{s}(\alpha_i)$ is also encoded using Elias-Fano and the $\mathrm{s}(\alpha_i)$ values are encoded in a fixed-width integer array. Since the `lcp` values are front-encoded, we force that every k-th `lcp` value is set to 0, with $k \in O(1)$. This setting does not have a significant impact on compression and ensures that we have to backtrack a constant number of rules to extract an individual rule prefix.

4.4 Suffix Array Construction During Decompression

The suffix array (SA) construction boils down to sorting all suffixes of T. Although GCIS compression does not sort suffixes, it executes Step 1 of SAIS and the production rules created correspond to the LMS-substrings in sorted order, which are used by SAIS for sorting all suffixes. We show next how to modify our decompression algorithm for building SA as a byproduct, without asymptotic additional overhead.

First, when $j = \ell$, the suffix array SA^j is built directly from T^ℓ as $\mathsf{SA}^j[T^j[i]] = i$. Nong et al. [2009] observed that SA^j also gives the order of all LMS-suffixes of string T^{j-1}. Then, T^{j-1} is decompressed (Section 4.2), and Steps 2, 3 and 4 of SAIS (Section 2.3.1) are executed to obtain SA^{j-1}, and so on. The algorithm proceeds for $j = \ell - 1, \ldots, 1$, obtaining the reduced string T^{j-1} together with SA^{j-1} at each iteration. At the end, the original string T is decompressed from T^1 and SA is induced from SA^1.

Running Example

Assuming the text T = AGCCTAAGCCTAAGTAAAG\$, GCIS would yield the grammar depicted in Figure 4.3.

When decompressing, we start from $X_S \to v_0^2 v_2^2 v_1^2$ and compute the suffix array SA^2 from $T^2 = v_2^2 v_1^2$, process illustrated by Figure 4.4.

After T^2 is decompressed into T^1, we use SA^2 to obtain the order of all LMS sufixes of T^1 which is $v_1^1 < v_3^1$. Thus we have sorted the LMS suffixes at the end of each c-bucket, as shown in Figure 4.5.

Then, the L-type suffixes are induced from the LMS-type suffixes (Figure 4.6), and the S-suffixes are induced from the L-type suffixes (Figure 4.7), thus definitively computing SA^2.

Now T^1 is decompressed into $T^0 = T$ and the remaining process is described by Figures 2.7 to 2.9.

Time Complexity

SA is built in $O(n)$ time, since each step of SAIS is linear and the length of all reduced strings is $O(n)$.

4.5 LCP Array Construction During Decompression

When $|\Sigma| \in O(1)$, the longest common prefix (LCP) array can also be computed in linear time within the induced suffix sorting framework [Fischer, 2011; Louza et al., 2017]. We show below how to modify our decompression algorithm to compute SA and LCP together, without assymptotic additional cost.

When $j = 1$, the original string T is decoded from T^1, and SA^1 stores the order of all LMS-suffixes of T. Then, in linear time, we compute the LCP array of the LMS-suffixes using a sparse variant of Φ-algorithm by [Kärkkäinen et al., 2009], which avoids storing auxiliary arrays by reusing the space of $\mathsf{SA}[n/2, n]$ and $\mathsf{LCP}[n/2, n]$. The LCP

values between the LMS-suffixes are used to induce the LCP values between the L-suffixes during Step 3 (Section 2.4), and these are used to induce the S-suffixes during Step 4 (see [Louza et al., 2017] for details). Given an additional stack of $O(\sigma \log n)$ bits [Gog and Ohlebusch, 2011] or the array M described in Section 2.3.1, each LCP value induction is done in $O(\sigma)$ time, which we assume to be constant at the top recursion level.

Time complexity

LCP is built in $O(n)$ time, since each step of SAIS is linear and the LCP-values induction can be done in constant time for each value provided $|\Sigma| \in O(1)$.

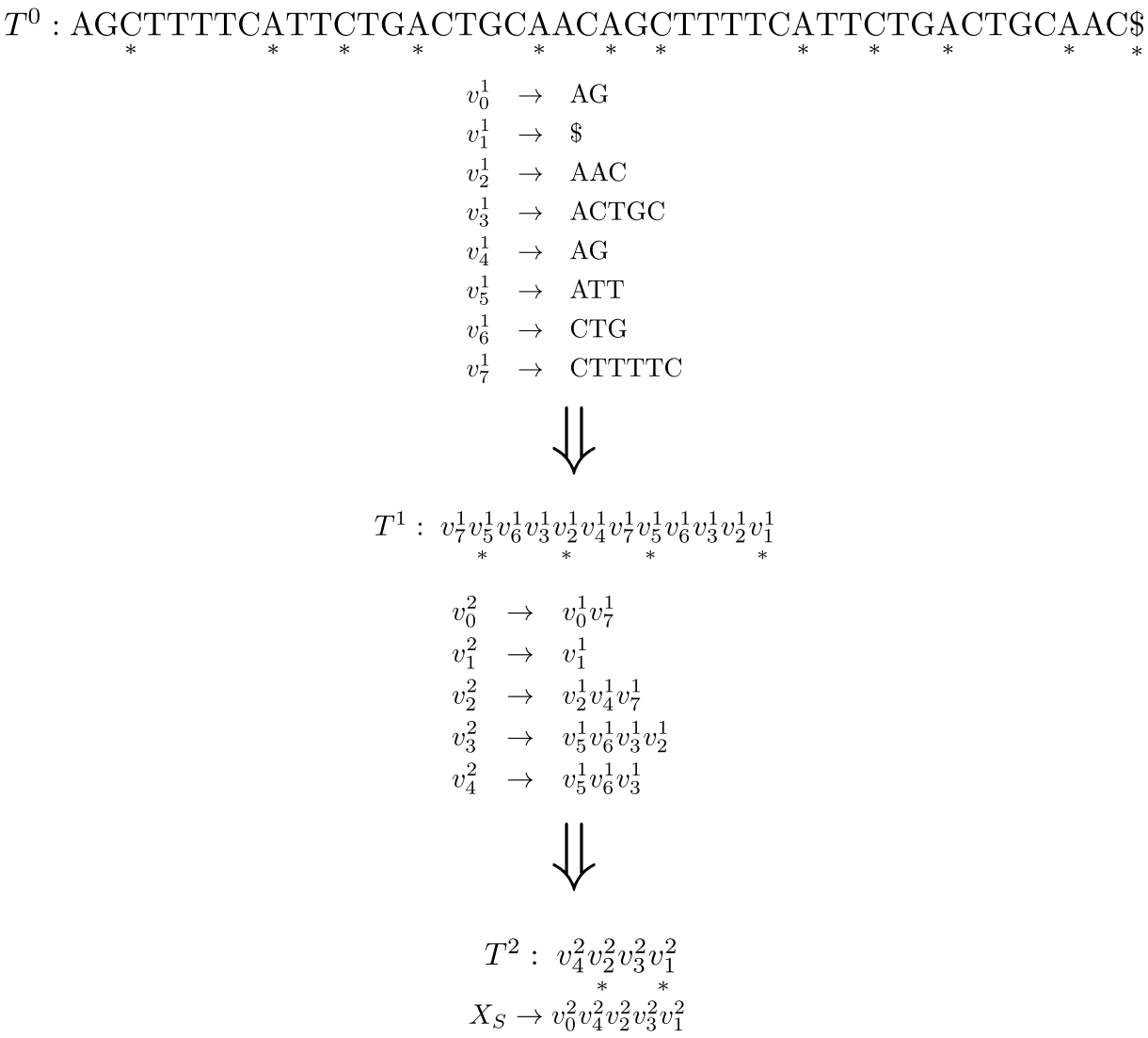


Figure 4.1: Grammar construction during GCIS. All LMS-substrings (those defined by consecutive '$*$' symbols), are sorted according to SAIS framework, and then rules $v_0^0 \rightarrow \text{AG}, v_1^0 \rightarrow \$, \ldots, v_7^0 \rightarrow CTTTTC$ are created. Next, T^1 is obtained by replacing every LMS-substring by the left-hand side of its rule. The procedure is applied recursively to T^1. When T^2 is created, the alphabet size is equal to $|T^2| = n^2$, and thus the starting rule X_S that generates T^0 is obtained.

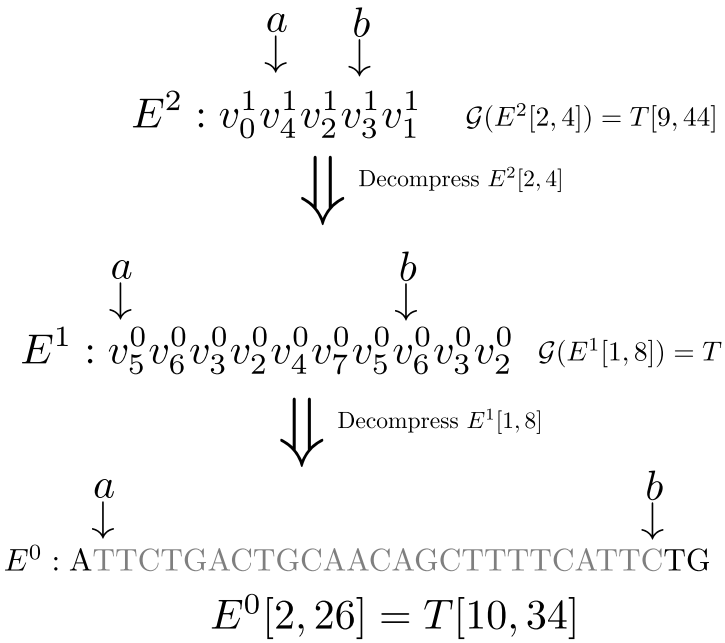


Figure 4.2: Extraction of the substring $T[10,34]$ of the text of Figure 4.1. Initially, a binary search is performed on P_S to identify the substring of E^2 that shall be decompressed: $E^2[2,4]$, which generates $T[9,43]$, is decompressed to obtain E^1. A linear scan is performed in both ends considering the length of the terminals generated by each rule of E^1 to find the indexes $a = 2$ and $b = 8$. $E^1[2,8]$ is then decompressed and $E^0 = T[9,36]$ is obtained, which makes possible to extract $T[10,34]$ by simply ignoring both ends.

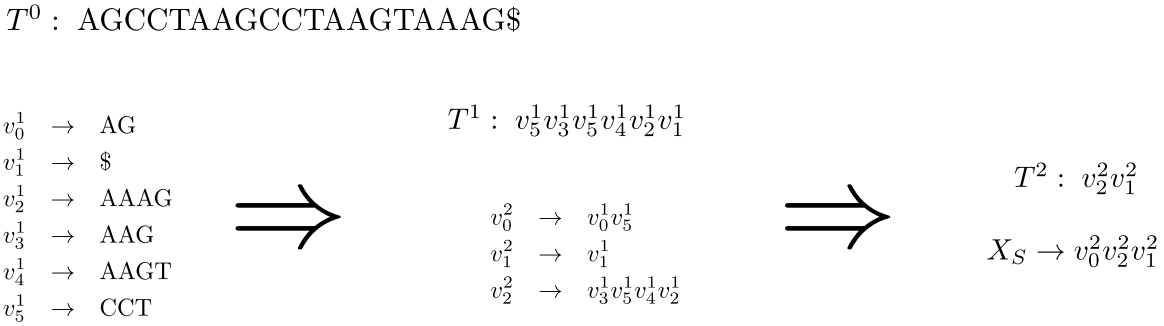


Figure 4.3: GCIS grammar for T = AGCCTAAGCCTAAGTAAAG$.

SA^2

1	2
2	1

Figure 4.4: SA^2 for T^2.

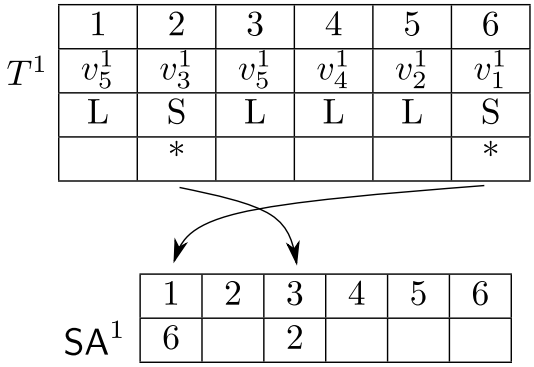


Figure 4.5: LMS suffixes in T^1 are now sorted in the end of its c-bucket.

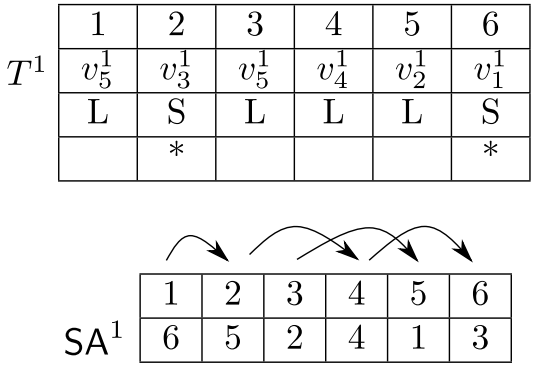


Figure 4.6: L-type suffixes in T^1 are induced from LMS-type suffixes.

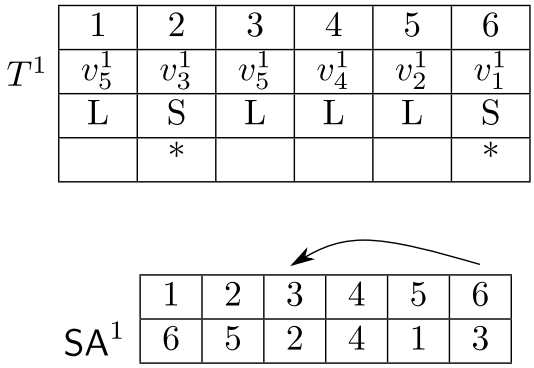


Figure 4.7: S-type suffixes in T^1 are induced from L-type suffixes.

Chapter 5

Experiments

To confirm the practical value of GCIS, we conducted experiments considering several *corpora* that includes different kinds of texts. We measured compression and decompression speed, compression ratio, and memory usage during compression and decompression of GCIS against classical and grammar-based compressors, evaluated the extraction of symbols and showed the efficiency of suffix array construction of GCIS during the decompression. Complementary experiments, showing empirical attributes of GCIS grammar, are also presented in this chapter.

5.1 Experimental Setup

5.1.1 Corpora

The experiments contained in this document consider a large variety of texts including: regular texts, repetitive texts and very large texts. Regular texts were taken from the corpora `large-corpus` [Trigell, 1998], `enwiki` [Mahoney, 2006], `manzini` [Manzini, 2003], `pizza-chili` [Ferragina and Navarro, 2005a] and `silesia` [Deorowicz, 2003]. Repetitive texts were chosen from `pizza-chili-repetitive` *corpus* [Ferragina and Navarro, 2005b]. Very large inputs were built by repeating and mutating strings such as `chr19` [Consortium, 2009], `sars-cov` [NCBI, 2020] and `salmonella` [NCBI, 2007] with a mutation rate of 0.1%, thus these texts are highly repetitive as well; each filename has an integer suffix that represents the number of repetitions. In addition, a 20GB prefix from November 2019 Wikipedia dump was taken [Wikipedia, 2019]. Tables 5.1, 5.2 and 5.3 summarize the chosen texts, required disk space and alphabet size, grouping in boxes texts from the same *corpus*.

Table 5.1: Regular texts.

Regular text	Size (MB)	$\|\Sigma\|$
archive	27.07	169
emacs	47.46	256
linux	47.60	256
samba	41.58	256
spamfile	84.22	123
enwiki8	100.00	205
enwiki9	1000.00	206
chr22	34.55	5
etext99	105.28	146
gcc-3.0.tar	86.83	150
howto	39.42	197
jdk13c	69.73	113
linux-2.4.5.tar	116.25	256
rctail96	114.71	93
rfc	116.42	120
sprot34.dat	109.62	66
w3c2	104.20	256
dblp.xml	296.14	97
dna	403.93	16
english	2210.40	239
pitches	55.83	133
sources	210.87	230
dickens	10.19	100
mozilla	51.22	256
mr	9.97	256
nci	33.55	62
oofice	6.15	256
osdb	10.09	256
reymont	6.63	256
samba	21.61	256
sao	7.25	256
webster	41.46	98
xray	8.457	256
xml	5.35	104

5.1.2 Compressors and Extractors Tools

To evaluate GCIS in compression speed, decompression speed and compression ratio, we chose the well-known compressors: Gzip [Gailly and Adler]; Bzip2 [Seward, 1996]; 7-zip [Pavlov]; the statistical compressor Ppmdj [Shkarin, 2006]; the grammar compressor Re-Pair [Wan, 2014]; the grammar compressor SOLCA [Takabatake et al., 2017]; the LZD

Table 5.2: Repetitive texts.

Repetitive text	Size (MB)	$\|\Sigma\|$
cere	461.29	5
coreutils	205.28	236
dblp.xml.00001.1	104.86	89
dblp.xml.00001.2	104.86	89
dblp.xml.0001.1	104.86	89
dblp.xml.0001.2	104.86	89
dna.001.1	104.86	5
einstein.de.txt	92.76	117
einstein.en.txt	467.63	139
english.001.2	104.86	106
Escherichia_Coli	112.69	15
influenza	154.81	15
kernel	257.96	160
para	429.27	5
proteins.001.1	104.86	21
sources.001.2	104.86	98
world_leaders	46.97	89

Table 5.3: Very large texts.

Very large text	Size (MB)	$\|\Sigma\|$
c050	2956.45	9
c100	5912.90	9
c150	8869.35	9
c200	11 825.80	9
c250	14 782.25	9
c300	17 738.69	9
c350	20 695.14	9
sars-cov100000	2990.30	4
sars-cov200000	5980.60	4
sars-cov300000	8970.90	4
sars-cov400000	11 961.20	4
sars-cov500000	14 951.50	4
sars-cov600000	17 941.80	4
sars-cov700000	20 932.10	4
enwiki-20191120-20G	20 000.00	213
salmonella1000	4928.40	4
salmonella2000	9856.80	4
salmonella3000	14 785.20	4
salmonella4000	19 713.60	4

compressor that relies on a clever LZ78 modification [Goto et al., 2015]; the bidirectional

parsing scheme LZRR [Nishimoto and Tabei, 2019]; the Lempel-Ziv approximation for very large texts ReLZ [Kosolobov et al., 2020]; the RePair approximation for very large texts BigRePair [Gagie et al., 2019].

Along with GCIS, we considered a set of compressors for each *corpus* type (*cf. Sec. 5.1.1*). For regular texts, we considered the compressors: Gzip, Bzip2, Ppmdj, 7-zip, RePair, LZD and SOLCA. For repetitive texts, the following compressors were adopted: 7-zip, RePair, ReLZ, LZD, SOLCA and LZRR. Finally, for very large texts, the compressor set was defined by: 7-zip, ReLZ, BigRePair, LZD and SOLCA. These choices were made regarding the effectiveness of the compressors for each *corpus* type and the relevance in comparison with GCIS. For example: LZRR and RePair cannot deal with large volumes of data due to its high memory consumption, hence it were not considered for very large-texts; Gzip, Bzip2 and Ppmdj does not capture repetitiveness very well, thus they were disconsidered for repetitive and very large texts.

Regarding extraction of symbols, we compared GCIS with different encodings of RePair grammars that allow fast extraction. These encodings can be represented in a more straight-forward way, storing $\mathcal{G}(X)$, for $X \in \Sigma \cup \Gamma$, or in a more elaborated way, creating succinct tree data structures that replace the original grammar encoding while allowing one to obtain the right-hand side of any rule, as described by Maruyama and Tabei [2013]. The implementation of such data structures was based on the work of Gagie et al. [2020] and can be found in [I, 2020]. We used the following encodings:

- `PlainSlp_32Fblc`: uses 32-bit integers for the array representations.
- `PlainSlp_FblcFblc`: employs the minimum bit length required to represent the maximum value of a given integer array.
- `PlainSlp_IblcFblc`: uses roughly $\lceil \lg i \rceil$ bits to represent the i-th rule exploiting that the i-th rule is less than i. For representing $\mathcal{G}(X)$, for $X \in \Sigma \cup \Gamma$, it uses the same strategy of `PlainSlp_FblcFblc`.
- `PoSlp_Iblc`: employs the approach POSLP of Maruyama and Tabei [2013] to represent the parse tree and encodes the leaves using roughly $\lceil \lg i \rceil$ bits for the i-th rule.
- `PoSlp_Sd`: applies the POSLP approach of Maruyama and Tabei [2013] to represent the parse tree and encodes the leaves with Elias-Fano.

In order to assess the computation of suffix and LCP arrays directly from decompression, GCIS was compared with efficient suffix and LCP construction algorithms implemented by `sais-lite` [Kurpicz, 2015; Mori, 2010] and `divsufsort` [Kurpicz, 2016; Mori, 2008].

GCIS source code and a detailed description of the processed data are available at https://github.com/danielsaad/gcis.

5.1.3 Environmental Setup

Due to memory capacity and availability, we conducted the experiments in two machines, one for the regular and repetitive corpora and another for the very large datasets. Their specifications follow:

Machine #1, used for regular and repetitive texts:

- CPU: `2x Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz` CPUs;
- RAM Memory: 64GB;
- Operating System: `Centos7`, kernel version `3.10`.

Machine #2, used for very large datasets:

- CPU: `2x Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz`;
- RAM Memory: 386GB;
- Operating System: `Debian GNU/Linux 8`, kernel version `3.16`.

We compiled GCIS, RePair (and its extractors), BigRePair, Ppmdj and ReLZ under `gcc` with `-O3 -NDEBUG` flags. The default command line parameters of Gzip, Bzip2, Ppmdj, ReLZ were used on the experiments. A dictionary size of 1 GB was used in 7-zip. BigRePair RAM usage was limited to 10 GB.

GCIS was implemented in `C++11` using the Succinct Data Structure Library (SDSL) version 2.0 [Gog et al., 2014].

5.2 Compression and decompression

We evaluated all compressors in terms of compression ratio, compression and decompression speed. We also considered their peak memory usage during compression and decompression. BigRePair could not compress some texts, so its corresponding data in the graphs are missing. Decompression in ReLZ is not implemented, nonetheless, ReLZ serves as a compression benchmark since it approximates the Lempel-Ziv parse.

It is important to remark that BigRePair does not produce a compact representation of rules, since it represents the right-hand side of its rules with 2 integers (all the rules are of length two). However, we optimized it by representing each rule with at most $\lceil \log_2 r \rceil$ bits, r being the number of rules, and integrating the non-terminals that occur only

once in their corresponding right-hand side. This saves $\lceil \log_2 r \rceil$ bits for each eliminated non-terminal.

5.2.1 Compression ratio

It stands for the ratio between the compressed and the original text size, and it is given as percentage.

Figure 5.1 shows that, for regular texts, LZD and RePair outperforms GCIS, and it is competitive with a basic Lempel-Ziv compressor such as Gzip. However, it is clearly outperformed by Bzip2, Ppmdj and 7-zip. The latter displays the best compression ratio overall, being Ppmdj a close competitor in some cases. GCIS presents better compression ratio than SOLCA.

Figure 5.2 shows the compression ratios for the repetitive texts. In particular, 7-zip obtains the best compression ratio in all cases, closely followed by RePair. The compression ratio of GCIS is about twice that of RePair in most cases, but it is still very good in absolute terms and outperforms LZD, SOLCA, LZRR and ReLZ.

The results for the very large texts are depicted in Figure 5.3. 7-zip is better for the text `enwiki-20191120-20G`. The situation stays as in the smaller repetitive files, with LZD pursuing 7-zip closely. BigRePair and LZD compresses more than GCIS, and GCIS compresses better than SOLCA and ReLZ, except for the chromosome 19 based texts.

5.2.2 Compression speed

Figures 5.4,5.5 and 5.6 show the compression speed, in MB/s, of the compressors for each text type.

Gzip is the fastest compressor in most regular texts. GCIS is the second-fastest compressor, followed by Bzip2, Ppmdj and LZD. In particular, GCIS is typically an order of magnitude faster than the other grammar compressors, RePair and SOLCA, which are its direct competitors.

Considering repetitive texts, GCIS is faster than ReLZ, SOLCA and 7-zip; it is a order of magnitude faster than RePair and LZRR, but is outperformed by LZD, the fastest compressor.

For very large texts, GCIS is much faster than 7-zip, which becomes the slowest of the considered compressors, and slightly faster than SOLCA. However, ReLZ and BigRePair become much faster than GCIS, as expected from being designed for this scenario. LZD is clearly the fastest compressor. A problem for GCIS on these very large files is that, once the text exceeds 2 GiB, it needs to use 64-bit integers, which doubles

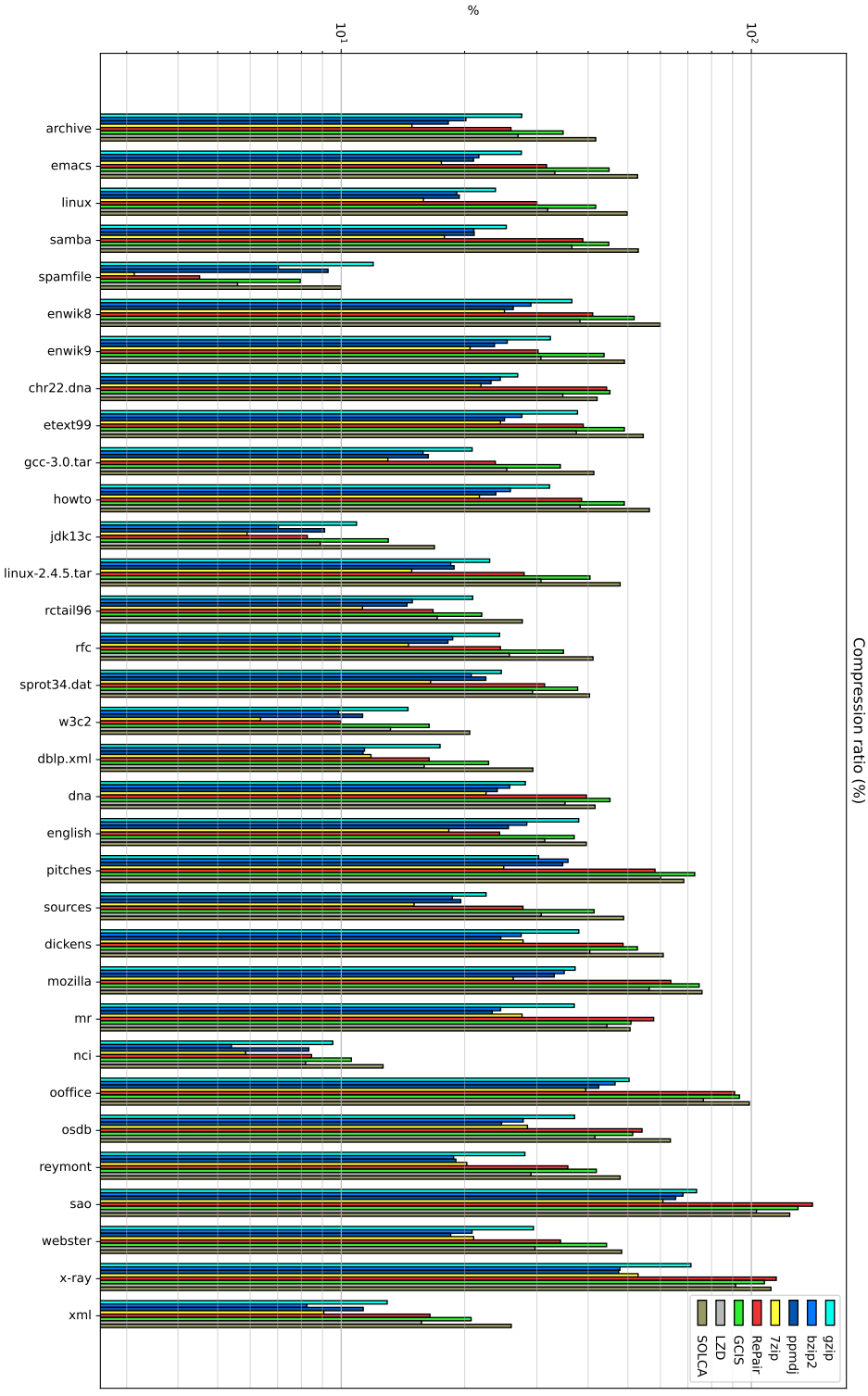


Figure 5.1: Compression ratio on regular texts.

the memory requirements. BigRePair and ReLZ do not suffer from this problem and require a small amount of main memory during compression.

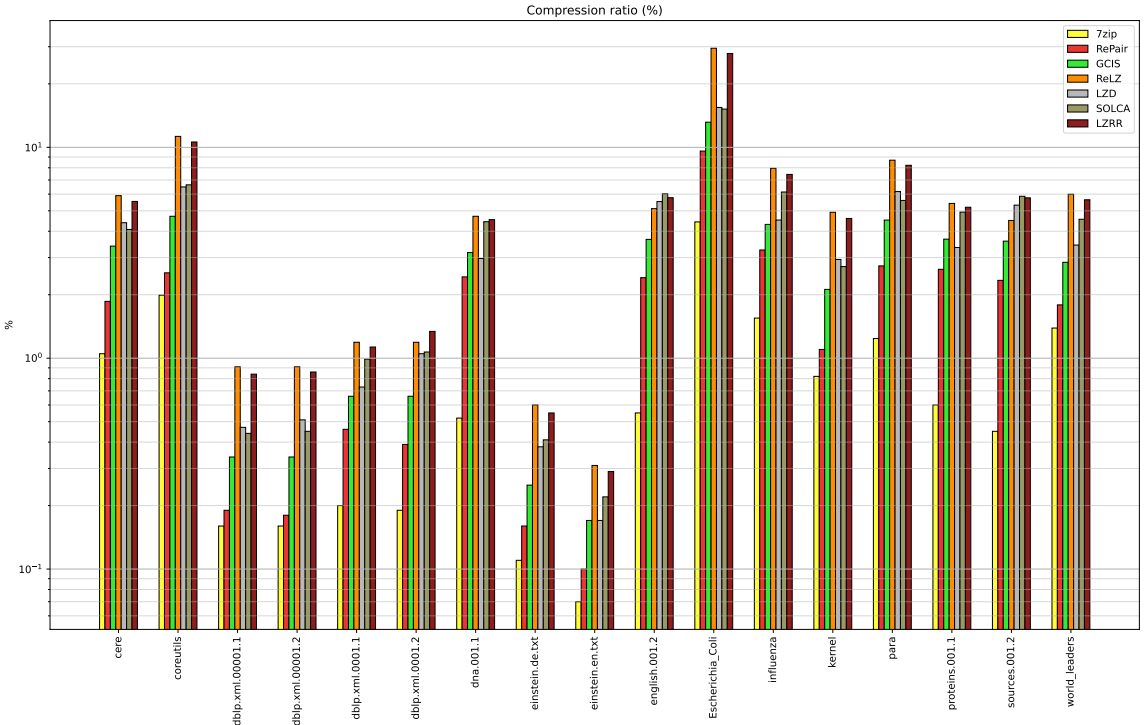


Figure 5.2: Compression ratio on repetitive texts.

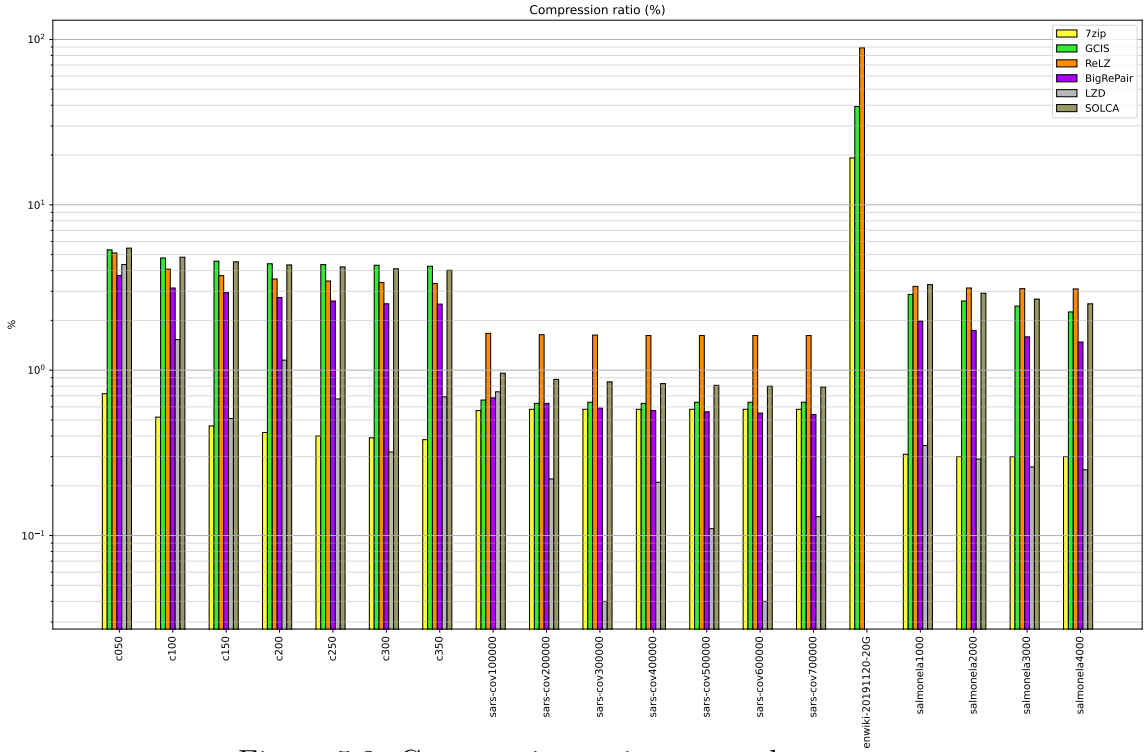


Figure 5.3: Compression ratio on very large texts.

5.2.3 Decompression speed

Figure 5.7 depicts the results for regular texts. Gzip and RePair are the fastest at decompressing, followed by 7-zip LZD and GCIS. Ppmdj and SOLCA are the slowest

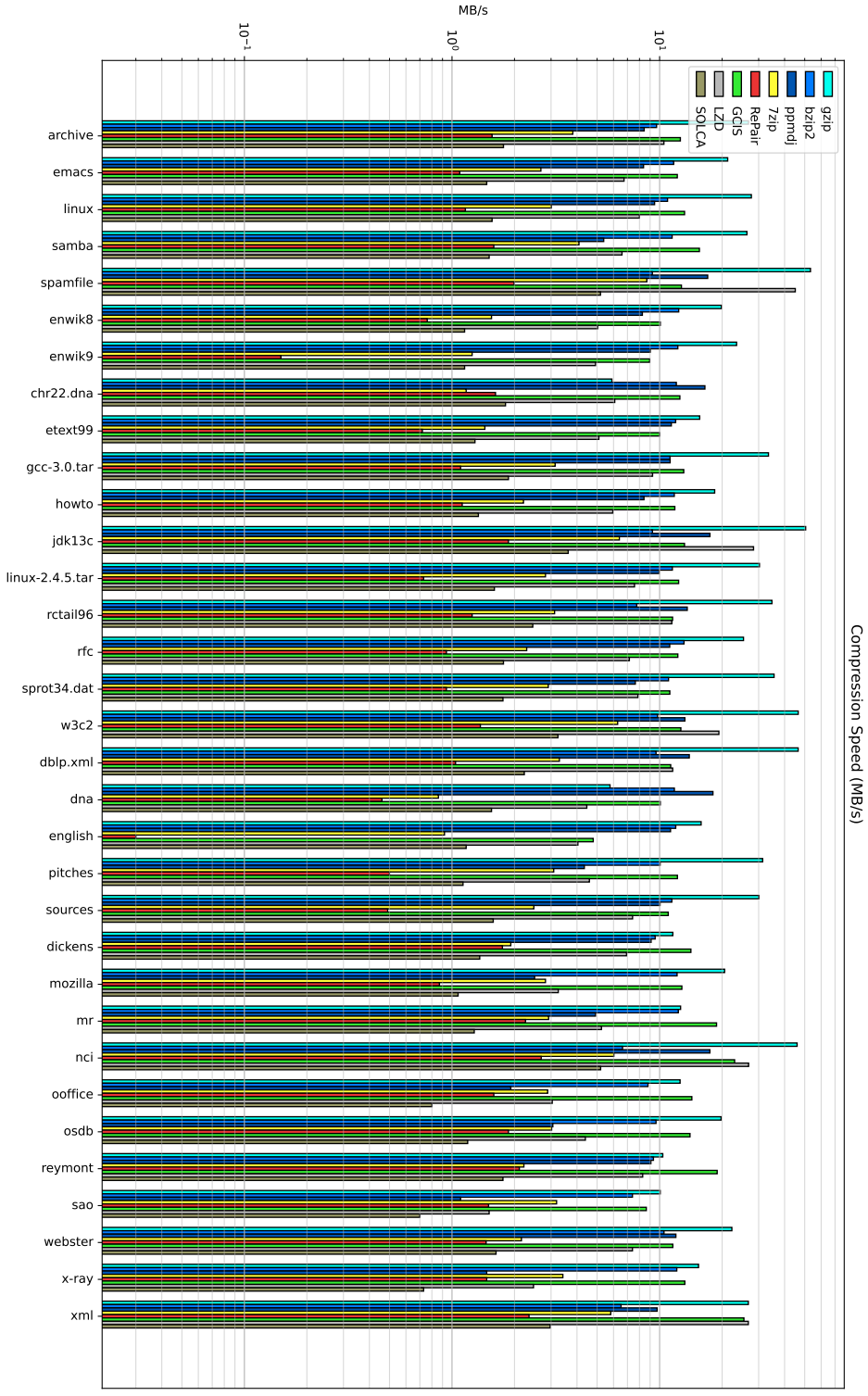


Figure 5.4: Compression speed on regular texts.

decompressors.

Figure 5.8 shows that the situation is similar on repetitive texts, except that 7-ZIP becomes way faster than the others in almost all cases. Despite the relative differences, in

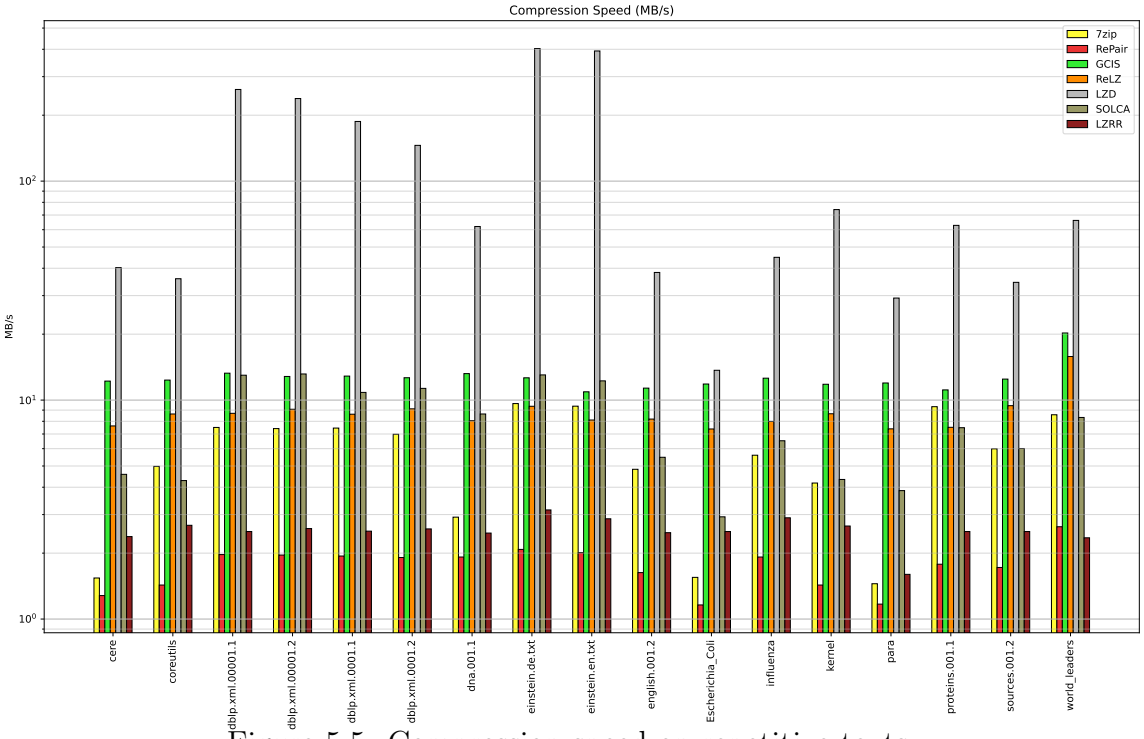


Figure 5.5: Compression speed on repetitive texts.

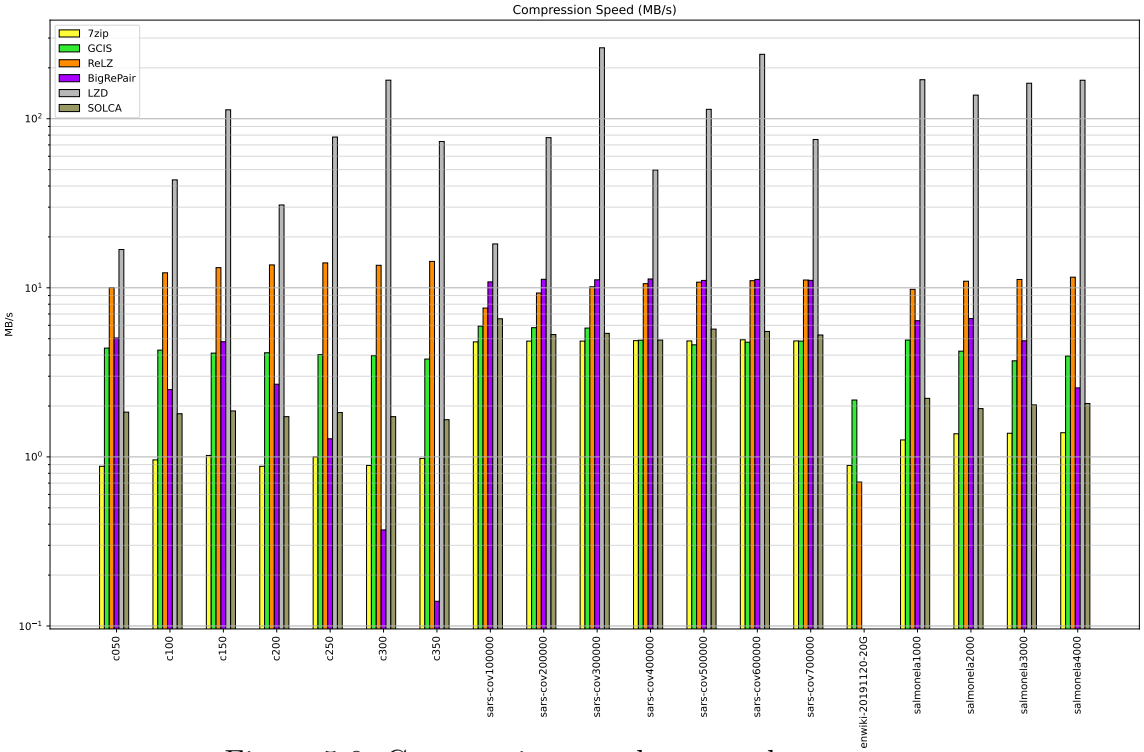


Figure 5.6: Compression speed on very large texts.

absolute terms GCIS is still fast, decompressing the files in around 50 MB/s. SOLCA is outperformed by LZRR and it is the slowest decompressor.

For very large texts, as shown in Figure 5.9, 7-zip is the fastest, followed by LZD,

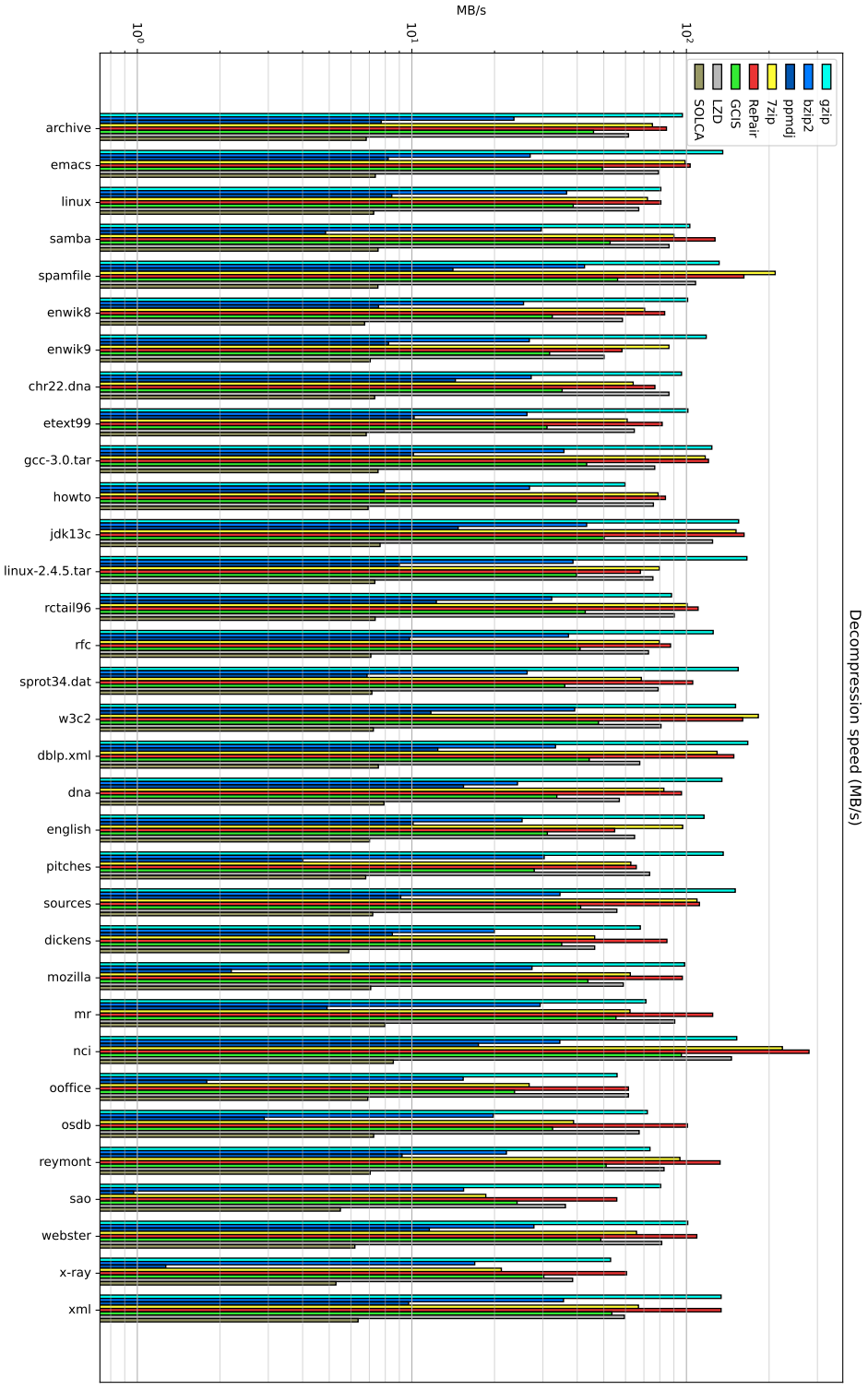


Figure 5.7: Decompression speed on regular texts.

GCIS and then by BigRePair. SOLCA is almost an order of magnitude slower than GCIS.

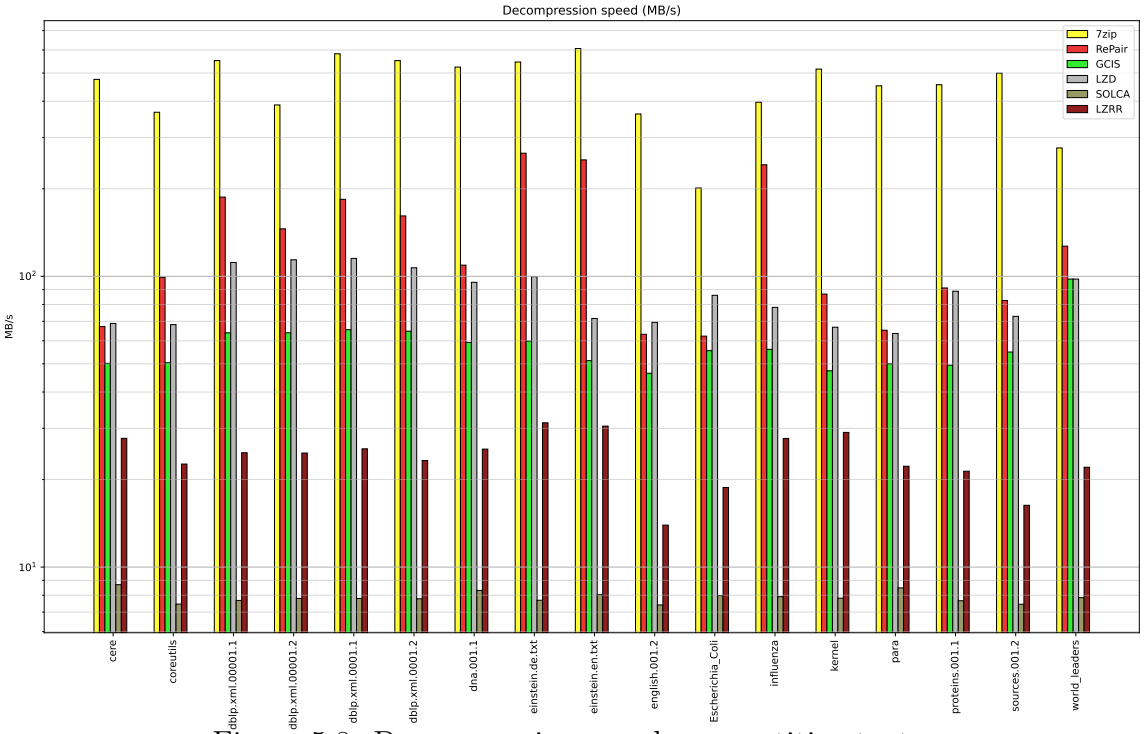


Figure 5.8: Decompression speed on repetitive texts.

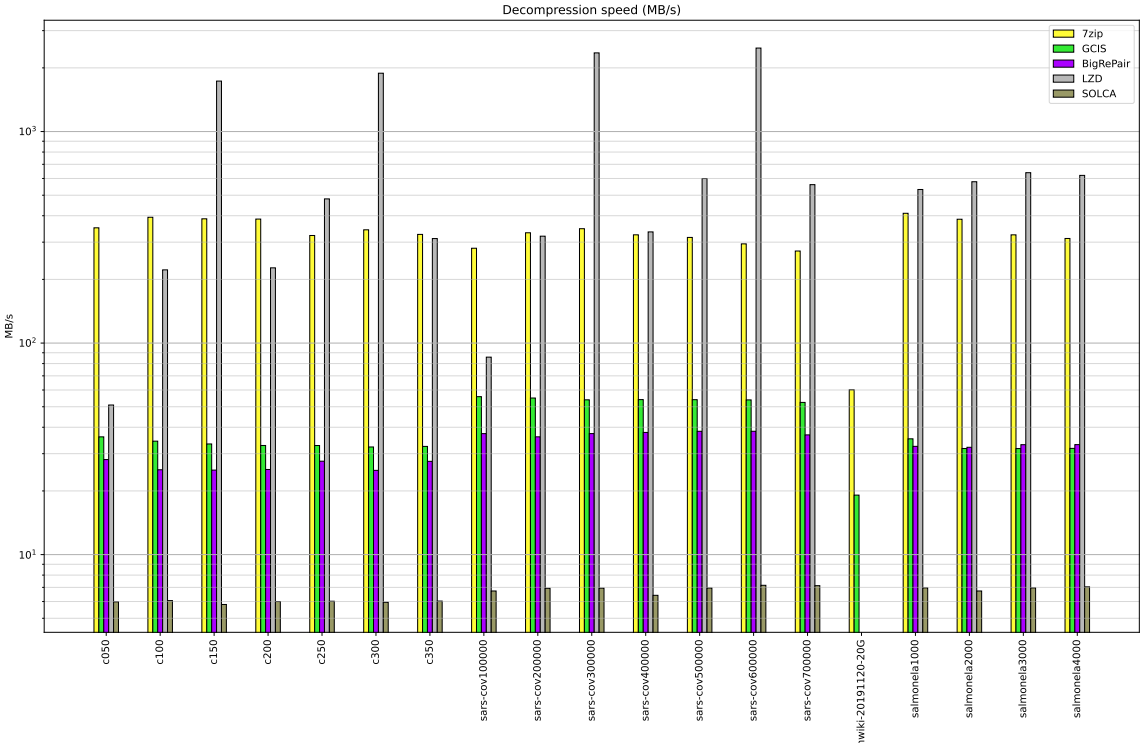


Figure 5.9: Decompression speed on very large texts.

5.2.4 Peak memory

We evaluated the peak memory consumption (resident size) of all compressors during compression and decompression for each type of text; the results are shown in Figures 5.10 to 5.15.

For regular texts, Gzip, Bzip2 and Ppmdj require negligible space to compress or decompress.GCIS is outperformed by SOLCA and followed by LZD, 7-zip, RePair, the last being behind by a large margin. The situation has some changes during decompression: SOLCA is followed by RePair, LZD and 7-zip, and GCIS.

On repetitive texts, regarding compression, GCIS is outperformed by SOLCA and LZD and followed by ReLZ, 7-zip and RePair. On decompression, GCIS stays behind every compressor.

On very large texts, during compression, SOLCA is the most space-efficient, followed by ReLZ, BigRePair, 7-zip LZD and lastly by GCIS. Considering decompression the order changes: LZD is the most efficient compressor, followed by BigRePair and SOLCA. GCIS consumes much more memory than the alternatives in this scenario because for the internal arrays needed for the suffix sorting procedure, which requires 64-bits per entry.

As expected, SOLCA excels for its memory consumption during compression or decompression when compared to other compressors designed for repetitive data.

5.2.5 Overview

Figures 5.16 to 5.18 present conceptual radar charts that summarize, for each text type, the performance of all compressors in each rated aspect. The closer the values are to the pentagon borders, the better the compressor performed on the corresponding aspect.

5.3 Extraction operation

Results depicted by Figures 5.19 and 5.20 show that GCIS is faster than the extractors on succinct encodings of RePair but slower than those running on the more straightforward representation using integer arrays. In turn, regarding space on regular and repetitive texts, GCIS is more space-efficient than the straightforward encodings but less space-efficient than the POSLP alternatives, as shown in Figures 5.21 and 5.22. GCIS is then a competitive alternative regarding the space-time trade-off.

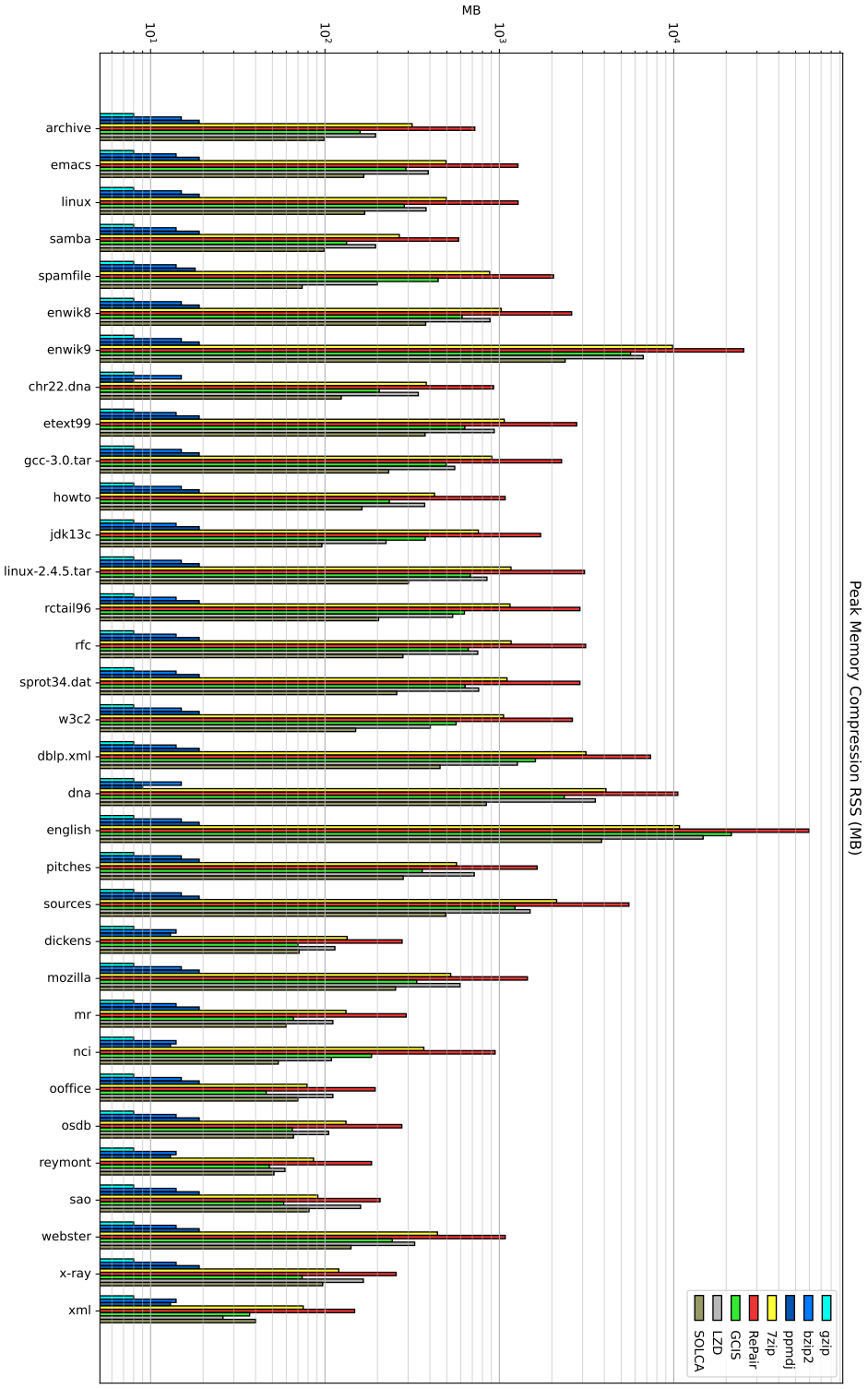


Figure 5.10: Peak memory (in MB) used by the compressors during compression for regular texts.

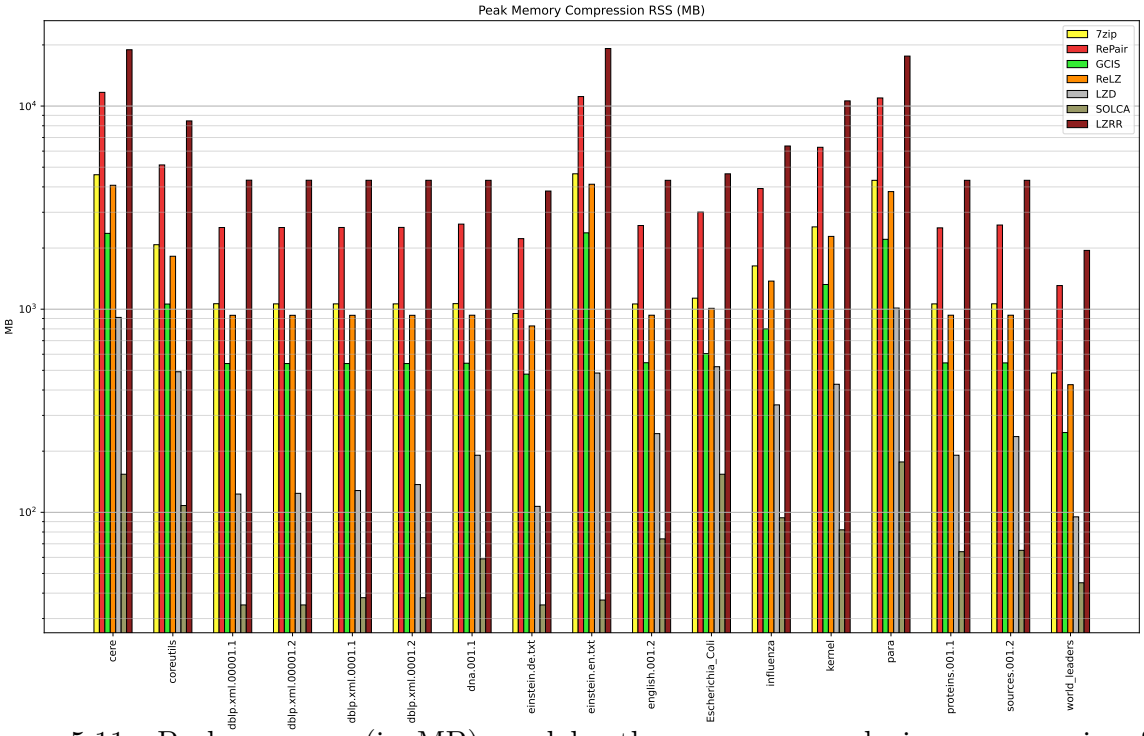


Figure 5.11: Peak memory (in MB) used by the compressors during compression for repetitive texts.

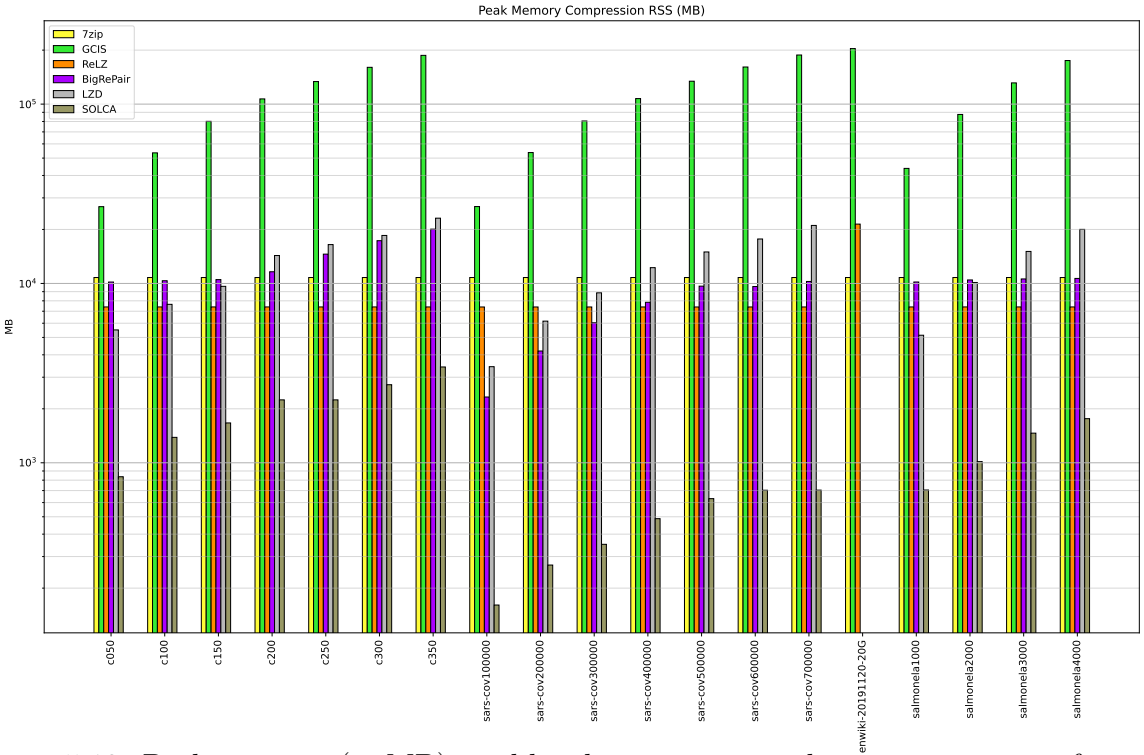


Figure 5.12: Peak memory (in MB) used by the compressors during compression for very large texts.

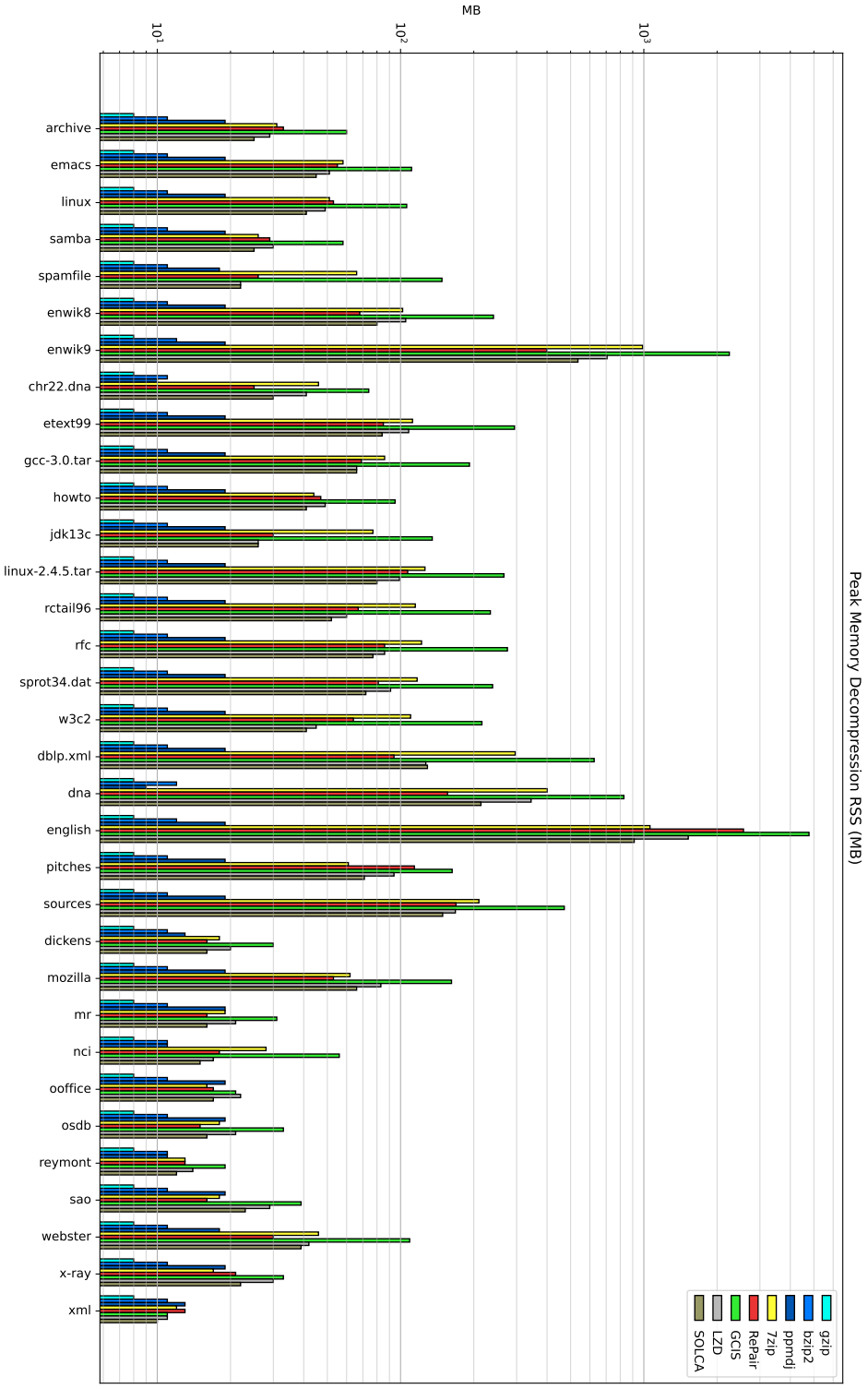


Figure 5.13: Peak memory (in MB) used by the compressors during decompression for regular texts.

5.4 Suffix Array and LCP Construction

Considering the computation of SA and LCP arrays during decompression, we measured the total time to decompress the files with GCIS without generating a plain-text file,

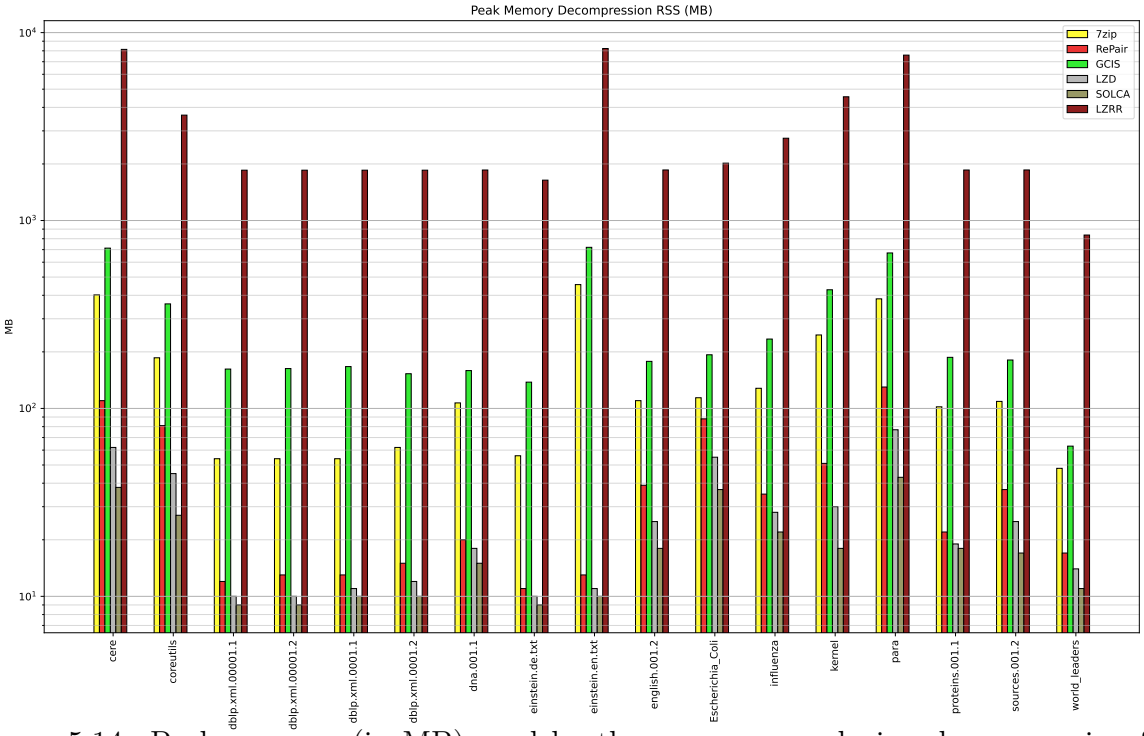


Figure 5.14: Peak memory (in MB) used by the compressors during decompression for repetitive texts.

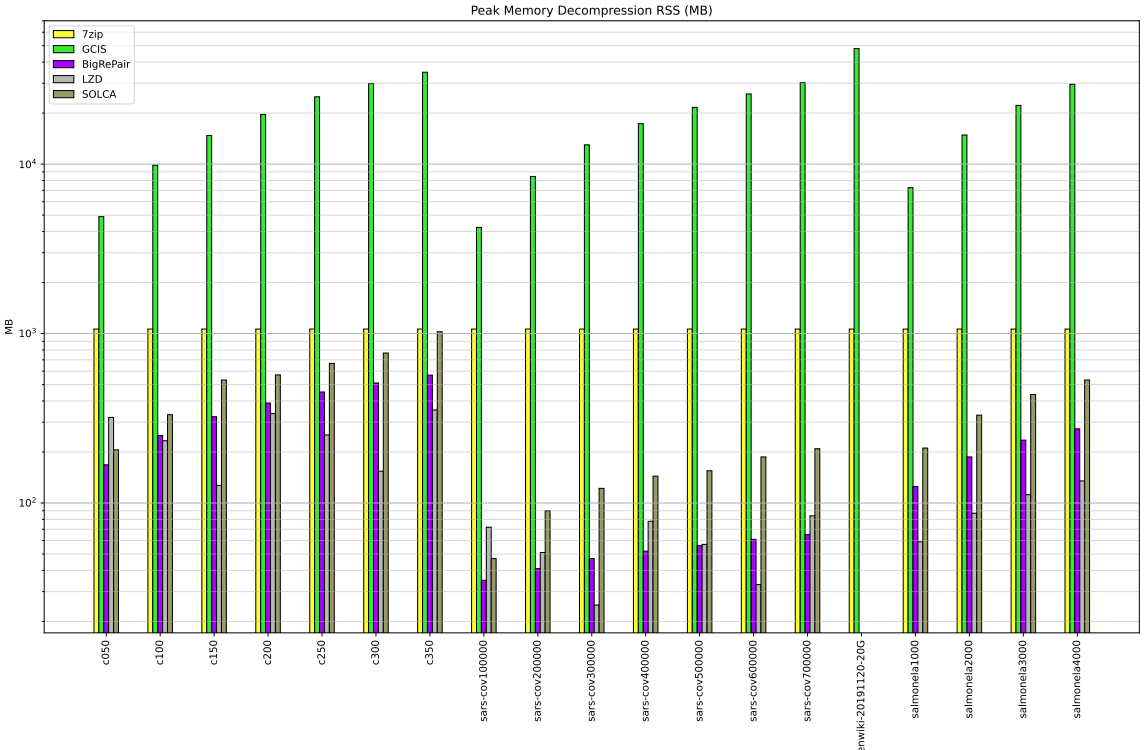


Figure 5.15: Peak memory (in MB) used by the compressors during decompression for very large texts.

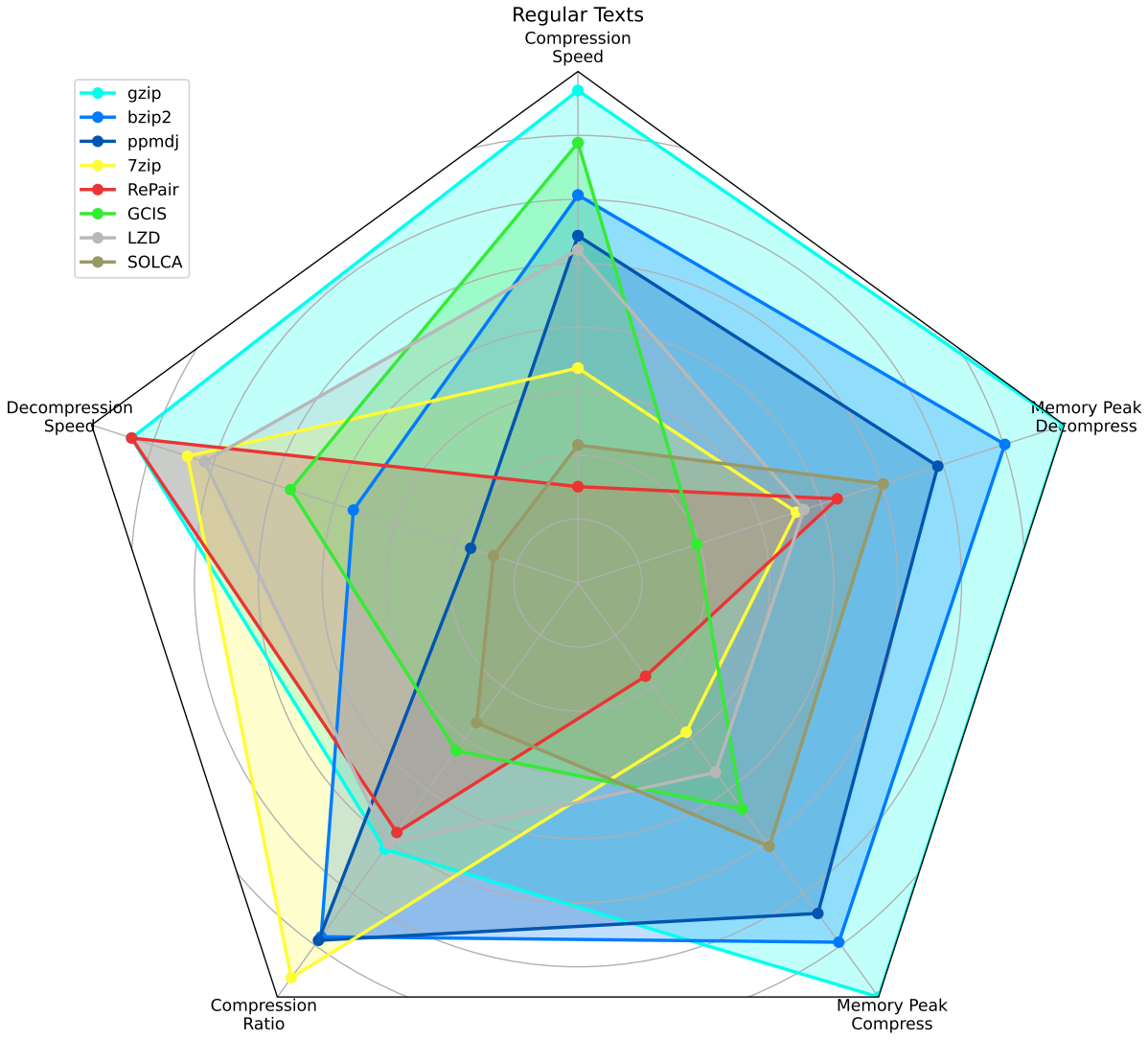


Figure 5.16: Compressor comparison on regular texts.

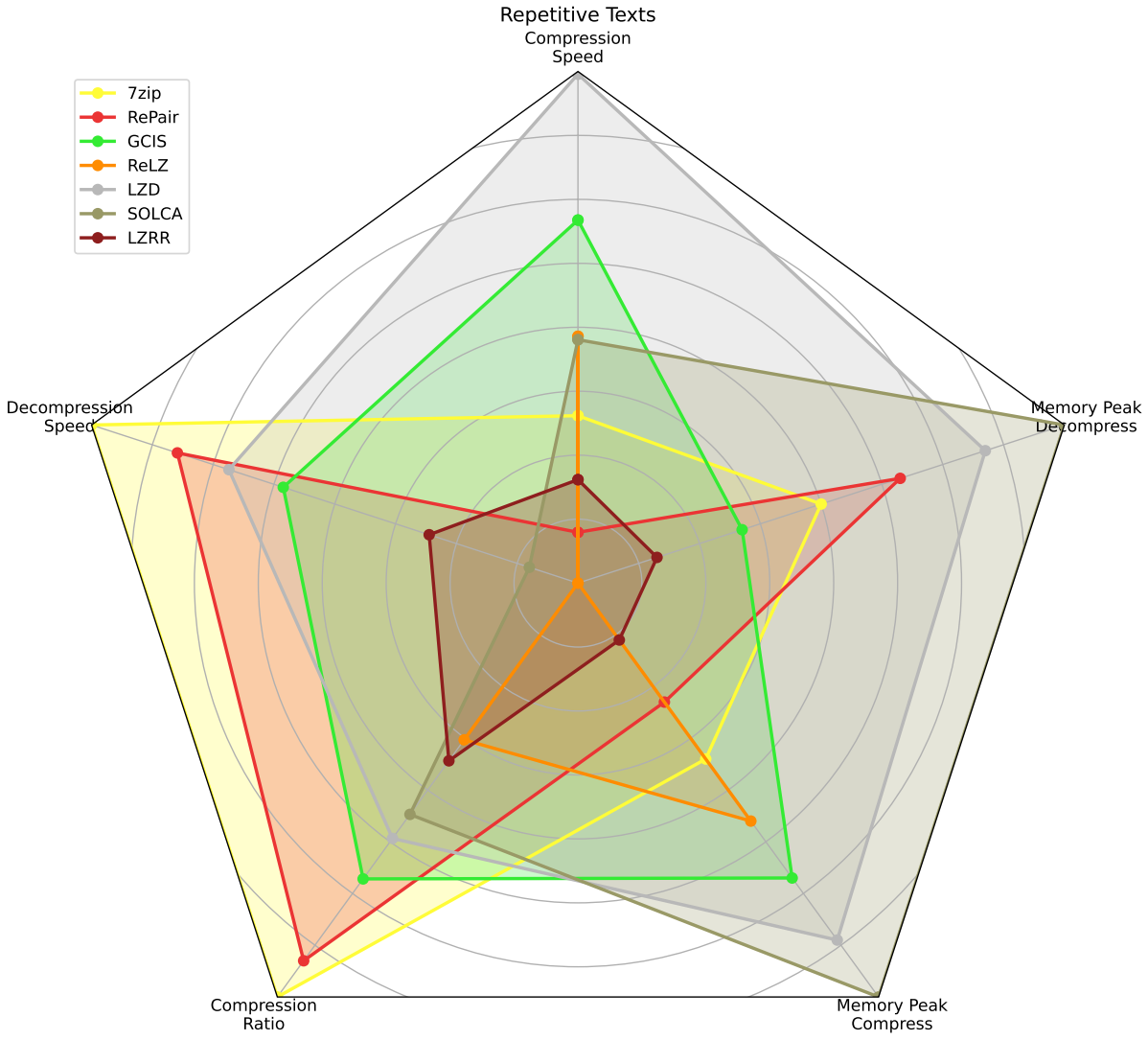


Figure 5.17: Compressor comparison on repetitive texts.

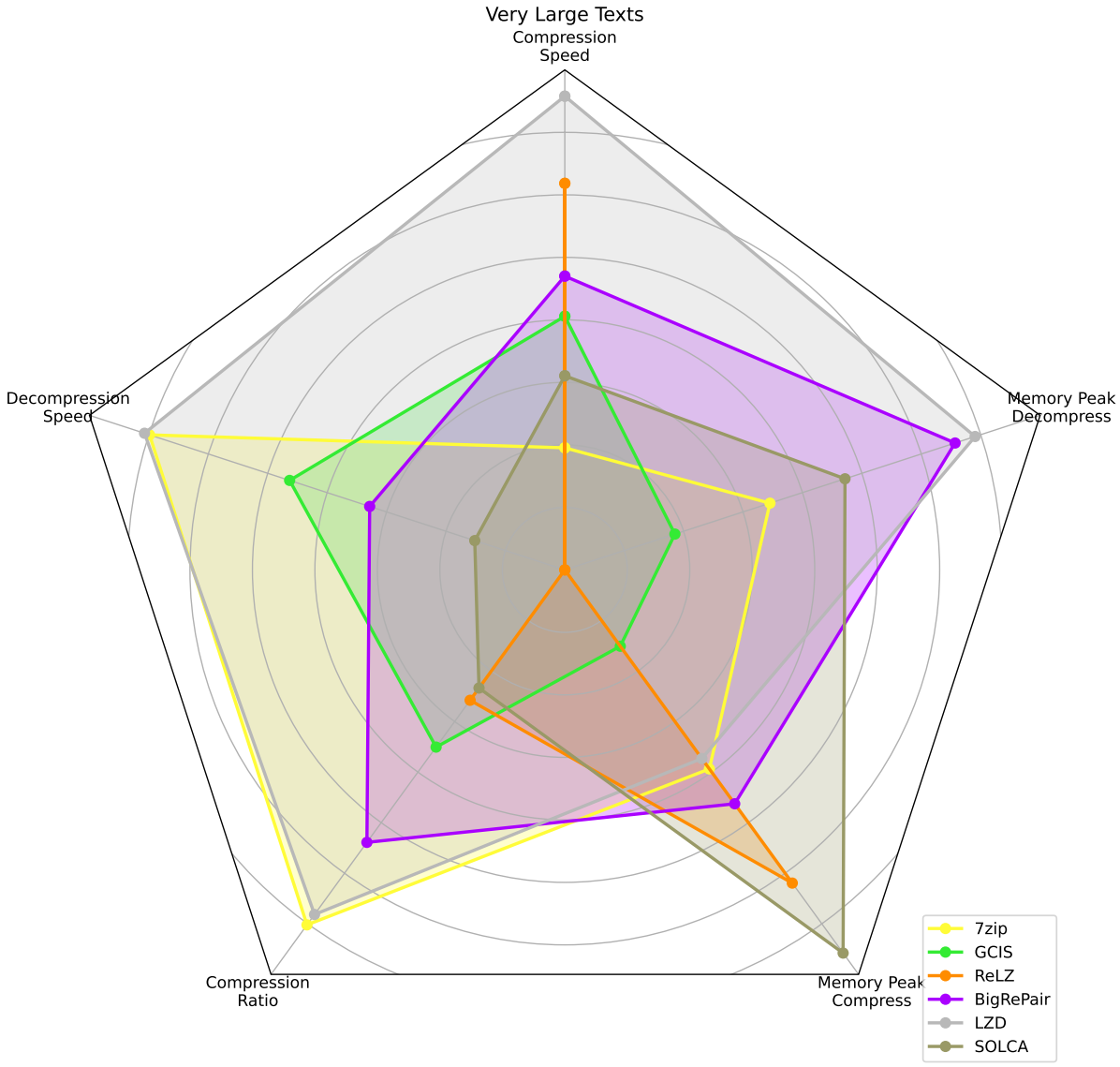


Figure 5.18: Compressor comparison on very large texts.

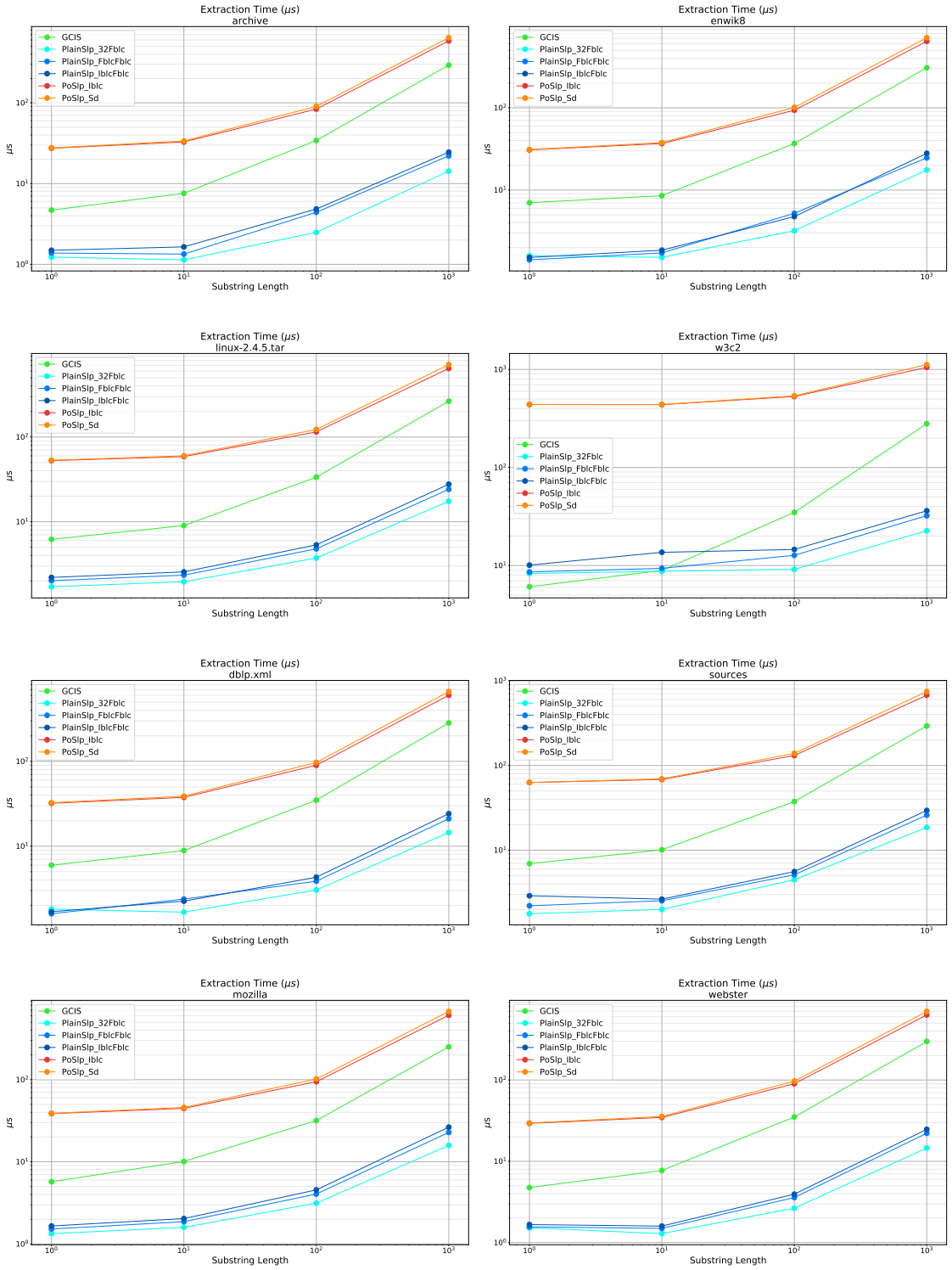


Figure 5.19: Substring length *vs.* extraction time (microseconds) on regular texts.

but instead inducing the SA and the LCP arrays. We compared these results with `SAIS` [Kurpicz, 2015] and `divsufsort` [Fischer and Kurpicz, 2017; Kurpicz, 2016] implementations based on those of Yuta Mori, which are known as the fastest suffix array construction algorithms in practice.

Figure 5.23 (Tables A.16 and A.17) shows the SA and LCP construction on the 8 most repetitive real texts when only the GCIS compressed texts are available. Very large texts were not considered because the implementations of Kurpicz [2015] and Kurpicz [2016] only deal with 32-bit integers. GCIS builds the SA and LCP arrays faster than decompressing and then using the suffix array construction algorithms over the plain text.

The hatched part corresponds to the LCP computation and the black bar corresponds to the time spent in decompressing the text with GCIS to calculate SA and LCP values using the `SAIS` and `divsufsort` implementations.

5.5 Complementary Experiments

Additional experiments were applied to check grammar empirical attributes. For the repetitive text corpus, where our grammar performed best, we measured for each grammar level i that generate rules v_j^i in GCIS:

- The text size to be compressed.
- Alphabet size.
- Number of rules.
- Average right-hand side length.
- Average `lcp` length for adjacent rules.
- The space consumed for grammar level.

Figure 5.24 shows the mentioned results for the file `coreutils`. The figures for every other text is found on appendix A.7.

It can be inferred that in the first grammar level, since the alphabet is small and the text is very repetitive, there are few distinct LMS-substrings and thus few rules. Moreover, as we progress through the compression, the alphabet size sharply increases and then decreases in the following levels. Due to the alphabet size in the deeper levels, we have more distinct LMS-substrings , and the compression is less effective. We can also observe that the `lcp` value between adjacent rules' right-hand side tends to decrease as the compression progress, since the similarity between adjacent LMS-substrings decreases, and thus, the reduced text becomes harder to compress. The factor between string sizes in the deeper levels is approximately 1/3, which recalls to Theorem 3.2 of Nong et al. [2009]: "given the probabilities for each character to be S or L-type are i.i.d. as 1/2, the mean size of a non-sentinel LMS substrings is 4, i.e., the reduction ratio is at most 1/3".

5.5.1 LCP Compression Contribution

In order to evaluate the effect of `lcp` compression, we implemented a variant that did not compress such information and applied it to the *pizza-chili-repetitive* corpus. This can be verified in Figure 5.25 (Table A.18). It is possible to observe that the variant that performs the `lcp` compression requires roughly 70% of the compression ratio required by the one that does not. The exceptions are the `cere` and `para` texts, where GCIS requires 35% and 53% of the compression ratio over the variant that does not compress the `lcp`. This can be explained by observing Figures A.1 and A.13. These texts share a much larger `lcp` length between right-hand sides of production rules, and thus, do benefit more from `lcp` compression than the other present in the *corpus*.

These empirical results show that the `lcp` compression scheme has an important role on GCIS approach, effectively enhancing the compression of GCIS grammars, by taking advantage of the fact that the right-hand sides of the production rules are sorted as a by-product of the suffix sorting procedure of Nong et al. [2009].

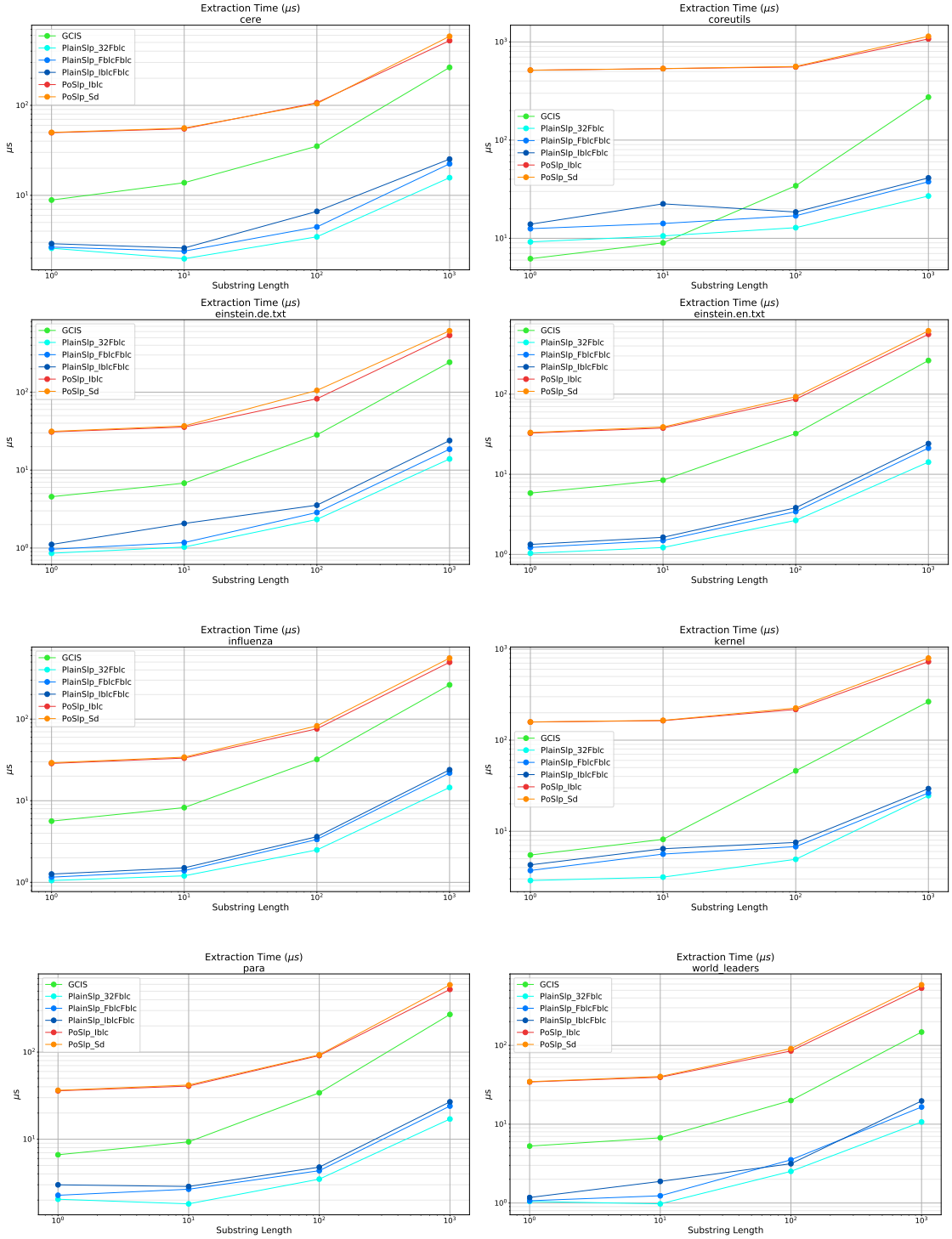


Figure 5.20: Substring length *vs.* extraction time (microseconds) on repetitive texts.

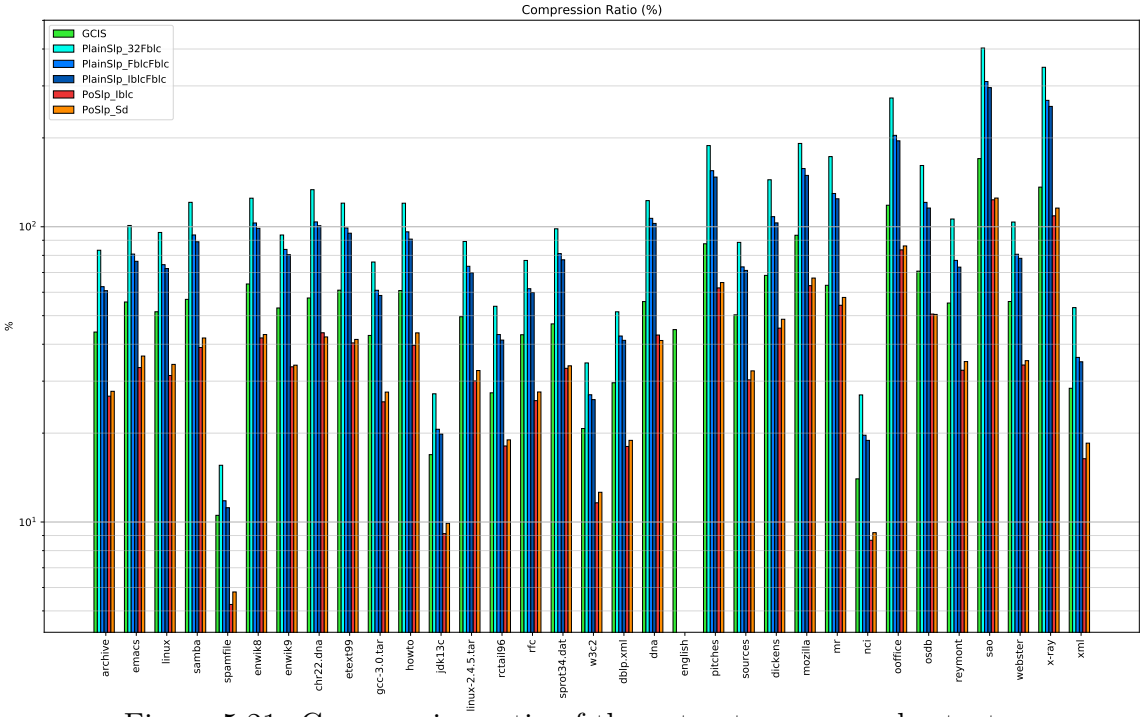


Figure 5.21: Compression ratio of the extractors on regular texts.

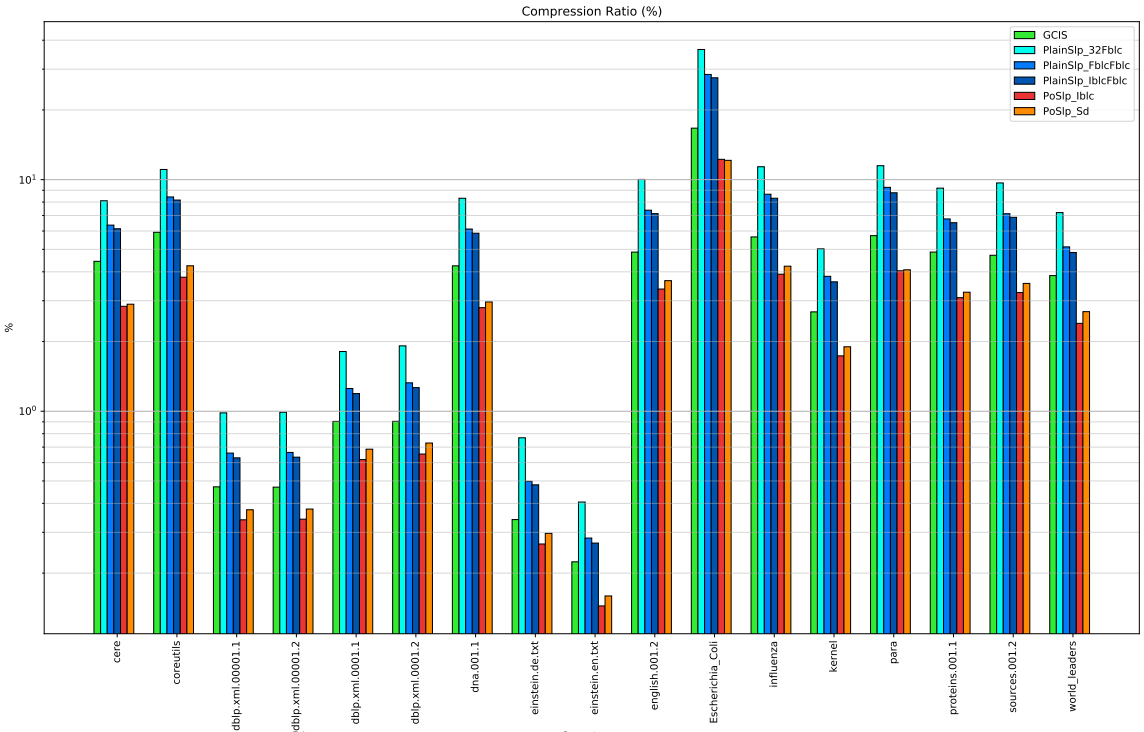


Figure 5.22: Compression ratio of the extractors on repetitive texts.

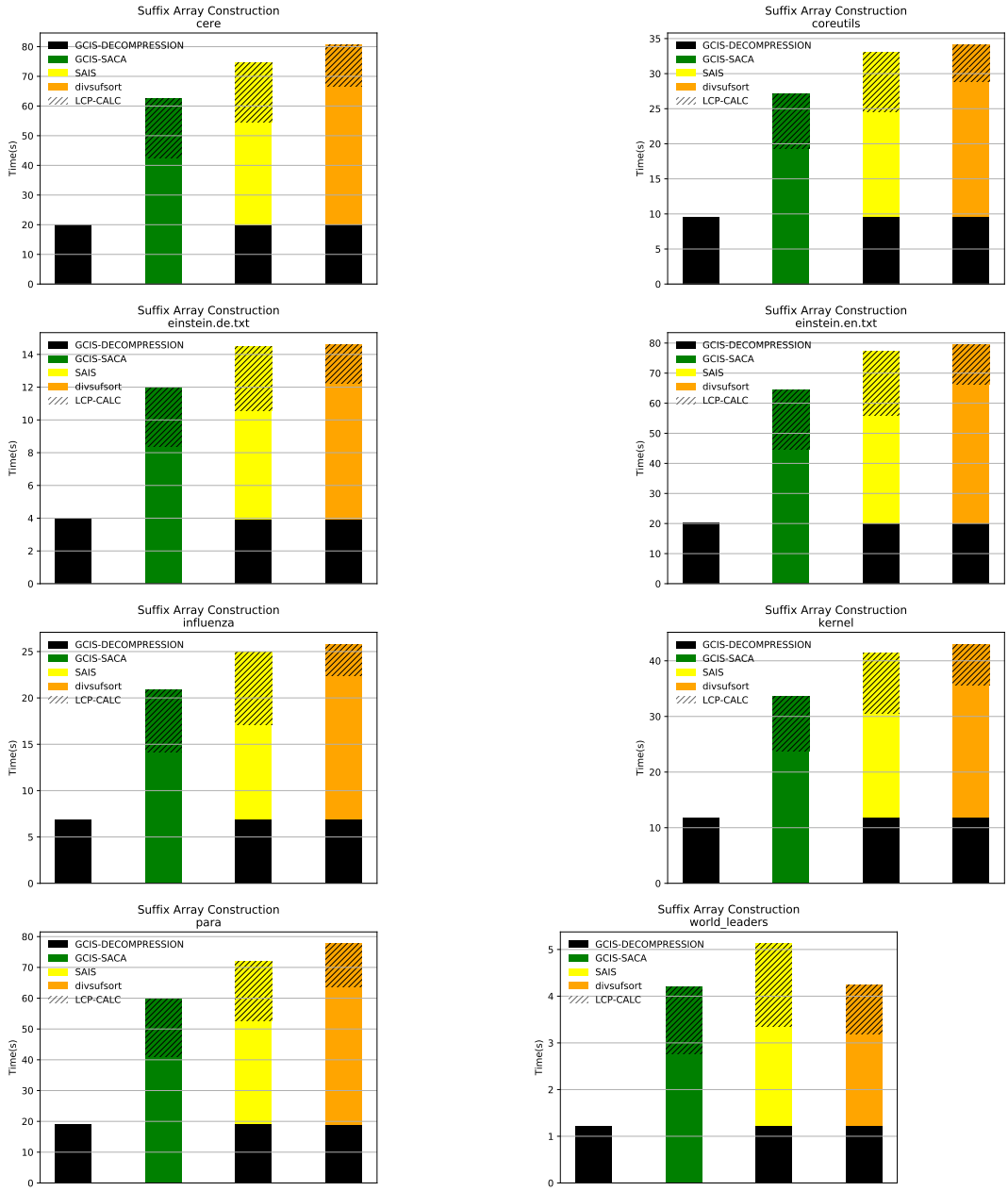


Figure 5.23: Time consumed during Suffix Array and LCP construction.

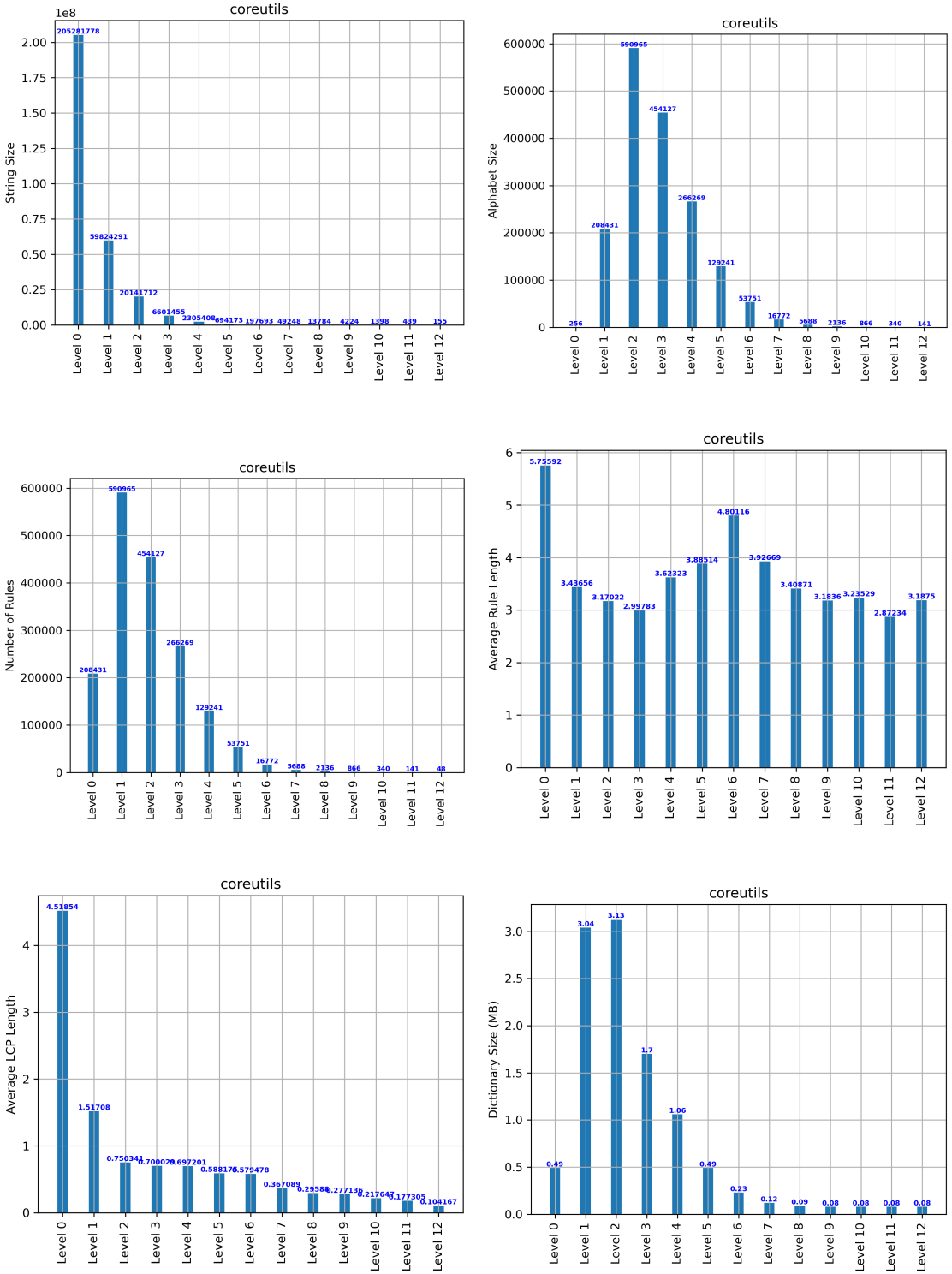


Figure 5.24: Empirical attributes for `coreutils` file.

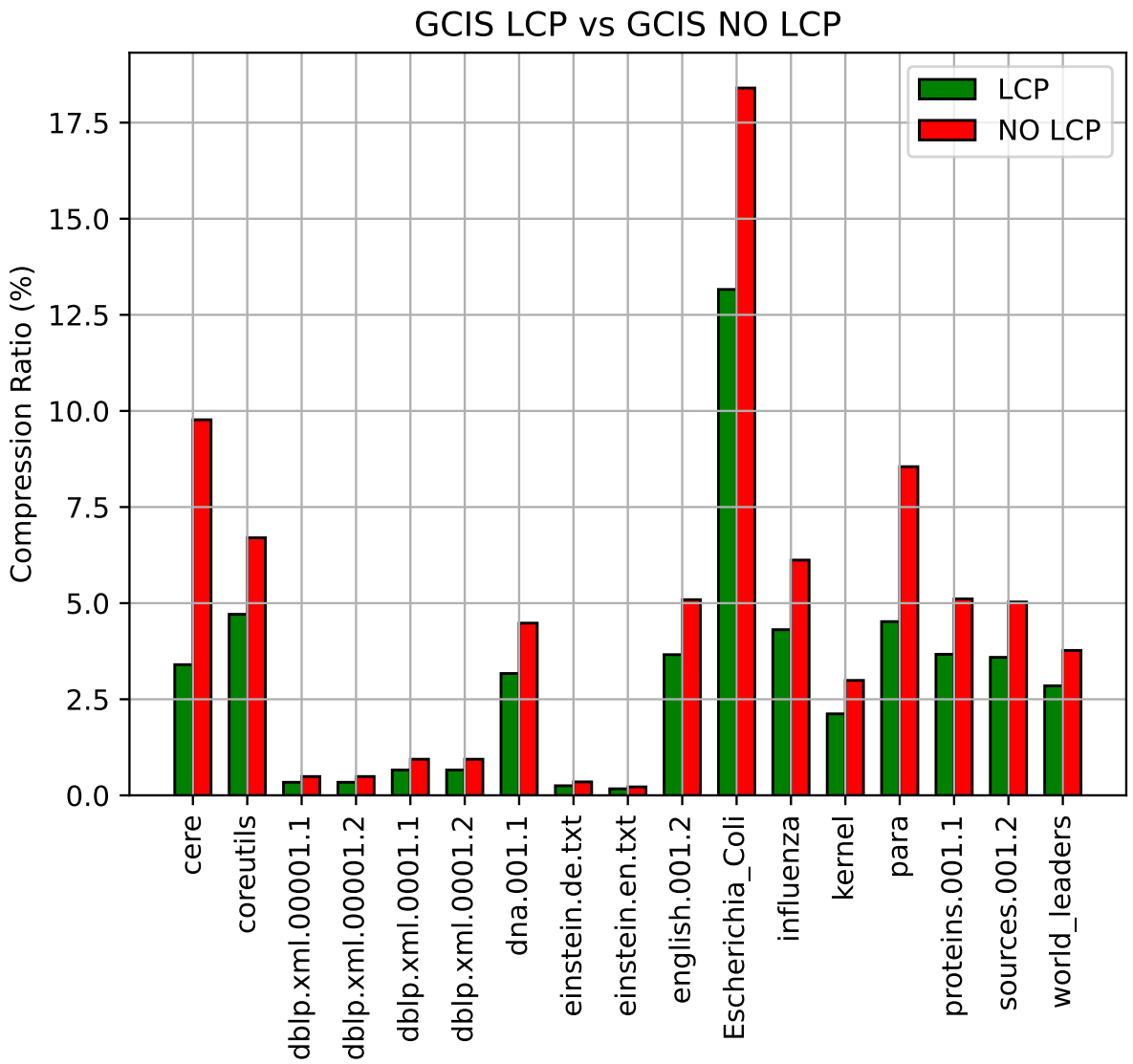


Figure 5.25: Comparison of compression ratios regarding GCIS with and without `lcp` compression.

Chapter 6

Final Considerations

We have introduced GCIS, a new grammar-based compression approach based on the induced suffix sorting framework of SAIS [Nong et al., 2009]. GCIS uses the meta-symbols introduced by SAIS to generate non-terminals of a balanced grammar that reproduces the original text. Our experiments on repetitive texts show that GCIS compresses 3–7 times faster than RePair and 7-zip. Compared to RePair, the grammar compressor that compresses the most, GCIS compresses using 3–5 times less memory, yet it obtains a compressed file twice as large (yet the absolute compression is still attractive, below 5% in most cases). GCIS decompresses 2–8 times slower than RePair and 7-zip, though.

Grammar-based compression is attractive because, unlike Lempel-Ziv, it can be enriched to support fast extraction of arbitrary text substrings. From this perspective and regarding the space-time relation, our experiments show that GCIS is a competitive option when compared to RePair-based extractors, being faster and less space-efficient than the succinct encoding of RePair extractors and slower, but more space-efficient, than the more straightforwardly encoded RePair extractors.

Finally, as a by-product of GCIS, the suffix array of the text can be obtained during the decompression algorithm, faster than decompressing and running on the original text.

All previously discussed features make GCIS especially attractive in scenarios where it is required to support random access on the compressed text. Grammar compression of very large files is challenging with RePair because of its large main memory footprint, for which GCIS offers an interesting alternative. Given its slowness at decompression, the GCIS grammar is best suited as a compressed data structure to be repeatedly accessed without decompressing it completely.

Future work will consider the enhancement of GCIS grammars, providing new features for them, or turning them into specialized grammars or even self-indices. Such work should consider evolutions or possible discoveries of new properties of GCIS like the following.

Parallelizing GCIS grammar construction could enhance the already fast compression time of GCIS. Another research question is if we can reduce memory usage by using induced suffix sorting disk-based algorithms, such as the one described in [Kärkkäinen et al., 2017].

It is possible to compute the suffix array during GCIS grammar decompression, but recovering arbitrary suffix array entries from the compressed information could be interesting, once it can allow the simulation of suffix trees in compressed space.

Combining GCIS with other methods, such as RePair, could lead to better compression ratios and times. One possible approach is to apply GCIS in the first recursion levels and change the approach for the deeper levels, where the compression achieved by GCIS is diminished.

Navarro [2021] establishes several measures for different paradigms of compression. One open question of GCIS relies on connecting the GCIS grammar size to such measures in asymptotic terms.

Akagi et al. [2021a] shows that when a single edit operation, *i.e.*, substitution, deletion, or insertion, is allowed, the GCIS grammar size for the modified text is at most 4 times the original grammar size. This work is not concerned with the structural changes needed to update the grammar after an operation, only with the compression ratio between the new grammar size and the old one. Hence, interesting research topics involve the development of an efficient grammar update algorithm for GCIS when an edit operation on the original text is allowed and verifying if such algorithm achieves good results in practical terms for repetitive texts.

Díaz-Domínguez and Navarro [2021] employ a procedure similar to GCIS for building a grammar capable of inferring the Burrows-Wheeler transform of the encoded text. This grammar specializes in treating a collection of genomic reads and showed fast random access while requiring a small footprint of memory during its construction and maintaining a competitive compression ratio.

Díaz-Domínguez et al. [2021] showed that GCIS based grammars have the property of being locally consistent, *i.e.*, equal substrings tend to be parsed in the same way. The authors proceeded to explain that when such grammars are used in conjunction with additional data structures described in [Claude and Navarro, 2012], they allow queries like extraction of substrings and pattern matching. These data structures consist of a succinct tree to encode the topology of the grammar tree and a grid, which tracks the so-called primary occurrences of each pattern that are used to find the secondary occurrences and demonstrated do be of extreme practical value [Claude et al., 2021]. Moreover, in conjunction with these data structures, the locally consistent property may be utilized to speed-up pattern matching significantly. This speed-up occurs because the number of

different pattern partitions to be queried by such data structures is only $O(\lg m)$, being m is the pattern length. The result is an index faster than other common alternatives, concerning pattern matching, at the cost of being slightly larger.

Akagi et al. [2021b] strategy uses a different approach: a generalized suffix tree over the expansion of all nonterminals of GCIS grammar is used to support pattern matching. As a result, this index is also very fast for locating patterns compared to its competitors regarding the space/time trade-off.

Despite the different strategies to allow self-indexing on GCIS grammars (regarding the approaches in Akagi et al. [2021b]; Díaz-Domínguez and Navarro [2021]; Díaz-Domínguez et al. [2021]), the outcome is similar: the resulting product demonstrated to be a practical alternative for aiding string processing.

References

Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical Compressed Suffix Trees. *Algorithms*, 6(2):319–351, 2013. doi:10.3390/a6020319. 19

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0. 7, 8

Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. 2021a. URL `https://arxiv.org/abs/2107.08615`. 70

Tooru Akagi, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar Index by Induced Suffix Sorting. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2021b. doi:10.1007/978-3-030-86692-1_8. 71

Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software Practice Experimental*, 40(2):131–147, 2010. doi:10.1002/spe.948. 19, 20

Djamal Belazzougui and Gonzalo Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Trans. on Algorithms*, 11(4):1–21, June 2015. doi:10.1145/2629339. 18

Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. Ultra-Large-Scale Repository Analysis via Graph Compression. In *Proc. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 184–194. IEEE, 2020. doi:10.1109/SANER48275.2020.9054827. 3

Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, 49(1):392–404, 2013. doi:10.1016/j.ipm.2012.08.003. 20

Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994. 9

Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116. 25

Francisco Claude and Gonzalo Navarro. Fast and Compact Web Graph Representations. *ACM Trans. Web*, 4(4):16:1–16:31, 2010. doi:10.1145/1841909.1841913. 26

Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011. doi:10.3233/FI-2011-565. 25

Francisco Claude and Gonzalo Navarro. Improved Grammar-Based Compressed Indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19. 25, 70

Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002. 18

Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *J. Computer and System Sciences*, 118:53–74, 2021. doi:10.1016/j.jcss.2020.12.001. 25, 70

Genome Reference Consortium. Genome reference consortium human reference 37. http://hgdownload.cse.ucsc.edu/goldenpath/hg19/chromosomes/, 2009. Accessed: 06/2020. 41

David Clark. *Compact PAT trees*. Phd, University of Waterloo, 1996. 18

Sebastian Deorowicz. Silesia corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia, 2003. Accessed: 06/2020. 41

D. Díaz-Domínguez and G. Navarro. A Grammar Compressor for Collections of Reads with Applications to the Construction of the BWT. In *Proc. IEEE Data Compression Conference (DCC)*, pages 93–102. IEEE, 2021. doi:10.1109/DCC50243.2021.00016. 70, 71

D. Díaz-Domínguez, G. Navarro, and A. Pacheco. An LMS-based Grammar Self-index with Local Consistency Properties. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science*, pages 100–113, 2021. doi:10.1007/978-3-030-86692-1_9. 70, 71

Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000. doi:10.1109/SFCS.2000.892127. 19

Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039. 17, 19

Paolo Ferragina and Gonzalo Navarro. Pizza-Chili Corpus. http://pizzachili.dcc.uchile.cl/texts.html, 2005a. Accessed: 06/2020. 41

Paolo Ferragina and Gonzalo Navarro. Pizza-Chili Repetitive Corpus. http://pizzachili.dcc.uchile.cl/repcorpus.html, 2005b. Accessed: 06/2020. 29, 41

Johannes Fischer. Inducing the LCP-Array. In *Proc. 12th Algorithms and Data Structures Symposium (WADS)*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2011. doi:10.1007/978-3-642-22300-6_32. 14, 15, 36

Johannes Fischer and Volker Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE): Revised Selected Papers*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007. doi:10.1007/978-3-540-74450-4_41. 15

Johannes Fischer and Florian Kurpicz. Dismantling DivSufSort. In *Proc. Prague Stringology Conference*, pages 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017. 61

Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. Mr-repair: Grammar compression based on maximal repeats. In *Proc. IEEE Data Compression Conference (DCC)*, pages 508–517. IEEE, 2019. doi:10.1109/DCC.2019.00059. 26

Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Scaling up RePair with Rsync. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 11811 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2019. doi:10.1007/978-3-030-32686-9_3. 26, 27, 44

Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, and Yoshimasa Takabatake. Practical Random Access to SLP-Compressed Texts. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12303 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 2020. doi:10.1007/978-3-030-59212-7_16. 44

Jean-Loup Gailly and Mark Adler. The gzip home page. http://www.gzip.org/. Accessed: 03/2017. 42

Michal Ganczorz and Artur Jez. Improvements on re-pair grammar compressor. In *Proc. IEEE Data Compression Conference (DCC)*, pages 181–190. IEEE, 2017. doi:10.1109/DCC.2017.52. 26

Genomics England. 100,000 genomes project. https://www.genomicsengland.co.uk/the-100000-genomes-project-by-numbers/, 2012. Accessed: 01/2022. 4

Simon Gog and Enno Ohlebusch. Fast and Lightweight LCP-Array Construction Algorithms. In *Proc. 13th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 25–34. SIAM, 2011. doi:10.1137/1.9781611972917.3. 37

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th Experimental Algorithms International Symposium (SEA)*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28. 45

Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. In *Information Retrieval*, pages 66–82. Prentice-Hall, Inc., 1992. ISBN 0-13-463837-9. URL https://dl.acm.org/doi/10.5555/129687.129692. 7

Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38. CTI Press and Ellinika Grammata, 2005. 18

Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD Factorization: Simple and Practical Online Grammar Compression with Variable-to-Fixed Encoding. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 2015. doi:10.1007/978-3-319-19929-0_19. 27, 43

Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. 32th ACM Symposium on Theory of Computing (STOC)*, pages 397–406. ACM, 2000. doi:10.1145/335305.335351. 18

Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Computing*, 35(2): 378–407, 2005. doi:10.1137/S0097539702402354. 18

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 841–850. ACM-SIAM, 2003. URL https://dl.acm.org/doi/10.5555/644108.644250. 18

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997. ISBN 0-521-58519-8. doi:10.1017/cbo9780511574931. 7

John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 6th Edition.* Morgan Kaufmann, 2019. ISBN 978-0-12-383872-8. 1, 2

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007. ISBN 978-0-321-47617-3. 24

Tomohiro I. Shaped SLP implementation. https://github.com/itomomoti/ShapedSlp, 2020. Accessed: 08/2020. 44

Hideo Itoh and Hozumi Tanaka. An Efficient Method for in Memory Construction of Suffix Arrays. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 81–88. IEEE, 1999. doi:10.1109/SPIRE.1999.796581. 9

Guy Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988. 3

Ming-Yang Kao, editor. *Encyclopedia of Algorithms.* Springer, 2016. ISBN 978-1-4939-2863-7. doi:10.1007/978-1-4939-2864-4. 3

Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17. 15, 36

Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In Sándor P. Fekete and Vijaya Ramachandran, editors, *Proc. of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017. doi:10.1137/1.9781611974768.8. 70

Pang Ko and Srinivas Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003. doi:10.1007/3-540-44888-8_15. 9

Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. Lempel-Ziv-Like Parsing in Small Space. *Algorithmica*, 82(11):3195–3215, 2020. doi:10.1007/s00453-020-00722-6. 28, 44

Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient Construction of a Complete Index for Pan-Genomics Read Alignment. *J. Computational Biology*, 27(4):500–513, 2020. doi:10.1089/cmb.2019.0309. URL https://doi.org/10.1089/cmb.2019.0309. 3, 4

Florian Kurpicz. Sais-lite suffix and LCP arrays construction algorithm. https://github.com/kurpicz/sais-lite-lcp, 2015. Accessed: 06/2020. 44, 61, 62

Florian Kurpicz. Divsufsort suffix and lcp arrays construction algorithm. https://github.com/kurpicz/libdivsufsort, 2016. Accessed: 06/2020. 44, 61, 62

Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice Experimental*, 29(13):1149–1171, 1999. 7

N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proc. IEEE Data Compression Conference (DCC)*, pages 296–305. IEEE, 1999. doi:10.1109/DCC.1999.755679. 25

Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324. 3

Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013. doi:10.1016/j.is.2013.06.006. 26

Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Optimal suffix sorting and LCP array construction for constant alphabets. *Information Processing Letters*, 118:30–34, 2017. doi:10.1016/j.ipl.2016.09.010. 36, 37

Felipe A. Louza, Neerja Mhaskar, and W. F. Smyth. A new approach to regular & indeterminate strings. *Theoretical Computer Science*, 854:105–115, 2021. doi:10.1016/j.tcs.2020.12.007. 14

Matt Mahoney. Large text compression benchmark. http://mattmahoney.net/dc/text.html, 2006. Accessed: 06/2020. 41

Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3537 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005. doi:10.1007/11496656_5. 18, 19

Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058. 7

Giovani Manzini. Manzini's lightweight corpus. http://people.unipmn.it/~manzini/lightweight/, 2003. Accessed: 06/2020. 41

Shirou Maruyama and Yasuo Tabei. Fully online grammar compression in constant space. In *Proc. IEEE Data Compression Conference (DCC)*, pages 173–182. IEEE, 2013. doi:10.1109/DCC.2014.69. 25, 26, 44

Yuta Mori. Divsufsort suffix array construction algorithm. https://github.com/y-256/libdivsufsort, 2008. Accessed: 06/2020. 44

Yuta Mori. Sais-lite suffix sorting algorithm. https://sites.google.com/site/yuta256/sais, 2010. Accessed: 06/2020. 44

Gonzalo Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1): 87–114, 2004. doi:10.1016/S1570-8667(03)00066-2. 19

Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. ISSN 1570-8667. doi:https://doi.org/10.1016/j.jda.2013.07.004. 23rd Annual Symposium on Combinatorial Pattern Matching. 18

Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 1st edition, 2016. ISBN 1107152380. 3, 18

Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2021. doi:10.1145/3434399. URL https://doi.org/10.1145/3434399. 16, 17, 70

NCBI. Salmonella enterica subsp. enterica serovar Paratyphi B str. SPB7, complete sequence. https://www.ncbi.nlm.nih.gov/nuccore/NC_010102, 2007. Accessed: 06/2020, NCBI Reference Sequence: NC 010102.1. 41

NCBI. Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome. https://www.ncbi.nlm.nih.gov/nuccore/NC_045512, 2020. Accessed: 06/2020, NCBI Reference Sequence: NC 045512.2. 41

Takaaki Nishimoto and Yasuo Tabei. LZRR: LZ77 parsing with right reference. In *Proc. IEEE Data Compression Conference (DCC)*, pages 211–220. IEEE, 2019. doi:10.1109/DCC.2019.00029. 29, 44

Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Information Systems*, 31(3):15, 2013. doi:10.1145/2493175.2493180. 14

Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *Proc. IEEE Data Compression Conference (DCC)*, pages 193–202. IEEE, 2009. doi:10.1109/DCC.2009.42. 9, 11, 14, 36, 62, 63, 69

Ge Nong, Sen Zhang, and Wai Hong Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. on Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188. 10

Daniel Saad Nogueira Nunes and Mauricio Ayala-Rincón. A compressed suffix tree based implementation with low peak memory usage. In *XXXIX Latin American Computer Conference: Selected Papers (CLEI)*, volume 302 of *Electronic Notes in Theoretical Computer Science*, pages 73–94. Elsevier, 2013. doi:10.1016/j.entcs.2014.01.021. 19

Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A Grammar Compression Algorithm Based on Induced Suffix Sorting. In *Proc. IEEE Data Compression Conference (DCC)*, pages 42–51. IEEE, 2018. doi:10.1109/DCC.2018.00012. 4, 25, 31

Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. Grammar compression by induced suffix sorting. *ACM J. Experimental Algorithmics (To appear)*, 2022. URL https://users.dcc.uchile.cl/~gnavarro/ps/jea21.2.pdf. 4, 31

Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. ISBN 978-3000413162. URL http://www.oldenbusch-verlag.de/. 7

Igor Pavlov. The 7zip home page. http://www.7-zip.org/. Accessed: 10/2017. 42

Alberto Ordóñez Pereira. *Statistical and repetition-based compressed data structures*. Phd, Universidad da Coruña, 2016. 3

Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. on Algorithms*, 3(4):43, November 2007. doi:10.1145/1290672.1290680. 18

Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x. 19

Julian Seward. The bzip home page. http://www.bzip.org/, 1996. Accessed: 3/2017. 42

Dmitry Shkarin. PPMd algorithm variant j revision 1. http://www.compression.ru/ds/, 2006. Accessed: 07/2020. 42

Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A space-optimal grammar compression. In *Proc. 25th Annual European Symposium on Algorithms (ESA)*, volume 87 of *LIPIcs*, pages 67:1–67:15, 2017. doi:10.4230/LIPIcs.ESA.2017.67. 42

Andrew Trigell. Andrew trigell's large corpus. https://www.samba.org/ftp/tridge/large-corpus/, 1998. Accessed: 06/2020. 41

Sebastiano Vigna. Quasi-succinct indices. In *Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92. ACM, 2013. doi:10.1145/2433396.2433409. 21

Raymond Wan. Offline Dictionary-based Compression (RePair, Recursive Pairing) . https://github.com/rwanwork/Re-Pair, 2014. Accessed: 06/2020. 42

Wikipedia. Wikipedia's Pages and Articles XML Dump. http://wikipedia.c3sl.ufpr.br/enwiki/20191120/, 11 2019. Accessed: 06/2020. 41

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999. ISBN 1-55860-570-3. 32

Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E. Hassan. A study of the performance of general compressors on log files. *Empirical Software Engineering*, 25(5):3043–3085, 2020. doi:10.1007/s10664-020-09822-x. 3

Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E. Hassan. Improving State-of-the-art Compression Techniques for Log Management Tools. *IEEE Transactions on Software Engineering*, 2021. doi:10.1109/TSE.2021.3069958. Early access. 3

Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Trans. Knowledge Data Engineering*, 27(7):1920–1948, 2015. doi:10.1109/TKDE.2015.2427795. 3

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714. 22

Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934. 23

Appendix A

Experimental Data

The following experimental data relates to the Graphs and Figures of Chapter 5.

A.1 Compression Ratio

Table A.1: Percentage compression-ratio on regular texts.

	Compression Ratio (%)							
Text	**gzip**	**bzip2**	**ppmdj**	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**
archive	27.56	20.14	18.25	14.88	25.94	34.76	26.99	41.75
emacs	27.5	21.66	21.03	17.55	31.67	44.99	33.18	52.8
linux	23.78	19.13	19.41	15.87	30	41.74	31.87	49.84
samba	28.74	20.93	20.27	6.99	16.73	23.54	22.01	27.19
spamfile	11.97	7.03	9.29	3.13	4.52	7.95	5.59	10
enwik8	36.52	29.01	26.28	25.01	41.04	51.79	38.21	59.9
enwik9	32.37	25.4	23.65	20.62	30.2	43.74	30.69	49.04
chr22.dna	26.94	24.43	23.19	21.92	44.37	45.21	34.7	42.07
etext99	37.69	27.58	25.02	24.45	38.92	49.02	37.43	54.51
gcc-3.0.tar	20.86	15.84	16.31	13	23.76	34.22	25.34	41.3
howto	32.21	25.86	23.83	21.73	38.57	49.02	38.24	56.42
jdk13c	10.9	7.03	9.11	5.9	8.27	13.03	8.89	16.88
linux-2.4.5.tar	23.02	18.5	18.85	14.86	27.93	40.42	30.7	47.88
rctail96	20.93	14.91	14.47	11.27	16.75	22.02	17.15	27.65
rfc	24.34	18.7	18.21	14.59	24.45	34.83	25.72	41.11
sprot34.dat	24.57	20.75	22.53	16.52	31.33	37.72	29.28	40.31
w3c2	14.55	9.85	11.28	6.36	9.98	16.39	13.19	20.58
dblp.xml	17.42	11.4	11.31	11.82	16.39	22.88	15.94	29.33
dna	28.12	25.76	24.02	22.56	39.61	45.22	35.14	41.59
english	37.94	28.35	25.57	18.3	24.34	37	31.36	39.59
pitches	30.29	35.73	34.69	24.91	58.23	72.76	60.13	68.42
sources	22.56	18.66	19.57	15.06	27.75	41.36	30.74	48.82
dickens	37.96	27.47	24.5	27.77	48.67	52.76	40.38	60.98
mozilla	37.19	34.98	33.08	26.28	63.63	74.61	56.36	75.81
mr	37.01	24.48	23.35	27.61	57.79	50.87	44.45	50.61
nci	9.54	5.4	8.34	5.85	8.47	10.59	8.19	12.65
ooffice	50.34	46.53	42.46	39.48	91.04	93.55	76.42	98.88
osdb	37.08	27.79	24.61	28.46	54.14	51.37	41.56	63.48
reymont	28.05	18.8	19.04	20.24	35.7	41.89	29.03	47.87
samba	25.27	21.06	21.1	17.85	38.83	44.91	36.52	53.02
sao	73.54	68.13	65.38	60.86	140.94	129.94	102.91	124.06
webster	29.43	20.85	18.5	21.04	34.26	44.36	29.69	48.29
x-ray	71.25	47.81	47.49	52.96	115.02	107.6	91.55	111.61
xml	12.95	8.25	11.32	9.07	16.46	20.74	15.7	25.98

Table A.2: Percentage compression-ratio on repetitive texts.

	Compression Ratio (%)						
Text	**7zip**	**RePair**	**GCIS**	**ReLZ**	**LZD**	**SOLCA**	**LZRR**
cere	1.05	1.86	3.4	5.9	4.39	4.08	5.54
coreutils	1.99	2.54	4.71	11.27	6.49	6.64	10.6
dblp.xml.00001.1	0.16	0.19	0.34	0.91	0.47	0.44	0.84
dblp.xml.00001.2	0.16	0.18	0.34	0.91	0.51	0.45	0.86
dblp.xml.0001.1	0.2	0.46	0.66	1.19	0.73	0.99	1.13
dblp.xml.0001.2	0.19	0.39	0.66	1.19	1.05	1.07	1.34
dna.001.1	0.52	2.43	3.17	4.71	2.97	4.44	4.54
einstein.de.txt	0.11	0.16	0.25	0.6	0.38	0.41	0.55
einstein.en.txt	0.07	0.1	0.17	0.31	0.17	0.22	0.29
english.001.2	0.55	2.41	3.66	5.12	5.53	6.02	5.77
Escherichia_Coli	4.43	9.6	13.16	29.51	15.46	15.19	27.86
influenza	1.55	3.26	4.31	7.95	4.52	6.14	7.44
kernel	0.82	1.1	2.12	4.92	2.94	2.72	4.6
para	1.24	2.74	4.52	8.69	6.17	5.6	8.22
proteins.001.1	0.6	2.64	3.67	5.42	3.35	4.93	5.2
sources.001.2	0.45	2.34	3.59	4.5	5.32	5.86	5.76
world_leaders	1.39	1.79	2.85	5.99	3.44	4.56	5.64

Table A.3: Percentage compression-ratio on very large texts.

	Compression Ratio (%)					
Text	**7zip**	**GCIS**	**ReLZ**	**BigRePair**	**LZD**	**SOLCA**
c050	0.72	5.34	5.11	3.74	4.35	5.47
c100	0.52	4.77	4.08	3.14	1.53	4.82
c150	0.46	4.56	3.73	2.94	0.51	4.52
c200	0.42	4.4	3.56	2.75	1.15	4.33
c250	0.4	4.35	3.46	2.62	0.67	4.21
c300	0.39	4.31	3.39	2.52	0.32	4.1
c350	0.38	4.25	3.34	2.51	0.69	4.02
covid100000	0.57	0.66	1.67	0.68	0.74	0.96
covid200000	0.58	0.63	1.64	0.63	0.22	0.88
covid300000	0.58	0.64	1.63	0.59	0.04	0.85
covid400000	0.58	0.63	1.62	0.57	0.21	0.83
covid500000	0.58	0.64	1.62	0.56	0.11	0.81
covid600000	0.58	0.64	1.62	0.55	0.04	0.8
covid700000	0.58	0.64	1.62	0.54	0.13	0.79
enwiki-20191120-20G	19.2	39.28	88.9	–	–	–
salmonela1000	0.31	2.87	3.21	1.98	0.35	3.29
salmonela2000	0.3	2.62	3.14	1.74	0.29	2.92
salmonela3000	0.3	2.44	3.11	1.59	0.26	2.69
salmonela4000	0.3	2.25	3.1	1.48	0.25	2.52

A.2 Compression Speed

Table A.4: Compression Speed on regular texts.

	Compression Speed (MB/s)							
Text	**gzip**	**bzip2**	**ppmdj**	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**
archive	26.8	9.67	8.43	3.82	1.56	12.59	10.49	1.77
emacs	21.28	11.69	8.37	2.68	1.09	12.17	6.74	1.47
linux	27.67	10.92	9.46	3.01	1.16	13.19	8	1.56
samba	23.9	12.38	6.19	4.8	1.39	14.19	10.66	2.21
spamfile	53.3	9.23	17.05	8.68	1.99	12.74	45.04	5.19
enwik8	19.84	12.36	8.26	1.55	0.76	10.08	5.03	1.15
enwik9	23.49	12.24	9.02	1.25	0.15	8.96	4.91	1.15
chr22.dna	5.89	12.04	16.53	1.17	1.62	12.52	6.08	1.81
etext99	15.6	11.95	11.39	1.44	0.72	10.01	5.1	1.29
gcc-3.0.tar	33.45	11.22	11.24	3.14	1.1	13.07	9.25	1.87
howto	18.42	11.77	8.41	2.21	1.12	11.84	5.95	1.34
jdk13c	50.53	9.24	17.48	6.4	1.87	13.18	28.35	3.63
linux-2.4.5.tar	30.27	11.52	9.95	2.82	0.73	12.35	7.57	1.6
rctail96	34.76	7.76	13.59	3.12	1.25	11.55	11.45	2.45
rfc	25.36	13.11	11.19	2.29	0.94	12.24	7.15	1.77
sprot34.dat	35.59	11.04	7.64	2.91	0.94	11.2	7.86	1.76
w3c2	46.52	9.77	13.24	6.28	1.37	12.66	19.3	3.24
dblp.xml	46.49	9.61	13.9	3.29	1.04	11.31	11.56	2.23
dna	5.77	11.78	18.07	0.86	0.46	10.08	4.46	1.55
english	15.85	11.96	11.29	0.92	0.03	4.79	4.04	1.17
pitches	31.37	10.06	4.34	3.09	0.5	12.19	4.59	1.13
sources	30.08	11.45	10.05	2.48	0.49	11.01	7.41	1.58
dickens	11.58	9.52	9.1	1.92	1.75	14.15	6.93	1.36
mozilla	20.57	12.14	2.5	2.82	0.87	12.8	3.25	1.07
mr	12.62	12.31	4.91	2.92	2.26	18.81	5.25	1.28
nci	45.96	6.63	17.47	6.02	2.69	22.98	26.84	5.2
ooffice	12.55	8.79	1.92	2.89	1.59	14.3	3.04	0.8
osdb	19.78	9.61	3.06	3.02	1.87	14.01	4.39	1.19
reymont	10.36	9.34	9.08	2.22	2.11	18.94	8.29	1.76
samba	26.35	11.49	5.38	4.09	1.59	15.55	6.59	1.51
sao	10.07	7.4	1.1	3.19	1.5	8.63	1.51	0.7
webster	22.29	10.5	11.98	2.16	1.46	11.58	7.4	1.63
x-ray	15.4	12.1	1.47	3.42	1.47	13.23	2.47	0.73
xml	26.75	6.52	9.73	5.82	2.35	25.48	26.75	2.97

Table A.5: Compression speed on repetitive texts.

	Compression Speed (MB/s)						
Text	**7zip**	**RePair**	**GCIS**	**ReLZ**	**LZD**	**SOLCA**	**LZRR**
cere	1.54	1.28	12.21	7.62	40.29	4.58	2.38
coreutils	4.98	1.43	12.34	8.64	35.83	4.29	2.68
dblp.xml.00001.1	7.5	1.97	13.27	8.7	262.15	12.99	2.51
dblp.xml.00001.2	7.41	1.96	12.82	9.08	238.32	13.16	2.59
dblp.xml.0001.1	7.45	1.94	12.87	8.62	187.25	10.84	2.52
dblp.xml.0001.2	6.98	1.91	12.65	9.12	145.64	11.31	2.58
dna.001.1	2.92	1.92	13.21	8.05	62.05	8.64	2.47
einstein.de.txt	9.64	2.08	12.64	9.37	403.3	13.03	3.15
einstein.en.txt	9.38	2.01	10.91	8.11	392.97	12.24	2.87
english.001.2	4.83	1.63	11.34	8.19	38.27	5.48	2.48
Escherichia_Coli	1.55	1.16	11.84	7.38	13.68	2.93	2.51
influenza	5.6	1.92	12.59	7.99	44.87	6.52	2.9
kernel	4.18	1.43	11.81	8.66	74.13	4.34	2.66
para	1.45	1.17	11.97	7.39	29.24	3.86	1.6
proteins.001.1	9.33	1.78	11.13	7.51	62.79	7.48	2.51
sources.001.2	5.99	1.72	12.47	9.43	34.49	6.01	2.51
world_leaders	8.57	2.64	20.25	15.81	66.15	8.33	2.35

Table A.6: Compression speed on very large texts.

	Compression Speed (MB/s)					
Text	**7zip**	**GCIS**	**ReLZ**	**BigRePair**	**LZD**	**SOLCA**
c050	0.88	4.4	10.03	5.05	16.85	1.84
c100	0.96	4.28	12.27	2.5	43.49	1.8
c150	1.02	4.11	13.16	4.8	112.94	1.87
c200	0.88	4.13	13.66	2.69	30.9	1.73
c250	1	4.03	14.04	1.28	77.94	1.83
c300	0.89	3.97	13.59	0.37	169.07	1.73
c350	0.98	3.79	14.33	0.14	73.36	1.66
covid100000	4.79	5.94	7.59	10.84	18.18	6.56
covid200000	4.84	5.81	9.32	11.23	77.3	5.29
covid300000	4.84	5.78	10.14	11.15	262.77	5.38
covid400000	4.87	4.9	10.58	11.28	49.73	4.91
covid500000	4.85	4.6	10.8	11.06	113.65	5.7
covid600000	4.93	4.77	11.02	11.21	240.47	5.52
covid700000	4.85	4.84	11.12	11.07	75.38	5.26
enwiki-20191120-20G	0.89	2.17	0.71	–	–	–
salmonela1000	1.26	4.91	9.8	6.39	170.13	2.22
salmonela2000	1.37	4.22	10.94	6.58	137.96	1.93
salmonela3000	1.38	3.7	11.2	4.86	162.18	2.03
salmonela4000	1.39	3.95	11.56	2.56	168.92	2.07

A.3 Decompression Speed

Table A.7: Decompression speed on regular texts.

	Compression Ratio (MB/s)							
Text	**gzip**	**bzip2**	**ppmdj**	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**
archive	96.68	23.54	7.73	75.19	84.59	45.88	61.52	6.82
emacs	135.6	26.97	8.2	98.88	103.17	49.44	79.1	7.36
linux	80.68	36.62	8.45	72.12	80.68	38.7	67.04	7.26
samba	68.16	31.98	5.49	166.32	90.39	55.44	96.7	7.55
spamfile	131.59	42.54	14.11	210.55	161.96	56.15	107.97	7.52
enwik8	101.01	25.51	7.56	70.42	83.33	32.47	58.48	6.73
enwik9	118.06	26.78	8.22	86.36	58.24	31.74	50.13	7.05
chr22.dna	95.97	27.2	14.4	63.98	76.78	35.26	86.37	7.32
etext99	101.23	26.25	10.19	60.86	81.61	31.06	64.59	6.82
gcc-3.0.tar	123.76	35.8	10.13	117.07	120.32	43.31	76.66	7.53
howto	59.73	26.82	7.93	78.84	83.87	39.82	75.81	6.93
jdk13c	154.96	43.31	14.74	151.59	162.16	50.17	124.52	7.66
linux-2.4.5.tar	166.07	38.62	9	79.62	67.98	39.81	75.49	7.33
rctail96	88.24	32.31	12.28	100.62	110.3	42.8	90.32	7.35
rfc	125.18	37.19	9.8	79.74	87.53	40.99	72.76	7.09
sprot34.dat	154.39	26.29	6.87	68.51	105.4	36.06	78.86	7.15
w3c2	151.01	39.17	11.73	182.81	160.31	47.8	80.78	7.24
dblp.xml	167.31	33.35	12.43	129.32	148.81	44.27	67.61	7.55
dna	134.64	24.25	15.42	82.77	95.95	33.72	56.97	7.91
english	115.91	25.2	10.1	96.9	54.79	31.15	64.73	7.01
pitches	136.17	30.34	4.01	62.73	65.68	27.91	73.46	6.79
sources	150.62	34.63	9.11	109.26	111.57	41.19	55.79	7.2
dickens	67.93	19.98	8.49	46.32	84.92	35.14	46.32	5.89
mozilla	98.5	27.39	2.2	62.46	96.64	43.78	58.87	7.08
mr	71.21	29.32	4.91	62.31	124.62	55.39	90.64	7.98
nci	152.5	34.59	17.47	223.67	279.58	95.86	145.87	8.56
ooffice	55.91	15.38	1.79	26.74	61.5	23.65	61.5	6.91
osdb	72.07	19.78	2.9	38.81	100.9	32.55	67.27	7.26
reymont	73.67	22.1	9.21	94.71	132.6	51	82.88	7.05
samba	102.9	29.6	4.85	90.04	127.12	52.71	86.44	7.53
sao	80.56	15.43	0.97	18.59	55.77	24.17	36.25	5.49
webster	101.12	27.83	11.58	65.81	109.11	48.78	81.29	6.2
x-ray	52.94	16.94	1.27	21.18	60.5	30.25	38.5	5.29
xml	133.75	35.67	9.73	66.88	133.75	53.5	59.44	6.37

Table A.8: Decompression speed on repetitive texts.

	Decompression Speed (MB/s)					
Text	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**	**LZRR**
cere	475.56	67.15	50.19	68.85	8.7	27.72
coreutils	366.57	99.17	50.44	68.2	7.46	22.61
dblp.xml.00001.1	551.89	187.25	63.94	111.55	7.68	24.73
dblp.xml.00001.2	388.37	145.64	63.94	113.98	7.8	24.67
dblp.xml.0001.1	582.56	183.96	65.54	115.23	7.8	25.51
dblp.xml.0001.2	551.89	161.32	64.73	107	7.78	23.25
dna.001.1	524.3	109.23	59.24	95.33	8.31	25.45
einstein.de.txt	545.65	265.03	59.85	99.74	7.69	31.34
einstein.en.txt	607.31	251.41	51.28	71.61	8.04	30.54
english.001.2	361.59	63.17	46.4	69.44	7.41	13.94
Escherichia_Coli	201.23	62.26	55.51	86.02	7.98	18.78
influenza	396.95	241.89	56.09	78.19	7.91	27.69
kernel	515.92	86.86	47.33	66.83	7.82	29.05
para	451.86	65.24	50.03	63.6	8.48	22.22
proteins.001.1	455.91	91.18	49.46	88.86	7.66	21.36
sources.001.2	499.33	82.57	54.9	72.82	7.45	16.31
world_leaders	276.29	126.95	97.85	97.85	7.85	22.05

Table A.9: Decompression speed on very large texts.

	Decompression Speed (MB/s)				
Text	**7zip**	**GCIS**	**BigRePair**	**LZD**	**SOLCA**
c050	350.71	36.03	28.14	50.96	5.98
c100	393.41	34.38	25.18	221.87	6.06
c150	386.8	33.35	25.09	1732.29	5.81
c200	385.71	32.8	25.26	226.81	6
c250	322.55	32.79	27.64	480.41	6.02
c300	343.17	32.27	25.02	1887.09	5.95
c350	326.42	32.47	27.6	312	6.03
covid100000	280.78	55.81	37.32	85.8	6.73
covid200000	332.26	54.99	36.04	320.16	6.93
covid300000	347.31	53.92	37.33	2354.57	6.94
covid400000	324.94	54.05	37.81	335.42	6.42
covid500000	315.83	54.01	38.28	599.74	6.95
covid600000	294.47	53.87	38.29	2481.58	7.16
covid700000	272.41	52.42	36.81	561.03	7.13
enwiki-20191120-20G	60.12	19.11	–	–	–
salmonela1000	410.73	35.27	32.49	532.19	6.96
salmonela2000	385.01	31.74	32.16	579.13	6.73
salmonela3000	324.86	31.72	33.09	639.05	6.96
salmonela4000	312.27	31.78	33.11	620.55	7.04

A.4 Peak Memory

A.4.1 Compression

Table A.10: Peak memory in MB during compression on regular texts.

	Peak Memory (MB)							
Text	**gzip**	**bzip2**	**ppmdj**	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**
archive	8	15	19	316	723	159	195	99
emacs	8	14	19	495	1279	291	391	167
linux	8	15	19	497	1281	285	380	169
samba	8	14	19	267	584	133	195	99
spamfile	8	14	18	880	2044	446	200	74
enwik8	8	15	19	1024	2598	612	884	378
enwik9	8	15	19	9851	25186	5657	6685	2385
chr22.dna	8	15	8	381	926	205	343	124
etext99	8	14	19	1067	2776	634	935	375
gcc-3.0.tar	8	15	19	905	2274	497	555	232
howto	8	15	19	425	1080	234	373	163
jdk13c	8	14	19	758	1724	376	224	96
linux-2.4.5.tar	8	15	19	1166	3080	682	849	301
rctail96	8	14	19	1151	2898	631	540	203
rfc	8	14	19	1168	3126	663	752	280
sprot34.dat	8	14	19	1107	2897	636	761	258
w3c2	8	15	19	1057	2624	565	402	150
dblp.xml	8	14	19	3138	7368	1611	1269	457
dna	8	15	9	4087	10560	2356	3551	842
english	8	15	19	10789	59687	21409	14741	3857
pitches	8	15	19	568	1646	361	718	281
sources	8	15	19	2130	5534	1229	1501	493
dickens	8	14	13	134	277	70	114	71
mozilla	8	15	19	526	1449	336	596	254
mr	8	14	19	132	292	66	111	60
nci	8	14	13	369	946	185	109	54
ooffice	8	15	19	79	194	46	111	70
osdb	8	14	19	132	276	65	105	66
reymont	8	14	13	86	185	48	59	51
samba	8	14	19	267	584	133	195	99
sao	8	14	19	91	207	58	160	81
webster	8	14	19	442	1081	243	327	141
x-ray	8	14	19	120	256	74	166	97
xml	8	14	13	75	148	37	26	40

Table A.11: Peak memory in MB during compression on repetitive texts.

	Peak Memory (MB)						
Text	**7zip**	**RePair**	**GCIS**	**ReLZ**	**LZD**	**SOLCA**	**LZRR**
cere	4591	11680	2362	4074	911	154	18933
coreutils	2077	5130	1060	1822	492	108	8450
dblp.xml.00001.1	1064	2524	540	933	123	35	4316
dblp.xml.00001.2	1062	2524	540	933	124	35	4313
dblp.xml.0001.1	1062	2525	540	933	128	38	4311
dblp.xml.0001.2	1062	2527	540	933	137	38	4311
dna.001.1	1065	2623	542	934	191	59	4310
einstein.de.txt	952	2225	479	827	107	35	3816
einstein.en.txt	4640	11137	2376	4122	485	37	19201
english.001.2	1061	2582	545	934	244	74	4310
Escherichia_Coli	1133	3006	605	1010	521	154	4638
influenza	1633	3926	800	1375	338	94	6359
kernel	2542	6266	1323	2282	427	82	10604
para	4309	10961	2208	3792	1013	177	17618
proteins.001.1	1062	2514	544	934	191	64	4310
sources.001.2	1063	2596	544	934	236	65	4310
world_leaders	485	1307	247	425	95	45	1947

Table A.12: Peak memory in MB during compression on very large texts.

	Peak Memory (MB)					
Text	**7zip**	**GCIS**	**ReLZ**	**BigRePair**	**LZD**	**SOLCA**
c050	10791	26803	7396	10182	5506	836
c100	10791	53472	7397	10354	7647	1386
c150	10791	80147	7397	10505	9635	1668
c200	10791	106914	7397	11619	14314	2243
c250	10791	133602	7396	14569	16471	2243
c300	10791	160375	7396	17315	18519	2725
c350	10791	186983	7396	20083	23120	3422
covid100000	10791	26842	7393	2327	3435	161
covid200000	10791	53690	7393	4201	6164	269
covid300000	10791	80525	7394	6038	8867	351
covid400000	10791	107357	7394	7853	12248	488
covid500000	10791	134185	7393	9659	14983	631
covid600000	10791	161038	7394	9612	17697	703
covid700000	10791	187855	7394	10258	21061	704
enwiki-20191120-20G	10791	203980	21425	–	–	–
salmonela1000	10791	43805	7396	10187	5143	705
salmonela2000	10791	87566	7397	10459	10124	1015
salmonela3000	10791	131320	7397	10595	15083	1463
salmonela4000	10791	175007	7396	10665	20023	1762

A.4.2 Decompression

Table A.13: Peak memory in MB during decompression on regular texts.

	Peak Memory (MB)							
Text	**gzip**	**bzip2**	**ppmdj**	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**
archive	8	11	19	31	33	60	29	25
emacs	8	11	19	58	55	111	51	45
linux	8	11	19	51	53	106	49	41
samba	8	11	19	26	29	58	30	25
spamfile	8	11	18	66	26	148	22	22
enwik8	8	11	19	102	68	241	105	80
enwik9	8	12	19	988	399	2246	706	536
chr22.dna	8	11	10	46	25	74	41	30
etext99	8	11	19	112	85	294	108	84
gcc-3.0.tar	8	11	19	86	69	192	66	66
howto	8	11	19	44	47	95	49	41
jdk13c	8	11	19	77	30	135	26	26
linux-2.4.5.tar	8	11	19	126	107	266	99	80
rctail96	8	11	19	115	67	234	60	52
rfc	8	11	19	122	86	275	86	77
sprot34.dat	8	11	19	117	81	239	91	72
w3c2	8	11	19	110	64	216	45	41
dblp.xml	8	11	19	296	94	625	127	129
dna	8	12	9	401	156	828	344	214
english	8	12	19	1060	2568	4775	1523	913
pitches	8	11	19	61	114	163	94	71
sources	8	11	19	210	169	471	168	149
dickens	8	11	13	18	16	30	20	16
mozilla	8	11	19	62	53	162	83	66
mr	8	11	19	19	16	31	21	16
nci	8	11	11	28	18	56	17	15
ooffice	8	11	19	16	17	21	22	17
osdb	8	11	19	18	15	33	21	16
reymont	8	11	11	13	13	19	14	12
samba	8	11	19	26	29	58	30	25
sao	8	11	19	18	16	39	29	23
webster	8	11	18	46	30	109	42	39
x-ray	8	11	19	17	21	33	30	22
xml	8	11	13	12	13	11	11	10

Table A.14: Peak memory in MB during decompression on repetitive texts.

	Peak Memory (MB)					
Text	**7zip**	**RePair**	**GCIS**	**LZD**	**SOLCA**	**LZRR**
cere	402	110	713	62	38	8142
coreutils	186	81	360	45	27	3638
dblp.xml.00001.1	54	12	162	10	9	1853
dblp.xml.00001.2	54	13	163	10	9	1853
dblp.xml.0001.1	54	13	167	11	10	1853
dblp.xml.0001.2	62	15	153	12	10	1853
dna.001.1	107	20	159	18	15	1857
einstein.de.txt	56	11	138	10	9	1640
einstein.en.txt	456	13	720	11	10	8230
english.001.2	110	39	178	25	18	1858
Escherichia_Coli	114	88	193	55	37	2020
influenza	128	35	234	28	22	2741
kernel	246	51	428	30	18	4555
para	383	130	672	77	43	7589
proteins.001.1	102	22	187	19	18	1857
sources.001.2	109	37	181	25	17	1858
world_leaders	48	17	63	14	11	837

Table A.15: Peak memory in MB during decompression on very large texts.

	Peak Memory (MB)				
Text	**7zip**	**GCIS**	**BigRePair**	**LZD**	**SOLCA**
c050	1063	4898	168	320	206
c100	1063	9867	250	233	332
c150	1063	14748	323	127	532
c200	1063	19623	389	337	570
c250	1063	24951	452	252	667
c300	1063	29901	511	154	767
c350	1063	34846	568	354	1024
covid100000	1063	4231	35	72	47
covid200000	1063	8454	41	51	90
covid300000	1063	12992	47	25	122
covid400000	1063	17315	52	78	144
covid500000	1063	21617	56	57	155
covid600000	1063	25941	61	33	187
covid700000	1063	30235	65	84	209
enwiki-20191120-20G	1063	48045	0	0	0
salmonela1000	1063	7257	125	59	211
salmonela2000	1063	14852	187	87	330
salmonela3000	1063	22200	235	112	437
salmonela4000	1063	29595	274	135	532

A.5 Suffix and LCP Arrays Construction under Decompression

Table A.16: Suffix array construction under decompression on repetitive texts.

Text	**GCIS (s)**	**sais-lite (s)**	**divsufsort (s)**
cere	42.56	54.5	66.66
coreutils	19.26	24.57	28.86
dblp.xml.00001.1	9.12	11.1	13.59
dblp.xml.00001.2	9.08	11.1	13.45
dblp.xml.0001.1	9.09	11.17	13.2
dblp.xml.0001.2	9.14	11.29	13.28
dna.001.1	9.56	12.12	14.08
einstein.de.txt	8.4	10.57	12.23
einstein.en.txt	44.65	56.01	66.31
english.001.2	10.9	13.61	15.49
Escherichia_Coli	11.79	14.85	17.09
influenza	14.19	17.1	22.39
kernel	23.77	30.59	35.61
para	41.14	52.63	63.78
proteins.001.1	10.95	13.7	16.31
sources.001.2	9.68	12.33	13.58
world_leaders	2.76	3.35	3.19

Table A.17: Suffix + LCP arrays construction under decompression on repetitive texts.

Text	**GCIS (s)**	**sais-lite (s)**	**divsufsort (s)**
cere	62.44	74.59	80.58
coreutils	27.15	33.02	34.09
dblp.xml.00001.1	13.26	15.38	16.16
dblp.xml.00001.2	13.23	15.52	16.65
dblp.xml.0001.1	13.28	15.62	16.49
dblp.xml.0001.2	13.32	15.94	16.82
dna.001.1	13.78	16.27	17.04
einstein.de.txt	11.98	14.49	14.6
einstein.en.txt	64.49	77.38	79.49
english.001.2	15.07	18.34	18.87
Escherichia_Coli	17.03	20.26	20.92
influenza	20.88	24.9	25.79
kernel	33.53	41.37	42.98
para	59.83	72.06	77.64
proteins.001.1	15.33	18.34	19.32
sources.001.2	13.83	17.07	16.79
world_leaders	4.19	5.12	4.25

A.6 Extraction

Since we have many points regarding extraction the data can be found at https://github.com/danielsaad/GCIS-EXTRACT-DATA.

A.7 Empirical Atributes

Figures A.1 to A.16 depict the empirical attributes of GCIS grammar for the different texts in `pizza-chili-repetitive` *corpus*.

A.7.1 LCP Compression Contribution

Table A.18 shows the compression ratios over `pizza-chili-repetitive` *corpus* for the GCIS variants with and without `lcp` compression with the quotient between such compression ratios.

Table A.18: Differences between GCIS with and without `lcp` compression regarding compression ratio.

	Compression Ratio (%)		
Text	**GCIS**	**GCIS NO LCP**	**Quotient**
cere	3.4	9.77	0.35
coreutils	4.71	6.7	0.71
dblp.xml.00001.1	0.34	0.49	0.7
dblp.xml.00001.2	0.34	0.49	0.7
dblp.xml.0001.1	0.66	0.94	0.71
dblp.xml.0001.2	0.66	0.94	0.71
dna.001.1	3.17	4.48	0.71
einstein.de.txt	0.25	0.35	0.72
einstein.en.txt	0.17	0.22	0.78
english.001.2	3.66	5.09	0.72
Escherichia_Coli	13.16	18.4	0.72
influenza	4.31	6.12	0.71
kernel	2.12	2.99	0.71
para	4.52	8.55	0.53
proteins.001.1	3.67	5.11	0.72
sources.001.2	3.59	5.03	0.72
world_leaders	2.85	3.77	0.76

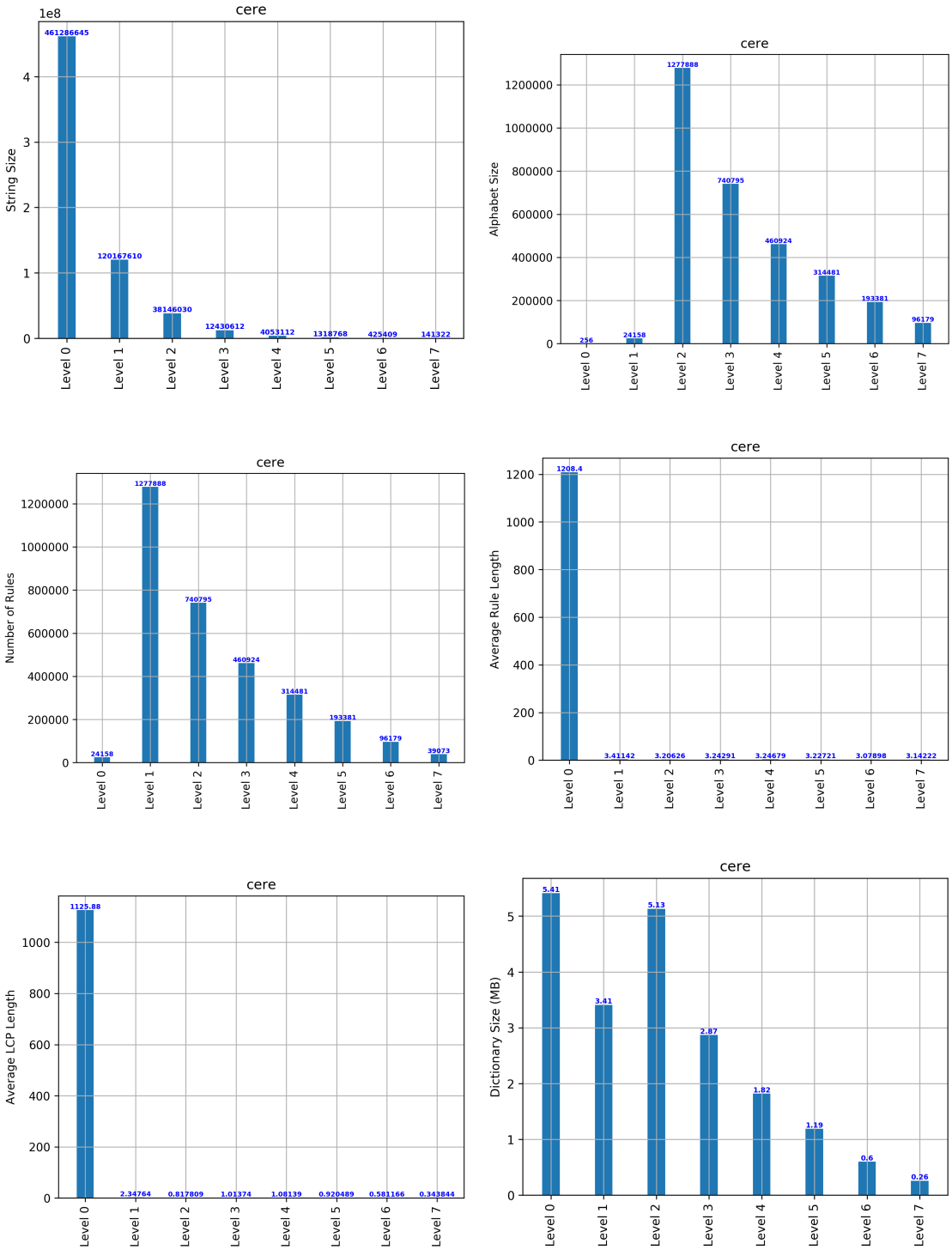


Figure A.1: Empirical attributes for `cere` file.

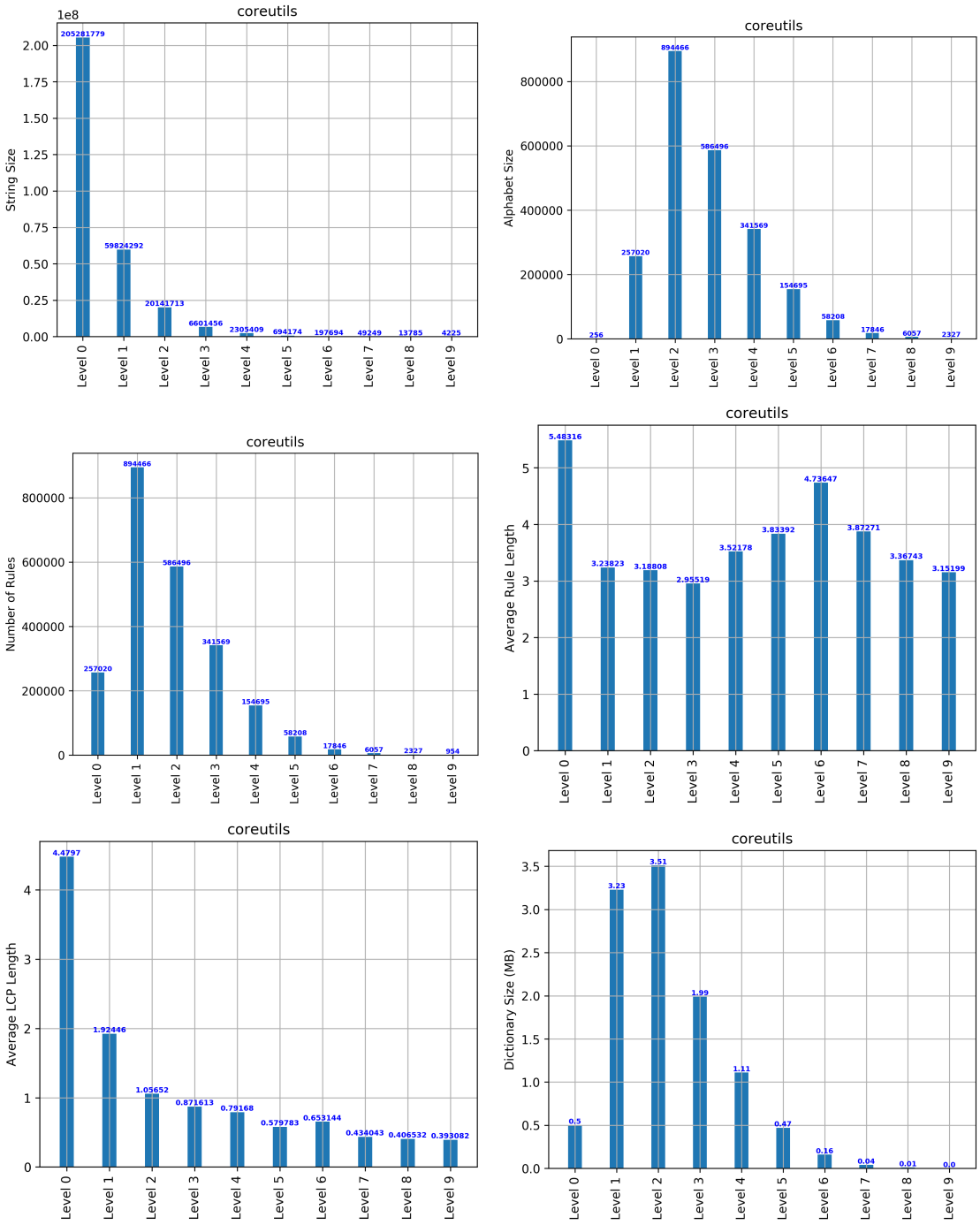


Figure A.2: Empirical attributes for `coreutils` file.

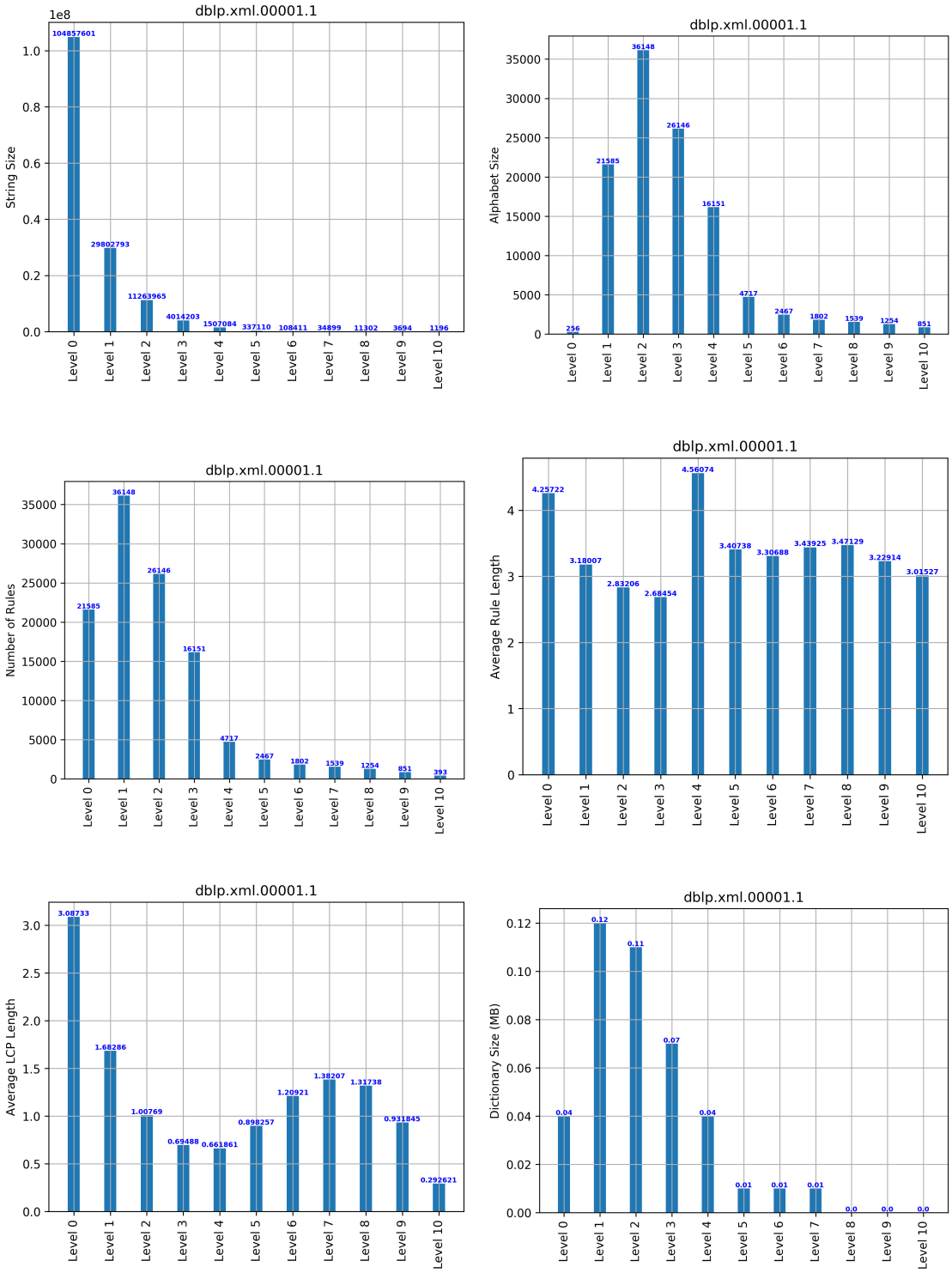


Figure A.3: Empirical attributes for `dblp.xml.00001.1` file.

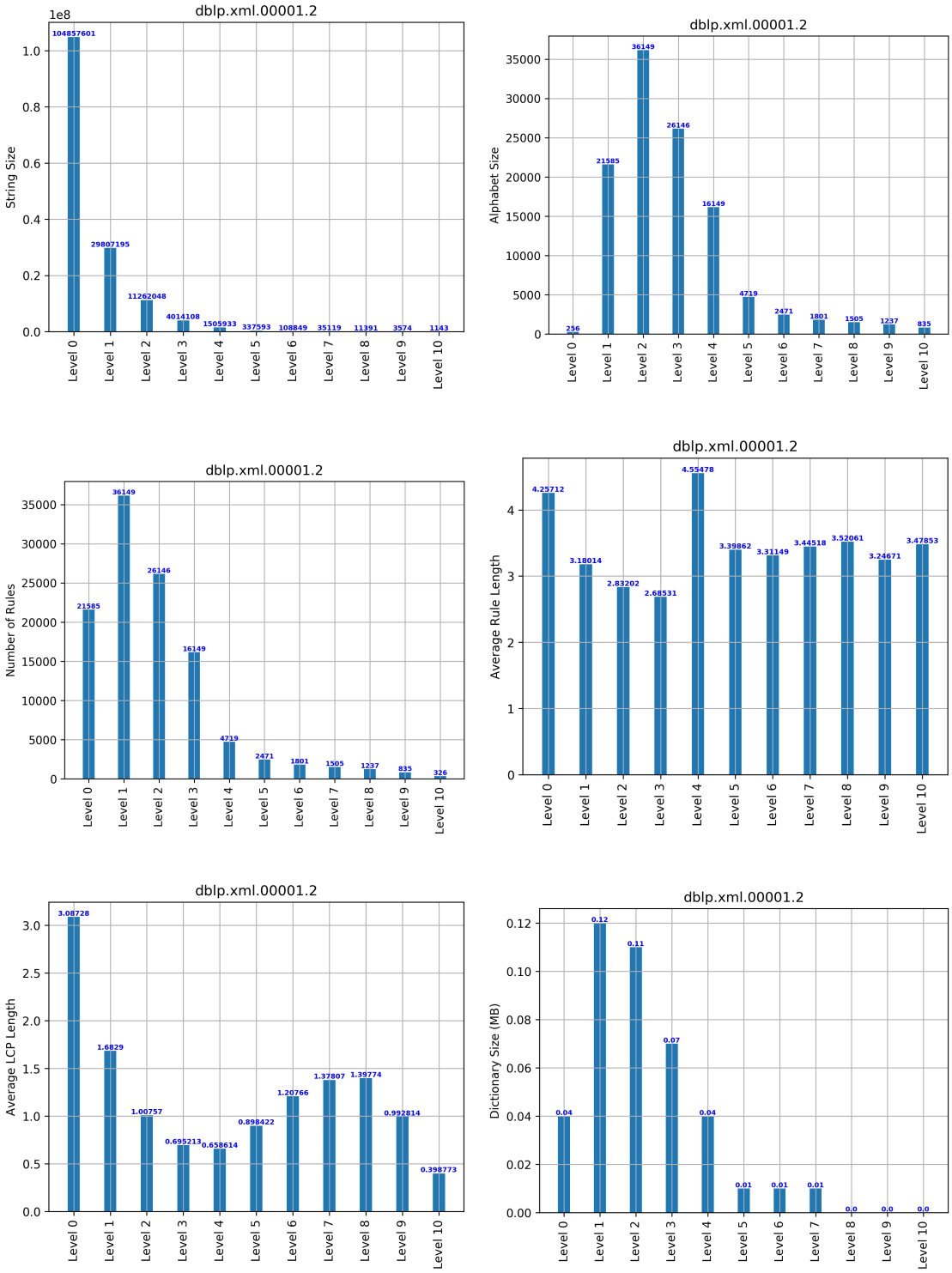


Figure A.4: Empirical attributes for `dblp.xml.00001.2` file.

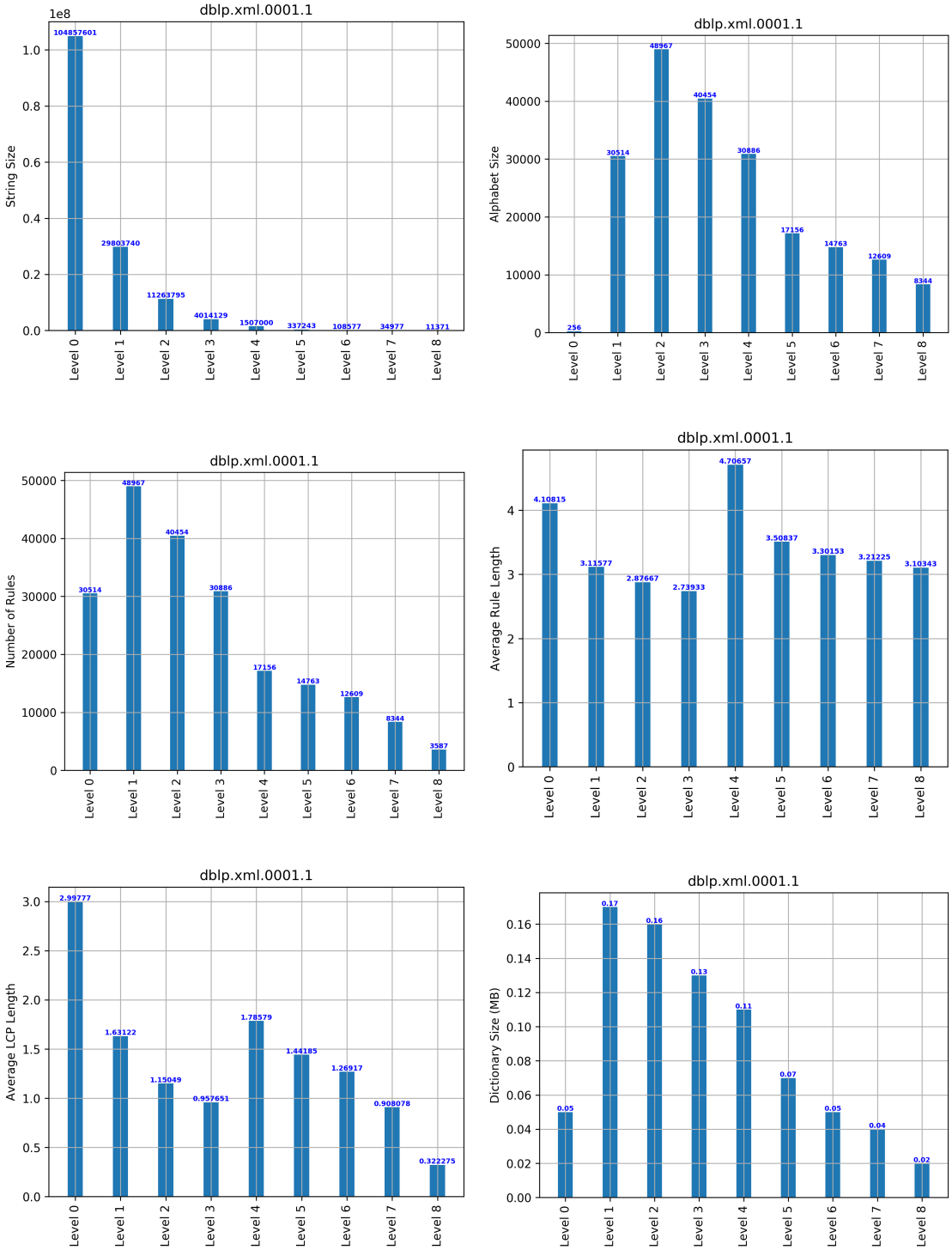


Figure A.5: Empirical attributes for `dblp.xml.0001.1` file.

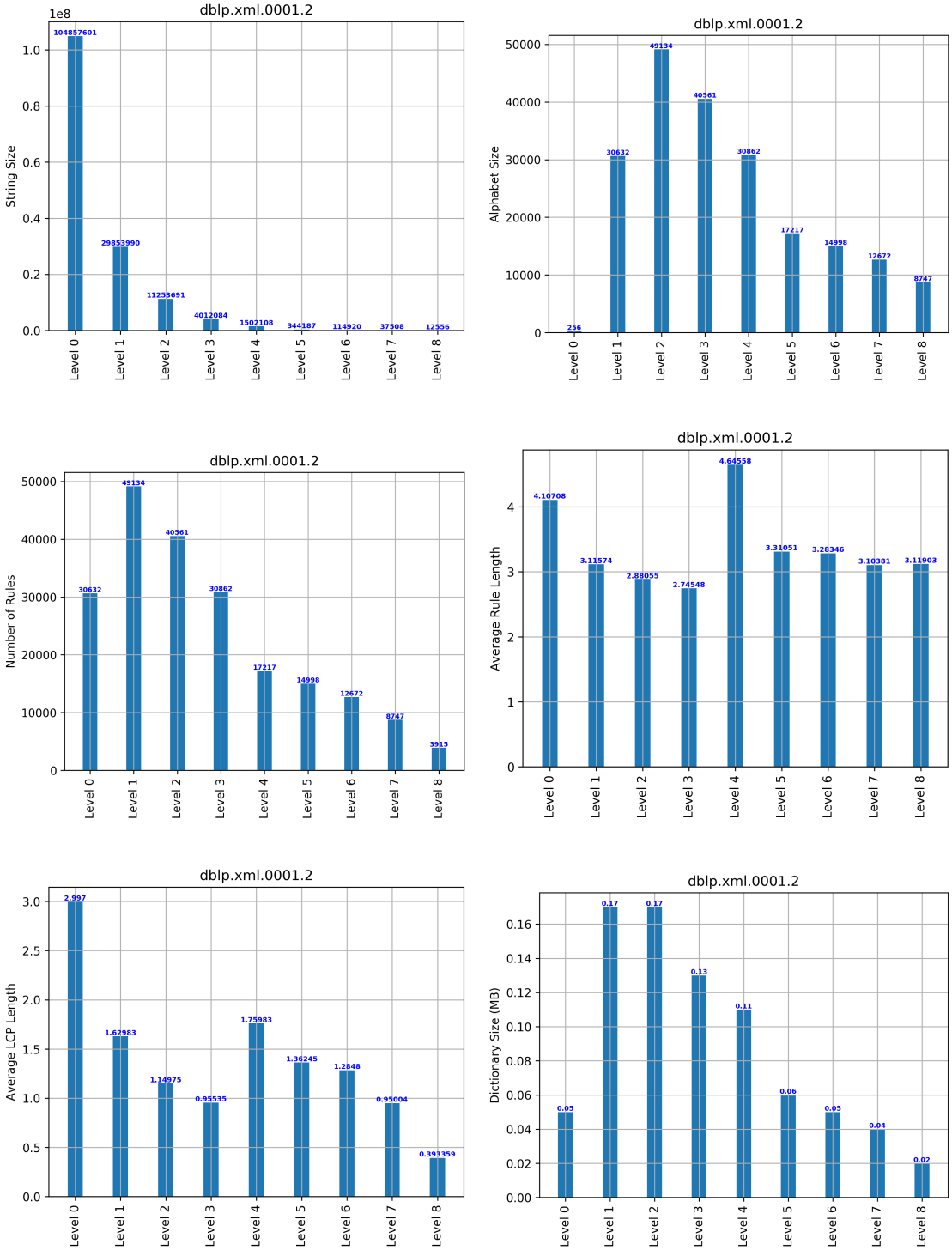


Figure A.6: Empirical attributes for `dblp.xml.0001.2` file.

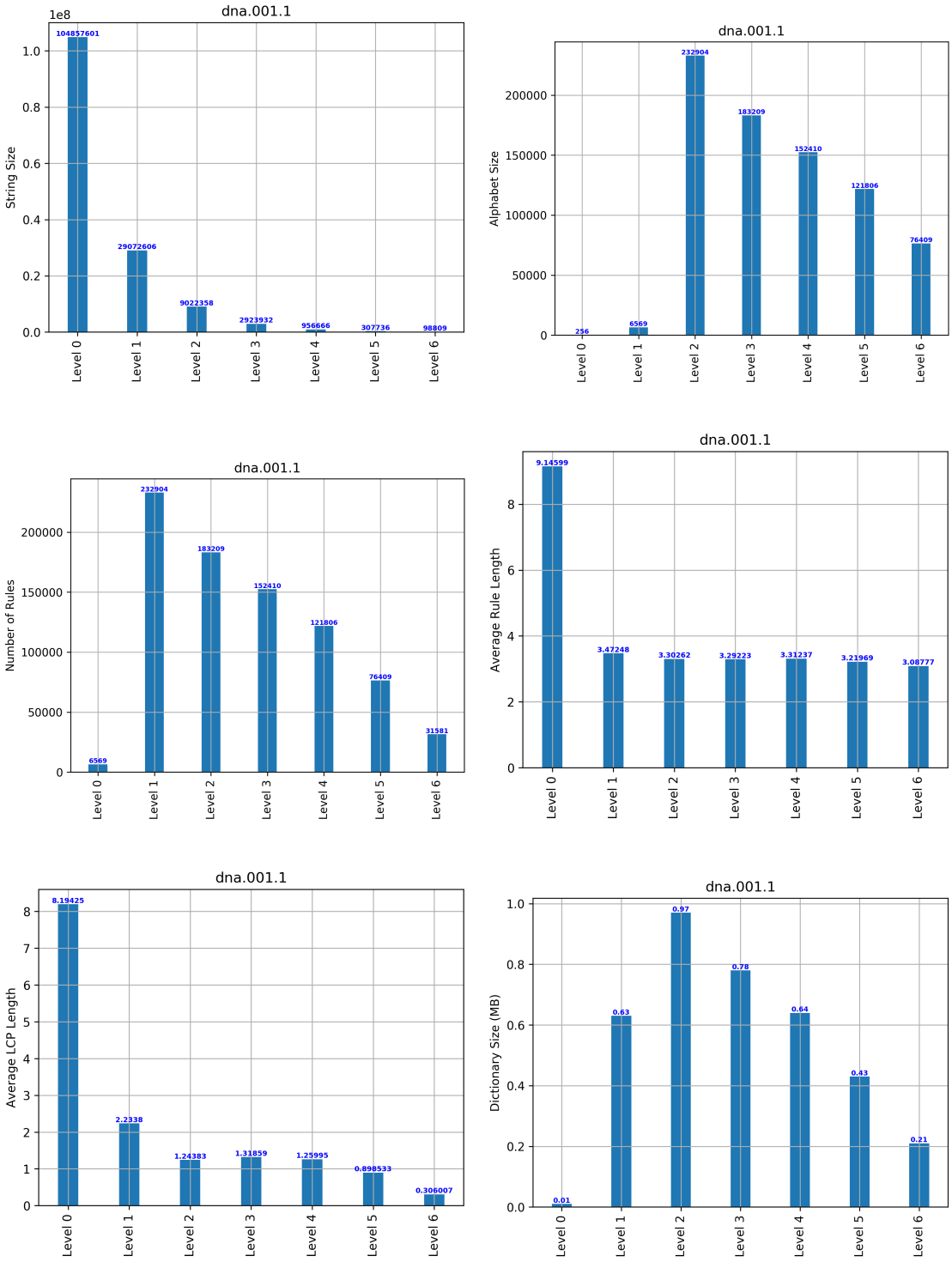


Figure A.7: Empirical attributes for `dna.001.1` file.

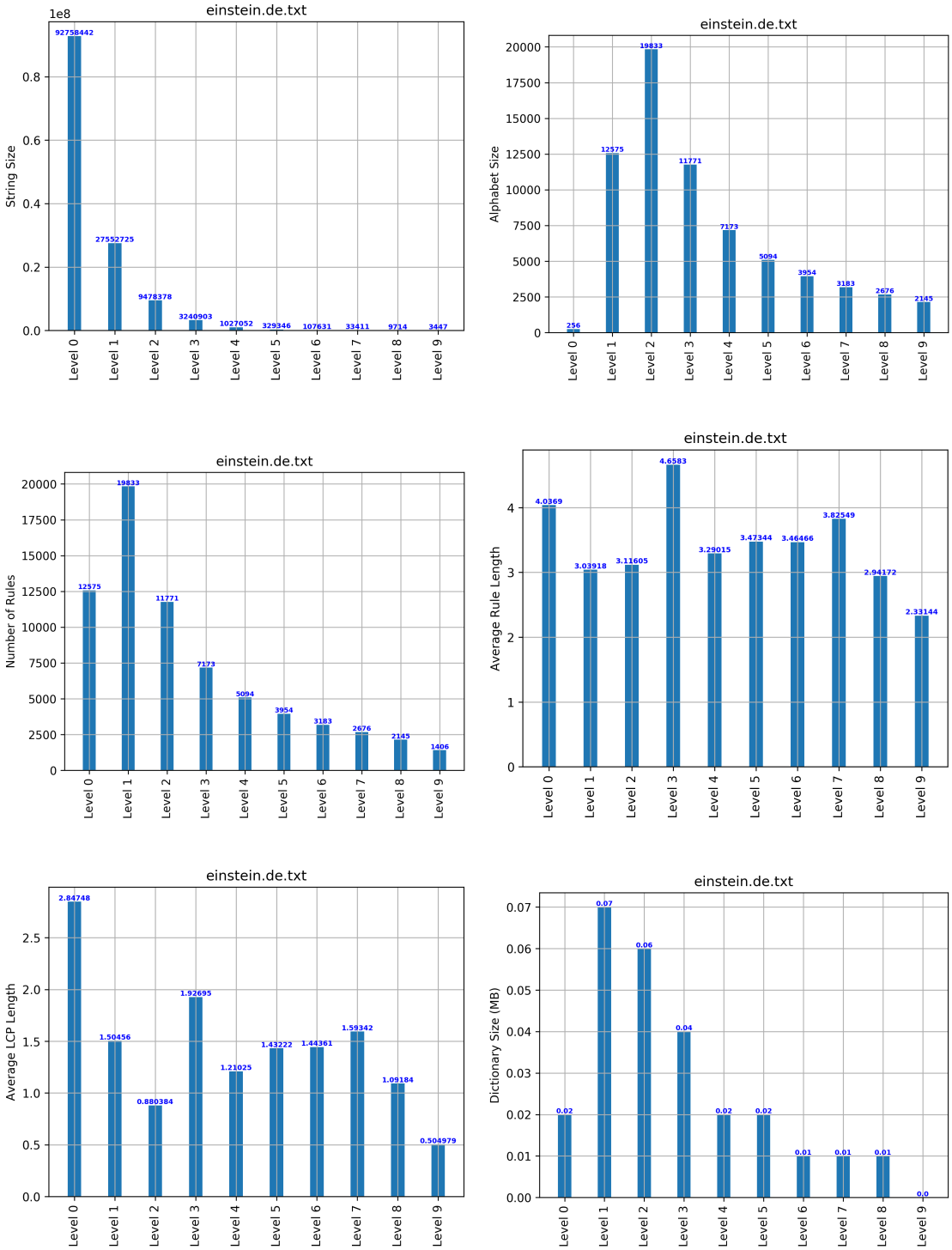


Figure A.8: Empirical attributes for `einstein.de.txt` file.

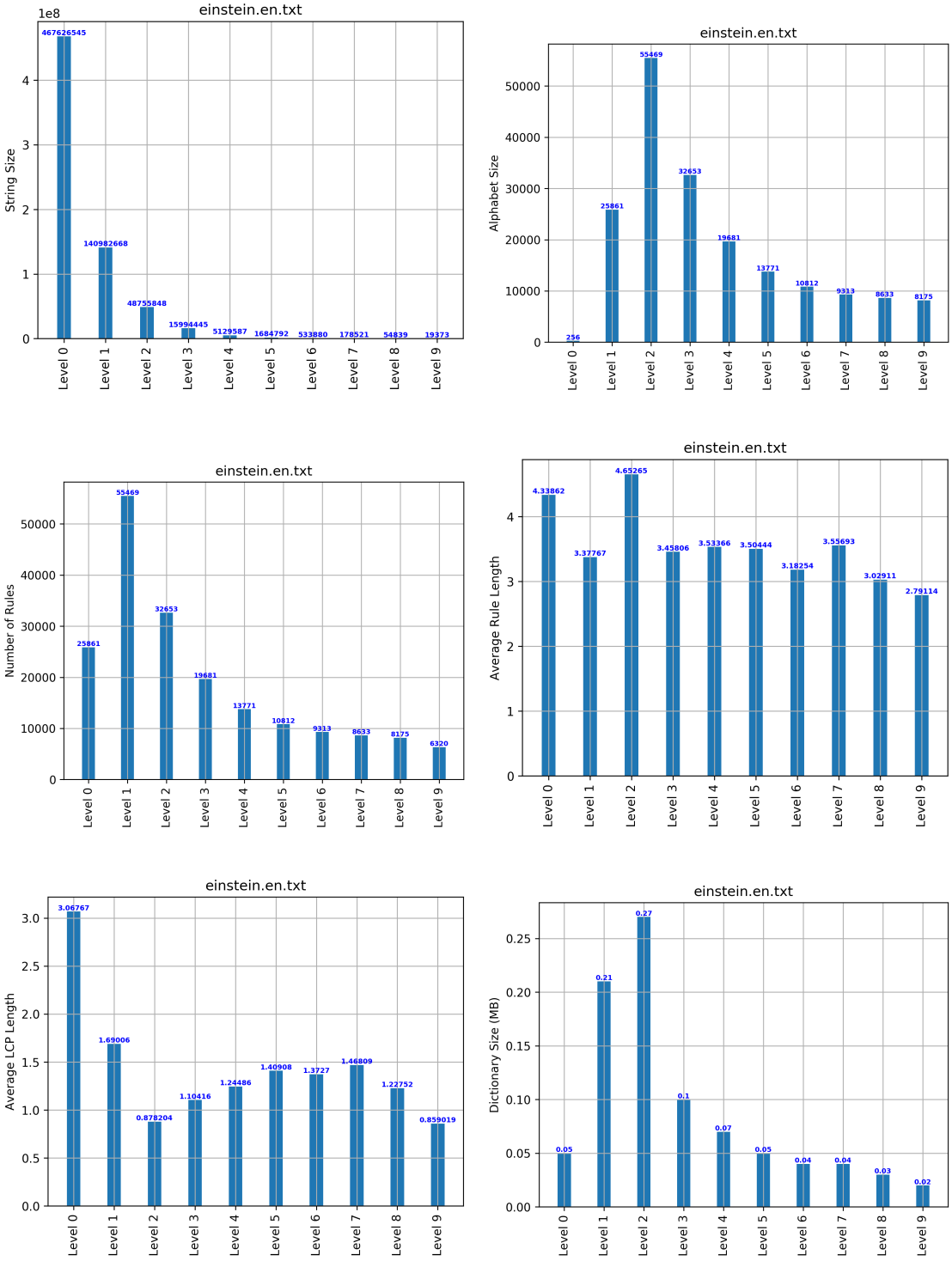


Figure A.9: Empirical attributes for `einstein.en.txt` file.

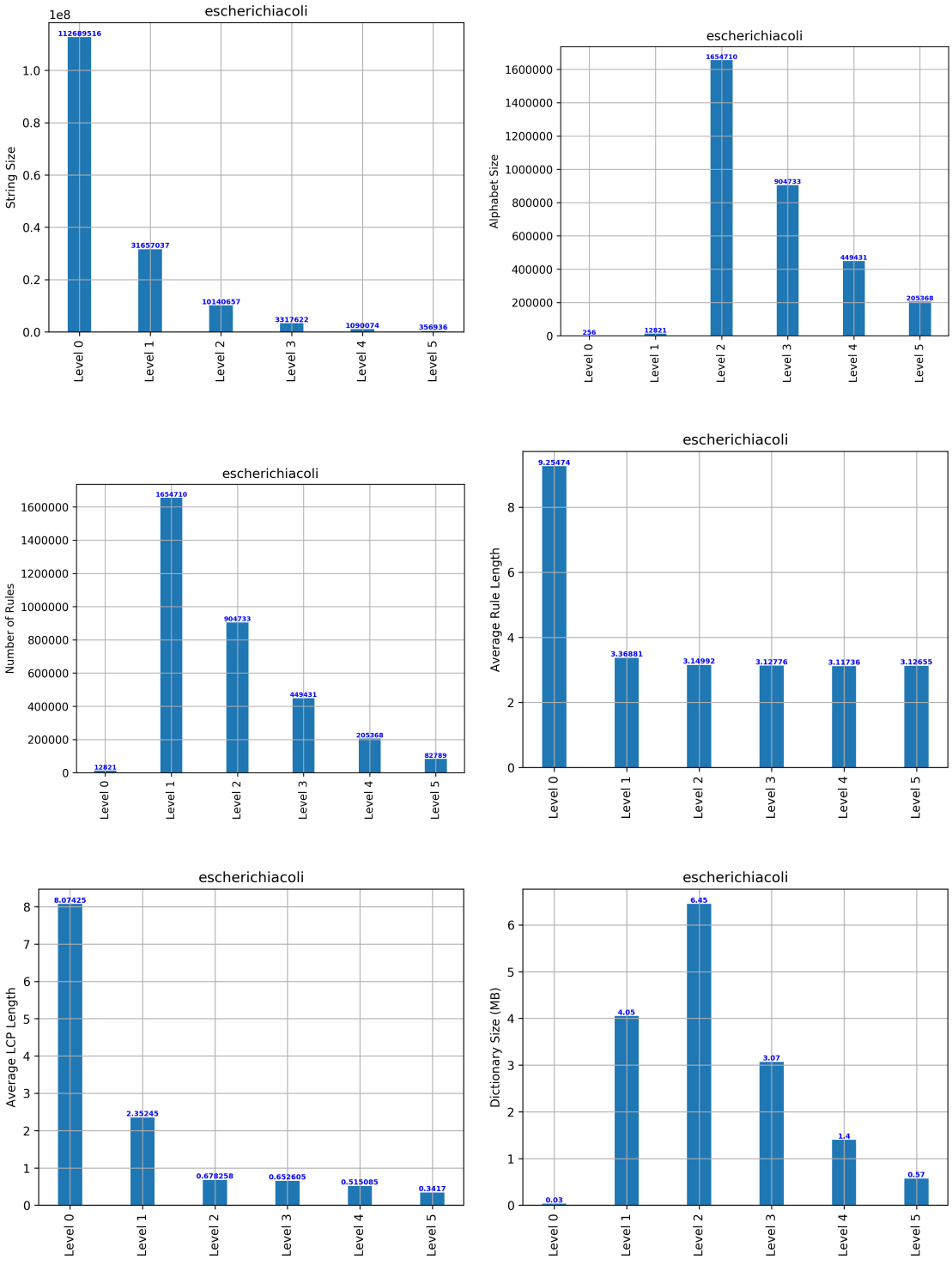


Figure A.10: Empirical attributes for `escherichiacoli` file.

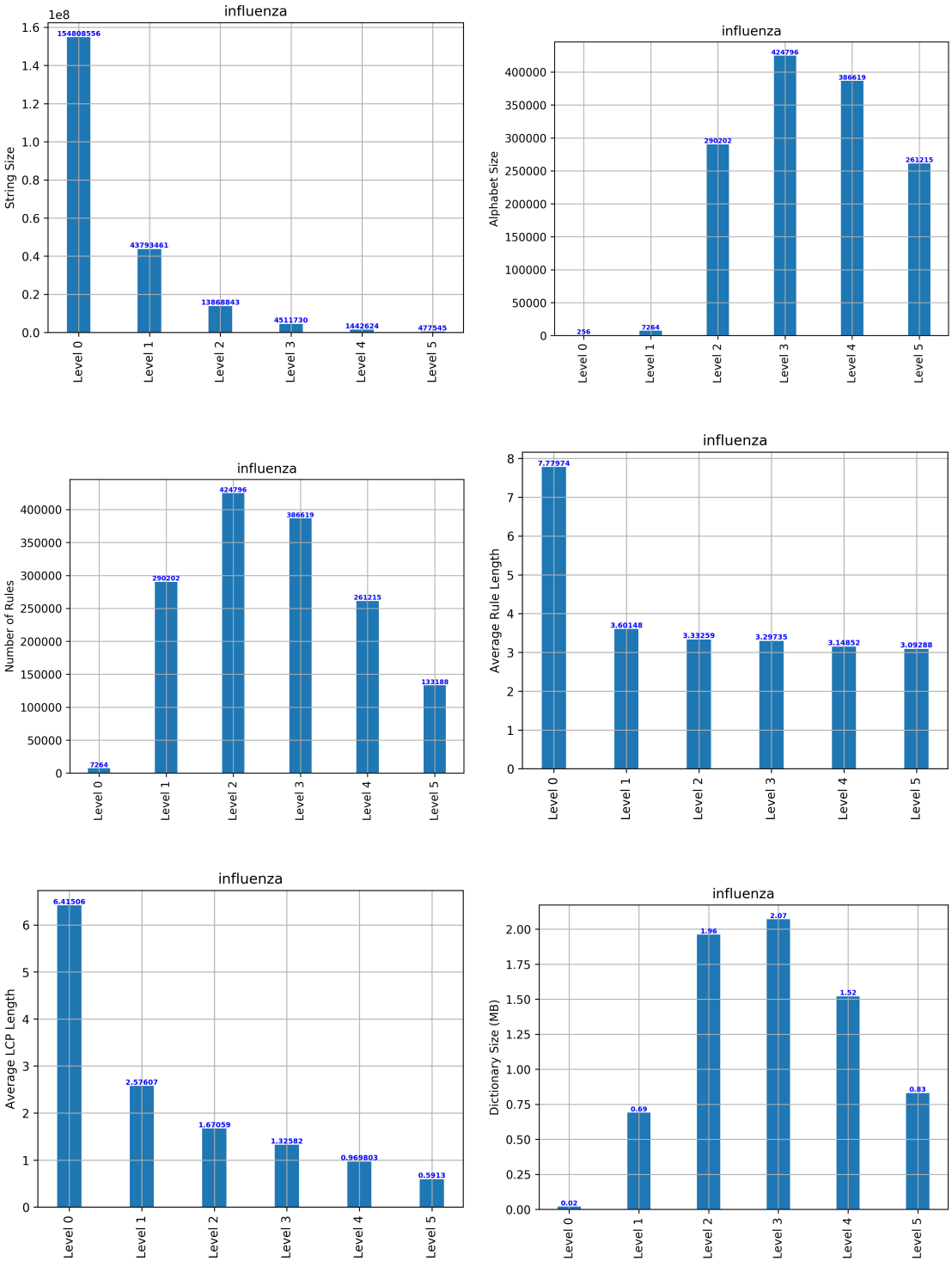


Figure A.11: Empirical attributes for `influenza` file.

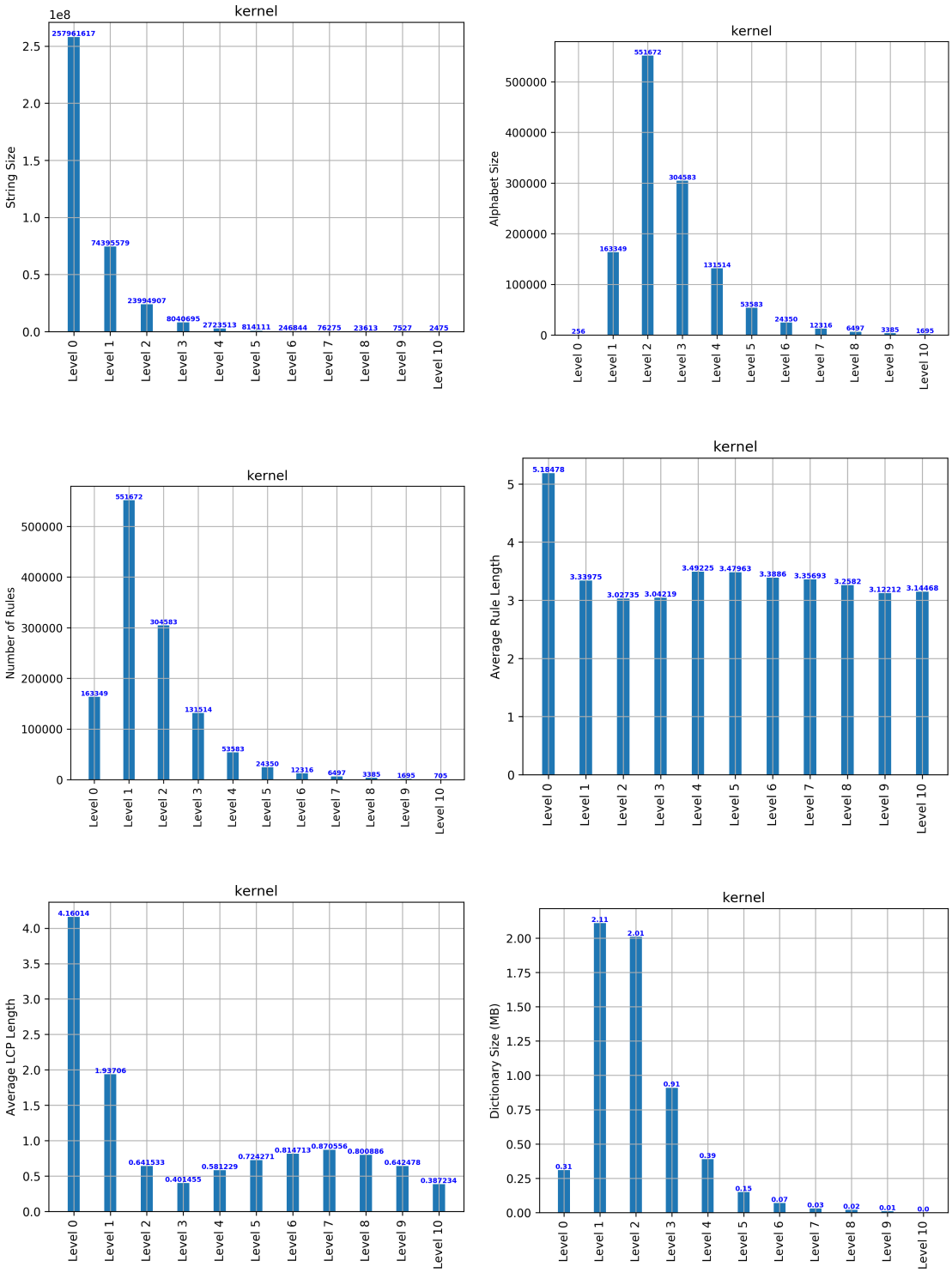


Figure A.12: Empirical attributes for `kernel` file.

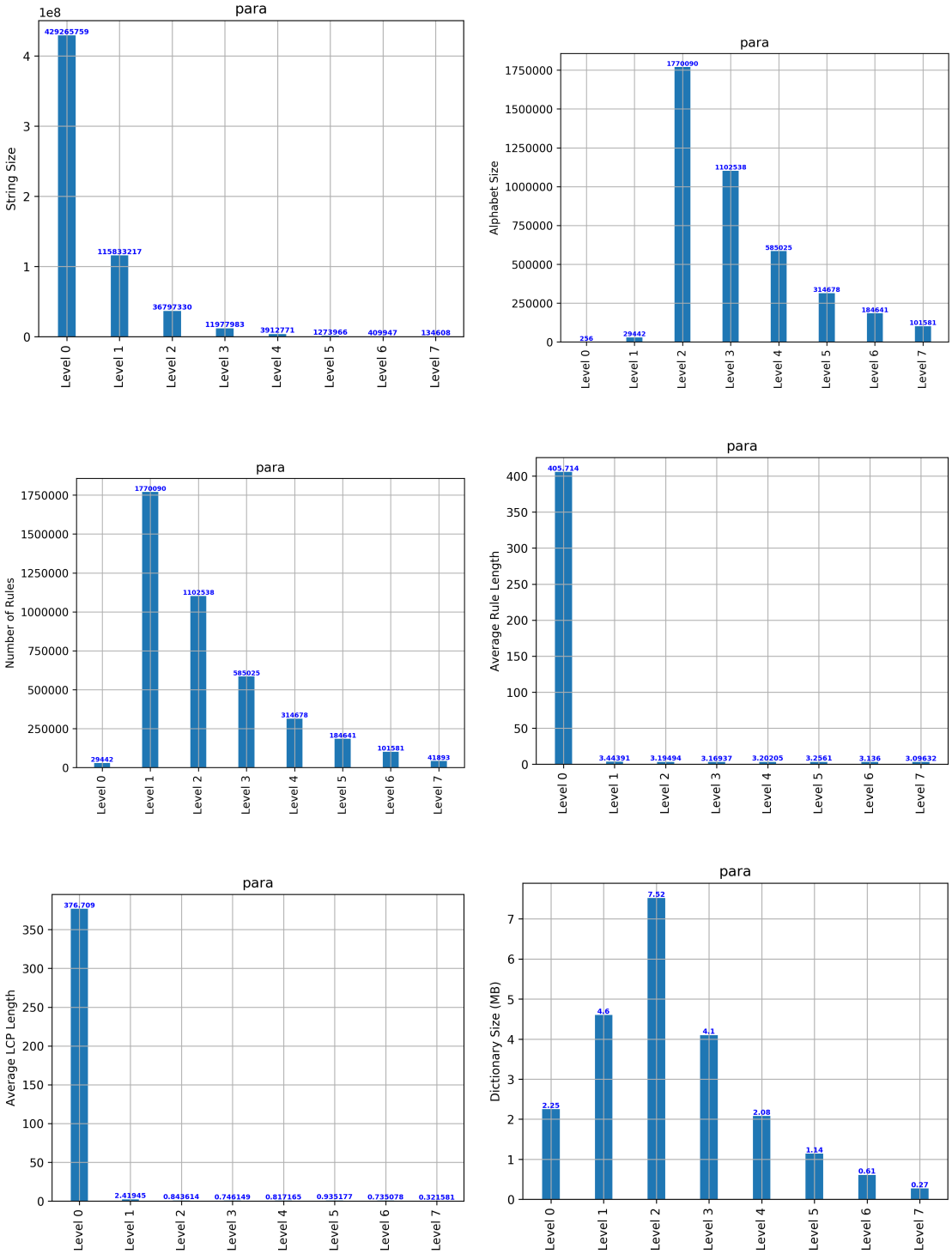


Figure A.13: Empirical attributes for `para` file.

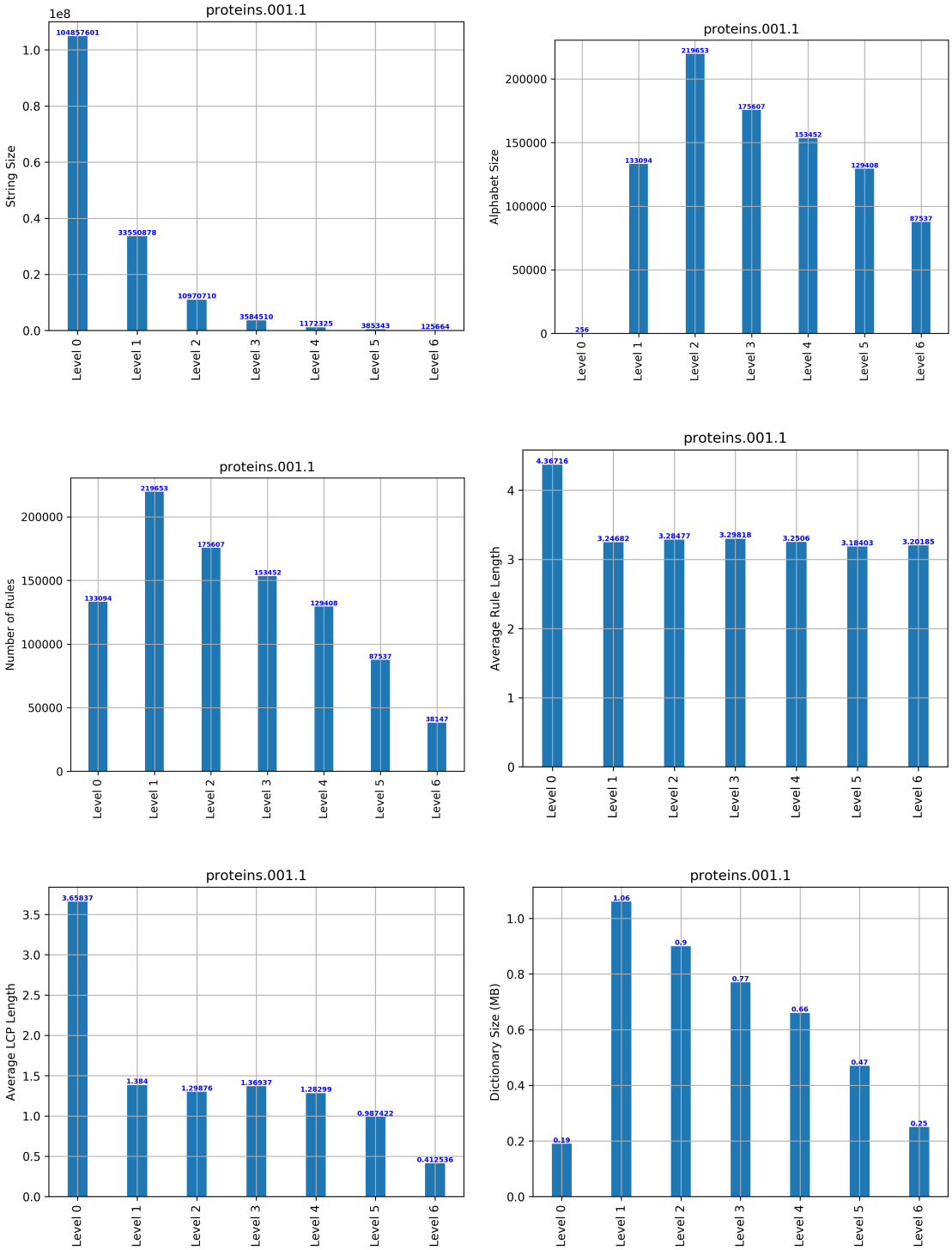


Figure A.14: Empirical attributes for `proteins.001.1` file.

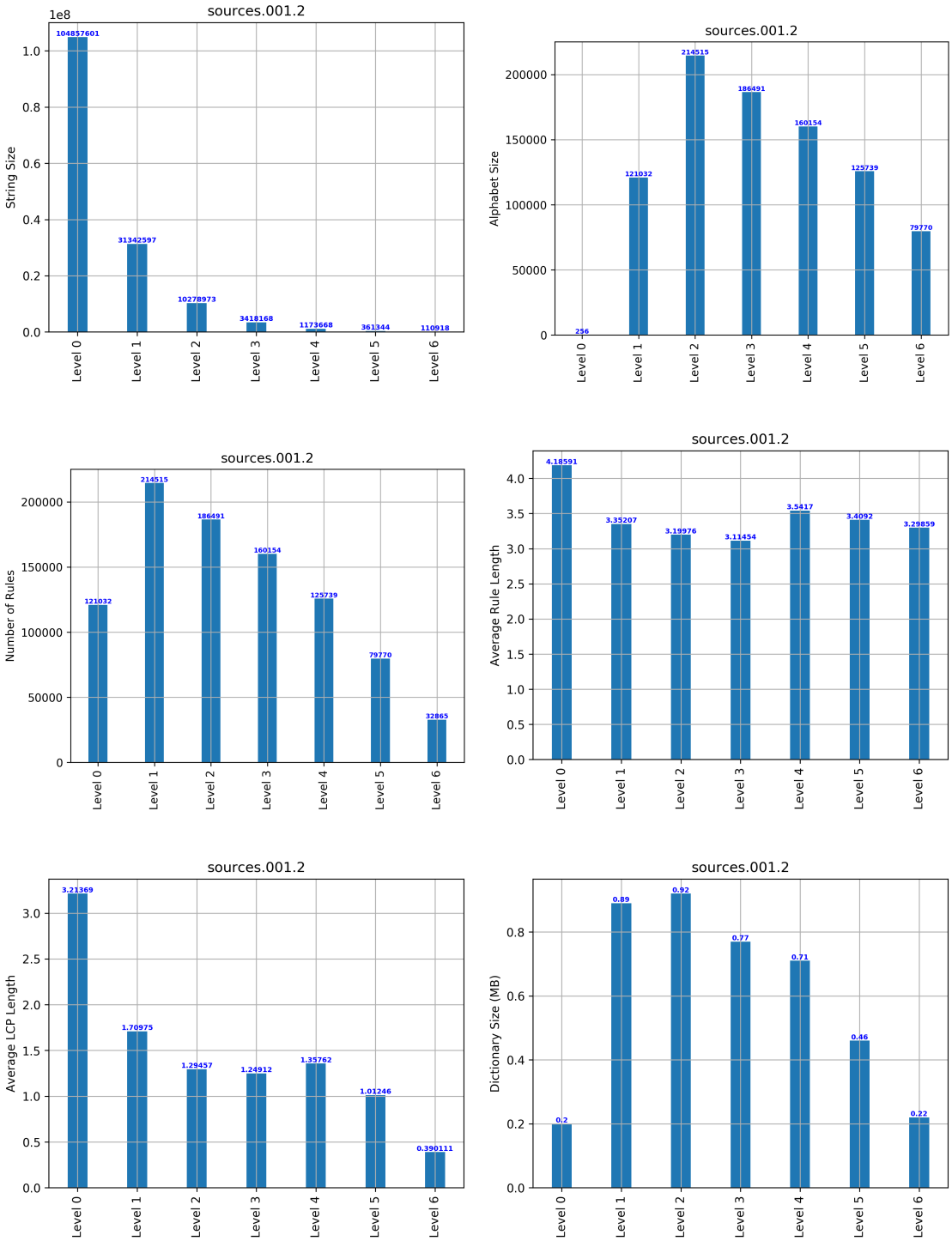


Figure A.15: Empirical attributes for `sources.001.2` file.

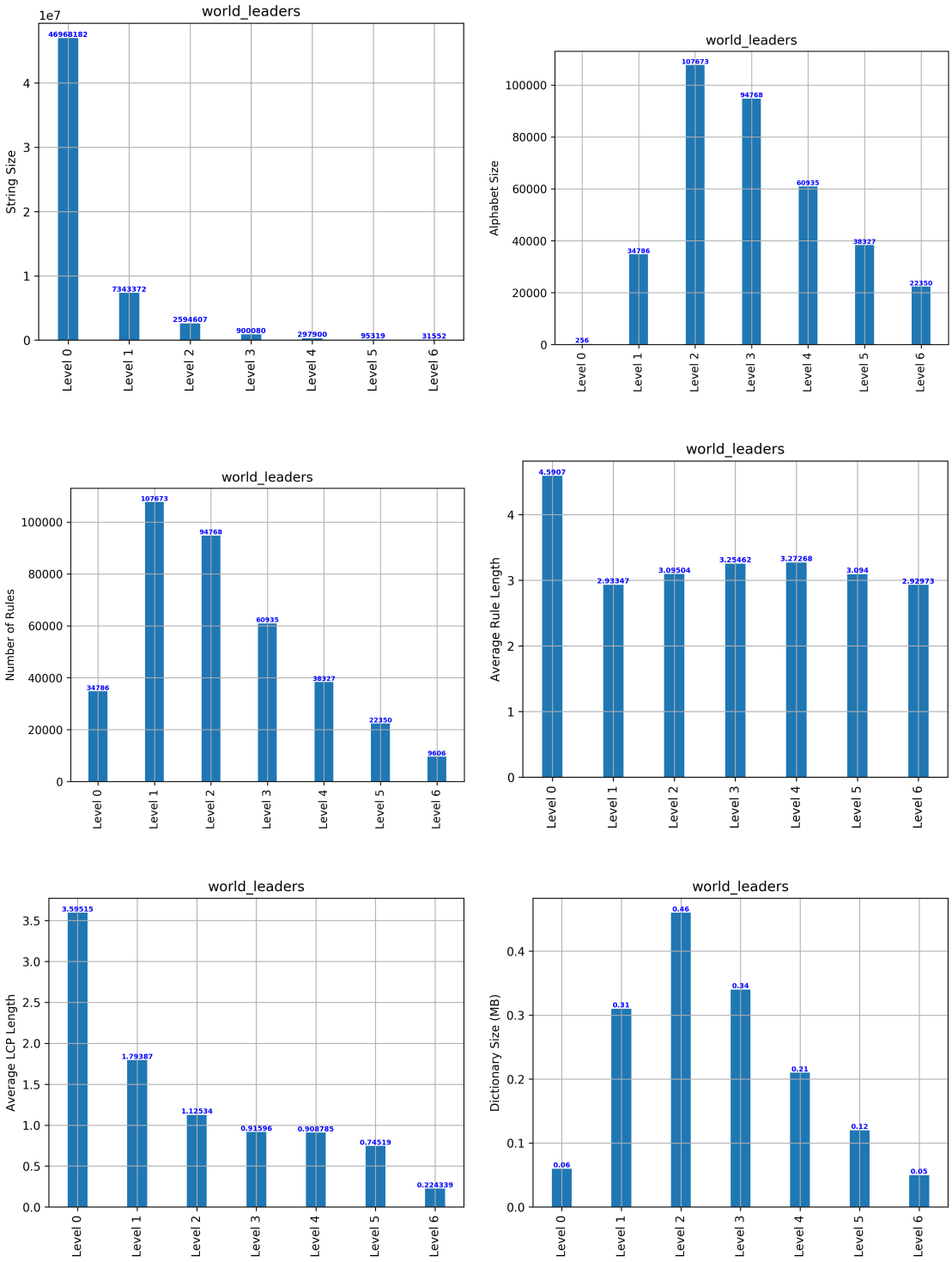


Figure A.16: Empirical attributes for `world_leaders` file.