Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures

Gonzalo Navarro
University of Chile, Chile

Two decades ago, a breakthrough in indexing string collections made it possible to represent them within their compressed space while at the same time offering indexed search functionalities. As this new technology permeated through applications like bioinformatics, the string collections experienced a growth that outperforms Moore's Law and challenges our ability to handle them even in compressed form. It turns out, fortunately, that many of these rapidly growing string collections are highly repetitive, so that their information content is orders of magnitude lower than their plain size. The statistical compression methods used for classical collections, however, are blind to this repetitiveness, and therefore a new set of techniques has been developed in order to properly exploit it. The resulting indexes form a new generation of data structures able to handle the huge repetitive string collections that we are facing. In this survey, formed by two parts, we cover the algorithmic developments that have led to these data structures.

In this first part, we describe the distinct compression paradigms that have been used to exploit repetitiveness, and the algorithmic techniques that provide direct access to the compressed strings. In the quest for an ideal measure of repetitiveness, we uncover a fascinating web of relations between those measures, as well as the limits up to which the data can be recovered, and up to which direct access to the compressed data can be provided. This is the basic aspect of indexability, which is covered in the second part of this survey.

Categories and Subject Descriptors: E.1 [**Data structures**]; E.2 [**Data storage representations**]; E.4 [**Coding and information theory**]: Data compaction and compression; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*; H.2.1 [**Database management**]: Physical design—*Access methods*; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Text indexing, string searching, compressed data structures, repetitive string collections.

Funded by ANID Basal Funds FB0001, Millennium Science Initiative Program - Code ICN17_002, and Fondecyt Grant 1-200038, Chile. Address: Gonzalo Navarro, Center for Biotechnology and Bioengineering (CeBiB) and Millennium Institute for Foundational Research on Data (IMFD), Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile, gnavarro@dcc.uchile.cl.

1. INTRODUCTION

Our increasing capacity for gathering and exploiting all sorts of data around us is shaping modern society into ways that were unthinkable a couple of decades ago. In bioinformatics, we have stepped in 20 years from sequencing the first human genome to completing projects for sequencing 100,000 genomes[1]. Just storing such a collection requires about 70 terabytes, but a common data analysis tool like a suffix tree [Apostolico 1985] would require 5.5 petabytes. In astronomy, telescope networks generating terabytes per hour are around the corner[2]. The web is estimated to have 60 billion pages, with a total size of about 4 petabytes counting just text content[3]. Estimations of the yearly amount of data generated in the world are around 1.5 exabytes[4].

Together with the immense opportunities brought by the data in all sorts of areas, we are faced to the immense challenge of efficiently storing, processing, and analyzing such volumes of data. Approaches such as parallel and distributed computing, secondary memory and streaming algorithms reduce time, but still pay a price proportional to the *data size* in terms of amount of computation, storage requirement, network use, energy consumption, and/or sheer hardware. This is problematic because the growth rate of the data has already surpassed Moore's law in areas like bioinformatics and astronomy [Stephens et al. 2015]. Worse, these methods must access the data in secondary memory, which is much slower than the main memory. Therefore, not only we have to cope with orders-of-magnitude larger data volumes, but we must operate on orders-of-magnitude slower storage devices.

A promising way to curb this growth is to focus on how much *actual information* is carried by those data volumes. It turns out that many of the applications where the data is growing the fastest feature large degrees of *repetitiveness* in the data, that is, most of the content in each element is equal to content of other elements. For example, let us focus on sequence and text data. Genome repositories typically store many genomes of the same species. Two human genomes differ by about 0.1% [Przeworski et al. 2000], and Lempel-Ziv-like compression [Lempel and Ziv 1976] on such repositories report compression ratios (i.e., compressed divided by uncompressed space) around 1% [Fritz et al. 2011]. A versioned document collection like Wikipedia stored 10 terabytes by 2015, and it reported over 20 versions per article, with the versions (i.e., near-repetitions) growing faster than original articles, and 1% Lempel-Ziv compression ratios[5]. A versioned software repository like GitHub stored over 20 terabytes in 2016 and it also reported over 20 versions per project[6]. Degrees of 40%–80% of duplication have been observed in tweets [Tao et al. 2013], emails [Elsayed and Oard 2006], web pages [Henzinger 2006], and general software repositories [Kapser and Godfrey 2005] as well.

These sample numbers show that we can aim at 100-fold reductions in the data

[1]`https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project`
[2]`https://www.nature.com/articles/d41586-018-01838-0`
[3]`https://www.worldwidewebsize.com` and `https://www.keycdn.com/support/the-growth-of-web-page-size`, see the average HTML size.
[4]`http://groups.ischool.berkeley.edu/archive/how-much-info/how-much-info.pdf`
[5]`https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`
[6]`https://blog.sourced.tech/post/tab_vs_spaces` and `http://blog.coderstats.net/github/2013/event-types`

representation size by using appropriate compression methods on highly repetitive data sets. Such a reduction would allow us handling much larger data volumes in main memory, which is considerably faster. Even in cases where the reduced data still does not fit in main memory, we can expect a 100-fold reduction in storage, network, hardware, and/or energy costs.

Just compressing the data, however, is not sufficient to reach this goal, because we still need to decompress it in order to carry out any processing on it. For the case of text documents, a number of "version control" systems like CVS[7], SVN[8], and Git[9], support particular types of repetitive collections, namely *versioned* ones, where documents follow a controlled structure (typically linear or hierarchical) and the systems can track which document is a variant of which. Those systems do a good job in reducing space while supporting extraction of any version of any document, mainly by storing the set of "edits" that distinguish each document from a close version that is stored in plain form.

Still, just extraction of whole documents is not sufficient. In order to process the data efficiently, we need *data structures* built on top it. What is needed is a more ambitious concept, a *compressed data structure* [Navarro 2016]. Such a data structure aims not only at representing the data within space close to its actual information content, but also, within that space, at efficiently supporting direct access, queries, analysis, and manipulation of the data without ever decompressing it. This is in sharp contrast with classical data structures, which add (sometimes very significant) extra space on top of the raw data (e.g., the suffix trees already mentioned typically use 80 times the space of a compacted genome).

Compressed data structures are now 30 years old [Jacobson 1989], have made their way into applications and companies [Navarro 2016], and include mature libraries[10]. Most compressed data structures, however, build on statistical compression [Cover and Thomas 2006], which is blind to repetitiveness [Kreft and Navarro 2013], and therefore fail to get even close to the compression ratios we have given for highly repetitive scenarios. The development of compressed data structures aimed at highly repetitive data is much more recent, and builds on variants of *dictionary compression* [Cover and Thomas 2006; Storer and Szymanski 1982].

The current article is the first part of a survey on *pattern matching on string collections*, one of the most fundamental problems that arise when extracting information from text data, in the case where the string collection is highly repetitive and thus stored in highly compressed form. In this part we focus on the measures of repetitiveness for text collections and their support of direct access. We start with the fascinating issue of how to best measure compressibility via repetitiveness, just like the entropy of Shannon [1948] is the right concept to measure compressibility via frequency skews: Section 3 covers a number of repetitiveness measures, from ad-hoc ones like the size of a Lempel-Ziv parse [Lempel and Ziv 1976] to the most recent and abstract ones based on string attractors and string complexity [Kempa and Prezza 2018; Kociumaka et al. 2020], and the relations between them. We then

[7]`https://savannah.nongnu.org/projects/cvs`
[8]`https://subversion.apache.org`
[9]`https://git-scm.com`
[10]`https://github.com/simongog/sdsl-lite`

consider the problem of how can one access parts of the compressed string collection directly without having to decompress it all: Section 4 explores the problem of giving such direct access on a string that is compressed using some of those measures, which distinguishes a compressed data structure from sheer compression. Some measures enable compression but apparently not direct access. The more ambitious goal of pattern matching indexes with size bounded in terms of some of those measures is developed in Part II of this survey [Navarro 2020].

2. NOTATION AND BASIC CONCEPTS

We assume basic knowledge on algorithms, data structures, and algorithm analysis. In this section we define some fundamental concepts on strings, preceded by a few more general concepts and notation remarks.

Computation model. We use the RAM model of computation, where we assume the programs run on a random-access memory where words of $w = \Theta(\log n)$ bits are accessed and manipulated in constant time, where n is the input size. All the typical arithmetic and logical operations on the machine words are carried out in constant time, including multiplication and bit operations.

Complexities. We will use big-O notation for the time complexities, and in many cases for the space complexities as well. Space complexities are measured in amount of computer words, that is, $O(X)$ space means $O(X \log n)$ bits. By poly x we mean any polynomial in x, that is, $x^{O(1)}$, and polylog x denotes poly $(\log x)$. Logarithms will be to the base 2 by default. Within big-O complexities, $\log x$ must be understood as $\lceil \log(2+x) \rceil$, to avoid border cases.

Indexed pattern matching. The problem consists in, given a string S, build a data structure (called an *index*) so that, later, given a short query string, one efficiently finds the places in S where the query string occurs. We aim at building indexes whose size is bounded by a repetitiveness measure on S.

2.1 Strings

A *string* $S = S[1\,..\,n]$ is a sequence of *symbols* drawn from a set Σ called the *alphabet*. We will assume $\Sigma = \{1, 2, \ldots, \sigma\}$. The *length* of $S[1\,..\,n]$ is n, also denoted $|S|$. We use $S[i]$ to denote the i-th symbol of S and $S[i\,..\,j] = S[i] \cdots S[j]$ to denote a *substring* of S. If $i > j$, then $S[i\,..\,j] = \varepsilon$, the empty string. A *prefix* of S is a substring of the form $S[1\,..\,j]$ and a *suffix* is a substring of the form $S[i\,..\,n] = S[i\,..]$. With SS' we denote the *concatenation* of the strings S and S', that is, the symbols of S' are appended after those of S. Sometimes we identify a single symbol with a string of length 1, so that aS and Sa, with $a \in \Sigma$, denote concatenations as well. In general, string collections will be viewed as a single string that concatenates them all, with a suitable separator symbol among them.

The *lexicographic order* among strings is defined as in a dictionary. Let $a, b \in \Sigma$ and let S and S' be strings. Then $aS \leq bS'$ if $a < b$, or if $a = b$ and $S \leq S'$; and $\varepsilon \leq S$ for every S.

For technical convenience, we will often assume that strings $S[1\,..\,n]$ are terminated with a special symbol $S[n] = \$$, which does not appear elsewhere in S nor in Σ. We assume that \$ is smaller than every symbol in Σ to be consistent with the lexicographic order. The string $S[1\,..\,n]$ read backwards is denoted $S^{rev} = S[n] \cdots S[1]$; note that in this case the terminator does not appear at the end of S^{rev}.

2.2 Suffix Trees and Suffix Arrays

Suffix trees and *suffix arrays* are the most classical pattern matching indexes. The *suffix tree* [Weiner 1973; McCreight 1976; Apostolico 1985] is a trie (or digital tree) containing all the suffixes of S. That is, every suffix of S labels a single root-to-leaf path in the suffix tree, and no node has two distinct children labeled by the same symbol. Further, the unary paths (i.e., paths of nodes with a single child) are compressed into single edges labeled by the concatenation of the contracted edge symbols. Every internal node in the suffix tree corresponds to a substring of S that appears more than once, and every leaf corresponds to a suffix. The leaves of the suffix tree indicate the position of S where their corresponding suffixes start. Since there are n suffixes in S, there are n leaves in the suffix tree, and since there are no nodes with a single child, it has less than n internal nodes. The suffix tree can then be represented within $O(n)$ space, for example by representing every string labeling edges with a couple of pointers to an occurrence of the label in S.

The suffix array [Manber and Myers 1993] of $S[1\,..\,n]$ is the array $A[1\,..\,n]$ of the positions of the suffixes of S in lexicographic order. If the children of the suffix tree nodes are lexicographically ordered by their first symbol, then the suffix array corresponds to the leaves of the suffix tree in left-to-right order.

Example: Figure 1 shows the suffix tree and array of the string $S =$ alabaralalabarda\$.

2.3 Karp-Rabin Fingerprints

Karp and Rabin [1987] proposed a technique to compute a *signature* or *fingerprint* of a string via hashing, in a way that enables (non-indexed) string matching in $O(n)$ average time. The signature $\kappa(Q)$ of a string $Q[1\,..\,q]$ is defined as

$$\kappa(Q) \quad = \quad \left(\sum_{i=1}^{q} Q[i] \cdot b^{i-1} \right) \bmod p,$$

where b is an integer and p a prime number. It is not hard to devise the arithmetic operations to compute the signatures of composed and decomposed strings, that is, compute $\kappa(Q \cdot Q')$ from $\kappa(Q)$ and $\kappa(Q')$, or $\kappa(Q)$ from $\kappa(Q \cdot Q')$ and $\kappa(Q')$, or $\kappa(Q')$ from $\kappa(Q \cdot Q')$ and $\kappa(Q)$ (possibly storing some precomputed exponents together with the signatures).

3. COMPRESSORS AND MEASURES OF REPETITIVENESS

In statistical compression, where the goal is to exploit frequency skew, the so-called *statistical entropy* defined by Shannon [1948] offers a measure of compressibility that is both optimal and reachable. While statistical entropy is defined for infinite sources, it can be adapted to individual strings. The resulting measure for individual strings, called *empirical entropy* [Cover and Thomas 2006], turns out to be a reachable lower bound (save for lower-order terms) to the space a semistatic statistical compressor can achieve on that string.

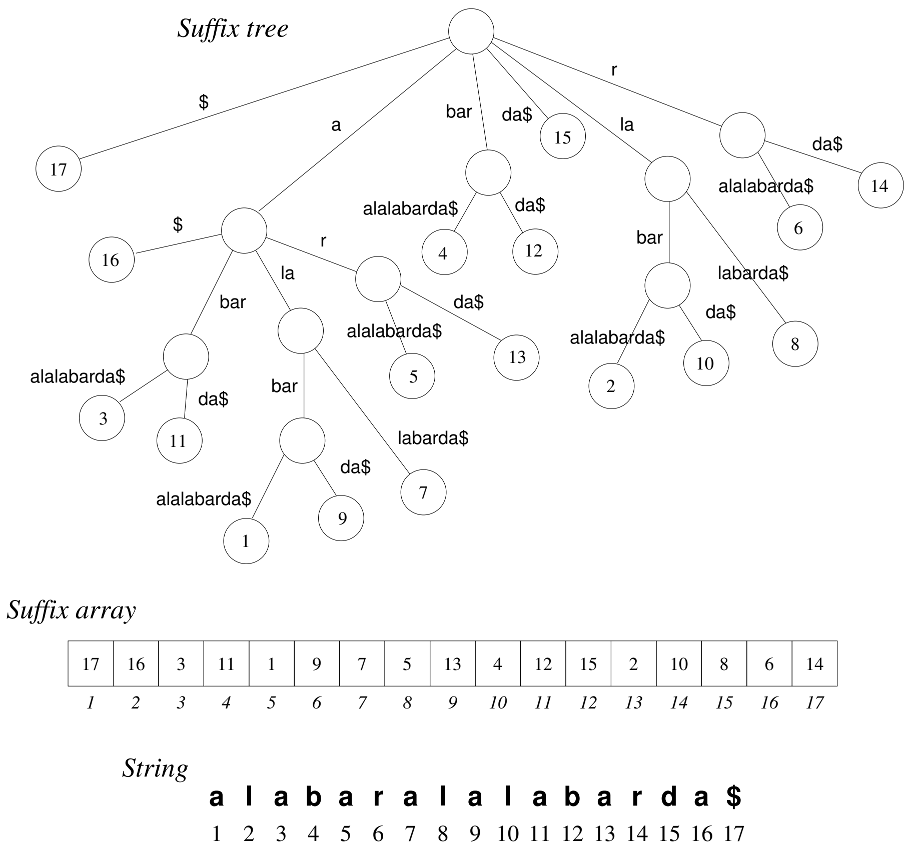

Fig. 1. The suffix tree and suffix array of the string $S =$ alabaralalabarda\$. The suffix tree leaves indicate the positions where the corresponding suffixes start, and those positions, collected left to right, form the suffix array.

Statistical entropy, however, does not adequately capture other sources of compressibility, particularly repetitiveness. In this arena, concepts are much less clear. Beyond the ideal but uncomputable measure of string complexity proposed by Kolmogorov [1965], most popular measures of (compressibility by exploiting) repetitiveness are ad-hoc, defined as the result of particular compressors, and there is not yet a consensus measure that is both reachable and optimal within a reasonable set of compression techniques. Still, many measures work well and have been used as the basis of compressed and indexed sequence representations. In this section we describe the most relevant concepts and measures.

3.1 The Unsuitability of Statistical Entropy

Shannon [1948] introduced a measure of compressibility that exploits the different probabilities of the symbols emitted by a source. In its simplest form, the source is "memoryless" and emits each symbol $a \in \Sigma$ with a fixed probability p_a. The *entropy* is then defined as

$$\mathcal{H}(\{p_a\}) \quad = \quad \sum_{a\in\Sigma} p_a \log \frac{1}{p_a}.$$

When all the probabilities p_a are equal to $1/\sigma$, the entropy is maximal, $\mathcal{H} = \log \sigma$. In general, the entropy decreases as the probabilities are more skewed. This kind of entropy is called *statistical entropy.*

In a more general form, the source may remember the last k symbols emitted, $C[1\mathinner{.\,.}k]$ and the probability $p_{a|C}$ of the next symbol a may depend on them. The entropy is defined in this case as

$$\mathcal{H}(\{p_{a,C}\}) \quad = \quad \sum_{C\in\Sigma^k} p_C \sum_{a\in\Sigma} p_{a|C} \log \frac{1}{p_{a|C}},$$

where p_C is the global probabilty of the source emitting C.

Other more general kinds of sources are considered, including those that have "infinite" memory of all the previous symbols emitted. Shannon [1948] shows that any encoder of a random source of symbols with entropy $\mathcal{H}$ must emit, on average, no less than $\mathcal{H}$ bits per symbol. The measure is also reachable: arithmetic coding [Witten et al. 1987] compresses n symbols from such a source into $n\mathcal{H}+2$ bits.

Shannon's entropy can also be used to measure the entropy of a *finite individual* sequence $S[1\mathinner{.\,.}n]$. The idea is to *assume* that the only source of compressibility of the sequence are the different frequencies of its symbols. If we take the frequencies as independent, the result is the *zeroth order empirical entropy* of S:

$$\mathcal{H}_0(S) \quad = \quad \sum_{a\in\Sigma} \frac{n_a}{n} \log \frac{n}{n_a},$$

where n_a is the number of times a occurs in S and we assume $0 \log 0 = 0$. This is exactly the Shannon's entropy of a memoryless source with probabilities $p_a = n_a/n$, that is, we use the relative frequencies of the symbols in S as an estimate of the probabilities of a hypothetical source that generated S (indeed, the most likely source). The string S can then be encoded in $n\mathcal{H}_0(S)+2$ bits with arithmetic coding based on the symbol frequencies.

If we assume, instead, that the symbols in S are better predicted by knowing their k preceding symbols, then we can use the *kth order empirical entropy* of S to measure its compressibility:

$$\mathcal{H}_k(S) \quad = \quad \sum_{C\in\Sigma^k} \frac{n_C}{n} \cdot \mathcal{H}_0(S_C),$$

where S_C is the sequence of the symbols following substring C in S and $n_C = |S_C|$. Note that, if S has a unique \$-terminator, n_C is also the number of times C occurs in S,[11] and the measure corresponds to the Shannon entropy of a source with memory k. Once again, an arithmetic coder encodes S into $n\mathcal{H}_k(S)+2$ bits.

[11]Except if C corresponds to the last k symbols of S, but this does not affect the measure because this substring contains \$, so it is unique and $n_C = 0$.

At this point, it is valid to wonder what disallows us to take $k = n$, so that $n_C = n_S = 1$ if $C = S$ and $n_C = 0$ for the other strings of length $k = n$, and therefore $\mathcal{H}_n(S) = 0$. We could then encode S with arithmetic coding into 2 bits!

The trick is that the encoding sizes we have given assume that the decoder knows the distribution, that is, the probabilities p_a or $p_{a|C}$ or, in the case of empirical entropies, the frequencies n_a and n_C. This may be reasonable when analyzing the average bit rate to encode a source that emits infinite sequences of symbols, but not when we consider actual compression ratios of finite sequences.

Transmitting the symbol frequencies (called the *model*) to the decoder (or, equivalently, storing it together with the compressed string) in plain form requires $\sigma \log n$ bits for the zeroth order entropy, and $\sigma^{k+1} \log n$ bits for the kth order entropy. With this simple model encoding, we cannot hope to achieve compression for $k \geq \log_\sigma n$, because encoding the model then takes more space than the uncompressed string. In fact, this is not far from the best that can be done: Gagie [2006] shows that, for about that value of k, $n\mathcal{H}_k(S)$ sometimes falls below Kolmogorov's complexity, and thus there is no hope of encoding S within that size. In his words, "kth-order empirical entropy stops being a reasonable complexity metric for almost all strings".

With the restriction $k \leq \log_\sigma n$, consider now that we concatenate two identical strings, $S \cdot S$. All the relative symbol frequencies in $S \cdot S$ are identical to those in S, except for the $k-1$ substrings C that cover the concatenation point; therefore we can expect that $\mathcal{H}_k(S \cdot S) \approx \mathcal{H}_k(S)$. Indeed, it can be shown that $\mathcal{H}_k(S \cdot S) \geq \mathcal{H}_k(S)$ [Kreft and Navarro 2013, Lem. 2.6]. That is, the empirical entropy is insensitive to the repetitiveness, and any compressor reaching the empirical entropy will compress $S \cdot S$ to about twice the space it uses to compress S. Instead, being aware of repetitivenss allows us to compress S in any form and then somehow state that a second copy of S follows.

This explains why the compressed indexes based on statistical entropy [Navarro and Mäkinen 2007] are not suitable for indexing highly repetitive string collections. In those, the space reduction that can be obtained by exploiting the repetitiveness is much more significant than what can be obtained by exploiting skewed frequencies.

Dictionary methods [Storer and Szymanski 1982], based on representing S as the concatenation of strings from a set (a "dictionary"), generally obtained from S itself, are more adequate to exploit repetitiveness: a small dictionary of distinct substrings of S should suffice if S is highly repetitive. Though many dictionary methods can be shown to converge to Shannon's entropy, our focus here is their ability to capture repetitiveness. In the sequel we cover various such methods, not only as compression methods but also as ways to measure repetitiveness.

We refer the reader to Cover and Thomas [2006] for a deeper discussion of the concepts of Shannon entropy and its relation to dictionary methods.

3.2 Lempel-Ziv Compression: Measures z and z_{no}

Lempel and Ziv [1976] proposed a technique to measure the "complexity" of individual strings based on their repetitiveness (in our case, complexity can be interpreted as incompressibility). The compressor LZ77 [Ziv and Lempel 1977] and many other variants [Bell et al. 1990] that derived from this measure have become very popular; they are behind compression software like `zip`, `p7zip`, `gzip`, `arj`, etc.

3.2.1 *The compressor.* The original Lempel-Ziv method *parses* (i.e., partitions) $S[1\,..\,n]$ into *phrases* (i.e., substrings) as follows, starting from $i \leftarrow 1$:

(1) Find the shortest prefix $S[i\,..\,j]$ of $S[i\,..\,n]$ that does not occur in S starting before position i.
(2) The next phrase is then $S[i\,..\,j]$.
(3) Set $i \leftarrow j+1$. If $i \leq n$, continue forming phrases.

This greedy parsing method can be proved to be optimal (i.e., producing the least number of phrases) among all *left-to-right* parses (i.e., those where phrases must have an occurrence starting to their left) [Lempel and Ziv 1976, Thm. 1].

A compressor can be obtained by encoding each phrase as a triplet: If $S[i\,..\,j]$ is the next phrase, then $S[i\,..\,j-1]$ occurs somewhere to the left of i in S. Let $S[i'\,..\,j']$ be one such occurrence (called the *source* of the phrase), that is, $i' < i$. The next triplet is then $\langle i', j-i, S[j]\rangle$. When $j-i=0$, any empty substring can be the source, and it is customary to assume $i'=0$, so that the triplet is $\langle 0, 0, S[j]\rangle$.

Example: The string $S = \mathsf{alabaralalabarda\$}$ *is parsed as* $\mathsf{a|l|ab|ar|alal|abard|a\$}$, *where we use the vertical bar to separate the phrases. A possible triplet encoding is* $\langle 0,0,\mathsf{a}\rangle\langle 0,0,\mathsf{l}\rangle\langle 1,1,\mathsf{b}\rangle\langle 1,1,\mathsf{r}\rangle\langle 1,3,\mathsf{l}\rangle\langle 3,4,\mathsf{d}\rangle\langle 11,1,\$\rangle$.

From the triplets, we easily recover S by starting with an empty string S and, for each new triplet $\langle p, \ell, c\rangle$, appending $S[p\,..\,p+\ell-1]$ and then c.[12]

This extremely fast decompression is one of the reasons of the popularity of Lempel-Ziv compression. Another reason is that, though not as easily as decompression, it is also possible to carry out the compression (i.e., the parsing) in $O(n)$ time [Rodeh et al. 1981; Storer and Szymanski 1982]. Recently, there has been a lot of research on doing the parsing within little extra space, see for example Kärkkäinen et al. [2016], Fischer et al. [2018], and references therein.

3.2.2 *The measure.* For this survey we will use a slightly different variant of the Lempel-Ziv parsing, which is less popular for compression but more coherent with other measures of repetitiveness, and simplifies indexing. The parsing into phrases is redefined as follows, also starting with $i \leftarrow 1$.

(1) Find the longest prefix $S[i\,..\,j]$ of $S[i\,..\,n]$ that occurs in S starting before position i.
(2) If $j \geq i$, that is, $S[i\,..\,j]$ is nonempty, then the next phrase is $S[i\,..\,j]$, and we set $i \leftarrow j+1$.
(3) Otherwise, the next phrase is the explicit symbol $S[i]$, which has not appeared before, and we set $i \leftarrow i+1$.
(4) If $i \leq n$, continue forming phrases.

We define the Lempel-Ziv measure of $S[1\,..\,n]$ as the number $z = z(S)$ of phrases into which S is parsed by this procedure.

Example: Figure 2 shows how the string $S = \mathsf{alabaralalabarda\$}$ *is parsed into* $z(S) = 11$ *phrases,* $\boxed{\mathsf{a}}|\boxed{\mathsf{l}}|\mathsf{a}|\boxed{\mathsf{b}}|\mathsf{a}|\boxed{\mathsf{r}}|\mathsf{ala}|\mathsf{labar}|\boxed{\mathsf{d}}|\mathsf{a}|\boxed{\$}$, *with the explicit symbols boxed.*

[12]Since the source may overlap the formed phrase, the copy of $S[p\,..\,p+\ell-1]$ to the end of S must be done left to right: consider recovering $S = \mathsf{a}^{n-1}\$$ from the encoding $\langle 0,0,\mathsf{a}\rangle\langle 1, n-2, \$\rangle$.

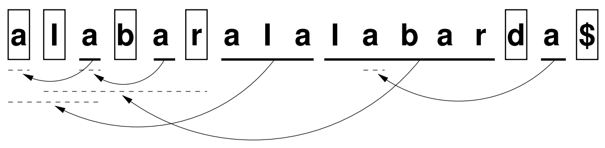

Fig. 2. Lempel-Ziv parse of $S = \mathsf{alabaralalabarda\$}$. Each phrase is either an underlined string, which appears before, or a boxed symbol. The arrows go from each underlined string to one of its occurrences to the left (which is underlined with a dashed line).

The two parsing variants are closely related. If the original variant forms the phrase $S[i\,..\,j]$ with $j > i$, then $S[i\,..\,j-1]$ is the longest prefix of $S[i\,..\,n]$ that appears starting to the left of i, so this new variant will form the phrase $S[i\,..\,j-1]$ (if $i < j$) and its next phrase will be either just $S[j]$ or a longer prefix of $S[j\,..\,n]$. It is not hard to see that the compression algorithms for both variants are the same, and that the greedy parsing is also optimal for this variant. It follows that $z' \le z \le 2z'$, where z' is the number of phrases created with the original method. Thus, z and z' are equivalent in asymptotic terms. In particular, the triplet encoding we described shows that one can encode S within $O(z \log n)$ bits (or $O(z)$ words), which makes z a reachable compressibility measure.

3.2.3 *A weaker variant.* Storer and Szymanski [1982] use a slightly weaker Lempel-Ziv parse, where the source $S[i'\,..\,j']$ of $S[i\,..\,j]$ must be completely contained in $S[1\,..\,i-1]$. That is, it must hold that $j' < i$, not just $i' < i$. The same greedy parsing described, using this stricter condition, also yields the least number of phrases [Storer and Szymanski 1982, Thm. 10 with $p = 1$]. The phrase encoding and decompression proceed in exactly the same way, and linear-time parsing is also possible [Crochemore et al. 2012]. The number of phrases obtained in this case will be called z_{no}, with *no* standing for "no overlap" between phrases and their sources.

This parsing simplifies, for example, direct pattern matching on the compressed string [Gasieniec et al. 1996; Farach and Thorup 1998] or creating context-free grammars from the Lempel-Ziv parse [Rytter 2003].[13] It comes with a price, however. Not only $z_{no}(S) \ge z(S)$ holds for every string S because the greedy parsings are optimal, but also z_{no} can be $\Theta(\log n)$ times larger than z, for example on the string $S = \mathsf{a}^{n-1}\$$, where $z = 3$ (with parsing $\boxed{\mathsf{a}}|\mathsf{a}^{n-2}|\boxed{\$}$) and $z_{no} = \Theta(\log n)$ (with parsing $\boxed{\mathsf{a}}|\mathsf{a}|\mathsf{a}^2|\mathsf{a}^4|\mathsf{a}^8|\cdots$).

3.2.4 *Evaluation.* Apart from fast compression and decompression, a reason for the popularity of Lempel-Ziv compression is that all of its variants converge to the statistical entropy [Lempel and Ziv 1976], even on individual strings [Kosaraju and Manzini 2000], though statistical methods converge faster (i.e., their sublinear extra space over the empirical entropy of $S[1\,..\,n]$ is a slower-growing function of n). In particular, it holds that $z_{no} = O(n/\log_\sigma n)$, so the space $O(z_{no})$ is at worst $\Theta((n/\log_\sigma n)\log n) = \Theta(n \log \sigma)$ bits, proportional to the plain size of S.

More important for us is that Lempel-Ziv captures repetitiveness. In our preced-

[13]A Lempel-Ziv based index also claims to need this restricted parsing [Kreft and Navarro 2013], but in fact they can handle the original parsing with no changes.

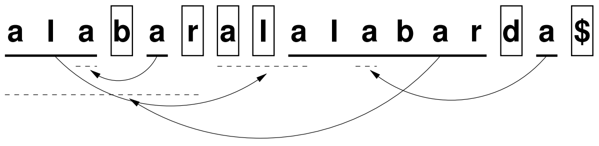

Fig. 3. A bidirectional macro scheme for $S =$ alabaralalabarda\$, with the conventions of Figure 2.

ing example of the string $S \cdot S$, we have $z(S \cdot S) \le z(S)+1$ and $z_{no}(S \cdot S) \le z_{no}(S)+1$ (i.e., we need at most one extra phrase to capture the second copy of S).

Despite the success of Lempel-Ziv compression and its frequent use as a gold standard to quantify repetitiveness, the measure z (and also z_{no}) has some pitfalls:

—It is asymmetric, that is, $z(S)$ may differ from $z(S^{rev})$. For example, removing the terminator \$ to avoid complications, alabaralalabarda is parsed into $z = 10$ phrases, whereas its reverse adrabalalarabala requires only $z = 9$.

—It is monotonic when removing suffixes, but not prefixes, that is, $z(S')$ can be larger than $z(S \cdot S')$. For example, aaabaaabaaa is parsed into $z = 4$ phrases, $\boxed{\mathsf{a}}|\mathsf{aa}|\boxed{\mathsf{b}}|\mathsf{aaabaaa}$, but aabaaabaaa needs $z = 5$, $\boxed{\mathsf{a}}|\mathsf{a}|\boxed{\mathsf{b}}|\mathsf{aa}|\mathsf{abaaa}$.

—Although it is the optimal size of a left-to-right parse, z is arguably not optimal within a broader class of plausible compressed representations. One can represent S using fewer phrases by allowing their sources to occur also to their right in S, as we show with the next measure.

3.3 Bidirectional Macro Schemes: Measure b

Storer and Szymanski [1982] proposed an extension of Lempel-Ziv parsing that allows sources to be to the left or to the right of their corresponding phrases, as long as every symbol can eventually be decoded by following the dependencies between phrases and sources. They called such a parse a "bidirectional macro scheme". Analogously to the Lempel-Ziv parsing variant we are using, a phrase is either a substring that appears elsewhere, or an explicit symbol.

The dependencies between sources and phrases can be expressed through a function f, such that $f(i) = j$ if position $S[i]$ is to be obtained from $S[j]$; we set $f(i) = 0$ if $S[i]$ is an explicit symbol. Otherwise, if $S[i\,..\,j]$ is a copied phrase, then $f(i+t) = f(i)+t$ for all $0 \le t \le j-i$, that is, $S[f(i)\,..\,f(j)]$ is the source of $S[i\,..\,j]$. The bidirectional macro scheme is valid if, for each $1 \le i \le n$, there is a $k > 0$ such that $f^k(i) = 0$, that is, every position is eventually decoded by repeatedly looking for the sources.

We call $b = b(S)$ the minimum number of phrases of a bidirectional macro scheme for $S[1\,..\,n]$. It obviously holds that $b(S) \le z(S)$ for every string S because Lempel-Ziv is just one possible bidirectional macro scheme.

Example: Figure 3 shows a bidirectional macro scheme for $S =$ alabaralalabarda\$ formed by $b = 10$ phrases, $S = \mathsf{ala}|\boxed{\mathsf{b}}|\mathsf{a}|\boxed{\mathsf{r}}|\boxed{\mathsf{a}}|\boxed{\mathsf{l}}|\mathsf{alabar}|\boxed{\mathsf{d}}|\mathsf{a}|\boxed{\$}$ (we had $z = 11$ for the same string, see Figure 2). It has function $f[1\,..\,n] = \langle 7, 8, 9, 0, 3, 0, 0, 0, 1, 2, 3, 4, 5, 6, 0, 11, 0\rangle$. One can see that every symbol can eventually be obtained from the explicit ones. For example, following the arrows in the figure (or, similarly, iterating the function f), we can obtain $S[13] = S[5] = S[3] = S[9] = S[1] = S[7] =$ a.

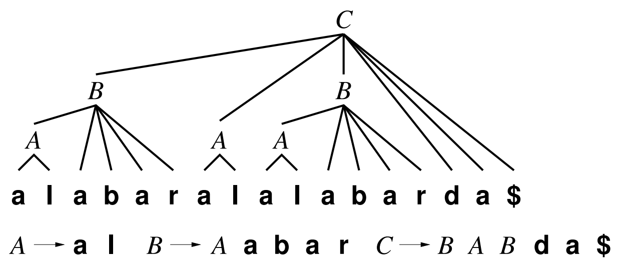

Fig. 4. A context-free grammar generating the string $S =$ alabaralalabarda\$. The rules of the grammar are on the bottom. On the top we show the parse tree.

Just as for Lempel-Ziv, we can compress S to $O(b)$ space by encoding the source of each of the b phrases. It is not hard to recover $S[1\,..\,n]$ in $O(n)$ time from the encoded phrases, because when we traverse t positions until finding an explicit symbol in time $O(t)$, we discover the contents of all those t positions. Instead, finding the smallest bidirectional macro scheme is NP-hard [Gallant 1982]. This is probably the reason that made this technique less popular, although some attempts exist to approach it heuristically [Nishimoto and Tabei 2019; Russo et al. 2020].

As said, it always holds that $b \le z$. Gagie et al. [2018] (corrected in Navarro et al. [2021]) showed that $z = O(b \log(n/b))$ for every string family, and that this bound is tight: in the Fibonacci words, where $F_1 =$ b, $F_2 =$ a, and $F_k = F_{k-1} \cdot F_{k-2}$ for $k > 2$, it holds that $b = O(1)$ and $z = \Theta(\log n)$.

Measure b is the smallest of those we study that is *reachable*, that is, we can compress $S[1\,..\,n]$ to $O(b)$ words. It is also symmetric, unlike z: $b(S) = b(S^{rev})$. Still, b is not monotonic: there are strings S and S' where $b(S) > b(S \cdot S')$.[14]

3.4 Grammar Compression: Measures g and g_{rl}

Kieffer and Yang [2000] introduced a compression technique based on context-free grammars (the idea can be traced back to Rubin [1976]). Given $S[1\,..\,n]$, we find a grammar that generates only the string S, and use it as a compressed representation. The size of a grammar is the sum of the lengths of the right-hand sides of the rules (we avoid empty right-hand sides).

Example: Figure 4 shows a context-free grammar that generates only the string $S =$ alabaralalabarda\$. The grammar has three rules, $A \to$ al, $B \to A$abar, and the initial rule $C \to BAB$da\$. The sum of the lengths of the right-hand sides of the rules is 13*, the grammar size.*

Note that, in a grammar that generates only one string, there is exactly one rule $A \to X_1 \cdots X_k$ per nonterminal A, where each X_i is a terminal or a nonterminal (if there is more than one rule per nonterminal, these must be redundant and we can leave only one).

Figure 4 also displays the *parse tree* of the grammar: an ordinal labeled tree where the root is labeled with the initial symbol, the leaves are labeled with the

[14]Paolo Ferragina and Francesco Tosoni (personal communication) show that $b($aabaaaabaa$) = 5$ but $b($aabaaaabaaa$) = 4$.

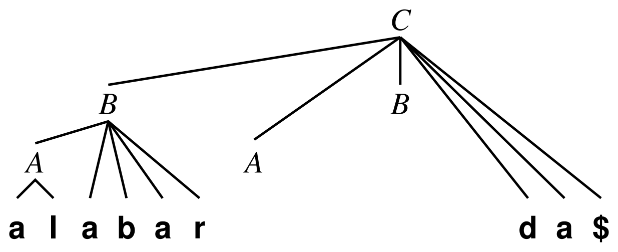

Fig. 5. The grammar tree of the grammar of Figure 4.

terminals that spell out S, and each internal node is labeled with a nonterminal A: if $A \rightarrow X_1 \cdots X_k$, then the node has k children labeled, left to right, $X_1, \ldots, X_k$.

Kieffer and Yang [2000] prove that grammars that satisfy a few reasonable rules reach the kth order entropy of a source, and the same holds for the empirical entropy of individual strings [Ochoa and Navarro 2019]. Their size is always $O(n/\log_\sigma n)$.

Grammar compression is interesting for us because repetitive strings should have small grammars. Our associated measure of repetitiveness is then the size $g = g(S)$ of the smallest grammar that generates only S. It is known that $z_{no} \le g = O(z_{no} \log(n/z_{no}))$ [Charikar et al. 2002; Rytter 2003; Charikar et al. 2005], and even $g = O(z \log(n/z))$ [Gawrychowski 2011, Lem. 8].

Finding such a smallest grammar is NP-complete, however [Storer and Szymanski 1982; Charikar et al. 2005]. This has not made grammar compression unpopular, because several efficient constructions yield grammars of size $O(z_{no} \log(n/z_{no}))$ and even $O(z \log(n/z))$ [Rytter 2003; Charikar et al. 2005; Sakamoto 2005; Jeż 2015; Jeż 2016]. Further, heuristics like RePair [Larsson and Moffat 2000] or Sequitur [Nevill-Manning et al. 1994] perform extremely well and are preferred in practice.

Since it always holds that $z_{no} \le g$, a natural question is why grammar compression is interesting. One important reason is that grammars allow for direct access to the compressed string in logarithmic time, as we will describe in Section 4.1. For now, a simple version illustrates its power. A grammar construction algorithm produces *balanced* grammars if the height of their parse tree is $O(\log n)$ when built on strings of length n. On a balanced grammar for $S[1\,..\,n]$, with constant-size rules, it is very easy to extract any symbol $S[i]$ by virtually traversing the parse tree, if one stores the lengths of the string represented by each nonterminal. The first grammar constructions built from a Lempel-Ziv parse [Charikar et al. 2002; Rytter 2003] had these properties, and thus they were the first structures of size $O(z_{no} \log(n/z_{no}))$ with access time $O(\log n)$.

A little more notation on grammars will be useful. We call $exp(A)$ the string of terminals to which nonterminal A expands, and $|A| = |exp(A)|$. To simplify matters, we forbid rules of right-hand side length 0 or 1. An important concept will be the *grammar tree* (cf. partial parse-tree [Rytter 2003]), which is obtained by pruning the parse tree: for each nonterminal A, only one internal node labeled A is retained; all the others are converted to leaves by pruning their subtree. Since the grammar tree will have the k children of each unique nonterminal $A \rightarrow X_1 \cdots X_k$, plus the root, its total number of nodes is $g + 1$ for a grammar of size g.

With the grammar tree we can easily see that $z_{no} \le g$, for example. Consider a grammar tree where only leftmost occurrence of every nonterminal is an internal node. The string S is then cut into at most g substrings, each covered by a leaf of the grammar tree. Each leaf is either a terminal or a pruned nonterminal. We can then define a left-to-right parse with g phrases: the phrase covered by pruned nonterminal A points to its copy below the internal node A, which is to its left; terminal leaves become explicit symbols. Since this is a left-to-right parse with no overlaps and z_{no} is the least size of such a parse, we have $z_{no} \leq g$.

Example: Figure 5 shows the grammar tree of the parse tree of Figure 4, with 14 *nodes. It induces the left-to-right parse* $S = \boxed{\mathsf{a}}|\boxed{\mathsf{l}}|\boxed{\mathsf{a}}|\boxed{\mathsf{b}}|\boxed{\mathsf{a}}|\boxed{\mathsf{r}}|\mathsf{al}|\mathsf{alabar}|\boxed{\mathsf{d}}|\boxed{\mathsf{a}}|\boxed{\$}$.

3.4.1 *Run-length grammars.* To handle some anomalies that occur when compressing repetitive strings with grammars, Nishimoto et al. [2016] proposed to enrich context-free grammars with *run-length* rules, which are of the form $A \rightarrow X^t$, where X is a terminal or a nonterminal and $t \geq 2$ is an integer. The rule is equivalent to $A \rightarrow X \cdots X$ with t repetitions of X, but it is assumed to be of size 2. Grammars that use run-length rules are called *run-length (context-free) grammars.*

We call $g_{rl} = g_{rl}(S)$ the size of the smallest run-length grammar that generates S. It obviously holds that $g_{rl}(S) \leq g(S)$ for every string S. It can also be proved that $z(S) \leq g_{rl}(S)$ for every string S [Gagie et al. 2018][15]. An interesting connection with bidirectional macro schemes is that $g_{rl} = O(b \log(n/b))$ (from where $z = O(b \log(n/b))$ is obtained) [Gagie et al. 2018; Navarro et al. 2021]. There is no clear dominance between g_{rl} and z_{no}, however: On the string family $S = \mathsf{a}^{n-1}\$$ we have $g_{rl} = O(1)$ and $z_{no} = \Theta(\log n)$ (as well as $g = \Theta(\log n)$), but there exist string families where $g_{rl} = \Omega(z_{no} \log n / \log\log n)$ [Bille et al. 2018] (the weaker result $g = \Omega(z_{no} \log n / \log\log n)$ was known before [Charikar et al. 2005; Hucke et al. 2016]).

The parse tree of a run-length grammar is the same as if rules $A \rightarrow X^t$ were written as $A \rightarrow X \cdots X$. The grammar tree, instead, is modified to ensure it has $g_{rl} + 1$ nodes if the grammar is of size g_{rl}: The internal node labeled A has two children, the left one is labeled X and the right one is labeled $X^{[t-1]}$. Those special marked nodes are treated differently in the various indexes and access methods on run-length grammars.

3.5 Collage Systems: Measure c

To generalize sequential pattern matching algorithms, Kida et al. [2003] proposed an extension of run-length grammars called *collage systems.* These allow, in addition, *truncation* rules of the form $A \rightarrow B^{[t]}$ and $A \rightarrow {}^{[t]}B$, which are of size 2 and mean that $exp(A)$ consists of the first or last t symbols of $exp(B)$, respectively. Collage systems also extend the *composition systems* of Gasieniec et al. [1996], which lack the run-length rules.

Example: A collage system generating $S = \mathsf{alabaralalabarda\$}$*, though larger than our grammar, is* $A \rightarrow \mathsf{al}$*,* $B \rightarrow AA\mathsf{abar}$*,* $B' \rightarrow {}^{[6]}B$*, and the initial rule* $C \rightarrow B'B\mathsf{da\$}$*.*

The size of the smallest collage system generating a string S is called $c = c(S)$, and thus it obviously holds that $c(S) \leq g_{rl}(S)$ for every string S. Kida et al. [2003]

[15]They claim $z \leq 2g_{rl}$ because they use a different definition of grammar size.

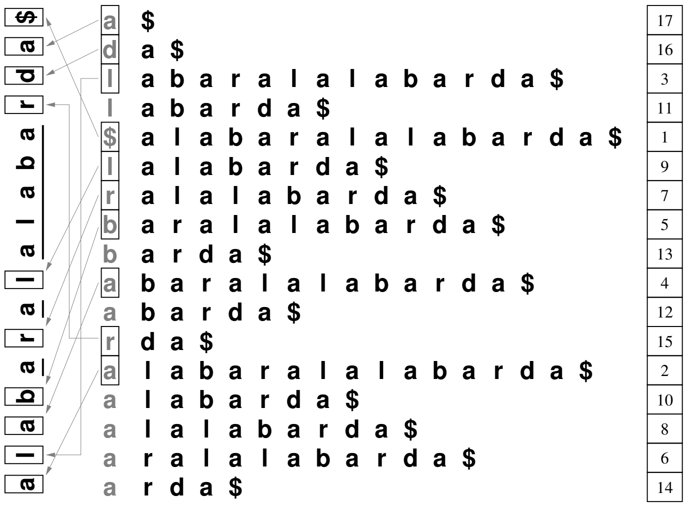
Fig. 6. The list of suffixes of $S =$ alabaralalabarda\$ in increasing lexicographic order. The sequence of preceding symbols (in gray) forms the BWT of S, $S^{bwt} =$ adll\$lrbbaaraaaaa. The run heads are boxed. To their left, we show how they are mapped to S and become the explicit symbols of the induced bidirectional macro scheme, using the same conventions of Figure 2. On the right we show the suffix array of S.

also proved that $c = O(z \log z)$; a better bound $c = O(z)$ was recently proved by Navarro et al. [2021]. This is interesting because it sheds light on what must be added to grammars in order to make them as powerful as Lempel-Ziv parses.

Navarro et al. [2021] also prove lower bounds on c when restricted to what they call *internal* collage systems, where $exp(A)$ must appear in S for every nonterminal A. This avoids collage systems that generate a huge string from which a small string S is then obtained by truncation. For internal collage systems it holds that $b = O(c)$, and on Fibonacci words it holds $b = O(1)$ and $c = \Theta(\log n)$. Instead, while the bound $c = O(z)$ also holds for internal collage systems, it is unknown if there are string families where $c = o(z)$, even for general collage systems.

3.6 Burrows-Wheeler Transform: Measure r

Burrows and Wheeler [1994] designed a reversible transformation with the goal of making strings easier to compress by local methods. The Burrows-Wheeler Transform (BWT) of $S[1..n]$, S^{bwt}, is a permutation of S obtained as follows:

(1) Sort all the suffixes of S lexicographically (as in the suffix array).
(2) Collect, in increasing order, the symbol *preceding* each suffix (the symbol preceding the longest suffix is taken to be \$).

Example: The BWT of $S =$ alabaralalabarda\$ *is $S^{bwt} =$* adll\$lrbbaaraaaaa*, as shown in Figure 6 (ignore the leftmost part of the figure for now).*

It turns out that, by properly partitioning S^{bwt} and applying zeroth-order compression to each piece, one obtains kth order compression of S [Manzini 2001;

Ferragina et al. 2005; Gog et al. 2019]. The reason behind this fact is that the BWT puts together all the suffixes starting with the same context C of length k, for any k. Then, encoding the symbols preceding those suffixes is the same as encoding together the symbols preceding the occurrences of each context C in S. Compare with the definition of empirical kth order entropy we gave in Section 3.1 applied to the reverse of S, $H_k(S^{rev})$.

The BWT has a strong connection with the suffix array A of S; see the right of Figure 6: it is not hard to see that

$$S^{bwt}[j] \quad = \quad S[A[j]-1],$$

if we interpret $S[0] = S[n]$. From the suffix array, which can be built in linear time [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen et al. 2006], we easily compute the BWT. The BWT is also easily reversed in linear time [Burrows and Wheeler 1994] and even within compact space [Kärkkäinen and Puglisi 2010; Kärkkäinen et al. 2012]. The connection between the BWT and the suffix array has been used to implement fast searches on $S[1\,..\,n]$ in $nH_k(S) + o(n \log \sigma)$ bits of space [Ferragina and Manzini 2005; Navarro and Mäkinen 2007].

In addition, the BWT has interesting properties on highly repetitive strings, with connections to the measures we have been studying. Let us define $r = r(S)$ as the number of equal-symbol *runs* in S^{bwt}. Since the BWT is reversible, we can represent S in $O(r)$ space, by encoding the r symbols and run lengths of S^{bwt}. While it is not known how to provide fast access to any $S[i]$ within this $O(r)$ space, it is possible to provide fast pattern searches by emulating the original BWT-based indexes [Mäkinen and Navarro 2005; Mäkinen et al. 2010; Gagie et al. 2020].

Example: The BWT of $S = \mathsf{alabaralalabarda\$}$, $S^{bwt} = \mathsf{adll\$lrbbaaraaaaa}$, *has* $r(S) = 10$ *runs. It can be encoded by the symbols and lengths of the runs:* $(\mathsf{a},1), (\mathsf{d},1), (\mathsf{l},2), (\$,1), (\mathsf{l},1), (\mathsf{r},1), (\mathsf{b},2), (\mathsf{a},2), (\mathsf{r},1), (\mathsf{a},5)$, *and then we can recover* S *from* S^{bwt}.

There is no direct dominance relation between BWT runs and Lempel-Ziv parses or grammars: on the de Bruijn sequences over binary alphabets,[16] it holds $r = \Theta(n)$ [Belazzougui et al. 2015] and thus, since $g = O(n/\log n)$, we have $r = \Omega(g \log n)$. On the even Fibonacci words, on the other hand, it holds that $r = O(1)$ and $z = \Theta(\log n)$ [Prezza 2016; Navarro et al. 2021].

Interestingly, a relation of r with bidirectional macro schemes can be proved, $b = O(r)$ [Navarro et al. 2021], by noting that the BWT runs induce a bidirectional macro scheme of size $2r$: If we map each position $S^{bwt}[j]$ starting a run to the position $S[i]$ where $S^{bwt}[j]$ occurs, then we define the phrases as all those explicit positions i plus the (non-explicit) substrings between those explicit positions. Since that is shown to be a valid bidirectional scheme, it follows that $b \le 2r$.

Example: On $S = \mathsf{alabaralalabarda\$}$, *with* $S^{bwt} = \mathsf{adll\$lrbbaaraaaaa}$, *the corresponding bidirectional macro scheme is* $S = \boxed{\mathsf{a}}\|\boxed{\mathsf{l}}\|\boxed{\mathsf{a}}\|\boxed{\mathsf{b}}\|\mathsf{a}\|\boxed{\mathsf{r}}\|\mathsf{a}\|\boxed{\mathsf{l}}\|\mathsf{alaba}\|\boxed{\mathsf{r}}\|\boxed{\mathsf{d}}\|\boxed{\mathsf{a}}\|\boxed{\$}$, *of size* 13. *See the leftmost part of Figure 6.*

As a final observation, we note that a drawback of r as a repetitiveness measure is that $r(S)$ and $r(S^{rev})$ may differ by a factor of $\Omega(\log n)$ [Giuliani et al. 2020].

[16]The de Bruijn sequence of degree k over an alphabet of size σ contains all the possible substrings of length k within the minimum possible length, $\sigma^k + k - 1$.

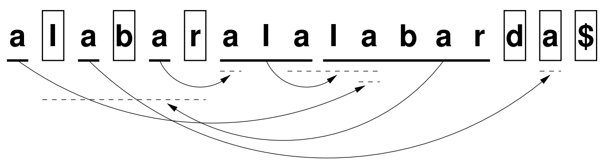

Fig. 7. The lex-parse of $S = \mathsf{alabaralalabarda\$}$, with the same conventions of Figure 2.

Further, r depends on the order of the alphabet; it is NP-hard to find the alphabet permutation that minimizes r [Bentley et al. 2019].

Example: Replacing a *by* e *in* $S = \mathsf{alabaralalabarda\$}$ *we obtain* $S' = \mathsf{eleberelelebede\$}$*, whose BWT has only* $r(S') = 8$ *runs.*

3.7 Lexicographic Parsing: Measure v

Navarro et al. [2021] generalized the Lempel-Ziv parsing into "ordered" parsings, which are bidirectional macro schemes where each nonexplicit phrase equals some substring of S that is smaller under some criterion (in Lempel-Ziv, the criterion is to start earlier in S). A particularly interesting case are the so-called *lexicographic parsings*, where each nonexplicit phrase $S[i\,..\,j]$ must have a copy $S[i'\,..\,j']$ where the suffix $S[i'\,..]$ is lexicographically smaller than $S[i\,..]$. The smallest lexicographic parse of S is called the *lex-parse* of S and has $v = v(S)$ phrases. It is obtained by processing S left to right and maximizing the length of each phrase, as for Lempel-Ziv, that is, $S[i\,..\,j]$ is the longest prefix of $S[i\,..]$ that occurs at some $S[i'\,..\,j']$ where the suffix $S[i'\,..]$ is lexicographically smaller than $S[i\,..]$. Note that this is the same to say that $S[i'\,..]$ is the suffix that lexicographically precedes $S[i\,..]$.

Example: Figure 7 gives the lex-parse of $S = \mathsf{a}|\boxed{\mathsf{l}}|\mathsf{a}|\boxed{\mathsf{b}}|\mathsf{a}|\boxed{\mathsf{r}}|\mathsf{ala}|\mathsf{labar}|\boxed{\mathsf{d}}|\boxed{\mathsf{a}}|\boxed{\$}$*, of size* $v(S) = 11$*. For example, the first phrase is* a *because the suffix* $\mathsf{alabaralalabarda\$}$ *shares a prefix of length* 1 *with its lexicographically preceding suffix,* $\mathsf{abarda\$}$*, and the second phrase is the explicit symbol* $\boxed{\mathsf{l}}$ *because it shares no prefix with its lexicographically preceding suffix,* $\mathsf{da\$}$*. Just like* r*, the value of* v *depends on the alphabet ordering, for example for* $S' = \mathsf{eleberelelebede\$}$ *we have* $v(S') = 10$*.*

Measure v has several interesting characteristics. First, it can be computed in linear time via the so-called longest common prefix array [Kasai et al. 2001]. Second, apart from $b = O(v)$ because the lex-parse is a bidirectional macro scheme, it holds that $v = O(r)$, because the bidirectional macro scheme induced by the runs of S^{bwt} is also lexicographic [Navarro et al. 2021] (v similarly subsumes other lexicographic parses like *lcpcomp* [Dinklage et al. 2017]). Note that, although r and z are incomparable, v is never asymptotically larger than r. Navarro et al. [2021] also connect v with grammars, by showing that $v \leq g_{rl}$, and therefore $v = O(b\log(n/b))$ and $v \leq nH_k + o(n\log\sigma)$. It follows that $r = \Omega(v\log n)$ on binary de Bruijn sequences, where $r = \Theta(n)$ and $v = O(n/\log n)$. They also show that $v = \Theta(\log n)$ (and thus $v = \Omega(b\log n)$) on the odd Fibonacci words and $r = O(1)$ (and thus $c = \Omega(r\log n)$) on the even ones. On the other hand, it is unknown if there are string families where $z = o(v)$.

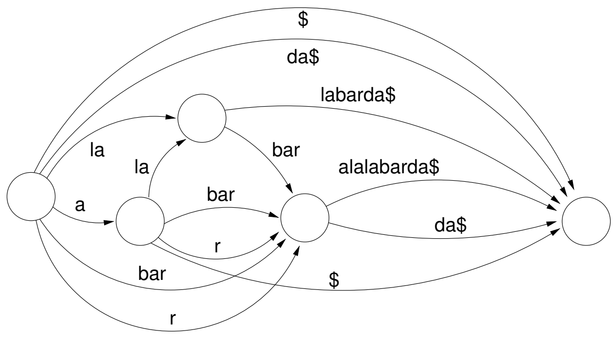

Fig. 8. The CDAWG of string $S =$ alabaralalabarda\$.

3.8 Compact Directed Acyclic Word Graphs: Measure e

Measure r exposes the regularities that appear in the suffix array of repetitive sequences S. As seen in Section 2.2, the suffix array corresponds to the leaves of the suffix tree, where each suffix of S labels a path towards a distinct leaf. A Compact Directed Acyclic Word Graph (CDAWG) [Blumer et al. 1987] is obtained by merging all the identical subtrees of the suffix tree. The suffix trees of repetitive strings tend to have large isomorphic subtrees, which yields small CDAWGs. The number e of nodes plus edges in the CDAWG of S, is then a repetitiveness measure. The CDAWG is also built in linear time [Blumer et al. 1987].

Example: Figure 8 shows the CDAWG of $S =$ alabaralalabarda\$*, and Figure 1 shows its suffix tree and array. The CDAWG has* 5 *nodes and* 14 *edges, so* $e = 19$*. Note, for example, that all the suffixes starting with* r *are preceded by* a*, all the suffixes starting with* ar *are preceded by* b*, and so on until* alabar*. This causes identical subtrees at the nodes reached by all those substrings,* r*,* ar*,* bar*,* abar*,* labar*, and* alabar*. All those nodes become the single CDAWG node that is reachable from the root by those strings. This also relates with r: the suffix tree nodes correspond to suffix array ranges* $A[16..17]$*,* $A[8..9]$*,* $A[10..11]$*,* $A[3..4]$*,* $A[13..14]$*, and* $A[5..6]$*. All but the last such intervals, consequently, fall inside BWT runs with symbols* a*,* b*,* a*,* l*, and* a*. There are other larger identical subtrees, like those rooted by the nodes reached by* la *and* ala*, corresponding to intervals* $A[13..15]$ *and* $A[5..8]$*, the first of which is within a run,* $S^{bwt}[13..15] =$ aaa*. Finally, the run* $S^{bwt}[13..17] =$ aaaaa *corresponds to two consecutive equal subtrees, with the suffix array intervals* $A[13..17]$ *and* $A[5..9]$*.*

This is the weakest repetitiveness measure among those we study. It always holds that $e = \Omega(\max(z, r))$ [Belazzougui et al. 2015] and $e = \Omega(g)$ [Belazzougui and Cunial 2017]. Worse, on some string families (as simple as $\mathsf{a}^{n-1}\$$) e can be $\Theta(n)$ times larger than r or z [Belazzougui et al. 2015] and $\Theta(n/\log n)$ times larger than g [Belazzougui and Cunial 2017]. The CDAWG is, on the other hand, well suited for pattern searching, due to its strong connection with the suffix tree.

3.9 String Attractors: Measure γ

Kempa and Prezza [2018] proposed a new measure of repetitiveness that takes a different approach: It is a direct measure on the string S instead of the result of a specific compression method. Their goal was to unify the existing measures into a cleaner and more abstract characterization of the string. An *attractor* of S is a set Γ of positions in S such that any substring $S[i\,..\,j]$ must have a copy including an element of Γ. The substrings of a repetitive string should be covered with small attractors. The measure is then $\gamma = \gamma(S)$, the smallest size of an attractor Γ of S. This measure is obviously invariant to string reversals, $\gamma(S) = \gamma(S^{rev})$, but it is not monotonic when we append symbols to the string [Mantaci et al. 2021].

Example: An attractor of string $S =$ alabaralalabarda\$ is $\Gamma = \{4, 6, 7, 8, 15, 17\}$. We know that this is the smallest possible attractor, $\gamma(S) = 6$, because it coincides with the alphabet size σ, and it must obviously hold that $\gamma \geq \sigma$.

In general, it is NP-complete to find the smallest attractor size for S [Kempa and Prezza 2018], but in exchange they show that $\gamma = O(\min(b, c, z, z_{no}, r, g_{rl}, g))$.[17] Note that, with current knowledge, it would be sufficient to prove that $\gamma = O(b)$, because b asymptotically lower-bounds all those measures, as well as v. Indeed, we can easily see that $\gamma \leq b$: given a bidirectional macro scheme, take its explicit symbol positions as the attractor Γ. Every substring $S[i\,..\,j]$ not containing an explicit symbol (i.e., a position of Γ) is inside a phrase and thus it occurs somewhere else, in particular at $S[f(i)\,..\,f(j)]$. If this new substring does not contain an explicit position, we continue with $S[f^2(i)\,..\,f^2(j)]$, and so on. In a valid macro scheme we must eventually succeed; therefore Γ is a valid attractor.

Example: Our example attractor $\Gamma = \{4, 6, 7, 8, 15, 17\}$ is derived in this way from the bidirectional macro scheme of Figure 3.

That is, γ is a lower bound to all the other repetitiveness measures. Yet, we do not know if it is reachable, that is, if we can represent S within space $O(\gamma)$. Instead, Kempa and Prezza [2018] show that $O(\gamma \log(n/\gamma))$ space suffices not only to encode S but also to provide logarithmic-time access to any $S[i]$ (Section 4.2.3).

Christiansen et al. [2020] also show how to support indexed searches within $O(\gamma \log(n/\gamma))$ space. They actually build a particular run-length grammar of that size, thus implying the bound $g_{rl} = O(\gamma \log(n/\gamma))$. A stronger bound $g = O(\gamma \log(n/\gamma))$ is very recent.[18]

3.10 String Complexity: Measure δ

Our final measure of repetitiveness for a string S, $\delta = \delta(S)$, is built on top of the concept of *string complexity*, that is, the number $S(k)$ of distinct substrings of length k. Raskhodnikova et al. [2013] define $\delta = \max\{S(k)/k, 1 \leq k \leq n\}$. It is not hard to see that $\delta(S) \leq \gamma(S)$ for every string S [Christiansen et al. 2020]: Since every substring of length k in S has a copy including some of its γ attractor elements, there can be only $k\gamma$ distinct substrings, that is, $S(k) \leq k\gamma$ for all k.

[17]For c they consider internal collage systems, recall Section 3.5.
[18]T. Kociumaka, personal communication.

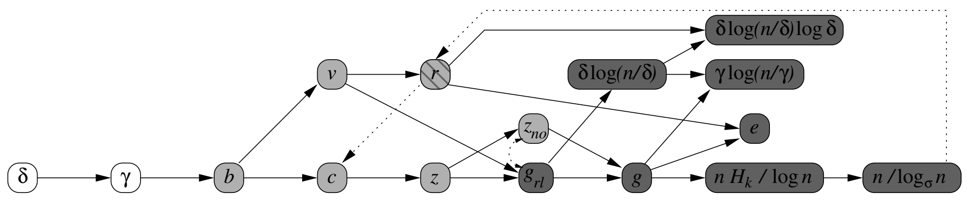

Fig. 9. Relations between the compressibility measures. A solid arrow from X to Y means that $X = O(Y)$ for all string families. For all solid and dotted arrows, there are string families where $X = o(Y)$, with the exceptions of $\gamma \to b$ and $c \to z$. Grayed measures X mean that we can encode every string in $O(X)$ space; darker gray means that we can also provide efficient access and indexed searches within $O(X)$ space; for r we can only provide indexed searches.

Christiansen et al. [2020] also show how δ can be computed in linear time. Further, δ is clearly monotonic and invariant upon reversals.

Example: For our string $S =$ alabaralalabarda\$ *we have* $S(1) = 6$*,* $S(2) = 9$*,* $S(3) = 10$*,* $S(4) = S(5) = S(6) = 11$*, and* $S(k) = 17 - k + 1$ *for* $k > 6$ *(i.e., all the substrings of length over 6 are different); therefore* $\delta(S) = 6$*.*

Kociumaka et al. [2020, 2021] show that, for every δ, there are string families whose members all have measure δ and where $\gamma = \Omega(\delta \log(n/\delta))$. Although δ is then strictly stronger than γ as a compressibility measure, they also show that it is possible not only to represent S within $O(\delta \log(n/\delta))$ space, but also to efficiently access any symbol $S[i]$ (see Section 4.2.3) and support indexed searches on S within that space. Indeed, this space is optimal as a function of δ: for every $2 \le \delta \le n$ there are string families that need $\Omega(\delta \log(n/\delta))$ space to be represented. This means that we know that $o(\delta \log n)$ space is unreachable in general, whereas it is unknown if $o(\gamma \log n)$ space can always be reached.

Raskhodnikova et al. [2013] prove that $z = O(\delta \log(n/\delta))$, and it can also be proved that $g_{rl} = O(\delta \log(n/\delta))$ by building a run-length grammar of that size [Kociumaka et al. 2021]. The same cannot be said about g: Kociumaka et al. [2021] prove that for every n and $2 \le \delta \le n$ there are string families where $g = \Omega(\delta \log^2(n/\delta)/\log\log(n/\delta))$. This establishes another separation between g_{rl} and g. Recently, the only upper bound on r in terms of another repetitiveness measure was obtained: $r = O(\delta \log(n/\delta) \log \delta)$ [Kempa and Kociumaka 2020]; this also shows that $r(S)$ and $r(S^{rev})$ can differ only by a factor up to $O(\log^2 n)$, because δ is invariant upon reversals.

3.11 Wrapping Up

Figure 9 summarizes what is known about the repetitiveness measures we have covered. Those in gray are reachable, and those in dark gray support efficient access (as we will see in Section 4) and indexing (as we see in Part II of this survey [Navarro 2020]). An intriguing case is r, which allows for efficient indexing but not access, as far as we know.

Note that we do not know if γ should be grayed or not, whereas we do know that δ must not be grayed. The smallest grayed measure is b, which is, by definition, the best space we can obtain via copying substrings from elsewhere in the string.

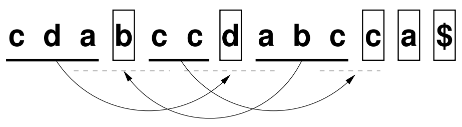

Fig. 10. The only possible, yet invalid, bidirectional macro scheme induced by the explicit positions $\Gamma = \{4, 7, 11, 12, 13\}$ of the attractor Γ of the string $S =$ cdabccdabcca\$. We use the same conventions of Figure 2.

We have shown that every bidirectional macro scheme can be converted into an attractor of at most the same size, and thus $\gamma \le b$. The converse is not direct: if we take the positions of an attractor Γ as the explicit symbols of a bidirectional macro scheme, and declare that the gaps are the nonexplicit phrases, the result may not be a valid macro scheme, because it might be impossible to define a target-to-source function f without cycles.

Example: Figure 10 shows an example of this case [Kempa and Prezza 2018]. The string $S =$ cdabccdabcca\$ *has attractor $\Gamma = \{4, 7, 11, 12, 13\}$. The only possible bidirectional macro scheme with those explicit symbols has the cycle $f(3) = f(8) = f(3)$, and thus it is not valid.*

Still, it could be that one can always add $O(\gamma)$ explicit symbols to break those cycles, in which case γ and b would be asymptotically equivalent. Otherwise, if there are string families where $\gamma = o(b)$, this still does not mean that we cannot represent a string S within $O(\gamma)$ space, but that representation will not consist of just substring copies.[19] To definitely show that not all strings can be represented in $O(\gamma)$ space, we should find a string family of common measure γ and of size $n^{\omega(\gamma)}$.

As explained, $\delta \log(n/\delta)$ is a reachable measure that is asymptotically optimal as a function of δ. That is, if we separate the set of all strings into subsets $\mathcal{S}_\delta$ where the strings have measure δ, then inside each set there are families that need $\Theta(\delta \log(n/\delta))$ space to be represented. The measure δ is then optimal in that coarse sense. Still, we know that b is string-wise better than $\delta \log(n/\delta)$ and sometimes asymptotically smaller; therefore it is a more refined measure.

3.11.1 *Some practical figures.* Several direct [Belazzougui et al. 2015; Gagie et al. 2020; Navarro et al. 2021; Russo et al. 2020] and less direct [Mäkinen et al. 2010; Kreft and Navarro 2013; Claude et al. 2016; Belazzougui et al. 2017] experiments suggest that in typical repetitive texts it holds that $b < z \approx v < g < r < e$, where "$<$" denotes a clear difference in magnitude.

Since those experiments have been made on different texts and it is hard to combine them, and because no experiments on δ have been published, Table 1 compares the measures that can be computed in polynomial time on a sample of repetitive collections obtained from the Repetitive Corpus of Pizza&Chili[20]. We include an upper bound on g obtained by a heuristically balanced RePair algorithm,[21] which

[19]Kutsukake et al. [2020] show that $\gamma \le 4$ and $\delta < 10/3$ for Thue-Morse words, and conjecture that $b = \Theta(\log n)$. If confirmed, it would be the first known strign family with $\gamma = o(b)$.
[20]`http://pizzachili.dcc.uchile.cl/repcorpus/real`
[21]`https://www.dcc.uchile.cl/gnavarro/software/repair.tgz`, directory `bal/`. To further reduce the grammar, we removed rules defining nonterminals that were used only once.

File	n	$\lfloor\delta\rfloor$	z	v	g	r	e
cere	461,286,644	1,003,280	1,700,630	1,649,448	4,069,452	11,574,640	35,760,304
escherichia	112,689,515	1,337,977	2,078,512	2,014,012	4,342,874	15,044,487	43,356,169
einstein	467,626,544	42,884	89,467	97,442	212,902	290,238	
worldleaders	46,968,181	68,651	175,740	179,696	399,667	573,487	
coreutils	205,281,778	636,101	1,446,468	1,439,918	2,409,429	4,684,460	
kernel	257,961,616	405,643	793,915	794,058	1,374,651	2,791,367	

Table 1. Several repetitiveness measures computed on a sample of the Pizza&Chili repetitive corpus. We chose two DNA, two natural language, and two source code files.

consistently outperforms other grammar-based compressors, including those that offer theoretical guarantees of approximation to g. We also build the CDAWG on the DNA collections, where the code we have allows it[22]. The table suggests $z \approx v \approx 1.5{-}2.5 \cdot \delta$, $g \approx 3{-}6 \cdot \delta$, $r \approx 7{-}11 \cdot \delta$, and $e \approx 32{-}35 \cdot \delta$.

4. ACCESSING THE COMPRESSED TEXT AND COMPUTING FINGERPRINTS

The first step beyond mere compression, and towards compressed indexing, is to provide direct access to the compressed string without having to fully decompress it. We wish to extract arbitrary substrings $S[i\,..\,j]$ in a time that depends only polylogarithmically on n. Further, some indexes also need to efficiently compute Karp-Rabin fingerprints (Section 2.3) of arbitrary substrings.

In this section we cover the techniques and data structures used to provide these functionalities, depending on the underlying compression method. In all cases, any substring $S[i\,..\,j]$ can be extracted in time $O(j-i+\log n)$ or less, whereas the fingerprint of any $S[i\,..\,j]$ can be computed in time $O(\log n)$ or less. Section 4.1 shows how this is done in $O(g)$ and even $O(g_{rl})$ space by enriching (run-length) context-free grammars, whereas Section 4.2 shows how to do this in space $O(z \log(n/z))$, and even $O(\delta \log(n/\delta))$, by using so-called block trees. Some indexes require more restricted forms of access, which those data structures can provide in less time. Section 4.3 shows another speedup technique called bookmarking. Finally, Section 4.4 shows a very recent development where the concept of persistent data structures is used to offer direct access on versioned collections where the edit structure is known, in space proportional to the base text plus the total amount of edits.

Experimental results [Belazzougui et al. 2021] show that practical access data structures built on block trees take about the same space as those built on balanced grammars (created with RePair [Larsson and Moffat 2000]), but block trees grow faster as soon as the repetitiveness decreases. On the other hand, access on block trees is over an order of magnitude faster than on grammars.

4.1 Enhanced Grammars

If the compressed string is represented with a context-free grammar of size g or a run-length grammar of size g_{rl}, we can enrich the nonterminals with information associated with the length of the string they expand to, so as to provide efficient access within space $O(g)$ or $O(g_{rl})$, respectively.

[22]`https://github.com/mathieuraffinot/locate-cdawg`, which works only on the alphabet $\Sigma = \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{T}\}$. For escherichia we converted the other symbols to A, and verified that this would have a negligible impact on the other measures.

For a simple start, let $A \rightarrow X_1 \cdots X_k$. Then, we store $\ell_0 = 0$, $\ell_1 = \ell_0 + |X_1|$, $\ell_2 = \ell_1 + |X_2|$, $\ldots$, $\ell_k = \ell_{k-1} + |X_k|$ associated with A. To extract the ith symbol of $exp(A)$, we look for the predecessor of i in those values, finding j such that $\ell_{j-1} < i \le \ell_j$, and then seek to obtain the i'th symbol of X_j, with $i' = i - \ell_{j-1}$. Since predecessors can be computed in time $O(\log\log_w n)$ [Belazzougui and Navarro 2015; Navarro and Rojas-Ledesma 2020], on a grammar of height h we can extract any $S[i]$ in time $O(h \log\log_w n)$, which is $O(\log n \log\log_w n)$ if the grammar is balanced. If the right-hand sides of the rules are of constant length, then the predecessors take constant time and the extraction time drops to $O(\log n)$, as with the simple method described in Section 3.4.

Bille et al. [2015] showed how this simple idea can be extended to extract any $S[i]$ in time $O(\log n)$ from arbitrary grammars, not necessarily balanced. They extract a *heavy path* [Sleator and Tarjan 1983] from the parse tree of S. A heavy path starts at the root $A \rightarrow X_1 \cdots X_k$ and continues by the child X_j with the longest expansion, that is, with maximum $|X_j|$ (breaking ties in some deterministic way), until reaching a leaf. We store the heavy path separately and remove all its nodes and edges from the parse tree, which gets disconnected and becomes a forest. We then repeat the process from each root of the forest until all the nodes are in the extracted heavy paths.

Consider the path going through a node labeled B in the parse tree, whose last element is the terminal $exp(B)[t_B]$. We associate with B its start and end values relative to t_B, $s_B = 1 - t_B$ and $e_B = |B| - t_B$, respectively. Note that these values will be the same wherever B appears in the parse tree, because the heavy path starting from B will be identical. Further, if C follows B in the heavy path, then $exp(C)[t_C]$ is the same symbol $exp(B)[t_B]$. For a heavy path rooted at A, the values s_B of the nodes we traverse downwards to the leaf, then the zero, and then the values e_B of the nodes we traverse upwards to A again, form an increasing sequence of positions, P_A. The search for $S[i]$ then proceeds as follows. We search for the predecessor of $i - t_A$ in the sequence P_A associated with the root symbol A. Say that B is followed by C downwards in the path and their starting positions are $s_B \le i - t_A < s_C$, or their ending positions are $e_C < i - t_A \le e_B$. Then the search for $S[i]$ must continue as the search for $i' = i - t_A + t_B$ inside B, because i is inside $exp(B)$ but not inside $exp(C)$. With another predecessor search for i' on the starting positions ℓ_j of the children of B, we find the child B_j by which our search continues, with $i'' = i' - \ell_{j-1}$. Note that B_j is the root of another heavy path, and therefore we can proceed recursively.

Example: Figure 11 (left) shows an example for the grammar we used in Figure 4 on our string $S =$ alabaralalabarda\$*. The first heavy path, extracted from the root, is $C \rightarrow B \rightarrow A \rightarrow$* l *(breaking ties arbitrarily). From the other B of the parse tree, which becomes a tree root after we remove the edges of the first heavy path, we extract another heavy path: $B \rightarrow A \rightarrow$* l*. The remaining A produces the final heavy path, $A \rightarrow$* l*. All the other paths have only one node. Note that $t_C = 10$ and $t_B = t_A = 2$, that is, $exp(C)[10] = exp(B)[2] = exp(A)[2] =$* l *is the last element in the heavy path. Therefore, $s_C = -9$, $e_C = 7$, $s_B = -1$, $e_B = 4$, $s_A = -1$, $e_A = 0$. The ℓ values for C are $0, 6, 8, 14, 15, 16, 17$. To find $S[12]$, we determine that $8 < 12 \le 14$, thus we have to descend by B, which follows the heavy path. We then search for*

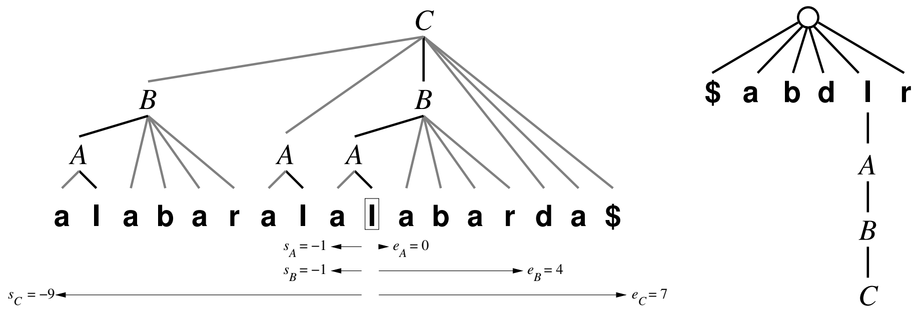

Fig. 11. On the left, the heavy path decomposition of the parse tree of Figure 4. Heavy edges are in black and light edges are in gray. For the heavy path that starts on the root, we box the last element and show how the values s_X and e_X of the intermediate nodes are computed. On the right, the trie storing all the heavy paths.

$12 - t_C = 2$ in the sequence $s_C, s_B, s_A, 0, e_A, e_B, e_C = -9, -1, -1, 0, 0, 4, 7$ to find $0 < 2 \leq 4$, meaning that we fall between $e_A = 0$ and $e_B = 4$. We thus follow the search from B, for the new position $12 - t_C + t_B = 4$. Since the ℓ values associated with B are $0, 2, 3, 4, 5, 6$, we descend by a light edge towards the b*, which is $S[12]$.*

The important property is that, if X_j follows A in the heavy path, then all the other children $X_{j'}$ of A satisfy $|X_{j'}| \leq |A|/2$, because otherwise $X_{j'}$ would have followed A in the heavy path. Therefore, every time we traverse a light edge to switch to another heavy path, the length of the expansion of the nonterminal is halved. As a consequence, we cannot switch more than $\log n$ times to another heavy path in our traversal from the root to the leaf that holds $S[i]$. Since we perform two predecessor searches to find the next heavy path, the total extraction cost is $O(\log n \log\log_w n)$, even if the grammar is unbalanced.

Bille et al. [2015] remove the $O(\log\log_w n)$ factor by using a different predecessor search data structure that, if i falls between positions p_{j-1} and p_j inside a universe of u positions, then the search takes time $O(\log(u/(p_j - p_{j-1})))$. This makes the successive searches on heavy paths and children telescope to $O(\log n)$.

The other problem is that the parse tree has $\Theta(n)$ nodes, and thus we cannot afford storing all the heavy paths. Fortunately, this is not necessary: If X_j is the child of A with the largest $|X_j|$, then X_j will follow A in every heavy path where A appears. We can then store all the heavy paths in a trie where X_j is the *parent* of A. Heavy paths are then read as upward paths in this trie, which has exactly one node per nonterminal and per terminal, the latter being children of the trie root. The trie then represents all the heavy paths within $O(g)$ space. Bille et al. [2015] show how an $O(g)$-space data structure on the trie provides the desired predecessor searches on upward trie paths.

Example: Figure 11 (right) shows the trie associated with our example parse tree. Every time B appears in the parse tree, the heavy path continues by A, so A is the parent of B in the trie.

4.1.1 *Extracting substrings.* Bille et al. [2015] also show how to extract $S[i\,..\,j]$ in time $O(j - i + \log n)$. They find the path towards i and the path towards j. These coincide up to some node in the parse tree, from which they descend by different children. From there, all the subtrees to the right of the path towards i (from the deepest to the shallowest), and then all the subtrees to the left of the path towards j (from the shallowest to the deepest), are fully traversed in order to obtain the $j-i+1$ symbols of $S[i\,..\,j]$ in optimal time (because they output all the leaves of the traversed trees, and internal nodes have at least two children).

With a bit more of sophistication, Belazzougui et al. [2015] obtain RAM-optimal time on the substring length, $O((j-i)/\log_\sigma n + \log n)$, among other tradeoffs. We note that the $O(\log n)$ additive overhead is almost optimal: any structure using $g^{O(1)}$ space requires $\Omega(\log^{1-\epsilon} n)$ time to access a symbol, for any constant $\epsilon > 0$ [Verbin and Yu 2013].

4.1.2 *Karp-Rabin fingerprints.* An easy way to obtain the Karp-Rabin fingerprint of any $S[i\,..\,j]$ is to obtain $\kappa(S[1\,..\,i-1])$ and $\kappa(S[1\,..\,j])$, and then operate them as sketched in Section 2.3. To compute the fingerprint of a prefix of S, Bille et al. [2017] store for each $A \rightarrow X_1 \cdots X_k$ the fingerprints $\kappa_1 = \kappa(exp(X_1))$, $\kappa_2 = \kappa(exp(X_1) \cdot exp(X_2))$, …, $\kappa_k = \kappa(exp(X_1) \cdots exp(X_k))$. Further, for each node B in a heavy path ending at a leaf $exp(B)[t_B]$, they store $\kappa(exp(B)[1\,..\,t_B-1])$. Thus, if we have to leave at B a heavy path that starts in A, the fingerprint of the prefix of $exp(A)$ that precedes $exp(B)$ is obtained by combining $\kappa(exp(A)[1\,..\,t_A-1])$ and $\kappa(exp(B)[1\,..\,t_B-1])$. In our path towards extracting $S[i]$, we can then compose the fingerprints so as to obtain $\kappa(S[1\,..\,i-1])$ at the same time. Any fingerprint $\kappa(S[i\,..\,j])$ can therefore be computed in $O(\log n)$ time.

Example: In the same example of Figure 11, say we want to compute $\kappa(S[1\,..\,11])$. We start with the heavy path that starts at C, which we leave at B. For the heavy path, we have precomputed $\kappa(exp(C)[1\,..\,t_C-1]) = \kappa(\mathsf{alabarala})$ for C and $\kappa(exp(B)[1\,..\,t_B-1]) = \kappa(\mathsf{a})$ for B. By operating them, we obtain the fingerprint of the prefix of $exp(C)$ that precedes $exp(B)$, $\kappa(\mathsf{alabaral})$. We now descend by the second child of B. We have also precomputed the fingerprints of the prefixes of $exp(B)$ corresponding to its children, in particular the first one, $\kappa(exp(A)) = \kappa(\mathsf{al})$. By composing both fingerprints, we have $\kappa(\mathsf{alabaralal})$ as desired.

4.1.3 *Extracting rule prefixes and suffixes in real time.* The typical search algorithm of compressed indexes (see Part II) does not need to extract arbitrary substrings, but only to expand prefixes or suffixes of nonterminals. Gasieniec et al. [2005] showed how one can extract prefixes or suffixes of any $exp(A)$ in real time, that is, $O(1)$ per additional symbol. They build a trie similar to that used to store all the heavy paths, but this time they store leftmost paths (for prefixes) or rightmost paths (for suffixes). That is, if $A \rightarrow X_1 \cdots X_k$, then X_1 is the parent of A in the trie of leftmost paths and X_k is the parent of A in the trie of rightmost paths.

Let us consider leftmost paths; rightmost paths are analogous. To extract the first symbol of $exp(A)$, we go to the root of the trie, descend to the child in the path to node A, and output its corresponding terminal, a. This takes constant time with level ancestor queries [Bender and Farach-Colton 2004]. Let $B \rightarrow aB_2 \cdots B_s$ be the child of a in the path to A (again, found with level ancestor queries from A). The next symbols are then extracted recursively from $B_2, \ldots, B_s$. Once those are exhausted, we continue with the child of B in the path to A, $C \rightarrow BC_2 \cdots C_t$, and extract $C_2, \ldots, C_t$, and so on, until we extract all the desired characters.

4.1.4 *Run-length grammars.* Christiansen et al. [2020] (App. A) showed how the results above can be obtained on run-length context-free grammars as well, relatively easily, by regarding the rule $A \rightarrow X^t$ as $A \rightarrow X \cdots X$ and managing to use only $O(1)$ words of precomputed data in order to simulate the desired operations.

4.1.5 *All context-free grammars can be balanced.* Recently, Ganardi et al. [2019] proved that every context-free grammar of size g can be converted into another of size $O(g)$, right-hand sides of size 2, and height $O(\log n)$. While the conversion seems nontrivial at first sight, once it is carried out we need only very simple information associated with nonterminals to extract any $S[i\,..\,j]$ in time $O(j-i+\log n)$ and to compute any fingerprint $\kappa(S[i\,..\,j])$ in time $O(\log n)$. It is not known, however, if run-length grammars can be balanced in the same way.

4.2 Block Trees and Variants

Block trees [Belazzougui et al. 2015] are in principle built knowing the size z of the Lempel-Ziv parse of $S[1\,..\,n]$. Built with a parameter τ, they provide a way to access any $S[i]$ in time $O(\log_\tau(n/z))$ with a data structure of size $O(z\tau\log_\tau(n/z))$. For example, with $\tau = O(1)$, the time is $O(\log(n/z))$ and the space is $O(z\log(n/z))$. Recall that several heuristics build grammars of size $g = O(z\log(n/z))$, and thus block trees are not asymptotically smaller than structures based on grammars, but they can be asymptotically faster.

The block tree is of height $\log_\tau(n/z)$. The root has z children, $u_1, \ldots, u_z$, which logically divide S into blocks of length n/z, $S = S_{u_1} \cdots S_{u_z}$. Each such node $v = u_j$ has τ children, $v_1, \ldots, v_\tau$, which divide its block S_v into equal parts, $S_v = S_{v_1} \cdots S_{v_\tau}$. The nodes v_i have, in turn, τ children that subdivide their block, and so on. After slightly less than $\log_\tau(n/z)$ levels, the blocks are of length $\log_\sigma n$, and can be stored explicitly using $\log n$ bits, that is, in constant space.

Some of the nodes v can be removed because their block S_v appears earlier in S. The precise mechanism is as follows: every consecutive pair of nodes v_1, v_2 where the concatenation $S_{v_1} \cdot S_{v_2}$ does not appear earlier is *marked* (that is, we mark v_1 and v_2). After this, every unmarked node v has an earlier occurrence, so instead of creating its τ children, we replace v by a leftward pointer to the first occurrence of S_v in S. This first occurrence spans in general two consecutive nodes v_1, v_2 at the same level of v, and these exist and are marked by construction. We then make v a leaf pointing to v_1, v_2, also recording the offset where S_v occurs inside $S_{v_1} \cdot S_{v_2}$.

To extract $S[i]$, we determine the top-level node $v = u_j$ where i falls and then extract its corresponding symbol. In general, to extract $S_v[i]$, there are three cases. (1) If S_v is stored explicitly (i.e., v is a node in the last level), we access $S_v[i]$ directly. (2) If v has τ children, we determine the corresponding child v' of v and the corresponding offset i' inside $S_{v'}$, and descend to the next level looking for $S_{v'}[i']$. (3) If v points to a pair of nodes v_1, v_2 to the left, at the same level, then S_v occurs inside $S_{v_1} \cdot S_{v_2}$. With the offset information, we translate the query $S_v[i]$ into a query inside S_{v_1} or inside S_{v_2}. Since nodes v_1 and v_2 are marked, they have children, so we are now in case (2) and can descend to the next level. Overall, we do $O(1)$ work per level of the block tree, for a total access time of $O(\log_\tau(n/z))$.

To see that the space of this structure is $O(z\tau\log_\tau(n/z))$, it suffices to show that

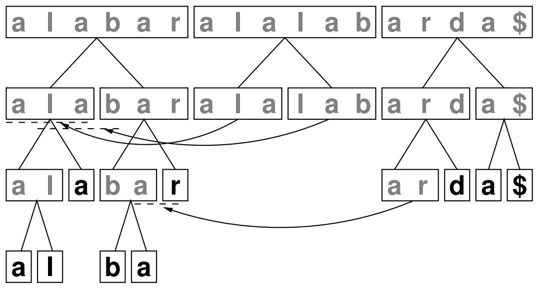
a l a b a r a l a l a b a r d a $
a l a b a r a l a l a b a r d a $
a l a b a r a r d a $
a l b a

Fig. 12. A block tree for S = alabaralalabarda\$, starting with 3 nodes, with arity $\tau = 2$, and stopping on substrings of length 1. Grayed characters are conceptual and not stored; only the black ones and the pointer structure of the tree are represented. Horizontal pointers are drawn with curved arrows and the source areas are shown with dashed lines.

there are $O(z)$ marked nodes per level: we charge $O(\tau)$ space to the marked nodes in a level to account for the space of all the nodes in the next level. Note that, in a given level, there are only z blocks containing Lempel-Ziv phrase boundaries. Every pair of nodes v_1, v_2 without phrase boundaries in $S_{v_1} \cdot S_{v_2}$ has an earlier occurrence because it is inside a Lempel-Ziv phrase. Thus, a node v containing a phrase boundary in S_v may be marked and force its preceding and following nodes to be marked as well, but all the other nodes are unmarked. In conclusion, there can be at most $3z$ marked nodes per level.

Still, the construction is conservative, possibly preserving internal nodes v such that S_v occurs earlier, and no other node points inside S_v nor inside the block of a descendant of v. Such nodes are identified and converted into leaves with a final postorder traversal [Belazzougui et al. 2021].

Example: Figure 12 shows a block tree for S = alabaralalabarda\$ *(we should start with $z(S) = 11$ blocks, but 3 is better to exemplify). To access $S[11]$ we descend by the second block (S_{u_2} =* alalab*), accessing $S_{u_2}[5]$. Since the block has children, we descend by the second (*lab*), aiming for its second position. But this block has no children because its content is replaced by an earlier occurrence of* lab*. The quest for the second position of* lab *then becomes the quest for the third position of* ala*, within the first block of the second level. Since this block has children, we descend to its second child (*a*), aiming for its first position. This block is explicit, so we obtain $S[11]$ =* a*.*

4.2.1 *Extracting substrings.* By storing the first and last $\log_\sigma n$ symbols of every block S_v, a chunk of S of that length can also be extracted in time $O(\log_\tau(n/z))$: we traverse the tree as for a single symbol until the paths for the distinct symbols of the chunk diverge. At this point, the chunk spans more than one block, and thus its content can be assembled from the prefixes and suffixes stored for the involved blocks. Therefore, we can extract any $S[i\,..\,j]$ by chunks of $\log_\sigma n$ symbols, in time $O((1 + (j - i)/\log_\sigma n)\log_\tau(n/z))$.

4.2.2 *Karp-Rabin fingerprints.* Navarro and Prezza [2019] show, on a slight block tree variant, that fingerprints of any substring $S[i\,..\,j]$ can be computed in time

$O(\log_\tau(n/z))$ by storing some precomputed fingerprints: (1) for every top-level node u_j, $\kappa(S_{u_1} \cdots S_{u_j})$; (2) for every internal node v with children $v_1, \ldots, v_\tau$, $\kappa(S_{v_1} \cdots S_{v_j})$ for all j; and (3) for every leaf v pointing leftwards to the occurrence of S_v inside $S_{v_1} \cdot S_{v_2}$, $\kappa(S_v \cap S_{v_1})$. Then, by using the composition operations of Section 2.3, the computation of any prefix fingerprint $\kappa(S[1\,..\,i])$ is translated into the computation of some $\kappa(S_{u_j}[1\,..\,i'])$, and the computation of any $\kappa(S_v[1\,..\,i'])$ is translated into the computation of some $\kappa(S_{v'}[1\,..\,i''])$ at the same level (at most once per level) or at the next level.

4.2.3 *Other variants.* The block tree concept is not as tightly coupled to Lempel-Ziv parsing as it might seem. Kempa and Prezza [2018] build a similar structure on top of an attractor Γ of S, of minimal size $\gamma \le z$. Their structure uses $O(\gamma \log(n/\gamma)) \subseteq O(z \log(n/z))$ space and extracts any $S[i\,..\,j]$ in time $O((1 + (j - i)/\log_\sigma n) \log(n/\gamma))$. Unlike the block tree, their structure divides S irregularly, defining blocks as the areas between consecutive attractor positions. We prefer to describe the so-called Γ-tree [Navarro and Prezza 2019], which is more similar to a block tree and more suitable for indexing.

The Γ-tree starts with γ top-level nodes (i.e., in level 0), each representing a block of length n/γ of S. The nodes of level l represent blocks of length $b_l = n/(\gamma \cdot 2^l)$. At each level l, every node whose block is at distance less than b_l from an attractor position is marked. Marked nodes point to their two children in the next level, whereas unmarked nodes v become leaves pointing to a pair of nodes v_1, v_2 at the same level, where S_v occurs inside $S_{v_1} \cdot S_{v_2}$ and the occurrence contains an attractor position. Because of our marking rules, nodes v_1, v_2 exist and are marked.

It is easy to see that the Γ-tree has height $\log(n/\gamma)$, at most 3γ marked nodes per level, and that it requires $O(\gamma \log(n/\gamma))$ space. This space is better than the classical block tree because $\gamma \le z$. The Γ-tree can retrieve any $S[i]$ in time $O(\log(n/\gamma))$ and can be enhanced to match the substring extraction time of Kempa and Prezza [2018]. As mentioned, they can also compute Karp-Rabin fingerprints of substrings of S in time $O(\log(n/\gamma))$.

Recently, Kociumaka et al. [2020, 2021] showed that the original block tree is easily tuned to use $O(\delta\tau \log_\tau(n/\delta)) \subseteq O(z\tau \log_\tau(n/z))$ space (recall that $\delta \le \gamma \le z$). The only change needed is to start with δ top-level blocks. It can then be seen that there are only $O(\delta)$ marked blocks per level (though the argument is more complex than the previous ones). The tree height is $O(\log_\tau(n/\delta))$, higher than the block tree. However, from $z = O(\delta \log(n/\delta))$, they obtain that $\log(n/\delta) = O(\log(n/g))$, and therefore the difference in query times is not asymptotically relevant.

4.3 Bookmarking

Gagie et al. [2012] combine grammars with Lempel-Ziv parsing to speed up string extraction over (Lempel-Ziv) phrase prefixes and suffixes, and Gagie et al. [2014] extend the result to fingerprinting. We present the ideas in simplified form and on top of the stronger concept of attractors.

Assume we have split S somehow into t phrases, and let Γ be an attractor on S, of size γ. Using a structure from Sections 4.1 or 4.2, we provide access to any substring $S[i\,..\,i+\ell]$ in time $O(\ell + \log n)$, and computation of its Karp-Rabin fingerprint in time $O(\log n)$. Bookmarking enables, within $O((t+\gamma) \log\log n)$ additional space, the extraction of phrase prefixes and suffixes in time $O(\ell)$ and their fingerprint computation in time $O(\log \ell)$.

Let us first handle extraction. We consider only the case $\ell \le \log n$, since otherwise $O(\ell + \log n)$ is already $O(\ell)$. We build a string $S'[1 \mathinner{.\,.} n']$ by collecting all the symbols of S that are at distance at most $\log n$ from an attractor position. It then holds that $n' = O(\gamma \log n)$, and every phrase prefix or suffix of S of length up to $\log n$ appears in S', because it has a copy in S that includes a position of Γ. By storing a pointer from each of the t phrase prefixes or suffixes to a copy in S', using $O(t)$ space, we can focus on extracting substrings from S'.

An attractor Γ' on S' can be obtained by projecting the positions of Γ. Further, if the area between two attractor positions in Γ is longer than $2 \log n$, its prefix and suffix of length $\log n$ are concatenated in S'. In that case we add the middle position to Γ', to cover the possibly novel substrings. Then, Γ' has a position every (at most) $\log n$ symbols of S', and it is an attractor of size $\gamma' \le 2\gamma$ for S'.

Example: Consider the attractor $\Gamma = \{4, 6, 7, 8, 15, 17\}$ *of* $S =$ ala$\boxed{\text{b}}$a$\boxed{\text{r}}\boxed{\text{a}}\boxed{\text{l}}$alabar$\boxed{\text{d}}a\boxed{\text{\$}}$ *(the boxed symbols are the attractor positions). Let us replace* $\log n$ *by* 2 *for the sake of the example. It then holds that* $S' =$ labaralalarda\$. *The attractor we build for* Γ' *includes the positions* $\{3, 5, 6, 7, 12, 14\}$*, projected from* Γ*. In addition, since some middle symbols of the area* $S[9 \mathinner{.\,.} 14] =$ alabar *are removed and it becomes* $S'[8 \mathinner{.\,.} 11] =$ alar*, we add one more attractor in the middle, to obtain* $\Gamma' = \{3, 5, 6, 7, 10, 12, 14\}$*, that is,* $S' =$ la$\boxed{\text{b}}$a$\boxed{\text{r}}\boxed{\text{a}}\boxed{\text{l}}al\boxed{\text{a}}r\boxed{\text{d}}a\boxed{\text{\$}}$.

As mentioned at the end of Section 3.9, we can build a run-length grammar of size $O(\gamma' \log(n'/\gamma')) = O(\gamma \log\log n)$ on S' (Christiansen et al. [2020] do this without finding the attractor, which would be NP-hard). This grammar is, in addition, locally balanced, that is, every nonterminal whose parse tree node has l leaves is of height $O(\log l)$.

Assume we want to extract a substring $S'[i' \mathinner{.\,.} i' + 2^k]$ for some fixed i' and $0 \le k \le \log\log n$. We may store the lowest common ancestor u in the parse tree of the i'th and $(i' + 2^k)$th leaves. Let v and w be the children of u that are ancestors of those two leaves, respectively. Let j_v be the rank of the rightmost leaf of v and j_w that of the leftmost leaf of w. Thus, we have $i' \le j_v < j_w \le i' + 2^k$. This implies that, if v' and w' are, respectively, the lowest common ancestors of the leaves with rank i' and j_v, and with rank j_w and $i' + 2^k$, then v' descends from v and w' descends from w, and both v' and w' are of height $O(\log 2^k) = O(k)$ because the grammar is locally balanced. We can then extract $S[i' \mathinner{.\,.} i' + 2^k]$ by extracting the $j_v - i' + 1$ rightmost leaves of v' by a simple traversal from the right, in time $O(j_v - i' + k)$, then the whole children of u that are between v and w, in time $O(2^k)$, and finally the $i' + 2^k - j_w + 1$ leftmost leaves of w', in time $O(i' + 2^k - j_w + k)$, with a simple traversal from the left. All this adds up to $O(2^k)$ work. See Figure 13.

We store the desired information on the nodes v' and w' for every position $S'[i']$ to which a phrase beginning $S[i]$ is mapped. Since this is stored for every value k, we use $O(t \log\log n)$ total space. To extract $S'[i' \mathinner{.\,.} i' + \ell]$ for some $0 \le \ell \le \log n$, we choose $k = \lceil \log \ell \rceil$ and extract the substring in time $O(2^k) = O(\ell)$. The arrangement for phrase suffixes is analogous.

Similarly, we can compute the Karp-Rabin signature of $S'[i' \mathinner{.\,.} i' + \ell]$ by storing the signature for every nonterminal, and combine the signatures of (1) the $O(k) =$

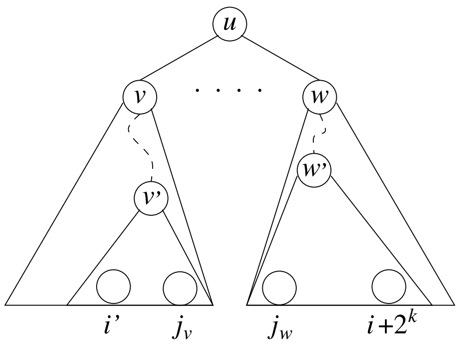

Fig. 13. Illustration of the bookmarking technique.

$O(\log \ell)$ subtrees that cover the area between $S'[i']$ and $S'[j_v]$, (2) the children of u between v and w (if any), and (3) the $O(k) = O(\log \ell)$ subtrees that cover the area between $S'[j_w]$ and $S'[i' + \ell]$ (if any). (If $i' + \ell < j_w$, we only combine the $O(\log \ell)$ subtrees that cover the area between $S'[i']$ and $S[i' + \ell]$.) This can be done in $O(\log \ell)$ time, recall Section 4.1.2.

4.4 Persistent Strings

Various repetitive collections consist of a sequence of versions of a base document, where each version differs from the previous one in a few "edits" (symbol insertions, deletions, or substitutions). Branching in the versions is also possible, so that we have a tree of versions where each version differs from its parent in a few edits. This can be addressed with *persistent data structures* [Driscoll et al. 1989], which allow updates but can be accessed as they were at any point in the past. Partial persistence allows updates only in the current version, and thus can model version sequences; full persistence allows updating any version and enables version trees.

The basic results of Driscoll et al. [1989] allow one to represent a fully-persistent balanced binary search tree of t leaves in $O(t)$ space, and then apply each update persistently by creating only one new path from the root to the affected node. The pointers sprouting from this path lead to the previous version's subtrees, and the root of the new path becomes the root of the new version. In this way, performing s updates take $O(s \log(t + s))$ additional space, and the operations are carried out in time $O(\log(t + s))$. The total space can be reduced to $O(t + s)$ with more careful binary tree implementations [Driscoll et al. 1989; Sarnak and Tarjan 1986].

If we use such a fully-persistent balanced tree to represent the symbols of a base document at its leaves, then we can represent the edit operations as updates to the tree. Maintaining the number of leaves that descend from any node leads us to any desired string position. A base document of length t on which s edit operations are carried out along all its versions is then represented in $O(t + s)$ space, and it supports edits and access to individual symbols in time $O(\log(t + s))$.

Recently, Bille and Gørtz [2020] showed how to improve the access time on persistent strings to $O(\log(t + s)/\log\log(t + s))$, on a model where they start with the empty string (so one can build the first string via t insertions). Further, they can extract a substring of length ℓ in $O(\ell)$ additional time. They use $O(t + s)$ space and show that their time is optimal for any space in $O((t+s)\,\mathrm{polylog}\,(t+s))$.

This result, which makes use of the particular structure of the repetitiveness, is not directly obtained with the general techniques we have covered. For example, it is easy to see that, processing this versioned collection from the beginning, we obtain $z_{no} = O(t/\log_\sigma t + s)$ Lempel-Ziv phrases, even without overlaps. It is unknown, however, if one can provide direct access within $O(z_{no})$ space in general. We can provide logarithmic access time by using space that is in general larger than $O(t+s)$. For example, we can use the block trees of Section 4.2, which would use space $O(\delta\log(n/\delta)) \subseteq O((t/\log_\sigma t + s)\log(s\log_\sigma t))$, because $n \le (t+s)s$ and $\delta \le z_{no}$. We can also use grammars. One of size $O(t/\log_\sigma t + s\log(t+s))$ can be obtained as follows [Navarro 2019]: First, we build a nonterminal A_0 expanding to the original text (the grammar is of size $O(t/\log_\sigma t)$ up to here); then, for each new version i that applies s_i edits to version j, produce the nonterminal A_i expanding to version i by copying the parse tree of A_j and modifying the paths to those edits. The grammar trees of all those nonterminals A_i add up to $O(s\log(t+s))$ nodes.

5. OPEN QUESTIONS

Various questions about the relations between the repetitiveness measures in Figure 9 remain open. In this line, perhaps the most important open question from Part I of this survey is the following:

What is the smallest reachable and computable measure of repetitiveness?

Like Shannon's entropy is a lower bound when we decide to exploit only frequencies, bidirectional macro schemes are a lower bound when we decide to exploit only string copies. Still, there may be some useful repetitiveness measure between b and the bottom line of the (uncomputable) Kolmogorov complexity. A suitable candidate could be γ, though we conjecture that it is not reachable.

Other equally fascinating questions can be asked about accessing and indexing:

What is the smallest reachable measure under which
we can access and/or index the strings efficiently?

For now, $O(g_{rl})$ is the best known limit for efficient access; it is unknown if one can access $S[i]$ efficiently within $O(z_{no})$ or $O(r)$ space. Indexing can also be supported within $O(g_{rl})$ space, but also in $O(r)$; we do not know if this is possible within $O(z_{no})$ or $O(v)$ space. Part II of this survey [Navarro 2020] focuses on how indexing is built on top of direct access.

Acknowledgements

We thank Travis Gagie, Nicola Prezza, and the reviewers for useful comments.

REFERENCES

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

Belazzougui, D., Cáceres, M., Gagie, T., Gawrychowski, P., Kärkkäinen, J., Navarro, G., Ordóñez, A., Puglisi, S. J., and Tabei, Y. 2021. Block trees. *Journal of Computer and System Sciences 117*, 1–22.

Belazzougui, D. and Cunial, F. 2017. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 7:1–7:13.

Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., and Raffinot, M. 2015. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 26–39.

Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., and Raffinot, M. 2017. Flexible indexing of repetitive collections. In *Proc. 13th Conference on Computability in Europe (CiE)*. 162–174.

Belazzougui, D., Gagie, T., Gawrychowski, P., Kärkkäinen, J., Ordóñez, A., Puglisi, S. J., and Tabei, Y. 2015. Queries on LZ-bounded encodings. In *Proc. 25th Data Compression Conference (DCC)*. 83–92.

Belazzougui, D. and Navarro, G. 2015. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms 11,* 4, article 31.

Belazzougui, D., Puglisi, S. J., and Tabei, Y. 2015. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*. 142–154.

Bell, T. C., Cleary, J., and Witten, I. H. 1990. *Text Compression*. Prentice Hall.

Bender, M. and Farach-Colton, M. 2004. The level ancestor problem simplified. *Theoretical Computer Science 321,* 1, 5–12.

Bentley, J., Gibney, D., and Thankachan, S. V. 2019. On the complexity of BWT-runs minimization via alphabet reordering. *CoRR 1911.03035*.

Bille, P., Gagie, T., Gørtz, I. L., and Prezza, N. 2018. A separation between RLSLPs and LZ77. *Journal of Discrete Algorithms 50*, 36–39.

Bille, P., Gørtz, I. L., Cording, P. H., Sach, B., Vildhøj, H. W., and Vind, S. 2017. Fingerprints in compressed strings. *Journal of Computer and System Sciences 86*, 171–180.

Bille, P. and Gørtz, I. L. 2020. Random access in persistent strings. *CoRR 2006.15575*.

Bille, P., Landau, G. M., Raman, R., Sadakane, K., Rao, S. S., and Weimann, O. 2015. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing 44,* 3, 513–539.

Blumer, A., Blumer, J., Haussler, D., McConnell, R. M., and Ehrenfeucht, A. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM 34,* 3, 578–595.

Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Rasala, A., Sahai, A., and Shelat, A. 2002. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*. 792–801.

Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., and Shelat, A. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory 51,* 7, 2554–2576.

Christiansen, A. R., Ettienne, M. B., Kociumaka, T., Navarro, G., and Prezza, N. 2020. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms 17,* 1, article 8.

Claude, F., Fariña, A., Martínez-Prieto, M., and Navarro, G. 2016. Universal indexes for highly repetitive document collections. *Information Systems 61*, 1–23.

Cover, T. and Thomas, J. 2006. *Elements of Information Theory*, 2nd ed. Wiley.

Crochemore, M., Iliopoulos, C. S., Kubica, M., Rytter, W., and Waleń, T. 2012. Efficient algorithms for three variants of the LPF table. *Journal of Discrete Algorithms 11*, 51–61.

Dinklage, P., Fischer, J., Köppl, D., Löbel, M., and Sadakane, K. 2017. Compression with the tudocomp framework. In *Proc. 16th International Symposium on Experimental Algorithms (SEA)*.

Driscoll, J., Sarnak, N., Sleator, D., and Tarjan, R. E. 1989. Making data structures persistent. *Journal of Computer and System Sciences 38*, 86–124.

Elsayed, T. and Oard, D. W. 2006. Modeling identity in archival collections of email: A preliminary study. In *Proc. 3rd Conference on Email and Anti-Spam (CEAS)*.

Farach, M. and Thorup, M. 1998. String matching in Lempel-Ziv compressed strings. *Algorithmica 20,* 4, 388–404.

Ferragina, P., Giancarlo, R., Manzini, G., and Sciortino, M. 2005. Boosting textual compression in optimal linear time. *Journal of the ACM 52,* 4, 688–713.

Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. *Journal of the ACM 52,* 4, 552–581.

Fischer, J., I, T., Köppl, D., and Sadakane, K. 2018. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica 80,* 7, 2048–2081.

Fritz, M. H.-Y., Leinonen, R., Cochrane, G., and Birney, E. 2011. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 734–740.

Gagie, T. 2006. Large alphabets and incompressibility. *Information Processing Letters 99,* 6, 246–251.

Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., and Puglisi, S. J. 2012. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*. 240–251.

Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., and Puglisi, S. J. 2014. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Symposium on Theoretical Informatics (LATIN)*. 731–742.

Gagie, T., Navarro, G., and Prezza, N. 2018. On the approximation ratio of Lempel-Ziv parsing. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*. 490–503.

Gagie, T., Navarro, G., and Prezza, N. 2020. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM 67,* 1, article 2.

Gallant, J. K. 1982. String compression algorithms. Ph.D. thesis, Princeton University.

Ganardi, M., Jeż, A., and Lohrey, M. 2019. Balancing straight-line programs. In *Proc. 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 1169–1183.

Gasieniec, L., Karpinski, M., Plandowski, W., and Rytter, W. 1996. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT)*. 392–403.

Gasieniec, L., Kolpakov, R., Potapov, I., and Sant, P. 2005. Real-time traversal in grammar-based compressed files. In *Proc. 15th Data Compression Conference (DCC)*. 458–458.

Gawrychowski, P. 2011. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*. 421–432.

Giuliani, S., Inenaga, S., Lipták, Z., Prezza, N., Sciortino, M., and Toffanello, A. 2020. Novel results on the number of runs of the Burrows-Wheeler-Transform. *CoRR 2008.08506*.

Gog, S., Kärkkäinen, J., Kempa, D., Petri, M., and Puglisi, S. J. 2019. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica 81,* 4, 1370–1391.

Henzinger, M. R. 2006. Finding near-duplicate web pages: A large-scale evaluation of algorithms. In *Proc. 29th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 284–291.

Hucke, D., Lohrey, M., and Reh, C. P. 2016. The smallest grammar problem revisited. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE)*. 35–49.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*. 549–554.

Jeż, A. 2015. Approximation of grammar-based compression via recompression. *Theoretical Computer Science 592*, 115–134.

Jeż, A. 2016. A really simple approximation of smallest grammar. *Theoretical Computer Science 616*, 141–150.

Kapser, C. and Godfrey, M. W. 2005. Improved tool support for the investigation of duplication in software. In *Proc. 21st IEEE International Conference on Software Maintenance (ICSM)*. 305–314.

Kärkkäinen, J., Kempa, D., and Puglisi, S. J. 2012. Slashing the time for BWT inversion. In *Proc. 22nd Data Compression Conference (DCC)*. 99–108.

Kärkkäinen, J., Kempa, D., and Puglisi, S. J. 2016. Lazy Lempel-Ziv factorization algorithms. *ACM Journal of Experimental Algorithmics 21,* 1, 2.4:1–2.4:19.

Kärkkäinen, J. and Puglisi, S. J. 2010. Medium-space algorithms for inverse BWT. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*. 451–462.

Kärkkäinen, J., Sanders, P., and Burkhardt, S. 2006. Linear work suffix array construction. *Journal of the ACM 53,* 6, 918–936.

Karp, R. M. and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 2*, 249–260.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

Kempa, D. and Kociumaka, T. 2020. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. 61st IEEE Symposium on Foundations of Computer Science (FOCS)*. 1002–1013.

Kempa, D. and Prezza, N. 2018. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*. 827–840.

Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A., and Arikawa, S. 2003. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science 298,* 1, 253–272.

Kieffer, J. C. and Yang, E.-H. 2000. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory 46,* 3, 737–754.

Kim, D. K., Sim, J. S., Park, H., and Park, K. 2005. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3,* 2-4, 126–142.

Ko, P. and Aluru, S. 2005. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3,* 2-4, 143–156.

Kociumaka, T., Navarro, G., and Prezza, N. 2020. Towards a definitive measure of repetitiveness. In *Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN)*. 207–219.

Kociumaka, T., Navarro, G., and Prezza, N. 2021. Towards a definitive compressibility measure for repetitive sequences. *CoRR 1910.02151.*

Kolmogorov, A. N. 1965. Three approaches to the quantitative definition of information. *Problems on Information Transmission 1,* 1, 1–7.

Kosaraju, R. and Manzini, G. 2000. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing 29,* 3, 893–911.

Kreft, S. and Navarro, G. 2013. On compressing and indexing repetitive sequences. *Theoretical Computer Science 483*, 115–133.

Kutsukake, K., Matsumoto, T., Nakashima, Y., Inenaga, S., Bannai, H., and Takeda, M. 2020. On repetitiveness measures of Thue-Morse words. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*. 213–220.

Larsson, J. and Moffat, A. 2000. Off-line dictionary-based compression. *Proceedings of the IEEE 88,* 11, 1722–1732.

Lempel, A. and Ziv, J. 1976. On the complexity of finite sequences. *IEEE Transactions on Information Theory 22,* 1, 75–81.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

Mäkinen, V., Navarro, G., Sirén, J., and Välimäki, N. 2010. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology 17,* 3, 281–308.

Manber, U. and Myers, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

Mantaci, S., Restivo, A., Romana, G., Rosone, G., and Sciortino, M. 2021. A combinatorial view on string attractors. *Theoretical Computer Science 850*, 236–248.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

McCreight, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23,* 2, 262–272.

Navarro, G. 2016. *Compact Data Structures – A practical approach.* Cambridge University Press.

Navarro, G. 2019. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science 777*, 58–72.

Navarro, G. 2020. Indexing highly repetitive string collections, Part II: Compressed indexes. *ACM Computing Surveys*. To appear.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

Navarro, G. and Prezza, N. 2019. Universal compressed text indexing. *Theoretical Computer Science 762*, 41–50.

Navarro, G., Prezza, N., and Ochoa, C. 2021. On the approximation ratio of ordered parsings. *IEEE Transactions on Information Theory 67,* 2, 1008–1026.

Navarro, G. and Rojas-Ledesma, J. 2020. Predecessor search. *ACM Computing Surveys 53,* 5, article 105.

Nevill-Manning, C., Witten, I., and Maulsby, D. 1994. Compression by induction of hierarchical grammars. In *Proc. 4th Data Compression Conference (DCC)*. 244–253.

Nishimoto, T., I, T., Inenaga, S., Bannai, H., and Takeda, M. 2016. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 72:1–72:15.

Nishimoto, T. and Tabei, Y. 2019. LZRR: LZ77 parsing with right reference. In *Proc. 29th Data Compression Conference (DCC)*. 211–220.

Ochoa, C. and Navarro, G. 2019. Repair and all irreducible grammars are upper bounded by high-order empirical entropy. *IEEE Transactions on Information Theory 65,* 5, 3160–3164.

Prezza, N. 2016. Compressed computation for text indexing. Ph.D. thesis, University of Udine.

Przeworski, M., Hudson, R. R., and Rienzo, A. D. 2000. Adjusting the focus on human variation. *Trends in Genetics 16,* 7, 296–302.

Raskhodnikova, S., Ron, D., Rubinfeld, R., and Smith, A. D. 2013. Sublinear algorithms for approximating string compressibility. *Algorithmica 65,* 3, 685–709.

Rodeh, M., Pratt, V. R., and Even, S. 1981. Linear algorithm for data compression via string matching. *Journal of the ACM 28,* 1, 16–24.

Rubin, F. 1976. Experiments in text file compression. *Communications of the ACM 19,* 11, 617–623.

Russo, L. M. S., Correia, A., Navarro, G., and Francisco, A. P. 2020. Approximating optimal bidirectional macro schemes. In *Proc. 30th Data Compression Conference (DCC)*. 153–162.

Rytter, W. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science 302,* 1-3, 211–222.

Sakamoto, H. 2005. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms 3,* 2–4, 416–430.

Sarnak, N. and Tarjan, R. E. 1986. Planar point location using persistent search trees. *Communications of the ACM 29,* 7, 669–679.

Shannon, C. E. 1948. A mathematical theory of communication. *Bell Systems Technical Journal 27*, 398–403.

Sleator, D. D. and Tarjan, R. E. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences 26,* 3, 362–391.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Chenxiang, Z., Efron, M. J., Iyer, R., Sinha, S., and Robinson, G. E. 2015. Big data: Astronomical or genomical? *PLoS Biology 17,* 7, e1002195.

Storer, J. A. and Szymanski, T. G. 1982. Data compression via textual substitution. *Journal of the ACM 29,* 4, 928–951.

Tao, K., Abel, F., Hauff, C., Houben, G., and Gadiraju, U. 2013. Groundhog day: Near-duplicate detection on Twitter. In *Proc. 22nd International World Wide Web Conference (WWW)*. 1273–1284.

Verbin, E. and Yu, W. 2013. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 247–258.

Weiner, P. 1973. Linear pattern matching algorithms. In *Proc. 14th IEEE Symposium on Switching and Automata Theory (FOCS)*. 1–11.

Witten, I. H., Neal, R. M., and Cleary, J. G. 1987. Arithmetic coding for data compression. *Communications of the ACM 30*, 520–540.

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23,* 3, 337–343.