IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 October 2022, accepted 8 November 2022, date of publication 10 November 2022, date of current version 16 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3221520

RESEARCH ARTICLE

# Compressing and Querying Integer Dictionaries Under Linearities and Repetitions

**PAOLO FERRAGINA, GIOVANNI MANZINI, AND GIORGIO VINCIGUERRA**

Department of Computer Science, University of Pisa, 56127 Pisa, Italy

Corresponding author: Giorgio Vinciguerra (giorgio.vinciguerra@di.unipi.it)

This work was supported in part by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale (PRIN)", Grant no. 2017WR7SHH "Multicriteria Data Structures and Algorithms".

**ABSTRACT** We revisit the fundamental problem of compressing an integer dictionary that supports efficient rank and select operations by exploiting simultaneously two kinds of regularities arising in real data: *repetitiveness* and *approximate linearity*. We attack this problem by proposing two novel compressed indexing approaches that extend the classic Lempel-Ziv compression scheme and the more recent block tree data structure with new algorithms and data structures that allow them to also capture regularities in terms of the approximate linearity in the data. Finally, we corroborate these theoretical results with a wide set of experiments on real and synthetic datasets, which allow us to show that our approaches achieve new interesting space-time trade-offs that characterise them as more robust in most practical scenarios compared to the known data structures that exploit only one of the two regularities.

**INDEX TERMS** Compressed data structures, data compression, entropy.

## I. INTRODUCTION

We focus on the fundamental problem of representing an ordered dictionary $A$ of $n$ distinct elements drawn from the integer universe $[u] = \{0, \ldots, u-1\}$ while supporting the operation $\mathsf{rank}(x)$, which returns the number of elements in $A$ that are smaller than or equal to $x$; and $\mathsf{select}(i)$, which returns the $i$th smallest element in $A$. Another way of looking at these operations is via the characteristic bitvector of $A$, i.e. a bitvector $\mathtt{bv}(A)$ of length $u$ such that $\mathtt{bv}(A)[i] = \mathtt{1}$ if and only if $i \in A$. Here, $\mathsf{rank}(x)$ counts the number of $\mathtt{1}$s up to position $x$, and $\mathsf{select}(i)$ finds the position of the $i$th $\mathtt{1}$, as depicted in Figure 1.[1]

Rank/select dictionaries are at the heart of virtually any compact data structure [1], such as text indexes [2], [3], [4], [5], [6], [7], succinct trees and graphs [8], [9], hash tables [10], permutations [11], etc. Unsurprisingly, the literature is abundant in solutions that offer compressed space and efficient support for rank/select operations, e.g. [9], [12], [13], [14], [15], [16], [17], [18]. Yet, the problem of designing theoretically and practically efficient rank/select data structures is anything but closed. The reason is threefold. First, there is an ever-growing list of applications of compact data structures (in bioinformatics [19], [20], information retrieval [21], and databases [22], just to mention a few) each having different characteristics and requirements on the use of computational resources, such as time, space, and energy consumption. Second, the hardware is evolving [23],

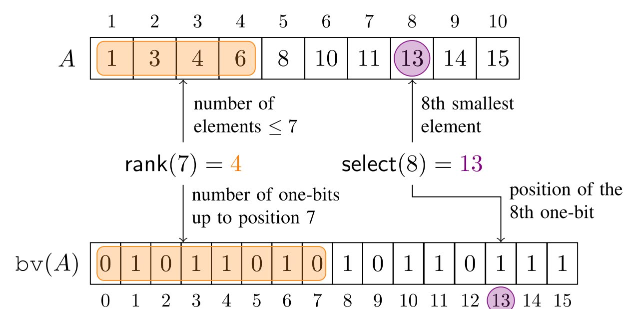


**FIGURE 1. The rank and select operations on a dictionary *A* of 10 elements over the universe [16], and on the corresponding characteristic bitvector bv(*A*).**

The associate editor coordinating the review of this manuscript and approving it for publication was Gustavo Olague.

[1] For a sequence $T[1, n]$ of characters, one can define instead: $\mathsf{rank}_a(i)$, which returns the number of occurrences of the character $a$ in $T[1, i]$; $\mathsf{select}_a(j)$, which returns the position of the $j$th occurrence of $a$ in $T$; and $\mathsf{access}(k)$, which returns the character $T[k]$. These new operations can be implemented by using rank and select on bitvectors as building blocks [1, §6].

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

sometimes requiring new data structuring techniques to fully exploit larger CPU registers, new instructions, parallelism, and next-generation memories, such as persistent memory. Third, data may present different kinds of regularities, which require different techniques to exploit them in novel and better performing rank/select data structures.

Among the latest and very promising regularities to be exploited, there is a geometric concept of *approximate linearity* [24], [25]. Let us regard $A$ as a sorted array $A = A[1, n]$ and let $A[i, j]$ denote the subarray $A[i], A[i+1], \dots, A[j]$. The key idea is to first map each element $A[i]$ to the point $(i, A[i])$ in the Cartesian plane, for $i = 1, 2, \dots, n$. This way, any function $f : [1, n] \to [u]$ that passes through all the points in this plane can be thought of as an encoding of $A$ because we can answer $\mathsf{select}(i) = A[i]$ by means of $f(i)$. From the ordering of $A$ and the simple retrieval of $A[i]$, the rank operation could be easily solved via a binary search. Now, the challenge is to find a representation of $f$ that is both fast to be computed and compressed in space and, also, suitable to support efficient rank, hence not passing through a binary search. To this end, the authors of [25] proposed to implement $f$ via a piecewise linear model whose error, measured as the vertical distance between the prediction and the actual value of $A$, is bounded by a given integer parameter $\varepsilon$.

*Definition 1: A piecewise linear $\varepsilon$-approximation for the integer array $A[1, n]$ is a partition of $A$ into subarrays of variable length, such that each subarray $A[i, j]$ of the partition is covered by a segment, represented by a pair $\langle \alpha, \beta \rangle$ of numbers, such that $|(\alpha k + \beta) - A[k]| \le \varepsilon$ for each $k \in [i, j]$.*

Among all possible piecewise linear $\varepsilon$-approximations, the authors of [25] aimed for the most succinct one, namely the one with the least number of segments. This is a classical computational geometry problem that admits an $\mathcal{O}(n)$-time solution [26]. The structure introduced by [25], named LA-vector, uses this succinct piecewise linear $\varepsilon$-approximation as a lossy representation of $A$, and it mends the information loss by storing the vertical errors into an array $C$ of $\lceil \log(2\varepsilon + 1) \rceil$-bit integers, called corrections (all logarithms are to the base two). To answer $\mathsf{select}(i)$, the LA-vector uses a constant-time rank data structure built on a bit-vector of size $n$ to find the $\langle \alpha, \beta \rangle$ corresponding to the segment covering $i$, and it returns the value $\lfloor \alpha i + \beta \rfloor + C[i]$. The $\mathsf{rank}(x)$ operation is implemented via an empowered binary search that exploits the information encoded in the piecewise linear $\varepsilon$-approximation to be faster [25].

In practical implementations, the LA-vector allocates $c \ge 0$ bits for each correction and sets $\varepsilon = \max(0, 2^{c-1} - 1)$. Its space usage in bits consists roughly of a term $\mathcal{O}(nc)$ accounting for the corrections array $C$, and a term $\mathcal{O}(m(\log u + \log n))$ that grows with the number of segments $m$ in the piecewise linear $\varepsilon$-approximation.[2] Despite the apparent simplicity of the piecewise linear representation,

[2]In Section III, we show that the $\mathcal{O}(\log u + \log n)$ term can be reduced to $\mathcal{O}(\log \frac{u}{m} + \log \frac{n}{m})$ bits.

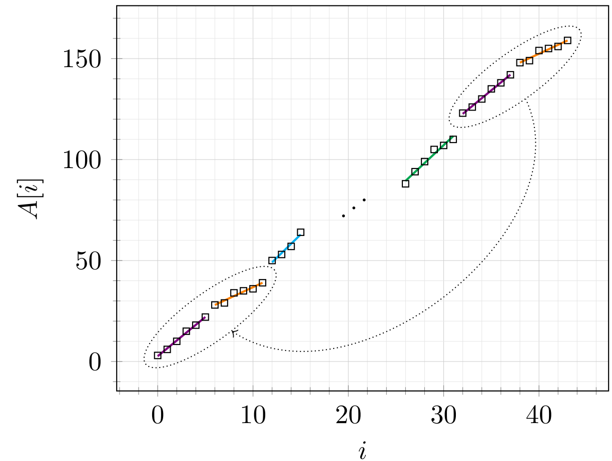


**FIGURE 2. The points in the top-right circle follows the same "pattern" (i.e. the same distance between consecutive points) of the ones in the bottom-left circle. A piecewise linear $\varepsilon$-approximation for the top-right set can be obtained by shifting the segments for the bottom-left set.**

the experiments in [25] have shown that the LA-vector offers the fastest select and competitive rank performance with respect to several state-of-the-art data structures implemented in the `sdsl` library [27]. Notably, the value $m$ has been proposed as a compressibility measure that accounts for the approximate linearity of $A$'s elements, and it has been shown that $m = \mathcal{O}(n/\varepsilon^2)$ when the gaps between the elements are random variables from a given distribution [28].

Despite their succinctness and power in capturing linear trends, piecewise linear $\varepsilon$-approximations still lack the capacity to find and exploit one fundamental and classical source of compressibility arising in real data: *repetitiveness* [29]. Although the input consists of an array $A$ of strictly increasing integers, there can be significant repetitiveness in the differences between consecutive elements. Consider the gap-string $S[1, n]$ defined as $S[i] = A[i] - A[i-1]$, with $A[0] = 0$, and suppose that the substring $S[i, j]$ has been encountered earlier at $S[i', i' + j - i]$ (we write $S[i, j] \equiv S[i', i' + j - i]$). Then, instead of finding a new set of segments $\varepsilon$-approximating the subarray $A[i, j]$, we could use the segments $\varepsilon$-approximating the subarray $A[i', j']$ properly shifted. Note that, even if $A[i', j']$ is covered by many segments, the same shift will transform all of them into an approximation for $A[i, j]$ (see example in Figure 2). Therefore, in this case, we could store only the *shift* and the *reference* to the segments of $A[i', j']$.

Unfortunately, the LA-vector is unable to take advantage of such regularities. And, in the extreme case where $A$ consists of the concatenation of a small subarray $A'$ shifted by some amounts $\Delta_i$s for $k$ times, that is $A = A', A' + \Delta_1, A' + \Delta_2, \dots, A' + \Delta_{k-1}$, the overall cost of representing $A$ with the LA-vector will be roughly $k$ times the cost of representing $A'$. On the other hand, take an order-$h$ De Bruijn binary sequence $B[1, 2^h]$ and define $A[i] = 2i + B[i]$. Then, the line with slope 2 and intercept 0 is a linear approximation of the entire array $A$ with $\varepsilon = 1$. Conversely, for the gap-string $S[i] = A[i] - A[i-1] = 2 + B[i] - B[i-1]$ we would not find repetitions longer than $h-1$. More pathological cases for the gap-string $S$, which

are nonetheless well compressible by LA-vector because of the approximate linearity of $A$, can be built by considering integers in $A$ whose mapping into the Cartesian plane gives points that distribute randomly around a line with a positive slope. As an example, fix an integer slope $\alpha$ and generate values $A[i] = i\,\alpha + \eta_i$, where $i = 1, \dots, n$ and $\eta_i$ is an integer chosen uniformly at random in a range $[-\varepsilon, \varepsilon]$ for every $i$. It is clear that the segment $\langle \alpha, 0 \rangle$ is a linear $\varepsilon$-approximation for $A$ which, however, will not show much long repeated substrings in the corresponding gap-string $S$ because of the random $\eta_i$s.

Other than the gap string $S$, another common approach in the literature to design a succinct dictionary (see e.g. [30]) is to compress the characteristic bitvector $\mathtt{bv}(A)$ (see again Figure 1). To compare these two approaches, we consider the $k$th order empirical entropy $H_k$ (see the Appendix and [1, §2.4] for the definition and significance of this measure), and we prove that for any dictionary $A$ it is $nH_k(S) \leq uH_k(\mathtt{bv}(A))$. This result provides a firm theoretical ground for the choice of representing $A$ using the gap-string $S$ rather than the characteristic bitvector $\mathtt{bv}(A)$. Since this result is of independent interest, and to not interrupt the "algorithmic flow" of the paper, its statement and proof are given in the Appendix.

The goal of this paper is therefore to design compressed indexing techniques that are able to exploit both the presence of repetitions in the gap-string $S$ and the presence of subarrays in $A$ which can be linearly $\varepsilon$-approximated well, while still supporting efficient rank/select primitives on $A$.

Our orchestration of repetitions and approximate linearities goes through the proper modification of two known compression methods so that they can take advantage of approximate linearity too. The first method is the Lempel-Ziv (LZ) parsing [31], [32], [33], [34], which is one of the best-known approaches to exploit repetitiveness [29]. The second method is the block tree [35], which is a recently proposed query-efficient alternative to LZ-parsing and grammar-based representations [36] suitable for highly repetitive inputs.

Technically speaking, our first contribution is a novel parsing scheme, the $\mathrm{LZ}_\varepsilon^\rho$ parsing, whose phrases are a combination of a backward copy and a linear $\varepsilon$-approximation, i.e., a segment and the corresponding correction values. We show that this solution supports $\mathsf{rank}$ and $\mathsf{select}$ in polylogarithmic time and has space bounds that show the sensitivity to both repetitiveness and approximate linearity. In particular, we bound the former in terms of the $k$th order empirical entropy (as it occurs for the known LZ-parsing methods, cited above) and the latter in terms of the efficient encoding of linear $\varepsilon$-approximations of $A$'s subarrays (as it occurs for the LA-vector), thus obtaining asymptotically the best of both worlds in the space bounds.

Our second contribution is the block-$\varepsilon$ tree, an orchestration of block trees [35], [37] and linear $\varepsilon$-approximations. Our main idea is to build the block tree over the gap-string $S$ and to prune the subtrees whose corresponding subarrays can be covered more succinctly by means of a linear $\varepsilon$-approximation in place of a block (sub)tree. Let us define the $\delta$ repetitiveness measure on $S$ as $\delta = \max\{d_k(S)/k \mid 1 \leq k \leq n\}$, where $d_k(S)$ is the number of distinct length-$k$ substrings of $S$ [29], [37]. We show that this solution supports $\mathsf{rank}$ in $\mathcal{O}(\log\log\frac{u}{\delta} + \log\frac{n}{\delta} + \log\varepsilon)$ time and $\mathsf{select}$ in $\mathcal{O}(\log\frac{n}{\delta})$ time using $\mathcal{O}(\delta\log\frac{n}{\delta}\log n)$ bits of space in the worst case. For comparison, a block tree built on $\mathtt{bv}(A)$ supports $\mathsf{rank}$ and $\mathsf{select}$ in $\mathcal{O}(\log\frac{u}{\delta'})$ time using $\mathcal{O}(\delta'\log\frac{u}{\delta'}\log u)$ bits of space, where $\delta'$ is the repetitiveness measure computed on $\mathtt{bv}(A)$. Unfortunately, the time and space bounds achieved by the block tree and by our block-$\varepsilon$ tree are not directly comparable due to the use of $\delta'$ instead of $\delta$.

Our last contribution is an experimental evaluation of ours and known approaches. On standard datasets (containing no evident repetitive or linearity trends), we show that there is no clear winner in space between the two known representative approaches [25], [35], namely block tree and LA-vector. In this scenario, our block-$\varepsilon$ tree achieves the best space or the second-best space in the majority of cases due to its effectiveness in exploiting both regularities. As far as the query time is concerned, the LA-vector obtains the fastest performance, followed by our block-$\varepsilon$ tree which generally achieves better performance than the block tree. Our $\mathrm{LZ}_\varepsilon$ parsing (a space-efficient configuration of $\mathrm{LZ}_\varepsilon^\rho$) on these standard datasets was, unfortunately, dominated by some other data structure in time and in space. Motivated by these results, to shed light on scenarios in which repetitions and linearities are more evident, we also consider synthetic datasets for which we show that the space of the LA-vector does not improve with repetitions, that the space of the block tree does not improve with approximate linearities, and that both our block-$\varepsilon$ tree and $\mathrm{LZ}_\varepsilon$ achieve an improved space occupancy, being able to successfully capture both forms of compressibility studied in this article.

In the Conclusions, we comment on several research directions that naturally arise from the novel approaches described in this paper. In particular, we highlight here that, although we consider in this paper only linear approximations, our techniques can be extended to other data approximation functions, such as polynomials and rational functions. Furthermore, they can be adapted to the simpler problem of compressing non-monotonic sequences while supporting random-access queries to their values (i.e. only $\mathsf{select}$), which is frequent e.g. in time-series scenarios.

As a final remark, we note that a preliminary version of this work appeared in [38]. In addition to minor improvements in the presentation, the present contribution contains the following new material: the already mentioned gap vs binary entropy inequality given in the Appendix, an improved presentation of the $\mathrm{LZ}_\varepsilon^\rho$ parsing, new improved space bounds for the $\mathrm{LZ}_\varepsilon^\rho$ parsing, the description of $\mathsf{rank}$ and $\mathsf{select}$ queries in the block-$\varepsilon$ tree, a discussion of the algorithm-engineering tricks used in our implementations, the implementation and experimentation of the $\mathrm{LZ}_\varepsilon$ parsing, and a greatly improved experimental section with new

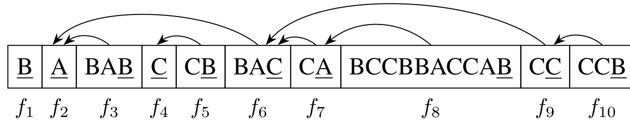


**FIGURE 3.** LZ-End parse of $T$ = BABABCCBBACCABCCBBACCABCCCCB into ten phrases. An arrow from a phrase $f_q$ to $f_r$ indicates that $f_r$ is the last phrase in the source of $f_q$, and the appended character is underlined.

datasets providing new insights on the efficacy of the proposed approaches.

## II. TOOLS

We use the *Elias-Fano* [39], [40] representation for compressing and randomly-accessing monotone integer sequences [1, §4.4].

*Lemma 1 (Elias-Fano encoding): We can store a sequence of n increasing positive integers over a universe of size u in* $n\lceil\log\frac{u}{n}\rceil + 2n + o(n) = n\log\frac{u}{n} + \mathcal{O}(n)$ *bits and access any integer of the sequence in* $\mathcal{O}(1)$ *time.*

Henceforth, we always assume that a piecewise linear $\varepsilon$-approximation for an input array $A$ is the most succinct one in terms of the number of segments, or equivalently, that we always maximise the length $\ell$ of the subarray $A[i, i+\ell-1]$ covered by a segment starting at $i$. This is possible thanks to the algorithm of O'Rourke [26], which computes in optimal $\mathcal{O}(n)$ time the piecewise linear $\varepsilon$-approximation with the smallest number of segments for the set of points $\{(i, A[i]) \mid i = 1, \ldots, n\}$.

Another key tool that we use is *LZ-End* of Kreft and Navarro [34]. Formally, the LZ-End parsing of a text $T[1, n]$ is a sequence $f_1, f_2, \ldots, f_z$ of phrases, such that $T = f_1 f_2 \cdots f_z$, built as follows. If $T[1, i]$ has been parsed as $f_1 f_2 \cdots f_{q-1}$, the next phrase $f_q$ is obtained by finding the longest prefix of $T[i+1, n]$ that appears also in $T[1, i]$ *ending* at a phrase boundary, i.e. the longest prefix of $T[i+1, n]$ which is a suffix of $f_1 \cdots f_r$ for some $r \leq q-1$. If $T[i+1, j]$ is the prefix with the above property, the next phrase is $f_q = T[i+1, j+1]$ (notice the character $T[j+1]$ is appended to the longest copied prefix). The occurrence in $T[1, i]$ of the prefix $T[i+1, j]$ is called the *source* of the phrase $f_q$. Figure 3 shows an example of LZ-End parsing.

Although LZ-End is less powerful than the classic LZ77 parsing, because this latter allows the *end* of a source to be anywhere in $T[1, i]$, it compresses any text $T$ up to its $k$th order entropy, and it allows extracting any length-$\ell$ substring of $T$ in $\mathcal{O}(\ell + M)$ time, where $M$ is the length of the longest phrase. Kreft and Navarro also conjectured that the ratio $z_e/z$ between the number of LZ-End phrases and the ones of LZ77 is upper bounded by 2, and examples of strings where this ratio is arbitrarily close to 2 were given both for a large alphabet [34] and for a binary alphabet [41]. Significant progress on this conjecture was recently made by [42], where it is shown that $z_e = \mathcal{O}(z \log^2 \frac{n}{z})$.

With the advent of large datasets containing many repetitions, researchers have observed that the entropy does not always provide a meaningful lower bound to the information content of such datasets [34]. Recently, Navarro [29] has given a complete picture of several alternative measures of information content and has shown that they are all lower bounded by the measure $\delta$, defined as $\max\{d_k(T)/k \mid 1 \leq k \leq n\}$, where $d_k(T)$ is the number of distinct length-$k$ substrings of $T$.

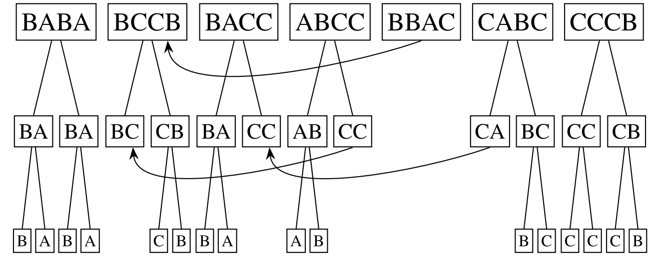


**FIGURE 4.** A block tree on $T$ = BABABCCBBACCABCCBBACCABCCCCB.

In [35] and [37], it is shown how to represent a text $T[1, n]$ in space bounded in terms of $\delta$ while supporting $\mathsf{rank}_a$, $\mathsf{select}_a$, and $\mathsf{access}$ operations (recall their definitions in Footnote 1) via a data structure called the *block tree*. Assume that $n = \delta 2^h$ for some integer $h$. Level zero of the block tree logically divides $T$ into $\delta$ blocks of size $n/\delta$. Blocks at level $\ell$ have size $n/(\delta 2^\ell)$ because they are recursively halved according to the following strategy. At any level, if two blocks $T_q$ and $T_{q+1}$ are consecutive in $T$ and they form the leftmost occurrence in $T$ of their content, then both $T_q$ and $T_{q+1}$ are said to be marked. A marked block is split into two equal-size sub-blocks. An unmarked block $T_r$ is not split further and is encoded by storing a leftward pointer to the marked blocks $T_q, T_{q+1}$ at the same level containing the leftmost occurrence of $T_r$. The last level is formed when the cost of explicitly storing $T_r$ becomes less than that of storing the leftward pointers, and this happens when $h = \mathcal{O}(\log\frac{n/\delta}{\log_\sigma n}) = \mathcal{O}(\log\frac{n}{\delta})$. By storing further data at each block, the block tree supports $\mathsf{rank}_a$, $\mathsf{select}_a$, and $\mathsf{access}$ in $\mathcal{O}(h)$ time, while taking $\mathcal{O}(\sigma\delta\log\frac{n}{\delta}\log n)$ bits of space, where $\sigma$ is the alphabet size. Figure 4 shows a block tree (where we assume $\delta = 7$ for the sake of example) on the same text of Figure 3.

Note that $\delta$ is also related to the number $z_e$ of LZ-End phrases as $z_e = \mathcal{O}(\delta\log^2\frac{n}{\delta})$ [42]. Thus, the block tree and the LZ-end parsing, which are the techniques we use as starting points for the design of our data structures, are both suitable to compress datasets with many repetitions, since their space occupancy is bounded in terms of $\delta$.

## III. TWO NOVEL LZ-PARSINGS: LZ$_\varepsilon$ AND LZ$_\varepsilon^\rho$

Assume that $A$ contains distinct positive integers and consider the gap-string $S[1, n]$ defined as $S[i] = A[i] - A[i-1]$, where $A[0] = 0$. To make the LA-vector repetition aware, we parse $S$ via a strategy that combines linear $\varepsilon$-approximations with the LZ-End parsing. We generalise the phrases of the LZ-End parsing in a way that they are a ''combination'' of a backward copy ending at a phrase boundary (as in the classic LZ-End), computed over the gap-string $S$, plus a segment covering

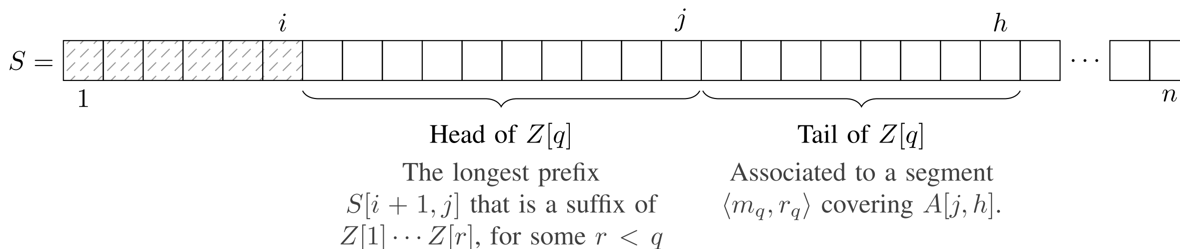


**FIGURE 5. Computation of the next phrase $Z[q]$ in the parsing of the gap-string $S$ of the array $A$, where $S[1, i]$ has already been parsed into $Z[1], \ldots, Z[q-1]$.**

a subarray of $A$ with an error of at most $\varepsilon$ (unlike classic LZ-End, which instead adds a single trailing character). We call this parsing the $LZ_\varepsilon$ parsing of $S$.

Suppose that $LZ_\varepsilon$ has partitioned $S[1, i]$ into $Z[1], Z[2], \ldots, Z[q-1]$. We determine the next phrase $Z[q]$ as follows (see Figure 5):

1) We compute the longest prefix $S[i+1, j]$ of $S[i+1, n]$ that is a suffix of the concatenation $Z[1] \cdots Z[r]$ for some $r \leq q-1$ (i.e. the source must end at a previous phrase boundary).
2) We find the longest subarray $A[j, h]$ that may be $\varepsilon$-approximated linearly, as well as the slope and intercept of such approximation. Note that using the algorithm of [26] the time complexity of this step is $\mathcal{O}(h-j)$, i.e. linear in the length of the processed array.

The new phrase $Z[q]$ is then the substring $S[i+1, j] \cdot S[j+1, h]$. If $h = n$, the parsing is complete. Otherwise, we continue the parsing with $i \leftarrow h+1$. As depicted in Figure 5, we call $S[i+1, j]$ the *head* of $Z[q]$ and $S[j+1, h]$ the *tail* of $Z[q]$. Note that the segment associated with the tail covers also the value $A[j]$ corresponding to the head's last position $S[j]$. When $S[i+1, j]$ is the empty string (e.g. at the beginning of the parsing), the head is empty thus no backward copy is executed, and the segment associated with the tail covers only the tail's positions. In the worst case, the longest subarray we can $\varepsilon$-approximate has length 2, which nonetheless guarantees that $Z[q]$ is nonempty. Experiments in [25] show that the average segment length ranges from 76 when $\varepsilon = 31$ to 1480 when $\varepsilon = 511$.

If the complete parsing consists of $z$ phrases, we store it via:

- An integer vector $\mathsf{PE}[1, z]$ (Phrase Ending position) such that $h = \mathsf{PE}[q]$ is the ending position of phrase $Z[q]$, that is, $Z[q] = S[i+1, h]$, where $i = \mathsf{PE}[q-1]$.
- An integer vector $\mathsf{HE}[1, z]$ (Head Ending position) such that $j = \mathsf{HE}[q]$ is the last position of $Z[q]$'s head. Hence, $Z[q]$'s head is $S[\mathsf{PE}[q-1]+1, \mathsf{HE}[q]]$, and $Z[q]$'s tail is $S[\mathsf{HE}[q]+1, \mathsf{PE}[q]]$.
- An integer vector $\mathsf{HS}[1, z]$ (Head Source) such that $r = \mathsf{HS}[q]$ is the index of the last phrase in $Z[q]$'s source. Hence, $Z[q]$'s head is a suffix of $Z[1] \cdots Z[r]$. If $Z[q]$'s head is empty then $\mathsf{HS}[q] = 0$.
- A vector of pairs $\mathsf{TL}[1, z]$ (Tail Line) such that $\mathsf{TL}[q] = \langle \alpha_q, \beta_q \rangle$ are the parameters of the segment associated with $Z[q]$'s tail.
- A vector of arrays $\mathsf{TC}[1, z]$ (Tail Corrections) such that $\mathsf{TC}[q]$ is an array storing one correction value for each element in the subarray $A[\mathsf{HE}[q], \mathsf{PE}[q]]$ covered by the segment associated with $Z[q]$'s tail (the subarray is $A[\mathsf{HE}[q]+1, \mathsf{PE}[q]]$ in the case $Z[q]$'s head is empty). By construction, such corrections are smaller than $\varepsilon$ in modulus.

Using the values in $\mathsf{TL}$ and $\mathsf{TC}$ we can recover the subarrays $A[j, h]$ corresponding to the phrases' tails. We show that using all the above vectors we can recover the whole array $A$.

*Lemma 2: Let $S[i+1, j]$ denote the head of phrase $Z[q]$, and let $r = \mathsf{HS}[q]$ and $e = \mathsf{PE}[r]$. Then, for $t = i+1, \ldots, j$, it holds*

$$A[t] = A[t-(j-e)] + (A[j] - A[e]), \tag{1}$$

*where $A[j]$ (resp. $A[e]$) can be retrieved in constant time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$ (resp. $\mathsf{TL}[r]$ and $\mathsf{TC}[r]$).*

*Proof:* By construction, $S[i+1, j]$ is identical to a suffix of $Z[1] \cdots Z[r]$. Since such a suffix ends at position $e = \mathsf{PE}[r]$, it holds $S[i+1, j] \equiv S[e-j+i+1, e]$ and

$$\begin{aligned} A[t] &= A[j] - (S[j] + S[j-1] + \cdots + S[t+1]) \\ &= (A[j] - A[e]) + A[e] - (S[e] + S[e-1] + \cdots \\ &\quad + S[t+1-(j-e)]) \\ &= (A[j] - A[e]) + A[t-(j-e)]. \end{aligned}$$

For the second part of the lemma, we notice that $A[j]$ is the first value covered by the segment associated with $Z[q]$'s tail, while $A[e]$ is the last value covered by the segment associated with $Z[r]$'s tail. □

### A. SUPPORTING SELECT QUERIES

Using Lemma 2 above, we can also show that given a position $t \in [1, n]$ we can retrieve $A[t]$ and thus implement $\mathsf{select}(t)$. The main idea is to use a binary search on $\mathsf{PE}$ to retrieve the phrase $Z[q]$ containing $t$. Then, if $t \geq \mathsf{HE}[q]$, we get $A[t]$ from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$; otherwise, we use Lemma 2 and get $A[t]$ by retrieving $A[t-(j-e)]$ using recursion, as formalised in Algorithm 1.

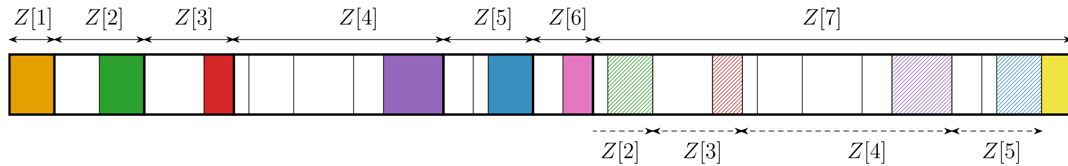


**FIGURE 6.** The $LZ_\varepsilon$ parsing with the definition of meta-characters. Cells represent meta-characters, and the coloured cells are also tails. $Z[7]$'s head consists of a copy of a substring that starts inside $Z[2]$ and ends at the end of $Z[5]$ (we show this using diagonal patterns in $Z[7]$'s head with the same colours of the tails in $Z[2]\cdots Z[5]$). Meta-characters in $Z[7]$'s head are defined from the meta-characters in the copy. Note that $Z[7]$'s first meta-character is a suffix of $Z[2]$'s first meta-character.

**Algorithm 1** Recursive select Procedure

```
 1: procedure Select(t)
 2:     q ← smallest i such that PE[i] ≥ t, found via a binary search on PE
 3:     return Select-Aux(t, q)
 4: procedure Select-Aux(t, q)          ▷ Invariant: PE[q − 1] < t ≤ PE[q]
 5:     if t > HE[q] then                ▷ If position t belongs to Z[q]'s tail
 6:         return A[t]                  ▷ A[t] is computed from TL[q], TC[q]
 7:     r ← q′ ← HS[q]                   ▷ Z[q]'s head is a suffix of Z[1]···Z[r]
 8:     j ← HE[q]                        ▷ j is the last position of Z[q]'s head
 9:     e ← PE[r]                        ▷ e is the last position of Z[r]
10:     Δ ← A[j] − A[e]                  ▷ Computed in O(1) time by Lemma 2
11:     t′ ← t − (j − e);                ▷ A[t] = A[t′] + Δ by Lemma 2
12:     while q′ > 1 and t′ ≤ PE[q′ − 1] do   ▷ Find phrase Z[q′] for t′
13:         q′ ← q′ − 1
14:     return Select-Aux(t′, q′) + Δ    ▷ Equals A[t] by Lemma 2
```

**Algorithm 2** Recursive rank Procedure

```
 1: procedure Rank(v)
 2:     q ← smallest i such that A[PE[i]] ≥ v, found via a binary search
            on PE, using TL and TC
 3:     return Rank-Aux(v, q)
 4: procedure Rank-Aux(v, q)     ▷ Invariant: A[PE[q−1]] < v ≤ A[PE[q]]
 5:     j ← HE[q]                        ▷ j is the last position of Z[q]'s head
 6:     if v ≥ A[j] then                 ▷ If v falls into Z[q]'s tail
 7:         return j + rank of v in A[j, PE[q]]   ▷ Compute rank from TL[q]
                                          and TC[q] in O(log ε) time
 8:     r ← q′ ← HS[q]                   ▷ Z[q]'s head is a suffix of Z[1]···Z[r]
 9:     e ← PE[r]                        ▷ e is the last position of Z[r]
10:     Δ ← A[j] − A[e]                  ▷ Computed in O(1) time by Lemma 4
11:     v′ ← v − Δ;                      ▷ rank(v) = rank(v′) + (j − e) by Lemma 4
12:     while q′ > 1 and v′ ≤ A[PE[q′−1]] do   ▷ Find phrase Z[q′] for v′
13:         q′ ← q′ − 1
14:     return Rank-Aux(v′, q′) + j − e  ▷ Equals rank(v) by Lemma 4
```

To analyse Algorithm 1, we now introduce the notion of *meta-characters* of the $LZ_\varepsilon$ parsing of $S$. The first phrase $Z[1] = S[1, \mathsf{PE}[1]]$ in the parsing is our first meta-character (note $Z[1]$ has an empty head, so $\mathsf{HE}[1] = 0$ and the pair $\langle \mathsf{TL}[1], \mathsf{TC}[1] \rangle$ encodes the subarray $A[0, \mathsf{PE}[1]]$). Now, assuming we have already parsed $Z[1]\cdots Z[q-1]$ and partitioned $S[1, \mathsf{PE}[q-1]]$ into meta-characters, we partition the next phrase $Z[q]$ into meta-characters as follows: $Z[q]$'s tail will form a meta-character by itself, while $Z[q]$'s head "inherits" the partition into meta-characters from its source. Indeed, recall that $Z[q]$'s head is a copy of a suffix of $Z[1]\cdots Z[r]$, with $r = \mathsf{HS}[q]$. Such a suffix, say $S[a, b]$, belongs to the portion of $S$ already partitioned into meta-characters. Since by construction $Z[r]$'s tail is a meta-character $X_r$, we know that $X_r$ is a suffix of $S[a, b]$. Working backwards from $X_r$ we obtain the sequence $X_0 \cdots X_r$ of meta-characters covering $S[a, b]$. Note that it is possible that $X_0$, the meta-character containing $S[a]$, starts before $S[a]$. We thus define $X_0'$ as the suffix of $X_0$ starting at $S[a]$ and define the meta-character partition of $Z[q]$'s head as $X_0' X_1 \cdots X_r$. This process is depicted in Figure 6. Note that each meta-character is either the tail of some phrase or it is the suffix of a tail.

Armed with the definition of meta-characters, we can now prove the following result.

*Lemma 3: Algorithm 1 computes* $\mathsf{select}(t) = A[t]$ *in* $\mathcal{O}(\log z + M_{\max})$ *time, where* $z$ *is the number of phrases in the* $LZ_\varepsilon$ *parsing and* $M_{\max}$ *is the maximum number of meta-characters in a single phrase.*

*Proof:* The correctness of the algorithm follows by Lemma 2. To prove the time bound, observe that Line 2 clearly takes $\mathcal{O}(\log z)$ time. Let $\ell$ denote the number of meta-characters between the one containing position $t$ up to the end of $Z[q]$. We show by induction on $\ell$ that Select-Aux$(t, q)$ takes $\mathcal{O}(\ell)$ time. If $\ell = 1$, then $t$ belongs to $Z[q]$'s tail, and the value $A[t]$ is retrieved in $\mathcal{O}(1)$ time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$.

If $\ell > 1$, the algorithm retrieves the value $A[t']$ from a previous phrase $Z[q']$, with $q' = r - k$, where $k$ is the number of times Line 13 is executed. Since $Z[q]$ meta-characters are induced by those in its source, we get that the number of meta-characters between the one containing $t'$ and the end of $Z[r]$ is $\ell - 1$, and the number of meta-characters between the one containing $t'$ and the end of $Z[q']$ is $\ell' \leq \ell - 1 - k$. By the inductive hypothesis, the call to Select-Aux$(t', q')$ takes $\mathcal{O}(\ell')$, and the overall cost of Select-Aux$(t, q)$ is $\mathcal{O}(k) + \mathcal{O}(\ell') = \mathcal{O}(\ell)$, as claimed. □

## B. SUPPORTING RANK QUERIES

We now show how to support rank queries, starting with the following lemma whose proof is analogous to the one of Lemma 2.

*Lemma 4: Let* $S[i+1, j]$ *denote the head of phrase* $Z[q]$*, and let* $r = \mathsf{HS}[q]$ *and* $e = \mathsf{PE}[r]$*. Then, for any* $v$ *such that* $A[i] < v \leq A[j]$*, it holds* $\mathsf{rank}(v) = \mathsf{rank}(v - (A[j] - A[e])) + (j - e)$*, where* $A[j]$ *(resp.* $A[e]$*) can be retrieved in constant time from* $\mathsf{TL}[q]$ *and* $\mathsf{TC}[q]$ *(resp.* $\mathsf{TL}[r]$ *and* $\mathsf{TC}[r]$*).*

*Lemma 5: Algorithm 2 computes* $\mathsf{rank}(v)$ *in* $\mathcal{O}(\log z + M_{\max} + \log \varepsilon)$ *time, where* $z$ *is the number of phrases in the* $LZ_\varepsilon$

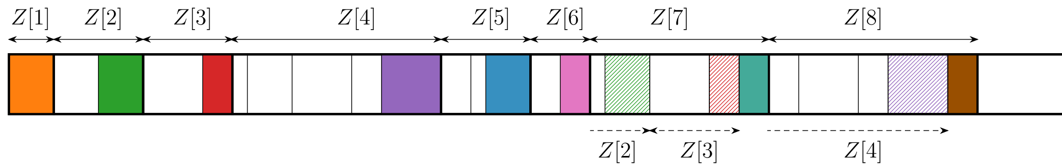


**FIGURE 7.** The $LZ_\varepsilon$ parsing of the same string of Figure 6 with $M = 5$. The phrase $Z[7]$ from Figure 6 is invalid since it has 13 meta-characters. $Z[7]$ head can have at most 4 meta-characters, so we define $Z[7]$ by setting $HS[7] = 3$ (Step 2b). Next, we define $Z[8]$ by setting $HS[8] = 4$ (Step 2c).

*parsing and $M_{\max}$ is the maximum number of meta-characters in a single phrase.*

*Proof:* Algorithm 2 follows closely the scheme of Algorithm 1. First, we compute the index $q$ of the phrase $Z[q]$ such that $A[\mathsf{PE}[q-1]] < v \le A[\mathsf{PE}[q]]$ with a binary search on the values $A[\mathsf{PE}[i]]$. This takes $\mathcal{O}(\log z)$ time, since we can retrieve $A[\mathsf{PE}[i]]$ in constant time using $\mathsf{PE}[i]$, $\mathsf{TL}[i]$ and $\mathsf{TC}[i]$.

Next, we set $j = \mathsf{HE}[q]$ and check in Line 6 if $v$ falls into $Z[q]$'s tail, i.e., $v \ge A[j]$ (observe we can retrieve $A[j]$ in constant time since it is the first value covered by the segment associated with $Z[q]$'s tail or, if $Z[q]$'s head is empty, it is the last value covered by the segment associated with $Z[q-1]$'s tail). If so, we return $j$ plus the rank of $v$ in $A[j, \mathsf{PE}[q]]$, which we can compute in $\mathcal{O}(\log \varepsilon)$ time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$ using the algorithm in [25, §3].

Otherwise, if $v < A[j]$, we use Lemma 4 and compute $\mathsf{rank}(v)$ recursively from a previous phrase $Z[q']$. Reasoning as in the proof of Lemma 3, we get that the overall time complexity is $\mathcal{O}(\log z + M_{\max} + \log \varepsilon)$. □

It is easy to see that, in general, Algorithms 1 and 2 take $\Theta(M_{\max})$ time. Unfortunately in the worst case it is $M_{\max} = \Theta(n)$: to see this, consider a parsing where each phrase $Z[q]$ is such that the head is a copy of $Z[1] \cdots Z[q-1]$ and the tail has length 2; then $Z[q]$ contains $2^{q-1}$ meta-characters, and the last phrase contains $M_{\max} \approx n/4$ meta-characters. To reduce this time complexity, we now show how to modify the $LZ_\varepsilon$ parsing so that $M_{\max}$ is upper bounded by a user-defined parameter $M > 1$. The resulting parsing could contain some repeated phrases, but note that Lemmas 3 and 5 do not require the phrases to be different: repeated phrases will only affect the space usage.

To build an $LZ_\varepsilon$ parsing in which each phrase contains at most $M$ meta-characters, we proceed as follows. Assuming $S[1, i]$ has already been parsed as $Z[1], \ldots, Z[q-1]$, we first compute the longest prefix $S[i+1, j]$ which is a suffix of $Z[1] \cdots Z[r]$ for some $r < q$. Let $m$ denote the number of meta-characters in $S[i+1, j]$. Then (see Figure 7):

1) If $m < M$, then $Z[q]$ is defined as usual with $\mathsf{HS}[q] = r$. Since $Z[q]$'s tail constitutes an additional meta-character, $Z[q]$ has $m + 1 \le M$ meta-characters, as required.
2) Otherwise, if $m \ge M$, we do the following.
   a) We scan $S[i+1, j]$ backward dropping copies of $Z[r], Z[r-1], \ldots$ until we are left with a prefix $S[i+1, k_s]$ which contains less than $M$ meta-characters. By construction, $S[i+1, k_s]$ is either empty or is a suffix of $Z[1] \cdots Z[s]$ for some $s < r$.
   b) We define $Z[q]$ by setting $S[i+1, k_s]$ as its head and by defining $Z[q]$'s tail as usual.
   c) Next, we consider $Z[s+1] \equiv S[k_s, k_{s+1}]$. If $Z[q]$ ends before position $k_{s+1}$ (i.e. $\mathsf{PE}[q] < k_{s+1}$), we define an additional phrase $Z[q+1]$ using $Z[s+1]$ as a source, i.e. $\mathsf{HS}[q+1] = s+1$, setting its head to $S[\mathsf{PE}[q]+1, k_{s+1}]$ and with a tail defined as usual. To see that $Z[q+1]$ has at most $M$ meta-characters, we observe that $Z[s+1]$ contains at most $M$ meta-characters and the first one is covered by $Z[q]$'s tail since, as we already observed, each meta-character is a tail or the suffix of a tail and therefore can be covered by a linear $\varepsilon$-approximation.

*Lemma 6: The $LZ_\varepsilon$ parsing with limit $M$ contains at most $2n/M$ repeated phrases.*

*Proof:* In the algorithm described above, repeated phrases are created only at Steps 2b and 2c. Indeed, both $Z[q]$ defined in Step 2b and $Z[q+1]$ defined in Step 2c could be identical to a previous phrase. However, the concatenation $Z[q]Z[q+1]$ covers at least $S[i+1, k_{s+1}]$ so by construction contains *at least* $M$ meta-characters. Hence, Steps 2b and 2c can be executed at most $n/M$ times. □

### C. DESIGNING THE FINAL PARSING $LZ_\varepsilon^\rho$

We denote by $LZ_\varepsilon^\rho$ the parsing computed with the above algorithm with $M = \lceil \log^{1+\rho} n \rceil$, where $\rho > 0$, and we denote by $z$ the number of phrases in the parsing.

The vectors $\mathsf{PE}$ and $\mathsf{HE}$ contain $z$ increasing values in the range $[1, n]$. We combine them in a single increasing integer sequence that we store in $2z \log \frac{n}{2z} + \mathcal{O}(z)$ bits using Lemma 1 (as a minor detail, this requires incrementing the elements $\mathsf{HE}$ by one so to avoid the case in which the head of a phrase $Z[q]$ is empty, and thus $\mathsf{HE}[q] = \mathsf{PE}[q-1]$ and the combined sequence is not increasing).

We encode $\mathsf{HS}$ using $z$ cells of size $\lceil \log z \rceil = \log z + \mathcal{O}(1)$ bits, for a total of $z \log z + \mathcal{O}(z)$ bits.

For what concerns $\mathsf{TL}$, we observe that each pair of parameters $\mathsf{TL}[q] = \langle \alpha_q, \beta_q \rangle$ is actually derived from the line passing through the points $(a, A[a])$ and $(b, A[b])$, where $a$ and $b$ are found by [26] and are such that $\mathsf{HE}[q] \le a < b \le \mathsf{PE}[q]$. Then, we create two increasing integer sequences by concatenating the first and the second coordinate of all these

$2z$ points, and we compress them in $2z \log \frac{n}{2z} + 2z \log \frac{u}{2z} + \mathcal{O}(z)$ bits using Lemma 1.

Finally, let $t = |\mathsf{TC}|$ denote the total number of corrections in the parsing, which is the sum of the tails' length (plus one for each nonempty head). Clearly $t \leq n$, and if the gap array $S$ contains many repetitions we expect that $t \ll n$. We store the corrections in an array with $\lceil \log(2\varepsilon+1) \rceil$-bit cells, and we store the $z$ indexes marking the beginning of each segment's corrections in $z \log \frac{t}{z} + \mathcal{O}(z)$ bits using Lemma 1.

The above compressed encoding of PE, HE, HS and TL supports constant-time access to their elements, hence we can combine it with Lemma 3 and 5 and notice that in the time bounds it holds $\mathcal{O}(\log z) = \mathcal{O}(\log n) = \mathcal{O}(\log^{1+\rho} n)$. By adding the contribution of the array of corrections TC, we obtain the following result.

*Theorem 1: The $LZ_\varepsilon^\rho$ parsing supports* select *in $\mathcal{O}(\log^{1+\rho} n)$ time and* rank *in $\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$ time using $z \log z + 2z \log \frac{u}{2z} + 4z \log \frac{n}{2z} + \mathcal{O}(z)$ bits plus $t \lceil \log(2\varepsilon+1) \rceil + z \log \frac{t}{z} + \mathcal{O}(z)$ bits for the corrections, where $z$ is the number of phrases, and $t = |\mathsf{TC}|$ is the total number of corrections in the parsing.*

The space bound in the theorem above, divided into a part that accounts for the parsing plus a part that accounts for the corrections, shows the sensitivity of $\mathrm{LZ}_\varepsilon^\rho$ to both repetitiveness and approximate linearity. On the one hand, the more repetitive $S$ is, the longer are the phrase heads, and thus the smaller is $z$ and the contribution of the first part. On the other hand, the more $A$ exhibits approximate linearity, the smaller $\varepsilon$ can be chosen and thus the smaller is the contribution of the second part; also, as the segments associated with the tails get longer, the value of $z$ decreases too.

Finally, in the same vein to [33] for LZ77 and [34] for LZ-End, we now establish an alternative space bound for the $\mathrm{LZ}_\varepsilon^\rho$ parsing's heads in terms of the $k$th order empirical entropy of $S$.

*Lemma 7: The number of phrases $z$ in the $LZ_\varepsilon^\rho$ parsing of the gap array $S$ derived from a dictionary $A[1, n]$ over $\{0, \dots, u-1\}$ is such that*

$$z = \mathcal{O}\left(\frac{n}{\log n}\left(\log \tfrac{u}{n} + \log\log n\right)\right). \tag{2}$$

*Proof:* We write $z = z_r + z_d$, where $z_r$ is the number of repeated phrases, and $z_d$ is the number of distinct phrases. By Lemma 6 it is $z_r \leq 2n/(\log^{1+\rho} n)$ so $z_r$ satisfies (2). To bound the number of distinct phrases $z_d$, recall that by construction it is $\sum_{i=1}^{n} S[i] = A[n] < u$. Hence there can be at most $n \log \frac{u}{n} / \log n$ distinct phrases containing a symbol $S[i] \geq \Lambda_{u,n} = (u/n)(\log n / \log \frac{u}{n})$. The remaining distinct phrases are taken from an alphabet of size at most $\Lambda_{u,n}$; since their overall length is at most $n$, by [34, Lemma 3.9] they are at most

$$\mathcal{O}\left(\frac{n \log \Lambda_{u,n}}{\log n}\right) = \mathcal{O}\left(\frac{n}{\log n}\left(\log \tfrac{u}{n} + \log\log n\right)\right).$$

□

*Theorem 2: Let $\sigma$ denote the number of distinct gaps in $S$. If $\sigma = o(n)$, the arrays* PE, HE, *and* HS *produced by the $LZ_\varepsilon^\rho$ parsing take $nH_k(S) + o(n \log \frac{u}{n})$ bits for any positive $k = o(\log_\sigma n / \log\log n)$.*

*Proof:* We preliminary show that $z = o(n)$. As in the previous proof let $z = z_r + z_d$. By Lemma 6 it is $z_r \leq 2n/(\log^{1+\rho} n) = o(n)$, while for the number $z_d$ of distinct phrases by [34, Lemmas 3.9] implies

$$z_d = \mathcal{O}\left(n/\log_\sigma n\right) = o(n).$$

Since $f(x) = x \log(n/x)$ is increasing for $x < n/e$ and $z = o(n)$, using (2) we get

$$\begin{aligned} z \log \tfrac{n}{z} &= \mathcal{O}\left(\frac{n}{\log n}\left(\log \tfrac{u}{n} + \log\log n\right) \log\log n\right) \\ &= o(n \log \tfrac{u}{n}). \end{aligned} \tag{3}$$

As we already observed, the encoding of PE and HE takes $2z \log \frac{n}{2z} + \mathcal{O}(z)$ bits which by (3) is $o(n \log \frac{u}{n})$.

By [34, Lemma 3.10] the number of distinct phrases $z_d$ is related to $H_k(S)$ for any $k \geq 0$ by the inequality

$$z_d \log z_d \leq nH_k(S) + z_d \log \tfrac{n}{z_d} + \mathcal{O}(z_d(1 + k \log \sigma)). \tag{4}$$

The encoding of HS using $z$ cells of size $\log z + \mathcal{O}(1)$ bits takes a total of

$$z_r \log z \ + \ z_d \log z \ + \ \mathcal{O}(z) \ \text{ bits.}$$

Since $z_r = \mathcal{O}(n/\log^{1+\rho} n)$ and $z = o(n)$, the first term is $o(n)$. The second term can be bounded by noticing that, if $z_d \leq z_r$, the second term is smaller than the first. Otherwise, from (4) we have

$$\begin{aligned} z_d \log z &\leq z_d \log(2z_d) \\ &\leq nH_k(S) + z_d \log \tfrac{n}{z_d} + \mathcal{O}(z_d(1 + k \log \sigma)). \end{aligned}$$

Reasoning as in (4), we have $z_d \log \frac{n}{z_d} = o(n \log \frac{u}{n})$. Finally, we have

$$z_d(1 + k \log \sigma) = o(n \log \tfrac{u}{n})$$

by Lemma 7 and the fact that $k = o(\log_\sigma n / \log\log n)$ implies $k \log \sigma = o(\log n / \log\log n)$. □

The significance of Theorem 2 is the following: the contribution of the arrays PE, HE, and HS used by $\mathrm{LZ}_\varepsilon^\rho$ to encode the repetitions gets smaller as $S$'s repetitiveness increases; if $nH_k(S)$ becomes $o(n \log \frac{u}{n})$, the theorem shows that such contribution becomes smaller than the size of a classical compact $\mathcal{O}(n \log \frac{u}{n})$-bit representation of an integer dictionary.

Theorem 2 does not account for the cost of TL and TC, which are analysed in Theorem 1, since their cost is not related to the repetitiveness of $S$, which is measured by the entropy $H_k$, but rather to the approximate linearity of $A$.

## IV. THE BLOCK-$\varepsilon$ TREE

In this section, we design a repetition-aware version of the LA-vector by building a variant of the block tree [35], [37] on a combination of the gap-string $S$ and the piecewise linear $\varepsilon$-approximation. We name this variant block-$\varepsilon$ tree, and show that it achieves time-space bounds which are competitive with the ones achieved by block trees and LA-vectors

because it combines successfully both forms of compressibility discussed in this paper: repetitiveness and approximate linearity. We will first support this statement from a theoretical point of view and, in the next section, we will execute a wide set of experiments on real and synthetic datasets that will corroborate our analysis, showing that our block-$\varepsilon$ tree achieves the best or the second-best space occupancy in the majority of cases, being able to capture in a robust way both forms of compressibility studied in this article.

The main idea of the block-$\varepsilon$ tree consists in first building a traditional block tree structure over the gap-string $S[1, n]$ of $A$. Recall that every node of the block tree represents a substring of $S$, and thus it implicitly represents the corresponding subarray of $A$. Then, we prune the tree by dropping the subtrees whose corresponding subarray of $A$ can be covered more succinctly by segments and corrections (i.e. whose LA-vector representation wins over the block-tree representation). Note that, compared to LA-vector, we do not encode segments and corrections corresponding to substrings of $S$ that have been encountered earlier, that is, we exploit the repetitiveness of $S$ to compress the piecewise linear $\varepsilon$-approximation at the core of the LA-vector. On the other hand, compared to block trees, we drop subtrees whose substrings can be encoded more efficiently by segments and corrections, that is, we exploit the approximate linearity of subarrays of $A$. Below we detail how to orchestrate this interplay to achieve efficient queries and compressed space occupancy in the block-$\varepsilon$ tree.

Let us define the $\delta$ repetitiveness measure on $S$ as $\delta = \max\{d_k(S)/k \mid 1 \leq k \leq n\}$, where $d_k(S)$ is the number of distinct length-$k$ substrings of $S$ [29], [37]. For simplicity of exposition, assume that $n = \delta 2^h$ for some integer $h$. The block-$\varepsilon$ tree is organised into $h' \leq h$ levels. The first level (level zero) logically divides the string $S$ into $\delta$ blocks of size $s_0 = n/\delta$. In general, blocks at level $\ell$ have size $s_\ell = n/(\delta 2^\ell)$, because they are recursively halved until possibly reaching the last level $h = \log \frac{n}{\delta}$, where blocks have size $s_h = 1$.

At any level, if two blocks $S_q$ and $S_{q+1}$ are consecutive in $S$ and they form the leftmost occurrence in $S$ of their content, then we say that both $S_q$ and $S_{q+1}$ are marked. A marked block $S_q$ that is not in the last level becomes an internal node of the tree. Such an internal node has two children corresponding to the two equal-size sub-blocks into which $S_q$ is split. On the other hand, an unmarked block $S_r$ becomes a leaf of the tree because, by construction, its content occurs earlier in $S$ and thus we can encode it by storing (i) a leftward pointer $q$ to the marked blocks $S_q, S_{q+1}$ at the same level $\ell$ containing its leftmost occurrence, taking $\log \frac{n}{s_\ell}$ bits; (ii) the offset $o$ of $S_r$ into the substring $S_q \cdot S_{q+1}$, taking $\log s_\ell$ bits.[3] Furthermore, to recover the values of $A$ corresponding to $S_r$, we store (iii) the difference $\Delta$ between the value of $A$ corresponding to the beginning of $S_r$ and the value of $A$ at the pointed occurrence of $S_r$, taking $\log u$ bits. Overall, each unmarked block needs $\log n + \log u$ bits of space.

[3] In this section, we omit ceilings from the bit-sizes for simplicity.

To describe the pruning process, we first define a cost function $c$ on the nodes of the block-$\varepsilon$ tree. For an unmarked block $S_r$, we define the cost $c(S_r) = \log n + \log u$, which accounts for the space in bits taken by $q$, $o$ and $\Delta$. For a marked block $S_q$ at the last level $h$, we define the cost $c(S_q) = \log u$, which accounts for the space in bits taken by its single corresponding element of $A$. Instead, consider a marked block $S_q$ at level $\ell < h$ for which there exists a segment approximating with error $\varepsilon_q \leq \varepsilon$ the corresponding elements of $A$. Suppose $\varepsilon_q$ is minimal, that is, there is no $\varepsilon' < \varepsilon_q$ such that there exists a segment $\varepsilon'$-approximating those same elements of $A$. Let $y = \log(2\varepsilon_q + 1)$ be the space in bits needed to store a correction, and let $\kappa$ be the space in bits taken by the parameters $\langle \alpha, \beta \rangle$ of the segment, e.g. $\kappa = 2 \log u + \log n$ if we encode $\beta$ in $\log u$ bits and $\alpha$ as a rational number with a $\log u$-bit numerator and a $\log n$-bit denominator [25, §2]. We assign to such $S_q$ a cost $c(S_q)$ defined recursively as

$$c(S_q) = \min \begin{cases} \kappa + s_\ell\, y + \log \log u \\ 2 \log n + \sum_{S_x \in child(S_q)} c(S_x) \end{cases} \tag{5}$$

The first branch of Equation 5 accounts for an encoding of the subarray of $A$ corresponding to $S_q$ via an $\varepsilon_q$-approximate segment, the corrections of $y$ bits each for the $s_\ell$ elements in $S_q$, and the value of $y$, respectively. The second branch of Equation 5 accounts for an encoding that recursively splits $S_q$ into two children, i.e. an encoding via two $\log n$-bit pointers plus the optimal cost of the children. Finally, if there is no linear $\varepsilon$-approximation (and thus no $\varepsilon_q$-approximation with $\varepsilon_q \leq \varepsilon$) for $S_q$, we assign to such $S_q$ the cost indicated in the second branch of Equation 5.

A postorder traversal of the block-$\varepsilon$ tree is sufficient to assign a cost to its nodes and possibly prune some of its subtrees. Specifically, after recursing on the two children of a marked block $S_q$ at level $\ell$, we check if the first branch of Equation 5 gives the minimum. In that case, we prune the subtree rooted at $S_q$ and store instead the encoding of the block via the parameters $\langle \alpha, \beta \rangle$ and the $s_\ell$ corrections in an array $C_q$. As a technical remark, this pruning requires fixing the destination of any leftward pointer that starts from an unmarked block $S_r$ and ends to a (pruned) descendant of $S_q$. For this purpose, we first make $S_r$ pointing to $S_q$. Then, since any leftward pointer points to a *pair* of marked blocks (unless the offset is zero), both or just one of them belongs to the pruned subtree. In the second case, we require an additional pointer from $S_r$ to the block that does not belong to the pruned subtree. This additional pointer does not change the asymptotic complexity of the structure.

Overall, the pruning process yields a tree with $h' \leq h$ levels. An example of a block-$\varepsilon$ tree is depicted in Figure 8. Observe in the figure that the leftward pointer from the block [1 1 1 3] bifurcates so as to indicate the additional pointer to the block [3 3 2 1] in the non-pruned subtree, as per the technical remark above.

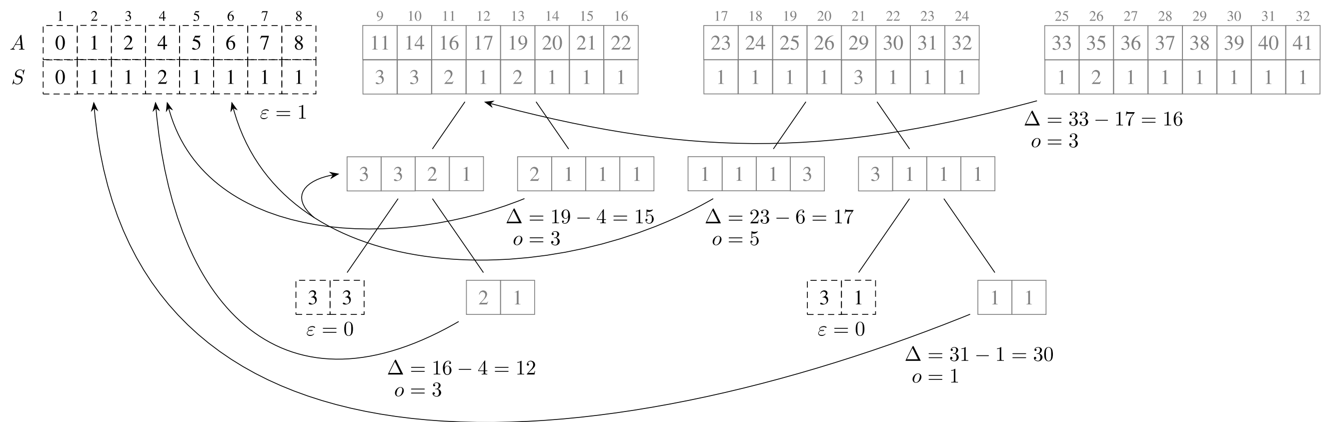


**FIGURE 8.** An example of a block-$\varepsilon$ tree built on an input array $A$ with corresponding gap-string $S$. The grey blocks are conceptual and not stored. The dashed blocks represent blocks encoded with a segment whose $\varepsilon$ value is shown below the block. A leftward pointer from a block $S_r$ to a block $S_q$ is annotated with the offset $o$ of the occurrence of $S_r$ into the substring $S_q \cdot S_{q+1}$ and with the difference $\Delta$ between the value of $A$ corresponding to the beginning of $S_r$ and the one at the pointed occurrence.

### A. SUPPORTING SELECT QUERIES

To answer $\mathsf{select}(i)$ in the block-$\varepsilon$ tree, we follow the path that starts from the first-level block into which position $i$ falls and proceeds towards a marked leaf block. We have the following cases for a visited block at level $\ell$:

- The block is an unmarked block $S_r$ pointing to $q$ with offset $o$ and difference value $\Delta = A[b] - A[a]$, where $b$ is the position corresponding to the beginning of $S_r$, and $a$ is the position corresponding to the beginning of the copy within $S_q$. First, we jump to either $S_q$ or $S_{q+1}$ depending on whether $o+i-b < s_\ell$, where $s_\ell$ is the size of the blocks at level $\ell$. Then, we turn the $\mathsf{select}(i) = A[i]$ query to $\Delta + \mathsf{select}(a+i-b) = \Delta + A[a+i-b]$. In fact, it holds

$$\begin{aligned}\Delta + A[a+i-b] &= \Delta + A[a] + S[a+1] + \cdots + S[a+i-b] \\ &= A[b] + S[a+1] + \cdots + S[a+i-b] \\ &= A[b] + S[b+1] + \cdots + S[b+i-b] \\ &= A[b+i-b] = A[i].\end{aligned}$$

- The block is a marked internal block. We jump to its left or right child depending on whether $i \bmod s_\ell < s_\ell/2$, and we continue computing $\mathsf{select}(i)$.
- The block is a marked leaf block $S_q$ storing the segment parameters $\langle \alpha, \beta \rangle$ and the local corrections $C_q$. We return $\lfloor \alpha i + \beta \rfloor + C_q[i \bmod s_\ell]$.
- The block is a marked leaf block $S_q$ at the last level $h$, thus we return its single element.

Let us now compute the time complexity of this traversal. First observe that, if we encounter a pruned block, the traversal stops. If we encounter an unmarked block, we follow its pointer to a pruned block or to an internal node. In this latter case, the traversal proceeds top-down with a constant amount of work per level. Therefore, the time complexity of $\mathsf{select}$ is $\mathcal{O}(h')$.

### B. SUPPORTING RANK QUERIES

For $\mathsf{rank}$ queries, we create a predecessor structure on the $\delta$ integers of $A$ corresponding to the last elements of the first-level blocks, i.e. the integers $A[in/\delta]$ for $i = 1, \ldots, \delta$. We use the structure of [30, Appendix A] giving a query time of $\mathcal{O}(\log\log_w \frac{u}{\delta})$, where $w$ is the word size, but there are many other possible trade-offs [43] that we skip for simplicity of exposition. Furthermore, in each marked block $S_q$ at any level $\ell$ except the first and the last ones, we store a sample of $A$ corresponding to the last element of $S_q$ to be able to descend to the correct child. The extra information does not change the asymptotic space complexity of our structure.

To answer $\mathsf{rank}(x)$, we start with a query to the predecessor structure, which indicates the first-level block into which $x$ falls, and then we proceed towards a marked leaf block. We have the following cases for a visited block at level $\ell$:

- The block is an unmarked block $S_r$ pointing to $q$ with offset $o$ and difference value $\Delta = A[b] - A[a]$. First, we jump to either $S_q$ or $S_{q+1}$ depending on whether $x - \Delta \le v$, where $v$ is the sample stored in $S_q$. Then, we recursively issue a $\mathsf{rank}$ query with argument $x - \Delta$, and we return $b - a + \mathsf{rank}(x - \Delta)$. The shift $b - a$ takes into account the leftward jump induced by the fact that we solve the $\mathsf{rank}$ query not on $S_r$ but on $S_q$ or $S_{q+1}$.
- The block is a marked internal block. We jump to its left or right child depending on whether $x \le v$, where $v$ is the sample stored in its left child, and we continue computing $\mathsf{rank}(x)$.
- The block is a marked leaf block $S_q$ storing the segment parameters $\langle \alpha, \beta \rangle$ and the local corrections $C_q$. We perform a binary search for $x$ on these $s_\ell$ corrections. Using the algorithm of [25, §3], this search costs $\mathcal{O}(\log \varepsilon_q)$ time and returns the result of $\mathsf{rank}(x)$, which is the position in $A$ of (the predecessor of) the value $x$.
- The block is a marked leaf block $S_q$ at the last level $h$, thus we return the rank of its single element.

Overall, the time complexity of rank is given by the sum of the costs of the initial predecessor search, the traversal of the block-$\varepsilon$ tree, and the final binary search, thus it is equal to $\mathcal{O}(\log\log_w \frac{u}{\delta} + h' + \log\varepsilon)$.

We observe that the block-$\varepsilon$ tree achieves space-time complexities no worse than a standard block tree constructed on $S$. This is due to the pruning of subtrees guided by the space-conscious cost function $c(\cdot)$ and by the resulting reduction in the number of levels, which positively impact the query time. Compared to LA-vector, the block-$\varepsilon$ tree can take advantage of repetitions and avoid the encoding of subarrays of $A$ corresponding to repeated substrings of $S$. Furthermore, since the block-$\varepsilon$ tree allocates the most succinct encoding for a subarray of $A$ by considering the smallest $\varepsilon_q \leq \varepsilon$ giving a linear $\varepsilon_q$-approximation, it could be regarded as the repetition-aware analogous of the space-optimised LA-vector [25, §5], in which different values of $\varepsilon$ are chosen for different chunks of $A$ so to minimise the overall space. Unlike LA-vector, the block-$\varepsilon$ tree has the advantage of potentially storing fewer corrections at the cost of storing the tree topology. Using the straightforward pointer-based encoding we discussed above, the tree topology takes $\mathcal{O}(\delta \log \frac{n}{\delta} \log n)$ bits in the worst case, but in the next section we propose an implementation that exhibits a more succinct pointer-less encoding (details in Section V-A). We notice, nonetheless, that the more repetitive the string $S$ is, the smaller is $\delta$, thus the overhead of the tree topology gets negligible.

Summing up, we proved the following result.

*Theorem 3: The block-$\varepsilon$ tree supports* rank *in $\mathcal{O}(\log\log \frac{u}{\delta} + \log \frac{n}{\delta} + \log\varepsilon)$ time and* select *in $\mathcal{O}(\log \frac{n}{\delta})$ time using $\mathcal{O}(\delta \log \frac{n}{\delta} \log n)$ bits of space.*

Finally, we mention that the block-$\varepsilon$ tree could employ other compressed rank/select dictionaries in its nodes, yielding a hybrid compression approach that can benefit from the orchestration of bicriteria optimisation and proper pruning of its topology to achieve the best space occupancy, given a bound on the query time, or vice versa (à la [44], [45], [46]).

## V. EXPERIMENTS

We experimented with an implementation of the $\mathrm{LZ}_\varepsilon$ parsing and the block-$\varepsilon$ tree on a machine with 202 GB of RAM, an Intel Xeon Gold 5118 CPU, and the GCC 10.2.1 compiler.[4]

We compare our proposals with the block tree of [35] built on the characteristic bitvector `bv`$(A)$ of a sorted input array $A$, with the LA-vector of [25] in both its fixed-$\varepsilon$ version and its space-optimised version (that vary $\varepsilon$ on different segments), and with Elias-Fano. All these implementations are written in C++ and build on the `sdsl` library [27]. A comparison with other rank/select dictionaries was already investigated in the literature for the individual LA-vector and the block tree [25], [35].

[4]The source code is available at https://github.com/gvinciguerra/BlockEpsilonTree and https://github.com/gvinciguerra/LZEpsilon.

### A. IMPLEMENTATION NOTES

For both $\mathrm{LZ}_\varepsilon$ and block-$\varepsilon$ tree, we consider segments using corrections of bit-size $c = 0, 2, 3, \ldots, 14$ and thus $\varepsilon = \max(0, 2^{c-1} - 1)$. In our implementation we avoid the use of floating point values by representing the slope as a rational number and considering the floor of the intercept. An elementary calculation shows that in this setting each correction is an integer in $[-\varepsilon, \varepsilon + 1]$ and therefore can be encoded with $c$ bits.

#### 1) IMPLEMENTING $\mathrm{LZ}_\varepsilon$

We compute the $\mathrm{LZ}_\varepsilon$ parsing via a simple adaptation of the LZ-End parsing algorithm of [34] (although more asymptotically efficient algorithms exist [47], [48]). We slightly alter the definition of our $\mathrm{LZ}_\varepsilon$ phrases, given in Section III, so that whenever $Z[q]$'s head is computed and its source does not overlap $Z[q-1]$, we try to extend $Z[q]$'s head leftward so as to shorten $Z[q-1]$'s tail and thus store fewer correction values in $\mathsf{TC}[q-1]$. Once the phrases are computed:

- we represent $\mathsf{TC}[1, z]$ via a contiguous array of $c$-bit cells and store the $z$ indexes marking the beginning of each segment's corrections via Lemma 1;
- we store $\mathsf{TL}$ as an array of structures, each storing the slope $\alpha$ and the intercept $\beta$ of the segment associated with a tail;
- we represent both arrays $\mathsf{PE}[1, z]$ and $\mathsf{HE}[1, z]$ with a sequence $X$ of $2z$ integers marking the left and the right boundaries of the segments, and then compress $X$ via Lemma 1.

Using additional $o(n)$ bits on top of the compressed $X$, we can replace the binary search in Line 2 of Algorithm 1 with an $\mathcal{O}(\min\{\log z, \log \frac{n}{z}\})$-time predecessor query on $X$ (see [1, §4.4.2]).

#### 2) IMPLEMENTING BLOCK-$\varepsilon$ TREE

Instead of starting from a pre-determined number of blocks, we follow [35] and construct a full block tree, and then remove the top levels that do not contain any unmarked blocks.[5] We use a pointerless representation of the tree topology via a plain bitvector for each level indicating with a `0` which block in the level is unmarked (hence, it has a leftward copy) or pruned by a segment, and with a `1` which block is marked but not pruned by a segment (hence, it is an internal node). We use $\mathsf{rank}_1$ on these bitvectors to traverse the tree downwards. If we reach an unmarked or pruned node, we use $\mathsf{rank}_0$ on the bitvector to access two separate packed arrays[6] storing the pointers and the $\Delta$-values, respectively, associated with each unmarked or pruned block, respectively. We store the segment blocks as an array of structures, with each structure storing the slope $\alpha$, the intercept $\beta$, a pointer

[5]We experimented with the theoretical proposal of starting with $\delta$ blocks. Although this makes the query time faster, it worsens the compression (up to 2.7 times) as it misses the copies longer than $n/\delta$.

[6]By packed array, we mean an array with fixed-length entries sized to contain the largest array element.

**TABLE 1.** Time performance (in nanoseconds) and space occupancy (in Bits Per Integer, BPI) of the $LZ_\varepsilon$ parsing and the LA-vector with fixed $\varepsilon$ on standard datasets.

| Dataset | | | LA-vector fixed $\varepsilon$ | | | | | $LZ_\varepsilon$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name $(n/u)$ | $n/10^6$ | $u/10^6$ | $c$ | select | rank | BPI | Segments | $c$ | select | rank | BPI | Phrases | Avg. head+tail length |
| GOV2 (76.6%) | 18.85 | 24.62 | 2 | 91 | 168 | 3.64 | 272661 | 0 | 850 | 2849 | **1.76** | 252876 | 45 + 30 |
| GOV2 (40.6%) | 9.85 | 24.62 | 4 | 63 | 155 | 5.40 | 110632 | 3 | 546 | 3371 | **4.91** | 177325 | 12 + 45 |
| GOV2 (4.1%) | 1.00 | 24.62 | 7 | 35 | 98 | 8.99 | 14695 | 0 | 1277 | 1622 | **8.43** | 66434 | 3 + 12 |
| URL (5.6%) | 57.98 | 1039.92 | 4 | 144 | 173 | 6.29 | 1251528 | 0 | 3415 | 2827 | **5.13** | 2269175 | 18 + 8 |
| URL (1.3%) | 13.56 | 1039.91 | 7 | 101 | 136 | 8.87 | 220655 | 7 | 426 | 2261 | **8.17** | 212917 | 10 + 55 |
| URL (0.4%) | 3.73 | 1039.86 | 0 | 33 | 72 | 3.48 | 112137 | 0 | 909 | 1432 | **2.42** | 69797 | 38 + 16 |
| 5GRAM (9.8%) | 145.40 | 1476.73 | 5 | 175 | 306 | 7.15 | 2973400 | 5 | 593 | 4127 | **6.92** | 2467588 | 6 + 53 |
| 5GRAM (2.0%) | 29.20 | 1476.73 | 8 | 111 | 206 | **9.78** | 474794 | 8 | 476 | 3112 | 9.92 | 460960 | 3 + 61 |
| 5GRAM (0.8%) | 11.22 | 1476.69 | 9 | 85 | 145 | **10.91** | 170379 | 9 | 359 | 2590 | **10.91** | 168441 | 2 + 65 |
| DNA (49.0%) | 490.10 | 1000.00 | 4 | 281 | 490 | 5.26 | 5887530 | 4 | 681 | 5917 | **5.11** | 5209320 | 11 + 84 |
| DNA (29.5%) | 294.68 | 1000.00 | 5 | 243 | 454 | **6.24** | 3476467 | 5 | 564 | 5226 | **6.24** | 3204445 | 7 + 85 |
| DNA (19.6%) | 195.42 | 1000.00 | 6 | 195 | 413 | **7.04** | 1931804 | 6 | 480 | 4735 | 7.09 | 1808182 | 5 + 103 |

**TABLE 2.** Time performance (in nanoseconds) and space occupancy (in Bits Per Integer, BPI) of Elias-Fano, the space-optimised LA-vector, the block tree over the characteristic bitvector bv($A$) and the block-$\varepsilon$ tree.

| Dataset $(n/u)$ | Elias-Fano | | | LA-vector space opt. | | | Block tree on $\mathtt{bv}(A)$ | | | | | Block-$\varepsilon$ tree | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | select | rank | BPI | select | rank | BPI | $b$ | select | rank | BPI | Depth | $b$ | select | rank | BPI | Depth (Avg) |
| GOV2 (76.6%) | 64 | 91 | 3.33 | 69 | 130 | 1.85 | 64 | 668 | 519 | **0.69** | 12 | 16 | 451 | 825 | 1.89 | 14 (9.98) |
| GOV2 (40.6%) | 55 | 72 | 4.34 | 60 | 129 | 3.48 | 128 | 686 | 531 | **1.56** | 11 | 256 | 367 | 638 | 3.26 | 10 (8.73) |
| GOV2 (4.1%) | 38 | 43 | 7.60 | 33 | 96 | 3.01 | 32 | 645 | 573 | 4.62 | 13 | 128 | 407 | 465 | **2.92** | 10 (9.73) |
| URL (5.6%) | 103 | 113 | 6.70 | 124 | 144 | 2.83 | 32 | 1017 | 733 | **2.58** | 18 | 16 | 762 | 909 | 3.41 | 16 (12.94) |
| URL (1.3%) | 68 | 64 | 8.83 | 98 | 123 | **6.34** | 32 | 987 | 753 | 8.57 | 18 | 32 | 463 | 664 | 7.32 | 10 (8.39) |
| URL (0.4%) | 48 | 42 | 10.82 | 34 | 87 | **1.28** | 32 | 831 | 783 | 1.84 | 19 | 16 | 400 | 553 | 1.51 | 11 (7.92) |
| 5GRAM (9.8%) | 146 | 160 | 6.51 | 171 | 249 | 4.40 | 32 | 1176 | 876 | **3.64** | 18 | 32 | 621 | 999 | 5.01 | 12 (10.27) |
| 5GRAM (2.0%) | 80 | 81 | 8.65 | 132 | 177 | **6.37** | 32 | 1143 | 863 | 8.80 | 18 | 64 | 483 | 733 | 6.96 | 9 (7.81) |
| 5GRAM (0.8%) | 73 | 64 | 10.10 | 95 | 125 | **7.56** | 32 | 1017 | 826 | 11.25 | 19 | 64 | 421 | 592 | 8.34 | 9 (7.61) |
| DNA (49.0%) | 175 | 218 | 3.58 | 250 | 446 | 5.27 | 512 | 1158 | 922 | **2.09** | 14 | 512 | 535 | 1070 | 3.65 | 3 (2.98) |
| DNA (29.5%) | 171 | 194 | 4.48 | 218 | 416 | 6.20 | 512 | 1227 | 989 | **3.46** | 14 | 512 | 368 | 718 | 4.57 | 2 (1.96) |
| DNA (19.6%) | 157 | 188 | **4.93** | 195 | 384 | 6.69 | 512 | 1206 | 972 | 5.21 | 14 | 512 | 335 | 654 | 5.01 | 2 (1.94) |

to the correction packed array $C_q$, and the bit-size $c$ of a correction. Marked leaf blocks containing less than a number $b$ of elements are not split further, and they are concatenated left to right and encoded with Lemma 1. Intuitively, since these blocks cannot be replaced by leftward pointers or pruned by segments, they lack both repetitiveness and approximate linearity, hence a compression via Lemma 1 (or any other method) is likely to be more appropriate. The samples at each level needed to support rank on $A$ are stored in a packed array. For the predecessor query on the first-level samples, we use a binary search.

### B. RESULTS ON STANDARD DATASETS

Our first set of experiments evaluates our two proposals on *standard and well-known datasets*, which are not expected to exhibit any noticeable repetitive or linearity trends, so to evaluate the robustness of our approaches under somewhat unfavourable conditions. These datasets are: (i) three postings lists with different densities $n/u$ from the GOV2 inverted index [46]; (ii) six integer lists obtained by enumerating the positions of the first, second and third most frequent character in each of the Burrows-Wheeler transform of two text files: URL and 5GRAM [25]; (iii) three integers lists obtained by enumerating, respectively, the positions of both `T`s and `G`s or either of them in the Burrows-Wheeler transform of the first gigabyte of the human reference genome `GRCh38.p13`.

We start by comparing $LZ_\varepsilon$ with the LA-vector of [25] in which the bit-size $c$ of a correction (and thus $\varepsilon$) is fixed. In both approaches, we vary $c$ as indicated in Section V-A and report the most space-efficient configuration. We show the space occupancy in Bits Per Integer (BPI), and we measure the average query time in nanoseconds using two batches of $10^5$ random rank and select queries. We show the results in Table 1, where we highlight in bold the most space-efficient solution on each dataset. Results show that $LZ_\varepsilon$ improves or matches (there are 2 ties) the space of the LA-vector on 10 datasets out of 12, at the cost of being 10.76× slower in select and 15.80× slower in rank. The slower performance is not surprising since unrolling a source phrase may cause several cache misses, especially for rank (Algorithm 2) whose Lines 2 and 12 perform several random accesses to the TL and TC vectors. The improvement in space and the reduction in the number of linear models (compare the Segments and the Phrases columns when $c$ is equal) show that that exploiting

repetitiveness is still beneficial even for these datasets do not contain long repetitions (as it can be inferred from the column on the Average head length in Table 1); we will see much greater improvements on repetitive and linear datasets (in Section V-C).

In Table 2, we show the results for Elias-Fano, the space-optimised LA-vector, the standard block tree, and our block-$\varepsilon$ tree. For these last two, we use a branching factor of two, vary the length $b$ of the last-level blocks as $b \in \{2^3, 2^4, \dots, 2^9\}$ and show the most space-efficient configuration. First and foremost, we note that LA-vector is 10.51× faster in select and 4.69× faster in rank than the block tree on average, while for space there is no clear winner over all the datasets. This result is evidence of the interestingness of the combination of approximate linearity and repetitiveness. Instead, an information-theoretic approach like the one used by Elias-Fano does not achieve a good compression here, since it has the worst or the second-worst space in all the datasets except DNA.

Let us now compare the performance of our block-$\varepsilon$ tree with the other solutions. The block-$\varepsilon$ tree is 2.19× faster in select than the block tree, and it is either faster (in 7 cases, by 1.32×) or slower (in 5 cases, by 1.27×) in rank. With respect to Elias-Fano and the LA-vector, the block-$\varepsilon$ tree is always slower but, for what concerns the space, it achieves the best result in the sparsest GOV2, the second-best result in the majority (6) of the remaining (11) datasets. This shows that space-wise, the block-$\varepsilon$ tree can be a robust data structure in that it often achieves a good compromise by exploiting both kinds of regularities: repetitiveness (block trees) and approximate linearity (LA-vectors).

For what concerns a comparison between our $\mathrm{LZ}_\varepsilon$ and block-$\varepsilon$ tree, we can conclude from the data in Tables 1 and 2 that the latter achieves better compression than the former in all datasets except the densest one, i.e. GOV2 76.6%. This is because the block-$\varepsilon$ tree optimises the choice of $\varepsilon$ for each block, while $\mathrm{LZ}_\varepsilon$ uses the same $\varepsilon$ value over the whole dataset, thus it possibly uses too many bits for the corrections in data chunks that show strict linearity (this has also been observed in [25] for the LA-vector). Optimising the extent and $\varepsilon$-value of the phrase tails in $\mathrm{LZ}_\varepsilon$, which also impacts on the extent of the phrase heads, appears to be a hard problem to tackle, and for which further research is needed.

### C. RESULTS ON REPETITIVE AND LINEAR DATASETS

We now evaluate the experimented data structures on datasets where repetitions or linearities (or both) are explicitly forced in a synthetic way.

First, we examine the case of *repetitive datasets* generated from the GOV2 (4.1%) postings list of the previous section by applying the following two steps: (i) we concatenate the corresponding gap-string $S$ for 3, 6 or 9 times; then, (ii) each single repeated gap, corresponding to a document identifier (docID), is deleted with a probability of 10%, 1% or 0.1%. The results, depicted in Figure 9, show the effectiveness of the repetition-aware approaches (block trees, $\mathrm{LZ}_\varepsilon$ and block-$\varepsilon$ tree) over the LA-vector and Elias-Fano. Among these, the block tree and block-$\varepsilon$ tree generally achieve the best compression, especially when the deletion probability is low and thus there are longer uninterrupted copies. On the other hand, $\mathrm{LZ}_\varepsilon$ approaches their space performance as the number of repetitions of $S$ increases. Again, the same consideration of the previous section about the disadvantage of using a fixed $\varepsilon$ value applies also here. For what concerns the query time, we report that, in line with the experiments of the previous section, the LA-vector[7] and Elias-Fano obtained the fastest performance (the former in select and the latter in rank), followed by the block-$\varepsilon$ tree (8.06× and 14.16× slower than LA-vector in select and than Elias-Fano in rank, respectively), the block tree (17.75× and 11.87× slower), and $\mathrm{LZ}_\varepsilon$ (67.33× and 146.79× slower).

Second, we examine the case of three datasets that show *approximate linearity*, generated by adding a "random noise" of amplitude $a = 3$, 15 and 63 around linearly increasing integer sequences, as follows. Given $a$, we create array $A$ by choosing an integer length $\ell \in [10, 1000]$ and an integer slope $\alpha \in [2a+1, 3a]$ uniformly at random, and generating values $i\alpha + \eta$, where $i = 1, \dots, \ell$, and $\eta$ is an integer chosen uniformly at random in $[-a, a]$. Once $\ell$ integers have been generated, we repeat the process by sampling another random segment length and slope. We stop as soon as $A$ contains 5 million increasing integers. The results, depicted in Figure 10, show (not surprisingly) that $\mathrm{LZ}_\varepsilon$ behaves similarly to the LA-vector approaches, followed by the block-$\varepsilon$ tree and Elias-Fano. Clearly, these linearity-aware approaches use a larger correction bit-size and thus more space whenever the noise-amplitude $a$ grows. Furthermore, we notice (again, not surprisingly) that the standard block tree requires much more space than the other approaches as it does not capture approximate linearities. For what concerns the query time, again LA-vector and Elias-Fano obtained the fastest performance (the former in select and the latter in rank), followed by the block-$\varepsilon$ tree (2.15× and 1.43× slower than LA-vector in select and than Elias-Fano in rank, respectively), the block tree (17.53× and 6.99× slower), and $\mathrm{LZ}_\varepsilon$ (4.69× and 28.11× slower).

The final experiment is devoted to examining the case of datasets *with both repetitions and approximate linearities*. They are constructed by modifying the process above so that (i) the amplitude $a$ is chosen randomly in $\{3, 15, 63\}$, and (ii) with probability $p$, we do not generate a new segment of random length $\ell$ but append a copy $S[i, i+\ell-1]$ of the gap-string $S[1, n]$ generated so far, where $i$ is an integer chosen uniformly at random in $[1, n-\ell]$. We obtain three datasets by varying $p$ as 25%, 50% and 75%, respectively. The results, depicted in Figure 11, show that the space of Elias-Fano and the LA-vector approaches is unaffected by repetitions, that the block tree obtains a good space performance only in the case of high repetitions ($p = 75\%$), and that our block-$\varepsilon$ tree

[7]For simplicity, we consider just the query time of the space-optimised LA-vector here.

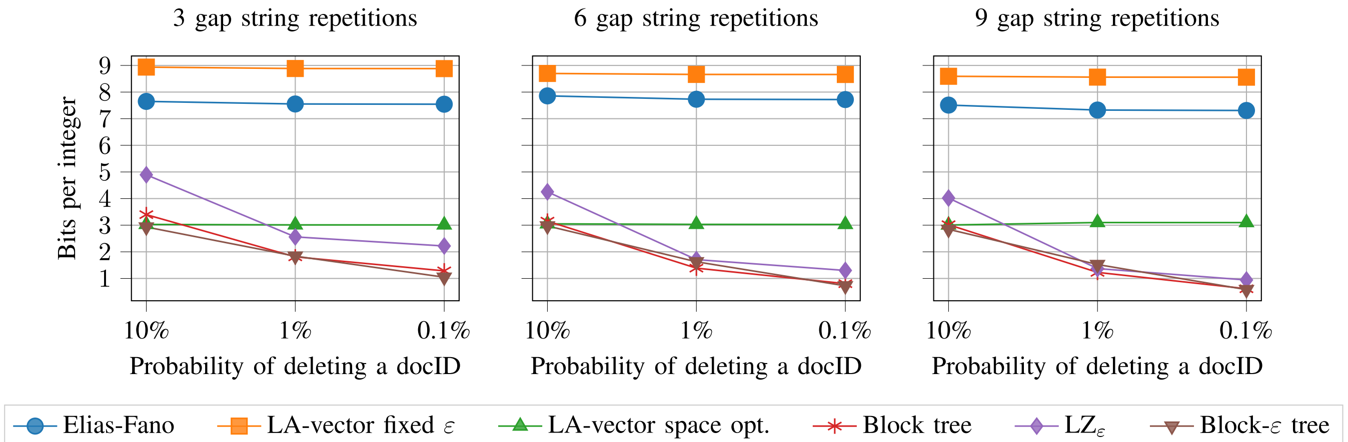


**FIGURE 9.** Space performance of several rank/select dictionaries on a postings list whose gap-string is repeated 3, 6 and 9 times, each time randomly deleting a docID with a given probability.

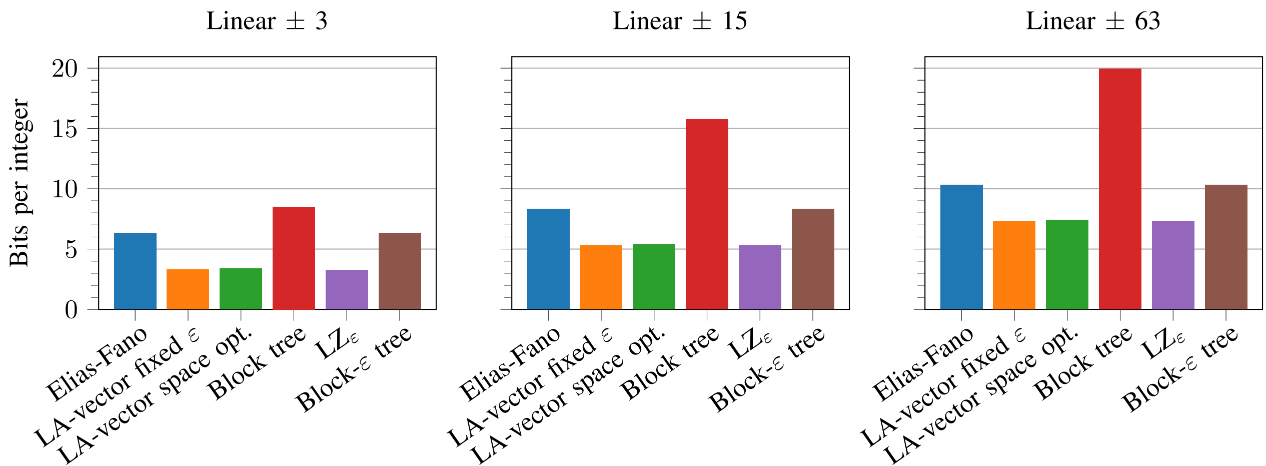


**FIGURE 10.** Space performance of several rank/select dictionaries on datasets with explicit linearities with a noise of amplitude 3, 15 and 63.

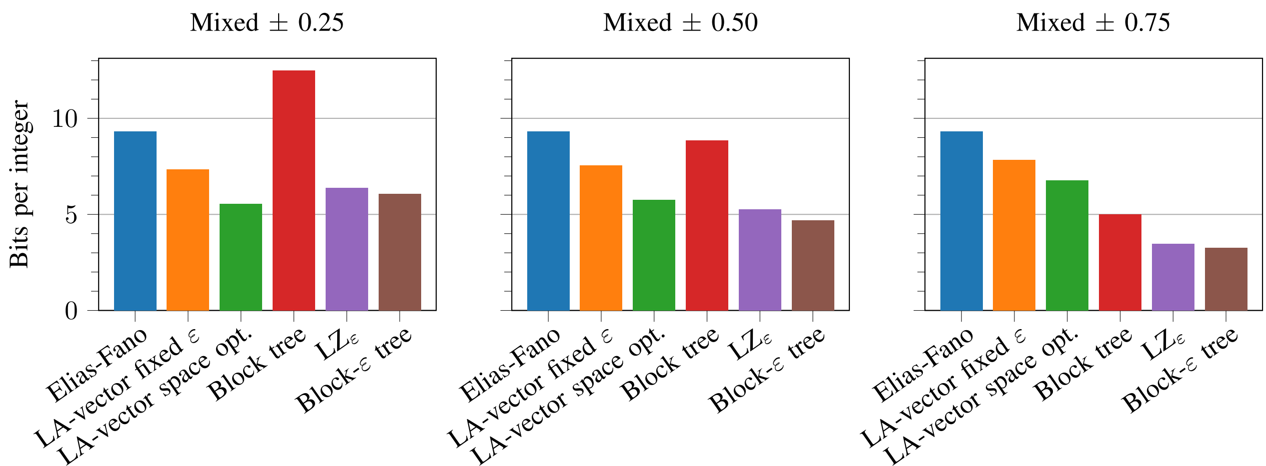


**FIGURE 11.** Space performance of several rank/select dictionaries on mixed datasets containing linearities and repetitions.

(followed closely by LZ$_\varepsilon$) achieves the best or the second-best space occupancy in all the cases, being able to capture both forms of compressibility studied in this article. For what concerns the query time, again LA-vector and Elias-Fano

obtained the fastest performance (the former in select and the latter in rank), followed by the block-$\varepsilon$ tree (4.46× and 7.55× slower than LA-vector in select and than Elias-Fano in rank, respectively), the block tree (18.82× and 10.18× slower), and $LZ_\varepsilon$ (7.74× and 32.46× slower).

### D. DISCUSSION

The experiments show that, also in a practical setting, it is indeed possible to exploit the presence of both approximate linearity and repetitions in the input to obtain significant space savings over state-of-the-art data structures for rank/select operations.

Indeed, on standard datasets (containing no evident repetitive or linearity trends), we found that there is no clear winner in space between the LA-vector and the block tree. In this scenario, our block-$\varepsilon$ tree achieved the best space or the second-best space in the majority of cases due to its effectiveness in exploiting both regularities. As far as the query time is concerned, the LA-vector and Elias-Fano obtained the fastest performance, followed by our block-$\varepsilon$ tree which generally achieved better performance than the block tree. Our $LZ_\varepsilon$ parsing on these standard datasets was, unfortunately, dominated by some other data structure in time and in space, mainly because it does not optimise the value of $\varepsilon$ for different chunks of the datasets (as instead the block-$\varepsilon$ tree and the LA-vector do), which appears to be a challenging open problem.

Motivated by these results, to shed light on scenarios in which repetitions and linearities are more evident, we considered synthetic datasets for which we proved that the space of the LA-vector does not improve with repetitions, that the space of the block tree does not improve with approximate linearities, and that both our block-$\varepsilon$ tree and $LZ_\varepsilon$ achieved improved space occupancy, being able to successfully capture both forms of compressibility studied in this article.

## VI. CONCLUSION

We introduced novel compressed rank/select dictionaries by exploiting two sources of regularity arising in real data: repetitiveness and approximate linearity. Our first contribution, the $LZ_\varepsilon^\rho$ parsing, supports queries in polylogarithmic time and has space bounds that show the sensitivity to both repetitiveness and approximate linearity, the former expressed in terms of the $k$th order empirical entropy. Our second contribution, the block-$\varepsilon$ tree, combines both sources of regularity in a tree data structure whose space and time bounds are expressed in terms of the repetitiveness measure $\delta$. We experimented with an implementation of these approaches showing that they effectively exploit both repetitiveness and approximate linearity.

Our study opens up a plethora of opportunities for future research. Firstly, we notice that the PGM-index [44] is also based on a variant of the piecewise linear $\varepsilon$-approximation, and thus it can still benefit from the ideas presented in this paper to make its space occupancy repetition aware. Secondly, the compression of segments and corrections in both $LZ_\varepsilon^\rho$ and the block-$\varepsilon$ tree is an orthogonal problem for which one can devise further compression mechanisms (see e.g. [44, Theorem 3]). Thirdly, co-optimising the choice of $\varepsilon$ and the tail length in the $LZ_\varepsilon^\rho$ phrases, which impact on the computation of the phrases' heads and the overall space, appears to be a non-trivial problem for which further research is needed. Fourthly, the construction of the $LZ_\varepsilon^\rho$ phrases and the block-$\varepsilon$ tree could be investigated inside a bicriteria framework, which seeks to optimise the query time and space usage under some given constraints [49]. Fifthly, inspired by the results achieved in this paper, we foresee a more query-efficient implementation of the block-$\varepsilon$ tree that computes an optimal node pruning using a family of compressed data structures in addition to $\varepsilon$-approximate segments. Readers interested in contributing to this algorithmic-engineering research line can look at the open-source code (see Footnote 4). Finally, we believe that repetitiveness and approximate linearity could be a perfect fit for arbitrary time series (hence, not necessarily the monotonic integer ones investigated in this paper), therefore we suggest an in-depth study and extension of our results to this scenario.

## APPENDIX A
## GAP VS BINARY ENTROPY INEQUALITY

Let $A[0, n]$ denote a sequence of strictly increasing integers, with $A[0] = 0$ and let $S[1, n]$ denote the gap-string $S[i] = A[i] - A[i-1]$. Finally, let $Z[1, u] = \mathrm{bv}(A)$, with $u = A[n]$ denote the characteristic bitvector of $A[1, n]$. In this appendix we prove the following result:

*Theorem 4: For every $k > 0$ it is $|S|H_k(S) \leq |Z|H_k(Z)$.*

*Proof:* Let $\Sigma$ denote the alphabet of $S[1, n]$, the alphabet of $Z$ is obviously $\{0, 1\}$. For any string $x$ over $\Sigma$, let $N(x)$ denote the number of occurrences of $x$ in $S$, and we write $x_S$ to denote the string of symbols immediately following each occurrence of $x$ in $S$. Notice that $|x_S| = N(x)$. We use a similar notation for $Z$: for any binary string $w$, $M(w)$ denotes the number of occurrences of $w$ in $Z$, and $w_Z$ denotes the string of bits immediately following each occurrence of $w$ in $Z$.

Fix a value $k > 0$. By definition the $k$th order empirical entropy is the optimal compression we can achieve using for each symbol a codeword that depends only on the $k$ symbols immediately preceding it. Hence, it can be expressed as [33]

$$|S|\,H_k(S) = \sum_{x \in \Sigma^k} |x_S|\,H_0(x_S), \tag{6}$$

where we assume $|x_S|\,H_0(x_S) = 0$ if $x_S$ is empty. Now consider a particular $x \in \Sigma^k$ such that $x_S$ is not empty, and let $m$ denote the largest value in $x_S$. We have[8]

$$|x_S|\,H_0(x_S) = -\sum_{i=1}^{m} N(xi)\,\log \frac{N(xi)}{N(x)}. \tag{7}$$

[8]In (7) the denominator should be $|x_S|$, which is equal to $N(x)$ except when $x = S[n-k+1, n]$ for which $|x_S| = N(x)-1$. For simplicity in the following we ignore this discrepancy, which however can be dealt with formally with some additional algebraic machinery.

Since the entropy is a lower bound to the average length of any prefix-free code, we know that for any set of values $\ell_1, \ldots, \ell_m$ that satisfy Kraft's inequality $\sum_i 2^{-\ell_i} \leq 1$, it is

$$|x_S|\, H_0(x_S) \leq \sum_{i=1}^{m} N(xi)\, \ell_i. \tag{8}$$

Let $x = g_1 g_2 \cdots g_k$. Every occurrence of $x$ in $S$ corresponds to an occurrence of the string $z_x = 0^{g_1-1}1\,0^{g_2-1}1 \cdots 0^{g_k-1}1$ in $Z$. For $i = 1, \ldots, m$, we define

$$\ell_i = -\log \frac{M(z_x\, 0^{i-1}\, 1)}{M(z_x)}.$$

It is

$$\sum_{i=1}^{m} 2^{-\ell_i} = \frac{M(z_x 1)}{M(z_x)} + \frac{M(z_x 01)}{M(z_x)} + \cdots + \frac{M(z_x 0^{m-1} 1)}{M(z_x)}.$$

The above summation is equal to one since each occurrence of $z_x$ in $Z$ is followed by up to $m-1$ 0's and followed by a 1, being $m$ the largest element in $x_S$. Since the values $\ell_i$ satisfy Kraft's inequality, setting

$$B(x, S) = -\sum_{i=1}^{m} N(xi)\, \log \frac{M(z_x 0^{i-1} 1)}{M(z_x)}.$$

Now, by (8) we have $|x_S|\, H_0(x_S) \leq B(x, S)$ and by (6), summing over all substrings $x \in \Sigma^k$ we get:

$$|S| H_k(S) \leq \sum_{x \in \Sigma^k} B(x, S).$$

To prove our claim, we will show that

$$\sum_{x \in \Sigma^k} B(x, S) \leq |Z| H_k(Z). \tag{9}$$

For our analysis, it is convenient to see $B(x, S)$ as a "cost" of encoding $x_S$. We split such cost among all symbols in $x_S$ by charging to each occurrence of the symbol $i$ in $x_S$ the cost $-\log M(z_x\, 0^{i-1} 1)/M(z_x)$. Since $i \in S$ is encoded with $0^{i-1}1$ in $Z$, we can further split the cost assigned to $i$ among the binary symbols in $0^{i-1}1$. Since

$$\frac{M(z_x\, 0^{i-1} 1)}{M(z_x)} = \frac{M(z_x\, 0)}{M(z_x)} \frac{M(z_x\, 00)}{M(z_x\, 0)} \cdots \frac{M(z_x\, 0^{i-1} 1)}{M(z_x\, 0^{i-1})},$$

we can split the cost $-\log(M(z_x\, 0^{i-1} 1)/M(z_x))$ by assigning, for $\ell = 1, \ldots, i-1$, to the $\ell$th 0 in $0^{i-1}1$ the cost $-\log(M(z_x\, 0^{\ell})/M(z_x\, 0^{\ell-1}))$, and to the final 1 in $0^{i-1}1$ the cost $-\log(M(z_x\, 0^{i-1} 1)/M(z_x\, 0^{i-1}))$.

In the above procedure, we have split the cost $B(x, S)$ among a set of symbols in $Z$. Given that each bit in $Z$ belongs to an encoding $0^{g-1}1$ of a value $g$ in $S$, it is easy to see that each bit in $Z$ gets charged exactly once with the only exception of bits in the prefix $Z[1, a_k]$, corresponding to the encoding of $S[1, k]$, which are not charged because the symbols in $S[1, k]$ do not belong to any $x_S$. We call the cost charged to each bit of $Z[a_k + 1, u]$ its $S$-cost.

Let $Z' = Z[a_k + 1 - k, u]$ be the charged portion of $Z$ prefixed by a context of size $k$. To prove (9), we show that the sum of the $S$-cost of all bits in $Z'$ is bounded by $|Z'|\, H_k(Z')$ which in turn is less than $|Z|\, H_k(Z)$, just because $Z'$ is a suffix of $Z$. Intuitively, the former inequality is true since $|Z'|\, H_k(Z')$ is the optimal cost of any encoding based on contexts of size $k$, while the total $S$-cost is the optimal cost of any encoding of $Z'$ based on variable-length contexts *all of them* of length $k$ or more (in fact, every context has the form $z_x\, 0^i$). This intuitive notion can be formalised using Jensen's inequality as follows. For any binary string $w$, let $M'(w)$ denote the number of occurrences of $w$ in $Z'$, and let $w_{Z'}$ the string consisting of the symbols immediately following each occurrence of $w$ in $Z'$. By definition we have

$$|Z'|\, H_k(Z') = \sum_{w \in \{0,1\}^k} |w_{Z'}|\, H_0(w_{Z'}). \tag{10}$$

We establish our result showing that any $w \in \{0, 1\}^k$ such that $w_{Z'}$ is not empty, the value $w_{Z'} H_0(w_{Z'})$ is never smaller than the sum of the $S$-costs of the symbols in $w_{Z'}$.

For any $e$, with $a_k \leq e < u$, let $v_e$ denote the longest suffix of $Z[1, e]$ containing exactly $k$ 1s (recall that each 1 is the last bit of the encoding of a gap value). For any $w \in \{0, 1\}^k$, let $E^0_w$ (resp. $E^1_w$) denote the set of positions $e$ such that $v_e$ ends with $w$ and $Z[e+1] = 0$ (resp. $Z[e+1] = 1$). By definition, the $S$-cost of $Z[e + 1]$ for $e \in E^0_w$ is equal to

$$-\log \frac{M(v_e 0)}{M(v_e)}.$$

Let $V = \{v_e | e \in E^0_w \cup E^1_w\}$. The total $S$-cost for all entries $Z[e + 1]$ for $e \in E^0_w$ is by definition

$$S(E^0_w) = -\sum_{v \in V} M(v0) \log \frac{M(v0)}{M(v)},$$

where we assume as usual $0 \log 0 = 0$. Note that by construction

$$\sum_{v \in V} M(v0) = M'(w0),$$

since every occurrence of $w0$ in $Z'$ corresponds to an occurrence of some $v0$ in $Z$ and vice versa, and similarly

$$\sum_{v \in V} M(v1) = M'(w1), \qquad \sum_{v \in V} M(v) = M'(w).$$

Applying Jensen's inequality to the concave function $\log(t)$ we have

$$\begin{aligned}
S(E^0_w) &= M'(w0) \sum_{v \in V} \frac{M(v0)}{M'(w0)} \log \frac{M(v)}{M(v0)} \\
&\leq M'(w0) \log \left( \sum_{v \in V} \frac{M(v0)}{M'(w0)} \frac{M(v)}{M(v0)} \right) \\
&\leq M'(w0) \log \frac{\sum_{v \in V} M(v)}{M'(w0)} \\
&\leq -M'(w0) \log \frac{M'(w0)}{M'(w)}.
\end{aligned}$$

Repeating the same argument for $S(E_w^1)$ we get

$$S(E_w^1) \le -M'(w1) \log \frac{M'(w1)}{M'(w)}.$$

So the total $S$-cost $S(E_w^0) + S(E_w^1)$ for encoding the entries $Z[e+1]$ for $e \in E_w^0 \cup E_w^1$, which are the symbols in $w_{Z'}$, is bounded by $|w_{Z'}|H_0(w_{Z'})$. Summing over all $w \in \{0,1\}^k$ and using (10) we get the thesis. □

## REFERENCES


[1] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.
[2] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, pp. 552–581, Jul. 2005.
[3] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM J. Comput.*, vol. 35, no. 2, pp. 378–407, Jan. 2005.
[4] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comput. Surveys*, vol. 39, no. 1, p. 2, Apr. 2007.
[5] V. Mäkinen and G. Navarro, "Rank and select revisited and extended," *Theor. Comput. Sci.*, vol. 387, no. 3, pp. 332–347, Nov. 2007.
[6] S. Gog, J. Kärkkäinen, D. Kempa, M. Petri, and S. J. Puglisi, "Fixed block compression boosting in FM-indexes: Theory and practice," *Algorithmica*, vol. 81, no. 4, pp. 1370–1391, Apr. 2019.
[7] T. Gagie, G. Navarro, and N. Prezza, "Fully functional suffix trees and optimal text searching in BWT-runs bounded space," *J. ACM*, vol. 67, no. 1, pp. 1–54, Apr. 2020.
[8] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *Proc. 38th Annu. Symp. Found. Comput. Sci.*, 1997, pp. 118–126.
[9] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding $k$-ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 43, Nov. 2007.
[10] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Theory and practice of monotone minimal perfect hashing," *ACM J. Experim. Algorithmics*, vol. 16, pp. 1–3, May 2011.
[11] J. Barbay and G. Navarro, "Compressed representations of permutations, and applications," in *Proc. 26th Int. Symp. Theoretical Aspects Comput. Sci. (STACS)*, 2009, pp. 111–122.
[12] D. Clark, "Compact pat trees," Ph.D. dissertation, Univ. Waterloo, Waterloo, ON, Canada, 1996.
[13] J. Ian Munro, "Tables," in *Proc. 16th Conf. Found. Softw. Technol. Theor. Comput. Sci. (FSTTCS)*, 1996, pp. 37–42.
[14] M. Patrascu, "Succincter," in *Proc. 49th Annu. IEEE Symp. Found. Comput. Sci.*, Oct. 2008, pp. 305–313.
[15] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. 9th Workshop Algorithm Eng. Exp. (ALENEX)*, 2007, pp. 60–70.
[16] A. Golynski, A. Orlandi, R. Raman, and S. S. Rao, "Optimal indexes for sparse bit vectors," *Algorithmica*, vol. 69, no. 4, pp. 906–924, Aug. 2014.
[17] J. Karkkainen, D. Kempa, and S. J. Puglisi, "Hybrid compression of bitvectors for the FM-index," in *Proc. Data Compress. Conf.*, Mar. 2014, pp. 302–311.
[18] D. Arroyuelo and R. Raman, "Adaptive succinctness," in *Proc. 26th Int. Symp. String Process. Inf. Retr. (SPIRE)*, 2019, pp. 467–481.
[19] P. Ferragina, S. Kurtz, S. Lonardi, and G. Manzini, "Computational biology," in *Handbook of Data Structures and Applications*, 2nd ed., D. P. Mehta and S. Sahni, Eds. Boca Raton, FL, USA: CRC Press, 2018, ch. 59.
[20] V. Mäkinen, D. Belazzougui, F. Cunial, and I. A. Tomescu, *Genome-Scale Algorithm Design*. Cambridge, U.K.: Cambridge Univ. Press, 2015.
[21] G. Navarro, "Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences," *ACM Comput. Surveys*, vol. 46, no. 4, pp. 1–47, Apr. 2014.
[22] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 337–350.
[23] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
[24] A. Boffa, P. Ferragina, and G. Vinciguerra, "A 'learned' approach to quicken and compress rank/select dictionaries," in *Proc. 23rd SIAM Symp. Algorithm Eng. Exp. (ALENEX)*, 2021, pp. 46–59.
[25] A. Boffa, P. Ferragina, and G. Vinciguerra, "A learned approach to design compressed rank/select data structures," *ACM Trans. Algorithms*, vol. 18, no. 3, pp. 1–28, Jul. 2022.
[26] J. O'Rourke, "An on-line algorithm for fitting straight lines between data ranges," *Commun. ACM*, vol. 24, no. 9, pp. 574–578, Sep. 1981.
[27] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. 13th Int. Symp. Exp. Algorithms (SEA)*, 2014, pp. 326–337.
[28] P. Ferragina, F. Lillo, and G. Vinciguerra, "On the performance of learned data structures," *Theor. Comput. Sci.*, vol. 871, pp. 107–120, Jun. 2021.
[29] G. Navarro, "Indexing highly repetitive string collections, Part I: Repetitiveness measures," *ACM Comput. Surveys*, vol. 54, no. 2, pp. 29:1–29:31, 2021.
[30] D. Belazzougui and G. Navarro, "Optimal lower and upper bounds for representing sequences," *ACM Trans. Algorithms*, vol. 11, no. 4, pp. 1–21, Jun. 2015.
[31] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. Inf. Theory*, vol. IT-22, no. 1, pp. 75–81, Jan. 1976.
[32] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.
[33] S. R. Kosaraju and G. Manzini, "Compression of low entropy strings with Lempel-Ziv algorithms," *SIAM J. Comput.*, vol. 29, no. 3, pp. 893–911, 1999.
[34] S. Kreft and G. Navarro, "On compressing and indexing repetitive sequences," *Theor. Comput. Sci.*, vol. 483, pp. 115–133, Apr. 2013.
[35] D. Belazzougui, M. Cáceres, T. Gagie, P. Gawrychowski, J. Kärkkäinen, G. Navarro, A. Onez, J. Simon Puglisi, and Y. Tabei, "Block trees," *J. Comput. Syst. Sci.*, vol. 117, pp. 1–22, May 2021.
[36] D. Belazzougui, P. H. Cording, J. Simon Puglisi, and Y. Tabei, "Access, rank, and select in grammar-compressed strings," in *Proc. 23rd Annu. Eur. Symp. Algorithms (ESA)*, 2015, pp. 142–154.
[37] T. Kociumaka, G. Navarro, and N. Prezza, "Towards a definitive measure of repetitiveness," in *Proc. 14th Latin Amer. Symp. Theor. Informat. (LATIN)*, 2020, pp. 207–219.
[38] P. Ferragina, G. Manzini, and G. Vinciguerra, "Repetition- and linearity-aware rank/select dictionaries," in *Proc. 32nd Int. Symp. Algorithms Comput. (ISAAC)*, 2021, pp. 64:1–64:16.
[39] P. Elias, "Efficient storage and retrieval by content and address of static files," *J. ACM*, vol. 21, no. 2, pp. 246–260, Apr. 1974.
[40] R. M. Fano, "On number bits required to implement associative memory. Memo 61," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. Memo 61, 1971.
[41] T. Ideue, T. Mieno, M. Funakoshi, Y. Nakashima, S. Inenaga, and M. Takeda, "On the approximation ratio of LZ-End to LZ77," in *Proc. 28th Int. Symp. String Process. Inf. Retr. (SPIRE)*, 2021, pp. 114–126.
[42] D. Kempa and B. Saha, "An upper bound and linear-space queries on the LZ-End parsing," in *Proc. 33rd Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2022, pp. 2847–2866.
[43] G. Navarro and J. Rojas-Ledesma, "Predecessor search," *ACM Comput. Surveys*, vol. 53, no. 5, pp. 1–35, Oct. 2020.
[44] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020.
[45] P. Ferragina, R. Giancarlo, and G. Manzini, "The myriad virtues of wavelet trees," *Inf. Comput.*, vol. 207, no. 8, pp. 849–866, Aug. 2009.
[46] G. Ottaviano, N. Tonellotto, and R. Venturini, "Optimal space-time trade-offs for inverted indexes," in *Proc. 8th ACM Int. Conf. Web Search Data Mining*, Feb. 2015, pp. 47–56.
[47] D. Kempa and D. Kosolobov, "LZ-End parsing in linear time," in *Proc. 25th Annu. Eur. Symp. Algorithms (ESA)*, 2017, pp. 53:1–53:14.
[48] D. Kempa and D. Kosolobov, "LZ-End parsing in compressed space," in *Proc. Data Compress. Conf. (DCC)*, Apr. 2017, pp. 350–359.
[49] A. Farruggia, P. Ferragina, A. Frangioni, and R. Venturini, "Bicriteria data compression," *SIAM J. Comput.*, vol. 48, no. 5, pp. 1603–1642, Jan. 2019.

**PAOLO FERRAGINA** received the Ph.D. degree in computer science from the University of Pisa, in 1996. He is currently a Professor of algorithms at the University of Pisa, where he also serves as the Vice-Rector for ICT. He founded and leads the Acube Laboratory, where researchers design algorithms for big data, mainly in the form of texts and graphs, in collaboration with companies worldwide, such as Google, Bloomberg, European Broadcasting Union (EBU), Tiscali, Yahoo!, and ST Microelectronics. He has (co)authored more than 170 (refereed) publications, some books and chapters, achieving an H-index of 34 on Scopus, and more than 9500 citations on Google Scholar. His research results got four U.S. patents and some international awards, such as ''the 1995 Best Land Transportation Paper Award'' from IEEE Vehicular Technology Society, ''the 1997 EATCS Ph.D. Thesis Award,'' ''the 1997 Philip Morris Award on Science and Technology,'' and three Google research awards. He was on the Steering Committee of the European Symposium on Algorithms (ESA), and one of the area editors of two encyclopedias, such as algorithms and big data technologies. He is serving on the Editor Board for the *Journal of Graph Algorithms and Applications* (JGAA).

**GIOVANNI MANZINI** received the Ph.D. degree in mathematics from the Scuola Normale Superiore of Pisa, in 1995. He is currently a Professor of computer science at the University of Pisa and a Research Associate at the Institute of Informatics and Telematics, National Research Council. Previously, he was a Professor of computer science at the University of Eastern Piedmont. He has been a Visiting Scientist at the Massachusetts Institute of Technology and a Visiting Professor at Johns Hopkins University and the University of Melbourne. His current research interests include the design of algorithms and data structures for solving theoretical and applied problems in the fields of data compression and indexing data structures for massive data sets.

**GIORGIO VINCIGUERRA** received the Ph.D. degree in computer science from the University of Pisa, in 2022. He is currently a Postdoctoral Researcher at the University of Pisa. His research interests include compressed data structures, data compression, and algorithm engineering. His thesis was awarded the ''2022 Best Ph.D. Thesis in Theoretical Computer Science'' by the Italian Chapter of the EATCS.