


Repetition- and Linearity-Aware Rank/Select Dictionaries

Paolo Ferragina   

Department of Computer Science, University of Pisa, Italy

Giovanni Manzini   

Department of Computer Science, University of Pisa, Italy

Giorgio Vinciguerra   

Department of Computer Science, University of Pisa, Italy

Abstract

We revisit the fundamental problem of compressing an integer dictionary that supports efficient rank and select operations by exploiting two kinds of regularities arising in real data: *repetitiveness* and *approximate linearity*. Our first contribution is a Lempel-Ziv parsing properly enriched to also capture approximate linearity in the data and still be compressed to the k th order entropy. Our second contribution is a variant of the block tree structure whose space complexity takes advantage of both repetitiveness and approximate linearity, and results highly competitive in time too. Our third and final contribution is an implementation and experimentation of this last data structure, which achieves new space-time trade-offs compared to known data structures that exploit only one of the two regularities.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Data structures design and analysis

Keywords and phrases Data compression, Compressed data structures, Entropy

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.64

Supplementary Material *Software*: <https://github.com/gvinciguerra/BlockEpsilonTree>
archived at `swh:1:dir:a0f2406c8797804e8aec0afda6c06a80e945f85b`

Funding The authors have been supported in part by the Italian MUR PRIN project 2017WR7SHH: *Multicriteria data structures and algorithms*.

1 Introduction

We focus on the fundamental problem of representing an ordered dictionary A of n distinct elements drawn from the integer universe $[u] = \{0, \dots, u\}$ while supporting the operation $\text{rank}(x)$, that returns the number of elements in A which are $\leq x$; and $\text{select}(i)$, that returns the i th smallest element in A .

Rank/select dictionaries are at the heart of virtually any compact data structure [34], such as text indexes [15, 18, 20, 22, 30, 36], succinct trees and graphs [32, 41], hash tables [4], permutations [3], etc. Unsurprisingly, the literature is abundant in solutions, e.g. [2, 8, 21, 24, 31, 37, 40, 41]. Yet, the problem of designing theoretically and practically efficient rank/select structures is anything but closed. The reason is threefold. First, there is an ever-growing list of applications of compact data structures (in bioinformatics [13, 29], information retrieval [33], and databases [1], 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], sometimes requiring new data structuring techniques to fully exploit it, e.g. larger CPU registers, new instructions, parallelism, next-generation memories such as PMem. Third, data may present different kinds of regularities, which require different techniques that exploit them to improve the space-time performance.



© Paolo Ferragina, Giovanni Manzini, and Giorgio Vinciguerra;
licensed under Creative Commons License CC-BY 4.0

32nd International Symposium on Algorithms and Computation (ISAAC 2021).

Editors: Hee-Kap Ahn and Kunihiko Sadakane; Article No. 64; pp. 64:1–64:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

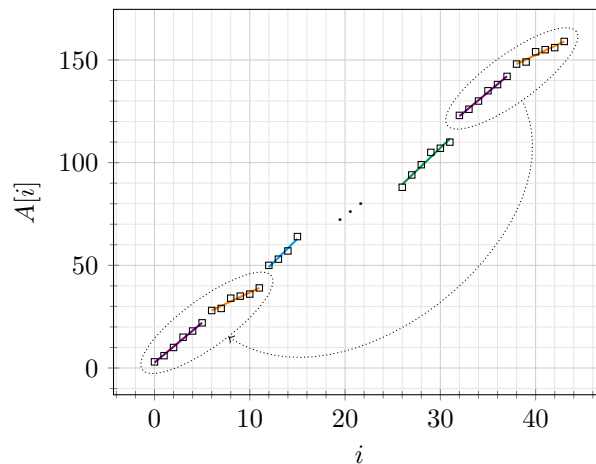
Among the latest of such regularities to be exploited, there is a geometric concept of *approximate linearity* [7]. Regard A as a sorted array $A[1, n]$, so that $\text{select}(i)$ can be implemented as $A[i]$. The idea is to first map each element $A[i]$ to a point $(i, A[i])$ in the Cartesian plane, for $i = 1, 2, \dots, n$. Intuitively, any function f that passes through all the points in this plane can be thought of as an encoding of A because we can recover $A[i]$ by querying $f(i)$. Now the challenge is to find a representation of f which is both fast to be computed and compressed in space. To this end, the authors of [7] implemented 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 ε .

► **Definition 1.** A piecewise linear ε -approximation for the integer array $A[1, n]$ is a partition of A into subarrays 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 $|(\alpha \cdot k + \beta) - A[k]| \leq \varepsilon$ for each $k \in [i, j]$.

Among all possible piecewise linear ε -approximations, one aims for the most succinct one, namely the one with the least amount of segments. This is a classical computational geometry problem that admits an $\mathcal{O}(n)$ -time algorithm [38]. The structure introduced by [7], named LA-vector, uses this succinct piecewise linear ε -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 $\text{select}(i)$, the LA-vector uses a constant-time rank structure on a bitvector of size n to find the segment $\langle \alpha, \beta \rangle$ covering i , and it returns the value $\lfloor (\alpha \cdot i + \beta) \rfloor + C[i]$. The $\text{rank}(x)$ operation is implemented via a sort of binary search that exploits the information encoded in the piecewise linear ε -approximation [7]. In practical implementations, we allocate $c \geq 0$ bits for each correction and set $\varepsilon = 2^c - 1$. The space usage in bits of an LA-vector consists therefore of a term $\mathcal{O}(nc)$ accounting for the corrections array C , and a term $\mathcal{O}(wm)$, where w is the word size, that grows with the number of segments m in the piecewise linear ε -approximation. Despite the apparent simplicity of the piecewise linear representation, the experiments in [7] show that the LA-vector offers the fastest select and competitive rank performance with respect to several well-established structures implemented in the `sds1` library [19]. In addition to its good practical performance, recent results [14, 17] have shown there are also theoretical reasons that justify the effectiveness of the piecewise linear ε -approximation in certain contexts.

Despite their succinctness and power in capturing linear trends, piecewise linear ε -approximations still lack the capacity to find and exploit one fundamental source of compressibility arising in real data: *repetitiveness* [35]. Although the input consists in an array A of strictly increasing values, 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 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 ε -approximating the subarray $A[i, j]$, we can use the segments ε -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 1). Therefore, in this case, we would need to store only the *shift* and the *reference* to the segments of $A[i', j']$. The LA-vector is unable to take advantage of such regularities. In the extreme case where A consists of the concatenation of a small subarray A' shifted by some amounts Δ_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 an LA-vector will be roughly $k + 1$ times the cost of representing A' .

The goal of this paper is to harness the repetitions in the gap-string S to make the LA-vector repetition aware. In fact, the approximate linearity and the repetitiveness of a string are different proxies of its compressibility and therefore it is interesting to take both of



■ **Figure 1** 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 ε -approximation for the top-right set can be obtained by shifting the segments for the bottom-left set.

them into account. Take as an example 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, following the argument above and considering the gap-string $S[i] = A[i] - A[i - 1] = 2 + B[i] - B[i - 1]$, we would not find repetitions longer than $h - 1$ in S . The challenge is to devise 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 ε -approximated well, while still supporting efficient rank/select primitives on A . We point out that, although in this paper we consider only linear approximations, our techniques can be applied also to other data approximation functions, such as polynomials and rational functions.

Our contribution in context. The most common approach in the literature to design a *succinct* dictionary for a set of distinct integers A over the universe $\{0, \dots, u\}$ is to represent A using the characteristic bitvector $\mathbf{bv}(A)$, which has length $u + 1$ and is such that $\mathbf{bv}(A)[i] = 1$ iff $i \in A$. In this paper, we use instead linear ε -approximations of A and the gap string S , and we show how to modify two known compression methods so that they can take advantage of approximate linearity. The first method is the Lempel-Ziv (LZ) parsing [26–28, 44], which is one of the best-known approaches to exploit repetitiveness [35]. The second method is the block tree [5], which is a recently proposed query-efficient alternative to LZ-parsing and grammar-based representations [6] suitable also for highly repetitive inputs since its space usage can be bounded in terms of the string complexity measure δ (see [25, 35] for the definition and significance of this measure).

Our first contribution is a novel parsing scheme, the LZ_ε^ρ parsing, whose phrases are a combination of a backward copy and a linear ε -approximation, i.e., a segment and the corresponding correction values. The LZ_ε^ρ parsing encapsulates a piecewise linear ε -approximation of the array A and supports efficient rank/select primitives on A . Surprisingly, this combination uses space bounded by the k th order entropy $H_k(S)$ of the gap-string S (see [26] for the definition and significance of k th order entropy). More precisely (Theorems 7 and 9), if σ denotes the number of distinct gaps in S , the LZ_ε^ρ parsing supports rank in $\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$ time and select in $\mathcal{O}(\log^{1+\rho} n)$ time using $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$

bits of space, for any positive ρ and $k = o(\log_\sigma n)$, plus the space to store the segments and the correction values that are used to advance the parsing (like the explicit characters in traditional LZ-parsing).

The best succinct data structure based on $\text{bv}(A)$ is the one by Sadakane and Grossi [43] that supports constant-time **rank** and **select** in $uH_k(\text{bv}(A)) + \mathcal{O}(u \log \log u / \log u)$ bits of space. This space bound cannot be compared to ours since it is given in terms of $H_k(\text{bv}(A))$ instead of $H_k(S)$. To achieve space $nH_k(S)$ one can use an entropy-compressed representation of S enriched with auxiliary data structures to support **rank/select** on A . For example, by sampling one value of A out of $\log n$ and performing a binary search followed by a prefix sum of the gaps one can support $\mathcal{O}(\log n)$ -time **rank** and **select** queries. Using the representation of [16], this solution uses $nH_k(S) + \mathcal{O}(n \log u / \log n) + o(n \log \sigma) = nH_k(S) + o(n \log u)$ bits of space, which is worse in space than our solution but faster in query time. Other trade-offs are possible: the crucial point however is that none of the known techniques is able to exploit simultaneously the presence of exact repetitions and approximate linearity in the input data as instead our LZ_ε^ρ does. In the best scenario, LZ_ε^ρ parsing uses segments to quickly consume any approximate linearity in A thus potentially reducing significantly the number of LZ-phrases. On the other hand, if A cannot be linearly approximated, segments will be short and the overall space occupancy of LZ_ε^ρ parsing will be $nH_k(S) + o(n \log \sigma)$ bits, i.e. no worse than a traditional LZ-parsing.

Our second contribution is the block- ε tree, an orchestration of block trees [5, 25] and linear ε -approximations. The main idea is to build the block tree over the gap-string S and to prune the subtrees whose corresponding subarray can be covered more succinctly using a linear ε -approximation in place of a block (sub)tree. We show that this solution 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 in the worst case, where δ is the string complexity of S [25, 35].

A block tree built on $\text{bv}(A)$, instead, supports **rank** and **select** in $\mathcal{O}(\log \frac{u}{\delta'})$ time using $\mathcal{O}(\delta' \log \frac{u}{\delta'} \log u)$ bits of space, where δ' is the string complexity measure on $\text{bv}(A)$. The time and space bounds achieved by the block tree and by our block- ε tree are not comparable due to the use of δ' instead of δ . Therefore, as our third contribution, we provide an implementation of our block- ε tree built on S , and we compare it with the standard block tree built on $\text{bv}(A)$. Our proposal turns out to be more space-efficient for some of the experimented sparse datasets and, as far as query time is concerned, it is $2.19\times$ faster in **select**, and it is either faster ($1.32\times$) or slower ($1.27\times$) in **rank** than the block tree.

In the Conclusions, we comment on several research directions that naturally arise from the novel approaches described in this paper.

2 Tools

We use as a black box the Elias-Fano [9, 10] representation for compressing and random-accessing monotone integer sequences [34, §4.4].

► **Lemma 2** (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 ε -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 ℓ 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 [38], which in optimal $\mathcal{O}(n)$ time computes the piecewise linear ε -approximation with the smallest number of segments for the set of points $\{(i, A[i]) \mid i = 1, \dots, n\}$.

Another key tool that we use is LZ-end [27]. Formally, the LZ-end parsing of a text $T[1, n]$ is a sequence f_1, f_2, \dots, f_z of phrases, such that $T = f_1 f_2 \dots f_z$, built as follows. If $T[1, i]$ has been parsed as $f_1 f_2 \dots f_{q-1}$, 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 $f_1 \dots f_r$ 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 addition of $T[j+1]$ 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 .

Although LZ-end is less powerful than the classic LZ77 parsing, which 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 is more efficient than LZ77 in extracting any substring of T .

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 [27]. Recently, [35] has given a complete picture of several alternative measures of information content and has shown that they are all lower bounded by the complexity measure δ defined as $\max\{T(k)/k \mid 1 \leq k \leq n\}$, where $T(k)$ is the number of distinct length- k substrings of T [42]. In [25], it is shown that using the block tree [5] it is possible to represent a text $T[1, n]$ in space bounded in terms of δ while supporting: $\text{rank}_a(i)$, which returns the number of occurrences of the character a in $T[1, i]$, and $\text{select}_a(j)$, which returns the position of the j th occurrence of a in T . Specifically, their block tree supports rank_a and select_a in $\mathcal{O}(\log \frac{n}{\delta})$ time using $\mathcal{O}(\sigma \delta \log \frac{n}{\delta} \log n)$ bits of space.

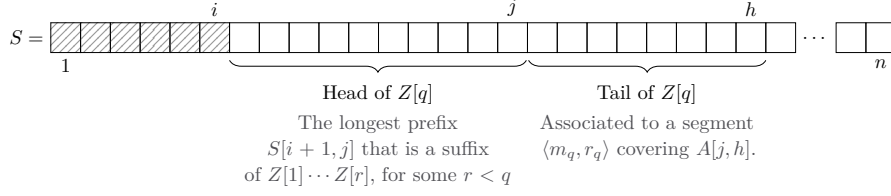
3 Two novel LZ-parsings: LZ_ε and $\text{LZ}_\varepsilon^\rho$

Assume that A contains distinct positive elements 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 ε -approximation with 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 a subarray of A with an error of at most ε (unlike classic LZ-end, which instead adds a single trailing character). We call this parsing the LZ_ε parsing of S .

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

1. We compute the longest prefix $S[i+1, j]$ of $S[i+1, n]$ that is a suffix of the concatenation $Z[1] \dots Z[r]$ 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 ε -approximated linearly, as well as the slope and intercept of such approximation. Note that using the algorithm of [38] 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 2, we call $S[i+1, j]$ the *head* of $Z[q]$ and $S[j+1, h]$ the *tail* of $Z[q]$. Note that 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, and thus no backward copy is executed. In the worst case, the longest subarray we can ε -approximate has length 2, which nonetheless guarantees that $Z[q]$ is nonempty. Experiments in [7] show that the average segment length ranges from 76 when $\varepsilon = 31$ to 1480 when $\varepsilon = 511$.



■ **Figure 2** Computation of the next phrase $Z[q]$ in the parsing of the gap-string S of the array A , where the prefix $S[1, i]$ has already been parsed into $Z[1], Z[2], \dots, Z[q-1]$.

If the complete parsing consists of λ phrases, we store it using:

- An integer vector $\text{PE}[1, \lambda]$ (Phrase Ending position) such that $h = \text{PE}[q]$ is the ending position of phrase $Z[q]$, that is, $Z[q] = S[i+1, h]$, where $i = \text{PE}[q-1] + 1$.
- An integer vector $\text{HE}[1, \lambda]$ (Head Ending position) such that $j = \text{HE}[q]$ is the last position of $Z[q]$'s head. Hence, $Z[q]$'s head is $S[\text{PE}[q-1] + 1, \text{HE}[q]]$, and $Z[q]$'s tail is $S[\text{HE}[q] + 1, \text{PE}[q]]$.
- An integer vector $\text{HS}[1, \lambda]$ (Head Source) such that $r = \text{HS}[q]$ is the index of the last phrase in $Z[q]$'s source. Hence, the head of $Z[q]$ is a suffix of $Z[1] \cdots Z[r]$. If the head of $Z[q]$ is empty then $\text{HS}[q] = 0$.
- A vector of pairs $\text{TL}[1, \lambda]$ (Tail Line) such that $\text{TL}[q] = \langle \alpha_q, \beta_q \rangle$ are the coefficients of the segment associated to the tail of $Z[q]$. By construction, such segment provides a linear ε -approximation for the subarray $A[\text{HE}[q], \text{PE}[q]]$.
- A vector of arrays $\text{TC}[1, \lambda]$ (Tail Corrections) such that $\text{TC}[q]$ is an array of length $\text{PE}[q] - \text{HE}[q] + 1$ providing the corrections for the elements in the subarray $A[\text{HE}[q], \text{PE}[q]]$ covered by $Z[q]$'s tail. By construction, such corrections are smaller than ε in modulus.

Using the values in TL and 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 3.** *Let $S[i+1, j]$ denote the head of phrase $Z[q]$, and let $r = \text{HS}[q]$ and $e = \text{PE}[r]$. Then, for $t = i+1, \dots, j$, it holds*

$$A[t] = A[t - (j - e)] + (A[j] - A[e]), \quad (1)$$

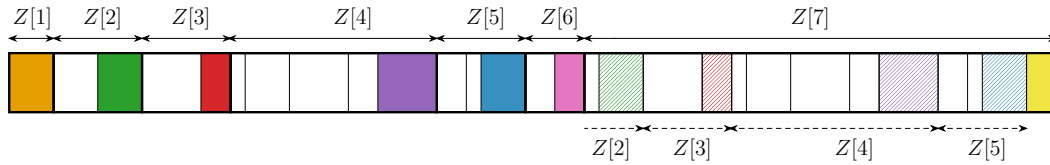
where $A[j]$ (resp. $A[e]$) can be retrieved in constant time from $\text{TL}[q]$ and $\text{TC}[q]$ (resp. $\text{TL}[r]$ and $\text{TC}[r]$).

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 $e = \text{PE}[r]$, it holds $S[i+1, j] \equiv S[e - j + i + 1, e]$ and

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

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

Using the above lemma, we can show by induction that given a position $t \in [1, n]$ we can retrieve $A[t]$. The main idea is to use a binary search on PE to retrieve the phrase $Z[q]$ containing t . Then, if $t \geq \text{HE}[q]$, we get $A[t]$ from $\text{TL}[q]$ and $\text{TC}[q]$; otherwise, we use Lemma 3 and get $A[t]$ by retrieving $A[t - (j - e)]$ using recursion. In the following, we will formalise this intuition in a complete algorithm, but before doing so, we need to introduce some additional notation.



■ **Figure 3** The LZ_ϵ 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 $Z[5]$ (notice the diagonal patterns in $Z[7]$'s head with the same colours of the tails in $Z[2] \cdots [5]$). 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.

Using the LZ_ϵ parsing, we partition the string S into *meta-characters* as follows. The first phrase in the parsing $Z[1] = S[1, PE[1]]$ is our first meta-character (note $Z[1]$ has an empty head, so $HE[1] = 0$ and the pair $\langle TL[1], TC[1] \rangle$ encodes the subarray $A[0, PE[1]]$). Now, assuming we have already parsed $Z[1] \cdots Z[q-1]$ and partitioned $S[1, PE[q-1]]$ 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 $r = HS[q]$. 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 3. Note that each meta-character is either the tail of some phrase or it is the suffix of a tail. We do not really compute the meta-characters but only use them in our analysis, as in the following result.

► **Lemma 4.** *Algorithm 1 computes $select(t) = A[t]$ in $\mathcal{O}(\log \lambda + M_{\max})$ time, where λ is the number of phrases in the LZ_ϵ parsing and M_{\max} is the maximum number of meta-characters in a single phrase.*

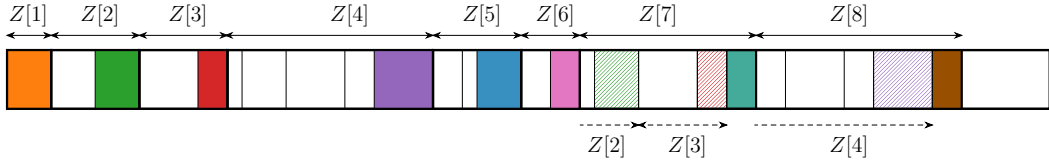
Proof. The correctness of the algorithm follows by Lemma 3. To prove the time bound, observe that Line 2 clearly takes $\mathcal{O}(\log \lambda)$ time. Let ℓ denote the number of meta-characters between the one containing position t up to the end of $Z[q]$. We show by induction on ℓ that $SELECT-AUX(t, q)$ 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 $TL[q]$ and $TC[q]$.

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. ◀

It is easy to see that for some input t Algorithm 1 takes $\Theta(M_{\max})$ time. To reduce the complexity, we now show how to modify the 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 Lemma 4 does not require the phrases to be different: repeated phrases will only affect the space usage.

■ **Algorithm 1** Recursive select procedure.

1:	procedure SELECT(t)	
2:	$q \leftarrow$ the smallest i such that $\text{PE}[i] \geq t$, found via a binary search on PE	
3:	return SELECT-AUX(t, q)	
4:	procedure SELECT-AUX(t, q)	▷ Invariant: $\text{PE}[q - 1] < t \leq \text{PE}[q]$
5:	if $t > \text{HE}[q]$ then	▷ If t belongs to the tail of $Z[q]$
6:	return $A[t]$	▷ $A[t]$ is computed from $\text{TL}[q], \text{TC}[q]$
7:	$r \leftarrow q' \leftarrow \text{HS}[q]$	▷ The head of $Z[q]$ is a suffix of $Z[1] \cdots Z[r]$
8:	$j \leftarrow \text{HE}[q]$	▷ j is the last position of the head of $Z[q]$
9:	$e \leftarrow \text{PE}[r]$	▷ e is the last position of $Z[r]$
10:	$\Delta \leftarrow A[j] - A[e]$	▷ Δ can be computed in $\mathcal{O}(1)$ time by Lemma 3
11:	$t' \leftarrow t - (j - e)$;	▷ $A[t] = A[t'] + \Delta$ by Lemma 3
12:	while $t' > \text{PE}[q']$ do	▷ Find the phrase $Z[\cdot]$ containing t'
13:	$q' \leftarrow q' - 1$	▷ Go back one word
14:	return SELECT-AUX(t', q') + Δ	▷ The returned value is $A[t]$ by Lemma 3



■ **Figure 4** The LZ_ϵ parsing of the same string of Figure 3 with $M = 5$. The phrase $Z[7]$ from Figure 3 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 $\text{HS}[7] = 3$ (Step 2b). Next, we define $Z[8]$ by setting $\text{HS}[8] = 4$ (Step 2c).

To build a LZ_ϵ parsing in which each phrase contains at most M meta-characters, we proceed as follows. Assuming $S[1, i]$ has already been parsed as $Z[1], \dots, Z[q - 1]$, 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 4):

1. If $m < M$, then $Z[q]$ is defined as usual with $\text{HS}[q] = r$. Since $Z[q]$'s tails constitute an additional meta-character, $Z[q]$ has $m + 1 \leq M$ meta-characters, as required.
2. Otherwise, if $m \geq M$, we do the following.
 - a. We scan $S[i + 1, j]$ backward dropping copies of $Z[r], Z[r - 1], \dots$ until we are left with a prefix $S[i + 1, k_s]$ containing less than M meta-characters. By construction, $S[i + 1, k_s]$ is a suffix of $Z[1] \cdots Z[s]$ for some $s < r$ and since each phrase contains at most M meta-characters, $S[i + 1, k_s]$ is non-empty.
 - b. We define $Z[q]$ by setting $S[i + 1, k_s]$ as its head, $\text{HS}[q] = s$, and by defining $Z[q]$'s tail as usual.
 - c. Next, we consider $Z[s + 1] \equiv S[k_s, k_{s+1}]$. By construction, $Z[s + 1]$ contains at most M meta-characters while $S[i + 1, k_{s+1}]$ contains more than M meta-characters. If $Z[q]$ ends before position k_{s+1} (i.e. $\text{PE}[q] < k_{s+1}$), we define an additional phrase $Z[q + 1]$ with heads equal to $S[\text{PE}[q] + 1, k_{s+1}]$, $\text{HS}[q + 1] = s + 1$ and with a tail defined as usual. This ensures that $Z[q]$ alone or $Z[q]Z[q + 1]$ contains at least M meta-characters.

► **Lemma 5.** *The LZ_ϵ 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. ◀

In the following, let σ denote the number of distinct gaps in S (i.e., the alphabet size of S), for any $\rho > 0$, we denote by LZ_ε^ρ the parsing computed with the above algorithm with $M = \log^{1+\rho} n$. The following lemma shows that the space to represent the parsing can be bounded in terms of the k th order entropy of the gap-string S plus $o(n \log \sigma)$ bits.

► **Lemma 6.** *Let σ denote the number of distinct gaps in S . The arrays PE, HE, and HS produced by the LZ_ε^ρ parsing can be stored in $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$ bits for any positive $k = o(\log_\sigma n)$, and still support constant-time access to their elements.*

Proof. Let λ denote the number of phrases in the parsing. We write $\lambda = \lambda_r + \lambda_d$, where λ_r is the number of repeated phrases, and λ_d is the number of distinct phrases. By Lemma 5 it is $\lambda_r \leq n/(2 \log^{1+\rho} n)$, while for the number λ_d of distinct phrases it is [27, Lemmas 3.9 and 3.10]

$$\lambda_d = \mathcal{O}\left(\frac{n}{\log_\sigma n}\right) \quad \text{and} \quad \lambda_d \log \lambda_d \leq nH_k(S) + \lambda_d \log \frac{n}{\lambda_d} + \mathcal{O}(\lambda_d(1 + k \log \sigma)) \quad (2)$$

for any constant $k \geq 0$. The vectors PE and HE contain λ increasing values in the range $[1, n]$. We encode each of them in $\lambda \log \frac{n}{\lambda} + \mathcal{O}(\lambda)$ bits using Lemma 2. Since $f(x) = x \log(n/x)$ is increasing for $x \leq n/e$ and $\lambda = \mathcal{O}(n/\log_\sigma n)$, it is $\lambda \log \frac{n}{\lambda} + \mathcal{O}(\lambda) = \mathcal{O}(n(\log \sigma)(\log \log n)/\log n) = o(n \log \sigma)$.

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

$$\lambda_r \log(\lambda_r + \lambda_d) + \lambda_d \log(\lambda_r + \lambda_d) + \mathcal{O}(\lambda) \quad \text{bits.}$$

Since $\lambda_d = \mathcal{O}(n/\log_\sigma n)$ and $\lambda_r = \mathcal{O}(n/\log^{1+\rho} n)$, it is $\lambda_d + \lambda_r = \mathcal{O}(n/\log_\sigma n)$ and the first term is $\mathcal{O}(n/\log^\rho n)$. The second term can be bounded by noticing that, if $\lambda_d \leq \lambda_r$, the second term is smaller than the first. Otherwise, from (2) we have

$$\lambda_d \log(\lambda_r + \lambda_d) \leq \lambda_d \log(2\lambda_d) \leq nH_k(S) + \lambda_d \log \frac{n}{\lambda_d} + \mathcal{O}(\lambda_d(1 + k \log \sigma)).$$

By the same reasoning as above, we have $\lambda_d \log \frac{n}{\lambda_d} = o(n \log \sigma)$ and $\lambda_d(1 + k \log \sigma) = \mathcal{O}((nk \log \sigma)/\log_\sigma n) = o(n \log \sigma)$ for $k = o(\log_\sigma n)$. ◀

Combining Lemma 6 with 4 and recalling that $\log \lambda = \mathcal{O}(\log^{1+\rho} n)$, we get

► **Theorem 7.** *Let σ denote the number of distinct gaps in S . Using the LZ_ε^ρ parsing we can compute $\text{select}(t)$ in $\mathcal{O}(\log^{1+\rho} n)$ time using $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$ bits of space plus the space used for the λ segments (array TL) and for the corrections of the elements in A covered by the tails in the parsing (array TC), for any positive $k = o(\log_\sigma n)$.*

In the proof of Lemma 6 one can see the interplay between the term $\mathcal{O}(n/\log^\rho n)$ coming from the repeated phrases and the term $o(n \log \sigma)$ coming from the distinct phrases in LZ_ε^ρ . In particular, if σ is small (i.e., there are few distinct gaps), then $o(n \log \sigma)$ becomes $\mathcal{O}(n \log \log n/\log n)$ and the space bound turns out to be $nH_k(S) + \mathcal{O}(n/\log^\rho n)$ bits. Also, note that the number of segments λ in LZ_ε^ρ is always smaller than the number of segments in a plain LA-vector. Also, the total length of the LZ_ε^ρ tails is always smaller than n . Hence, our approach is no worse than the LA-vector in space.

We now show that the LZ_ε^ρ parsing support efficient rank queries. The starting point is the following lemma, whose proof is analogous to the one of Lemma 3.

► **Lemma 8.** *Let $S[i+1, j]$ denote the head of phrase $Z[q]$, and let $r = \text{HS}[q]$ and $e = \text{PE}[r]$. Then, for any v such that $A[i] < v \leq A[j]$, it holds $\text{rank}(v) = \text{rank}(v - (A[j] - A[e])) + (j - e)$.*

► **Theorem 9.** *Using the LZ_ε^ρ parsing we can compute $\text{rank}(v)$ in $\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$ time within the space stated in Theorem 7.*

Proof. We answer $\text{rank}(v)$ with an algorithm similar to Algorithm 1. First, we compute the index q of the phrase $Z[q]$ such that $A[\text{PE}[q-1]] < v \leq A[\text{PE}[q]]$ with a binary search on the values $A[\text{PE}[i]]$. If the parsing has λ phrases this takes $\mathcal{O}(\log \lambda)$ time, since we can retrieve $A[\text{PE}[i]]$ in constant time using $\text{PE}[i]$, $\text{TL}[i]$ and $\text{TC}[i]$.

Next, we set $j = \text{HE}[q]$ and compare v with $A[j]$ (which again we can retrieve in constant time since it is the first value covered by $Z[q]$'s tail). If $v \geq A[j]$, we return j plus the rank of v in $A[j, \text{PE}[q]]$, which we can compute in $\mathcal{O}(\log \varepsilon)$ time from $\text{TL}[q]$ and $\text{TC}[q]$ using the algorithm in [7, §4]. If $v < A[j]$, we set $e = \text{PE}[\text{HS}[q]]$ and compute $\text{rank}(v)$ recursively using Lemma 8. Before the recursive call, we need to compute the index q' of the phrase such that $A[\text{PE}[q'-1]] < v' \leq A[\text{PE}[q']]$, for $v' = v - (A[j] - A[e])$. To this end, we execute the same while loop as the one in Lines 12–13 of Algorithm 1 with the test $t' > \text{PE}[q']$ replaced by $v' > A[\text{PE}[q']]$. Reasoning as in the proof of Lemma 4, we get that the overall time complexity is $\mathcal{O}(\log \lambda + M_{\max} + \log \varepsilon) = \mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$. ◀

4 The block- ε tree

In this section, we design a repetition aware version of the LA-vector by following an approach that focuses on query efficiency and uses space bounded in terms of the complexity measure δ reviewed in Section 2. We do so by building a variant of the block tree [5] on a combination of the gap-string S and the piecewise linear ε -approximation. We name this variant block- ε tree, and show that it achieves time-space bounds which are competitive with the ones achieved by block trees and LA-vectors [7] because it combines both forms of compressibility discussed in this paper: repetitiveness and approximate linearity.

The main idea of the block- ε 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 ε -approximation at the core of 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- ε tree.

For simplicity of exposition, assume that $n = \delta 2^h$ for some integer h , where δ is the string complexity of S . The block- ε tree is organised into $h' \leq h$ levels. The first level (level zero) logically divides the string S into δ blocks of size $s_0 = n/\delta$. In general, blocks at level ℓ 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 in which S_q is split into. 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 ℓ 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. Furthermore, to recover the values of A corresponding to S_r , we store (iii) the difference Δ 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.

To describe the pruning process, we first define a cost function c on the nodes of the block- ε 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 Δ . 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 ε_q is minimal, that is, there is no $\varepsilon' < \varepsilon_q$ such that there exists a segment ε' -approximating those same elements of A . Let κ 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 β in $\log u$ bits and α as a rational number with a $\log u$ -bit numerator and a $\log n$ -bit denominator [7, §2]. We assign to such S_q a cost $c(S_q)$ defined recursively as

$$c(S_q) = \min \begin{cases} \kappa + s_\ell \log \varepsilon_q + \log \log u \\ 2 \log n + \sum_{S_x \in \text{child}(S_q)} c(S_x) \end{cases} \quad (3)$$

The first branch of Equation (3) accounts for an encoding of the subarray of A corresponding to S_q via an ε_q -approximate segment, the corrections of $\log \varepsilon_q$ bits for each of the s_ℓ elements in S_q , and the exponent y of $\varepsilon_q = 2^y - 1$ to keep track of its value, respectively. The second branch of Equation (3) 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 ε -approximation (and thus no ε_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 (3).

A postorder traversal of the block- ε 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 ℓ , we check if the first branch of Equation (3) 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_ℓ 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, this pruning process yields a tree with $h' \leq h$ levels.

In the full paper, we show how to support **rank** and **select** queries on the block- ε tree in worst-case $\mathcal{O}(\log \log_w \frac{u}{\delta} + h' + \log \varepsilon)$ time and $\mathcal{O}(h')$ time, respectively.

We observe that the block- ε tree achieves space-time complexities no worse than a standard block tree construction 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- ε tree can take advantage of repetitions and avoid the encoding of subarrays of A corresponding to repeated substrings of S . Furthermore, since the block- ε tree allocates the most succinct encoding for a subarray of A by considering the smallest $\varepsilon_q \leq \varepsilon$ giving a linear ε_q -approximation, it could be regarded as the repetition-aware analogous of the space-optimised LA-vector [7, §6], in which all values of $\varepsilon = 0, 2^0, 2^1, \dots, 2^{\log u}$ are considered. The block- ε 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 for the next section we implement a more succinct pointerless encoding (details in the full paper). We notice, nonetheless, that the more repetitive is the string S , the smaller is δ , thus the overhead of the tree topology gets negligible.

Finally, we mention that the block- ε tree could employ other compressed rank-select dictionaries in its nodes, yielding a hybrid compression approach [39] 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 [12, 17, 39]).

5 Experiments

We implemented the block- ε tree, as it is the simplest and most practical contribution of this paper.

We compare our block- ε tree with the block tree of [5], built on the characteristic bitvector $\text{bv}(A)$, and with the space-optimised LA-vector of [7]. All these implementations are written in C++ and build on the `sds1` library [19]. For both the block tree and the block- ε tree, we use a branching factor of two and vary the length b of the last-level blocks as $b \in \{2^3, 2^4, \dots, 2^9\}$. Due to space limitations, we do not show the full space-time trade-off of each structure but report only the most space-efficient configurations. A comparison with other rank/select dictionaries is beyond the scope of our work, and it was already investigated in the literature for the individual LA-vector and the block tree [5, 7]. On the other hand, we note that our experimental study is the first to compare LA-vectors and block trees.

As datasets, we use (i) three postings lists with different densities n/u from the GOV2 inverted index [39]; (ii) six integers 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 [7]; (iii) three integers lists obtained by enumerating, respectively, the positions of both Ts and Gs, of Ts, and of Gs in the Burrows-Wheeler transform of the first gigabyte of the human reference genome GRCh38.p13.

For each tested structure, query operation, and dataset, we generate a batch of 10^5 random queries and measure the average query time in nanoseconds and the space occupancy of the structure in bits per integer on a machine with 202 GB of RAM and a 2.30 GHz Intel Xeon Gold 5118 CPU.

Table 1 shows the results. First and foremost, we note that LA-vector is $10.51\times$ faster in select and $4.69\times$ faster in rank than the block tree on average, while for space there is no clear winner over all the datasets. This comparison, which was not known in the literature, illustrates that the combination of approximate linearity and repetitiveness is interesting not only from a theoretical point of view, as commented in the introduction, but also from a practical point of view.

Let us now compare the performance of our block- ε tree against the block tree and the LA-vector. The block- ε tree is $2.19\times$ faster in select than the block tree, and it is either faster ($1.32\times$) or slower ($1.27\times$) in rank. With respect to LA-vector, the block- ε tree is always

■ **Table 1** The performance of LA-vector, the block tree over the characteristic bitvector $\text{bv}(A)$ and the block- ε tree over twelve datasets of different size n and universe size u . The **select** and **rank** columns show the average query time of the operations in nanoseconds. The space of each structure is shown in Bits Per Integer (BPI). For the block tree and the block- ε tree, the value b denotes the length of the last-level block that gave the most space-efficient configuration.

Dataset			LA-vector			Block tree on $\text{bv}(A)$					Block- ε tree on S				
Name (n/u)	$n/10^6$	$u/10^6$	select	rank	BPI	b	select	rank	BPI	Depth	b	select	rank	BPI	Depth (Avg)
GOV2 (76.6%)	18.85	24.62	69	130	1.85	64	668	519	0.69	12	16	451	825	1.89	14 (9.98)
GOV2 (40.6%)	9.85	24.62	60	129	3.48	128	686	531	1.56	11	256	367	638	3.26	10 (8.73)
GOV2 (4.1%)	1.00	24.62	33	96	3.01	32	645	573	4.62	13	128	407	465	2.92	10 (9.73)
URL (5.6%)	57.98	1039.92	124	144	2.83	32	1017	733	2.58	18	16	762	909	3.41	16 (12.94)
URL (1.3%)	13.56	1039.91	98	123	6.34	32	987	753	8.57	18	32	463	664	7.32	10 (8.39)
URL (0.4%)	3.73	1039.86	34	87	1.28	32	831	783	1.84	19	16	400	553	1.51	11 (7.92)
5GRAM (9.8%)	145.40	1476.73	171	249	4.40	32	1176	876	3.64	18	32	621	999	5.01	12 (10.27)
5GRAM (2.0%)	29.20	1476.73	132	177	6.37	32	1143	863	8.80	18	64	483	733	6.96	9 (7.81)
5GRAM (0.8%)	11.22	1476.69	95	125	7.56	32	1017	826	11.25	19	64	421	592	8.34	9 (7.61)
DNA (49.0%)	490.10	1000.00	250	446	5.27	512	1158	922	2.09	14	512	535	1070	3.65	3 (2.98)
DNA (29.5%)	294.68	1000.00	218	416	6.20	512	1227	989	3.46	14	512	368	718	4.57	2 (1.96)
DNA (19.6%)	195.42	1000.00	195	384	6.69	512	1206	972	5.21	14	512	335	654	5.01	2 (1.94)

slower. But, for what concerns the space, the block- ε tree improves both the LA-vector and the block tree in the sparsest GOV2 and DNA, and in the vast majority of the remaining datasets it is the second-best structure for space occupancy (except for the densest GOV2, URL and 5GRAM). This shows that space-wise, the block- ε 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 future work, we believe the block- ε tree can be improved along at least two avenues. First, the block- ε tree at a certain level is constrained to use fixed-length blocks (and thus segments), whilst the LA-vector minimises its space occupancy using segments whose start/end positions do not have to coincide with a subdivision in blocks. Removing this limitation, inherited from the block tree, would help to better capture approximate linearity and improve the space occupancy of the block- ε tree. Second, the block- ε tree captures the repetitiveness of the gap string S , while for the densest datasets of Table 1 it appears worthwhile to consider the repetitiveness in $\text{bv}(A)$, as done by the block tree. Therefore, adapting our pruning strategy to $\text{bv}(A)$ is likely to improve the space occupancy in these densest datasets (though, the space-time bounds will then depend on u instead of n).

6 Conclusions

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_ε^p parsing, combines backward copies with linear ε -approximation thus supporting efficient queries within a space complexity bounded by the k th order entropy of the gaps in the input data. Our second contribution, the block- ε tree, is a structure that adapts smoothly to both sources of regularities by offering an improved query-time efficiency compared to LZ_ε^p . We experimented with a preliminary implementation of the block- ε tree showing that it effectively exploits both repetitiveness and approximate linearity.

Our study opens up a plethora of opportunities for future research. Firstly, we notice that the PGM-index [17] is also based on a variant of the piecewise linear ε -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_ε^p and

the block- ε tree is an orthogonal problem for which one can devise further compression mechanisms (see e.g. [17, Theorem 3]). Thirdly, the construction of the LZ_ε^p phrases and the block- ε tree could be investigated inside a bicriteria framework, which seeks to optimise the query time and space usage under some given constraints [11]. Finally, inspired by our preliminary results, we plan to engineer a more query-efficient implementation of the block- ε tree that computes an optimal node pruning using a family of compressed data structures in addition to ε -approximate segments.

References

- 1 Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: enabling queries on compressed data. In *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
- 2 Diego Arroyuelo and Rajeev Raman. Adaptive succinctness. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2019.
- 3 Jeremy Barbay and Gonzalo Navarro. Compressed representations of permutations, and applications. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2009.
- 4 Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16, 2008.
- 5 Djamel Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez, Simon J. Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021.
- 6 Djamel Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, pages 142–154, 2015.
- 7 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A “learned” approach to quicken and compress rank/select dictionaries. In *Proc. 23rd SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59, 2021.
- 8 David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
- 9 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.
- 10 Robert Mario Fano. *On the number of bits required to implement an associative memory. Memo 61*. Massachusetts Institute of Technology, Project MAC, 1971.
- 11 Andrea Farruggia, Paolo Ferragina, Antonio Frangioni, and Rossano Venturini. Bicriteria data compression. *SIAM Journal on Computing*, 48(5):1603–1642, 2019.
- 12 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.
- 13 Paolo Ferragina, Stefan Kurtz, Stefano Lonardi, and Giovanni Manzini. Computational biology. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 59. CRC Press, 2nd edition, 2018.
- 14 Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theoretical Computer Science*, 871:107–120, 2021.
- 15 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
- 16 Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.
- 17 Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.
- 18 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1), 2020.

- 19 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.
- 20 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.
- 21 Alexander Golynski, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.
- 22 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- 23 John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.
- 24 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. 24th Data Compression Conference (DCC)*, pages 302–311, 2014.
- 25 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In *Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN)*, 2020.
- 26 Sambasiva Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
- 27 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
- 28 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
- 29 Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
- 30 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- 31 J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1996.
- 32 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, 1997.
- 33 Gonzalo Navarro. Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4), 2014.
- 34 Gonzalo Navarro. *Compact data structures: a practical approach*. Cambridge University Press, 2016.
- 35 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2), 2020.
- 36 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
- 37 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.
- 38 Joseph O’Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, 1981.
- 39 Giuseppe Ottaviano, Nicola Tonello, and Rossano Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. 8th ACM International Conference on Web Search and Data Mining (WSDM)*, 2015.
- 40 Mihai Pătraşcu. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2008.

64:16 Repetition- and Linearity-Aware Rank/Select Dictionaries

- 41 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
- 42 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013.
- 43 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.
- 44 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.