A Learned Approach to Design Compressed Rank/Select Data Structures

ANTONIO BOFFA, PAOLO FERRAGINA, and GIORGIO VINCIGUERRA, University of Pisa, Italy

We address the problem of designing, implementing and experimenting with compressed data structures that support rank and select queries over a dictionary of integers. We shine a new light on this classical problem by showing a connection between the input integers and the geometry of a set of points in a Cartesian plane suitably derived from them. We then build upon some results in computational geometry to introduce the first compressed rank/select dictionary based on the idea of "learning" the distribution of such points via proper *linear approximations* (LA). We therefore call this novel data structure the `la_vector`.

We prove time and space complexities of the `la_vector` in several scenarios: in the worst case, in the case of input distributions with finite mean and variance, and taking into account the kth order entropy of some of its building blocks. We also discuss improved *hybrid* data structures, namely ones that suitably orchestrate known compressed rank/select dictionaries with the `la_vector`.

We corroborate our theoretical results with a large set of experiments over datasets originating from a variety of applications (Web search, DNA sequencing, information retrieval and natural language processing) and show that our approach provides new interesting space-time trade-offs with respect to many well-established compressed rank/select dictionary implementations. In particular, we show that our select is the fastest, and our rank is on the space-time Pareto frontier.

CCS Concepts: • **Theory of computation → Data compression**; **Predecessor queries**.

Additional Key Words and Phrases: Compressed data structures, Rank/Select dictionaries, Piecewise linear approximations, High order entropy, Algorithm engineering

ACM Reference Format:
Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A Learned Approach to Design Compressed Rank/Select Data Structures. *ACM Trans. Algor.* 1, 1, Article 1 (January 2022), 29 pages. https://doi.org/10.1145/3524060

1 INTRODUCTION

We consider the classical problem of representing, in compressed form, an ordered dictionary S of n elements drawn from the integer universe $[u] = \{0, \ldots, u-1\}$ while supporting the following operations:

- $\mathsf{rank}(x)$. Given $x \in [u]$, return the number of elements in S that are less than or equal to x;
- $\mathsf{select}(i)$. Given $i \in \{1, \ldots, n\}$, return the ith smallest element in S.

A preliminary version of this article appeared in [8]. The present contribution includes several new results as detailed at the end of Section 1.2.

Authors' address: Antonio Boffa, antonio.boffa@phd.unipi.it; Paolo Ferragina, paolo.ferragina@unipi.it; Giorgio Vinciguerra, giorgio.vinciguerra@phd.unipi.it, University of Pisa, Largo Bruno Pontecorvo 3, Pisa, Italy, 56127.

1549-6325/2022/1-ART1
https://doi.org/10.1145/3524060

Despite their simple definitions, rank and select are powerful enough to solve the ubiquitous *predecessor search problem* [47], which asks for the largest $y \in S$ smaller than a given element $x \in [u]$. Indeed, it suffices to execute $y = \mathsf{select}(\mathsf{rank}(x-1))$, where we assumed that $\mathsf{select}(0) = -1$ to denote the absence of a predecessor for x in S.

Another way of looking at these operations is via the indexing of a binary array B_S of length u which is the characteristic bitvector of S over the universe $[u]$. This way, $\mathsf{rank}(x)$ counts the number of bits set to 1 in $B_S[0\,..\,x]$, and $\mathsf{select}(i)$ finds the position of the ith bit set to 1 in B_S. This interpretation allows generalising the above operations to count and locate symbols in non-binary arrays [30, 33, 45], which are frequently at the core of several text mining and indexing problems.

It is therefore unsurprising that rank and select have been studied far and wide since the end of the '80s [35], with tons of important theoretical and practical results, which we review in Section 1.1. Currently, they are the building blocks of many compact data structures [44] used for designing compressed text indexes [20, 33, 45], succinct trees and graphs [42, 56], monotone minimal perfect hashing [7], sparse hash tables [58], and permutations [6]. Consequently, they have countless applications in bioinformatics [17, 38], information retrieval [43], and databases [1], just to mention a few.

In this paper, we show that the problem above has a surprising connection with the geometry of a set of points in the Cartesian plane suitably derived from the integers in S. We then build upon some classical results in computational geometry to introduce a novel data-aware compressed storage and indexing scheme for S that deploys *linear approximations* of the distribution of these points to "learn" a compact encoding of the input data. We call this novel data structure `la_vector` because its building blocks are Linear Approximations. We prove theoretical bounds on its time and space performance in the worst case, in the case of input distributions with finite mean and variance, and in terms of the kth order entropy of some of its building blocks. We show that the `la_vector` can be used in conjunction with other compression schemes, thus originating new hybrid data structures which compare favourably with [50]. Finally, we corroborate these theoretical results with a large set of experiments over a variety of real-world datasets and well-established approaches.

Overall, our theoretical and practical achievements are particularly interesting not only for novel space-time trade-offs, which add themselves to this highly-productive algorithmic field active since 1989 [44], but also because, we argue, they introduce a new way of designing compressed rank/select data structures which deploy computational geometry tools to "learn" the distribution of the input data [24]. As such, we foresee that this novel design may offer research opportunities and stimulate new results from which many applications will hopefully benefit.

1.1 Related work

We assume the standard word RAM model of computation with word size $w = \Theta(\log u)$ and $w = \Omega(\log n)$. Existing rank/select dictionaries differ by the way they encode S and how they use redundancy to squeeze the space and still support fast operations.

In the most basic case, S is represented via its characteristic bitvector B_S, namely a bitvector of length u such that $B_S[i] = 1$ if $i \in S$, and $B_S[i] = 0$ otherwise, for $0 \le i < u$. Then, $\mathsf{rank}(x)$ is the number of 1s in $B_S[0\,..\,x]$, and $\mathsf{select}(i)$ is the position of the ith 1 in B_S. One can also be interested in rank_0 and select_0, which look instead for the 0s in the bitvector, but it holds $\mathsf{rank}_0(i) = i - \mathsf{rank}(i)$, while select_0 can be reduced to select via other known reductions [55].

It is long known that $u + o(u)$ bits are sufficient to have constant-time rank and select [10, 41]. Provided that we keep B_S in plain form (i.e. read-only) and look for constant-time operations, the best that we can aim for the redundancy term $o(u)$ is $\Theta(u \log\log u / \log u)$ bits [29]. Later, optimal trade-offs were also given in terms of the density of 1s in B_S [31] or for the cell-probe model [54, 64].

Practical implementations of rank/select on plain bitvectors have also been extensively studied and evaluated experimentally [27, 28, 32, 46, 48, 60].

If S is sparse, i.e. B_S contains few 1s, then it may be convenient to switch to compressed representations. The information-theoretic minimum space to store S is $\mathcal{B} = \lceil \log \binom{u}{n} \rceil$, which may be much smaller than u. The value $\mathcal{B}$ is related to the (empirical) zero-order entropy of B_S, $H_0(B_S)$, defined as $uH_0(B_S) = n \log \frac{u}{n} + (u-n) \log \frac{u}{u-n}$. In fact, $\mathcal{B} = uH_0(B_S) - O(\log u)$. Here, the best upper bound on the redundancy was attained in [52], whose solution takes $\mathcal{B} + u/(\frac{\log u}{t})^t + O(u^{3/4} \log u)$ bits and supports both rank and select in $O(t)$ time, that is, constant-time operations in $\mathcal{B} + O(u/\text{poly} \log u)$ bits. This essentially matches the lower bounds provided in [54]. A widely known solution for a sparse S is the RRR encoding [56], which supports constant-time rank and select in $\mathcal{B} + O(u \log \log u / \log u)$ bits of space. We will experimentally compare our proposal with its practical implementations described in [11, 28]. There are also representations bounded by the kth order entropy of B_S, defined as $uH_k(B_S) = \sum_{x \in \{0,1\}^k} |B_x| H_0(B_x)$ where B_x is the bitvector concatenating the bits immediately following an occurrence of x in B_S. For example, the solution of [57] achieves constant-time operations in $uH_k(B_S) + O(u(\log \log u + k + 1)/\log u)$ bits.

In general, to further reduce the space, one has to give up the constant time for both operations. An example is given by the Elias-Fano representation [14, 15], which supports select in $O(1)$ time and rank in $O(\log \frac{u}{n})$ time while taking $n \log \frac{u}{n} + 2n + o(n)$ bits of space. Its implementations and refinements proved to be very effective in a variety of real-world contexts [48, 50, 51, 60, 61]. We will compare our `la_vector` against the best implementations to date [28, 50].

Another compressed representation for S is based on gap encoding. In this case, instead of $\mathcal{B}$ or the zero-order entropy, it is common to use more data-aware measures [4, 26, 34, 39, 57]. Consider the gaps g_i between consecutive integers in S taken in sorted order, i.e. $g_i = \text{select}(i) - \text{select}(i-1)$, and suppose we could store each g_i in $\lceil \log(g_i + 1) \rceil$ bits. Then the gap measure is defined as $gap(S) = \sum_i \lceil \log(g_i + 1) \rceil$. An example of a data-aware structure whose space occupancy is bounded in terms of gap is presented in [34], which takes $gap(S)(1 + o(1))$ bits while supporting select in $O(\log \log n)$ time and rank in time matching the optimal predecessor search bounds [47, 53]. Another example is given in [39] taking $gap(S) + O(n) + o(u)$ bits and supporting constant-time operations. Important ingredients of these gap-based data-aware structures are self-delimiting codes such as Elias γ- and δ-codes [62]. To provide a complete comparison with our `la_vector`, we will experiment with the practical approaches to gap compression implemented in the `sdsl` library [28].

Recent work [4] explored further interesting data-aware measures for bounding the space occupancy of rank/select dictionaries that take into account *runs* of consecutive integers in S. They introduced data structures supporting constant-time rank and select in a space bounded by these new data-aware measures. This proposal is mainly theoretical, and indeed the authors evaluated only its space occupancy. A more practical approach, described in [3, 5], combines gap and run-length encoding by fitting as many gaps g_i as possible within a single 32-bit word. This is done via a 4-bit header indicating how the remaining 28 bits must be decoded (e.g. one gap of 28 bits, two gaps of 14 bits each, etc.). We will compare our proposal also against this recent approach.

1.2 Our contribution

We introduce a novel lossless compressed storage scheme for an integer dictionary S based on the idea of approximating a set of points in the Cartesian plane via segments so that the storage of S can be defined by means of a compressed encoding of these segments and the "errors" they do in approximating the input integers (Section 2). Proper algorithms and data structures are then added to this compressed storage scheme to support fast rank and select operations (Section 3).

Our study shows that our approach is asymptotically efficient in time and space, with worst-case bounds that relate their performance with the number ℓ of segments approximating S and the (controlled) error ε. In particular, Theorems 2.2 and 2.3 state some interesting space bounds that are proven to be superior to the ones achievable by well-established Elias-Fano approaches for proper (and widely satisfied) conditions among n, ℓ and ε.

We extend these results also to the case of input sequences drawn from a distribution with finite mean and variance (Section 4). In this case, it turns out that our scheme is competitive with Elias-Fano approaches for $\varepsilon = \omega(\sqrt{\log n})$, which is a condition easily satisfied in practical settings [18].

Another theoretical contribution is the design of an algorithm that computes a provably-good approximation of the optimal set of segments which *minimises* the space occupancy of our compression scheme (Section 5).

We also consider hybrid solutions that optimally partition the datasets into chunks and apply the best encoding to each chunk. Consequently, we show that our approach can be used in conjunction with other known and effective compression schemes, yielding improved hybrid rank/select structures (Section 6).

We corroborate these theoretical results with a large set of experiments over datasets originating from a variety of sources and applications (the Web, DNA sequencing, information retrieval and natural language processing), and we show in Section 7 that our data-aware approach provides new interesting space-time trade-offs with respect to several other well-established implementations of Elias-Fano [27, 50], RRR-vectors [11, 27], random-access vectors of Elias γ/δ-coded gaps [28], and gap/run-length encoded bitvectors [5, 40]. Our select is the fastest, whereas our rank is on the space-time Pareto frontier.

For the sake of presentation, we summarise in Table 1 the main notation used throughout the paper. And, as a final remark, we note that a preliminary version of this work appeared in [8]. The present contribution includes several new results: the theoretical and experimental study of high-order compression of the aforesaid approximation "errors" constituting our encoding of S (Section 2.2); an improved data partitioning/compression algorithm (Theorem 5.2) that offers a very simple proof about the quality of the returned solution and also a practical improvement of 1.22% in space on average over all the datasets, without impairing the time and space complexity of its construction; the discussion and experimentation of a new improved hybrid data structure that combines our `la_vector` with existing rank/select dictionaries (Sections 6 and 7.5); an extended discussion of the algorithm-engineering tricks used in our implementations (Section 7.1); a more comprehensive experimental evaluation of the `la_vector` that includes other recently proposed rank/select dictionary implementations (Section 7.4).

Table 1. Summary of main notations used in the paper.

Symbol	Definition
S	Input set of integer elements
n	Number of integer elements in S
u	Size of the integer universe
B_S	Characteristic bitvector of S of size u and n 1s
C	Array of n corrections values of c bits each
c	Bits allotted to each correction ($0 \leq c \leq \log u$)
ε	Maximum absolute correction value, equal to $\max(0, 2^{c-1} - 1)$
ℓ	Number of segments (Definition 2.1)
s_j	The jth segment
r_j	Rank of the first element compressed by the segment s_j
α_j	Slope of the segment s_j
β_j	Intercept of the segment s_j
f_j	Linear function implemented by the segment s_j

2 COMPRESSING VIA LINEAR APPROXIMATIONS

Let us assume that $S = \{x_1, x_2, \ldots, x_n\}$ is a sorted sequence of n distinct integers. We begin by mapping each element $x_i \in S$ to a point (i, x_i) in the Cartesian plane, for $i = 1, 2, \ldots, n$ [2]. It is easy to see that any function f that passes through all the points in this plane can be thought of as an encoding of S because we can recover x_i by querying $f(i)$. Clearly, f should be fast to be computed and occupy little space.

Here, we aim at implementing f via a sequence of segments. Segments capture certain data patterns naturally. Any run of consecutive and increasing integers, for example, can be encoded by one segment with slope 1. Generalising, any run of increasing integers with a constant gap g can be encoded by one segment with slope g. Slight deviations from these data patterns can still be captured if we allow a segment to make some "errors" in approximating x_i at position i, provided that we fix these errors by storing some additional information.

This is the main idea behind our proposal. We reduce the problem of compressing S to the one of "learning" the mapping $\mathsf{select}\colon \{1, \ldots, n\} \to S$, which is in turn reduced to the problem of approximating the set of points $\{(i, x_i)\}_{i=1,\ldots,n}$ via a Piecewise Linear Approximation (PLA), that is a sequence of segments such that every point (i, x_i) is vertically far from one of these segments by an error bound ε, to be fixed later. In some sense, the sequence of segments introduces an "information loss" of ε on the integers in S. Among all such sequences of segments (i.e. PLAs), we further aim for the most succinct one, namely the one with the least amount of segments. This is a classical computational geometry problem that admits an $O(n)$-time algorithm by O'Rourke [49]. This algorithm processes each point (i, x_i) left-to-right, hence for $i = 1, \ldots, n$, while shrinking a convex polygon in the parameter space of slopes-intercepts. Any coordinate (α, β) inside the polygon represents a line with slope α and intercept β that approximates with error ε the current set of processed points. When the kth point causes the polygon to be empty, a segment (α, β) is chosen inside the previous polygon and returned, and a new polygon is started from (k, x_k).

We represent the jth segment output by the algorithm above as the triple $s_j = (r_j, \alpha_j, \beta_j)$, where α_j is the slope, β_j is the intercept, and r_j is the abscissa of the point that started the segment. If ℓ is the number of segments forming the PLA, we set $r_{\ell+1} = n$ and observe that $r_1 = 1$. The values r_js partition the set of positions $\{1, 2, \ldots, n\}$ into ℓ ranges so that, for any integer i between r_j and r_{j+1}

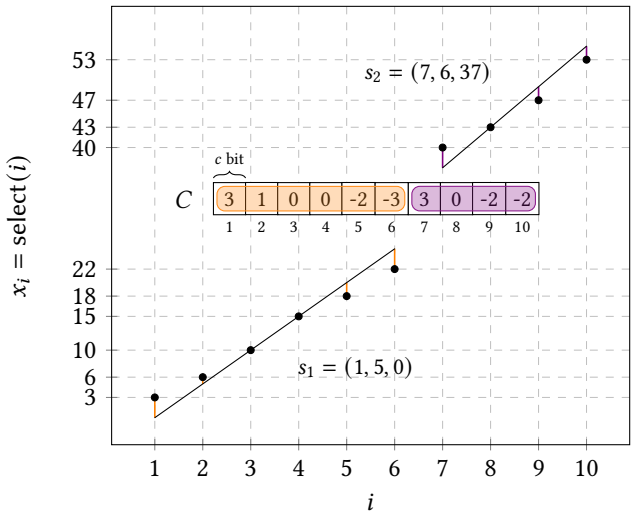


Fig. 1. The `la_vector` encoding of $S = \{3, 6, 10, 15, 18, 22, 40, 43, 47, 53\}$ for $c = 3$ is given by the two segments s_1, s_2 and the array C. A segment $s_j = (r_j, \alpha_j, \beta_j)$ approximates the value of an item with rank i via $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$, and C corrects the approximation. For example, $x_5 = \lfloor f_1(5) \rfloor + C[5] = 20 - 2 = 18$ and $x_8 = \lfloor f_2(8) \rfloor + C[8] = 43 + 0 = 43$.

(non-inclusive), we use the segment s_j to approximate the value x_i as follows:

$$f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j.$$

But $f_j(i)$ is an inexact approximation of x_i bounded by ε. Thus, in order to turn it into a lossless representation, we complement the values returned by f_j with an array $C[1\,..\,n]$ of integers whose modulo is bounded by ε. Precisely, each $C[i]$ represents the small "correction value" $x_i - \lfloor f_j(i) \rfloor$, which belongs to the set $\{-\varepsilon, -\varepsilon + 1, \ldots, -1, 0, 1, \ldots, \varepsilon\}$. If we allocate $c \geq 2$ bits for each correction in C, then the PLA is allowed to err by at most $\varepsilon = 2^{c-1} - 1$. We also consider the case $c = 0$, for which we set $\varepsilon = 0$. We ignore the case $c = 1$, because one bit is not enough to distinguish corrections in $\{-1, 0, 1\}$.

The vector C completes our encoding, which we name *linear approximation vector* (`la_vector`) and illustrate in Figure 1. Recovering the original sequence S is as simple as scanning the segments s_j of the PLA and writing the value $\lfloor f_j(i) \rfloor + C[i] = x_i$ to the output, for $j = 1, \ldots, \ell$ and for $i = r_j, \ldots, r_{j+1} - 1$. This process, formalised in Algorithm 1, is appealing in practice because the array C contains tightly-packed integers that are accessed sequentially, and the computation of f_j is fast because its values are loaded into three registers when s_j is first accessed. Moreover, there are no data dependencies among the iterations (as it happens for example when integers are delta-coded and a prefix sum is needed).

Recovering a single integer x_i requires first the identification of the segment s_j that includes the position i, and then the computation of $\lfloor f_j(i) \rfloor + C[i]$. A binary search over the starting positions r_j of the segments in the PLA would be enough and takes $O(\log \ell)$ time, but we will aim for something more sophisticated in terms of algorithmic design and engineering to squeeze the most from this novel approach, as commented in the following sections.

For completeness, we observe that the PGM-index [25] might appear similar to this idea because it uses PLAs and supports predecessor queries. However, the PGM-index does not compress the input

Algorithm 1 Decompression

Input: PLA $\{s_1, \ldots, s_\ell\}$, corrections $C[1\,..\,n]$
Output: Uncompressed set S

```
out ← an empty array of size n
for all segments s_j = (r_j, α_j, β_j) in the PLA do
  for i ← r_j to r_{j+1} − 1 do
    out[i] ← ⌊f_j(i)⌋ + C[i], where f_j = (i − r_j) · α_j + β_j
return out
```

keys but only the index, and it is tailored to the external-memory model, as B-trees. Nonetheless, the PGM-index could take advantage of the `la_vector` to compress the data stored in its leaves.

2.1 On compression effectiveness

Two counterpoising factors influence the effectiveness of the compressed space occupancy of the `la_vector`.

(1) How the integers in S map on the Cartesian plane, and thus how many segments they require for a lossy ε-approximation. The larger is ε, the smaller is "expected" to be the number ℓ of these segments.
(2) The value of the parameter $c \geq 0$, which determines the space occupancy of the array C, having size nc bits. From above, we know that $\varepsilon = \max(0, 2^{c-1} - 1)$, so the smaller is c, the smaller is the space occupancy of C, but the larger is "expected" to be the number ℓ of segments of the PLA built for S.

We say "expected" because ℓ depends on the distribution of the points (i, x_i) on the Cartesian plane. In the best scenario, the points lie on one line, so $\ell = 1$ and we can set $c = 0$. The more these points follow a linear trend, the smaller c can be chosen and, in turn, the smaller is the number ℓ of segments approximating these points with error ε. Although in the worst case it holds $\ell \leq \min\{u/(2\varepsilon), n/2\}$, because of a simple adaptation of [25, Lemma 2], we will show in Section 4 that for sequences drawn from a distribution with finite mean and variance there are tighter bounds on ℓ. This leads us to argue that the combination of the PLA and the array C, on which the storage scheme of the `la_vector` hinges upon, is an interesting algorithmic tool to design novel compressed rank/select dictionaries.

At this point, it is useful to formally define the interplay among S, c and ℓ. We argue in this paper that the number ℓ of segments of the optimal PLA (namely the one using the smallest ℓ) can be thought of as a new compressibility measure for the information present in S, possibly giving some insights (such as the degree of approximate linearity of the data) that the classical entropy measures do not explicitly capture. In the following, we assume $c \leq \log u$ to avoid the case in which nc exceeds the $O(n \log u)$ bits needed by an explicit representation of S.

Definition 2.1. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a sorted sequence of n distinct integers drawn from the universe $[u]$. Given an integer parameter $c \in \{0, \ldots, \log u\}$, we define ℓ as the number of segments which constitute the optimal PLA of maximum error $\varepsilon = \max(0, 2^{c-1} - 1)$ computed on the set of points $\{(i, x_i) \mid i = 1, \ldots, n\}$.

We are ready to compute the space taken by the `la_vector`. As far as the representation of a segment $s_j = (r_j, \alpha_j, \beta_j)$ is concerned, we note that: (i) the value r_j is an abscissa in the Cartesian plane, thus it can be represented in $\log n$ bits;[1] (ii) the slope α_j can be encoded as a rational number

[1]For ease of exposition, we assume that logarithms hide their ceiling and thus return integers.

with a numerator of $\log u$ bits and a denominator of $\log n$ bits [49, 63]; (iii) the intercept β_j is an ordinate in the plane, thus it can be represented in $\log u$ bits. Therefore, the overall cost of the PLA is $2\ell(\log n + \log u)$ bits. Summing the nc bits taken by C gives our first result.

Theorem 2.2. *Let S be a set of n integers drawn from the universe $[u]$. Given integers c and ℓ as in Definition 2.1, a plain implementation of the* `la_vector` *takes $nc + 2\ell(\log n + \log u)$ bits of space.*

We can further improve the space taken by the segments as follows. The r_js form an increasing sequence of ℓ positive integers bounded by n. The β_js form an increasing sequence of ℓ positive integers bounded by u.[2] Using the Elias-Fano representation [44, §4.4], we reduce the space of the two sequences to $\ell \log \frac{n}{\ell} + \ell \log \frac{u}{\ell} + 4\ell + o(\ell) = \ell(\log \frac{un}{\ell^2} + 4 + o(1))$ bits. Then, accessing r_j or β_j amounts to calling the constant-time select(j) on the corresponding Elias-Fano compressed sequence. Summing the nc bits taken by C and the $\ell(\log n + \log u)$ bits taken by the α_js gives our second result.

Theorem 2.3. *Let S be a set of n integers drawn from the universe $[u]$. Given integers c and ℓ as in Definition 2.1, there exists a more compressed version of the* `la_vector` *that takes $nc + \ell(2 \log \frac{un}{\ell} + 4 + o(1))$ bits of space.*

Finally, we mention the existence of a lossless compressor for the α_js that can be beneficial when multiple segments share the same or similar slope [25, Theorem 3].

2.2 Entropy-coding the corrections

In this section, we show how to further reduce the space of the `la_vector` by entropy-coding the vector of corrections C.

Regard C as a string of length n from an integer alphabet $\Sigma = \{-\varepsilon, -\varepsilon + 1, \ldots, \varepsilon\}$, and let n_x denote the number of occurrences of a symbol x in C. The zero-order entropy of C is defined as

$$H_0(C) = \sum_{x \in \Sigma} \frac{n_x}{n} \log \frac{n}{n_x}.$$

The value $nH_0(C)$ is the output size of an ideal compressor that uses $-\log \frac{n_x}{n}$ bits for coding the symbol x unambiguously [13, 36]. In order to further squeeze the output size, one could take advantage not only of the frequency of symbols but also of their preceding context in C. Let C_y be the string of length $|C_y|$ that concatenates all the single symbols following each occurrence of a context y inside C. The kth order entropy of C is defined as

$$H_k(C) = \frac{1}{n} \sum_{y \in \Sigma^k} |C_y| \, H_0(C_y).$$

A well-known data structure achieving zero-order entropy compression is the wavelet tree [33] with the bitvectors stored in its nodes compressed using RRR [56]. Considering the `la_vector` of Theorem 2.3 but compressing C via this approach (see also [45, Theorem 8]), we obtain:

Theorem 2.4. *Let S be a set of n integers drawn from the universe $[u]$. Given integers c and ℓ as in Definition 2.1, there exists a zero-order entropy-compressed version of the* `la_vector` *for S that takes $nH_0(C) + o(nc) + \ell(2 \log \frac{un}{\ell} + 4 + o(1))$ bits of space, and $O(c)$ time to access a position in C, where C is the vector of corrections.*

[2] This is because β_j is the ordinate where s_j starts, i.e. $\beta_j = f_j(r_j)$ (see Figure 1 and the definition of f_j). In the text, we referred to β_j as the "intercept", but this is improper because β_j is not the ordinate of the intersection between f_j and the y-axis.

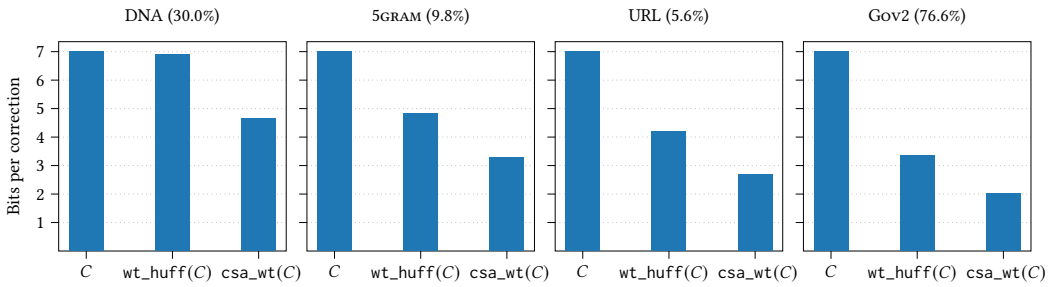


Fig. 2. The space needed by the plain corrections C, zero-order entropy-coded corrections $\texttt{wt\_huff}(C)$, and the high-order entropy-coded corrections $\texttt{csa\_wt}(C)$ in an `la_vector` with $c = 7$. Next to each dataset name, we show the density value n/u as a percentage.

A well-performing high-order entropy-compressed data structure over strings drawn from an integer alphabet is the alphabet-friendly FM-index [21, 22]. Using the alphabet-friendly FM-index to store C, we obtain:

THEOREM 2.5. *Let S be a set of n integers drawn from the universe $[u]$. Given integers c and ℓ as in Definition 2.1, there exists a kth order entropy-compressed version of the* `la_vector` *for S that takes $nH_k(C) + o(nc) + \ell(2\log\frac{un}{\ell} + 4 + o(1))$ bits of space, and $O(c(\log^{1+\tau} n)/\log\log n)$ time to access a position in C, where C is the vector of corrections, and $\tau > 0$ is an arbitrary constant.*

To get a practical sense of the real compression achieved by the above two entropy-compressed versions of the `la_vector`, we compare experimentally the space taken by the uncompressed corrections (as adopted in the plain `la_vector`) with the space taken by (i) a Huffman-shaped wavelet tree with RRR-compressed bitvectors on C (implementing the solution in Theorem 2.4), and (ii) a compressed suffix array based on a Huffman-shaped wavelet tree with RRR-compressed bitvectors on the Burrows-Wheeler Transform of C (implementing the solution in Theorem 2.5). We denote the space taken by these two choices by $\texttt{wt\_huff}(C)$ and $\texttt{csa\_wt}(C)$, respectively, given the name of the corresponding classes in the `sdsl` library [28]. For $\texttt{csa\_wt}(C)$, we do not take into account the space taken by the sampled suffix array because we do not need to support the locate query, which returns the list of positions in C where a given pattern string occurs. Rather, to get individual corrections from C, we need the sampled inverse suffix array, which indeed we store and account for in the space occupancy of $\texttt{csa\_wt}(C)$.

Figure 2 shows the results with a value $c = 7$ on four real-world datasets, described in detail in Section 7. For the DNA dataset, there is no significant difference between the plain corrections and the zero-order entropy-coder $\texttt{wt\_huff}(C)$. Instead, the high-order entropy-coder $\texttt{csa\_wt}(C)$ is 33% smaller. For the other three datasets (5GRAM, URL, and GOV2), both $\texttt{wt\_huff}(C)$ and $\texttt{csa\_wt}(C)$ are up to 56% and 72% smaller than the plain corrections, respectively. This shows that there is some statistical redundancy within the array of corrections C that the `la_vector` could deploy to squeeze its space occupancy further.

Another important issue to investigate concerns the impact on the compression of C that is induced by changing the slopes of the segments in the optimal PLA computed for `la_vector`. Intuitively, as depicted in Figure 3, different slope-intercept pairs satisfying the same ε-bound generate different vectors C with different entropies. As a consequence, instead of picking a random slope-intercept pair within the ones that are ε-approximation, one can choose the slope-intercept pair minimising the entropy of C.

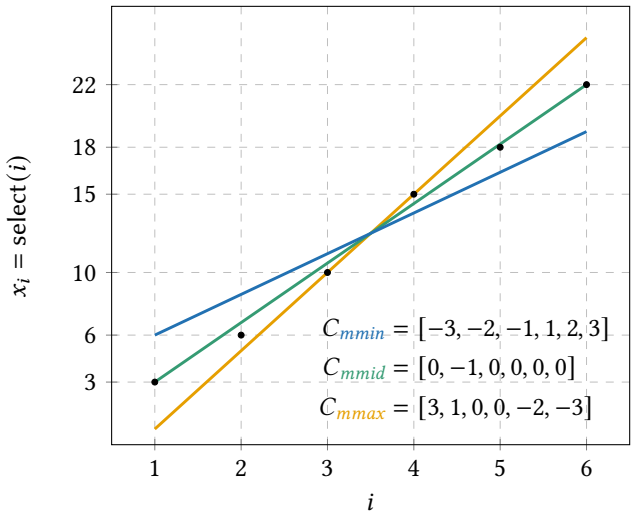


Fig. 3. Three possible slopes, *mmin*, *mmid* and *mmax*, for a segment encoding the set $S = \{3, 6, 10, 15, 18, 22\}$ with $c = 3$. Each slope generates a different vector C with a different entropy: $H_0(C_{mmin}) = 2.58$, $H_0(C_{mmid}) = 0.65$, and $H_0(C_{mmax}) = 2.25$.

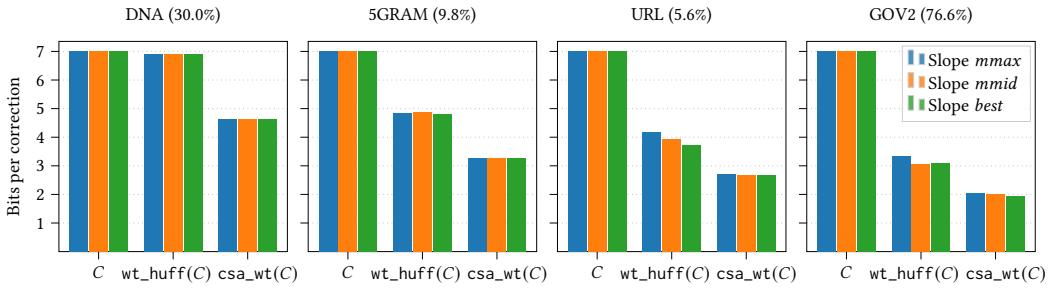


Fig. 4. A different choice of the slope of the segments in an `la_vector` may yield a reduced space occupancy of the entropy-coded correction vector C. Here we show three choices: *mmax*, *mmid* and *best* (see Section 2.2).

For the experiment in Figure 2, we adopted the strategy that always chooses the maximum slope among the ε-approximate segments for S. In Figure 4, we compare this strategy, which we call *mmax*, with two other strategies: (i) *mmid*, which chooses the average slope between the smallest and the largest feasible slopes, and (ii) *best*, a heuristic that selects nine slopes at regular intervals between the smallest and the largest feasible slopes and picks the one minimising $H_0(C)$. For the DNA and 5GRAM datasets, there is no noticeable improvement in changing the slope of the segments of the `la_vector`. Instead, for URL and Gov2, changing the slope of each segment from *mmax* to *mmid* or *best* reduces $H_0(C)$. Of course, since the choice *best* targets only the zero-order entropy of the corrections, the plots show little or no reduction of $H_k(C)$.

To sum up, we can further reduce the space occupancy of the `la_vector` by entropy-coding its correction vector C. This reduction is particularly interesting in applications in which the `la_vector` is used as an archival method, that is, when efficient random access and queries are not required. In the following, we concentrate on how to support efficient select and rank queries over S, so we explore the variant of the `la_vector` in which C is kept uncompressed.

3 SUPPORTING SELECT AND RANK

To answer select(i) on the `la_vector` (either on the plain implementation of Theorem 2.2 or the compressed implementation of Theorem 2.3), we build a *predecessor* structure $\mathcal{D}$ on the set $R = \{r_j \mid 1 \le j \le \ell\}$ and proceed in three steps. First, we use $\mathcal{D}$ to retrieve the segment s_j in which i falls into via $j = \text{pred}(i)$. Second, we compute $f_j(i)$, i.e. the approximate value of x_i given by the segment s_j. Third, we read $C[i]$ and return the value $\lfloor f_j(i) \rfloor + C[i]$ as the answer to select(i). The last two steps take $O(1)$ time. Treating $\mathcal{D}$ as a black box yields the following result.

Lemma 3.1. *The* `la_vector` *supports select queries in $t + O(1)$ time and b bits of additional space, where t is the query time and b is the occupied space of a predecessor structure $\mathcal{D}$ constructed on a set of ℓ integers over the universe $[n]$.*

If $\mathcal{D}$ is represented as the characteristic bitvector of the set R augmented with a data structure supporting constant-time predecessor queries (or rank queries, as termed in the case of bitvectors [44]), then we achieve constant-time select by using only $n + o(n)$ additional bits, i.e. about one bit per integer of S more than what Theorem 2.2 requires. Note that this bitvector encodes R, so that the $\ell \log n$ bits required in Theorem 2.2 for the representation of the r_js can be dropped.

Corollary 3.2. *Let S be a set of n integers drawn from the universe $[u]$. Given integers c and ℓ as in Definition 2.1, there exists a compressed representation of the* `la_vector` *for S that takes $n(c + 1 + o(1)) + \ell(2 \log u + \log n)$ bits of space while supporting select in $O(1)$ time.*

Let us compare the space occupancy achieved by the compressed `la_vector` of Corollary 3.2 to the one of Elias-Fano, namely $n(\log \frac{u}{n} + 2) + o(n)$ bits [44, §4.4], as both solutions support constant-time select. The inequality turns out to be

$$\ell \le \frac{n\,(\log \frac{1}{d} + o(1))}{2 \log \frac{n}{d} + \log n} = O\left(\frac{n}{\log n}\right),$$

where $d = n/u$ denotes the density of 1s in B_S.

To solve rank(x), it would be sufficient to perform a binary search on the interval $[1, n]$ to find the largest i such that select$(i) \le x$. This naïve implementation takes $O(t \log n)$ time, because of the implementation of select in $O(t)$ time by Lemma 3.1.

We can easily improve this solution to $O(\log \ell + \log n)$ time as follows. First, we binary search on the set of ℓ segments to find the segment s_j that contains x or its predecessor. Formally, we binary search on the interval $[1, \ell]$ to find the largest j such that select$(r_j) = \lfloor f_j(r_j) \rfloor + C[r_j] \le x$. Second, we binary search on the $r_{j+1} - r_j \le n$ integers compressed by segment s_j to find the largest i such that $\lfloor f_j(i) \rfloor + C[i] \le x$. Finally, we return i as the answer to rank(x).

Surprisingly, we can further speed up rank queries without adding any redundancy on top of the encoding of Theorem 2.2. The key idea is to narrow down the second binary search to a subset of the elements covered by s_j (i.e. a subset of the ones in positions $[r_j, r_{j+1} - 1]$), which is determined by exploiting the fact that s_j approximates all these elements by up to an additive term ε. Technically, we know that $|f_j(i) - x_i| \le \varepsilon$, and we aim to find i such that $x_i \le x < x_{i+1}$. Hence, we can narrow the range to those $i \in [r_j, r_{j+1} - 1]$ such that $f_j(i) - \varepsilon \le x < f_j(i + 1) + \varepsilon$. By expanding $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$ and noting that f is linear and increasing, we get all candidate i as the ones satisfying

$$(i - r_j) \cdot \alpha_j + \beta_j - \varepsilon \le x < (i + 1 - r_j) \cdot \alpha_j + \beta_j + \varepsilon.$$

By solving for i, we get

$$\frac{x - \beta_j}{\alpha_j} + r_j - \left(\frac{\varepsilon}{\alpha_j} + 1\right) < i \le \frac{x - \beta_j}{\alpha_j} + r_j + \frac{\varepsilon}{\alpha_j}.$$

Algorithm 2 Rank implementation by Lemma 3.3

Input: Value x, PLA $\{s_1, s_2, \ldots, s_\ell\}$, corrections $C[1..n]$
Output: Returns $\mathsf{rank}(x)$
1: Find max $j \in [1, \ell]$ such that $\lfloor f_j(r_j) \rfloor + C[r_j] \leq x$ by binary search
2: $pos \leftarrow \lfloor (x - \beta_j)/\alpha_j \rfloor + r_j$
3: $err \leftarrow \lceil \varepsilon/\alpha_j \rceil$, where $\varepsilon = \max(0, 2^{c-1} - 1)$
4: $lo \leftarrow \max\{pos - err, r_j\}$
5: $hi \leftarrow \min\{pos + err, r_{j+1}\}$
6: Find max $i \in [lo, hi]$ such that $\lfloor f_j(i) \rfloor + C[i] \leq x$ by binary search
7: **return** i

Since i is an integer, we can round the left and the right side of the last inequality, and then we set $pos = \lfloor (x - \beta_j)/\alpha_j \rfloor + r_j$ and $err = \lceil \varepsilon/\alpha_j \rceil$, so that the searched position i falls in $[pos - err, pos + err]$.

The pseudocode of Algorithm 2 exploits these ideas to perform a binary search on the first integers compressed by the segments (Line 1), to compute the approximate rank and the corresponding approximation error (Lines 2–3), and finally to binary search on the restricted range specified above (Lines 4–6). As a final note, we observe that $\alpha_j \geq 1$ for every j, because the elements in S are increasing, and thus the segments have a slope of at least 1. Consequently, $\varepsilon/\alpha_j \leq \varepsilon$ and the range on which we perform the second binary search has size $2\varepsilon < 2^c$, thus this second binary search takes $O(\log \frac{\varepsilon}{\alpha_j}) = O(c)$ time.

Lemma 3.3. *The* `la_vector` *supports rank queries in* $O(\log \ell + c)$ *time and no additional space.*

Note that Lemma 3.3 applies to: (i) the plain `la_vector` representation provided in Theorem 2.2 (ii) the compressed `la_vector` representation provided in Theorem 2.3 (the one that compresses β_js and r_js), (iii) the representation provided in Lemma 3.1 (the one supporting select in parametric time t), and (iv) the representation provided in Corollary 3.2 (the one supporting select in constant time).

We can improve the bound of Lemma 3.3 by replacing the binary search at Line 1 of Algorithm 2 with the following predecessor data structure.

Lemma 3.4 ([53]). *Given a set Q of q integers over a universe of size u, let us define $a = \log \frac{s \log u}{q}$, where $s \log u$ is the space usage in bits chosen at building time. Then, the optimal predecessor search time is*

$$\begin{aligned} PT(u, q, a) = \Theta(\min\{ & \log q / \log\log u, \\ & \log \tfrac{\log(u/q)}{a}, \\ & \log \tfrac{\log u}{a} \,/ \log(\tfrac{a}{\log q} \cdot \log \tfrac{\log u}{a}), \\ & \log \tfrac{\log u}{a} \,/ \log(\log \tfrac{\log u}{a} \,/\, \log \tfrac{\log q}{a})\}). \end{aligned}$$

Let $T = \{\mathsf{select}(r_j) \mid 1 \leq j \leq \ell\}$ be the subset of S containing the first integer covered by each segment. We sample one element of T out of $\Theta(2^c)$ and insert the samples into the predecessor data structure of Lemma 3.4 so that $s = q = \ell/2^c$ and thus $a = \log\log u$. Then, we replace Line 1 of Algorithm 2 with a predecessor search followed by an $O(c)$-time binary search in-between two samples.

Corollary 3.5. *The* `la_vector` *supports rank queries in* $PT(u, \ell/2^c, \log\log u) + O(c)$ *time and* $O((\ell/2^c) \log u)$ *bits of additional space.*

We can restrict our attention to the first two branches of the min-formula describing the PT term in Lemma 3.4, as the latter two are instead relevant for universe sizes that are super-polynomial in q, i.e. $\log u = \omega(\log q)$. The time complexity of rank in Corollary 3.5 then becomes $O(\min\{\log_w \frac{\ell}{2^c}, \log\log \frac{u}{\ell}\} + c)$, where $w = \Omega(\log u)$ is the word size of the machine.

4 SPECIAL SEQUENCES

For input sequences drawn from a distribution with finite mean and variance there exist bounds on the number of segments ℓ, as stated in the following theorem adapted from [19].

Theorem 4.1 ([19]). *Let S be a set of n integers drawn from the universe $[u]$. Suppose that the gaps between consecutive integers in S are a realisation of a random process consisting of positive, independent and identically distributed random variables with mean μ and variance σ^2. Given the integers ε and ℓ as in Definition 2.1, if ε is sufficiently larger than σ, then $\ell = n\sigma^2/\varepsilon^2$ with high probability.*

Plugging this result into the constant-time select of Corollary 3.2 and the rank implementation of Lemma 3.3, we obtain the following result.

Theorem 4.2. *Under the assumptions of Theorem 4.1, there exists a compressed version of the* `la_vector` *for S that supports select in $O(1)$ time and rank in $O(\log \ell + c)$ time within $n[c + 1 + (2\log u + \log n)\frac{\sigma^2}{\varepsilon^2} + o(1)]$ bits of space with high probability.*

We stress the fact that the data structure of Theorem 4.2 is deterministic. In fact, the randomness is over the gaps between consecutive integers of the input data, and the result holds for any probability distribution as long as the mean and variance are finite. Moreover, according to the experiments in [19], the hypotheses of Theorem 4.1 are very realistic in several applicative scenarios.

Having said that, we observe that the hypothesis "ε is sufficiently larger than σ" implies that the ratio σ/ε is much smaller than 1. Hence, it is reasonable to assume that the space bound in Theorem 4.2 is dominated by the term $n(c + 1)$ which is independent of the universe size while still ensuring constant time select and fast rank operations. If we compare the factor $c + 1$ present in the space bound of the `la_vector` with the factor $\log \frac{u}{n}$ present in the space bound of Elias-Fano, we notice that the latter gets larger as the data is sparse ($n \ll u$). On the other hand, the time complexity of select is constant in both cases, whereas our rank is faster whenever $\log(n\sigma^2)$ is asymptotically smaller than $\log \frac{u}{n}$, which is indeed for $u = \omega(n^2\sigma^2)$.[3]

In general terms, some results of the previous sections, such as Corollary 3.2 and Lemma 3.3, showed that our `la_vector` is better than Elias-Fano whenever $\ell = O(n/\log n)$. Since Theorem 4.1 proves that $\ell = \Theta(n/\varepsilon^2)$ for a large class of input sequences, we can derive that for such sequences our solution is better than Elias-Fano if $\varepsilon = \omega(\sqrt{\log n})$.

5 ON OPTIMAL DATA PARTITIONING TO IMPROVE SPACE

So far, we assumed a fixed number of bits $c \geq 0$ for each of the n corrections in the `la_vector`, which is equivalent to saying that the ℓ segments in the PLA guarantee the same error $\varepsilon = \max(0, 2^{c-1} - 1)$ over all the integers in the input set S. However, the input data may exhibit a variety of regularities that allow to compress it further if we use a different c for different partitions of S. The idea of partitioning data to improve its compression has been studied in the past [9, 23, 50, 59, 62], and it will be further developed in this section with regard to our piecewise linear approximations.

We reduce the problem of minimising the space of our rank/select dictionary to a single-source shortest path problem over a properly defined weighted Directed Acyclic Graph (DAG) $\mathcal{G}$ defined

[3]Here we are considering the rank implementation of Lemma 3.3, taking $O(\log \ell + c) = O(\log(n\sigma^2))$ time, but an improved analysis can be obtained by deploying the rank implementation of Corollary 3.5.

as follows. The graph has n vertices, one for each element in S, plus one sink vertex denoting the end of the sequence. An edge (i, j) of weight $w(i, j, c)$ indicates that there exists a segment compressing the integers $x_i, x_{i+1}, \ldots, x_{j-1}$ of S by using $w(i, j, c) = (j - i)\, c + \kappa$ bits of space, where c is the bit-size of the corrections, and κ is the space taken by the segment representation in bits (e.g. using the plain encoding of Theorem 2.2 or the compressed encoding of Theorem 2.3). We consider all the possible values of c except $c = 1$, because one bit is not enough to distinguish corrections in $\{-1, 0, 1\}$. Namely, we consider $c \in \{0, 2, 3, \ldots, c_{max}\}$, where $c_{max} = O(\log u)$ is defined as the correction size that produces one single segment on S. Since each vertex is the source of at most c_{max} edges, one for each possible value of c, the total number of edges in $\mathcal{G}$ is $O(n\, c_{max}) = O(n \log u)$. It is not difficult to prove the following:

Fact 5.1. *The shortest path from vertex 1 to vertex $n + 1$ in the weighted DAG $\mathcal{G}$ defined above corresponds to the PLA for S whose cost is the minimum among the PLAs that use a different error ε on different segments.*

Fact 5.1 provides a solution to the rank/select dictionary problem which minimises the space occupancy of the approaches stated in Theorems 2.2 and 2.3.

Since $\mathcal{G}$ is a DAG, the shortest path can be computed in $O(n \log u)$ time by taking the vertices in topological order and by relaxing their outgoing edges [12, §24.2]. However, one cannot approach the construction of $\mathcal{G}$ in a brute-force manner because this would take $O(n^2 \log u)$ time and $O(n \log u)$ space, as each of the $O(n \log u)$ edges requires computing a segment in $O(j - i) = O(n)$ time with the algorithm of O'Rourke [49].

To avoid this prohibitive cost, we propose an algorithm that computes a solution on the fly by working on a properly defined graph $\mathcal{G}'$ derived from $\mathcal{G}$, taking $O(n \log u)$ time and $O(n)$ space. This reduction in both time and space complexity is crucial to make the approach feasible in practice. Moreover, we will see that the obtained solution is not "too far" from the one given by the shortest path in $\mathcal{G}$.

Consider an edge (i, j) of weight $w(i, j, c)$ in $\mathcal{G}$, which corresponds to a segment compressing the integers $x_i, x_{i+1}, \ldots, x_{j-1}$ of S by using $w(i, j, c)$ bits of space. Clearly, the same segment compresses any subsequence $x_a, x_{a+1} \ldots, x_{b-1}$ of $x_i, \ldots, x_{j-1}$ still using c bits per correction. Therefore, the edge (i, j) "induces" sub-edges of the kind (a, b), where $i \leq a < b \leq j$, of weight $w(a, b, c)$. We observe that the edge (a, b) may not be an edge of $\mathcal{G}$ because a segment computed from position a with correction size c could end past b, thus including more integers on its right. Nonetheless this property is crucial to define our graph $\mathcal{G}'$.

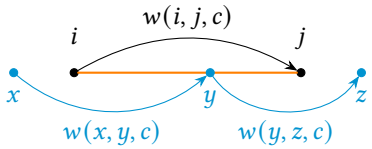
The vertices of $\mathcal{G}'$ are the same as the ones of $\mathcal{G}$. For the edges of $\mathcal{G}'$, we start from the subset of edges of $\mathcal{G}$ that correspond to the segments in the PLAs built for the input set S for all the values of $c = 0, 2, 3, \ldots, c_{max}$. We call these, the *full edges* of $\mathcal{G}'$. Then, for each full edge (i, j), we generate the *prefix edge* $(i, i + k)$ and the *suffix edge* $(i + k, j)$, for all $k = 1, \ldots, j - i$. This means that we are "covering" every full edge with all of its possible "splits" in two shorter edges having the same correction c as (i, j). The total size of $\mathcal{G}'$ is still $O(n \log u)$.

We are now ready to show that the graph $\mathcal{G}'$ has a path whose weight is just an *additive* term far from the weight of the shortest path in $\mathcal{G}$. (Notice that this contrasts with the approaches that obtain a multiplicative approximation factor [23, 50].)

Lemma 5.1. *There exists a path in $\mathcal{G}'$ from vertex 1 to vertex $n + 1$ whose weight is at most $\kappa\ell$ bits larger (in an additive sense) than the weight of the shortest path in $\mathcal{G}$, where κ is the space taken by a segment in bits, and ℓ is the number of edges in the shortest path of $\mathcal{G}$.*

Proof. We show that a generic edge (i, j) of weight $w(i, j, c) = (j-i)c+\kappa$ in $\mathcal{G}$ can be decomposed into at most two edges of $\mathcal{G}'$ whose total weight is at most $w(i, j, c) + \kappa$. The statement will then follow by recalling that ℓ is the number of edges in the shortest path of $\mathcal{G}$.

Consider the PLA for S with the same correction size c as (i, j). This PLA has surely one segment that either starts from i or overlaps i. In the former case we are done because the segment corresponds to the edge (i, j), which appears in $\mathcal{G}'$ as a full edge. In the latter case, the segment corresponds to a full edge (x, y) such that $x < i < y < j$, and it is followed by a segment that corresponds to a full edge (y, z) such that $z > j$, as shown in the following picture. In fact, y cannot occur after j otherwise the segment corresponding to the edge (i, j) would be longer, because the length of a segment in a PLA is maximised.



Given this situation, we decompose the edge (i, j) of $\mathcal{G}$ into: the suffix edge (i, y) of (x, y), and the prefix edge (y, j) of (y, z). Both edges (i, y) and (y, j) belong to $\mathcal{G}'$ by construction, they have correction size c, and their total weight is $w(i, y, c)+w(y, j, c) = (y-i)c+\kappa+(j-y)c+\kappa = (j-i)c+2\kappa$. Since $w(i, j, c) = (j-i)c+\kappa$, the previous total weight can be rewritten as $w(i, j, c)+\kappa$, as claimed. □

We now describe an algorithm that computes the shortest path in $\mathcal{G}'$ without generating the full graph $\mathcal{G}$ but expanding $\mathcal{G}'$ incrementally so to use $O(n)$ working space. The algorithm processes the vertices of $\mathcal{G}$ from left to right, while maintaining the following invariant: for $i = 1, \ldots, n + 1$, each processed vertex i is covered by one segment for each correction size c, and all these segments form the frontier set J.

We begin from vertex $i = 1$ and compute the c_{max} segments that start from i and have any possible correction size $c = 0, 2, 3, \ldots, c_{max}$. We set J as the set of these segments. As in the classic step of the shortest path algorithm for DAGs, we do a relaxation step on all the (full) edges (i, j), where j is the set of ending positions of the segments in J, that is, we test whether the shortest path to j found so far can be improved by going through i (initially, the shortest-path estimates are set to ∞ for each vertex) and update such shortest path accordingly [12, §24.2]. This completes the first iteration.

At a generic iteration i, we first check whether there is a segment in J that ends at i. If so, we replace that segment with the longest segment starting at i and using the same correction size, computed as usual using the algorithm of O'Rourke [49]. Afterwards, for each full edge (a, b) that corresponds to a segment in J, we first relax the set of prefix edges of the kind (a, i), then we relax the set of suffix edges of the kind (i, b). This is depicted in Figure 5.

Theorem 5.2. *There exists an algorithm that in $O(n \log u)$ time and $O(n)$ space outputs a path from vertex 1 to vertex $n + 1$ whose weight is at most $\kappa\ell$ bits larger (in an additive sense) than the shortest path of $\mathcal{G}$, where κ is the space taken by a segment in bits, and ℓ is the number of edges in the shortest path of $\mathcal{G}$.*

Proof. It is easy to see that the algorithm finds the shortest path in $\mathcal{G}'$. Indeed, it computes and relaxes: (i) the full edges of $\mathcal{G}'$ corresponding to the segments in a PLA with correction size c when updating the frontier set J; and (ii) all prefix (resp. suffix) edges ending (resp. beginning) at a vertex i when this vertex is processed. Therefore, the algorithm relaxes all the edges of $\mathcal{G}'$ and, according to Lemma 5.1, it finds a path whose weight is the claimed one.

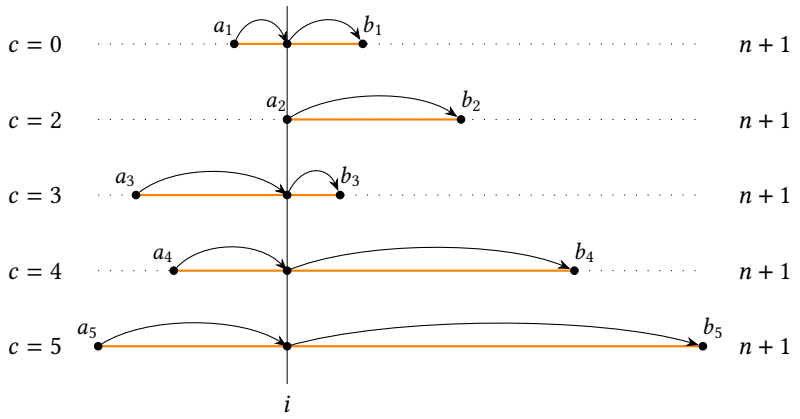


Fig. 5. The algorithm of Theorem 5.2 keeps a frontier with the segments (in orange) crossing the processed vertex i for each value of the correction size c (here $c_{max} = 5$). For each segment with endpoints a_c and b_c, which corresponds to a full edge (a_c, b_c) with correction size c, the algorithm relaxes the prefix edge (a_c, i) and the suffix edge (i, b_c).

As far as the space occupancy is concerned, the algorithm uses $O(n + |J|) = O(n + \log u) = O(n)$ space at each iteration, since the size of the frontier set is $|J| = c_{max} = O(\log u)$. The running time is $O(|J|) = O(\log u)$ per iteration, plus the cost of replacing a segment in J when it ends before the processed vertex, i.e. the cost of computing a full edge. This latter cost is $O(n)$ time for any given value of c and over all n elements (namely, it is $O(1)$ amortised time per processed element [49]), thus $O(n \log u)$ time over all the values of c. □

Given Theorem 5.2, the PLA computed by our algorithm can be used to design a rank/select dictionary which minimises the space occupancy of the solutions based on the approaches of Theorems 2.2 and 2.3. Section 7 will experiment with this approach.

6 ON HYBRID RANK/SELECT DICTIONARIES

As recalled in Section 1.1, the literature offers a plethora of compressed rank/select dictionaries. Some take into account the statistical or the combinatorial properties of the input, others exploit the compressibility of clusters of consecutive integers. The compression scheme introduced in this paper, on the other hand, exploits the "geometric properties" of the input data by accommodating their slight deviations from linear trends with the use of small correction values. The choice of the best compression scheme in terms of space occupancy heavily depends on the characteristics of the input data, and thus it is reasonable to expect the best gains in space if we design *hybrid* solutions that combine several different approaches [4, 50, 59].

In the following, we combine the ideas of Section 5 with the hybrid rank/select dictionary of [50] and thus design an *improved* hybrid rank/select dictionary. This uses a two-level scheme in which the lower level stores S, properly partitioned into chunks (as detailed below), and the upper level stores, for each lower-level chunk $x_i, x_{i+1}, \ldots, x_j$, the integer $x_i = \text{select}(i)$, the length $j - i + 1$, and a pointer to the encoding in the lower level. Therefore, the amount of bits stored in the upper level for each chunk is upper bounded by $F = \log u + 2 \log n$.

Following [50], we assign to a generic chunk $x_i, x_{i+1}, \ldots, x_j$ a cost $w(i, j)$ given by the sum of F and a cost that depends on the encoding of the elements in that chunk. If $u' = x_j - x_i$ is the

universe size of the chunk, and $n' = j - i + 1$ is the number of elements in the chunk, then the cost of that encoding is the minimum of:

- 0 bits, if $u' = n'$ and thus the chunk is a run (R) of consecutive integers in which rank/select can be computed in constant time from x_i and i.
- $u' + o(u')$ bits, if we use a characteristic bitvector (BV) of size u' augmented with the information to support rank and select in constant time.
- $n'(\lceil \log \frac{u'}{n'} \rceil + 2)$ bits, if we use Elias-Fano (EF), which supports select in $O(1)$ time and rank in $O(\log \frac{u'}{n'})$ time.
- $n'c + w + c + \log c_{max}$ bits, if there exists one single segment approximating all elements of the chunk with correction size c. In this case, select takes $O(1)$ time and rank takes $O(c)$ time.

Note that the last cost slightly differs from the one in Theorem 2.2. Similarly to Theorem 2.2, we use $n'c$ bits for the vector of corrections. Differently from Theorem 2.2, we use c bits to encode the intercept instead of $\log n$ bits because the intercept value is guaranteed to be at most ε far from the value i (which is already stored in the upper level of the two-level structure), and thus it can be encoded by shifting i by an amount stored in $c = \Theta(\log \varepsilon)$ bits. Also, we encode the slope in a word of w bits. Finally, since we need to keep the value c (which possibly changes for each segment), we use additional $\log c_{max}$ bits per segment, where $c_{max} = O(\log u)$ is defined as in Section 5 as the minimum correction size that produces one single segment on S.

The overall cost in bits of the two-level structure corresponding to a partition P of S into k chunks with endpoints $1 = i_0, i_1, \ldots, i_k = n$ is given by $w(P) = \sum_{h=0}^{k-1} w(i_h, i_{h+1} - 1)$. To solve the problem of finding an optimal partition P that minimises $w(P)$, we slightly alter the algorithm of [23, 50] to consider also an encoding via segments. The algorithm of [23, 50] finds in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space a partition whose cost is only $1 + \epsilon$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$. This is done via a left-to-right scan of S, hence for $i = 1, \ldots, n$, that keeps $O(\log_{1+\epsilon} \frac{1}{\epsilon})$ sliding windows that start all from i and are such that the kth window covers a chunk $[i, j]$ such that either $w(i, j) \leq F(1 + \epsilon)^k < w(i, j + 1)$ or $j = n$.

A crucial property used in [50] is that computing $w(i, j)$ for the first three encoders above (namely, EF, BV, and R) takes constant time. Instead, computing whether there is a segment approximating the integers in a chunk requires $O(n')$ time. Since we need to compute a segment for each value of $c \in \{0, 2, 3, \ldots, c_{max}\}$, computing the $(1 + \epsilon)$-optimal partition P minimising $w(P)$ takes $O(c_{max}\, n^2 \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space in the presence of segments, where $c_{max} = O(\log u)$.

In Section 7.5, we experiment with an approach that uses the algorithm of Section 5 to compute a partition in $O(c_{max}\, n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n + c_{max}) = O(n)$ space. It operates by keeping a frontier of c_{max} segments that overlap the corresponding window and by updating the frontier when the window moves, as we have seen in Section 5 (see the example in Figure 5).

7 EXPERIMENTS

Our experiments were run on a machine with 40 GB of RAM and an Intel Xeon E5-2407v2 CPU.

7.1 Implementation notes

The implementation of our `la_vector` is done in C++, and its code is available at https://github.com/gvinciguerra/la_vector. In the following, we will use the notation `la_vector<c>`, where c is the correction size, to refer to our plain dictionary described in Sections 2 and 3, and use `la_vector_opt` to denote our space-optimised dictionary described in Section 5.

We store the segments triples $s_j = (r_j, \alpha_j, \beta_j)$ as an array of structures with memory-aligned fields. This allows for better locality and aligned memory accesses. Since in practice the segments are few (see Figure 6) and fit the last-level cache, we avoid complex structures on top of the r_js and

the select(r_j)s (as suggested by Corollaries 3.2 and 3.5 to asymptotically speed up select and rank, respectively).

To further speed up rank and select, we introduce two small tables of size 2^{16} each that allow accessing in one hop a narrower range of segments to binary search on. These two tables use fixed-size cells of $\lceil \log \ell \rceil$ bits, because they index segments. Specifically, the table T_1 of size 2^{16} partitions the n keys into blocks of size $d_1 = \lceil n/2^{16} \rceil$, so that $T_1[k]$ points to the segment covering the first key of the kth block. This way, a select(i) query can be answered by binary searching the segments between positions $T_1[k]$ and $T_1[k+1]$, where $k = \lfloor i/d_1 \rfloor$ is the index of the block containing the ith key. Similarly, the small table T_2 of size 2^{16} partitions the universe into blocks of size $d_2 = \lceil u/2^{16} \rceil$, so that $T_2[y]$ points to the segment covering the first position of block y. This way, a rank(x) query can be answered by binary searching the segments between positions $T_2[y]$ and $T_2[y+1]$, where $y = \lfloor x/d_2 \rfloor$ is the index of the block containing value x.

We introduce two other algorithm engineering tricks. The first one is to copy the first correction $C[r_j]$ into the segment s_j structure. This improves the spatial locality of Line 1 in Algorithm 2, because both $C[r_j]$ and the values needed to compute $f_j(r_j)$ are stored nearby. The second trick is a two-level layout for C that reduces the number of cache misses of Line 6 in Algorithm 2. Specifically, we split C into an array C_1 storing all the corrections $C[i]$ such that i is a multiple of an integer d, and an array C_2 containing the remaining corrections. Note that because of this split, we must slightly alter select(i) so that it accesses $C_1[\lfloor i/d \rfloor]$ if $i \bmod d = 0$, and $C_2[i - \lfloor i/d \rfloor]$ otherwise. Then, we modify Line 6 to perform two binary searches. The first one touches only the $C[i]$s such that $i \bmod d = 0$. The second one touches the $\Theta(d)$ correction values in C_2 in-between two consecutive positions found by the first binary search. Experimentally, we found that the best performance is achieved when d is roughly four cache lines of correction values (i.e. $d = \lceil 4 \cdot 512/c \rceil$ in our machine with 512-bit cache lines).

7.2 Baselines

We use the following rank/select dictionaries from the Succinct Data Structures Library (`sdsl`) [27]:

- **`sd_vector`:** the Elias-Fano representation for increasing integer sequences with constant-time select [48].
- **`rrr_vector<`t`>`:** a practical implementation of the H_0-compressed bitvector of Raman, Raman and Rao with t-bit blocks [11, 56].
- **`enc_vector<`$\gamma/\delta, s$`>`:** it encodes the gaps between consecutive integers via either Elias γ- or δ-codes. Random access is implemented by storing, with sample rate s, an uncompressed integer and a bit-pointer to the beginning of the code of the following gap. We implemented rank via a binary search on the samples, followed by the sequential decoding and prefix sum of the gaps in-between two samples.

We also use the following rank/select dictionaries from the Data Structures for Inverted Indexes (`ds2i`) library [50]:

- **`uniform_partitioned`:** it divides the input into fixed-sized chunks and encodes each chunk with Elias-Fano.
- **`opt_partitioned`:** it divides the input into variable-sized chunks and encodes each chunk with Elias-Fano. The endpoints are computed by a dynamic programming algorithm that minimises the overall space.

In both structures above, endpoints and boundary values of the chunks are stored in a separate Elias-Fano data structure. For a fair comparison, we disallow the use of encoding schemes for

Table 2. Characteristics of the datasets

Dataset	Density n/u	n (M)	u (M)	Size in MiB
Gov2 avg +10M (53.0%)	53.04%	13.06	24.62	49.81
Gov2 avg 1M-10M (13.4%)	13.37%	3.29	24.62	12.56
Gov2 avg 100K-1M (1.3%)	1.29%	0.31	24.56	1.20
URL (5.6%)	5.58%	57.97	1039.92	221.16
URL (1.3%)	1.30%	13.55	1039.91	51.72
URL (0.4%)	0.36%	3.73	1039.86	14.23
5gram (9.8%)	9.85%	145.39	1476.73	554.64
5gram (2.0%)	1.98%	29.19	1476.72	111.80
5gram (0.8%)	0.76%	11.21	1476.68	42.79
DNA (30.0%)	30.02%	300.23	999.99	1145.32
DNA (6.0%)	6.00%	60.03	999.99	229.00
DNA (1.2%)	1.20%	12.00	999.99	45.79

chunks different from Elias-Fano, and we defer the experimentation of such hybrid rank/select dictionaries to Section 7.5.[4]

To widen our experimental comparison, we also use:

`rle_vector<`*b*`>`: it implements a run-length encoding of the input bitvector by alternating the lengths of runs of `0`s and `1`s, coded in VByte (but over nibbles). To support efficient operations, two separate `sd_vectors` store, for each b-byte block, the position and the rank of the first 1-bit in the block. [40].[5]

`s18_vector<`*b*`>`: it uses gap and run-length encoding to compress the input bitvector via a sequence of 32-bit codes. To support efficient operations, it stores rank and select samples every b codes [5].

7.3 Datasets

We test lists of integers originating from different applications. We select these lists so that their density n/u vary significantly, viz. up to three orders of magnitude. The universe size u never exceeds $2^{32} - 1$, because the implementations in `ds2i` only support 32-bit integers. We use the following datasets, whose characteristics are summarised in Table 2.

Gov2 is an inverted index built on a collection of about 25M `.gov` sites, in which document identifiers were assigned according to the lexicographic order of their URLs [50]. In Figures 2, 4 and 6, we use the longest inverted list which has a density of 76.6%. In Figures 7 and 8, we instead test all solutions over each list separately and average the results over lists of lengths 100K–1M, 1M–10M and > 10M. This grouping of lists by length induces an average density of 1.29%, 13.37% and 53.04%, respectively.

URL is a text file of 1.03 GB containing URLs originating from three sources, namely a human-curated web directory, global news, and journal articles' DOIs.[6] On this file, we first applied

[4]We ought to mention that hybrid encoding schemes were not correctly disabled in the conference version [8]. This error caused Partitioned Elias-Fano to have an advantage of 41% space, 8% select time, and a disadvantage of 2% rank time.

[5]The implementation of `rle_vector` is available at https://github.com/vgteam/sdsl-lite.

[6]Available at https://kaggle.com/shawon10/url-classification-dataset-dmoz, https://doi.org/10.7910/DVN/ILAT5B, and https://archive.org/details/doi-urls, respectively.

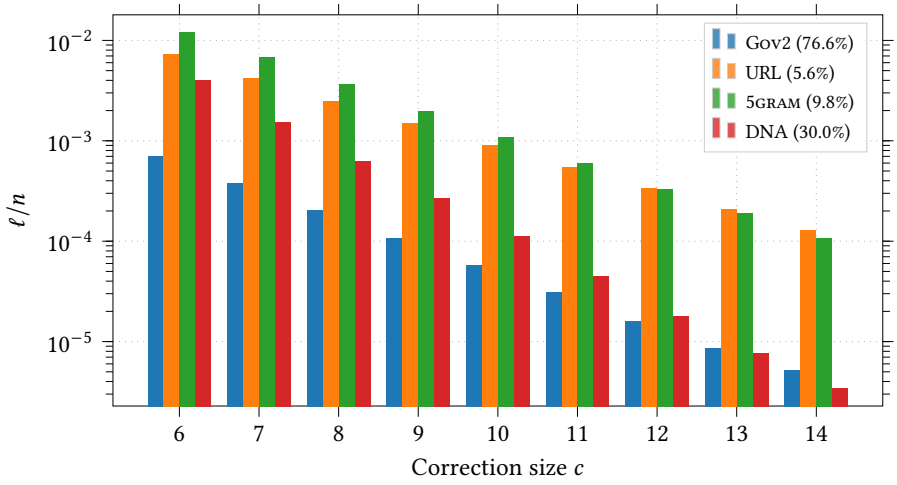


Fig. 6. The ratio between the number of segments ℓ and the size n of the largest datasets at different correction sizes c.

the Burrows-Wheeler Transform (BWT), as implemented by [16], and then we generated three integer lists by enumerating the positions of the ith most frequent character in the resulting BWT. The different list sizes (and densities) were achieved by properly setting i, and they were 3.7M (0.36%), 13M (1.30%) and 57M (5.58%).

5GRAM is a text file of 1.4 GB containing 60M different five-word sequences occurring in books indexed by Google.[7] As for URL, we first applied the BWT and then generated three integer lists of sizes (densities): 11M (0.76%), 29M (1.98%) and 145M (9.85%).

DNA is the first GB of the human reference genome.[8] We generated an integer list by enumerating the positions of the A nucleobase. Different densities were achieved by randomly deleting an A-occurrence with a fixed probability. The list sizes (and densities) are 12M (1.20%), 60M (6.00%) and 300M (30.02%).

As a first experiment, we show in Figure 6 that the number of segments ℓ composing the optimal PLA of the various input datasets is orders of magnitude smaller than the input size. These figures make our approach very promising, as argued at the beginning of this paper. The following experiments will assume $c \geq 6$ for `la_vector<c>` because, on these datasets, smaller values of c make ℓ too large and thus the space occupied by the segments becomes significantly larger than the space taken by the correction array C.

7.4 Experiments on rank and select

We now experiment with the time and space performance of rank/select dictionaries by running them on each dataset (of size n) with a batch of $0.2n$ random queries. For clarity and significance of the plots, we only show the implementations that use less than 16 bits per integer and whose average query time is not too high with respect to the others.

7.4.1 Performance of select. Figure 7 shows the results for select. We notice that our `la_vector<c>` variants consistently provide the best time performance. This comes at the cost of requiring c bits

[7] Available at https://storage.googleapis.com/books/ngrams/books/datasetsv3.html.
[8] Available at https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39.

per integer, plus the cost of storing the segments. For very low densities (plots in the first column) and low values of c, the overhead due to the segments may exceed the cost of storing C (see e.g. 5GRAM and DNA, where the set of `la_vector` configurations is U-shaped). This unlucky situation is solved by `la_vector_opt`, which avoids the tuning of c by computing the PLA that minimises the overall space, possibly adopting different c for different segments. Note that `la_vector_opt` is always faster than the plain Elias-Fano encoding (i.e. `sdsl::sd_vector`), except for large densities in DNA (i.e. 30%), and it is also more compressed on the GOV2, 5GRAM and URL datasets.

The other Elias-Fano encodings are generally fast as well, with `ds2i::uniform_partitioned` and `opt_partitioned` being more compressed but roughly 50 ns slower than `sdsl::sd_vector` due to the use of a two-level structure. In any case, our `la_vector_opt` and `la_vector<c>` are not dominated by these advanced Elias-Fano variants over all the datasets, except for large densities in DNA.

For what concerns `sdsl::enc_vector` and `sdsl::rrr_vector`, they are pretty slow although offering very good compression ratios. The slow performance of select in the latter is due to its implementation via a combination of a binary search on a sampled vector of ranks plus a linear search in-between two samples (see [44, §4.3]).

The same goes for `s18_vector`, which is very succinct but not fast, in fact, it is only on the Pareto frontier of the URL dataset.

Finally, we notice that `rle_vector` is dominated in time and space by some other data structure on all the datasets except for URL (0.4%).

7.4.2 Performance of rank. Figure 8 shows the results for rank. We observe that `sdsl::rrr_vector` and `sdsl::sd_vector` achieve the best time performance with `la_vector` following closely, i.e. within 120 ns or less. However, at low densities (first column of Figure 8), `sdsl::rrr_vector` has a very poor space performance, more than 10 bits per integer.

Not surprisingly, `sdsl::enc_vector<·, s>` has often the slowest rank, because it performs a binary search on a vector of n/s samples, followed by the linear decoding and prefix sum of at most s gaps coded with γ or δ.

`s18_vector` is very succinct but not fast, in fact, it is only on the Pareto frontier of the URL dataset, as it occurred for the select query.

`rle_vector` is dominated in time and space by some other data structure on all the datasets except for URL (0.4%), as it occurred for the select query.

Note that for GOV2, URL and 5GRAM our `la_vector_opt` falls on the Pareto frontier of Elias-Fano approaches thus offering an interesting space-time trade-off also for the rank query.

7.4.3 Discussion on the space-time performance. Overall, `sdsl::rrr_vector` provides the fastest rank but the slowest select. Its space is competitive with other implementations only for moderate and large densities of 1s.

The Elias-Fano approaches provide fast rank and moderately fast select in competitive space. In particular, the plain Elias-Fano (`sdsl::sd_vector`) offers fast operations but in a space competitive with other structures only on DNA; while the partitioned variants of Elias-Fano implemented in `ds2i` offer the best compression but at the cost of slower rank and select. On low densities of the DNA datasets (i.e. 6.0% and 1.2%) the implementations of `ds2i` provide the best time and space performance.

`sdsl::enc_vector<·, s>` provides a smooth space-time trade-off controlled by the s parameter, but it has non-competitive rank and select operations.

`s18_vector` is very succinct but provides generally slow rank and select operations. It is only on the Pareto frontier of the URL datasets.

`rle_vector` is only on the Pareto frontier of URL (0.4%).

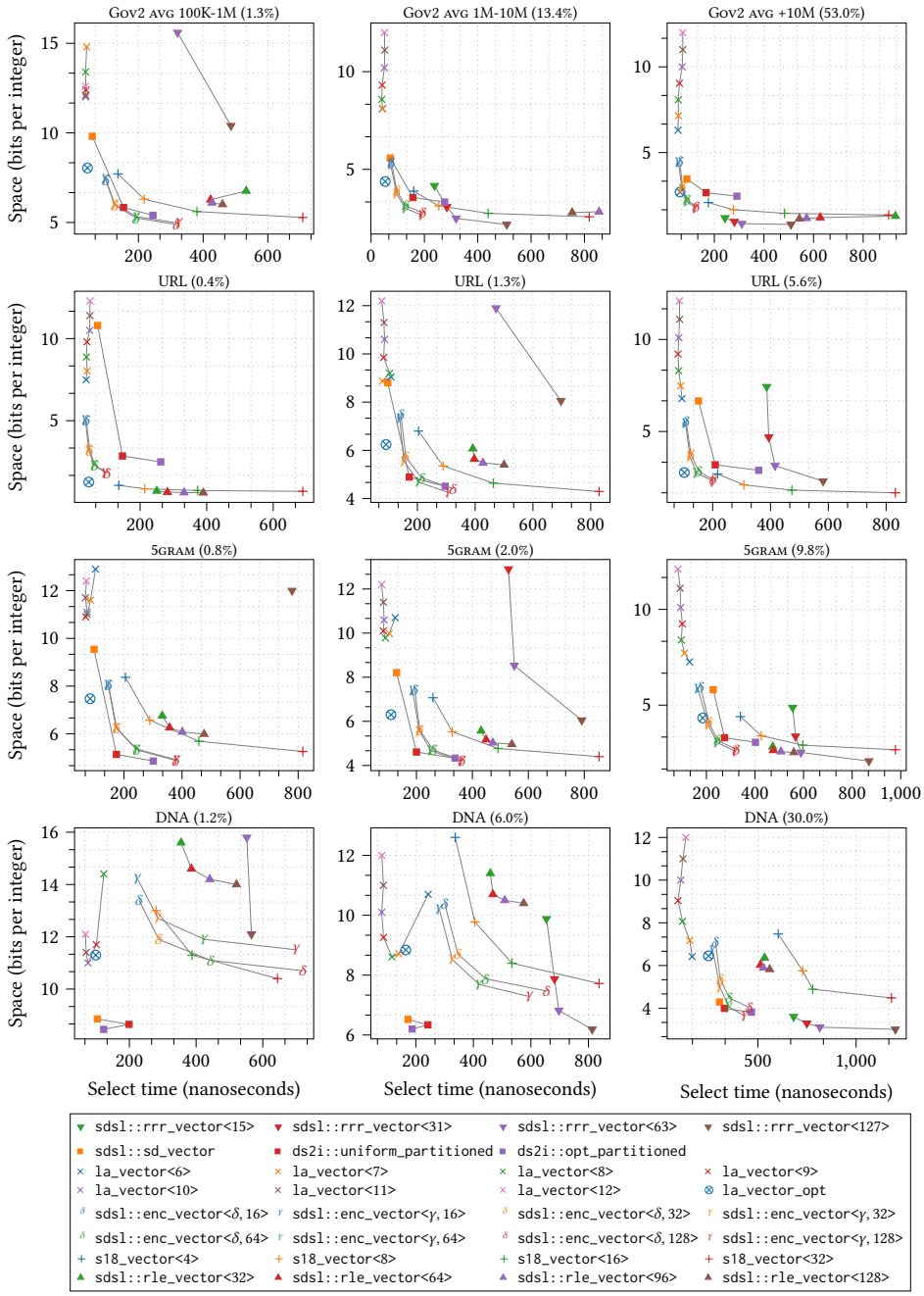


Fig. 7. Space-time performance of the select query.

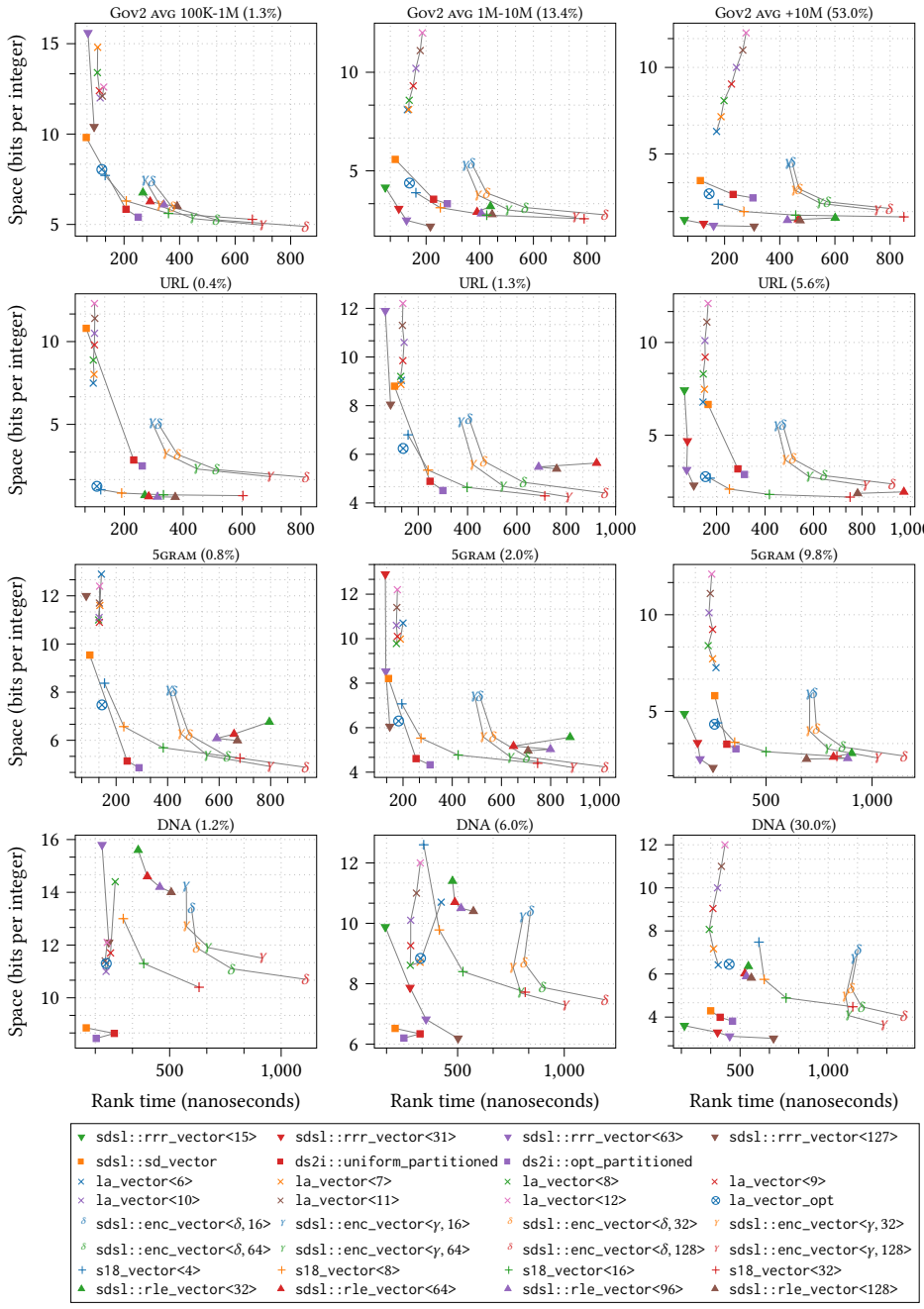


Fig. 8. Space-time performance of the rank query.

Table 3. Number of chunks in the hybrid approach of Section 6 that are better compressed by a segment (plus corrections).

Dataset	# of chunks that use a segment	% of chunks that use a segment	% of integers coded with segments
Gov2 avg +10M (53.0%)	2217	11.21%	12.43%
Gov2 avg 1M-10M (13.4%)	424	4.57%	9.53%
Gov2 avg 100K-1M (1.3%)	44	2.74%	4.91%
URL (5.6%)	30396	15.35%	22.69%
URL (1.3%)	7356	10.02%	9.99%
URL (0.4%)	2093	26.76%	54.97%
5gram (9.8%)	31911	8.44%	14.44%
5gram (2.0%)	5640	5.10%	13.49%
5gram (0.8%)	1640	3.41%	6.88%
DNA (30.0%)	215	0.03%	0.01%
DNA (6.0%)	0	0%	0%
DNA (1.2%)	0	0%	0%

Our `la_vector<c>` offers the fastest select, competitive rank, and a smooth space-time trade-off controlled by the c parameter, where values of $c \geq 6$ were found to "balance" the cost of storing the corrections and the cost of storing the segments. Our space-optimised `la_vector_opt` in most cases (i) dominates the space-time performance of `la_vector<c>`; (ii) offers a select which is faster than all the other tested approaches; (iii) offers a rank which is on the Pareto frontier of Elias-Fano approaches.

Finally, for the construction times over the various datasets, we report that `la_vector<c>` (we averaged over the values of c used in Figures 7 and 8) builds 1.41× faster than `sdsl::enc_vector`, 2.45× faster than `sdsl::rrr_vector`, 9.74× faster than `s18_vector`, and 1.89× slower than `sdsl::sd_vector`. For what concerns the space-optimised `la_vector_opt`, it builds 82.18× slower than the plain `la_vector<c>`, and 2.41× slower than the homologous space-optimised Elias-Fano (i.e. `ds2i::opt_partitioned`). Future work is needed to improve the construction performance of `la_vector_opt`.

7.5 Experiments on hybrid rank/select dictionaries

We evaluate the hybrid structure of Section 6 that combines segments, Elias-Fano (EF), plain bitvectors (BV), and runs (R) of consecutive integers. We look in particular at how many chunks and how many integers are encoded via segments, and thus the impact of our "geometric" approach on the hybrid rank/select dictionary of [50].

From the results in Table 3, we notice that the segments are chosen as encodings of the chunks in all the datasets except for DNA (6.0%) and DNA (1.2%). The overall amount of chunks that use segments is below 16% except for URL (0.4%), where the number of chunks that use segments is very large, namely 26.76%.

As far as the percentage of integers encoded with each compression scheme is concerned, Figure 9 shows that segments are often selected as the best compression scheme for a substantial part of every dataset. In particular, half of the URL (0.4%) dataset is encoded with segments. Therefore, we argue that our "geometric" approach can compete with well-established succinct encoding schemes.

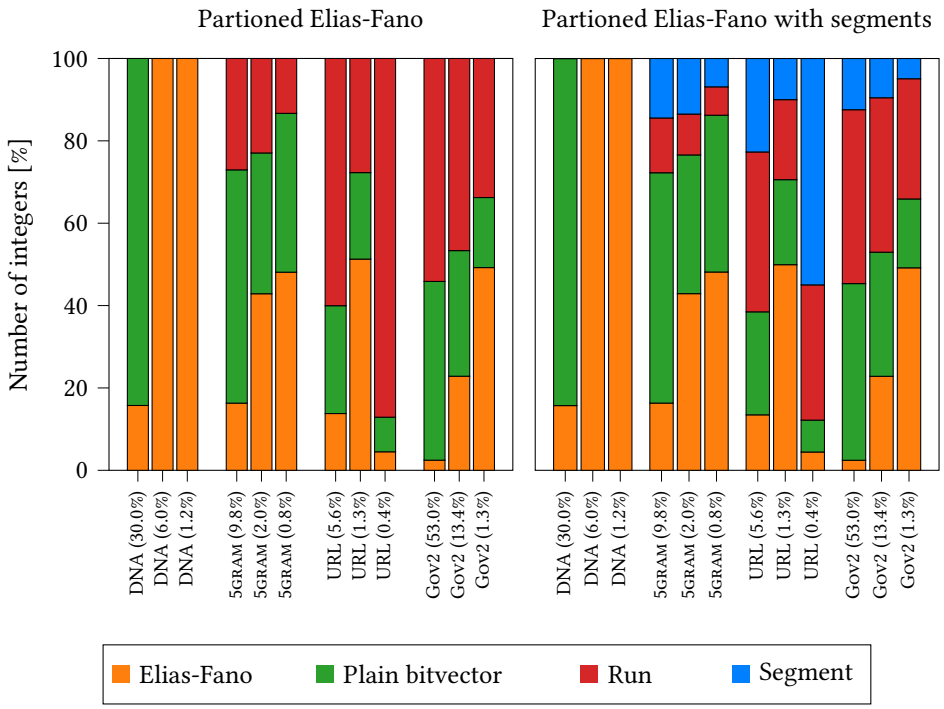


Fig. 9. Percentage of integers encoded with each compression scheme.

Looking at Figure 9, it is also clear that segments mainly substitute the run encoding (R). This could seem counter-intuitive since R uses 0 bits of space. But this can be explained by the fact that, for each chunk, we need to store some metadata (namely the first integer of the chunk, its length, and a pointer), and thus we can get better compression by reducing the overall number of chunks, as the introduction of segments does. For example, consider a characteristic bitvector composed of x equally long runs of 1 that are separated by a single 0. R would need x chunks to encode that and so x sets of metadata. Instead, just one segment is able to represent the x runs using a few bits per integer and just 1 set of metadata. Indeed, once we introduce the segments as an encoding scheme, the total number of chunks always decreases, up to 15%. A situation similar to the previous example often happens in the BWT of highly repetitive texts, and this explains the high presence of our encoding scheme in the URL and 5GRAM datasets.

Overall, the hybrid structure of Section 6 that combines segments, EF, BV, and R is able to use up to 1.34% less space and be just 6.5% slower on average both on rank and select than the hybrid solution without the segments. The space reduction on these datasets may not seem very impressive, but we remind the reader that the improved solution uses state-of-the-art encoders and thus it is already very squeezed. Finally, we observe that our hybrid solution turns out to be slightly slower because of the few more mathematical operations needed to work on segments in the place of R.

8 CONCLUSIONS AND FUTURE WORK

We have shined a new light on the classical problem of designing rank/select dictionaries by showing a connection between the input data and the geometry of a set of points in a Cartesian plane suitably derived from them. We have introduced new data structures based on this idea and

proved their good theoretical bounds and competitive experimental performance with respect to several well-established approaches.

For future work, we mention the study of a relation between classical compressibility measures, such as entropy, and the measure introduced in Section 2.1 based on the number of segments ε-approximating the input data. For what concerns Section 2.2, we argue that the space of `la_vector` can be further improved by computing segments in such a way that the statistical redundancy of the correction values in C is increased. This could be possibly achieved by jointly optimising the space occupied by the segments and the space occupied by the compressed C, playing on both the segments' lengths and their correction values. We also mention the use of vectorised instructions [37] to achieve fast compression and fast scanning of the corrections in C. Finally, we suggest an in-depth study, design and experimentation of hybrid rank/select structures, possibly integrating nonlinear models.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their valuable comments. We also thank the staff of the Green Data Centre at the University of Pisa for providing us with machines and technical support to execute the numerous experiments that have been presented in this paper.

This work has been supported in part by the Italian MIUR PRIN project "Multicriteria data structures and algorithms: from compressed to learned indexes, and beyond" (Prot. 2017WR7SHH), by Regione Toscana (under POR FSE 2014/2020), and by the EU H2020 projects "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (grant #871042) and "HumanE AI Network" (grant #952026).

REFERENCES

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: enabling queries on compressed data. In *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 337–350.

[2] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *PVLDB* 4, 8 (2011), 470–481.

[3] Diego Arroyuelo, Mauricio Oyarzún, Senen Gonzalez, and Victor Sepulveda. 2018. Hybrid compression of inverted lists for reordered document collections. *Information Processing & Management* 54 (05 2018), 1308–1324. https://doi.org/10.1016/j.ipm.2018.05.007

[4] Diego Arroyuelo and Rajeev Raman. 2019. Adaptive Succinctness. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*. 467–481. https://doi.org/10.1007/978-3-030-32686-9_33

[5] Diego Arroyuelo and Manuel Weitzman. 2020. A Hybrid Compressed Data Structure Supporting Rank and Select on Bit Sequences. In *Proc. 39th International Conference of the Chilean Computer Science Society (SCCC)*. https://doi.org/10.1109/SCCC51225.2020.9281244

[6] Jeremy Barbay and Gonzalo Navarro. 2009. Compressed Representations of Permutations, and Applications. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, Vol. 3. 111–122. https://doi.org/10.4230/LIPIcs.STACS.2009.1814

[7] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2008. Theory and Practice of Monotone Minimal Perfect Hashing. *ACM Journal of Experimental Algorithmics* 16, Article 3.2 (Nov. 2008). https://doi.org/10.1145/1963190.2025378

[8] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2021. A "learned" approach to quicken and compress rank/select dictionaries. In *Proc. 23rd SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 46–59. https://doi.org/10.1137/1.9781611976472.4

[9] Adam L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. 2003. Improving Table Compression with Combinatorial Optimization. *J. ACM* 50, 6 (Nov. 2003), 825–851. https://doi.org/10.1145/950620.950622

[10] David Richard Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo, Canada.

[11] Francisco Claude and Gonzalo Navarro. 2008. Practical Rank/Select Queries over Arbitrary Sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. 176–187. https://doi.org/10.1007/978-3-540-89097-3_18

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[13] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory* (2nd ed.). Wiley.

[14] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (April 1974), 246–260. https://doi.org/10.1145/321812.321820

[15] Robert Mario Fano. 1971. *On the number of bits required to implement an associative memory. Memo 61.* Massachusetts Institute of Technology, Project MAC.

[16] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. 2012. Lightweight Data Indexing and Compression in External Memory. *Algorithmica* 63, 3 (2012), 707–730. https://doi.org/10.1007/s00453-011-9535-0

[17] Paolo Ferragina, Stefan Kurtz, Stefano Lonardi, and Giovanni Manzini. 2018. Computational Biology. In *Handbook of Data Structures and Applications* (2nd ed.), Dinesh P. Mehta and Sartaj Sahni (Eds.). CRC Press, Chapter 59, 917–934. https://doi.org/10.1201/9781315119335-59

[18] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why are learned indexes so effective?. In *Proc. 37th International Conference on Machine Learning (ICML)*. 3123–3132.

[19] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2021. On the performance of learned data structures. *Theoretical Computer Science* 871 (2021), 107–120.

[20] Paolo Ferragina and Giovanni Manzini. 2005. Indexing Compressed Text. *J. ACM* 52, 4 (July 2005), 552–581. https://doi.org/10.1145/1082036.1082039

[21] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2004. An Alphabet-Friendly FM-Index. In *Proc. 11th International Conference on String Processing and Information Retrieval (SPIRE)*. 150–160. https://doi.org/10.1007/978-3-540-30213-1_23

[22] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2007. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms* 3, 2 (May 2007), article 20. https://doi.org/10.1145/1240233.1240243

[23] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2011. On Optimally Partitioning a Text to Improve Its Compression. *Algorithmica* 61, 1 (2011), 51–74. https://doi.org/10.1007/s00453-010-9437-6

[24] Paolo Ferragina and Giorgio Vinciguerra. 2020. Learned Data Structures. In *Recent Trends in Learning From Data*, Luca Oneto, Nicolò Navarin, Alessandro Sperduti, and Davide Anguita (Eds.). Springer, 5–41. https://doi.org/10.1007/978-3-030-43883-8_2

[25] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[26] Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2006. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. *ACM Transactions on Algorithms* 2, 4 (Oct. 2006), 611–639. https://doi.org/10.1145/1198513.1198521

[27] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. 326–337. https://doi.org/10.1007/978-3-319-07959-2_28

[28] Simon Gog and Matthias Petri. 2014. Optimized succinct data structures for massive data. *Software: Practice and Experience* 44, 11 (2014), 1287–1314. https://doi.org/10.1002/spe.2198

[29] Alexander Golynski. 2007. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science* 387, 3 (2007), 348–359. https://doi.org/10.1016/j.tcs.2007.07.041

[30] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. 2006. Rank/Select Operations on Large Alphabets: a Tool for Text Indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*. 368–373.

[31] Alexander Golynski, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. 2014. Optimal Indexes for Sparse Bit Vectors. *Algorithmica* 69, 4 (Aug. 2014), 906–924. https://doi.org/10.1007/s00453-013-9767-2

[32] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Proc. Poster of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 27–38.

[33] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-Order Entropy-Compressed Text Indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

[34] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. 2007. Compressed data structures: dictionaries and data-aware measures. *Theoretical Computer Science* 387, 3 (2007), 313–331. https://doi.org/10.1016/j.tcs.2007.07.042

[35] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*. 549–554.

[36] Sambasiva Rao Kosaraju and Giovanni Manzini. 1999. Compression of Low Entropy Strings with Lempel-Ziv Algorithms. *SIAM J. Comput.* 29, 3 (1999), 893–911. https://doi.org/10.1137/S0097539797331105

[37] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. https://doi.org/10.1002/spe.2203

[38] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press.
[39] Veli Mäkinen and Gonzalo Navarro. 2007. Rank and select revisited and extended. *Theoretical Computer Science* 387, 3 (2007), 332–347. https://doi.org/10.1016/j.tcs.2007.07.013
[40] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. 2010. Storage and Retrieval of Highly Repetitive Sequence Collections. *Journal of Computational Biology* 17, 3 (2010), 281–308. https://doi.org/10.1089/cmb.2009.0169
[41] J. Ian Munro. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 37–42. https://doi.org/10.1007/3-540-62034-6_35
[42] J. Ian Munro and Venkatesh Raman. 1997. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*. 118–126. https://doi.org/10.1137/S0097539799364092
[43] Gonzalo Navarro. 2014. Spaces, Trees, and Colors: the Algorithmic Landscape of Document Retrieval on Sequences. *Comput. Surveys* 46, 4, Article 52 (March 2014), 47 pages. https://doi.org/10.1145/2535933
[44] Gonzalo Navarro. 2016. *Compact data structures: a practical approach.* Cambridge University Press.
[45] Gonzalo Navarro and Veli Mäkinen. 2007. Compressed Full-Text Indexes. *Comput. Surveys* 39, 1 (April 2007), 61 pages. https://doi.org/10.1145/1216370.1216372
[46] Gonzalo Navarro and Eliana Providel. 2012. Fast, Small, Simple Rank/Select on Bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*. 295–306. https://doi.org/10.1007/978-3-642-30850-5_26
[47] Gonzalo Navarro and Javiel Rojas-Ledesma. 2020. Predecessor Search. *Comput. Surveys* 53, 5, Article 105 (2020), 35 pages.
[48] Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 60–70. https://doi.org/10.1137/1.9781611972870.6
[49] Joseph O'Rourke. 1981. An On-line Algorithm for Fitting Straight Lines Between Data Ranges. *Commun. ACM* 24, 9 (1981), 574–578. https://doi.org/10.1145/358746.358758
[50] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proc. 37th International ACM Conference on Research & Development in Information Retrieval (SIGIR)*. 273–282. https://doi.org/10.1145/2600428.2609615
[51] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Clustered Elias-Fano Indexes. *ACM Transactions on Information Systems* 36, 1, Article 2 (April 2017), 33 pages. https://doi.org/10.1145/3052773
[52] Mihai Pătraşcu. 2008. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 305–313. https://doi.org/10.1109/FOCS.2008.83
[53] Mihai Pătraşcu and Mikkel Thorup. 2006. Time-Space Trade-Offs for Predecessor Search. In *Proc. of the 38th Annual ACM Symposium on Theory of Computing (STOC)*. 232–240. https://doi.org/10.1145/1132516.1132551
[54] Mihai Pătraşcu and Emanuele Viola. 2010. Cell-Probe Lower Bounds for Succinct Partial Sums. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 117–122. https://doi.org/10.1137/1.9781611973075.11
[55] Rajeev Raman. 2016. Rank and Select Operations on Bit Strings. In *Encyclopedia of Algorithms* (2nd ed.), Ming-Yang Kao (Ed.). Springer, 1772–1775. https://doi.org/10.1007/978-1-4939-2864-4_332
[56] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct Indexable Dictionaries with Applications to Encoding k-Ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms* 3, 4 (Nov. 2007), article 43. https://doi.org/10.1145/1290672.1290680
[57] Kunihiko Sadakane and Roberto Grossi. 2006. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1230–1239. https://doi.org/10.1145/1109557.1109693
[58] Craig Silverstein. 2005. Google SparseHash. https://github.com/sparsehash/sparsehash.
[59] Fabrizio Silvestri and Rossano Venturini. 2010. VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming. In *Proc. 19th ACM International Conference on Information and Knowledge Management (CIKM)*. 1219–1228. https://doi.org/10.1145/1871437.1871592
[60] Sebastiano Vigna. 2008. Broadword Implementation of Rank/Select Queries. In *Proc. 7th International Workshop on Experimental Algorithms (WEA)*. 154–168. https://doi.org/10.1007/978-3-540-68552-4_12
[61] Sebastiano Vigna. 2013. Quasi-Succinct Indices. In *Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 83–92. https://doi.org/10.1145/2433396.2433409
[62] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). Morgan Kaufmann.
[63] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum Error-bounded Piecewise Linear Representation for Online Stream Approximation. *The VLDB Journal* 23, 6 (2014), 915–937. https://doi.org/10.1007/s00778-014-0355-0
[64] Huacheng Yu. 2019. Optimal Succinct Rank Data Structure via Approximate Nonnegative Tensor Decomposition. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 955–966. https://doi.org/10.1145/3313276.

3316352