A "learned" approach to quicken and compress rank/select dictionaries[*]

Antonio Boffa[†] Paolo Ferragina[†] Giorgio Vinciguerra[†]

Abstract

We address the well-known problem of designing, implementing and experimenting compressed data structures for supporting rank and select queries over a dictionary of integers. This problem has been studied far and wide since the end of the '80s with tons of important theoretical and practical results.

Following a recent line of research on the so-called *learned* data structures, we first show that this problem has a surprising connection with the geometry of a set of points in the Cartesian plane suitably derived from the input integers. We then build upon some classical results in computational geometry to introduce the first "learned" scheme for implementing a compressed rank/select dictionary. We prove theoretical bounds on its time and space performance both in the worst case and in the case of input distributions with finite mean and variance.

We corroborate these theoretical results with a large set of experiments over datasets originating from a variety of sources and applications (Web, DNA sequencing, information retrieval and natural language processing), and we show that a carefully engineered version of our approach provides new interesting space-time trade-offs with respect to several well-established implementations of Elias-Fano, RRR-vector, and random-access vectors of Elias γ/δ-coded gaps.

1 Introduction

We consider the classical problem of representing, in compressed form, an ordered dictionary S of n elements over 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 which are less than or equal to x;
- $\mathsf{select}(i)$. Given $i \in \{1, \ldots, n\}$, return the ith smallest element in S.

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

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 [23, 26, 39], which are frequently at the core of 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 [29], 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 [38] used for designing compressed text indexes [15, 26, 39], succinct trees and graphs [36, 49], monotone minimal perfect hashing [5], sparse hash tables [51], and permutations [4]. Consequently, they have countless applications in bioinformatics [13, 32], information retrieval [37], 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 thus 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 via linear approximations. We prove theoretical bounds on its time and space performance both in the worst case and in the case of input distributions with finite mean and variance. We corroborate these theoretical results with a large set of experiments over a variety of 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 [38]; but also because, we argue, they introduce a new way of designing compact rank/select data structures which deploy computational geometry tools to "learn" the distribution of the input data. As such, we foresee that this novel data-aware design may offer research opportunities and stimulate new results of which many applications will hopefully benefit.

[*]Supported 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), by the European Integrated Infrastructure for Social Mining and Big Data Analytics (SoBigData++, Grant Agreement #871042), and by PRA UniPI 2018 "Emerging Trends in Data Science."

[†]Università di Pisa, Dipartimento di Informatica, Italy. Correspondence to: <giorgio.vinciguerra@phd.unipi.it>.

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 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 $B_S[i] = \mathtt{1}$ if $i \in S$, and $B_S[i] = \mathtt{0}$ otherwise, for $0 \leq i < u$. 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 [48].

It is long known that $u + o(u)$ bits are sufficient to have constant-time rank and select [8, 35]. 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 [22]. Later, optimal trade-offs were also given in terms of the density of 1s in B_S [24] or for the cell-probe model [47, 56]. Practical implementations of rank/select on plain bitvectors have also been extensively studied and experimented [20, 21, 25, 40, 41, 53].

If S is sparse, i.e. B_S contains few 0s or 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.[1] The best upper bound on the redundancy was attained by [45], 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 $\mathcal{B} + O(u/\operatorname{poly}\log u)$ bits. This essentially matches the lower bounds provided in [47]. A widely known solution for sparse S is the RRR encoding [49], 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 by [9, 21].

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 [10, 11], which supports select in $O(1)$ time and rank in $O(\log(u/n))$ time while taking $n \log(u/n) + 2n + o(n) = \mathcal{B} + O(n)$ bits of space. Its implementations and refinements proved to be very effective in a variety of real-world contexts [41, 43, 44, 53, 54]. We will compare our proposal with the best implementations to date [21, 43] and theoretical results [38].

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 [3, 19, 27, 33, 50]. Consider the gaps g_i between consecutive integers in S taken in sorted order, i.e. $g_i = \mathsf{select}(i) - \mathsf{select}(i-1)$, and suppose we could store each g_i in $\lceil \log(g_i + 1) \rceil$ bits. Then the gap measure is defined as $\mathit{gap}(S) = \sum_i \lceil \log(g_i + 1) \rceil$. An example of data-aware structure whose space occupancy is bounded in terms of *gap* is presented in [27], which takes $\mathit{gap}(S)(1 + o(1))$ bits while supporting select in $O(\log\log n)$ time and rank in time matching the predecessor search bounds [46]. Another example is given in [33] taking $\mathit{gap}(S) + O(n) + o(u)$ bits and supporting constant-time operations. An important ingredient of these *gap*-based data-aware structures are self-delimiting codes such as Elias γ- and δ-codes [55]. To provide a complete comparison with our proposal, we will experiment with some of these approaches implemented in the `sdsl` library [21].

Recent work [3] 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 close to the information-theoretic minimum. The proposal is mainly theoretic, and indeed authors evaluated only its space occupancy.

1.2 Our contribution. Following a recent line of research on the so-called *learned* data structures (see e.g. [17, 28, 30, 34]), we provide the first "learned" scheme for implementing a rank/select dictionary that builds upon some results of [18, 2] and extends them to deal with rank and select operations over compressed space. In particular, we introduce a novel lossless compressed storage scheme for the input dictionary S which turns this problem into the one 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 (Sections 3 and 4).

Our study shows that our data-aware approach is asymptotically efficient in time and space, with worst-case bounds that relate its performance with the number ℓ of segments approximating S and their (controlled) errors ε. In particular, Theorems 2.1 and 2.2 state some interesting space bounds which 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 studies also to the case of input sequences drawn from a

[1] $\mathcal{B}$ is related to the 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, $\mathcal{B} = uH_0(B_S) - O(\log u)$. We can further bound $uH_0(B_S) \leq n \log \frac{u}{n} + 1.44n$ [38].

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
ε	Maximum absolute correction value ($= 2^{c-1} - 1$)
ℓ	Number of segments (Definition 2.1)
s_j	The jth segment
r_j	Rank of the first element compressed by segment s_j
α_j	Slope of segment s_j
β_j	Intercept of segment s_j
f_j	Linear function implemented by segment s_j

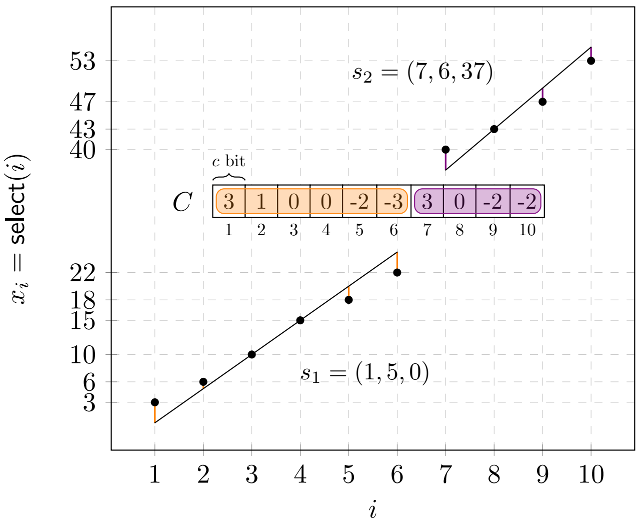

Figure 1: The encoding of $S = \{3, 6, 10, 15, 18, 22, 40, 43, 47, 53\}$ for $c = 3$ is given by the two segments s_1, s_2 and the array C. A segment $s_j = (r_j, \alpha_j, \beta_j)$ approximates the value of an item with rank i via $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$, and C corrects the approximation. For example, $x_5 = \lfloor f_1(5) \rfloor + C[5] = 20 - 2 = 18$ and $x_8 = \lfloor f_2(8) \rfloor + C[8] = 42 + 0 = 42$.

distribution with finite mean and variance (Section 5); 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 the practical setting [18]. Our final theoretical contribution is the design of a greedy algorithm that computes a provably-good approximation of the optimal set of segments which *minimises* the space occupancy of our compression scheme (Theorem 6.1 in 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 [20, 43], RRR-vector [9, 20], and random-access vectors of Elias γ/δ-coded gaps [21]. Our select is the fastest, whereas our rank is on the Pareto curve of the Elias-Fano approaches.

For the sake of presentation, we summarise in Table 1 the main notation used throughout the paper.

2 Compressing via linear approximations

We map each element $x_i \in S$ to a point (i, x_i) in the Cartesian plane, for $i = 1, 2, \ldots, n$. 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 $f(i)$. However, points might not be aligned, so that we seek for an encoding f which is fast to be computed and compressed in space.

In this paper, 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 a segment with slope 1. Generalising, any run of integers with a constant gap g can be encoded by a 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. Namely, 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). This PLA is a succinct and lossy representation of the input set S that, together with some additional information and algorithms, can be used to reconstruct S and return *exact* answers to rank/select queries.

To capture the aforementioned non-linear patterns, we search for a sequence of segments such that every point (i, x_i) is vertically far from one of these segments by an integer value ε, to be fixed later. In some sense, the sequence of segments introduces on the set of integers in S an "information loss" of ε. 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 [42]. This algorithm processes each point (i, x_i) left-to-right, hence $i = 1, \ldots, n$, 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 output of the algorithm above as the segment $s_j = (r_j, \alpha_j, \beta_j)$, 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, we set $r_{\ell+1} = n$ and observe that $r_1 = 1$. Thus the r_js partition the universe $[u]$ in ℓ ranges so that, for any integer i between r_j and r_{j+1} (non-inclusive), we use the segment s_j to approximate the value x_i as follows:

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

Now, we complement the inexact approximations computed by f_j with an array $C[1\,..\,n]$ of integers whose modulo is bounded above by ε. Precisely, each $C[i]$ represents the small "correction value" $x_i - \lfloor f_j(i) \rfloor$, which belongs to the set $\{-\varepsilon, -\varepsilon + 1, \dots, -1, 0, 1, \dots, \varepsilon\}$. If we allocate $c \geq 2$ bits for each correction in C, then our PLA is allowed to err by at most $\varepsilon = 2^{c-1} - 1$.

The vector C completes our encoding (depicted 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 $j = 1, \dots, \ell$ and for $i = r_j, \dots, r_{j+1} - 1$.

Recovering integer x_i requires first the identification of the segment s_j, which includes position i, and then the evaluation 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 [18] might appear similar to this idea because it uses PLAs and supports predecessor queries. However, the PGM-index does not compress the input keys but only the index, and it is tailored to the external-memory model, as B-trees.

On the compression effectiveness. Two counterpoising factors influence the effectiveness of our new compression method:

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 segments.
2. The value of the parameter $c \geq 2$ which determines the space occupancy of the array C, having size nc bits. From above, we know that $\varepsilon = 2^{c-1} - 1$, so the smaller is c, the smaller is the space occupancy of C, but the larger is "expected" to be the size ℓ of the PLA built for S.

We say "expected" because ℓ depends on the distribution of the points (i, x_i) over 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 Lemma 2 in [18], we will show in Section 5 that for sequences drawn from a distribution with finite mean and variance there are more stringent bounds on ℓ. This leads us to argue that the combination of the PLA and the array C 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 can be thought of as a new compressibility measure for the information present in S, possibly giving some insights (such as the degree of linearity of the data) that the classical entropy measures do not explicitly capture.[2]

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

Now, we are ready to compute the space taken by our new encoding. 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;[3] (ii) the slope α_j can be encoded into a memory word of w bits as a floating-point value; (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 $\ell\,(\log n + \log u + w)$ bits. Summing the nc bits taken by C gives our first result.

Theorem 2.1. *Let $S = \{x_1, \dots, x_n\}$ be a set of n integers drawn from the universe $[u]$. Given an integer parameter $c \in \{2, \dots, \log u\}$, there exists a compressed representation of S that takes $nc + \ell\,(\log n + \log u + w)$ bits of space, where ℓ is the number of segments in the optimal PLA for S of maximum error $\varepsilon = 2^{c-1} - 1$.*

[2]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.

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

We can further improve the space taken by the segments as follows. The r_js form an increasing sequence of ℓ integers bounded by n. The β_js form an increasing sequence of ℓ integers bounded by u.[4] Using the Elias-Fano representation [10, 11], we reduce the space of the two sequences to $\ell \log(n/\ell) + \ell \log(u/\ell) + 4\ell + o(\ell) = \ell\,(\log(un/\ell^2) + 4 + o(1))$ bits. Then, accessing r_j or β_j amounts to call the constant-time $\mathsf{select}(j)$ on the corresponding Elias-Fano compressed sequence.

Theorem 2.2. *Let S be a set of n integers over the universe $[u]$. Given an integer parameter $c \in \{2, \dots, \log u\}$, there exists a compressed representation of S that takes $nc + \ell\,(\log \frac{un}{\ell^2} + w + 4 + o(1))$ bits of space, where ℓ is the number of segments in the optimal PLA for S of maximum error $\varepsilon = 2^{c-1} - 1$.*

Note that one can avoid the floating-point representation and remove the dependence from w by implementing the algorithm of [42] with integer arithmetic and by encoding each slope as a rational number with a numerator of $\log u$ bits and a denominator of $\log n$ bits.

3 Supporting select

To answer $\mathsf{select}(i)$ on the representation of Theorem 2.1 (or Theorem 2.2), 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 $j = \mathsf{pred}(i)$. 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 $\mathsf{select}(i)$. The last two steps take $O(1)$ time. Treating $\mathcal{D}$ as a black box yields the following result.

Lemma 3.1. *The compressed representation of Theorem 2.1 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 $R \subseteq [n]$ augmented with a data structure supporting constant-time predecessor queries (or rank queries, as termed in the case of bitvectors [38]), 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 2.1 requires. Note that this bitvector encodes R, so that the $\ell \log n$ bits required in Theorem 2.1 for the representation of the r_js can be dropped.

[4]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.

Corollary 3.1. *Let S be a set of n integers over the universe $[u]$. Given an integer parameter $c \in \{2, \dots, \log u\}$, there exists a compressed representation for S that takes $n(c + 1 + o(1)) + \ell\,(\log u + w)$ bits of space while supporting select in $O(1)$ time. Here, ℓ is the number of segments in the optimal PLA for S of maximum error $\varepsilon = 2^{c-1} - 1$.*

Let us compare the space occupancy of Corollary 3.1 to the one of Elias-Fano, namely $n(\log(u/n) + 2) + o(n)$ bits, as both solutions support constant-time select. The inequality turns out to be

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

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

4 Supporting rank

We can implement $\mathsf{rank}(x)$ via a binary search on $[n]$ to find the largest i such that $\mathsf{select}(i) \le x$. This naïve implementation takes $O(t \log n)$ time, because of the implementation of select in $O(t)$ time by Theorem 2.1.

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 contain x or its predecessor. Formally, we binary search on the interval $[1, \ell]$ to find the largest j such that $\mathsf{select}(r_j) = \lfloor f_j(r_j) \rfloor + C[r_j] \le x$. Second, we binary search on the $r_{j+1} - r_j \le n$ integers compressed by 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 $\mathsf{rank}(x)$.

Surprisingly, we can further speed up rank queries without adding any redundancy on top of the encoding of Theorem 2.1. The key idea is to narrow 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 $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$, 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

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

Since i is an integer, we can round the left and the right side of the last inequality, and then we set

Algorithm 1 Rank implementation by Lemma 4.1

```
Input: $x$, PLA $\{s_1, s_2, \ldots, s_\ell\}$, corrections $C[1\,..\,n]$
Output: Returns rank($x$)
1: Find max $j \in [1, \ell]$ such that $\lfloor f_j(r_j) \rfloor + C[r_j] \le x$ 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 $\lfloor f_j(i) \rfloor + C[i] \le x$ by binary search
7: return $i$
```

$pos = \lfloor (x - \beta_j)/\alpha_j \rfloor + r_j$ and $err = \lceil \varepsilon/\alpha_j \rceil$, so that the searched position i falls in $[pos - err, pos + err]$.

The pseudocode of Algorithm 1 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 $\alpha_j \ge 1$ for every j, because $x_i \in S$ are increasing, and thus the segments have slope at least 1. Consequently, $\varepsilon/\alpha_j \le \varepsilon$ and the range on which we perform the second binary search has size $2\varepsilon < 2^c$, thus that binary search takes $O(\log(\varepsilon/\alpha_j)) = O(c)$ time.

Lemma 4.1. *The compressed representation of Theorem 2.1 supports* rank *queries in $O(\log \ell + c)$ time and no additional space.*

Note that Lemma 4.1 enables the rank operation with the same time bound also on: (i) the compressed representation described in Theorem 2.2 (the one that compresses β_js and r_js), (ii) the representation provided in Lemma 3.1 (the one supporting select in parametric time t), and (iii) the representation provided in Corollary 3.1 (the one supporting select in constant time). Again, the space occupancy of the solution above is the same commented in the previous section, and thus it results better than Elias-Fano if $\ell = O(n/\log n)$.

We can improve the bound of Lemma 4.1 by replacing the binary search of Line 1 of Algorithm 1 with the following predecessor data structure.

Lemma 4.2. ([46]) *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*

$$\mathrm{PT}(u, q, a) = \Theta(\min\{\, \log q/\log\log u,\ \log \tfrac{\log(u/q)}{a},\ \log \tfrac{\log u}{a} / \log(\tfrac{a}{\log q} \cdot \log \tfrac{\log u}{a}),\ \log \tfrac{\log u}{a} / \log(\log \tfrac{\log u}{a} / \log \tfrac{\log q}{a})\}).$$

Let $T = \{\mathsf{select}(r_j) \mid 1 \le j \le \ell\}$ 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 4.2 so that $s = q = \ell/2^c$ and thus $a = \log\log u$. Then, we replace Line 1 of Algorithm 1 with a predecessor search followed by a $O(c)$-time binary search in-between two samples.

Corollary 4.1. *The compressed representation of Theorem 2.1 supports* rank *queries in* $\mathrm{PT}(u, \ell/2^c, \log\log u) + O(c)$ *time and* $O((\ell/2^c)\log u)$ *bits of additional space.*

We can restrict our attention to the first two branches of the PT term, as the latter two are relevant for universe sizes that are super-polynomial in q, i.e. $\log u = \omega(\log q)$. The time complexity of rank in Corollary 4.1 then becomes $O(\min\{\log_w(\ell/2^c), \log\log(u/\ell)\} + c)$, where $w = \Omega(\log u)$ is the word size of the machine and c is a constant which, we recall, denotes the bits reserved to encode the errors in C.

5 Special sequences

For 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.

Theorem 5.1. ([14]) *Let S be a set of n integers over the universe $[u]$, and let $c \ge 2$ be an integer. 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. If $\varepsilon = 2^{c-1} - 1$ is sufficiently larger than σ/μ, then $\ell = n\sigma^2/(\mu\varepsilon)^2$ with high probability.*

Plugging this result into the constant-time select of Corollary 3.1 and rank implementation of Corollary 4.1, we obtain the following result.

Theorem 5.2. *Under the assumptions of Theorem 5.1, there exists a compressed representation of S that supports* select *in $O(1)$ time and* rank *in* $\mathrm{PT}(u, \frac{n\sigma^2}{\mu^2 2^{3c-2}}, \log\log u) + O(c)$ *time within* $n[c + 1 + o(1) + ((1 + 1/2^c)\log u + w)\frac{\sigma^2}{\mu^2 2^{2c-2}}]$ *bits of space with high probability, where $c \in \{2, \ldots, \log u\}$ is a given integer parameter.*

We stress that the data structure of Theorem 5.2 is deterministic. The randomness is over the gaps between consecutive integers of the input data, and the result holds for any probability distribution as long as its mean and variance are finite. Moreover, according to the experiments in [14], the hypotheses of Theorem 5.1 are very realistic in several application scenarios.

Said this, we observe that the hypothesis "$\varepsilon = 2^{c-1} - 1$ is sufficiently larger than σ/μ" implies that the ratio $\sigma/(\mu 2^{c-1})$ is much smaller than 1. Hence, it is reasonable to assume that the space bound in Theorem 5.2 is dominated by $n(c+1)$ bits which results "almost" independent of the universe size while still ensuring constant time select and fast rank operations.

As a final remark, and following the practice of the previous sections, we compare our encoding against Elias-Fano encoding. The space comparison between the factor $c + 1$ and the Elias-Fano factor $\log(u/n)$ is in favour of our encoding as much as the dataset is sparse with respect to the universe size. On the other hand, the time complexity of select is constant time in both cases, whereas the one of rank is better in our case whenever $\log(n\sigma/\mu)$ is asymptotically smaller than $\log(u/n)$, which is indeed $u = \omega(n^2\sigma/\mu)$.

We also mention that some results of the previous sections, such as Corollary 3.1 and Lemma 4.1, showed that our proposal is better than Elias-Fano whenever $\ell = O(n/\log n)$. Since Theorem 5.2 showed that $\ell = \Theta(n/\varepsilon^2)$, we can conclude that our solutions is better than Elias-Fano iff $\varepsilon = \omega(\sqrt{\log n})$, and this holds with high probability under the hypotheses of Theorem 5.2.

6 On optimal partitioning to improve the space

So far, we assumed a fixed number of bits c for correcting each of the n elements in S, which is equivalent to say that the ℓ segments in the PLA guarantee the same error $\varepsilon = 2^{c-1} - 1$ over all the integers in S. However, the input set 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 [6, 16, 43, 52, 55], and it will be specialised in this section to our scenario.

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 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 $w(i, j, c) = (j - i)\,c + b$ bits of space, where c is the bit-size of the corrections, and b is the space taken by the segment in bits (e.g. using the plain encoding of Theorem 2.1 or the compressed encoding of Theorem 2.2). 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 $c_1 = 0$, $c_2 = 2, c_3 = 3, \ldots, c_m = m$, where $m = O(\log u)$ is defined as the minimum correction value that produces one single segment on S. Since each vertex is the source of at most $O(\log u)$ edges, one for each possible value of c, the total number of edges in $\mathcal{G}$ is $O(n \log u)$. It is not difficult to prove the following:

Fact 6.1. *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.*

Such PLA provides a solution to the rank/select dictionary problem which minimises the space occupancy of the approaches stated in Theorems 2.1 and 2.2.

Since $\mathcal{G}$ is a DAG, the shortest path can be computed in $O(n \log u)$ time by taking the vertices in the topological order and by relaxing their outgoing edges. However, one cannot approach the construction of $\mathcal{G}$ in a brute-force manner because this could 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 [42].

To avoid this prohibitive cost, we now propose a greedy algorithm that computes the shortest path on-the-fly by working on a properly defined sub-graph of $\mathcal{G}$, 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 because the graph has one vertex per integer to be compressed.

Consider an edge (i, j) of cost $w(i, j, c)$ in the full graph $\mathcal{G}$. We recall that edge (i, j) 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. Hence, any "suffix-edge" $(i + k, j)$ of (i, j), with $k = 1, \ldots, j - i - 1$, corresponds to a shorter segment that compresses the (suffix-)sequence of integers $x_{i+k}, x_{i+k+1}, \ldots, x_{j-1}$, still using a correction size of c bits per integer. We call *subsumed edge* of (i, j) any suffix-edge $(i + k, j)$ and define its cost as $w(i + k, j, c) = (j - i - k)\,c + b$, for $1 \leq k < j - i$. Note that $(i + k, j)$ may not be an edge of the full graph $\mathcal{G}$ because a segment computed from position $i + k$ with correction size c could end past j, thus including more integers on its right.

Given these definitions, we now describe our greedy algorithm that processes the vertices of $\mathcal{G}$ from left to right while maintaining the following invariant: *each visited vertex i is covered by one segment (i, j) for each value of c, and all these segments form the frontier set J.* The segment starting at i is either a "full" segment with the appropriate value of c (computed by using the algorithm of [42]), or it is a subsumed edge with correction size c (derived from a longer edge that crossed i and had that correction size). The former edges are the "best" among the ones starting from i and with correction size c, but they are costly to be computed; the latter edges are "sub-optimal" but can be derived in constant time. The goal next is to show how to work with full and subsumed edges in a way that the resulting shortest-path algorithm is fast and still computes a path which is (provably) not much longer than the shortest path of the whole graph $\mathcal{G}$.

To show this, we begin from vertex $i = 1$ and compute the m segments that start from i and have correction sizes $c_1, c_2, \ldots, c_m$. We set J as the set of ending positions of those segments: i.e. $J = \{j_1, j_2, \ldots, j_m\}$. As in the classic step of the shortest path algorithm for DAGs, we relax all the edges (i, j) for $j \in J$. This completes the first iteration.

The next iteration picks the smallest vertex in J, say i'. By the invariant on J, we know that it exists the edge (i, i'), whose correction size is e.g. c'. This edge is added to the shortest path under computation, and we set $J' = J \setminus \{i'\}$. We then repeat the computation above to insert in J' an edge with bit-correction c'. Hence, we determine the longest segment from i' having correction-size c' by using the algorithm of [42]. By construction, this new segment (i', j') is an edge of the full graph $\mathcal{G}$; we add j' to J', and then we relax all edges (i', j) with $j \in J'$. We notice that among those edges we have the just-inserted edge (i', j'), plus all subsumed edges (i', j) which crossed i' and ended in j. This completes the second iteration, and the algorithm continues in this way for all vertices $2, \ldots, n+1$. It stops when all the edges ending in the sink vertex $n+1$ are relaxed.

Figure 2 depicts a generic step. We have $m = 5$ possible correction costs because, in this example, we assume that c_5 is the minimum correction value that produces one single segment. The currently examined vertex is i, and the frontier of vertices to examine is $J = \{j_1, j_2, j_3, j_4, j_5\}$. The next vertex to process is $j_3 = \min J$, so $i' = j_3$. We add (i, j_3) to the shortest path with its cost c_3, and we relax the subsumed edges starting from j_3 and ending into $\{j_1, j_2, j_4, j_5\}$ (showed as dashed grey arrows). We also relax the new edge (j_3, j') which is computed via the algorithm of [42].

As far as the space is concerned, the algorithm uses $O(n + |J|) = O(n + \log u) = O(n)$ space at each iteration, since $|J| = O(m) = O(\log u)$. The running time is $O(|J|) = O(\log u)$ per iteration, plus the cost of computing the "full" segment for each extracted vertex from J. This latter cost is $O(n)$ for any given value of c and over all n elements (namely, it is $O(1)$ amortised cost per processed element [42]), thus $O(n \log u)$ over all values of c. In the full paper, we show that this greedy algorithm finds a path containing at most twice the number of edges of the shortest path of the full

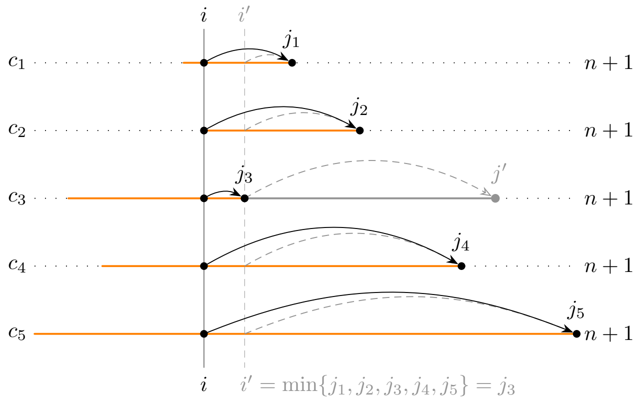

Figure 2: The algorithm of Theorem 6.1 keeps one segment (in orange) crossing the visited vertex i for each value of c, and it relaxes the edges with source i and target set to the end of each segment. The next vertex i' to visit is the minimum of the target vertices from i, and its outgoing edges are shown in dashed grey.

graph $\mathcal{G}$, and that the sum of the weights of this path is at most $(\log u + \log n + w)\,\ell$ bits more than the cost of the shortest path.

THEOREM 6.1. *There exists a greedy algorithm that in $O(n \log u)$ time and $O(n)$ space outputs a path of the weighted DAG $\mathcal{G}$ whose cost is at most $(\log u + \log n + w)\,\ell$ larger than its shortest path.*

Given Fact 6.1, the PLA that corresponds to the path computed by the algorithm above can be used to design a rank/select dictionary which minimises the space occupancy of the solutions based on the approaches of Theorems 2.1 and 2.2. Section 7 will experiment with this approach in terms of space occupancy and time efficiency of rank and select operations.

7 Experiments

7.1 Implementation notes. We store the segments triples (r_j, α_j, β_j) as an array of structures with memory-aligned fields. This allows for better locality and aligned memory accesses. We avoid complex structures on top of the r_js and the select(r_j)s (as suggested by Corollaries 3.1 and 4.1 to asymptotically speed up select and rank, respectively), since in practice the segments are few (see Figure 3) and fit the last-level cache.

Nevertheless, to speed up rank and select, we introduce two small tables of size 2^{16} each that allow accessing in one hop either the segment including a given position or a narrower range of segments to binary search on (details in the full paper). Another algorithm engineering trick was introduced to speed up the binary search over the array C by splitting it in two arrays: one contains the corrections at multiple d positions, the other contains the rest of C's corrections. Experimentally, we found that the best performance is achieved when d is set so that the latter binary search touches roughly four cache lines of correction values (i.e. $d = \lceil 4 \cdot 512/c \rceil$).

The implementation of our rank/select dictionary, which we name *linear approximation vector* (shortly, `la_vector`), was done in C++.[5] We use the notation `la_vector<`c`>`, where c is the correction size; and use `la_vector_opt` to denote our space-optimised dictionary described in Section 6.

All our experiments were run on a machine with 40 GB of RAM and an Intel Xeon E5-2407v2 CPU with a clock rate of 2.40 GHz.

7.2 Baselines. For the experimental comparison, we used the following rank/select dictionaries from the Succinct Data Structures Library (`sdsl`) [20]:

`sd_vector` the Elias-Fano representation for increasing integer sequences with constant-time select [41].

`rrr_vector<`t`>` a practical implementation of the H_0-compressed bitvector of Raman, Raman and Rao with blocks of t bits [9, 49].

`enc_vector<`$\gamma/\delta, s$`>` encodes the gaps between consecutive integers via either Elias γ- or δ-codes. Random access is implemented in `sdsl` by storing, with sampling 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.

To widen our experimental comparison, we also used the following rank/select dictionaries from the Data Structures for Inverted Indexes (`ds2i`) library [43]:

`uniform_partitioned` divides the input into fixed-sized chunks and encodes each chunk with Elias-Fano.

`opt_partitioned` 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.[6]

In both structures above, endpoints and boundary values of the chunks are stored in a separate Elias-Fano.

7.3 Datasets. We tested the software above on lists of integers originating from different applications and datasets. We selected these lists so that their density n/u varied significantly, viz. up to three orders of magnitude. We built all the mentioned rank/select dictionaries and our `la_vector` from these integer lists, and then we measured the bits per integer and the time they took to perform 10^5 `rank` and `select` queries. The universe size u never exceeds $2^{32} - 1$, because the implementations in `ds2i` only support 32-bit integers.

GOV2 is an inverted index built on a collection of about 25M `.gov` sites [43]. We encoded each list in the index separately and averaged the space-time performance over lists of lengths 100K–1M, 1M–10M and $>$10M. This grouping of lists by length induced average densities of 1.29%, 12.26% and 53.03%.

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.[7] On this file, we first applied the Burrows-Wheeler Transform (BWT), as implemented by [12], and then we generated three integer lists by enumerating the positions of the ith most frequent character. The different list sizes (and densities) were achieved by properly setting i, and they were 3.73M (0.36%), 13.56M (1.30%) and 57.98M (5.58%).

5GRAM is a text file of 1.4 GB containing 60M different five-word sequences occurring in books indexed by Google.[8] As for URL, we first applied the BWT and then we generated three integer lists of sizes (densities): 11.21M (0.76%), 29.20M (1.98%) and 145.40M (9.85%).

DNA is the first GB of the human reference genome.[9] 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%).

Figure 3 shows 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. In particular, the following experiments will assume $c \geq 6$, because smaller values of c make the space occupied by the segments significantly larger than the space taken by the correction array C.

[5]The code of `la_vector` and the benchmark to reproduce the experiments are available at https://github.com/gvinciguerra/la_vector and https://github.com/aboffa/Learned-Rank-Select-ALENEX21, respectively.

[6]For a fair comparison with our optimised variant, we disallow hybrids encodings, e.g. chunks compressed with Elias-Fano and others compressed with plain bitvectors.

[7]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.

[8]Available at https://storage.googleapis.com/books/ngrams/books/datasetsv3.html.

[9]Available at https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39.

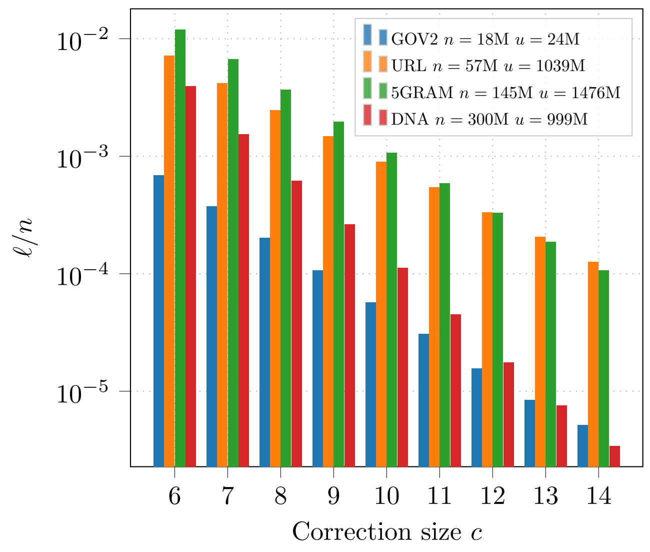

Figure 3: The ratio between the number of segments ℓ and the size n of the largest datasets from our experiments at different correction sizes c.

7.4 Experiments on select. From Figure 4 we notice that our `la_vector<c>` variants consistently provide the best time performance. This comes at the cost of requiring c bits 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 c (see e.g. 5GRAM and DNA, where the Pareto frontier of `la_vector` 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 `ds2i::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>` variants are not dominated by those advanced Elias-Fano variants over all 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 [38, §4.3]).

7.5 Experiments on rank. Figure 5 shows 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 5), `sdsl::rrr_vector` has a very poor space performance, more than 10 bits per integer.

Not surprisingly, `sdsl::enc_vector<`$\cdot, s$`>` provides 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.

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

7.6 Discussion. 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.

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

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 curve of Elias-Fano approaches.

8 Conclusions and future work

For space reasons, we refer the reader to the introduction for a summary of our contributions.

As future work, we plan to investigate the use of vectorised decoding [31] to achieve effective compression and fast scanning of the corrections in C. Moreover, we will study the use of mixed compressors in `la_vector_opt`, as in Partitioned Elias-Fano [43]. Finally, we argue that the greedy algorithm of Section 6 can be improved to be the optimal one by using [7] to dynamically remove points from the beginning of a segment in $O(1)$ amortised time.

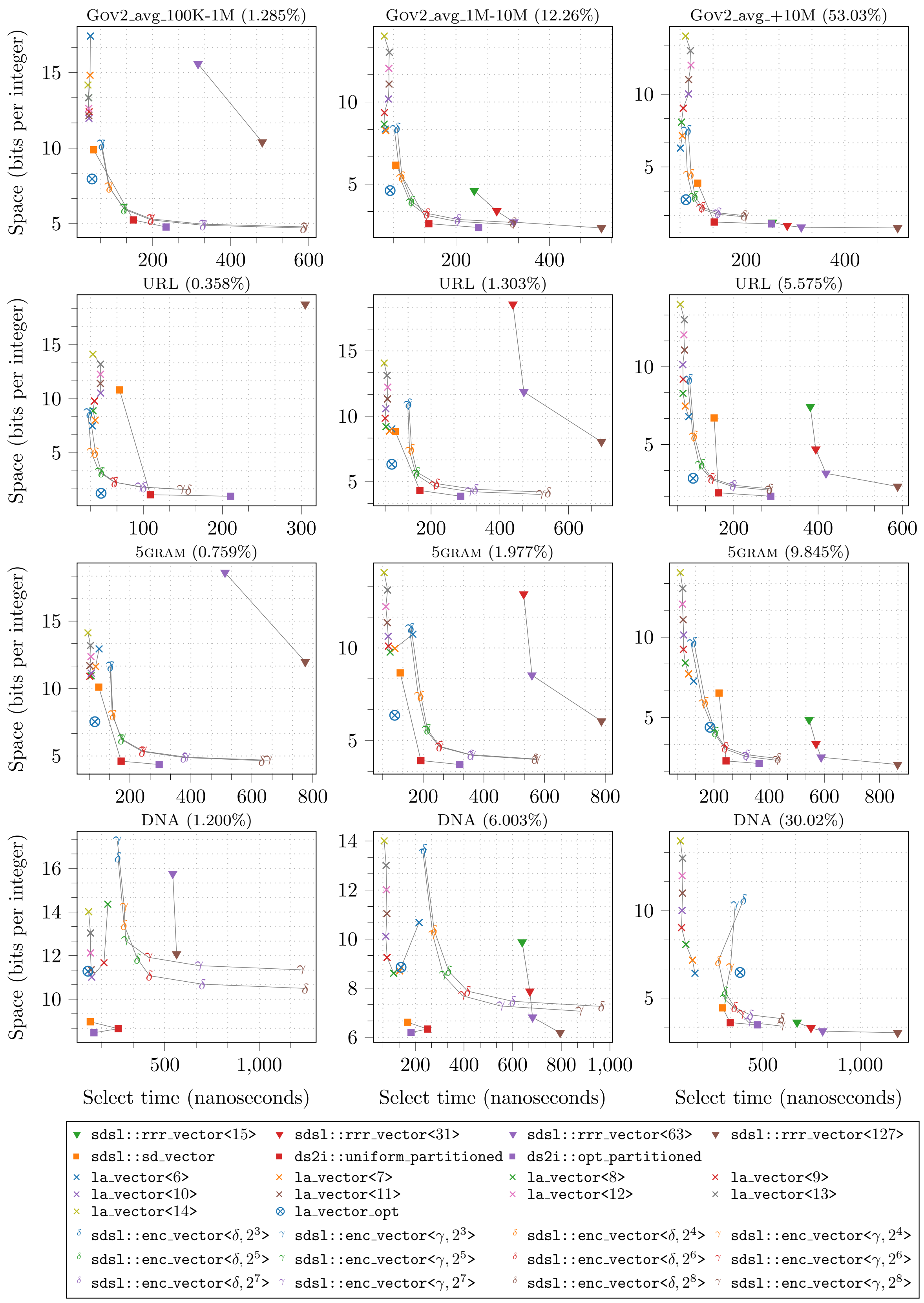

Figure 4: Space-time performance of the select query.

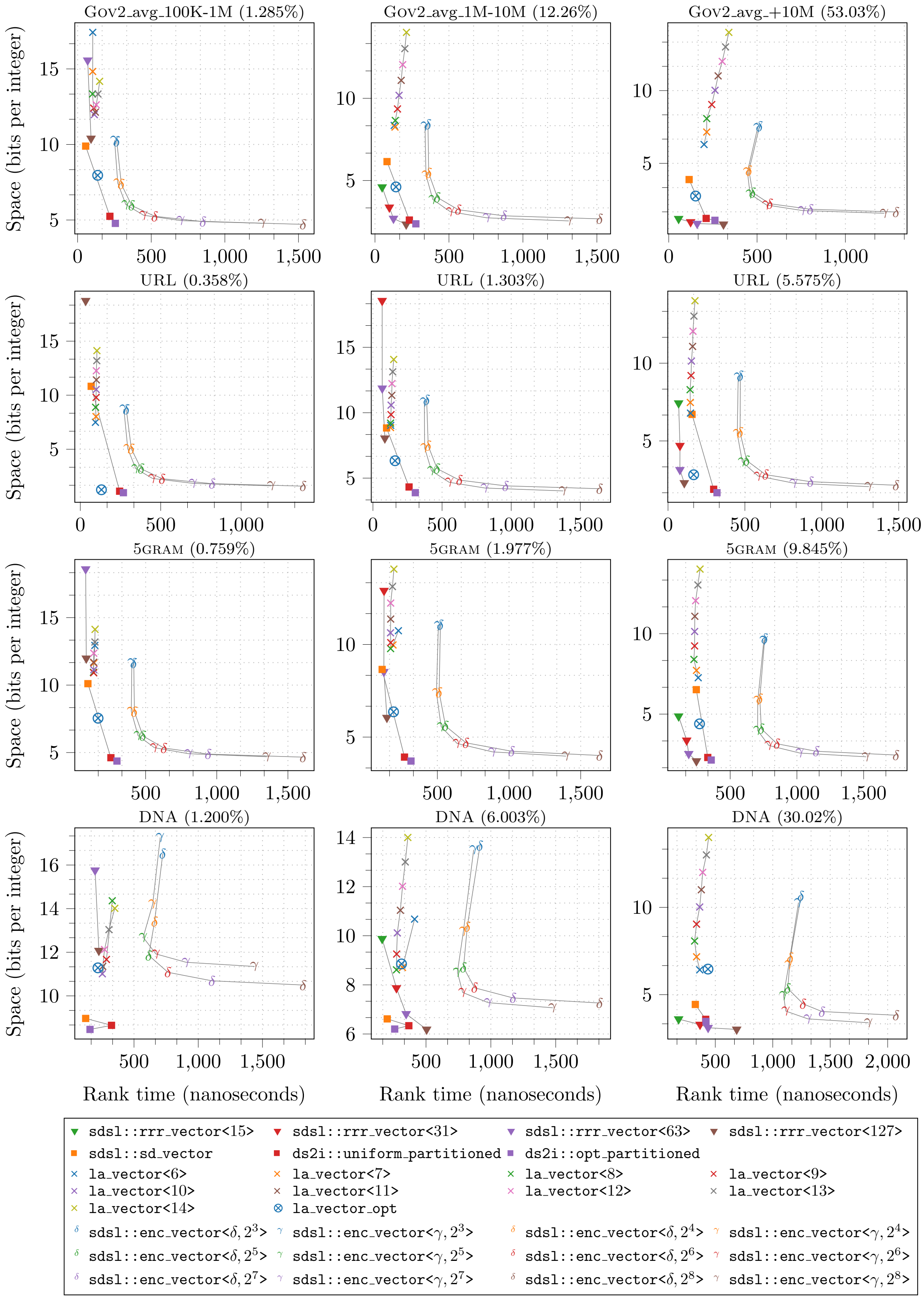

Figure 5: Space-time performance of the rank query.

References

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

[2] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[3] D. Arroyuelo and R. Raman. Adaptive succinctness. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 11811 of *LNCS*, pages 467–481, 2019.

[4] J. Barbay and G. Navarro. Compressed Representations of Permutations, and Applications. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3 of *LIPIcs*, pages 111–122, 2009.

[5] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16, Nov. 2008.

[6] A. L. Buchsbaum, G. S. Fowler, and R. Giancarlo. Improving table compression with combinatorial optimization. *Journal of the ACM*, 50(6):825–851, Nov. 2003.

[7] N. Bus and L. Buzer. Dynamic convex hull for simple polygonal chains in constant amortized time per update. In *Proc. 31th European Workshop on Computational Geometry (EUROCG)*, 2015.

[8] D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[9] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5280 of *LNCS*, 2008.

[10] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, Apr. 1974.

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

[12] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[13] P. Ferragina, S. Kurtz, S. Lonardi, and G. Manzini. Computational biology. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 59. CRC Press, 2 edition, 2018.

[14] P. Ferragina, F. Lillo, and G. Vinciguerra. Why are learned indexes so effective? In *Proc. 37th International Conference on Machine Learning (ICML)*, July 2020.

[15] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, July 2005.

[16] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, 61(1):51–74, 2011.

[17] P. Ferragina and G. Vinciguerra. Learned data structures. In L. Oneto, N. Navarin, A. Sperduti, and D. Anguita, editors, *Recent Trends in Learning From Data*, pages 5–41. Springer, 2020.

[18] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.

[19] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, Oct. 2006.

[20] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 326–337, 2014.

[21] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

[22] A. Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[23] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 368–373, 2006.

[24] A. Golynski, A. Orlandi, R. Raman, and S. S. Rao. Optimal Indexes for Sparse Bit Vectors. *Algorithmica*, 69(4):906–924, Aug. 2014.

[25] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. Poster of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[26] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[27] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

[28] S. Idreos, K. Zoumpatianos, S. Chatterjee, W. Qin, A. Wasay, B. Hentschel, M. Kester, N. Dayan, D. Guo, M. Kang, and Y. Sun. Learning data structure alchemy. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 42(2):46–57, 2019.

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

[30] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proc. 44th International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[31] D. Lemire and L. Boytsov. Decoding billions of integers

per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[32] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

[33] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[34] M. Mitzenmacher. A model for learned Bloom filters and optimizing by sandwiching. In *Proc. 32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[35] J. I. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 37–42, 1996.

[36] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[37] G. Navarro. Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4), Mar. 2014.

[38] G. Navarro. *Compact data structures: a practical approach.* Cambridge University Press, 2016.

[39] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), Apr. 2007.

[40] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *LNCS*, pages 295–306, 2012.

[41] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[42] J. O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, 1981.

[43] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM Conference on Research & Development in Information Retrieval (SIGIR)*, pages 273–282, 2014.

[44] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Transactions on Information Systems*, 36(1), Apr. 2017.

[45] M. Pătraşcu. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[46] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. of the 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[47] M. Pătraşcu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 117–122, 2010.

[48] R. Raman. Rank and select operations on bit strings. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 1772–1775. Springer, 2nd edition, 2016.

[49] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, Nov. 2007.

[50] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[51] C. Silverstein. Google sparsehash. https://github.com/sparsehash/sparsehash.

[52] F. Silvestri and R. Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *Proc. 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1219–1228, 2010.

[53] S. Vigna. Broadword implementation of rank/select queries. In *Proc. 7th International Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 154–168, 2008.

[54] S. Vigna. Quasi-succinct indices. In *Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

[55] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 2nd edition, 1999.

[56] H. Yu. Optimal succinct rank data structure via approximate nonnegative tensor decomposition. In *Proc. of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 955–966, 2019.