Algorithmica (2014) 69:232–268
DOI 10.1007/s00453-012-9726-3

Efficient Fully-Compressed Sequence Representations

Jérémy Barbay · Francisco Claude · Travis Gagie · Gonzalo Navarro · Yakov Nekrich

Received: 4 February 2012 / Accepted: 4 December 2012 / Published online: 15 December 2012

Abstract We present a data structure that stores a sequence $s[1..n]$ over alphabet $[1..\sigma]$ in $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s)+1)$ bits, where $\mathcal{H}_0(s)$ is the zero-order entropy of s. This structure supports the queries access, rank and select, which are fundamental building blocks for many other compressed data structures, in worst-case time $\mathcal{O}(\lg\lg\sigma)$ and average time $\mathcal{O}(\lg\mathcal{H}_0(s))$. The worst-case complexity matches the best previous results, yet these had been achieved with data structures using $n\mathcal{H}_0(s) + o(n\lg\sigma)$ bits. On highly compressible sequences the $o(n\lg\sigma)$ bits of the redundancy may be significant compared to the $n\mathcal{H}_0(s)$ bits that encode the data. Our representation, instead, compresses the redundancy as well. Moreover, our average-case complexity is unprecedented.

An early version of this article appeared in *Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, part II, pp. 215–326, 2010. J. Barbay was funded in part by Fondecyt grants 1-110066 and 1120054, Chile. F. Claude was funded in part by Google PhD Fellowship Program. G. Navarro was funded in part by Fondecyt grant 1-110066, Chile. Work partially done during Y. Nekrich stay at the University of Chile.

J. Barbay · G. Navarro (✉)
Department of Computer Science, University of Chile, Santiago, Chile
e-mail: gnavarro@dcc.uchile.cl

J. Barbay
e-mail: jbarbay@dcc.uchile.cl

F. Claude
David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
e-mail: fclaude@cs.uwaterloo.ca

T. Gagie
Department of Computer Science, Aalto University, Helsinki, Finland
e-mail: travis.gagie@gmail.com

Y. Nekrich
Department of Electrical Engineering & Computer Science, University of Kansas, Kansas, USA
e-mail: yasha@dcc.uchile.cl

Our technique is based on partitioning the alphabet into characters of similar frequency. The subsequence corresponding to each group can then be encoded using fast uncompressed representations without harming the overall compression ratios, even in the redundancy.

The result also improves upon the best current compressed representations of several other data structures. For example, we achieve (i) compressed redundancy, retaining the best time complexities, for the smallest existing full-text self-indexes; (ii) compressed permutations π with times for $\pi()$ and $\pi^{-1}()$ improved to loglogarithmic; and (iii) the first compressed representation of dynamic collections of disjoint sets. We also point out various applications to inverted indexes, suffix arrays, binary relations, and data compressors.

Our structure is practical on large alphabets. Our experiments show that, as predicted by theory, it dominates the space/time tradeoff map of all the sequence representations, both in synthetic and application scenarios.

Keywords Compressed sequence representations · Rank and select on sequences · Compact data structures · Entropy-bounded structures · Compressed text indexing

1 Introduction

A growing number of important applications require data representations that are space-efficient and at the same time support fast query operations. In particular, suitable representations of *sequences* supporting a small set of basic operations yield space- and time-efficient implementations for many other data structures such as full-text indexes [22, 28, 32, 48], labeled trees [3, 4, 20], binary relations [2, 4], permutations [6] and two-dimensional point sets [11, 41], to name a few.

Let $s[1..n]$ be a sequence of characters belonging to alphabet $[1..\sigma]$. In this article we focus on the following set of operations, which is sufficient for many applications:

$s.\mathsf{access}(i)$	returns the ith character of sequence s, which we denote $s[i]$;
$s.\mathsf{rank}_a(i)$	returns the number of occurrences of character a up to position i in s; and
$s.\mathsf{select}_a(i)$	returns the position of the ith occurrence of a in s.

Table 1 shows the best sequence representations and the complexities they achieve for the three queries, where $\mathcal{H}_k(s)$ refers to the k-th order empirical entropy of s [43]. To implement the operations efficiently, the representations require some *redundancy* space on top of the $n\mathcal{H}_0(s)$ or $n\mathcal{H}_k(s)$ bits needed to encode the data. For example, multiary wavelet trees (row 1) represent s within zero-order entropy space plus just $o(n)$ bits of redundancy, and support queries in time $\mathcal{O}(1 + \frac{\lg \sigma}{\lg \lg n})$. This is very attractive for relatively small alphabets, and even constant-time for polylog-sized ones. For large σ, however, all the other representations in the table are exponentially faster, and some even achieve high-order compression. However, their redundancy is higher, $o(n \lg \sigma)$ bits. While this is still asymptotically negligible compared to the size of a plain representation of s, on highly compressible sequences such redundancy is not always negligible compared to the space used to encode the compressed data. This

Table 1 Best previous bounds and our new ones for data structures supporting access, rank and select. The space bound of the form $\mathcal{H}_k(s)$ holds for any $k = o(\log_\sigma n)$, and those of the form $(1+\epsilon)$ hold for any constant $\epsilon > 0$. On average $\lg \sigma$ becomes $\mathcal{H}_0(s)$ in our time complexities (see Corollary 3) and in *row 1* [7, Theorem 5]

	space (bits)	access	rank	select
[29, Theorem 4]	$n\mathcal{H}_0(s) + o(n)$	$\mathcal{O}(1 + \frac{\lg \sigma}{\lg \lg n})$	$\mathcal{O}(1 + \frac{\lg \sigma}{\lg \lg n})$	$\mathcal{O}(1 + \frac{\lg \sigma}{\lg \lg n})$
[4, Lemma 4.1]	$n\mathcal{H}_0(s) + o(n \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
[33, Corollary 2]	$n\mathcal{H}_k(s) + o(n \lg \sigma)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$
[28, Theorem 2.2]	$(1+\epsilon) n \lg \sigma$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
Theorem 2	$n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
Theorem 2	$n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$
Corollary 4	$(1+\epsilon) n\mathcal{H}_0(s) + o(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$

raises the challenge of retaining the efficient support for the queries *while compressing the index redundancy* as well.

In this paper we solve this challenge in the case of zero-order entropy compression, that is, the redundancy of our data structure is asymptotically negligible compared to the zero-order compressed text size (not only compared to the plain text size), plus $o(n)$ bits. The worst-case time our structure achieves is $\mathcal{O}(\lg \lg \sigma)$, which matches the best previous results for large σ. Moreover, the average time is *logarithmic on the entropy of the sequence*, $\mathcal{O}(\lg \mathcal{H}_0(s))$, under reasonable assumptions on the query distribution. This average time complexity is also unprecedented: the only previous entropy-adaptive time complexities we are aware of come from Huffman-shaped wavelet trees [32], which have recently been shown capable of achieving $\mathcal{O}(1 + \frac{\mathcal{H}_0(s)}{\lg \lg n})$ query time with just $o(n)$ bits of redundancy [7, Theorem 5].

Our technique is described in Sect. 3. It can be summarized as partitioning the alphabet into sub-alphabets that group characters of similar frequency in s, storing in a multiary wavelet tree [22] the sequence of sub-alphabet identifiers, and storing separate sequences for each sub-alphabet, containing the subsequence of s formed by the characters of that sub-alphabet. Golynski et al.'s [28] or Grossi et al.'s [33] structures are used for these subsequences, depending on the tradeoff to be achieved. We show that it is sufficient to achieve compression in the multiary wavelet tree, while benefitting from fast operations on the representations of the subsequences.

The idea of alphabet partitioning is not new. It has been used in practical scenarios such as fax encoding, JPEG and MPEG formats [36, 52], and in other image coding methods [51], with the aim of speeding up decompression: only the (short) sub-alphabet identifier is encoded with a sophisticated (and slow) method, whereas the sub-alphabet characters are encoded with a simple and fast encoder (even in plain form). Said [59] gave a more formal treatment to this concept, and designed a dynamic programming algorithm to find the optimal partitioning given the desired number of sub-alphabets, that is, the one minimizing the redundancy with respect to the zero-order entropy of the sequence. He proved that an optimal partitioning defines sub-alphabets according to ranges of character frequencies, which reduces the cost of finding such partitioning to polynomial time and space (more precisely, quadratic on the alphabet size).

Our contribution in this article is, on one hand, to show that a particular way to define the sub-alphabets, according to a quantization of the logarithms of the inverse probabilities of the characters, achieves $o(\mathcal{H}_0(s)+1)$ bits of redundancy per character of the sequence s. This value, in particular, upper bounds the coding efficiency of Said's optimal partitioning method. On the other hand, we apply the idea to sequence data structures supporting operations access/rank/select, thus achieving efficient support of indexed operations on the sequence, not only fast decoding.

We also consider various extensions and applications of our main result. In Sect. 4 we show how our result can be used to improve an existing text index that achieves k-th order entropy [4, 22], so as to improve its redundancy and query times. In this way we achieve the first self-index with space bounded by $n\mathcal{H}_k(s)+o(n)(\mathcal{H}_k(s)+1)$ bits, for any $k=o(\log_\sigma n)$, able to count and locate pattern occurrences and extract any segment of s within the time complexities achieved by its fastest predecessors. We also achieve new space/time tradeoffs for inverted indexes and binary relations. In Sects. 5 and 6 we show how to apply our data structure to store a compressed permutation and a compressed function, respectively, supporting direct and inverse applications and in some cases improving upon previous results [6, 7, 38, 47]. We describe further applications to text indexes and binary relations. In particular, an application of our structure on permutations, at the end of Sect. 5, achieves for the first time compressed redundancy to store function Ψ of text indexes [32, 35, 57]. Section 7 shows how to maintain a dynamic collection of disjoint sets, while supporting operations union and find, in compressed form. This is, to the best of our knowledge, the first result of this kind.

2 Related Work

Sampling A basic attempt to provide rank and select functionality on a sequence $s[1..n]$ over alphabet $[1..\sigma]$ is to store s in plain form and the values $s.\mathsf{rank}_a(k\cdot i)$ for all $a\in[1..\sigma]$ and $i\in[1..n/k]$, where $k\in[1..n]$ is a sampling parameter. This yields constant-time access, $\mathcal{O}(k/\log_\sigma n)$ time rank, and $\mathcal{O}(k/\log_\sigma n+\lg\lg n)$ time select if we process $\Theta(\log_\sigma n)$ characters of s in constant time using universal tables, and organize the rank values for each character in predecessor data structures. The total space is $n\lg\sigma+\mathcal{O}((n/k)\sigma\lg n)$. For example, we can choose $k=\sigma\lg n$ to achieve total space $n\lg\sigma+\mathcal{O}(n)$ (that is, the data plus the redundancy space). Within this space we can achieve time complexity $\mathcal{O}(\sigma\lg\sigma)$ for rank and $\mathcal{O}(\sigma\lg\sigma+\lg\lg n)$ for select.

Succinct Indexes The previous construction separates the sequence data from the "index", that is, the extra data structures to provide fast rank and select. There are much more sophisticated representations for the sequence data that offer constant-time access to $\Theta(\log_\sigma n)$ consecutive characters of s (i.e., just as if s were stored in plain form), yet achieving $n\mathcal{H}_k(s)+o(n\lg\sigma)$ bits of space, for any $k=o(\log_\sigma n)$ [23, 31, 58]. We recall that $\mathcal{H}_k(s)$ is the k-th order empirical entropy of s [43], a lower bound to the space achieved by any statistical compressor that models the character probabilities using the context of their k preceding characters, so $0\le\mathcal{H}_k(s)\le$

$\mathcal{H}_{k-1}(s) \le H_0(s) \le \lg\sigma$. Combining such sequence representations with sophisticated indexes that require $o(n\lg\sigma)$ bits of redundancy [4, 33] (i.e., they are "succinct"), we obtain results like row 3 of Table 1.

Bitmaps A different alternative is to maintain one bitmap $b_a[1..n]$ per character $a \in [1..\sigma]$, marking with a 1 the positions i where $s[i] = a$. Then $s.\mathsf{rank}_a(i) = b_a.\mathsf{rank}_1(i)$ and $s.\mathsf{select}_a(j) = b_a.\mathsf{select}_1(j)$. The bitmaps can be represented in compressed form using "Fully Indexable Dictionaries" (FIDs) [55], so that they operate in constant time and the total space is $n\mathcal{H}_0(s) + \mathcal{O}(n) + o(\sigma n)$ bits. Even with space-optimal FIDs [53, 54], this space is $n\mathcal{H}_0(s) + \mathcal{O}(n) + \mathcal{O}(\frac{\sigma n}{\lg^c n})$ (and the time is $\mathcal{O}(c)$) for any constant c, which is acceptable only for polylog-sized alphabets, that is, $\sigma = \mathcal{O}(\mathrm{polylog}(n))$. An alternative is to use weaker compressed bitmap representations [35, 50] that can support select_1 in constant time and rank_1 in time $\mathcal{O}(\lg n)$, and yield an overall space of $n\mathcal{H}_0(s) + \mathcal{O}(n)$ bits. This can be considered as a succinct index over a given sequence representation, or we can note that we can actually solve $s.\mathsf{access}(i)$ by probing all the bitmaps $b_a.\mathsf{access}(i)$. Although this takes at least $\mathcal{O}(\sigma)$ time, it is a simple illustration of another concept: rather than storing an independent index on top of the data, the data is represented in a way that provides access, rank and select operations with reasonable efficiency.

Wavelet Trees The wavelet tree [32] is a structure integrating data and index, that provides more balanced time complexities. It is a balanced binary tree with one leaf per alphabet character, and storing bitmaps in its internal nodes, where constant-time rank and select operations are supported. By using FIDs [55] to represent those bitmaps, wavelet trees achieve $n\mathcal{H}_0(s) + \mathcal{O}(\frac{n\lg\sigma\lg\lg n}{\lg n})$ bits of space and support all three operations in time proportional to their height, $\mathcal{O}(\lg\sigma)$. Multiary wavelet trees [22] replace the bitmaps by sequences over sublogarithmic-sized alphabets $[1..\sigma']$, $\sigma' = \mathcal{O}(\lg^\epsilon n)$ for $0 < \epsilon < 1$, in order to reduce that height. The FID technique is extended to alphabets of those sizes while retaining constant times. Multiary wavelet trees obtain the same space as the binary ones, but their time complexities are reduced by an $\mathcal{O}(\lg\lg n)$ factor. Indeed, if σ is small enough, $\sigma = \mathcal{O}(\mathrm{polylog}(n))$, the tree height is a constant and so are all the query times. Recently, the redundancy of multiary (and binary) wavelet trees has been reduced to just $o(n)$ [29], which yields the results in the first row of Table 1.[1]

Huffman-Shaped Wavelet Trees Another alternative to obtain zero-order compression is to give Huffman shape to the wavelet tree [32]. This structure uses $n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s)) + \mathcal{O}(n)$ bits even if the internal nodes use a plain representation, using $|b| + o(|b|)$ bits [13, 46], for their bitmaps b. Limiting the height to $\mathcal{O}(\lg\sigma)$ retains the worst-case times of the balanced version and also the given space [7]. In order to reduce the time complexities by an $\mathcal{O}(\lg\lg n)$ factor, we can build multiary wavelet

[1] Because of these good results on polylog-sized alphabets, we focus on larger alphabets in this article, and therefore do not distinguish between redundancies of the form $o(n)\lg\sigma$ and $no(\lg\sigma)$, writing $o(n\lg\sigma)$ for all. See also footnote 6 of Barbay et al. [4].

trees over multiary Huffman trees [39]. This can be combined with the improved representation for sequences over small alphabets [29] so as to retain the $n\mathcal{H}_0(s) + o(n)$ bits of space and $\mathcal{O}(1 + \frac{\lg \sigma}{\lg \lg n})$ worst-case times of balanced multiary wavelet trees. The interesting aspect of using Huffman-shaped trees is that, if the access queries distribute uniformly over the text positions, and the character arguments a to rank_a and select_a are chosen according to their frequency in s, then the average time complexities are $\mathcal{O}(1 + \frac{\mathcal{H}_0(s)}{\lg \lg n})$, the weighted leaf depth. This result [7, Theorem 5] improves upon the multiary wavelet tree representation [29] in the average case. We note that this result [7] involves $\mathcal{O}(\sigma \lg n)$ extra bits of space redundancy, which is negligible only for $\sigma = o(n/\lg n)$.

Reducing to Permutations A totally different sequence representation [28] improves the times to poly-loglogarithmic on σ, that is, exponentially faster than multiary wavelet trees when σ is large enough. Yet, this representation requires again uncompressed space, $n \lg \sigma + \mathcal{O}(\frac{n \lg \sigma}{\lg \lg \sigma})$.[2] It cuts the sequence into chunks of length σ and represents each chunk using a permutation π (which acts as an inverted index of the characters in the chunk). As both operations $\pi()$ and $\pi^{-1}()$ are needed, a representation [47] that stores the permutation within $(1 + \epsilon)\sigma \lg \sigma$ bits and computes $\pi()$ in constant time and $\pi^{-1}()$ in time $\mathcal{O}(1/\epsilon)$ is used. Depending on whether π or π^{-1} is represented explicitly, constant time is achieved for select or for access. Using a constant value for ϵ yields a slightly larger representation that solves both access and select in constant time. Later, the space of this representation was reduced to $n\mathcal{H}_0(s) + o(n \lg \sigma)$ bits while retaining the time complexities of one of the variants (constant-time select) [4]. In turn, the variant offering constant-time access was superseded by the index of Grossi et al. [33], which achieves high-order compression and also improves upon a slower alternative that takes the same space [4]. The best current times are either constant or $\mathcal{O}(\lg \lg \sigma)$. We summarize them in rows 2 to 4.

Our contribution, in rows 5 to 7 of Table 1, is to retain times loglogarithmic on σ, as in rows 2 to 4, while compressing the redundancy space. This is achieved only with space $\mathcal{H}_0(s)$, not $\mathcal{H}_k(s)$. We also achieve average times depending on $\mathcal{H}_0(s)$ instead of $\lg \sigma$.

3 Alphabet Partitioning

Let $s[1..n]$ be a sequence over effective alphabet $[1..\sigma]$.[3] We represent s using an alphabet partitioning scheme. Our data structure has three components:

1. A character mapping $m[1..\sigma]$ that separates the alphabet into sub-alphabets. That is, m is the sequence assigning to each character $a \in [1..\sigma]$ the sub-alphabet

$$m[a] = \lceil \lg(n/|s|_a) \lg n \rceil,$$

[2]The representation actually compresses to the k-th order entropy of a different sequence, not s (A. Golynski, personal communication).

[3]By *effective* we mean that every character appears in s, and thus $\sigma \le n$. In Sect. 3.3 we handle the case of larger alphabets.

where $|s|_a$ denotes the number of occurrences of character a in s; note that $m[a] \leq \lceil \lg^2 n \rceil$ for any $a \in [1..\sigma]$.

2. The sequence $t[1..n]$ of the sub-alphabets assigned to each character in s. That is, t is the sequence over $[1..\lceil \lg^2 n \rceil]$ obtained from s by replacing each occurrence of a by $m[a]$, namely $t[i] = m[s[i]]$.
3. The subsequences $s_\ell[1..\sigma_\ell]$ of characters of each sub-alphabet. For $0 \leq \ell \leq \lceil \lg^2 n \rceil$, let $\sigma_\ell = |m|_\ell$, that is, the number of distinct characters of s replaced by ℓ in t. Then $s_\ell[1..|t|_\ell]$ is the sequence over $[1..\sigma_\ell]$ defined by

$$s_\ell\big[t.\mathsf{rank}_\ell(i)\big] = m.\mathsf{rank}_\ell\big(s[i]\big),$$

for all $1 \leq i \leq n$ such that $t[i] = \ell$.

Example 1 Let $s =$ `"alabar a la alabarda"`. Then $n = 20$ and $|s|_{\mathtt{a}} = 9$, $|s|_{\mathtt{l}} = |s|_{\mathtt{"}} = 3$, $|s|_{\mathtt{b}} = |s|_{\mathtt{r}} = 2$, and $|s|_{\mathtt{d}} = 1$. Accordingly, we define the mapping as $m[\mathtt{a}] = 5$, $m[\mathtt{l}] = m[\mathtt{"}] = 12$, $m[\mathtt{b}] = m[\mathtt{r}] = 15$, and $m[\mathtt{d}] = 19$. As this is the effective alphabet, and assuming that the order is `"",a,b,d,l,r"`, we have $m = (\mathtt{12, 5, 15, 19, 12, 15})$. So the sequence of sub-alphabet identifiers is $t[1..20] = (\mathtt{5, 12, 5, 15, 5, 15, 12, 5, 12, 12, 5, 12, 5, 12, 5, 15, 5, 15, 19, 5})$, and the subsequences are $s_5 = (\mathtt{1, 1, 1, 1, 1, 1, 1, 1, 1})$, $s_{12} = (\mathtt{2, 1, 1, 2, 1, 2})$, $s_{15} = (\mathtt{1, 2, 1, 2})$, and $s_{19} = (\mathtt{1})$.

With these data structures we can implement the queries on s as follows:

$$s.\mathsf{access}(i) = m.\mathsf{select}_\ell\big(s_\ell.\mathsf{access}\big(t.\mathsf{rank}_\ell(i)\big)\big), \quad \text{where } \ell = t.\mathsf{access}(i);$$

$$s.\mathsf{rank}_a(i) = s_\ell.\mathsf{rank}_c\big(t.\mathsf{rank}_\ell(i)\big), \quad \text{where } \ell = m.\mathsf{access}(a) \text{ and } c = m.\mathsf{rank}_\ell(a);$$

$$s.\mathsf{select}_a(i) = t.\mathsf{select}_\ell\big(s_\ell.\mathsf{select}_c(i)\big), \quad \text{where } \ell = m.\mathsf{access}(a) \text{ and } c = m.\mathsf{rank}_\ell(a).$$

Example 2 In the representation of Example 1, we solve $s.\mathsf{access}(6)$ by first computing $\ell = t.\mathsf{access}(6) = \mathtt{15}$ and then $m.\mathsf{select}_{15}(s_{15}.\mathsf{access}(t.\mathsf{rank}_{15}(6))) = m.\mathsf{select}_{15}(s_{15}.\mathsf{access}(2)) = m.\mathsf{select}_{15}(2) = \mathtt{r}$. Similarly, to solve $s.\mathsf{rank}_{\mathtt{l}}(14)$ we compute $\ell = m.\mathsf{access}(\mathtt{l}) = 12$ and $c = m.\mathsf{rank}_{12}(\mathtt{l}) = 2$. Then we return $s_{12}.\mathsf{rank}_2(t.\mathsf{rank}_{12}(14)) = s_{12}.\mathsf{rank}_2(6) = 3$. Finally, to solve $s.\mathsf{select}_{\mathtt{r}}(2)$, we compute $\ell = m.\mathsf{access}(\mathtt{r}) = 15$ and $c = m.\mathsf{rank}_{15}(\mathtt{r}) = 2$, and return $t.\mathsf{select}_{15}(s_{15}.\mathsf{select}_2(2)) = t.\mathsf{select}_{15}(4) = 18$.

3.1 Space Analysis

Recall that the zero-order entropy of $s[1..n]$ is defined as

$$\mathcal{H}_0(s) = \sum_{a \in [1..\sigma]} \frac{|s|_a}{n} \lg \frac{n}{|s|_a}. \tag{1}$$

Recall also that, by convexity, $n\mathcal{H}_0(s) \geq (\sigma - 1)\lg n + (n - \sigma + 1)\lg \frac{n}{n-\sigma+1}$. The next lemma gives the key result for the space analysis.

Lemma 1 *Let s, t, σ_ℓ and s_ℓ be as defined above. Then $n\mathcal{H}_0(t) + \sum_\ell |s_\ell| \lg \sigma_\ell \in n\mathcal{H}_0(s) + o(n)$.*

Proof First notice that, for any character $1 \le \ell \le \lceil \lg^2 n \rceil$ it holds that

$$\sum_{c,\ell=m[c]} |s|_c = |s_\ell|. \tag{2}$$

Now notice that, if $m[a] = m[b] = \ell$, then

$$\begin{aligned} \ell &= \lceil \lg(n/|s|_a) \lg n \rceil = \lceil \lg(n/|s|_b) \lg n \rceil, \\ &\text{therefore } \lg(n/|s|_b) - \lg(n/|s|_a) < 1/\lg n, \\ &\text{and so } |s|_a < 2^{1/\lg n} |s|_b. \end{aligned} \tag{3}$$

Now, fix a, call $\ell = m[a]$, and sum Eq. (3) over all those b such that $m[b] = \ell$. The second step uses Eq. (2):

$$\begin{aligned} \sum_{b,\ell=m[b]} |s|_a &< \sum_{b,\ell=m[b]} 2^{1/\lg n} |s|_b, \\ \sigma_\ell |s|_a &< 2^{1/\lg n} |s_\ell|, \\ \sigma_\ell &< 2^{1/\lg n} |s_\ell| / |s|_a. \end{aligned} \tag{4}$$

Since $\sum_a |s|_a = \sum_\ell |s_\ell| = n$, we have, using Eq. (1), (2), and (4),

$$\begin{aligned} & n\mathcal{H}_0(t) + \sum_\ell |s_\ell| \lg \sigma_\ell \\ &\quad = \sum_\ell |s_\ell| \lg(n/|s_\ell|) + \sum_\ell \sum_{a,\ell=m[a]} |s|_a \lg \sigma_\ell \\ &\quad < \sum_\ell \sum_{a,\ell=m[a]} |s|_a \lg(n/|s_\ell|) + \sum_\ell \sum_{a,\ell=m[a]} |s|_a \lg\bigl(2^{1/\lg n} |s_\ell| / |s|_a\bigr) \\ &\quad = \sum_\ell \sum_{a,\ell=m[a]} |s|_a \lg(n/|s|_a) + \sum_\ell \sum_{a,\ell=m[a]} |s|_a / \lg n \\ &\quad = \sum_a |s|_a \lg(n/|s|_a) + n/\lg n \\ &\quad \in n\mathcal{H}_0(s) + o(n). \end{aligned}$$

□

In other words, if we represent t with $\mathcal{H}_0(t)$ bits per character and each s_ℓ with $\lg \sigma_\ell$ bits per character, we achieve a good overall compression. Thus we can obtain a very compact representation of a sequence s by storing a compact representation of t and storing each s_ℓ as an "uncompressed" sequence over an alphabet of size σ_ℓ.

3.2 Concrete Representation

We represent t and m as multiary wavelet trees [22]; we represent each s_ℓ as either a multiary wavelet tree or an instance of Golynski et al.'s [28, Theorem 2.2] access/rank/select data structure, depending on whether $\sigma_\ell \le \lg n$ or not. The wavelet tree for t uses at most $n\mathcal{H}_0(t) + \mathcal{O}(\frac{n(\lg\lg n)^2}{\lg n})$ bits and operates in constant time, because its alphabet size is polylogarithmic (i.e., $\lceil \lg^2 n \rceil$). If s_ℓ is represented as a wavelet tree, it uses at most $|s_\ell|\mathcal{H}_0(s_\ell) + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell \lg\lg n}{\lg n})$ bits[4] and again operates in constant time because $\sigma_\ell \le \lg n$; otherwise it uses at most $|s_\ell|\lg\sigma_\ell + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\sigma_\ell}) \le |s_\ell|\lg\sigma_\ell + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\lg n})$ bits (the latter because $\sigma_\ell > \lg n$). Thus in either case the space for s_ℓ is bounded by $|s_\ell|\lg\sigma_\ell + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\lg n})$ bits. Finally, since m is a sequence of length σ over an alphabet of size $\lceil \lg^2 n \rceil$, the wavelet tree for m takes $\mathcal{O}(\sigma \lg\lg n)$ bits and also operates in constant time. Because of the convexity property we referred to in the beginning of this section, $n\mathcal{H}_0(s) \ge (\sigma - 1)\lg n$, the space for m is $\mathcal{O}(\frac{n\lg\lg n}{\lg n}) \cdot \mathcal{H}_0(s)$.

Therefore we have $n\mathcal{H}_0(t) + o(n)$ bits for t, $\sum_\ell |s_\ell|\lg\sigma_\ell(1 + \mathcal{O}(\frac{1}{\lg\lg\lg n}))$ bits for the s_ℓ sequences, and $o(n)\mathcal{H}_0(s)$ bits for m. Using Lemma 1, this adds up to $n\mathcal{H}_0(s) + o(n)\mathcal{H}_0(s) + o(n)$, where the $o(n)$ term is $\mathcal{O}(\frac{n}{\lg\lg\lg n})$.

Using the variant of Golynski et al.'s data structure [28, Theorem 4.2], that gives constant-time select, and $\mathcal{O}(\lg\lg\sigma)$ time for rank and access, we obtain our first result in Table 1(row 4). To obtain our second result (row 5), we use instead Grossi et al.'s result [33, Corollary 2], which gives constant-time access, and $\mathcal{O}(\lg\lg\sigma)$ time for rank and select. We note that their structure takes space $|s_\ell|\mathcal{H}_k(s_\ell) + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\sigma_\ell})$, yet we only need this to be at most $|s_\ell|\lg\sigma_\ell + \mathcal{O}(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\lg n})$.

Theorem 2 *We can store* $s[1..n]$ *over effective alphabet* $[1..\sigma]$ *in* $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1)$ *bits and support* access, rank *and* select *queries in* $\mathcal{O}(\lg\lg\sigma)$, $\mathcal{O}(\lg\lg\sigma)$, *and* $\mathcal{O}(1)$ *time, respectively* (*variant* (i)); *or in* $\mathcal{O}(1)$, $\mathcal{O}(\lg\lg\sigma)$ *and* $\mathcal{O}(\lg\lg\sigma)$ *time, respectively* (*variant* (ii)).

We can refine the time complexity by noticing that the only non-constant times are due to operating on some sequence s_ℓ, where the alphabet is of size $\sigma_\ell < 2^{1/\lg n}|s_\ell|/|s|_a$, where a is the character in question, thus $\lg\lg\sigma_\ell = \mathcal{O}(\lg\lg(n/|s|_a))$. If we assume that the characters a used in queries distribute with the same frequencies as in sequence s (e.g., access queries refer to randomly chosen positions in s), then the average query time becomes $\mathcal{O}(\sum_a \frac{|s|_a}{n}\lg\lg\frac{n}{|s|_a}) = \mathcal{O}(\lg\mathcal{H}_0(s))$ by the log-sum inequality.[5]

[4]This is achieved by using block sizes of length $\frac{\lg n}{2}$ and not $\frac{\lg|s_\ell|}{2}$, at the price of storing universal tables of size $\mathcal{O}(\sqrt{n}\text{polylog}(n)) = o(n)$ bits. Therefore all of our $o(\cdot)$ expressions involving n and other variables will be asymptotic in n.

[5]Given σ pairs of numbers $a_i, b_i > 0$, it holds that $\sum a_i \lg\frac{a_i}{b_i} \ge (\sum a_i)\lg\frac{\sum a_i}{\sum b_i}$. Use $a_i = |s|_i/n$ and $b_i = -a_i \lg a_i$ to obtain the result.

Corollary 3 *The $\mathcal{O}(\lg\lg\sigma)$ time complexities in Theorem 2 are also $\mathcal{O}(\lg\lg(n/|s|_a))$, where a stands for $s[i]$ in the* access *query, and for the character argument in the* rank_a *and* select_a *queries. If these characters a distribute on queries with the same frequencies as s, the average time complexity for those operations is $\mathcal{O}(\lg\mathcal{H}_0(s))$.*

Finally, to obtain our last result in Table 1 we use again Golynski et al.'s representation [28, Theorem 4.2]. Given $\epsilon|s_\ell|\lg\sigma_\ell$ extra space to store the inverse of a permutation inside chunks, it answers select queries in time $\mathcal{O}(1)$ and access queries in time $\mathcal{O}(1/\epsilon)$ (these two complexities can be interchanged), and rank queries in time $\mathcal{O}(\lg\lg\sigma_\ell)$. While we initially considered $1/\epsilon = \lg\lg\sigma_\ell$ to achieve the main result, using a constant ϵ yields constant-time select and access simultaneously.

Corollary 4 *We can store $s[1..n]$ over effective alphabet $[1..\sigma]$ in $(1+\epsilon)n\mathcal{H}_0(s) + o(n)$ bits, for any constant $\epsilon > 0$, and support* access, rank_a *and* select *queries in $\mathcal{O}(1/\epsilon)$, $\mathcal{O}(\lg\lg\min(\sigma, n/|s|_a))$, and $\mathcal{O}(1)$ time, respectively (variant* (i)); *or in $\mathcal{O}(1)$, $\mathcal{O}(\lg\lg\min(\sigma, n/|s|_a))$, and $\mathcal{O}(1/\epsilon)$, respectively (variant* (ii)).

3.3 Handling Arbitrary Alphabets

In the most general case, s is a sequence over an alphabet Σ that is not an effective alphabet, and σ characters from Σ occur in s. Let Σ' be the set of elements that occur in s; we can map characters from Σ' to elements of $[1..\sigma]$ by replacing each $a \in \Sigma'$ with its rank in Σ'. All elements of Σ' are stored in the "indexed dictionary" (ID) data structure described by Raman et al. [55], so that the following queries are supported in constant time: for any $a \in \Sigma'$ its rank in Σ' can be found (for any $a \notin \Sigma'$ the answer is -1); and for any $i \in [1..\sigma]$ the i-th smallest element in Σ' can be found. The ID structure uses $\sigma\lg(e\mu/\sigma) + o(\sigma) + \mathcal{O}(\lg\lg\mu)$ bits of space, where e is the base of the natural logarithm and μ is the maximal element in Σ'; the value of μ can be specified with additional $\mathcal{O}(\lg\mu)$ bits. We replace every element in s by its rank in Σ', and the resulting sequence is stored using Theorem 2. Hence, in the general case the space usage is increased by $\sigma\lg(e\mu/\sigma) + o(\sigma) + \mathcal{O}(\lg\mu)$ bits and the asymptotic time complexity of queries remains unchanged. Since we are already spending $\mathcal{O}(\sigma\lg\lg n)$ bits in our data structure, this increases the given space only by $\mathcal{O}(\sigma\lg(\mu/\sigma))$.

3.4 Application to Fast Encode/Decode

Given a sequence s to encode, we can build mapping m from its character frequencies $|s|_a$, and then encode each $s[i]$ as the pair $(m[s[i]], m.\mathsf{rank}_{m[s[i]]}(s[i]))$. Lemma 1 (and some of the discussion that follows in Sect. 3.2) shows that the overall output size is $n\mathcal{H}_0(s) + o(n)$ bits if we represent the sequence of pairs by partitioning it into three sequences: (1) the left part of the pairs in one sequence, using Huffman coding on chunks (see next); (2) the right part of the pairs corresponding to values where $\sigma_\ell < \lg n$ in a second sequence, using Huffman coding on chunks; (3) the remaining right parts of the pairs, using plain encoding in $\lceil\lg\sigma_\ell\rceil$ bits (note $\sigma_\ell = m.\mathsf{rank}_\ell(\sigma)$). The Huffman coding on chunks groups $\frac{\lg n}{4\lg\lg n}$ characters, so that even in the case of the left parts, where the alphabet is of size $\lceil \lg^2 n \rceil$, the total length of a chunk is at most $\frac{\lg n}{2}$ bits, and hence the Huffman coding table occupies just $\mathcal{O}(\sqrt{n} \lg n)$ bits. The redundancy on top of $\mathcal{H}_0(s)$ adds up to $\mathcal{O}(\frac{n \lg \lg n}{\lg n})$ bits in sequences (1) and (2) (one bit of Huffman redundancy per chunk) and $\mathcal{O}(\frac{n}{\lg \lg n})$ in sequence (3) (one bit, coming from the ceil function, per $\lg \sigma_\ell > \lg \lg n$ encoded bits).

The overall encoding time is $\mathcal{O}(n)$. A pair (ℓ, o) is decoded as $s[i] = m.\mathsf{select}_\ell(o)$, where after reading ℓ we can compute σ_ℓ to determine whether o is encoded in sequence (2) or (3). Thus decoding also takes constant time if we can decode Huffman codes in constant time. This can be achieved by using canonical codes and limiting the height of the tree [24, 45].

This construction gives an interesting space/time tradeoff with respect to classical alternatives. Using just Huffman coding yields $\mathcal{O}(n)$ encoding/decoding time, but only guarantees $n\mathcal{H}_0(s) + \mathcal{O}(n)$ bits of space. Using arithmetic coding achieves $n\mathcal{H}_0(s) + \mathcal{O}(1)$ bits, but encoding/decoding is not linear-time. The tradeoff given by our encoding, $n\mathcal{H}_0(s) + o(n)$ bits and linear-time decoding, is indeed the reason why it is used in practice in various folklore applications, as mentioned in the Introduction. In Sect. 8.2 we experimentally evaluate these ideas and show they are practical. Next, we give more far-fetched applications of the rank/select capabilities of our structure, which go much beyond the mere compression.

4 Applications to Text Indexing

Our main result can be readily carried over various types of indexes for text collections. These include self-indexes for general texts, and positional and non-positional inverted indexes for natural language text collections.

4.1 Self-Indexes

A self-index represents a sequence and supports operations related to text searching on it. A well-known self-index [22] achieves k-th order entropy space by partitioning the Burrows-Wheeler transform [12] of the sequence and encoding each partition to its zero-order entropy. Those partitions must support queries access and rank. By using Theorem 2(i) to represent such partitions, we achieve the following result, improving previous ones [4, 22, 28].

Theorem 5 *Let* $s[1..n]$ *be a sequence over effective alphabet* $[1..\sigma]$. *Then we can represent* s *using* $n\mathcal{H}_k(s) + o(n)(\mathcal{H}_k(s) + 1)$ *bits, for any* $k \leq (\delta \log_\sigma n) - 1$ *and constant* $0 < \delta < 1$, *while supporting the following queries*:

(i) *count the number of occurrences of a pattern* $p[1..m]$ *in* s, *in time* $\mathcal{O}(m \lg \lg \sigma)$;
(ii) *locate any such occurrence in time* $\mathcal{O}(\lg n \lg \lg \lg n \lg \lg \sigma)$;
(iii) *extract* $s[l, r]$ *in time* $\mathcal{O}((r - l) \lg \lg \sigma + \lg n \lg \lg \lg n \lg \lg \sigma)$.

Proof To achieve $n\mathcal{H}_k(s)$ space, the Burrows-Wheeler transformed text s^{bwt} is partitioned into $r \leq \sigma^k$ sequences $s^1 \cdots s^r$ [22]. Since $k \leq (\delta \log_\sigma n) - 1$, it follows that $\sigma^{k+1} \leq n^\delta$. The space our Theorem 2(i) achieves using such a partition is $\sum_i |s^i|\mathcal{H}_0(s^i) + (\mathcal{H}_0(s^i)+1) \cdot \mathcal{O}(\frac{|s^i|}{\lg\lg\lg|s^i|})$. Let $\gamma = (1-\delta)/2$ (so $0 < \delta + \gamma < 1$ whenever $0 < \delta < 1$) and classify the sequences s^i according to whether $|s^i| < n^\gamma$ (short sequences) or not (long sequences). The total space occupied by the short sequences can be bounded by $r \cdot \mathcal{O}(n^\gamma \lg \sigma) = \mathcal{O}(n^{\delta+\gamma}) = o(n)$ bits. In turn, the space occupied by the long sequences can be bounded by $\sum_i (1 + \frac{c}{\lg\lg\lg n}) \cdot |s^i|\mathcal{H}_0(s^i) + \frac{d|s^i|}{\lg\lg\lg n}$ bits, for some constants c, d. An argument very similar to the one used by Ferragina et al. [22, Theorem 4.2] shows that these add up to $(1 + \frac{c}{\lg\lg\lg n}) \cdot n\mathcal{H}_k(s) + \frac{dn}{\lg\lg\lg n}$. Thus the space is $n\mathcal{H}_k(s) + o(n)(\mathcal{H}_k(s)+1)$. Other structures required by the alphabet partitioning technique [22] add $o(n)$ more bits if $\sigma^{k+1} \leq n^\delta$.

The claimed time complexities stem from the rank and access times on the partitions. The partitioning scheme [22] adds just constant time overheads. Finally, to achieve the claimed locating and extracting times we sample one out of every $\lg n \lg\lg\lg n$ text positions. This maintains our lower-order space term $o(n)$ within $\mathcal{O}(\frac{n}{\lg\lg\lg n})$. □

In case $[1..\sigma]$ is not the effective alphabet we proceed as described in Sect. 3.3. Our main improvement compared to Theorem 4.2 of Barbay et al. [4] is that we have compressed the redundancy from $o(n \lg \sigma)$ to $o(n)(\mathcal{H}_k(s)+1)$. Our improved locating times, instead, just owe to the denser sampling, which Barbay et al. could also use.

Note that, by using the zero-order representation of Golynski et al. [29, Theorem 4], we could achieve even better space, $n\mathcal{H}_k(s) + o(n)$ bits, and time complexities $\mathcal{O}(1 + \frac{\lg \sigma}{\lg\lg n})$ instead of $\mathcal{O}(\lg\lg\sigma)$.[6] Such complexities are convenient for not so large alphabets.

4.2 Positional Inverted Indexes

These indexes retrieve the positions of any word in a text. They may store the text compressed up to the zero-order entropy of the *word* sequence $s[1..n]$, which allows direct access to any word. In addition they store the list of the positions where each distinct word occurs. These lists can be compressed up to a second zero-order entropy space [48], so the overall space is at least $2n\mathcal{H}_0(s)$. By regarding s as a sequence over an alphabet $[1..\nu]$ (corresponding here to the vocabulary), Theorem 2 represents s within $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s)+1)$ bits, which provides state-of-the-art compression ratios. Variant (ii) supports constant-time access to any text word $s[i]$, and access to the jth entry of the list of any word a ($s.\mathsf{select}_a(j)$) in time $\mathcal{O}(\lg\lg\nu)$. These two time complexities are exchanged in variant (i), or both can be made constant by spending $\epsilon n\mathcal{H}_0(s)$ redundancy for any constant $\epsilon > 0$ (using Corollary 4). The length of the inverted lists can be stored within $\mathcal{O}(\nu \lg n)$ bits (we also need at least this space to store the sequence content of each word identifier).

[6] One can retain $\lg\lg n$ in the denominator by using block sizes depending on n and not on $|s^i|$, as explained in the footnote at the beginning of Sect. 3.2.

Apart from supporting this basic access to the list of each word, this representation easily supports operations that are more complex to implement on explicit inverted lists [5]. For example, we can find the phrases formed by two words w_1 and w_2, that appear n_1 and n_2 times, by finding the occurrences of one and verifying the other in the text, in time $\mathcal{O}(\min(n_1, n_2) \lg\lg \nu)$. Other more sophisticated intersection algorithms [5] can be implemented by supporting operations such as "find the position in the list of w_2 that follows the jth occurrence of word w_1" ($s.\mathsf{rank}_{w_2}(s.\mathsf{select}_{w_1}(j)) + 1$, in time $\mathcal{O}(\lg\lg \nu)$) or "give the list of word w restricted to the range $[x..y]$ in the collection" ($s.\mathsf{select}_w(s.\mathsf{rank}_w(x-1)+j)$, for $j \geq 1$, until exceeding y, in time $\mathcal{O}(\lg\lg \nu)$ plus $\mathcal{O}(1)$ per retrieved occurrence). In Sect. 8.4 we evaluate this representation in practice.

4.3 Binary Relations and Non-positional Inverted Indexes

Let $R \subseteq L \times O$, where $L = [1..\lambda]$ are called *labels* and $O = [1..\kappa]$ are called *objects*, be a binary relation consisting of n pairs. Barbay et al. [3] represent the relation as follows. Let $l_{i_1} < l_{i_2} < \cdots < l_{i_k}$ be the labels related to an object $o \in O$. Then we define sequence $s_o = l_{i_1} l_{i_2} \cdots l_{i_k}$. The representation for R is the concatenated sequence $s = s_1 \cdot s_2 \cdots s_\kappa$, of length n, and the bitmap $b = 10^{|s_1|}10^{|s_2|} \cdots 10^{|s_\kappa|}1$, of length $n + \kappa + 1$.

This representation allows one to efficiently support various queries [3]:

`table_access`: is l related to o?, $s.\mathsf{rank}_l(b.\mathsf{rank}_0(b.\mathsf{select}_1(o+1))) > s.\mathsf{rank}_l(b.\mathsf{rank}_0(b.\mathsf{select}_1(o)))$;

`object_select`: the ith label related to an object o, $s.\mathsf{access}(b.\mathsf{rank}_0(b.\mathsf{select}_1(o) + i))$;

`object_nb`: the number of labels an object o is related to, $b.\mathsf{select}_1(o+1) - b.\mathsf{select}_1(o) - 1$;

`object_rank`: the number of labels $< l$ an object o is related to, carried out with a predecessor search in $s[b.\mathsf{rank}_0(b.\mathsf{select}_1(o))..b.\mathsf{rank}_0(b.\mathsf{select}_1(o+1))]$, an area of length $\mathcal{O}(\lambda)$. The predecessor data structure requires $o(n)$ bits as it is built over values sampled every $\lg^2 \lambda$ positions, and the query is completed with a binary search;

`label_select`: the ith object related to a label l, $b.\mathsf{rank}_1(b.\mathsf{select}_0(s.\mathsf{select}_l(i)))$;

`label_nb`: the number of objects a label l is related to, $s.\mathsf{rank}_l(n)$. It can also be solved like `object_nb`, using a bitmap similar to b that traverses the table label-wise;

`label_rank`: the number of objects $<o$ a label l is related to, $s.\mathsf{rank}_l(b.\mathsf{rank}_0(b.\mathsf{select}_1(o)))$.

Bitmap b can be represented within $\mathcal{O}(\kappa \lg \frac{n}{\kappa}) = o(n) + \mathcal{O}(\kappa)$ bits and support all the operations in constant time [55], and its label-wise variant needs $o(n) + \mathcal{O}(\lambda)$ bits. The rest of the space and time complexities depend on how we represent s.

Barbay et al. [3] used Golynski et al.'s representation for s [28], so they achieved $n \lg \lambda + o(n \lg \lambda)$ bits of space, and the times at rows 2 or 3 in Table 1 for the operations on s (later, Barbay et al. [4] achieved $n\mathcal{H}_k(s) + o(n \lg \lambda)$ bits and slightly worse times). By instead representing s using Theorem 2, we achieve compressed redundancy and slightly improve the times.

To summarize, we achieve $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1) + \mathcal{O}(\kappa + \lambda)$ bits, and solve `label_nb` and `object_nb` in constant time, and `table_access` and `label_rank` in time $\mathcal{O}(\lg\lg\lambda)$. For `label_select`, `object_select` and `object_rank` we achieve times $\mathcal{O}(1)$, $\mathcal{O}(\lg\lg\lambda)$ and $\mathcal{O}((\lg\lg\lambda)^2)$, respectively, or $\mathcal{O}(\lg\lg\lambda)$, $\mathcal{O}(1)$ and $\mathcal{O}(\lg\lg\lambda)$, respectively. Corollary 4 yields a slightly larger representation with improved times, and a multiary wavelet tree [29, Theorem 4] achieves less space and different times; we leave the details to the reader.

A non-positional inverted index is a binary relation that associates each vocabulary word with the documents where it appears. A typical representation of the lists encodes the differences between consecutive values, achieving overall space $\mathcal{O}(\sum_v n_v \lg \frac{n}{n_v})$, where word v appears in n_v documents [61]. In our representation as a binary relation, it turns out that $\mathcal{H}_0(s) = \sum_v n_v \lg \frac{n}{n_v}$, and thus the space achieved is comparable to the classical schemes. Within this space, however, the representation offers various interesting operations apart from accessing the ith element of a list (using `label_select`), including support for various list intersection algorithms; see Barbay et al. [3, 4] for more details.

5 Compressing Permutations

Barbay and Navarro [6] measured the compressibility of a permutation π in terms of the entropy of the distribution of the lengths of *runs* of different kinds. Let π be covered by ρ runs (using any of the previous definitions of runs [6, 40, 44]) of lengths $\mathsf{runs}(\pi) = \langle n_1, \dots, n_\rho \rangle$. Then $\mathcal{H}(\mathsf{runs}(\pi)) = \sum \frac{n_i}{n} \lg \frac{n}{n_i} \leq \lg \rho$ is called the *entropy* of the runs (and, because $n_i \geq 1$, it also holds $n\mathcal{H}(\mathsf{runs}(\pi)) \geq (\rho - 1)\lg n$). In their most recent variant [7] they were able to store π in $2n\mathcal{H}(\mathsf{runs}(\pi)) + o(n) + \mathcal{O}(\rho \lg n)$ bits for runs consisting of interleaved sequences of increasing or decreasing values, and $n\mathcal{H}(\mathsf{runs}(\pi)) + o(n) + \mathcal{O}(\rho \lg n)$ bits for contiguous sequences of increasing or decreasing values (or, alternatively, interleaved sequences of consecutive values). In all cases they can compute $\pi()$ and $\pi^{-1}()$ in $\mathcal{O}(\frac{\lg \rho}{\lg\lg n})$ time, which on average drops to $\mathcal{O}(1 + \frac{\mathcal{H}(\mathsf{runs}(\pi))}{\lg\lg n})$ if the queries are uniformly distributed in $[1..n]$.

We now show how to use access/rank/select data structures to support the operations more efficiently while retaining compressed redundancy space. In general terms, we exchange their $\mathcal{O}(\rho \lg n)$ space term by $o(n)\mathcal{H}(\mathsf{runs}(\pi))$, and improve their times to $\mathcal{O}(\lg\lg \rho)$ in the worst case, and to $\mathcal{O}(\lg \mathcal{H}(\mathsf{runs}(\pi)))$ on average (again, this is an improvement only if ρ is not too small).

We first consider interleaved sequences of increasing or decreasing values as first defined by Levcopoulos and Petersson [40] for adaptive sorting, and later on for compression [6], and then give improved results for more restricted classes of runs. In both cases we first consider the application of the permutation $\pi()$ and its inverse, $\pi^{-1}()$, and later show how to extend the support to the iterated application of the permutation, $\pi^k()$, extending and improving previous results [47].

Theorem 6 *Let π be a permutation on n elements that consists of ρ interleaved increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Suppose we have a data structure that stores a sequence $s[1..n]$ over effective alphabet $[1..\rho]$ within $\psi(n, \rho, \mathcal{H}_0(s))$*

bits, supporting queries access, rank, *and* select *in time* $\tau(n,\rho)$. *Then, given its run decomposition, we can store* π *in* $2\psi(n,\rho,\mathcal{H}(\mathsf{runs}(\pi)))+\rho$ *bits, and perform* $\pi()$ *and* $\pi^{-1}()$ *queries in time* $\mathcal{O}(\tau(n,\rho))$.

Proof We first replace all the elements of the rth run by r, for $1\le r\le\rho$. Let s be the resulting sequence and let s' be s permuted according to π, that is, $s'[\pi(i)]=s[i]$. We store s and s' using the given sequence representation, and also store ρ bits indicating whether each run is increasing or decreasing. Note that $\mathcal{H}_0(s)=\mathcal{H}_0(s')=\mathcal{H}(\mathsf{runs}(\pi))$, which gives the claimed space.

Notice that an increasing run preserves the relative order of the elements of a subsequence. Therefore, if $\pi(i)$ is part of an increasing run, then $s'.\mathsf{rank}_{s[i]}(\pi(i))=s.\mathsf{rank}_{s[i]}(i)$, so

$$\pi(i)=s'.\mathsf{select}_{s[i]}\bigl(s.\mathsf{rank}_{s[i]}(i)\bigr).$$

If, instead, $\pi(i)$ is part of a decreasing run, then $s'.\mathsf{rank}_{s[i]}(\pi(i))=s.\mathsf{rank}_{s[i]}(n)+1-s.\mathsf{rank}_{s[i]}(i)$, so

$$\pi(i)=s'.\mathsf{select}_{s[i]}\bigl(s.\mathsf{rank}_{s[i]}(n)+1-s.\mathsf{rank}_{s[i]}(i)\bigr).$$

A $\pi^{-1}()$ query is symmetric (exchange s and s' in the formulas). Therefore we compute $\pi()$ and π^{-1} with $\mathcal{O}(1)$ calls to access, rank, and select on s or s'. □

Example 3 Let $\pi=1,\mathit{8},\mathbf{9},3,\mathit{6},\mathbf{10},5,\mathit{4},\mathbf{11},7,\mathit{2},\mathbf{12}$ be formed by three runs (indicated by the different fonts). Then $s=(1,2,3,1,2,3,1,2,3)$ and $s'=(1,2,1,2,1,2,1,2,3,3,3,3)$.

By combining Theorem 6 with the representations in Theorem 2, we obtain a result that improves upon previous work [6, 7] in time complexity. Note that if the queried positions i are uniformly distributed in $[1..n]$, then all the access, rank, and select queries follow the same character distribution of the runs, and Corollary 3 applies. Note also that the ρ bits are contained in $o(n)\mathcal{H}(\mathsf{runs}(\pi))$ because $n\mathcal{H}(\mathsf{runs}(\pi))\ge(\rho-1)\lg n$.

Corollary 7 *Let* π *be a permutation on* n *elements that consists of* ρ *interleaved increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Then, given its run decomposition, we can store* π *in* $2n\mathcal{H}(\mathsf{runs}(\pi))+o(n)(\mathcal{H}(\mathsf{runs}(\pi))+1)$ *bits and perform* $\pi()$ *and* $\pi^{-1}()$ *queries in* $\mathcal{O}(\lg\lg\rho)$ *time. On uniformly distributed queries the average times are* $\mathcal{O}(\lg\mathcal{H}(\mathsf{runs}(\pi)))$.

The case where the runs are contiguous is handled within around half the space, as a simplification of Theorem 6.

Corollary 8 *Let* π *be a permutation on* n *elements that consists of* ρ *contiguous increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Suppose we have a data structure that stores a sequence* $s[1..n]$ *over effective alphabet* $[1..\rho]$ *within* $\psi(n,\rho,\mathcal{H}_0(s))$ *bits, supporting queries* access *and* rank *in time* $\tau_{\mathtt{ar}}(n,\rho)$, *and* select *in time* $\tau_{\mathtt{s}}(n,\rho)$. *Then, given its run decomposition, we can store* π *in* $\psi(n,\rho,\mathcal{H}(\mathsf{runs}(\pi)))+\rho\lg\frac{n}{\rho}+\mathcal{O}(\rho)+o(n)$ *bits of space, and perform* $\pi()$ *queries in time* $\mathcal{O}(\tau_{\mathtt{s}}(n,\rho))$ *and* $\pi^{-1}()$ *queries in time* $\mathcal{O}(\tau_{\mathtt{ar}}(n,\rho))$.

Proof We proceed as in Theorem 6, yet now sequence s is of the form $s = 1^{n_1}2^{n_2}\cdots\rho^{n_\rho}$, and therefore it can be represented as a bitmap $b = 10^{n_1-1}10^{n_2-1}\cdots 10^{n_\rho-1}1$. The required operations are implemented as follows: $s.\mathsf{access}(i) = b.\mathsf{rank}_1(i)$, $s.\mathsf{rank}_{s[i]}(i) = i - b.\mathsf{select}_1(s[i])+1$, $s.\mathsf{rank}_{s[i]}(n) = b.\mathsf{select}_1(s[i]+1) - b.\mathsf{select}_1(s[i])$, and $s.\mathsf{select}_a(i) = b.\mathsf{select}_1(a)+i-1$. Those operations are solved in constant time using a representation for b that takes $(\rho+1)\lg(e(n+1)/(\rho+1)) + o(n)$ bits [55]. Added to the ρ bits that mark increasing or decreasing sequences, this gives the claimed space. The claimed time complexities correspond to the operations on s', as those in s take constant time. □

Once again, by combining the corollary with representation (i) in Theorem 2, we obtain results that improve upon previous work [6, 7]. The $\rho\lg\frac{n}{\rho}$ bits are in $o(n)(\mathcal{H}(\mathsf{runs}(\pi))+1)$ because they are $o(n)$ as long as $\rho = o(n)$, and otherwise they are $\mathcal{O}(\rho) = o(\rho\lg n)$, and $(\rho-1)\lg n \le n\mathcal{H}(\mathsf{runs}(\pi))$.

Corollary 9 *Let* π *be a permutation on* n *elements that consists of* ρ *contiguous increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Then, given its run decomposition, we can store* π *in* $n\mathcal{H}(\mathsf{runs}(\pi))+o(n)(\mathcal{H}(\mathsf{runs}(\pi))+1)$ *bits and perform* $\pi()$ *queries in time* $\mathcal{O}(1)$ *and* $\pi^{-1}()$ *queries in time* $\mathcal{O}(\lg\lg\rho)$ (*and* $\mathcal{O}(\lg\mathcal{H}(\mathsf{runs}(\pi)))$ *on average for uniformly distributed queries*).

If π is formed by interleaved but strictly incrementing $(+1)$ or decrementing (-1) runs, then π^{-1} is formed by contiguous runs, in the same number and length [6]. This gives an immediate consequence of Corollary 8.

Corollary 10 *Let* π *be a permutation on* n *elements that consists of* ρ *interleaved strict increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Suppose we have a data structure that stores a sequence* $s[1..n]$ *over effective alphabet* $[1..\rho]$ *within* $\psi(n,\rho,\mathcal{H}_0(s))$ *bits, supporting queries* access *and* rank *in time* $\tau_{\mathtt{ar}}(n,\rho)$, *and* select *in time* $\tau_{\mathtt{s}}(n,\rho)$. *Then, given its run decomposition, we can store* π *in* $\psi(n,\rho,\mathcal{H}(\mathsf{runs}(\pi)))+\rho\lg\frac{n}{\rho}+\mathcal{O}(\rho)+o(n)$ *bits of space, and perform* $\pi()$ *queries in time* $\mathcal{O}(\tau_{\mathtt{ar}}(n,\rho))$ *and* $\pi^{-1}()$ *queries in time* $\mathcal{O}(\tau_{\mathtt{s}}(n,\rho))$.

For example we can achieve the same space of Corollary 9, yet with the times for π and π^{-1} reversed. Finally, if we consider runs for π that are both contiguous and incrementing or decrementing, then so are the runs of π^{-1}. Corollary 8 can be further simplified as both s and s' can be represented with bitmaps.

Corollary 11 *Let* π *be a permutation on* n *elements that consists of* ρ *contiguous and strict increasing or decreasing runs, of lengths* $\mathsf{runs}(\pi)$. *Then, given its run decomposition, we can store* π *in* $2\rho\lg\frac{n}{\rho}+\mathcal{O}(\rho)+o(n)$ *bits, and perform* $\pi()$ *and* $\pi^{-1}()$ *in* $\mathcal{O}(1)$ *time.*

We now show how to achieve exponentiation, $\pi^k(i)$ or $\pi^{-k}(i)$, within compressed space. Munro et al. [47] reduced the problem of supporting exponentiation on a permutation π to the support of the direct and inverse application of another permutation, related but with quite distinct runs than π. Combining it with any of our results does yield compression, but one where the space depends on the lengths of both the runs and cycles of π. The following construction, extending the technique by Munro et al. [47], retains the compressibility in terms of the runs of π, which is more natural. It builds an index that uses small additional space to support the exponentiation, thus allowing the compression of the main data structure with any of our results.

Theorem 12 *Suppose we have a representation of a permutation π on n elements that supports queries $\pi()$ in time τ^+ and queries $\pi^{-1}()$ in time τ^-. Then for any $t \le n$, we can build a data structure that takes $\mathcal{O}((n/t)\lg n)$ bits and, used in conjunction with operation $\pi()$ or $\pi^{-1}()$, supports $\pi^k()$ and $\pi^{-k}()$ queries in $\mathcal{O}(t\min(\tau^+, \tau^-))$ time.*

Proof The key to computing $i' = \pi^k(i)$ is to discover that i is in a cycle of length ℓ and to assign it a position $0 \le j < \ell$ within its cycle (note j is arbitrary, yet we must operate consistently once it is assigned). Then $\pi^k(i)$ lies in the same cycle, at position $j' = (j + k \bmod \ell)$, hence $\pi^k(i) = \pi^{j'-j}(i)$ or $\pi^{j'+\ell-j}(i)$. Thus all we need is to find out j and ℓ, compute j', and finally find the position i' in π that corresponds to the j'th element of the cycle. We decompose π into its cycles and, for every cycle of length at least t, store the cycle's length ℓ and an array containing the position i in π of every tth element in the cycle. Those positions i are called 'marked'. We also store a binary sequence $b[1..n]$, so that $b[i] = 1$ iff i is marked. For each marked element i we record to which cycle i belongs and the position j of i in its cycle. To compute $\pi^k(i)$, we repeatedly apply $\pi()$ at most t times until we either loop or find a marked element. In the first case, we have found ℓ, so we can assume $j = 0$, compute $j' < \ell \le t$, and apply $\pi()$ at most t more times to find $i' = \pi^{j'}(i) = \pi^k(i)$ in the loop. If we reach a marked element, instead, we have stored the cycle identifier to which i belongs, as well as j and ℓ. Then we compute j' and know that the previous marked position is $j^* = t \cdot \lfloor j'/t \rfloor$. The corresponding position i^* is found at cell j^*/t of the array of positions of marked elements, and we finally move from i^* to i' by applying $j' - j^* \le t$ times operation $\pi()$, $i = \pi^{j'-j^*}(i^*) = \pi^k(i)$. A π^{-k} query is similar (note that it does not need to use $\pi^{-1}()$ as we can always move forward). Moreover, we can also proceed using $\pi^{-1}()$ instead of $\pi()$, whichever is faster, to compute both $\pi^k()$ and $\pi^{-k}()$. The space is $\mathcal{O}((n/t)\lg n)$ both for the samples and for a compressed representation of bitmap b. Note that we only compute rank at the positions i such that $b[i] = 1$. Thus we can use the ID structure [55], which uses $\mathcal{O}((n/t)\lg t)$ bits. □

5.1 Application to Self-Indexes

These results on permutations apply to a second family of self-indexes, which is based on the representation of the so-called Ψ function [32, 35, 57]. Given the suffix array $A[1..n]$ of sequence $s[1..n]$ over alphabet $[1..\sigma]$, Ψ is defined as $\Psi(i) =$

$A^{-1}[(A[i] \bmod n)+1]$. Counting, locating, and extracting is carried out through permutation Ψ, which replaces s and A. It is known [35] that Ψ contains σ contiguous increasing runs so that $\mathcal{H}(\mathsf{runs}(\Psi)) = \mathcal{H}_0(s)$, which allows for its compression. Grossi et al. [32] represented Ψ within $n\mathcal{H}_k(s) + \mathcal{O}(n)$ bits, while supporting operation $\Psi()$ in constant time, or within $n\mathcal{H}_k(s) + o(n \lg \sigma)$ while supporting $\Psi()$ in time $\mathcal{O}(\lg \sigma)$. By using Corollary 9, we can achieve the unprecedented space $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1)$ and support $\Psi()$ in constant time. In addition we can support the inverse $\Psi^{-1}()$ in time $\mathcal{O}(\lg \lg \sigma)$. Having both $\Psi()$ and $\Psi^{-1}()$ allows for bidirectional indexes [56], which can for example display a snippet around any occurrence found without the need for any extra space for sampling. Our construction of Theorem 12 can be applied on top of any of those representations so as to support operation $\Psi^k()$, which is useful for example to implement compressed suffix trees, yet the particularities of Ψ allow for sublogarithmic-time solutions [32]. Note also that using Huffman-shaped wavelet trees to represent the permutation [7] yields even less space, $n\mathcal{H}_0(s) + o(n) + \mathcal{O}(\sigma \lg n)$ bits, and the time complexities are relevant for not so large alphabets.

6 Compressing Functions

Hreinsson, Krøyer and Pagh [38] recently showed how, given a domain $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{N}$ of numbers that fit in a machine word, they can represent any $f : X \to [1..\sigma]$ in compressed form and provide constant-time evaluation. Let us identify function f with the sequence of values $f[1..n] = f(x_1)f(x_2)\cdots f(x_n)$. Then their representation uses at most $(1+\epsilon)n\mathcal{H}_0(f) + \mathcal{O}(n) + o(\sigma)$ bits, for any constant $\epsilon > 0$. We note that this bound holds even when σ is much larger than n. In the special case where $X = [1..n]$ and $\sigma = o(n)$, we can achieve constant-time evaluation and a better space bound using our sequence representations. Moreover, we can support extra functionality such as computing the pre-image of an element. A first simple result is obtained by representing f as a sequence.

Lemma 13 *Let $f : [1..n] \to [1..\sigma]$ be a function. We can represent f using $n\mathcal{H}_0(f) + o(n)(\mathcal{H}_0(f)+1) + \mathcal{O}(\sigma)$ bits and compute $f(i)$ for any $i \in [1..n]$ in $\mathcal{O}(1)$ time, and any element of $f^{-1}(a)$ for any $a \in [1..\sigma]$ in time $\mathcal{O}(\lg \lg \sigma)$, or vice versa. Using more space, $(1+\epsilon)\mathcal{H}_0(f) + o(n)$ bits for any constant $\epsilon > 0$, we support both queries in constant time. The size $|f^{-1}(a)|$ is always computed in $\mathcal{O}(1)$ time.*

Proof We represent sequence $f[1..n]$ using Theorem 2 or Corollary 4, so $f(i) = f.\mathsf{access}(i)$ and the jth element of $f^{-1}(a)$ is $f.\mathsf{select}_a(j)$. To compute $|f^{-1}(a)|$ in constant time we store a binary sequence $b = 10^{|f^{-1}(1)|}10^{|f^{-1}(2)|}1 \cdots 10^{|f^{-1}(\sigma)|}1$, so that $|f^{-1}(a)| = b.\mathsf{select}_1(a+1) - b.\mathsf{select}_1(a) - 1$. The space is the one needed to represent s plus $\mathcal{O}(\sigma \lg \frac{n}{\sigma})$ bits to represent b using an ID [55]. This is $o(n)$ if $\sigma = o(n)$, and otherwise it is $\mathcal{O}(\sigma)$. This extra space is also necessary because $[1..\sigma]$ may not be the effective alphabet of sequence $f[1..n]$ (if f is not surjective). □

Another source of compressibility frequently arising in real-life functions is nondecreasing or nonincreasing runs. Let us start by allowing interleaved runs. Note that in this case $\mathcal{H}(\mathsf{runs}(f)) \le \mathcal{H}_0(f)$, where equality is achieved if we form runs of equal values only.

Theorem 14 *Let $f : [1..n] \to [1..\sigma]$ be a function such that sequence $f[1..n]$ consists of ρ interleaved non-increasing or non-decreasing runs. Then, given its run decomposition, we can represent f in $2n\mathcal{H}(\mathsf{runs}(f)) + o(n)(\mathcal{H}(\mathsf{runs}(f)) + 1) + \mathcal{O}(\sigma)$ bits and compute $f(i)$ for any $i \in [1..n]$, and any element in $f^{-1}(a)$ for any $a \in [1..\sigma]$, in time $\mathcal{O}(\lg\lg\rho)$. The size $|f^{-1}(a)|$ is computed in $\mathcal{O}(1)$ time.*

Proof We store function f as a combination of the permutation π that stably sorts the values $f(i)$, plus the binary sequence b of Lemma 13. Therefore, it holds

$$f(i) = b.\mathsf{rank}_1\big(b.\mathsf{select}_0\big(\pi^{-1}(i)\big)\big).$$

Similarly, the jth element of $f^{-1}(a)$ is

$$\pi\big(b.\mathsf{rank}_0\big(b.\mathsf{select}_1(a)\big) + j\big).$$

Since π^{-1} has the same runs as f (the runs in f can have equal values but those of π^{-1} cannot), we can represent π^{-1} using Corollary 7 to obtain the claimed time and space complexities. □

Example 4 Let $f[1..9] = \pi^{-1}(1, 3, 2, 5, 4, 9, 8, 9, 8)$. The odd positions form an increasing run $(1, 2, 4, 8, 8)$ and the even positions form $(3, 5, 9, 9)$. The permutation π sorting the values is $(1, 3, 2, 5, 4, 7, 9, 6, 8)$, and its inverse is $\pi^{-1} = (1, 3, 2, 5, 4, 8, 6, 9, 7)$. The bitmap b is 101010101011100100.

If we consider only contiguous runs in f, we obtain the following result by representing π^{-1} with Corollary 9. Note the entropy of contiguous runs is no longer upper bounded by $\mathcal{H}_0(f)$.

Corollary 15 *Let $f : [1..n] \to [1..\sigma]$ be a function, where sequence f consists of ρ contiguous non-increasing or non-decreasing runs. Then, given its run decomposition, we can represent f in $n\mathcal{H}(\mathsf{runs}(f)) + o(n)(\mathcal{H}(\mathsf{runs}(f)) + 1) + \mathcal{O}(\sigma)$ bits, and compute any $f(i)$ in $\mathcal{O}(1)$ time, as well as retrieve any element in $f^{-1}(a)$ in time $\mathcal{O}(\lg\lg\rho)$. The size $|f^{-1}(a)|$ can be computed in $\mathcal{O}(1)$ time.*

In all the above results we can use Huffman-shaped wavelet trees [7] to obtain an alternative space/time tradeoff. We leave the details to the reader.

6.1 Application to Binary Relations, Revisited

Recall Sect. 4.3, where we represent a binary relation in terms of a sequence s and a bitmap b. By instead representing s as a function, we can capture another source of compressibility, and achieve slightly different time complexities. Note that $\mathcal{H}_0(s)$ corresponds to the distribution of the number o_i of objects associated with a label i, let us call it $\mathcal{H}_{\mathrm{lab}} = \mathcal{H}_0(s) = \sum \frac{o_i}{n} \lg \frac{n}{o_i}$. On the other hand, if we regard the contiguous increasing runs of s, the entropy corresponds to the distribution of the number l_i of labels associated with an object i, let us call it $\mathcal{H}_{\text{obj}} = \mathcal{H}(\mathsf{runs}(s)) = \sum \frac{l_i}{n} \lg \frac{n}{l_i}$. While Sect. 4.3 compresses B in terms of $\mathcal{H}_{\text{lab}} = \mathcal{H}_0(s)$, we can use Corollary 15 to achieve $n\mathcal{H}_{\text{obj}} + o(n)(\mathcal{H}_{\text{obj}} + 1) + \mathcal{O}(\kappa + \lambda)$ bits of space. Since $f.\mathsf{access}(i) = f(i)$ and $f.\mathsf{select}_a(j)$ is the jth element of $f^{-1}(a)$, this representation solves `label _nb`, `object_nb` and `object _select` in constant time, and `label_select` and `object_rank` in time $\mathcal{O}(\lg\lg\lambda)$. Operations `label_rank` and `table_access` require $f.\mathsf{rank}$, which is not directly supported. The former can be solved in time $\mathcal{O}(\lg\lg\lambda \lg\lg\kappa)$ as a predecessor search in π (storing absolute samples every $\lg^2\kappa$ positions), and the latter in time $\mathcal{O}(\lg\lg\lambda)$ as the difference between two `object_rank` queries. We can also achieve $n\mathcal{H}_{\text{obj}} + o(n) + \mathcal{O}(\kappa + \lambda)$ bits using Huffman-shaped wavelet trees; we leave the details to the reader.

7 Compressing Dynamic Collections of Disjoint Sets

Finally, we now give what is, to the best of our knowledge, the first result about storing a compressed collection of disjoint sets while supporting operations union and find [60]. The key point in the next theorem is that, as the sets in the collection C are merged, our space bound shrinks with the zero-order entropy of the distribution of the function s that assigns elements to sets in C. We define $\mathcal{H}(C) = \sum \frac{n_i}{n} \lg \frac{n}{n_i} \le \lg |C|$, where n_i are the sizes of the sets, which add up to n.

Theorem 16 *Let C be a collection of disjoint sets whose union is $[1..n]$. For any $\epsilon > 0$, we can store C in $(1+\epsilon)n\mathcal{H}(C) + \mathcal{O}(|C|\lg n) + o(n)$ bits and perform any sequence of r* union *and* find *operations in $\mathcal{O}(r\alpha(n) + (1/\epsilon)n\lg\lg n)$ total time, where $\alpha(n)$ is the inverse Ackermann's function.*

Proof We first use Theorem 2 to store the sequence $s[1..n]$ in which $s[i]$ is the representative of the set containing i. We then store the representatives in a standard disjoint-set data structure D [60]. Since $\mathcal{H}_0(s) = \mathcal{H}(C)$, our data structures take $n\mathcal{H}(C) + o(n)(\mathcal{H}(C) + 1) + \mathcal{O}(|C|\lg n)$ bits. We can perform a query $\mathsf{find}(i)$ on C by performing $D.\mathsf{find}(s[i])$, and perform a $\mathsf{union}(i, j)$ operation on C by performing $D.\mathsf{union}(D.\mathsf{find}(s[i]), D.\mathsf{find}(s[j]))$. As we only need access functionality on s, we use a simple variant of Theorem 2. We support only rank and select on the multiary wavelet tree that represents sequence t, and store the s_ℓ subsequences as plain arrays. The mapping m is of length $|C|$, so it can easily be represented in plain form to support constant-time operations, within $\mathcal{O}(|C|\lg n)$ bits. This yields constant time access, and therefore the cost of the r union and find operations is $\mathcal{O}(r\alpha(n))$ [60]. For our data structure to shrink as we merge sets, we keep track of $\mathcal{H}(C)$ and, whenever it shrinks by a factor of $1+\epsilon$, we rebuild our entire data structure on the updated values $s[i] \leftarrow \mathsf{find}(s[i])$. First, note that all those find operations take $\mathcal{O}(n)$ time because of path-compression [60]: Only the first time one accesses a node $v \in D$ it may occur that the representative is not directly v's parent. Thus the overall time can be split into $\mathcal{O}(n)$ time for the n instructions $\mathsf{find}(s[i])$ plus $\mathcal{O}(n)$ for the n times a node $v \in D$ is visited for the first time. Reconstructing the structure of

Theorem 2 also takes $\mathcal{O}(n)$ time. The plain structures for m and s_ℓ are easily built in linear time, and so is the multiary wavelet tree supporting rank and access [22], as it requires just tables of sampled counters. Since $\mathcal{H}(C)$ is always less than $\lg n$, we rebuild only $\mathcal{O}(\log_{1+\epsilon} \lg n) = \mathcal{O}((1/\epsilon) \lg \lg n)$ times. Thus the overall cost of rebuilding is $\mathcal{O}((1/\epsilon) n \lg \lg n)$. This completes our time complexity. Finally, the space term $o(n)\mathcal{H}(C)$ is absorbed by $\epsilon\mathcal{H}(C)$ by slightly adjusting ϵ, and this gives our final space formula. $\square$

8 Experimental Results

In this section we explore the performance of our structure in practice. We first introduce, in Sect. 8.1, a simpler and more practical alphabet partitioning scheme we call "dense", which experimentally performs better than the one we describe in Sect. 3, although in theory has an $\mathcal{O}(n)$-bit redundancy term in the space. Next, in Sect. 8.2 we study the performance of both alphabet partitioning methods, as well as the optimal one [59], in terms of compression ratio and decompression performance. Given the results of these experiments, we continue only with our dense partitioning for the rest of the section. In Sect. 8.3 we compare our new sequence representation with the state of the art, considering the tradeoff between space and time of operations rank, select, and access. Then, Sects. 8.4, 8.5, and 8.6 compare the same data structures on different real-life applications of sequence representations. In the first, the operations are used to emulate an inverted index on the compressed sequence using (almost) no extra space. In the second, they are used to emulate self-indexes for text [48]. In the third, they provide access to direct and reverse neighbors on graphs represented with adjacency lists. The machine used for the experiments has an Intel® Xeon® E5620 at 2.40 GHz, 94 GB of RAM. We did not use multithreading in our implementations; times are measured using only one thread, and in RAM. The operating system is Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. The code was compiled using GNU/GCC version 4.4.3 with optimization flags `-O9`. Our code is available in LIBCDS version 1.0.10, downloadable from http://libcds.recoded.cl/.

8.1 Dense Alphabet Partitioning

Said [59] proved that an optimal assignment to sub-alphabets must group consecutive symbols once sorted by frequency. A simple alternative to the partitioning scheme presented in Sect. 3, and that follows this optimality principle, is to make mapping m group elements into consecutive chunks of doubling size, that is, $m[a] = \lfloor \lg r(a) \rfloor$, where $r(a)$ is the rank of a according to its frequency. The rest of the scheme to define $t[1..n]$ and the sequences $s_\ell[1..\sigma_\ell]$ is as in Sect. 3. The classes are in the range $0 \le \ell \le \lfloor \lg \sigma \rfloor$, and each element in s_ℓ is encoded in ℓ bits. As we use all the available bits of each symbol in sequences s_ℓ (except possibly in the last one), we call this scheme *dense*. We show that this scheme is not much worse than the one proposed in Sect. 3 (which will be called *sparse*). First consider the total number of bits we use to encode the sequences s_ℓ, $\sum_\ell \ell |s_\ell| = \sum_a |s|_a \lfloor \lg r(a) \rfloor$. Since $|s|_a \le n/r(a)$ because the symbols are sorted by decreasing frequency, it holds that

$r(a) \leq n/|s|_a$ and $\sum |s|_a \lfloor \lg r(a) \rfloor \leq \sum |s|_a \lg(n/|s|_a) = n\mathcal{H}_0(s)$. Now consider the number of bits we use to encode $t = \lfloor \lg r(s[1]) \rfloor, \ldots, \lfloor \lg r(s[n]) \rfloor$. We could store each element $\lfloor \lg r(s[i]) \rfloor$ of t in $2\lfloor \lg(\lfloor \lg r(s[i]) \rfloor + 1) \rfloor - 1$ bits using γ-codes [61], and such encoding would be lower bounded by $n\mathcal{H}_0(t)$. Thus $n\mathcal{H}_0(t) \leq 2 \sum_i \lg\lg(r(s[i]) + 1) \leq 2\sum_a |s|_a \lg\lg(n/|s|_a + 1) = \mathcal{O}(n(\lg \mathcal{H}_0(s) + 1)) = o(nH_0(s)) + \mathcal{O}(n)$ (recall Sect. 3.2). It follows that the total encoding length is $n\mathcal{H}_0(s) + \mathcal{O}(n \lg \mathcal{H}_0(s)) = n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s)) + \mathcal{O}(n)$ bits. Apart from the pretty tight upper bound, it is not evident whether this scheme is more or less efficient than the sparse encoding. Certainly the dense scheme uses the least possible number of classes (which could allow storing t in plain form using $\lg\lg\sigma$ bits per symbol). On the other hand, the sparse method uses in general more classes, which allows for smaller sub-alphabets using fewer bits per symbol in sequences s_ℓ. As we will see in Sect. 8.2, the dense scheme uses less space than the sparse one for t, but more for the sequences s_ℓ.

Example 5 Consider the same sequence $s =$ `"alabar a la alabarda"` of Example 1. The dense partitioning will assign $m[\texttt{a}] = 0$, $m[\texttt{l}] = m[\texttt{"}] = 1$, $m[\texttt{b}] = m[\texttt{r}] = m[\texttt{d}] = 2$. So the sequence of sub-alphabet identifiers is $t[1..20] =$ `(0, 1, 0, 2, 0, 2, 1, 0, 1, 1, 0, 1, 0, 1, 0, 2, 0, 2, 2, 0)`, and the subsequences are $s_0 =$ `(1, 1, 1, 1, 1, 1, 1, 1, 1)`, $s_1 =$ `(2, 1, 1, 2, 1, 2)`, and $s_2 =$ `(1, 3, 1, 3, 2)`. This dense scheme uses 16 bits for the sequences s_ℓ, and the zero-order compressed t requires $n\mathcal{H}_0(t) = 30.79$ bits. The overall compression is 2.34 bits per symbol. The sparse partitioning of Example 1 used 10 bits in the sequences s_ℓ, and the zero-order compressed t required $n\mathcal{H}_0(t) = 34.40$ bits. The total gives 2.22 bits per symbol. In our real applications, the dense partitioning performs better.

Note that the question of space optimality is elusive in this scenario. Since the encoding in t plus that in the corresponding sequence s_ℓ forms a unique code per symbol, the optimum is reached when we choose one sub-alphabet per symbol, so that the sequences s_ℓ require zero bits and all the space is in $n\mathcal{H}_0(t) = n\mathcal{H}_0(s)$. The alphabet partitioning always gives away some space, in exchange for faster decompression (or, in our case, faster rank/select/access operations). Said's optimal partitioning [59] takes care of this problem by using a parameter k that is the maximum number of sub-alphabets to use. We sort the alphabet by decreasing frequency and call $S(c, k)$ the total number of bits required to encode the symbols $[c..\sigma]$ of the alphabet using a partitioning into at most k sub-alphabets. In general, we can make a sub-alphabet with the symbols $[c..c']$ and solve optimally the rest, but if $k = 1$ we are forced to choose $c' = \sigma$. When we can choose, the optimization formula is as follows:

$$S(c, k) = \min_{c \leq c' \leq \sigma} \left(f \lg \frac{n}{f} + f \left\lceil \lg\bigl(c' - c + 1\bigr) \right\rceil + S\bigl(c' + 1, k - 1\bigr) \right),$$

where f is the total frequency of symbols c-th to c'-th in s. The first term of the sum accounts for the increase in $t\mathcal{H}_0(s)$, the second for the size in bits of the new sequence s_ℓ, and the third for the smaller subproblem, where it also holds $S(\sigma + 1, k) = 0$ for any k. This dynamic programming algorithm requires $\mathcal{O}(k\sigma)$ space and $\mathcal{O}(\sigma^2)$ time. We call this partitioning method *optimal*.

Example 6 The optimal partitioning using 3 classes just like the dense approach in Example 5 leaves `'a'` in its own class, then groups `'l'`, `''`, `'b'` and `'r'` in a second class, and finally leaves `'d'` alone in a third class. The overall space $n\mathcal{H}_0(t) + \sum |s_\ell| \lceil \lg \sigma_\ell \rceil$ is 2.23 bits per symbol, less than the 2.34 reached by the dense partitioning. If, instead, we let it use four classes, it gives the same solution as the sparse method in Example 1.

Finally, in Sect. 3 we represent the sequences s_ℓ with small alphabets σ_ℓ using wavelet trees (just like t) instead of using the representation of Golynski et al. [28], which is used for large $\sigma_\ell > \lg n$. In theory, this is because Golynski et al.'s representation does not ensure sublinearity on smaller alphabets when used inside our scheme. While this may appear to be a theoretical issue, the implementation of such data structure (e.g., in LIBCDS) is indeed unattractive for small alphabets. For this reason, we also avoid using it on the chunks where σ_ℓ is small (in our case, the first ones). Note that using a wavelet tree for t and then another for the symbols in a sequence s_ℓ is equivalent to replacing the wavelet tree leaf corresponding to ℓ in t by the whole wavelet tree of s_ℓ. The space used by such an arrangement is worse than the one obtained by building, from scratch, a wavelet tree for t where the symbols $t[i] = \ell$ are actually replaced by the corresponding symbol $s[i]$.

In our *dense* representation we use a parameter $\ell_{\min}$ that controls the minimum ℓ value that is represented outside of t. All the symbols that would belong to s_ℓ, for $\ell < \ell_{\min}$, are represented directly in t. Note that, by default, since $\sigma_0 = 1$, we have $\ell_{\min} = 1$.

8.2 Compression Performance

For all the experiments in Sect. 8, except Sect. 8.6, we used real datasets extracted from Wikipedia. We considered two large collections, Simple English and Spanish, dated from 06/06/2011 and 03/02/2010, respectively. Both are regarded as sequences of words, not characters. These collections contain several versions of each article. Simple English, in addition, uses a reduced vocabulary. We collected a sample of 100,000 versions at random from all the documents of Simple English, which makes a long and repetitive sequence over a small alphabet. For the Spanish collection, which features a much richer vocabulary, we took the oldest version of each article, which yields a sequence of similar size, but with a much larger alphabet.

We generated a single sequence containing the word identifiers of all the articles concatenated, obtained after stemming the collections using Porter for English and Snowball for Spanish. Table 2 shows some basic characteristics of the sequences obtained.[7]

We measured the compression ratio achieved by the three partitioning schemes, `dense`, `sparse`, and `optimal`. For `dense` we did not include any individual symbols (other than the most frequent) in sequence t, i.e., we let $\ell_{\min} = 1$. For `optimal` we allow $1 + \lfloor \lg \sigma \rfloor$ sub-alphabets, just like `dense`.

[7]The code for generating these sequences is available at https://github.com/fclaude/txtinvlists.

Table 2 Main characteristics of the datasets used

Collection	Articles	Total words (n)	Distinct words (σ)	Entropy ($\mathcal{H}_0(s)$)
`Simple English`	100,000	766,968,140	664,194	11.60
`Spanish`	1,590,453	511,173,618	3,210,671	11.37

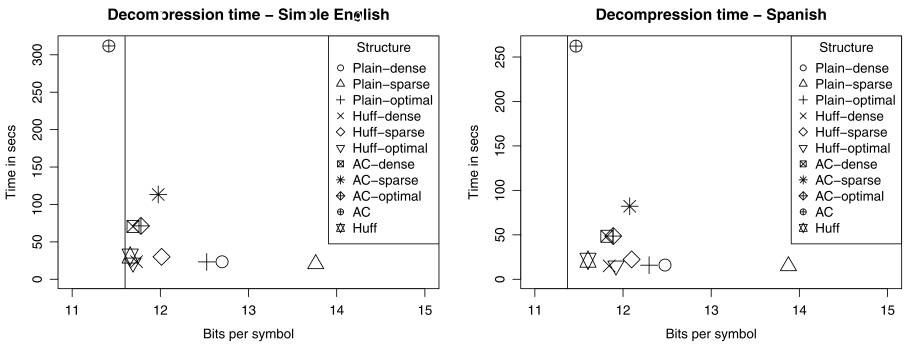

Fig. 1 Space versus decompression time for basic and alphabet-partitioned schemes. *The vertical line* marks the zero-order entropy of the sequences. AC can slightly break the entropy barrier on Simple English because it is adaptive

In all cases, the symbols in each s_ℓ are represented using $\lceil \lg \sigma_\ell \rceil$ bits. The sequence of classes t, instead, is represented in three different forms: `Plain` uses a fixed number of bits per symbol, $\lceil \lg \ell \rceil$ where ℓ is the maximum class; `Huff` uses Huffman coding of the symbols,[8] and `AC` uses Arithmetic coding of the symbols.[9] The former encodings are faster, whereas the latter use less space. In addition we consider compressing the original sequences using Huffman (`Huffman`) and Arithmetic coding (`Arith`).

As explained, the main interest in using alphabet partitioning in a compressor is to speed up decompression without sacrificing too much space. Figure 1 compares all these alternatives in terms of space usage (percentage of the original sequence) and decompression time per symbol. It can be seen that alphabet partitioning combined with `AC` compression of t wastes almost no space due to the partitioning, and speeds up considerably the decompression of the bare `AC` compression. However, bare `Huffman` also uses the same space and decompresses several times faster. Therefore, alphabet partitioning combined with `AC` compression is not really interesting. The other extreme is the combination with a `Plain` encoding of t. In the best combinations, this alphabet partitioning wastes close to 10 % of space, and in exchange decompresses around 30 % faster than bare `Huffman`. The intermediate

[8]We use G. Navarro's Huffman implementation; the code is available in LIBCDS.

[9]We use the code by J. Carpinelli, A. Moffat, R. Neal, W. Salamonsen, L. Stuiver, A. Turpin and I. Witten, available at http://ww2.cs.mu.oz.au/~alistair/arith_coder/arith_coder-3.tar.gz. We modified the decompressor to read the whole stream before timing decompression.

Table 3 Breakdown, in bits per symbol, of the space used in sequence t versus the space used in all the sequences s_ℓ, for the different combinations. *The third column* in each collection is the percentage of symbols that lie in sequences s_ℓ with alphabet sizes $\sigma_\ell = 1$

Combination	Simple English			Spanish		
	t	s_ℓ	%	t	s_ℓ	%
`Plain-dense`	4.96	7.74	6.67	4.76	7.72	11.70
`Plain-sparse`	9.97	3.79	19.01	9.79	4.08	32.91
`Plain-optimal`	4.96	7.66	10.21	4.76	7.61	16.88
`Huff-dense`	3.99	7.74	6.67	4.13	7.72	11.70
`Huff-sparse`	8.22	3.79	19.01	8.01	4.08	32.91
`Huff-optimal`	4.08	7.66	10.21	4.24	7.61	16.88
`AC-dense`	3.95	7.74	6.67	4.10	7.72	11.70
`AC-sparse`	8.18	3.79	19.01	7.99	4.08	32.91
`AC-optimal`	4.03	7.66	10.21	4.20	7.61	16.88

combination, `Huff`, wastes less than 1 % of space, while improving decompression time by almost 25 % over bare `Huffman`.

Another interesting comparison is that of partitioning methods. In all cases, variant `dense` performs better than `sparse`. The difference is larger when combined with `Plain`, where `sparse` is penalized for the larger alphabet size of t, but still there is a small difference when combined with `AC`, which shows that t has also (slightly) lower entropy in variant `dense`. Table 3 gives a breakdown of the bits per symbol in t versus the sequences s_ℓ in all the methods. It can be seen that `sparse` leaves much more information on sequence t than the alternatives, which makes it less appealing since the operation of t is slower than that of the other sequences. However, this can be counterweighted by the fact that `sparse` produces many more sequences with alphabet size 1, which need no time for accessing. It is also confirmed that `dense` leaves slightly less information on t than `optimal`, and that the difference in space between the three alternatives is almost negligible (unless we use `Plain` to encode t, which is not interesting).

Finally, let us consider how the partitioning method affects decompression time, given an encoding method for t. For method `AC`, `sparse` is significantly slower. This is explained by the t component having many more bits, and the decompression time being dominated by the processing of t by the (very slow) arithmetic decoder. For method `Plain`, instead, `sparse` is slightly faster, despite the fact that it uses more space. Since now the reads on t and s_ℓ take about the same time, this difference is attributable to the fact that `sparse` leaves more symbols on sequences s_ℓ with alphabets of size 1, where only one read in t is needed to decode the symbol (see Table 3). For `Huff` all the times are very similar, and very close to the fastest one. Therefore, for the rest of the experiments we use the variant `Huff` with `dense` partitioning, which performs best in space/time.

8.3 Rank, Select and Access

We now consider the efficiency in the support for the operations rank, select, and access. We compare our sequence representation with the state of the art, as implemented in LIBCDS v1.0.10, a library of highly optimized implementations of compact data structures. As said, LIBCDS already includes the implementation of our new structure.

We compare six data structures for representing sequences. Those based on wavelet trees are obtained in LIBCDS by combining sequence representations (`WaveletTreeNoptrs`, `WaveletTree`) with bitmap representations (`BitSequenceRG`, `BitSequenceRRR`) for the data on wavelet tree nodes.

- `WTNPRG`: Wavelet tree without pointers, obtained as `WaveletTreeNoptrs`+`BitSequenceRG` in LIBCDS. This corresponds to the basic balanced wavelet tree structure [32], where all the bitmaps of a level are concatenated [41]. The bitmaps are represented in plain form and their operations are implemented using a one-level directory [30] (where rank is implemented in time proportional to a sampling step and select uses a binary search on rank). The space is $n \lg \sigma + o(n \lg \sigma)$ and the times are $\mathcal{O}(\lg \sigma)$. In practice the absence of pointers yields a larger number of operations to navigate in the wavelet tree, and also select operation on bitmaps is much costlier than rank. A space/time tradeoff is obtained by varying the sampling step of the bitmap rank directories.
- `WTNPRRR`: Wavelet tree without pointers with bitmap compression, obtained in LIBCDS as `WaveletTreeNoptrs`+`BitSequenceRRR`. This is similar to `WTNPRG`, but the bitmaps are represented in compressed form using the FID technique [55] (select is also implemented with binary search on rank). The space is $n\mathcal{H}_0(s) + o(n \lg \sigma)$ and the times are $\mathcal{O}(\lg \sigma)$. In practice the FID representation makes it considerably slower than the version with plain bitmaps, yet select operation is less affected. A space/time tradeoff is obtained by varying the sampling step of the bitmap rank directories.
- `GMR`: The representation proposed by Golysnki et al. [28], named `SequenceGMR` in LIBCDS. The space is $n \lg \sigma + o(n \lg \sigma)$, yet the lower-order term is sublinear on σ, not n. The time is $\mathcal{O}(1)$ for select and $\mathcal{O}(\lg \lg \sigma)$ for rank and access, although on average rank is constant-time. A space/time tradeoff, which in practice affects only the time for access, is obtained by varying the permutation sampling inside the chunks [28].
- `WTRG`: Wavelet tree with pointers and Huffman shape, obtained as `WaveletTree`+`BitSequenceRG` in LIBCDS. The space is $n\mathcal{H}_0(s) + \mathcal{O}(n) + o(n\mathcal{H}_0(s)) + \mathcal{O}(\sigma \lg n)$. The time is $\mathcal{O}(\lg \sigma)$, but in our experiments it will be $\mathcal{O}(\mathcal{H}_0(s))$ for access, since the positions are chosen at random from the sequence and then we navigate less frequently to deeper Huffman leaves.
- `WTRRR`: Wavelet tree with pointers, obtained with `WaveletTree`+`BitSequenceRRR` in LIBCDS. The space is $n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s)) + \mathcal{O}(\sigma \lg n)$. The time is as in the previous structure, except that in practice the FID representation is considerably slower.
- `AP`: Our new alphabet partitioned structure, named `SequenceAlphPart` in LIBCDS. We use dense partitioning and include the 2^{10} most frequent symbols directly in t, $\ell_{\min} = 10$. Sequence t is represented with a `WTRG` (since its alphabet is small and the pointers pose no significant overhead), and the sequences σ_ℓ are represented with structures `GMR`. The space is $n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s))$, although the lower-order term is actually sublinear on σ (and only very slightly on n). The times are as in `GMR`, although there is a small additive overhead due to the wavelet tree

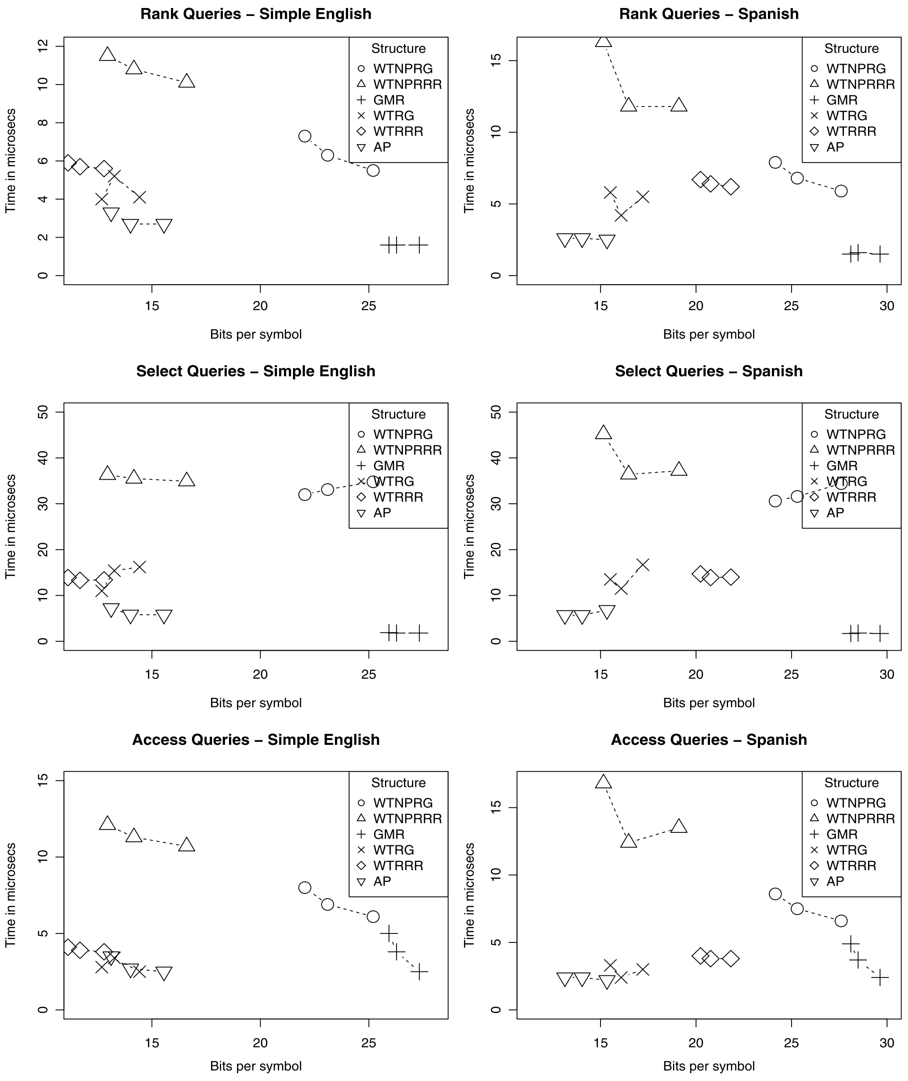

Fig. 2 Time for the three operations. The x axis starts at the entropy of the sequence

on t. A space/time tradeoff is obtained with the permutations sampling, just as in GMR.

Figure 2 shows the results obtained for both text collections, giving the average over 100,000 measures. The rank queries were generated by choosing a symbol from $[1..\sigma]$ and a position from $[1..n]$, both uniformly at random. For select we chose the symbol a in the same way, and the other argument uniformly at random in $[1..|s|_a]$. Finally, for access we generated the position uniformly at random in $[1..n]$. Note that the latter choice favors Huffman-shaped wavelet trees, on which we descend to leaf a

with probability $|s|_a/n$, whereas for rank and select we descend to any leaf with the same probability.

Let us first analyze the case of Simple English, where the alphabet is smaller. Since σ is 1000 times smaller than n, the $\mathcal{O}(\sigma \lg n)$ terms of Huffman-shaped wavelet trees are not significant, and as a result the variant WTRRR reaches the least space, essentially $n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s))$. It is followed by three variants that use similar space: WTRG (which has an additional $\mathcal{O}(n)$-bit overhead), AP (whose $o(n\mathcal{H}_0(s))$ space term is higher than that of wavelet trees), and WTNPRRR (whose sublinear space term is of the form $o(n \lg \sigma)$, that is, uncompressed). The remaining structures, WTNPRG and GMR, are not compressed and use much more space.

In terms of time, structure AP is faster than all the others except GMR (which in exchange uses much more space). The exception is on access queries, where as explained Huffman-shaped wavelet trees, WTRG and WTRRR, are favored and reach the same performance of AP. In general, the rule is that variants using plain bitmaps are faster than those using FID compression, and that variants using pointers and Huffman shape are faster than those without pointers (as the latter need additional operations to navigate the tree). These differences are smaller on select queries, where the binary searches dominate most of the time spent.

The Spanish collection has a much larger alphabet: σ is only 100 times smaller than n. This impacts on the $\mathcal{O}(\sigma \lg n)$ bits used by the pointer-based wavelet trees, and as a result the space of AP, $n\mathcal{H}_0(s) + o(n\mathcal{H}_0(s))$, is unparalleled. Variants WTRRR and WTRG use significantly more space and are followed, far away, by WTNPRRR, which has uncompressed redundancy. The uncompressed variants WTNPRG and RG use significantly more space. The times are basically as on Simple English.

This second collection illustrates more clearly that, for large alphabets, our structure AP sharply dominates the whole space/time tradeoff. It is only slightly slower than GMR in some cases, but in exchange it uses half the space. From the wavelet trees, the most competitive alternative is WTRG, but it always loses to AP. The situation is not too different on smaller alphabets (as in Simple English), except that variant WTRRR uses clearly less space, yet at the expense of doubling the operation times of AP.

8.4 Intersecting Inverted Lists

An interesting application of rank/select operations on large alphabets was proposed by Clarke et al. [14], and recently implemented by Arroyuelo et al. [1] using wavelet trees. The idea is to represent the text collections as a sequence of word tokens (as done for Simple English and Spanish), use a compressed and rank/select/access-capable sequence representation for them, and use those operations to emulate an inverted index on the collection, without spending any extra space on storing explicit inverted lists.

More precisely, given a collection of d documents $T_1, T_2, \ldots, T_d$, we concatenate them in $\mathcal{C} = T_1 T_2 \cdots T_d$, and build an auxiliary bitmap $b[1..|\mathcal{C}|]$ where we mark the beginning of each document with a 1. We can provide access to the text of any document in the collection via access operations on sequence $\mathcal{C}$ (and select on b). In order

```
input  : $\mathcal{C}, w, p$
output: next document after $T_p$ that contains $w$
$pos \leftarrow b.\mathsf{select}_1(p+1)$;
$cnt \leftarrow \mathcal{C}.\mathsf{rank}_w(pos-1)$;
return $b.rank_1(\mathcal{C}.select_w(cnt+1))$
```

Algorithm 1: Function *nextDoc*($\mathcal{C}, w, p$), retrieves the next document after p containing w. The x axis starts at the entropy of the sequence

```
input  : $\mathcal{C}, W = w_1, w_2, \dots, w_k$
output: documents that contain $w_1, \dots, w_k$
sort $W$ by increasing number of occurrences in the collection;
$res \leftarrow \emptyset$;
$p \leftarrow \mathtt{nextDoc}(\mathcal{C}, w_1, 0)$;
while $p$ is valid do
    if $w_2, \dots, w_k$ are contained in $p$ (i.e., $p = \mathtt{nextDoc}(\mathcal{C}, w_j, p-1)$ for $2 \le j \le k$) then
        Add $p$ to $res$;
        $p \leftarrow \mathtt{nextDoc}(\mathcal{C}, w_1, p)$
    end
    else
        Let $w_j$ be the first word not contained in $p$;
        $p \leftarrow \mathtt{nextDoc}(\mathcal{C}, w_1, \mathtt{nextDoc}(\mathcal{C}, w_j, p-1))$
    end
end
return $res$
```

Algorithm 2: Retrieving the documents where all $w_1, \dots, w_k$ appear

to emulate the inverted list of a given term w, we just need to list all the distinct documents where w occurs. This is achieved by iterating on procedure *nextDoc*($\mathcal{C}, w, p$) of Algorithm 1 (called initially with $p = 0$ and then using the last p value returned).

Algorithm 1 also allows one to test whether a given document contains a term or not (p contains w iff $p = nextDoc(\mathcal{C}, w, p-1)$). Using this primitive we implemented Algorithm 2, which intersects several lists (i.e., returns the documents where all the given terms appear) based on the algorithm by Demaine et al. [18]. We tested this algorithm for both Simple English and Spanish collections, searching for phrases extracted at random from the collection. We considered phrases of lengths 2 to 16. We averaged the results over 1,000 queries. As all the results were quite similar, we only show the cases of 2 and 6 words. We tested the same structures as in Sect. 8.3.

Figure 3 shows the results obtained by the different structures. For space, of course, the results are as before: `AP` is the best on Spanish and is outperformed by `WTRRR` on Simple English. With respect to time, we observe that Huffman-shaped wavelet trees are favored compared to the random rank and select queries of Sect. 8.3. The reason is that the queries in this application, at least in the way we have gener-

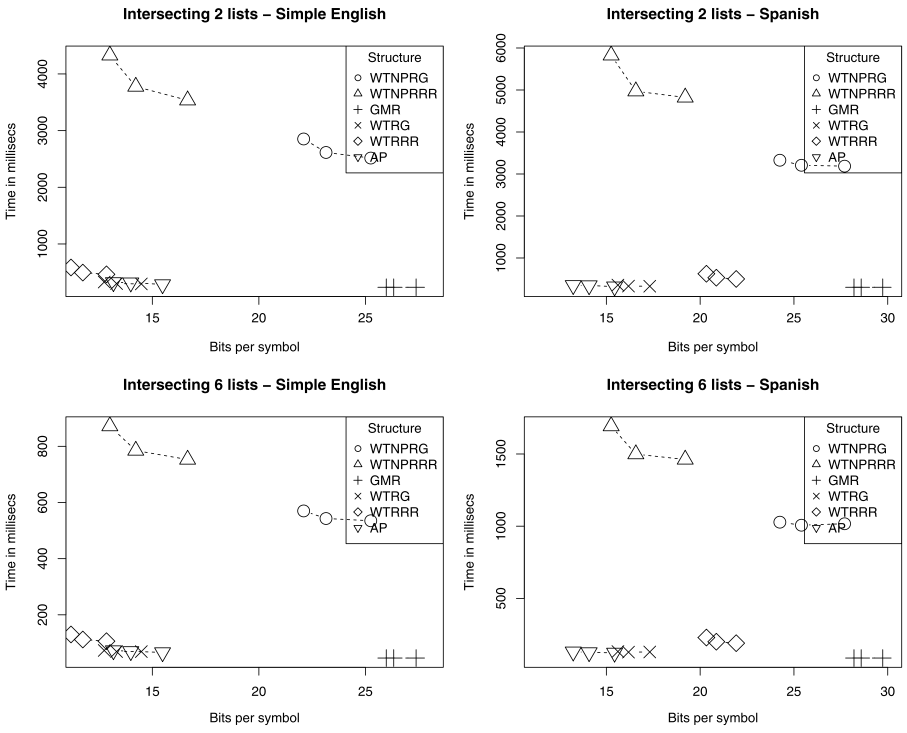

Fig. 3 Results for intersection queries. The x axis starts at the entropy of the sequence

ated them, do not distribute uniformly at random: the symbols for rank and select are chosen according to their probability in the text, which favors Huffman-shaped trees. As a result, structures WTRG perform similarly to AP in time, whereas WTRRR is less than twice as slow.

8.5 Self-Indexes

A second application of the sequence operations on large alphabets was explored by Fariña et al. [19]. The idea is to take a self-index [48] designed for text composed of characters, and apply it to a word-tokenized text, in order to carry out word-level searches on natural language texts. This requires less space and time than the character-based indexes and competes successfully with word-addressing inverted indexes. One of the variants they explore is to build an FM-index [21, 22] on words [15]. The FM-index represents the Burrows-Wheeler transform (BWT) [12] s^{bwt} of s. Using rank and access operations on s^{bwt} the FM-index can, among other operations, *count* the number of occurrences of a pattern $p[1..k]$ (in our case, a phrase of k words) in $s[1..n]$. This requires $\mathcal{O}(k)$ applications of rank and access on s^{bwt}. A self-index is also able to retrieve any passage of the original sequence s.

We implemented the word-based FM-index with the same structures measured so far, plus a new variant called APRRR. This is a version of AP where the bitmaps of the wavelet tree of t are represented using FIDs [55]. The reason is that it was proved [42]

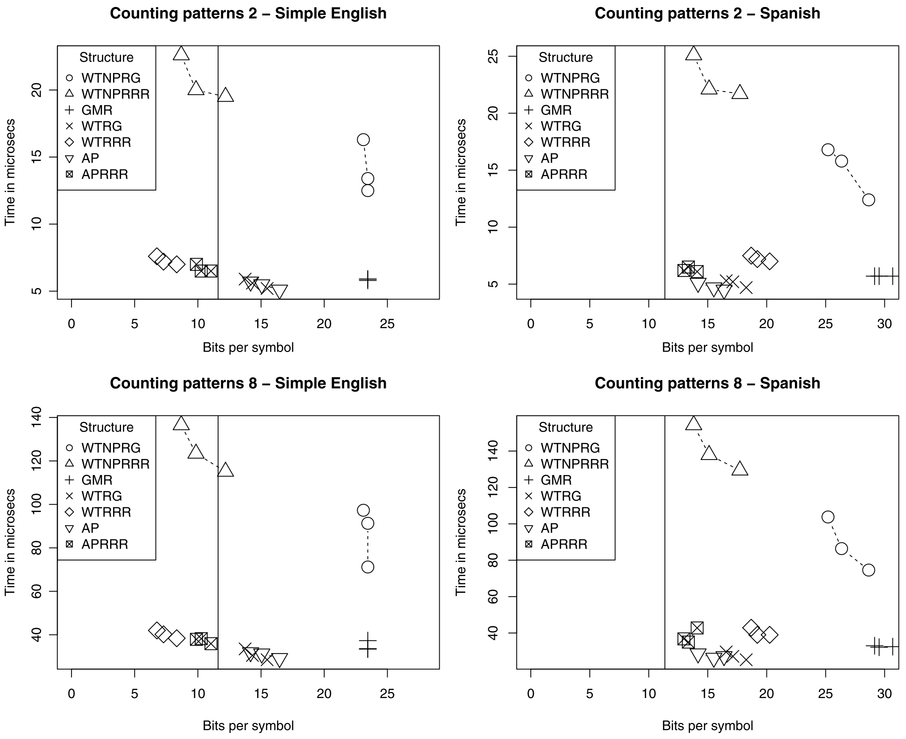

Fig. 4 Time for counting queries on word-based FM-indexes. *The vertical line* marks the zero-order entropy of the sequences; remember that some schemes achieve high-order entropy spaces

that the wavelet tree of s^{bwt}, if the bitmaps are represented using Raman et al.'s FID [55], achieves space $n\mathcal{H}_k(s) + o(n \lg \sigma)$. Since the wavelet tree t of sub-alphabets of s^{bwt} is a coarsened version of that of s^{bwt}, we expect it to take advantage of Raman et al.'s representation.

We extracted phrases at random text positions, of lengths 2 to 16, and counted their number of occurrences using the FM-index. We averaged the results over 100,000 searches. As the results are similar for all lengths, we show the results for lengths 2 and 8. Figure 4 shows the time/space tradeoff obtained.

Confirming the theoretical results [42], the versions using compressed bitmaps require much less space than the other alternatives. In particular, `APRRR` uses much less space than `AP`, especially on Simple English. In this text the least space is reached by `WTRRR`. On Spanish, instead, the $\mathcal{O}(\sigma \lg n)$ bits of Huffman-shaped wavelet trees become relevant and the least space is achieved by `APRRR`, closely followed by `AP` and `WTNPRRR`. The space/time tradeoff is dominated by `APRRR` and `AP`, the two variants of our structure.

8.6 Navigating Graphs

Finally, our last application scenario is the compact representation of graphs. Let $G = (V, E)$ be a directed graph. If we concatenate the adjacency lists of the nodes,

Table 4 Description of the Web crawls considered

Name	Nodes	Edges	Plain adj. list (bits per edge)
EU (EU-2005)	862,664	19,235,140	20.81
In (Indochina-2002)	7,414,866	194,109,311	23.73

the result is a sequence $s[1..|E|]$ over an alphabet of size $|V|$. If we add a bitmap $b[1..|E|]$ that marks with a 1 the beginning of the lists, it is very easy to retrieve the adjacency list of any node $v \in V$, that is, its *neighbors*, with one select operation on b followed by one access operation on s per neighbor retrieved.[10]

It is not hard to reach this space with a classical graph representation. However, classical representations do not allow one to retrieve efficiently the *reverse neighbors* of v, that is, the nodes that point to it. The classical solution is to double the space to represent the transposed graph. Our sequence representation, however, allows us to retrieve the reverse neighbors using select operations on s, much as Algorithm 1 retrieves the documents where a term w appears: our "documents" are the adjacency lists of the nodes, and the document identifier is the node $v \in V$ that points to the desired node. Similarly, it is possible to determine whether a given node v points to a given node v', which is not an easy operation with classical adjacency lists. This idea has not only been used in this simple form [15], but also in more sophisticated scenarios where it was combined with grammar compression of the adjacency lists, or with other transformations, to compress Web graphs and social networks [16, 17, 37].

For this experiment we used two crawls obtained from the well-known *WebGraph* project.[11] The main characteristics of these crawls are shown in Table 4. Note that the alphabets are comparatively much larger than on documents, just around 22–26 times smaller than the sequence length.

Figure 5 shows the results obtained. The nodes are sorted alphabetically by URL. A well-known property of Web graphs [10] is that nodes tend to point to other nodes of the same domain. This property turns into substrings of nearby symbols in the sequence, and this turns into runs of 0 s or 1 s in the bitmaps of the wavelet trees. This makes variants like `WTNPRRR` very competitive in space, whereas `APRRR` does not benefit so much. The reason is that the partitioning into classes reorders the symbols, and the property is lost. Note that variant `WTRRR` does not perform well in space, since the number of nodes is too large for a pointer-based tree to be advantageous. For the same reason, even `WTRG` uses more space than `GMR`.[12] Overall, we note that our variants largely dominate the space/time tradeoff, except that `WTNPRRR` uses less space (but much more time).

[10]Note that this works well as long as each node points to at least one node. We solve this problem by keeping an additional bitmap marking the nodes whose list is not empty.

[11]http://law.dsi.unimi.it.

[12]Note that `WTRRR` is almost 50 % larger than `WTRG`. This is because the former is a more complex structure and requires a larger (constant) number of pointers to be represented. Multiplying by the σ nodes of the Huffman-shaped wavelet tree makes a significant difference when the alphabet is so large.

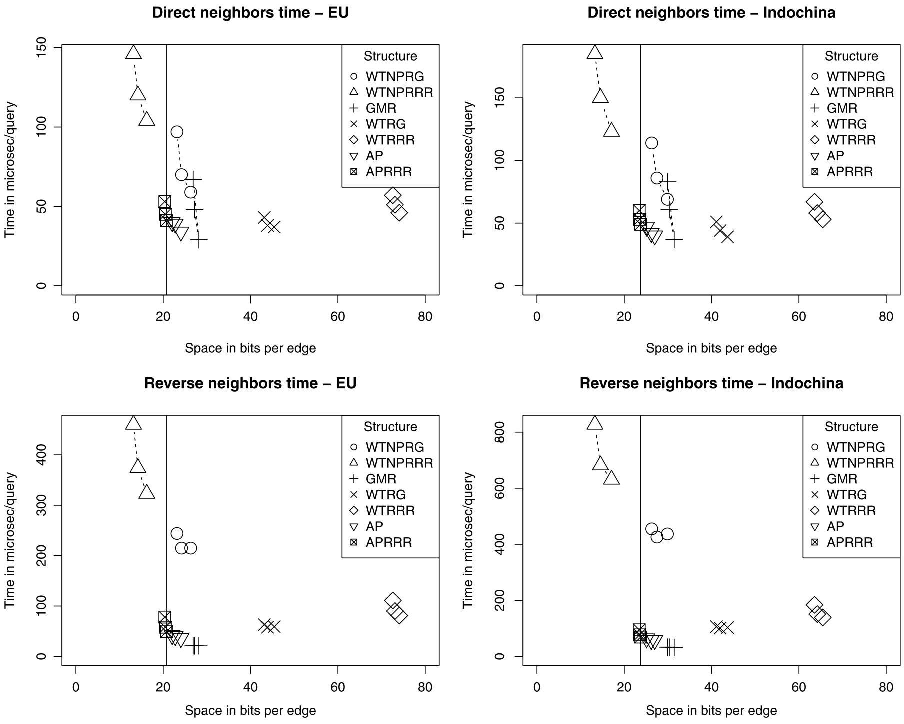

Fig. 5 Performance on Web graphs, to retrieve direct and reverse neighbors. *The vertical line* marks the bits per edge required by a plain adjacency list representation

9 Conclusions and Future Work

We have presented the first zero-order compressed representation of sequences supporting queries access, rank, and select in loglogarithmic time, so that the redundancy of the compressed representation is also compressed. That is, our space for sequence $s[1..n]$ over alphabet $[1..\sigma]$ is $n\mathcal{H}_0(s) + o(n)(\mathcal{H}_0(s) + 1)$ instead of the usual $n\mathcal{H}_0(s) + o(n \lg \sigma)$ bits. This is very important in many practical applications where the data is so highly compressible that a redundancy of $o(n \lg \sigma)$ bits would dominate the overall space. While there exist representations using even $n\mathcal{H}_0(s) + o(n)$ bits, ours is the first one supporting the operations in time $\mathcal{O}(\lg \lg \sigma)$ while breaking the $o(n \lg \sigma)$ redundancy barrier. Moreover, our time complexities are adaptive to the compressibility of the sequence, reaching average times $\mathcal{O}(\lg \mathcal{H}_0(s))$ under reasonable assumptions. We have given various byproducts of the result, where the compressed-redundancy property carries over representations of text indexes, permutations, functions, binary relations, and so on. It is likely that still other data structures can benefit from our compressed-redundancy representation. Finally, we have shown experimentally that our representation is highly practical, on large alphabets, both in synthetic and real-life application scenarios.

On the other hand, various interesting challenges on sequence representations remain open:

1. Use $n\mathcal{H}_k(s)+o(n)(\mathcal{H}_k(s)+1)$ bits of space, rather than $n\mathcal{H}_k(s)+o(n\lg\sigma)$ [4, 33] or our $n\mathcal{H}_0(s)+o(n)(\mathcal{H}_0(s)+1)$ bits, while still supporting the queries access, rank, and select efficiently.
2. Remove the $o(n\mathcal{H}_0(s))$ term from the redundancy while retaining loglogarithmic query times. Golynski et al. [29] have achieved $n\mathcal{H}_0(s)+o(n)$ bits of space, but the time complexities are exponentially higher on large alphabets, $\mathcal{O}(1+\frac{\lg\sigma}{\lg\lg n})$.
3. Lower the $o(n)$ redundancy term, which may be significant on highly compressible sequences. Our $o(n)$ redundancy is indeed $\mathcal{O}(\frac{n}{\lg\lg\lg n})$. That of Golynski et al. [29], $o(\frac{n\lg\sigma}{\lg n})$, is more attractive, at least for small alphabets. Moreover, for the binary case, Pătraşcu [53] obtained $\mathcal{O}(\frac{n}{\lg^c n})$ for any constant c, and this is likely to carry over multiary wavelet trees.

After the publication of the conference version of this paper, Belazzougui and Navarro [8] achieved a different tradeoff for one of our byproducts (Theorem 5). By spending $\mathcal{O}(n)$ further bits, they completely removed the terms dependent on σ in all time complexities, achieving $\mathcal{O}(m)$, $\mathcal{O}(\lg n)$ and $\mathcal{O}(r-l+\lg n)$ times for counting, locating and extracting, respectively. Their technique is based on monotone minimum perfect hash functions (mmphfs), which can also be used to improve some of our results on permutations, for example obtaining constant time for query $\pi(i)$ in Theorem 6 and thus improving all the derived results.[13] This is just one example of how lively current research is on this fundamental problem. Another example is the large amount of recent work attempting to close the gap between lower and upper bounds when taking into account compression, time and redundancy [25–27, 29, 33, 34, 53, 54]. Very recently, Belazzougui and Navarro [9] proved a lower bound of $\Omega(\lg\frac{\lg\sigma}{\lg w})$ for operation rank on a RAM machine of word size w, which holds for any space of the form $\mathcal{O}(nw^{\mathcal{O}(1)})$, and achieved this time within $\mathcal{O}(n\lg\sigma)$ bits of space. Then, making use of the results we present in this paper, they reduced the space to $n\mathcal{H}_0(s)+o(n\mathcal{H}_0(s))+o(n)$ bits. This illustrates how our technique can be easily used to move from linear-space to compressed-redundancy-space sequence representations.

To conclude, it is worth mentioning Navarro and Nekrich's recent result [49] on optimal representations of dynamic sequences, where in addition to the three operations we consider in this paper, one can insert and delete symbols at arbitrary positions. They obtain the optimal time $\mathcal{O}(\log n/\log\log n)$ for all the operations within essentially $n\mathcal{H}_0(s)+\mathcal{O}(n)$ bits of space. Using the alphabet partitioning idea, combining structures different from those we used here, is the key to remove any dependence on the alphabet size from the query times, to remove $o(n\lg\sigma)$ terms from the space, and to handle unbounded alphabets. This shows how the alphabet partitioning concept may have many more applications than those we are able to envision at this moment.

Acknowledgements We thank Djamal Belazzougui for helpful comments on a draft of this paper, and Meg Gagie for righting our grammar.

[13]Djamal Belazzougui, personal communication.

References

1. Arroyuelo, D., González, S., Oyarzún, M.: Compressed self-indices supporting conjunctive queries on document collections. In: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), pp. 43–54 (2010)
2. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN). LNCS, vol. 6034, pp. 170–183 (2010)
3. Barbay, J., Golynski, A., Munro, J.I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theor. Comput. Sci. **387**(3), 284–297 (2007)
4. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Trans. Algorithms **7**(4), 52 (2011)
5. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. ACM J. Exp. Algorithmics **14**(3), 7 (2009)
6. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. 26th Symposium on Theoretical Aspects of Computer Science (STACS), pp. 111–122 (2009)
7. Barbay, J., Navarro, G.: On compressing permutations and adaptive sorting. CoRR (2011). 1108.4408v1
8. Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. In: Proc. 19th Annual European Symposium on Algorithms (ESA). LNCS, vol. 6942, pp. 748–759 (2011)
9. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Proc. 20th Annual European Symposium on Algorithms (ESA). LNCS, vol. 7501, pp. 181–192 (2012)
10. Boldi, P., Vigna, S.: The WebGraph framework I: compression techniques. In: Proc. 13th World Wide Web Conference (WWW), pp. 595–602 (2004)
11. Brisaboa, N., Luaces, M., Navarro, G., Seco, D.: A new point access method based on wavelet trees. In: Proc. 3rd International Workshop on Semantic and Conceptual Issues in GIS (SeCoGIS). LNCS, vol. 5833, pp. 297–306 (2009)
12. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
13. Clark, D.: Compact Pat Trees. Ph.D. Thesis, University of Waterloo, Canada (1996)
14. Clarke, C., Cormack, G., Tudhope, E.: Relevance ranking for one to three term queries. In: Proc. 5th International Conference on Computer-Assisted Information Retrieval (RIAO), pp. 388–401 (1997)
15. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE), pp. 176–187 (2008)
16. Claude, F., Navarro, G.: Extended compact web graph representations. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Algorithms and Applications (Ukkonen Festschrift). LNCS, vol. 6060, pp. 77–91. Springer, Berlin (2010)
17. Claude, F., Navarro, G.: Fast and compact web graph representations. ACM Trans. Web **4**(4), 16 (2010)
18. Demaine, E., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 743–752 (2000)
19. Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A., Rodríguez, E.: Word-based self-indexes for natural language text. ACM Trans. Inf. Syst. **30**(1), 1 (2012)
20. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57**(1), 4 (2009)
21. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM **52**(4), 552–581 (2005)
22. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms **3**(2), 20 (2007)
23. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. **372**(1), 115–121 (2007)
24. Gagie, T., Nekrich, Y.: Worst-case optimal adaptive prefix coding. In: Proc. 11th International Symposium on Algorithms and Data Structures (WADS). LNCS, vol. 5664, pp. 315–326 (2009)
25. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comput. Sci. **387**(3), 348–359 (2007)
26. Golynski, A.: Cell probe lower bounds for succinct data structures. In: Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 625–634 (2009)
27. Golynski, A., Grossi, R., Gupta, A., Raman, R., Srinivasa Rao, S.: On the size of succinct indices. In: Proc. 15th Annual European Symposium on Algorithms (ESA). LNCS, vol. 4698, pp. 371–382 (2007)

28. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 368–373 (2006)
29. Golynski, A., Raman, R., Rao, S.: On the redundancy of succinct data structures. In: Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT). LNCS, vol. 5124, pp. 148–159 (2008)
30. González, R., Grabowski, Sz., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA), pp. 27–38 (2005). Posters
31. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 294–305 (2006)
32. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003)
33. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. In: Proc. 37th International Colloquim on Automata, Languages and Programming (ICALP), pp. 678–689 (2010)
34. Grossi, R., Orlandi, A., Raman, R., Srinivasa Rao, S.: More haste, less waste: lowering the redundancy in fully indexable dictionaries. In: Proc. 26th Symposium on Theoretical Aspects of Computer Science (STACS), pp. 517–528 (2009)
35. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2006)
36. Haskel, B., Puri, A., Netravali, A.: Digital Video: an Introduction to MPEG-2. Chapman & Hall, London (1997)
37. Hernández, C., Navarro, G.: Compression of web and social graphs supporting neighbor and community queries. In: Proc. 5th ACM Workshop on Social Network Mining and Analysis (SNA-KDD). ACM, New York (2011)
38. Hreinsson, J.B., Krøyer, M., Pagh, R.: Storing a compressed function with constant time access. In: Proc. 17th European Symposium on Algorithms (ESA), pp. 730–741 (2009)
39. Huffman, D.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1090–1101 (1952)
40. Levcopoulos, C., Petersson, O.: Sorting shuffled monotone sequences. Inf. Comput. **112**(1), 37–50 (1994)
41. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. **387**(3), 332–347 (2007)
42. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Trans. Algorithms **4**(3), 32 (2008)
43. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3), 407–430 (2001)
44. Mehlhorn, K.: Sorting presorted files. In: Proc. 4th GI-Conference on Theoretical Computer Science. LNCS, vol. 67, pp. 199–212 (1979)
45. Moffat, A., Turpin, A.: On the implementation of minimum-redundancy prefix codes. IEEE Trans. Commun. **45**(10), 1200–1207 (1997)
46. Munro, I.: Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 1180, pp. 37–42 (1996)
47. Munro, I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations and functions. Theor. Comput. Sci. **438**, 74–88 (2012)
48. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), 2 (2007)
49. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. In: Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2013, to appear)
50. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 10th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 60–70 (2007)
51. Pearlman, W., Islam, A., Nagaraj, N., Said, A.: Efficient, low-complexity image coding with a set-partitioning embedded block coder. IEEE Trans. Circuits Syst. Video Technol. **14**(11), 1219–1235 (2004)
52. Pennebaker, W., Mitchell, J.: JPEG: Still Image Data Compression Standard. Van Nostrand-Reinhold, New York (1992)
53. Pătraşcu, M.: Succincter. In: Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 305–313 (2008)
54. Pătraşcu, M.: A lower bound for succinct rank queries. CoRR (2009). arXiv:0907.1103v1 [cs.DS]
55. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43 (2007)

56. Russo, L., Navarro, G., Oliveira, A., Morales, P.: Approximate string matching with compressed indexes. Algorithms **2**(3), 1105–1136 (2009)
57. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms **48**(2), 294–313 (2003)
58. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1230–1239 (2006)
59. Said, A.: Efficient alphabet partitioning algorithms for low-complexity entropy coding. In: Proc. 15th Data Compression Conference (DCC), pp. 193–202 (2005)
60. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984)
61. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes, 2nd edn. Morgan Kaufmann, San Mateo (1999)