Alphabet Partitioning for Compressed Rank/Select and Applications

Jérémy Barbay[1], Travis Gagie[1*], Gonzalo Navarro[1*], and Yakov Nekrich[2]

[1] Department of Computer Science, University of Chile,
{jbarbay, tgagie, gnavarro}@dcc.uchile.cl
[2] Department of Computer Science, University of Bonn, yasha@cs.uni-bonn.de

Abstract. We present a data structure that stores a string $s[1..n]$ over the alphabet $[1..\sigma]$ in $nH_0(s) + o(n)(H_0(s){+}1)$ bits, where $H_0(s)$ is the zero-order entropy of s. This data structure supports the queries access and rank in time $\mathcal{O}(\lg\lg\sigma)$, and the select query in constant time. This result improves on previously known data structures using $nH_0(s) + o(n\lg\sigma)$ bits, where on highly compressible instances the redundancy $o(n\lg\sigma)$ cease to be negligible compared to the $nH_0(s)$ bits that encode the data. The technique is based on combining previous results through an ingenious partitioning of the alphabet, and practical enough to be implementable. It applies not only to strings, but also to several other compact data structures. For example, we achieve (*i*) faster search times and lower redundancy for the smallest existing full-text self-index; (*ii*) compressed permutations π with times for $\pi()$ and $\pi^{-1}()$ improved to log-logarithmic; and (*iii*) the first compressed representation of dynamic collections of disjoint sets.

1 Introduction

Search queries on strings have many important applications, to the point that one is willing to sacrifice some additional space to index the string in order to support the queries in less time. The most important queries serve as primitives to implement many other operations, in particular pattern matching in full-text databases (see, e.g., [18, 7, 14, 19] for recent discussions): given a string s, $s.\mathsf{access}(i)$ returns the ith character of s, which we denote $s[i]$; $s.\mathsf{rank}_a(i)$ returns the number of occurrences of the character a up to position i; and $s.\mathsf{select}_a(i)$ returns the position of the ith occurrence of a in s.

Wavelet trees [11] represent a string $s[1..n]$ over alphabet $[1..\sigma]$ within $n\lg\sigma + o(n\lg\sigma)$ bits, where lg denotes the logarithm in base two. The indexing space in $o(n\lg\sigma)$ is considered asymptotically “negligible” compared to the $n\lg\sigma$ bits required to hold the main data, while providing support for the queries in time $\mathcal{O}(\lg\sigma)$. Later results [10] improved the times to $\mathcal{O}(\lg\lg\sigma)$.

Regularities in the string permit further reductions in the space, from $n\lg\sigma$ bits down to $nH_k(s)$ bits, where $H_k(s)$ denotes the kth-order empirical entropy

[*] Funded in part by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

Table 1. Recent bounds and our new ones for data structures supporting access, rank and select. The first row holds for $\sigma = \mathcal{O}(\text{polylog}(n))$ and the second for $\sigma = o(n)$. The space bound in the sixth row holds for $k = o(\log_\sigma n)$. The times of our Thm. 1 can be refined into a more complicated formula (see also Cor. 1).

	space (bits)	access	rank	select
[8, Thm. 3.2]	$nH_0(s) + o(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
[8, Cor. 3.3]	$nH_0(s) + o(n \lg \sigma)$	$\mathcal{O}\left(1 + \frac{\lg \sigma}{\lg \lg n}\right)$	$\mathcal{O}\left(1 + \frac{\lg \sigma}{\lg \lg n}\right)$	$\mathcal{O}\left(1 + \frac{\lg \sigma}{\lg \lg n}\right)$
[10, Thm. 2.2]	$n \lg \sigma + o(n \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
[10, Thm. 2.2]	$n \lg \sigma + o(n \lg \sigma)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma \lg \lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$
[3, Lem. 4.1]	$nH_0(s) + o(n \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
[3, Thm. 4.2]	$nH_k(s) + o(n \lg \sigma)$	$\mathcal{O}(1)$	$\mathcal{O}\left((\lg \lg \sigma)^2 \lg \lg \lg \sigma\right)$	$\mathcal{O}(\lg \lg \sigma \lg \lg \lg \sigma)$
Thm 1	$nH_0(s) + o(n)(H_0(s) + 1)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$	$\mathcal{O}(1)$
Thm 1	$nH_0(s) + o(n)(H_0(s) + 1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg \lg \sigma \lg \lg \lg \sigma)$	$\mathcal{O}(\lg \lg \sigma)$

of s (i.e., the minimum self-information of s with respect to a kth-order Markov source; see Manzini [15] for a definition and discussion). The challenge of compressing the string *while still supporting the queries efficiently* was also achieved, using as little as $nH_0(s) + o(n \lg \sigma)$ [11, 8, 3] and even $nH_k(s) + o(n \lg \sigma)$ bits [3] (for any $k = o(\log_\sigma n)$) while retaining the time complexities.

One problem with such space is that, on highly compressible data, the $o(n \lg \sigma)$ bits of the index are not always negligible compared to the space used to encode the compressed data. Hence the challenge is to retain 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 entropy of the text, plus $o(n)$ bits.

For comparison, the representation by Golynski et al. [10] does not compress[1] s and uses additional $\mathcal{O}\left(\frac{n \lg \sigma}{\lg \lg \sigma}\right) = o(n \lg \sigma)$ bits, but offers log-logarithmic times for the queries. Ferragina et al.'s wavelet tree [8] achieves zero-order compression plus $\mathcal{O}\left(\frac{n \lg \sigma \lg \lg n}{\lg n}\right) = o(n \lg \sigma)$ bits, supporting the queries in $\mathcal{O}\left(1 + \frac{\lg \sigma}{\lg \lg n}\right)$ time. Barbay et al. [3] obtain zero-order space and log-logarithmic times, but their redundancy is still $o(n \lg \sigma)$. See Table 1 for a summary of our bounds and previous ones.[2]

In Section 2 we show how to combine the strengths of these data structures, obtaining not only zero-order compressed space and log-logarithmic times, but

[1] In terms of the usual entropy measures. It compresses to the k-th order entropy of a different sequence (A. Golynski, personal communication).

[2] When we write $o(n \lg \sigma)$ bits we mean $o(n) \lg \sigma$. Although in some cases [10, 3] the results are actually $n\, o(\lg \sigma)$, we point out that this can be taken as $o(n) \lg \sigma$ because, if $\sigma = \mathcal{O}(\text{polylog}(n))$, one can use a structure by Ferragina et al. [8, Thm. 3.2] that solves access, rank, and select in constant time using $nH_0(s) + o(n)$ bits. Thus one can assume $\sigma = \omega(1)$ at the very least. See also Footnote 6 of Barbay et al. [3].

also compressed redundancy. The technique can be summarized as partitioning the alphabet into sub-alphabets according to the characters' frequencies in s, storing in a multiary wavelet tree [8] the string that results from replacing the characters in s by identifiers of their sub-alphabets, and storing separate strings, each the projection of s to the characters of s belonging to each sub-alphabet, this time using Golynski et al.'s [10] structure for large alphabets. We achieve a data structure that stores a string $s[1..n]$ in $nH_0(s)+o(n)(H_0(s)+1)$ bits, thus guaranteeing that the redundancy stays negligible even when the text is very compressible. It supports queries in the times shown in Table 1 (rows 7 and 8 give two alternatives).

Then we consider various extensions and applications of our main result. In Section 3 we show how our result can be used to improve an existing text index that achieves k-th order entropy [8, 3], so as to improve its redundancy and query times. This way we achieve the first self-index with space bounded by $nH_k(s)+o(n)(H_k(s)+1)$ bits, able of counting and locating pattern occurrences and extracting any substring of s, within the time complexities achieved by either of its predecessors. In Sections 4 and 5, respectively, we show how to apply our data structure to store a compressed permutation, a compressed function and a compressed dynamic collection of disjoint sets, while supporting a rich set of operations on those. This improves or gives alternatives to the best previous results [4, 17, 12]. We have approached these applications in such a way that an improvement to our main result, however achieved, translates into improved bounds for them as well.

2 Alphabet partitioning

Let $s[1..n]$ be a sequence over effective alphabet $[1..\sigma]$, i.e., every character appears in s, so $\sigma \leq n$. (At the end of the section we handle the case of large alphabets.) The zero-order entropy of s is $H_0(s) = \sum_{a \in [1..\sigma]} \frac{|s|_a}{n} \lg \frac{n}{|s|_a}$, where $|s|_a$ is the number of occurrences of the character a in s. Note that by convexity we have $nH_0(s) \geq (\sigma-1)\lg n + (n-\sigma+1)\lg(n/(n-\sigma+1))$, a property we will use later.

Our results are based on the following alphabet partitioning scheme. Let $m[1..\sigma]$ be the sequence assigning to each character $a \in [1..\sigma]$ the value

$$m[a] = \lceil \lg(n/|s|_a) \lg n \rceil \leq \lceil \lg^2 n \rceil \ .$$

Let $t[1..n]$ be the string over $\left[1..\lceil \lg^2 n \rceil\right]$ obtained from s by replacing each occurrence of a by $m[a]$, for $1 \leq a \leq \sigma$. For $0 \leq \ell \leq \lceil \lg^2 n \rceil$, let σ_ℓ be the number of occurrences of ℓ in m or, equivalently, the number of distinct characters of s replaced by ℓ in t. Finally, let $s_\ell[1..|t|_\ell]$ be the string over $[1..\sigma_\ell]$ defined by $s_\ell[t.\mathsf{rank}_\ell(i)] = m.\mathsf{rank}_\ell(s[i])$.

Notice that, if both a and b are replaced by the same number in t, then $\lg(n/|s|_b) - \lg(n/|s|_a) < 1/\lg n$ and so $|s|_a/|s|_b < 2^{1/\lg n}$. It follows that, if a is replaced by ℓ in t, then $\sigma_\ell < 2^{1/\lg n}|s_\ell|/|s|_a$ (by fixing a and summing over all those b replaced by ℓ). Since

$$\sum_{\lceil \lg(n/|s|_a)\lg n\rceil=\ell} |s|_a = |s_\ell| \qquad \text{and} \qquad \sum_a |s|_a = \sum_\ell |s_\ell| = n\,,$$

we have

$$\begin{aligned}
& nH_0(t) + \sum_\ell |s_\ell| \lg \sigma_\ell \\
&< \sum_\ell |s_\ell| \lg(n/|s_\ell|) + \sum_\ell \sum_{\lceil \lg(n/|s|_a)\lg n\rceil=\ell} |s|_a \lg\left(2^{\frac{1}{\lg n}}|s_\ell|/|s|_a\right) \\
&= \sum_a |s|_a \lg(n/|s|_a) + n/\lg n \\
&= nH_0(s) + o(n)\,.
\end{aligned}$$

In other words, if we represent t with $H_0(t)$ bits per symbol and each s_ℓ with $\lg \sigma_\ell$ bits per symbol, we achieve a good overall compression. Thus we can obtain a very compact representation of a string s by storing a compact representation of t and storing each s_ℓ as an "uncompressed" string over an alphabet of size σ_ℓ.

Now we show how our approach can be used to obtain a fast and compact rank/select data structure. Suppose we have a data structure T that supports access, rank and select queries on t; another structure M that supports the same queries on m; and data structures $S_1, \ldots, S_{\lceil \lg^2 n\rceil}$ that support the same queries on $s_1, \ldots, s_{\lceil \lg^2 n\rceil}$. With these data structures we can implement

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

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

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

We implement T and M as multiary wavelet trees [8]; we implement each S_ℓ as either a multiary wavelet tree or an instance of Golynski et al.'s [10, Thm. 2.2] access/rank/select data structure, depending on whether $\sigma_\ell \leq \lg n$ or not. The wavelet tree for T uses at most $nH_0(t) + \mathcal{O}\left(\frac{n(\lg\lg n)^2}{\lg n}\right)$ bits and operates in constant time, because its alphabet size is polylogarithmic. If S_ℓ is implemented as a wavelet tree, it uses at most $|s_\ell|H_0(s_\ell) + \mathcal{O}\left(\frac{|s_\ell|\lg|s_\ell|\lg\lg n}{\lg n}\right)$ bits[3] and operates in constant time for the same reason; otherwise it uses at most $|s_\ell|\lg\sigma_\ell + \mathcal{O}\left(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\sigma_\ell}\right) \leq |s_\ell|\lg\sigma_\ell + \mathcal{O}\left(\frac{|s_\ell|\lg\sigma_\ell}{\lg\lg\lg n}\right)$ bits (the latter because $\sigma_\ell > \lg n$). Thus in either case the space for s_ℓ is bounded by $|s_\ell|\lg\sigma_\ell + \mathcal{O}\left(\frac{|s_\ell|\lg|s_\ell|}{\lg\lg\lg n}\right)$

[3] 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 $\mathcal{O}\left(\sqrt{n}\,\mathrm{polylog}(n)\right) = o(n)$ bits. Therefore all of our $o(\cdot)$ expressions involving n and other variables will be asymptotic in 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 $\mathcal{O}\left(\sigma \lg \lg n\right)$ bits. Because of the property we referred to in the beginning of this section, $nH_0(s) \geq (\sigma - 1) \lg n$, this space is $H_0(s)\, \mathcal{O}\left(\frac{n \lg \lg n}{\lg n}\right)$. By these calculations, the space for T, M and the S_ℓ's adds up to $nH_0(s) + o(n)H_0(s) + o(n)$, where the $o(n)$ term is $\mathcal{O}\left(\frac{n}{\lg \lg \lg n}\right)$.

Depending on which time tradeoff we use for Golynski et al.'s data structure, we obtain the results of Table 1. We can refine the time complexity by noticing that the only non-constant times are due to operating on some string 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 $\lg \lg \sigma_\ell = \mathcal{O}\left(\lg \lg \min(\sigma, n/|s|_a)\right)$.

Theorem 1. *We can store* $s[1..n]$ *over effective alphabet* $[1..\sigma]$ *in* $nH_0(s) + o(n)(H_0(s) + 1)$ *bits and support* access*,* rank *and* select *queries in* $\mathcal{O}\left(\lg \lg \sigma\right)$*,* $\mathcal{O}\left(\lg \lg \sigma\right)$*, and* $\mathcal{O}\left(1\right)$ *time, respectively (variant* (i)*). Alternatively, we can support* access*,* rank *and* select *queries in* $\mathcal{O}\left(1\right)$*,* $\mathcal{O}\left(\lg \lg \sigma \lg \lg \lg \sigma\right)$ *and* $\mathcal{O}\left(\lg \lg \sigma\right)$ *time, respectively (variant* (ii)*). Any of the* σ *terms in these time complexities is actually* $\min(\sigma, n/|s|_a)$*, where* a *stands for* $s[i]$ *in the time of the* access *query, and for the character argument in the time of the* rank *and* select *query.*

Moreover, by implementing S_ℓ as a wavelet tree whenever $\sigma_\ell \leq (\lg n)^{\lg \lg \lg n}$, we ensure to achieve the complexities of wavelet trees if those are better than the ones given above. That is, for example, $\mathcal{O}\left(\min\left(1 + \frac{\lg \sigma_\ell}{\lg \lg n}, \lg \lg \sigma_\ell\right)\right)$ instead of just $\mathcal{O}\left(\lg \lg \sigma_\ell\right)$. We can similarly match the complexity $\mathcal{O}\left(\lg \lg \sigma_\ell \lg \lg \lg \sigma_\ell\right)$. Note that, if we do this, the complexities that were $\mathcal{O}\left(1\right)$ become $\mathcal{O}\left(1 + \frac{\lg \sigma_\ell}{\lg \lg n}\right)$.

Corollary 1. *All the time complexities up to* $\mathcal{O}\left(\lg \lg \sigma\right)$ *in variants* (i) *or* (ii) *of Theorem 1 can be made* $\mathcal{O}\left(\min\left(1 + \frac{\lg \sigma}{\lg \lg n}, \lg \lg \sigma\right)\right)$*. Alternatively, all time complexities in variant* (ii) *can be made* $\mathcal{O}\left(\min\left(1 + \frac{\lg \sigma}{\lg \lg n}, \lg \lg \sigma \lg \lg \lg \sigma\right)\right)$*. As in Theorem 1, the* σ *term is actually* $\min(\sigma, n/|s|_a)$*.*

In the most general case, s is a sequence over an alphabet Σ which is not an effective alphabet, and σ symbols 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 data structure described by Raman et al. [20], 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); for any $i \in [1..\sigma]$ the i-th smallest element in Σ' can be found. The indexed dictionary of Raman et al. [20] uses $\sigma \lg(e\mu/\sigma) + o(\sigma) + \mathcal{O}\left(\lg \lg \mu\right)$ 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 $\mathcal{O}\left(\lg \mu\right)$ bits. We replace every element in s by its rank in Σ', and the resulting string is stored using Theorem 1. Hence, in the general case the space usage is increased by $\sigma \lg(e\mu/\sigma) + o(\sigma) + \mathcal{O}\left(\lg \mu\right)$ bits and the asymptotic time complexity of queries remains unchanged.

3 Reduced redundancy on self-indexes

Our result can be readily carried over self-indexes. These also represent a sequence, but they support other operations related to text searching. A well known self-index [8] achieves k-th order entropy space by partitioning the Burrows-Wheeler transform [6] of the sequence and encoding each partition to its zero-order entropy. Those partitions must support queries access and rank. By using Theorem 1(i) to represent such partitions, we achieve the following result, improving previous ones [8, 10, 3].

Theorem 2. *Let $s[1..n]$ be a string over alphabet*[4] *$[1..\sigma]$. Then we can represent s using $nH_k(s) + o(n)(H_k(s) + 1)$ bits of space, for any $k \leq (\alpha \log_\sigma n) - 1$ and constant $0 < \alpha < 1$, 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 $\mathcal{O}((r - l) \lg\lg \sigma + \lg n \lg\lg\lg n \lg\lg \sigma)$. The $\lg\lg \sigma$ times can be reduced to $\mathcal{O}\left(1 + \frac{\lg \sigma}{\lg\lg n}\right)$ if convenient.*

For these particular locating and extracting times we are sampling one out of every $\lg n \lg\lg\lg n$ text positions, which maintains our lower-order space term $o(n)$ at $\mathcal{O}(n/ \lg\lg\lg n)$. Compared to Theorem 4.2 of Barbay et al. [3], we reduce the redundancy from $o(n) \lg \sigma$ to $o(n)(H_k(s) + 1)$. Our improved locating times, however, just owe to the denser sampling, which they could also use.

4 Compressing permutations

We now show how to use access/rank/select data structures to store a compressed permutation. We follow Barbay and Navarro's notation [4] and improve their space and, especially, their time performance. They measure 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 [13, 4, 16]) of lengths $\mathsf{runs}(\pi) = \langle n_1, \ldots, n_\rho \rangle$. Then $H(\mathsf{runs}(\pi)) = \sum \frac{n_i}{n} \lg \frac{n}{n_i} \leq \lg \rho$ is called the *entropy* of the runs (and, because $n_i \geq 1$, it also holds $nH(\mathsf{runs}(\pi)) \geq (\rho - 1) \lg n$). We first consider permutations which are interleaved sequences of increasing or decreasing values as first defined by Levcopoulos et al. [13] for adaptive sorting, and later on for compression [4], and then give improved results for more specific classes of runs. In both cases we consider first the application of the permutation $\pi()$ and its inverse, $\pi^{-1}()$, to show later how to extend the support to the iterated applications of the permutation, $\pi^k()$, extending and improving previous results [17].

Theorem 3. *Let π be a permutation on n elements that consists of ρ interleaved increasing or decreasing runs, of lengths $\mathsf{runs}(\pi)$. We can store π in $2nH(\mathsf{runs}(\pi)) + o(n)(H(\mathsf{runs}(\pi)) + 1)$ bits and perform $\pi()$ and $\pi^{-1}()$ queries in $\mathcal{O}\left(\min\left(1 + \frac{\lg \rho}{\lg\lg n}, \lg\lg \rho\right)\right)$ time.*

[4] Again, $[1..\sigma]$ does not need to be the effective alphabet (see paragraph after Thm. 1).

Proof. We first replace all the elements of the rth run by r, for $1 \leq r \leq \rho$. Let s be the resulting string and let s' be s permuted according to π, that is, $s'[\pi(i)] = s[i]$. We store s and s' using Theorem 1(i) and store ρ bits indicating whether each run is increasing or decreasing. Notice that, if $\pi(i)$ is part of an increasing run, then $s'.\mathsf{rank}_{s[i]}(\pi(i)) = s.\mathsf{rank}_{s[i]}(i)$, so

$$\pi(i) = s'.\mathsf{select}_{s[i]}\left(s.\mathsf{rank}_{s[i]}(i)\right) ;$$

if $\pi(i)$ 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

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

A $\pi^{-1}()$ query is symmetric. The space of the bitmap is $\rho \in o(n)H(\mathsf{runs}(\pi))$ because $nH(\mathsf{runs}(\pi)) \geq (\rho - 1)\lg n$. □

We now consider the case of runs restricted to be strictly incrementing (+1) or decrementing (−1), while still letting them be interleaved: such runs were not directly considered before.

Theorem 4. *Let π be a permutation on n elements that consists of ρ interleaved strictly incrementing or decrementing runs. For any constant $\epsilon > 0$, we can store π in $nH(\mathsf{runs}(\pi)) + o(n)(H(\mathsf{runs}(\pi)) + 1) + \mathcal{O}(\rho n^{\epsilon})$ bits and perform $\pi()$ queries in $\mathcal{O}\left(\min\left(1 + \frac{\lg \rho}{\lg\lg n}, \lg\lg \rho\right)\right)$ time and $\pi^{-1}()$ queries in $\mathcal{O}(1/\epsilon)$ time.*

Proof. We first replace all the elements of the rth run by r, for $1 \leq r \leq \rho$, considering the runs in order by minimum element. Let $s \in \{1, \ldots, \rho\}^n$ be the resulting string. We store s using Theorem 1(i); we also store an array containing the runs' lengths, directions (incrementing or decrementing), and minima, in order by minimum element; and store a predecessor data structure containing the runs' minima as keys with their positions in the array as auxiliary information. The predecessor data structure is based on Lemma 4 of Andersson's paper [1]. It is an n^{ϵ}-ary trie where the keys are sought considering $\epsilon \lg n$ bits per trie node, and hence found in $\mathcal{O}(1/\epsilon)$ time. Each of the ρ elements may require $\mathcal{O}((1/\epsilon)n^{\epsilon} \lg n)$ bit space for the n^{ϵ}-size children arrays along its $\mathcal{O}(1/\epsilon)$-length path. By slightly adjusting ϵ the space is $\mathcal{O}(\rho n^{\epsilon})$ bits. With these data structures, we can retrieve a run's data given either its array index or any of its elements.

If $\pi(i)$ is the jth element in an incrementing run whose minimum element is m, then $\pi(i) = m + j - 1$; on the other hand, if $\pi(i)$ is the jth element of a decrementing run of length l whose minimum element is m, then $\pi(i) = m + l - j$. It follows that, given i, we can compute $\pi(i)$ by using the query $j = s.\mathsf{rank}_{s[i]}(i)$ and then an array lookup at position $s[i]$ to find m, l and the direction, finally computing $\pi(i)$ from them. Also, given $\pi(i)$, we can compute i by first using a predecessor query to find the run's array position r, then an array lookup to find m, l and the direction, then computing $j = \pi(i) - m + 1$ (increasing) or $j = m + l - \pi(i)$ (decreasing), and finally using the query $i = s.\mathsf{select}_r(j)$. □

Notice that, if π consists of ρ contiguous increasing or decreasing runs, then π^{-1} consists of ρ interleaved incrementing or decrementing runs. Therefore, Theorem 4 applies to such permutations as well, with the time bounds for $\pi()$ and $\pi^{-1}()$ queries reversed, which yields the following corollary:

Corollary 2. *Let π be a permutation on n elements that consists of ρ contiguous increasing or decreasing runs. For any constant $\epsilon > 0$, we can store π in $nH(\mathsf{runs}(\pi)) + o(n)(H(\mathsf{runs}(\pi)) + 1) + \mathcal{O}(\rho n^{\epsilon})$ bits and perform $\pi()$ queries in $\mathcal{O}(1/\epsilon)$ time and $\pi^{-1}()$ queries in $\mathcal{O}\left(\min\left(1 + \frac{\lg \rho}{\lg \lg n}, \lg \lg \rho\right)\right)$ time.*

If π's runs are both contiguous and incrementing or decrementing, then so are the runs of π^{-1}. In this case we can store π in $\mathcal{O}(\rho n^{\epsilon})$ bits and answer $\pi()$ and $\pi^{-1}()$ queries in $\mathcal{O}(1)$ time. To do this, we use two predecessor data structures: for each run, in one of the data structures we store the position j in π of the first element of the run, with $\pi(j)$ as auxiliary information; in the other, we store $\pi(j)$, with j as auxiliary information. To perform a query $\pi(i)$, we use the first predecessor data structure to find the starting position j of the run containing i, and return $\pi(j)+i-j$. A $\pi^{-1}()$ query is symmetric. Decreasing runs are handled as before.

Corollary 3. *Let π be a permutation on n elements that consists of ρ contiguous incrementing or decrementing runs. For any constant $\epsilon > 0$, we can store π in $\mathcal{O}(\rho n^{\epsilon})$ bits and perform $\pi()$ and $\pi^{-1}()$ queries in $\mathcal{O}(1/\epsilon)$ time.*

We now show how to achieve exponentiation ($\pi^{k}(i)$, $\pi^{-k}(i)$) within compressed space. Munro et al. [17] 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. Expressing their result as a succinct index and combining it with any of our results does yield a compression, but one where the space depends of the lengths of both the runs and cycles of π. The following construction, extending the technique from Munro et al. [17], retains the compressibility properties of π by building a companion data structure that uses small space to support the exponentiation, thus allowing the compression of the main data structure with any of our results.

Theorem 5. *Suppose we have a data structure D that stores a permutation π on n elements and supports queries $\pi()$ in time $g(\pi)$. Then for any $t \leq n$, we can build a succinct index that takes $\mathcal{O}((n/t)\lg n)$ bits and, when used in conjunction with D, supports $\pi^{k}()$ and $\pi^{-k}()$ queries in $\mathcal{O}(t\, g(\pi))$ time.*

Proof. We decompose π into its cycles and, for every cycle of length at least t, store the cycle's length and an array containing pointers to every tth element in the cycle, which we call 'marked'. We also store a compressed binary string, aligned to π, indicating the marked elements. For each marked element, we record to which cycle it belongs and its position in the array of that cycle.

To compute $\pi^{k}(i)$, we repeatedly apply $\pi()$ at most t times until we either loop (in which case we need apply $\pi()$ at most t more times to find $\pi^{k}(i)$ in the loop) or we find a marked element. Once we have reached a marked element, we use its array position and cycle length to find the pointer to the last marked element in the cycle before $\pi^k(i)$, and the number of applications of $\pi()$ needed to map that to $\pi^k(i)$ (at most t). A π^{-k} query is similar (note that it does not need to use $\pi^{-1}()$). □

As an example, given a constant $\epsilon > 0$ and a value $t \leq n$, we can combine Corollary 2 and Theorem 5 to obtain a data structure that stores Sadakane's Ψ function [21] for s in $nH_0(s) + o(n)(H_0(s) + 1) + \mathcal{O}\left(\sigma n^\epsilon + (n/t) \lg n\right)$ bits and supports $\Psi^k()$ and $\Psi^{-k}()$ queries in $\mathcal{O}\left(1/\epsilon + t\right)$ time; these queries are useful when working on compressed suffix arrays and trees.

5 Compressing functions and dynamic collections of disjoint sets

Hreinsson, Krøyer and Pagh [12] recently showed how, given $X = \{x_1, \ldots, x_n\} \subseteq [U]$ and $f : [U] \rightarrow [1..\sigma]$, where $[U]$ is the set of numbers whose binary representations fit in a machine word, they can store f restricted to X in compressed form with constant-time evaluation. Their representation uses at most $(1+\delta)nH_0(f) + n \min(p_{\max} + 0.086, 1.82(1 - p_{\max})) + o(\sigma)$ bits, where $\delta > 0$ is a given constant and $p_{\max}$ is the relative frequency of the most common function value. We note that this bound holds even when $\sigma \gg n$.

Notice that, in the special case where $X = [1..n]$ and $\sigma \leq n$, we can achieve constant-time evaluation and a better space bound using Theorem 1. We can also find all the elements in $[1..n]$ that f maps to a given element in $[1..\sigma]$ (using select), find an element's rank among the elements with the same image, or the size of the preimage (using rank), etc.

Theorem 6. *Let $f : [1..n] \rightarrow [1..\sigma]$ be a surjective function.*[5] *We can represent f using $nH_0(f) + o(n)(H_0(f) + 1)$ bits so that any $f(i)$ can be computed in $\mathcal{O}(1)$ time. Moreover, each element of $f^{-1}(a)$ can be computed in $\mathcal{O}(\lg\lg\sigma)$ time, and $|f^{-1}(a)|$ requires time $\mathcal{O}(\lg\lg\sigma \lg\lg\lg\sigma)$. Alternatively we can compute $f(i)$ and $|f^{-1}(a)|$ in time $\mathcal{O}(\lg\lg\sigma)$ and deliver any element of $f^{-1}(a)$ in $\mathcal{O}(1)$ time.*

We omit the other improvements of Theorem 1 and Corollary 1 for conciseness. We can also achieve interesting results with our theorems from Section 4, as runs arise naturally in many real-life functions. For example, suppose we decompose $f(1), \ldots, f(n)$ into ρ interleaved non-increasing or non-decreasing runs. Then we can store it as a combination of the permutation π that stably sorts the values $f(i)$, plus a compressed rank/select data structure storing a binary string $b[1..n + \sigma + 1]$ with $\sigma + 1$ bits set to 1: if f maps i values in $[1..n]$ to a value j in

[5] So that $[1..\sigma]$ is the effective alphabet size of string f. General functions with image of size $\sigma' < \sigma$ require $\mathcal{O}\left(\sigma' \lg(\sigma/\sigma')\right) + o(\sigma)$ extra bits, or we can handle them using $\mathcal{O}\left(\sigma \lg\lg n\right)$ bits with our structure M.

$[1..\sigma]$ then, in b, there are i bits set to 0 between the jth and $(j+1)$th bits set to 1. Therefore,

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

and the theorem below follows immediately from Theorem 3. Similarly, $f^{-1}(a)$ is obtained by applying $\pi^{-1}()$ to the area $b.\mathsf{rank}_0(b.\mathsf{select}_1(a))+1 \ldots b.\mathsf{rank}_0(b.\mathsf{select}_1 (a+1))$, and $|f^{-1}(a)|$ is computed in $\mathcal{O}(1)$ time. Notice that $H(\mathsf{runs}(\pi)) = H(\mathsf{runs}(f)) \leq H_0(f)$, and that b can be stored in $\mathcal{O}\left(\sigma \lg \frac{n}{\sigma}\right) + o(n)$ bits [20].

Corollary 4. *Let $f : [1..n] \to [1..\sigma]$ be a surjective function*[6] *with $f(1), \ldots, f(n)$ consisting of ρ interleaved non-increasing or non-decreasing runs. Then we can store f in $2nH(\mathsf{runs}(f)) + o(n)(H(\mathsf{runs}(f)) + 1) + \mathcal{O}\left(\sigma \lg \frac{n}{\sigma}\right)$ bits and compute any $f(i)$, as well as retrieve any element in $f^{-1}(a)$, in $\mathcal{O}(\lg \lg \rho)$ time. The size $|f^{-1}(a)|$ can be computed in $\mathcal{O}(1)$ time.*

We can obtain a more competitive result if f is split into contiguous runs, but their entropy is no longer bounded by the zero-order entropy of string f.

Corollary 5. *Let $f : [1..n] \to [1..\sigma]$ be a surjective function with $f(1), \ldots, f(n)$ consisting of ρ contiguous non-increasing or non-decreasing runs. Then we can represent f in $nH(\mathsf{runs}(f)) + o(n)(H(\mathsf{runs}(f)) + 1) + \mathcal{O}(\rho n^{\epsilon}) + \mathcal{O}\left(\sigma \lg \frac{n}{\sigma}\right)$ bits, for any constant $\epsilon > 0$, and compute any $f(i)$ in $\mathcal{O}(\lg \lg \sigma)$ time, as well as retrieve any element in $f^{-1}(a)$ in $\mathcal{O}(1/\epsilon)$ time. The size $|f^{-1}(a)|$ can be computed in $\mathcal{O}(1)$ time.*

Finally, we now give what is, to the best of our knowledge, the first result about storing a compressed collection of disjoint sets. The key point in the next theorem is that, as the sets in the collection C are merged, our space bound shrinks with the entropy of the distribution $\mathsf{sets}(C)$ of elements to sets.

Theorem 7. *Let C be a collection of disjoint sets whose union is $[1..n]$. For any constant $\epsilon > 0$, we can store C in $(1+\epsilon)nH(\mathsf{sets}(C)) + \mathcal{O}(|C| \lg n) + o(n)$ bits and perform any sequence of m union and find operations in a total of $\mathcal{O}((1/\epsilon)(m+n) \lg \lg n)$ time.*

Proof. We first use Theorem 1 to store the string $s[1..n]$ in which each $s[i]$ is the representative of the set containing i. We then store the representatives in a standard disjoint-set data structure D [22]. Together, our data structures take $nH(\mathsf{sets}(C)) + \mathcal{O}(|C| \lg n) + o(n)(H(\mathsf{sets}(C)) + 1)$ 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 $D.\mathsf{union}(D.\mathsf{find}(s[i]), D.\mathsf{find}(s[j]))$.

For our data structure to shrink as we union sets, we keep track of $H(\mathsf{sets}(C))$ 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 [22]: Only the first time one accesses a node $v \in C$ it may occur that the representative is not directly v's parent.

[6] Otherwise we proceed as usual to map the domain to the effective one.

Reconstructing the structure of Theorem 1 takes also $\mathcal{O}(n)$ time: As we need just access on s, we need only rank and access on our multiary wavelet tree and access on the s_ℓ sequences. Thus the latter are implemented simply as arrays and the former are also easily built in linear time for these two queries [8].

Since $H(\mathsf{sets}(C))$ is always less than $\lg n$, we rebuild only $\mathcal{O}\left(\log_{1+\epsilon} \lg n\right) = \mathcal{O}\left((1/\epsilon) \lg \lg n\right)$ times. Finally, the space term $o(n)H(\mathsf{sets}(C))$ is absorbed by $\epsilon H(\mathsf{sets}(C))$ by slightly adjusting ϵ. □

6 Conclusions and future work

We have presented the first zero-order compressed representation of strings efficiently supporting queries access, rank, and select, so that the redundancy of the compressed representation is also compressed. That is, our space for string $s[1..n]$ over alphabet $[1..\sigma]$ is $nH_0(s) + o(n)(H_0(s) + 1)$ instead of the usual $nH_0(s) + o(n) \lg \sigma$ bits. This is very important in many practical applications where the data is highly compressible and the redundancy would otherwise dominate the overall space.

In the full paper we will work on several improvements and further applications. First, we can reduce the dependence on the alphabet size from $\mathcal{O}(\sigma \lg \lg n)$ to $\mathcal{O}(\sigma)$ by storing a length-restricted Shannon code in $\mathcal{O}(\sigma)$ bits [9] instead of the data structure M. To avoid the $\mathcal{O}(1)$ extra redundancy per character associated with using a length-restricted prefix code, we replace each character in s whose codeword length is at most $\lg \lg n$ by a distinct number in t. This increases the alphabet size of t by at most $\lg n$; calculation shows that our space bound increases by an $\mathcal{O}(1 + 1/\lg \lg n)$-factor and, thus, remains at most $nH_0(s) + o(n)(H_0(s) + 1)$. Second, given any constant c, we can reduce the $\min(\sigma, n/|s|_a)$ in our time bounds by a factor of $(\lg n)^c$; to do this, we further partition each sub-alphabet into $(\lg n)^c$ sub-sub-alphabets. Third, our alphabet partitioning techniques yields a compressed representation of posting lists of sizes $(n_1, \ldots, n_\sigma)$ which supports access, rank and select on the rows in time $\mathcal{O}(\lg \lg \sigma)$, and uses total space for data and index proportional to the entropy $H(n_1, \ldots, n_\sigma)$ of the distribution of those sizes (if the posting lists refer to the words of a text, this is also the zero-order word-based entropy of the text). This is achieved by encoding the string of labels encountered during a row-first traversal, writing a special symbol (e.g. \$) at each change of row. This improves the space of previously known data structures [2], and improves the time complexity of previous compression results [5].

Naturally, the next challenge ahead is to obtain a data structure using space $nH_k(s)+o(n)(H_k(s)+1)$ bits rather than $nH_k(s)+o(n) \lg \sigma$, while still supporting the queries access, rank, and select, in reasonable time. Note that Barbay et al. [3] achieve $nH_k(s) + o(n) \lg \sigma$ for such a structure: we have reduced the redundancy to $o(n)(H_k(s) + 1)$ for the case $k = 0$ and for self-indexes, but not for the basic problem in the general case where $k = o(\log_\sigma n)$.

Acknowledgments. Many thanks to Djamal Belazzougui for helpful comments on a draft of this paper.

References

1. A. Andersson. Sorting and searching revisited. In *Proc. 5th SWAT*, pages 185–197, 1996.
2. J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.
3. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th SODA*, pages 680–689, 2007.
4. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th STACS*, pages 111–122, 2009.
5. D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. DCC*, pages 342–351, 2002.
6. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
7. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, pages 176–187, 2008.
8. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.
9. T. Gagie, G. Navarro, and Y. Nekrich. Fast and compact prefix codes. In *Proc. 36th SOFSEM*, LNCS 5901, pages 419–427, 2010.
10. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
11. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
12. J. B. Hreinsson, M. Krøyer, and R. Pagh. Storing a compressed function with constant time access. In *Proc. 17th ESA*, pages 730–741, 2009.
13. C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Information and Computation*, 112(1):37–50, 1994.
14. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
15. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
16. Kurt Mehlhorn. Sorting presorted files. In *Proc. 4th GI-Conference on Theoretical Computer Science*, LNCS 67, pages 199–212, 1979.
17. J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, pages 345–356, 2003.
18. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
19. N. Rahman and R. Raman. Rank and select operations on binary strings. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.
20. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
21. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
22. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.