Journal of Discrete Algorithms 43 (2017) 54–71

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Grammar compressed sequences with rank/select support ☆

Alberto Ordóñez [a], Gonzalo Navarro [b,*], Nieves R. Brisaboa [a]

[a] *Database Laboratory, Universidade da Coruña, Spain*
[b] *Department of Computer Science, University of Chile, Chile*

ARTICLE INFO

Article history:
Available online 14 October 2016

Keywords:
Grammar compression
Repetitive sequences
Text indexing

ABSTRACT

Sequence representations supporting not only direct access to their symbols, but also rank/select operations, are a fundamental building block in many compressed data structures. Several recent applications need to represent highly repetitive sequences, and classical statistical compression proves ineffective. We introduce, instead, grammar-based representations for repetitive sequences, which use up to 6% of the space needed by statistically compressed representations, and support direct access and rank/select operations within tens of microseconds. We demonstrate the impact of our structures in text indexing applications.

1. Introduction

Given a sequence $S[1,n]$ over an alphabet $\Sigma = [1,\sigma]$, an intensively studied problem in recent years has been how to represent S space-efficiently while supporting these three operations:

- $\mathtt{access}(S,i)$, which returns $S[i]$, with $1 \le i \le n$.
- $\mathtt{rank}_b(S,i)$, which returns number of occurrences of $b \in \Sigma$ in $S[1,i]$, with $0 \le i \le n$.
- $\mathtt{select}_b(S,i)$, which returns the position of the i-th occurrence of $b \in \Sigma$ in S, with $0 \le i \le \mathtt{rank}_b(S,n)$ and $\mathtt{select}_b(S,0) = 0$.

The data structures supporting these three operations will be called rsa structures (for rank, select, access). Their popularity owes to the wide number of applications in which they are particularly useful. For instance, we can simulate and improve the functionalities of *inverted indices* [6,54] by concatenating the posting lists and representing the resulting sequence with an rsa structure [10,4,3]. We can also build full-text *self-indices* like the FM-Index [23,24] on an rsa-capable representation of the Burrows–Wheeler Transform [16] of the text. Several other applications of rsa structures have been studied, for example document listing in sequence collections [42], XML/XPath systems [2], positional inverted indices [5], graphs [20], binary relations [8], tries and labeled trees [22].

In many applications, keeping the data in main memory is essential for high performance. Therefore, one aims at using little space for an rsa structure. The best known such sequence representations [28,21,9,29,7,13] use *statistical* compression,

☆ An early partial version of this paper appeared in *Proc. SPIRE 2014* [46]. Funded in part by European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 690941 (project BIRDS), FONDECYT Grant 1-140796, Chile, CDTI EXP 000645663/ITC-20133062 (CDTI, MEC, and AGI), Xunta de Galicia (PGE and FEDER) ref. GRC2013/053, and by MICINN (PGE and FEDER) refs. TIN2009-14560-C03-02, TIN2010-21246-C02-01, TIN2013-46238-C4-3-R and TIN2013-47090-C3-3-P and AP2010-6038 (FPU Program).

* Corresponding author.
E-mail addresses: alberto.ordonez@udc.es (A. Ordóñez), gnavarro@dcc.uchile.cl (G. Navarro), brisaboa@udc.es (N.R. Brisaboa).

http://dx.doi.org/10.1016/j.jda.2016.10.001
1570-8667/© 2016 Elsevier B.V. All rights reserved.

which exploits the frequencies of the symbols in S. The smallest ones achieve $nH_k(S) + o(n \log \sigma)$ bits for any $k = o(\log_\sigma n)$. The measure $H_k(S)$ is the minimum bit-per-symbol rate achieved by a statistical compressor based on the frequencies of each symbol conditioned to the k symbols preceding it. Statistically-compressed representations can, on a RAM machine with word size w, answer `access` in $O(1)$ time and `select` in any time in $\omega(1)$, or vice versa, and `rank` in time $O(\log \log_w \sigma)$. These times match lower bounds [13].

Although statistical compression is appropriate in many contexts, it is unsuitable in various other domains. This is the case of an increasing number of applications that deal with highly repetitive sequences: software repositories, versioned document collections, genome datasets of individuals of the same species, and so on, which contain many near-copies of the same source code, document, or genome [41]. In this scenario, statistical compression does not take proper advantage of the repetitiveness [33]: for $k = 0$, the entropy does not change if we concatenate many copies of the same sequence, and for $k > 0$ the situation is similar, as in most cases the near-copies are much farther apart than $k = o(\log_\sigma n)$ positions.

Instead, grammar [32,17] and Lempel–Ziv [35,55] compressors are very efficient to represent repetitive sequences, and thus could be excellent candidates for applications that require `rsa` functionality on them. However, even supporting `access` is difficult on those formats. The fastest schemes take $O(\log n)$ time, using either $O(g \log n)$ bits of space on a grammar of size g [14], or more than $O(z \log n)$ bits on a Lempel–Ziv parsing of z phrases [25]. This time is essentially optimal [52]. Therefore, supporting `access` is intrinsically harder than with statistically compressed sequence representations.

The support for `rank` and `select` is even more rare on repetitive sequences. Only for bitmaps (i.e., bit sequences) compressed with balanced grammars (whose grammar tree is of height $O(\log n)$), the $O(g \log n)$ bits and $O(\log n)$ time obtained for `access` on grammar-compressed strings is extended to all `rsa` queries [47]. However, for larger σ, the space becomes $O(g\sigma \log n)$ bits and the time raises to $O(\log \sigma \log n)$.

In this paper we propose two new solutions for `rsa` queries over grammar compressed sequences, and compare them with various alternatives on a number of real-life repetitive sequences. Our first structure, tailored to sequences over small alphabets, extends and improves the current representation of bitmaps [47]. On a balanced grammar of size g, it obtains $O(\log n)$ time for all the `rsa` operations with $O(g\sigma \log n)$ bits of space, using in practice similar space while being much faster than previous work [47]. We dub this solution `GCC` (Grammar Compression with Counters). It can be used, for example, on sequences of XML tags or DNA.

Our second structure combines `GCC` with alphabet partitioning [7] and is aimed at sequences with larger alphabets. Alphabet partitioning splits the sequence S into subsequences over smaller alphabets. If these alphabets are small enough, we apply `GCC` on them. On the subsequences with larger alphabets, we use representations similar to previous work [47]. The resulting time/space guarantees are as in previous work [47], but the scheme is much faster in practice while using about the same space. Recent work [11] (see next section) shows that time complexities of `GCC` are essentially optimal.

While up to an order of magnitude faster than the alternative grammar-compressed representation, our solutions are still an order of magnitude slower than statistically compressed representations, but they are also an order of magnitude smaller on repetitive sequences. We also evaluate our data structures on two applications: full-text self-indices and XML collections.

This paper is organized as follows: Section 2 describes the basic concepts and previous work; Section 3 explains our `rsa` data structures for small alphabets; Section 4 presents our solution for `rsa` on large alphabets; Section 5 experimentally evaluates our proposals; Section 6 explores their performance in several applications; and finally Section 7 gives conclusions and future research lines.

2. Basic concepts and related work

2.1. Statistical compression measures

Given a sequence $S[1,n]$ over $\Sigma = [1,\sigma]$, let $0 \le p_i \le 1$ be the relative frequency of symbol i in S. The *zero-order empirical entropy* of S is defined as[1]

$$H_0(S) = \sum_{i=1}^{\sigma} p_i \lg \frac{1}{p_i},$$

and it is a lower bound of the bit-per-symbol rate achievable by a compressor that encodes i only considering its frequency p_i. A richer model considers the frequency of each symbol within the context of k symbols preceding it. This leads to the *k-order empirical entropy* measure,

$$H_k(S) = \sum_{C \in \Sigma^k} \frac{|S_C|}{n} H_0(S_C),$$

where S_C is the string formed by collecting the symbols that follow each occurrence of the context C in S. It holds $H_k(S) \le H_{k-1}(S) \le H_0(S) \le \lg \sigma$ for any $k \ge 1$.

[1] We use lg to denote the logarithm in base 2.

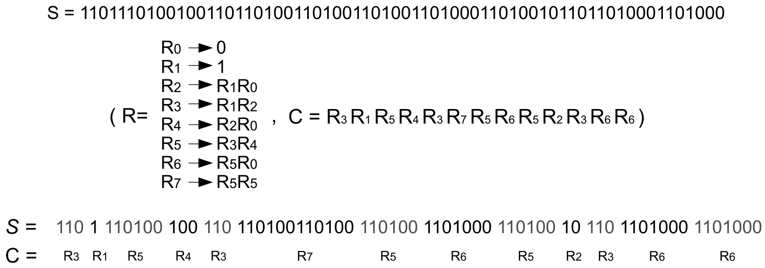

S = 1101111010010011011010011010011010011010001101001011011010001101000
(R=
R0 → 0
R1 → 1
R2 → R1R0
R3 → R1R2
R4 → R2R0
R5 → R3R4
R6 → R5R0
R7 → R5R5
, C = R3 R1 R5 R4 R3 R7 R5 R6 R5 R2 R3 R6 R6)
S = 110 1 110100 100 110 110100110100 110100 1101000 110100 10 110 1101000 1101000
C = R3 R1 R5 R4 R3 R7 R5 R6 R5 R2 R3 R6 R6

Fig. 1. The data structures (R, C) are the result of executing the RePair algorithm on the input sequence S with $\sigma = 2$.

2.2. Grammar compression

Grammar-compressing a sequence S means finding a context-free grammar that generates (only) S. Finding the smallest such grammar is NP-complete [17], but heuristics like RePair [34] run in linear time and find very good grammars.

RePair finds the most frequent pair of symbols ab in S, adds a rule $X \to ab$ to a dictionary R, and replaces each occurrence of ab in S by X.[2] This process is repeated (X can be involved in future pairs) until the most frequent pair appears only once. The result is a pair (R, C), where the dictionary R contains $r = |R|$ rules and C, of length $c = |C|$, is the final result of S after all the replacements are done. Note that C is drawn from the alphabet of terminals and nonterminals. For simplicity we assume that the first σ rules generate the σ terminal symbols, so that r counts terminals plus nonterminals. Thus, the total output size of (R, C) is $(2(r - \sigma) + c) \lg r$ bits. Fig. 1 shows an example of applying RePair on a binary input S.

By using the technique of Tabei et al. [51], it is possible to represent the dictionary in $r \lg r + O(r)$ bits, reducing the total space to $(r + c) \lg r + O(r)$ bits. However, our experiments in the conference version [46] show that the resulting access method is much slower, so in this paper we use a plain representation of the rules.

Finally, it is possible to force the grammar to be *balanced*, that is, with the grammar tree being of height $O(\log n)$ [50]. We use instead a simple heuristic that modifies RePair so that the newly created pairs are added at the end of the list of the pairs with the same frequency. This is sufficient to make the grammars balanced in all the cases we tested.

2.3. Variable-length encoding of integers

In several cases one must encode a sequence of numbers, most of which are small. A variable-length integer encoding aims to use fewer bits when encoding a smaller number. For example, γ-codes [54] encode a number $x > 0$ using $2 \lg x$ bits, by writing its length $|x|$ in unary followed by x itself in binary (devoided of its highest 1). For larger numbers, δ-codes [54] encode $|x|$ using γ-codes instead of unary codes, and thus require $\lg x + O(\lg \lg x)$ bits to encode x.

For even larger numbers, the so-called Variable Byte [53] (VByte) representation is interesting, as it offers fast decoding by accessing byte-aligned data. The idea is to split each integer into 7-bit chunks and encode each chunk in a byte. The highest bit of the byte is used to indicate whether the number continues in the next byte or not. Then encoding x requires at most $(8/7) \lg x + 7$ bits.

2.4. Statistically compressed bitmaps

Several classical solutions represent a binary sequence $B[1, n]$ with `rsa` support. Clark and Munro [18,40] (`CM`) use $o(n)$ bits on top of B and answer the `rsa` queries in $O(1)$ time.

Raman et al. [48] (`RRR`) also support the operations in $O(1)$ time, but they compress B statistically, to $nH_0(B) + o(n)$ bits. This solution is well suited for scenarios where the distribution of 0/1 is skewed. However, it is not adequate to exploit repetitiveness in the bitmaps.

If the bitmaps are very sparse, the $o(n)$-bits term of the previous solution may be dominant. In this case, it is better to encode the differences between consecutive positions of the 1s with an encoding that favors small numbers, like δ-codes, and add absolute pointers to regularly sampled positions. This encoding uses $nH_0(B) + o(nH_0(B))$ bits and handles `rsa` operations in $O(\log n)$ time. This folklore idea, which we call `DELTA`, has been used repeatedly; see e.g. [33].

2.5. Grammar-compressed bitmaps

The only bitmap representation we are aware of that exploits repetitiveness in the bitmaps is due to Navarro et al. [47] (`RPB`). They RePair-compress B with a balanced grammar and enhance the output (R, C) with extra information to answer `rsa` queries. For each rule $X \in R$, let $exp(X)$ be the string of terminals X expands to. Then they store two numbers per nonterminal X:

[2] Note that, if $a = b$, we can only replace every other occurrence of aa in a sequence of as.

- $\ell(X) = |exp(X)|$,
- $z(X) = \mathtt{rank}_0(exp(X), \ell(X))$ (the number of 0s contained in $exp(X)$).

Note that these values can be recursively computed: If $X \to YZ$, then $exp(X) = exp(Y)exp(Z)$; $\ell(X) = \ell(Y) + \ell(Z)$, with $\ell(0) = \ell(1) = 1$; and $z(X) = z(Y) + z(Z)$, with $z(0) = 1$ and $z(1) = 0$.

To save space, they store $\ell(\cdot)$ and $z(\cdot)$ only for a subset of nonterminals, and compute the others recursively by partially expanding the nonterminal. Given a parameter δ, they guarantee that, to compute any $\ell(X)$ or $z(X)$, we have to expand at most 2δ rules. The sampled rules are marked in a bitmap $B_d[1,r]$ and the sampled values are stored in two vectors, S_ℓ and S_z, of length $\mathtt{rank}_1(B_d, r)$. To obtain $\ell(X)$ we check whether $B_d[X] = 1$. If so, then $\ell(X) = S_\ell[\mathtt{rank}_1(B_d, X)]$. Otherwise $\ell(X)$ is obtained recursively as $\ell(Y) + \ell(Z)$. The process for $z(X)$ is analogous.

Finally, every sth position of B is sampled, for a parameter s. Note that $B = exp(C[1])\,exp(C[2])\ldots exp(C[c])$, where the position where each $exp(C[p])$ starts in B is $L(p) = 1 + \sum_{k=1}^{p-1} \ell(C[k])$. Then, the sampling array $S_n[0, n/s]$ stores a tuple (p, o, rnk) at $S_n[k]$, where $exp(C[p])$ contains $B[k \cdot s]$, that is, $p = \max\{j, L(j) \le k \cdot s\}$. The other components are $o = k \cdot s - L(p)$, that is, the offset of $B[k \cdot s]$ within $exp(C[p])$; and $rnk = \mathtt{rank}_0(B, L(p) - 1)$ is the number of 0s before $exp(C[p])$ starts. We also set $S_n[0] = (0, 0, 0)$.

To answer $\mathtt{rank}_0(B, i)$, let $S_n[\lfloor i/s \rfloor] = (p, o, rnk)$ and set $l = s \cdot \lfloor i/s \rfloor - o$. Then we move forward from $C[p]$, updating $l = l + \ell(C[p])$, $rnk = rnk + z(C[p])$, and $p = p + 1$, as long as $l + \ell(C[p]) \le i$. When $l \le i < l + \ell(C[p])$, we have reached the rule $C[p] = X \to YZ$ whose expansion contains $B[i]$. Then, we recursively traverse X as follows. If $l + \ell(Y) > i$, we recursively traverse Y. Otherwise we update $l = l + \ell(Y)$ and $rnk = rnk + z(Y)$, and recursively traverse Z. This is repeated until $l = i$ and we reach a terminal symbol in the grammar. Then we return rnk. Obviously, we can also compute $\mathtt{rank}_1(B, i) = i - \mathtt{rank}_0(B, i)$. Supporting $access(B, i)$ is completely equivalent, but instead of maintaining rnk we just return the terminal symbol we reach when $l = i$.

To answer $\mathtt{select}_0(B, j)$, we binary search S_n to find $S_n[i] = (p, o, rnk)$ and $S_n[i+1] = (p', o', rnk')$ such that $rnk < j \le rnk'$. Then we proceed as for $\mathtt{rank}_0$, but updating l and rnk as long as $rnk + z(C[p]) \le j$, and then traversing by going left (to Y) when $rnk + z(Y) > j$, and going right (to Z) otherwise. At the end, we return l. The process for $\mathtt{select}_1(B, j)$ is analogous (note that X contains $\ell(X) - z(X)$ 1s).

On a balanced grammar, a rule is traversed in $O(\log n)$ time. The time to iterate over C between samples is $O(s)$. Therefore, if we set $s = \Theta(\log n)$, the total time for $\mathtt{rsa}$ queries is $O(s + \log n) = O(\log n)$ and the total space is $O(r \log n + (n/s)\log n) + c \lg r = O((r + c)\log n + n)$ bits.[3] The time is multiplied by δ if we use sampling to avoid storing all the information for all the rules.

2.6. Wavelet trees

The wavelet tree [28,43] ($\mathtt{WT}$) is a complete balanced binary tree that represents a sequence $S[1, n]$ over alphabet $\Sigma = [1, \sigma]$. Assume we assign a plain encoding of $\lceil \lg \sigma \rceil$ bits to the symbols. Let us call $S[i]\langle j \rangle$ the jth most significant bit of the code associated with $S[i]$. The $\mathtt{WT}$ construction proceeds as follows: At the root node it splits the alphabet Σ into two halves, Σ_1 and Σ_2. A symbol belongs to Σ_1 iff $S[i]\langle 1 \rangle = 0$, and to Σ_2 otherwise. We store that information in a bitmap $B[1, n]$ associated with the node, being $B[i] = 0$ iff $S[i] \in \Sigma_1$ and 1 otherwise. The left child of the root will then represent the subsequence of S containing symbols in Σ_1, while the right node will do the same with Σ_2. The process is then recursively repeated in both children until the alphabet of the current node is unary. The height the $\mathtt{WT}$ is $\lceil \lg \sigma \rceil$.

The only information we need to store from a $\mathtt{WT}$ are the bitmaps stored in the internal tree nodes and the tree pointers. The total space for the sequences is $n\lceil \lg \sigma \rceil$ bits, while for tree pointers we use $O(\sigma \log n)$ bits. Thus, the total space becomes $n \lg \sigma + O(n + \sigma \log n)$ bits.

Although we will focus on the binary case, we can generalize the concept of $\mathtt{WT}$ to the multi-ary case: Instead of recursively dividing the alphabet into two halves, we can split it into 2^b disjoint sets. This is known as Multi-ary $\mathtt{WT}$ or $\mathtt{MWT}$. Now the internal $\mathtt{MWT}$ nodes store sequences drawn over alphabet $[1, 2^b]$ instead of bitmaps, and the height is reduced to $\lceil (\log \sigma)/b \rceil$.

Algorithm 1 shows how $\mathtt{rsa}$ queries on S are built on $\mathtt{rsa}$ queries on the bitmaps or sequences of the $\mathtt{MWT}$ of S. A key aspect in $\mathtt{WT}$'s performance is how we represent those bitmaps or sequences. In the binary case (Section 2.4), if we use $\mathtt{CM}$ for bitmaps, the total space is $n \lg \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits and $\mathtt{rsa}$ times are $O(\log \sigma)$. By using $\mathtt{RRR}$, the time complexity is retained (although its times are higher in practice) but the space shrinks to $nH_0(S) + o(n \log \sigma) + O(\sigma \log n)$ bits. Zero-order compression is also obtained by using a Huffman [31] encoding for the symbols and giving the $\mathtt{WT}$ the shape of the Huffman tree: using $\mathtt{CM}$ for the bitmaps results in $n(H_0(S) + 1)(1 + o(1)) + O(\sigma \log n)$ bits, whereas using $\mathtt{RRR}$ for the bitmaps the space becomes $nH_0(S)(1 + o(1)) + O(\sigma \log n)$ bits [10]. This solution is called Huffman-shaped $\mathtt{WT}$ ($\mathtt{WTH}$). The main advantage of using a $\mathtt{WTH}$ is that, if queries follow the same statistical distribution of symbols, then the average query time for any $\mathtt{rsa}$ query becomes $O(1 + H_0(S))$ instead of $O(\log \sigma)$ [10]. A Huffman-shaped multi-ary wavelet tree will be called $\mathtt{MWTH}$. For any $2^b = o(\log n / \log \log n)$, the $\mathtt{MWTH}$ retains the same space complexities of a $\mathtt{WTH}$, whereas the worst-case and average query time are divided by b [10].

[3] We can obtain $O((r + c)\log n)$ bits and the same time by sampling C instead of B, as we show later.

Algorithm 1 Standard MWT algorithms on a sequence S. The sequence associated with node v is S_v and its ith child is v_i. For access(S, i) we return **acc**$(root, i, 0)$, where *root* is the MWT root; rank$_a(S, i)$ returns **rnk**$(root, a, i, \lceil(\log\sigma)/b\rceil)$; and select$_a(S, j)$ returns **sel**$(root, a, j, \lceil(\log\sigma)/b\rceil)$. Function *leaf*$(v)$ returns whether node v is a leaf, and $chunk(a, b, \ell) = (a \gg (\ell-1)b)$ & $((1 \ll b) - 1)$ takes the ℓth chunk of b most significant bits from a.

```
acc($v, i, c$)
  if $leaf(v)$ then
    return $c$
  $c \leftarrow (c \ll b) \mid S_v[i]$
  $i \leftarrow \mathtt{rank}_{S_v[i]}(S_v, i)$
  return acc($v_{S_v[i]}, i, c$)
```

```
rnk($v, a, i, \ell$)
  if $leaf(v)$ then
    return $i$
  $c \leftarrow chunk(a, b, \ell)$
  $i \leftarrow \mathtt{rank}_c(S_v, i)$
  return rnk($v_c, a, i, \ell - 1$)
```

```
sel($v, a, j, \ell$)
  if $leaf(v)$ then
    return $j$
  $c \leftarrow chunk(a, b, \ell)$
  $j \leftarrow$ sel($v_c, a, j, \ell - 1$)
  return $\mathtt{select}_c(S_v, j)$
```

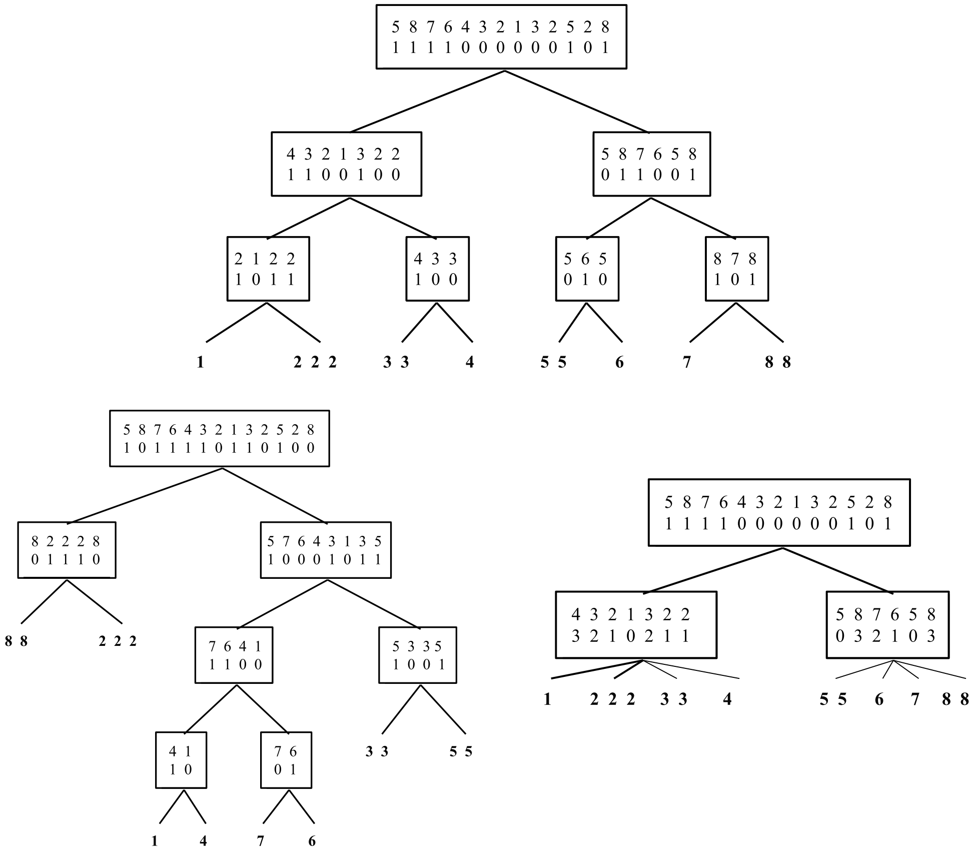


Fig. 2. Wavelet tree representations of sequence $S = 5876432132528$. On the top a WT, on the bottom left WTH, and on the bottom right a MWT with $2^b = 4$ (the first level can only have arity 2).

Fig. 2 exemplifies all these wavelet tree variants.

2.7. Wavelet matrix

If σ is close to n, the $O(\sigma \log n)$ bits to store the tree pointers in a WT will become dominant. To skip this term, the *levelwise* WT [37] concatenates all the bitmaps at the same depth and simulates the tree pointers with rsa operations. This variant obtains the same space of the WT or MWT but without the $O(\sigma \log n)$ term. The time performance is asymptotically the same, but it is slower in practice because pointers are simulated. More recently, the *wavelet matrix* (WM) [21] was proposed, which speeds up the levelwise WT by reshuffling the bits at each level in a different way so that the tree pointers can be simulated with fewer rsa operations. Assume we start with $S_l = S$ at level $l = 1$; then the wavelet matrix is built as follows:

1. Build a single bitmap $B_l[1, n]$ where $B_l[i] = S_l[i]\langle l \rangle$;
2. Compute $z_l = \mathtt{rank}_0(B_l, n)$;
3. Build sequence S_{l+1} such that, for $k \le z_l$, $S_{l+1}[k] = S_l[\mathtt{select}_0(B_l, k)]$, and for $k > z_l$, $S_{l+1}[k] = S_l[\mathtt{select}_1(B_l, k - z_l)]$;
4. Repeat the process until $l = \lceil \log \sigma \rceil$.

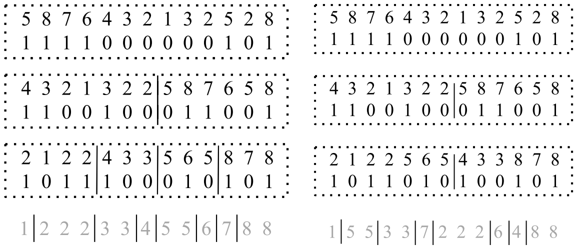


Fig. 3. Wavelet tree/matrix representations of sequence $S = 5876432132528$. On the left a levelwise WT, and on the right a WM.

Algorithm 2 Standard WM algorithms on a sequence S. The bitmap at level l is denoted by B_l and $z_l = \mathtt{rank}_0(B_l, n)$. For $\mathtt{access}(S,i)$ we return $\mathbf{acc}(1,i,0)$; $\mathtt{rank}_a(S,i)$ returns $\mathbf{rnk}(1,a,i,0)$; and $\mathtt{select}_a(S,j)$ returns $\mathbf{sel}(1,a,j,0)$.

```
acc(l, i, c)
    if $l = \lceil \lg \sigma \rceil$ then
        return $c$
    $c \leftarrow (c \ll 1) \mid B_l[i]$
    $i \leftarrow \mathtt{rank}_{B_l[i]}(B_l, i) + z_l \cdot B_l[i]$
    return acc($l+1, i, c$)

rnk(l, a, i, p)
    if $l = \lceil \lg \sigma \rceil$ then
        return $i - p$
    $i \leftarrow \mathtt{rank}_{a\langle l \rangle}(B_l, i) + z_l \cdot a\langle l \rangle$
    $p \leftarrow \mathtt{rank}_{a\langle l \rangle}(B_l, p) + z_l \cdot a\langle l \rangle$
    return rnk($l+1, a, i, p$)

sel(l, a, j, p)
    if $l = \lceil \lg \sigma \rceil$ then
        return $p + j$
    $p \leftarrow \mathtt{rank}_{a\langle l \rangle}(B_l, p) + z_l \cdot a\langle l \rangle$
    $j \leftarrow$ sel($l+1, a, j, p$)
    return $\mathtt{select}_{a\langle l \rangle}(B_l, j - z_l \cdot a\langle l \rangle)$
```

Algorithm 3 Alphabet partition algorithms for access, rank, and select.

```
access(S, i)
    $j \leftarrow K[i]$
    $v \leftarrow S_j[\mathtt{rank}_j(K, i)]$
    return $\mathtt{select}_j(M, v)$

rank_a(S, i)
    $j \leftarrow M[a]$
    $v \leftarrow \mathtt{rank}_j(M, a)$
    $r \leftarrow \mathtt{rank}_j(K, i)$
    return $\mathtt{rank}_v(S_j, r)$

select_a(S, i)
    $j \leftarrow M[a]$
    $v \leftarrow \mathtt{rank}_j(M, a)$
    $s \leftarrow \mathtt{select}_v(S_j, i)$
    return $\mathtt{select}_j(K, s)$
```

This reshuffling of the bits of $S[i]\langle j \rangle$, akin to radix sorting the symbols of S, uses $n\lceil \lg \sigma \rceil$ bits in total (plus $\lg n \lg \sigma$ for the values z_l). Therefore, the total space of the WM is $n \lg \sigma + o(n \log \sigma)$. Fig. 3 exemplifies the levelwise WT and the WM. As in the case of the WT, this space can be further reduced to $nH_0(S) + o(n \log \sigma)$ if we use RRR (Section 2.4) to compress the bitmaps, or to $n(H_0(S)+1)(1+o(1)) + O(\sigma \log n)$ by using plain bitmaps (CM) and giving Huffman shape to the WM [21] (Section 2.6). The latter is called a WMH. We can also convert a MWT into a multi-ary WM (MWM) by increasing the number of counters z_l at each level: if 2^b is the arity, we need $2^b - 1$ counters z_l per level.

Algorithm 2 shows how the algorithms are implemented on a WM. Although better than the levelwise WT, it still requires more operations on the bitmaps than the WT.

2.8. Alphabet partitioning

An alternative solution for rsa queries over large alphabets is *Alphabet Partitioning* (AP) [7], which obtains $nH_0(S) + o(n(H_0(S)+1))$ bits and supports rsa operations in $O(\log\log\sigma)$ time. The main idea is to partition Σ into several subalphabets Σ_j, and S into the corresponding subsequences S_j, each defined over Σ_j (see Fig. 4). The practical variant sorts the σ symbols by decreasing frequency and then splits that sequence into disjoint subsets, or subalphabets, of increasingly exponential size, so that Σ_j contains the 2^{j-1}th to the $(2^j - 1)$th most frequent symbols. The information on the partitioning is kept in a sequence M, where $M[i] = j$ iff $i \in \Sigma_j$. A new string $K[1,n]$ indicates the subalphabet each symbol of S belongs to: $K[i] = M[S[i]]$. Analogously to wavelet trees, the sequences S_j are defined as $S_j[i] = \mathtt{rank}_j(M, S[\mathtt{select}_j(K,i)])$. Note that the number of subalphabets is at most $\lfloor \lg \sigma \rfloor + 1$, and this is the alphabet size of M and K. Therefore, a binary WT representation of M and K handles rsa operations in time $O(\log\log\sigma)$. Further, the symbols in each Σ_j are of roughly the same frequency, thus a fast compact (but not compressed) representation of S_j (GMR [26]) yields $O(\log\log\sigma)$ time and retains the statistical compression of S [7].

Algorithm 3 shows how the rsa operations on S translate into rsa operations on M, K, and on some subsequence S_j, thus obtaining $O(\log\log\sigma)$ times. In practice, the sequences S_j with the smallest alphabets are better integrated directly into the WT of K.

There are other representations that improve upon this solution in theory, but are unlikely to do better in practice. For example, it is possible to retain similar time complexities while reducing the space to $nH_k(S) + o(n \log \sigma)$ bits, for any $k = o(\log_\sigma n)$ [9,29]. It is also possible, within zero-order entropy space, to support access and select in $O(1)$ and any $\omega(1)$ time, or vice versa, and rank in time $O(\log\log_w \sigma)$, on a RAM machine with word size w, which matches lower bounds [13].

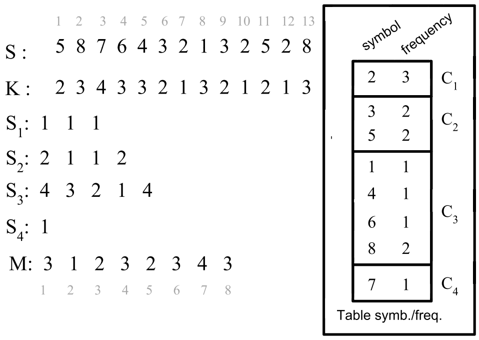


Fig. 4. Alphabet partitioning example.

2.9. RePair compressed WT

As far as we know, what we will call WTRP [47] is the only solution to support rsa on grammar-compressed sequences. The structure is a levelwise WT where each bitmap B_l is compressed with RPB (Section 2.5). The rationale is that the repetitiveness of S is reflected in the bitmaps of the WT.

However, since the WT construction splits the alphabet at each level, those repetitions are cut into shorter ones at each new level, and become blurred after some depth. Therefore, the bitmaps of the first few WT levels are likely to be compressible with RePair, while the remaining ones are not. The authors [47] use at each level l the technique to represent B_l that yields the least space, RPB, RRR, or CM (Sections 2.4 and 2.5). In case of a highly compressible sequence, the space can be drastically reduced, but the search performance degrades by one or more orders of magnitude compared to using CM or RRR: If all the levels use RPB, the rsa time becomes $O(\log\sigma \log n)$.

On the other hand, as repetitiveness is destroyed at deeper levels, the total space is far from that of a plain RePair compression of S. A worst-case analysis, albeit pessimistic, can be made as follows: Each node stores a subsequence of S, whose alphabet is mapped onto a binary one (or of size r in an r-ary wavelet tree). We could then take the same grammar that compresses S for each node, remove all the terminal symbols not represented in that node, and map the others onto $\{0, 1\}$ or $[1, r]$. This is not the best grammar for that node, but it is correct and at most of the same size g of the original one. Therefore, each node can be grammar-compressed to at most $O(g \log n)$ bits, and summed over all the wavelet tree nodes, this yields $O(g\sigma \log n)$. Therefore, the size grows at most linearly with σ.

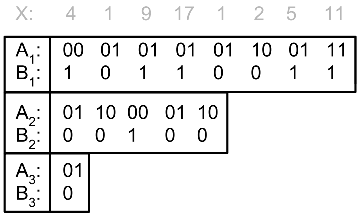
2.10. Other grammar-compressed rsa solutions

Let a grammar compressor produce a grammar of size g with r nonterminals for $S[1, n]$. Thus S can be represented in $g \lg(r + \sigma)$ bits. Bille et al. [14] show how to represent S using $O(g \log n)$ bits so that access(S, i) is answered in $O(\log n)$ time. This time is essentially optimal [52]: any structure using $g^{O(1)} \log n$ bits requires $\Omega(\log^{1-\epsilon} n/ \log g)$ time for access, for any $\epsilon > 0$. If S is not very compressible and $g = \Omega(n^{\alpha})$ for some constant α, then the time is $\Omega(\log n/ \log\log n)$ for any structure using $O(n \,\mathrm{polylog}\, n)$ bits.

As said, we are not aware of any previous rsa structure building on grammar compression apart from WTRP [47], which handles queries in $O(\log\sigma \log n)$ time and uses $O(g\sigma \log n)$ bits. Our simplest variant, GCC, obtains $O(\log n)$ time for the three rsa operations within $O(g\sigma \log n)$ bits. For larger alphabets, we can increase the time to $O(\log\sigma \log n)$ and keep the worst-case space in $O(g\sigma \log n)$ bits (yet in practice the solution takes less space and time than WTRP, and less space than GCC). Alternatively, we can retain the $O(\log n)$ time but lose the space guarantee.

After the publication of the conference version of our article [46], Belazzougui et al. [11] gave more theoretical support to our results. They obtained our same $O(\log n)$ time for rsa operations with $O(g\sigma \log n)$ bits on arbitrary grammars of size g (not only balanced ones). They also show how to obtain $O(\log n/ \log\log n)$ time using $O(g\sigma \log(n/g) \log^{1+\epsilon} n)$ bits, for any constant $\epsilon > 0$. Most importantly, they prove that it is unlikely that these times for rank and select can be significantly improved, since long-standing reachability problems on graphs would then be improved as well. This shows that the time complexity of their (and our) solutions are essentially the best one can expect.

Lempel–Ziv [35,55] compression is able to outperform grammar compression [32,17], because the number of phrases it generates is never larger than the size g of the best possible grammar. However, its support for rsa queries is even more difficult. Let z be the number of phrases into which a Lempel–Ziv parser factors S. Then a Lempel–Ziv compressor can represent S in $z(\lg n + \lg\sigma)$ bits. We are not aware of any scheme supporting $O(\log n)$ time access within $O(z \log n)$ bits. Gagie et al. [25] do achieve this time, but they use $O(z \log n \log(n/z))$ bits, which is superlinear in the compressed size of S. A more recent work [12] supports access and rank in time $O(\log(n/z))$ and select in time $O(\log(n/z) \log\log n)$. The lower bound [52] also holds for this compression, replacing g by z.



X:	4	1	9	17	1	2	5	11
A_1:	00	01	01	01	01	10	01	11
B_1:	1	0	1	1	0	0	1	1
A_2:	01	10	00	01	10			
B_2:	0	0	1	0	0			
A_3:	01							
B_3:	0							

Fig. 5. Example of a DAC for the sequence $X = 4, 1, 9, 17, 1, 2, 5, 11$ and $b = 2$.

2.11. Directly Addressable Codes

A Directly Addressable Code [15] (DAC) is a variable-length encoding for integers that supports direct access operations (`access`) efficiently, but not `rank` and `select`. Assume we have to encode a sequence $X = x_1 \ldots x_n$ of integers and are given a chunk size b. Then we divide each $x_i = X[i]$ into $j = \lceil(\lfloor \lg x_i \rfloor + 1)/b\rceil$ chunks, from least to most significant. At the most significant position of each chunk we will prepend a bit 0 if that chunk is the last one, and a 1 otherwise. Therefore, the number x_i is encoded as

$$b_{1,i}a_{1,i}b_{2,i}a_{2,i}\ldots b_{k,i}a_{k,i},$$

where $b_{j,i}$ is the bit prepended to the chunk $a_{j,i} = x_i\langle jb, (j-1)b+1\rangle$. Note the similarity with the VByte codes of Section 2.3.

Instead of concatenating the encoding of x_{i+1} after that of x_i, however, we build a multi-layer data structure. At each layer $l \geq 1$, we concatenate the lth chunks of all the numbers that have one, and do the same with the bits prepended to each chunk. For instance, for layer $l = 1$ we obtain a binary sequence B_1 and a sequence A_1 as follows:

$$B_1 = b_{1,1}b_{1,2}\ldots b_{1,n},$$
$$A_1 = a_{1,1}a_{1,2}\ldots a_{1,n}.$$

The next layer is then built by concatenating the second chunk of each number that has one, and the process is repeated for M layers, where $M = \lceil(\lfloor \lg(\max x_i)\rfloor + 1)/b\rceil$. Fig. 5 shows an example DAC over a sequence X using $b = 2$.

To provide efficient direct access, we preprocess each sequence of prepended bits (B_i) to support `rank` and `access` queries in $O(1)$ time. Thus, we access $X[i]$ as follows. We start by setting $i_1 = i$, and reading $A_1[i_1] = a_{1,i_1}$. We set $res = A_1[i_1]$ and if $B_1[i_1] = 0$ we are done because this chunk is the last of x_i. If, instead, $B_1[i_1] = 1$, x_i continues in the next layer. To compute the position of the next chunk in the next layer we set $i_2 = \mathtt{rank}_1(B_1, i)$. In the second layer we concatenate $A_2[i_2]$ with the current result: $res = A_2[i_2]A_1[i_1]$ and then check $B_2[i_2]$, repeating the process until we get a $B_k[i_k] = 0$. Then the time to extract an element of X when represented with a DAC is worst-case $O(M)$.

It is possible to define a different b value for each level, and to choose them so as to optimize the total space used, even with a restriction on M [15].

3. Efficient `rsa` for sequences on small alphabets

Our first proposal, dubbed GCC *(Grammar Compression with Counters)* is aimed at handling `rsa` queries on grammar-compressed sequences with small alphabets. We first generalize the existing solution for bitmaps (RPB, Section 2.4), to sequences with $\sigma > 2$. We also introduce several enhancements regarding how we store the additional information to handle `rsa` queries. Finally, we propose two different sampling approaches that yield different space–time tradeoffs, both in theory and in practice.

Let (R, C) be the result of a balanced RePair grammar compression of S. We store $S_\ell[X] = \ell(X)$ for each grammar rule $X \in R$. In addition, we store an array of counters $S_a[X]$ for each symbol $a \in \Sigma$: $S_a[X] = \mathtt{rank}_a(exp(X), \ell(X))$ is the number of occurrences of a in $exp(X)$.

The input sequence S is also sampled according to the new scenario: each element (p, o, rnk) of $S_n[1, n/s]$ is now replaced by $(p, o, lrnk[1, \sigma])$, where $lrnk[a] = \mathtt{rank}_a(S, L(p) - 1)$ for all $a \in \Sigma$, s being the sampling period.

The extra space incurred by σ can be reduced by using the same δ-sampling of RPB, which increases the time by a factor δ. In this case we also use the bitmap $B_d[1, r]$ that marks which rules store counters. We further reduce the space by noting that many rules are short, and therefore the values in S_ℓ and S_a are usually small. We represent them using direct access codes (DACs, recall Section 2.11), which store variable-length numbers while retaining direct access to them. The o components of S_n are also represented with DACs for the same reason.

On the other hand, the p and $lrnk[1, \sigma]$ values are not small but are increasing. We reduce their space using a two-layer strategy: we sample S_n at regular intervals of length s'. We store $SS_n[j] = S_n[j \cdot s']$, and then represent the values of $S_n[i] = (p, o, lrnk[1, \sigma])$ in differential form, in array $S'_n[i] = (p', o, lrnk'[1, \sigma])$, where $p' = p - p^*$ and $lrnk'[a] = lrnk[a] - lrnk^*[a]$, with $SS_n[\lfloor i/s' \rfloor] = (p^*, o^*, lrnk^*[1, \sigma])$.

The total space for the p and $lrnk[1,\sigma]$ components is $O(\sigma((n/s)\log(s \cdot s') + (n/(s \cdot s'))\log n))$ bits, whereas the o components use $O((n/s)\log n)$ bits in the worst case. For example, if we use $s' = \lg n$ and $s = \log^{O(1)} n$ (a larger value would imply an excessively high query time), the space becomes $O(r\sigma \log n + (n/\log^{O(1)} n)(\sigma \log\log n + \log n)) + c\lg(\sigma + r)$ bits.

A further improvement is aimed to reduce the space on extremely repetitive sequences. In this scenario, many elements of S_n may contain the same values: if a rule covers a wide range of S, we store the same S_n values for many samples of S. Thus, we sample the vector C instead of sampling the whole sequence S. Instead of $(p, o, lrnk[1,\sigma])$ we store a tuple $(i, lrnk[1,\sigma])$, where i is the position where the sampled cell of C starts in S, and $lrnk$ is computed up to $i-1$. On the other hand, the two-layer scheme cannot be applied, because now the samples may cover arbitrarily long ranges of S.

The total space with this sampling then becomes $O(r\sigma \log n + \sigma(c/s)\log n) + c\lg(\sigma + r) = O((r + c)\sigma \log n)$ bits. This removes any linear dependency on n from the space formula. The size of the RePair grammar is $g = O(r + c)$, thus the space can be written as $O(g\sigma \log n)$ bits.

The rsa algorithms stay practically the same as for RPB; now we use the symbol counter of a for $\texttt{rank}_a$ and $\texttt{select}_a$. The resulting data structure performs rsa operations in time $O(s + \log n)$. In case C is sampled instead of S, there is an additional $O(\log c)$ time to binary search for the right sample. This is still within $O(s + \log n)$. If we choose $s = O(\log n)$, then the time is $O(\log n)$. The space is still $O(g\sigma \log n)$ if we sample C.

When σ is small and the sequence is repetitive, this data structure is very space- and time-efficient. It outperforms WTRP [47] (Section 2.9) in time: WTRP takes $O(\log\sigma \log n)$ time and our GCC uses $O(\log n)$. In terms of space, both use $O(g\sigma \log n)$ bits and perform similarly in practice. In the next section we develop a variant for large alphabets that uses much less space in practice, even if the worst-case guarantee it offers is still as bad as $O(g\sigma \log n)$ bits.

4. Efficient rsa for sequences on large alphabets

Our main idea for large alphabets is to use wavelet trees/matrices or alphabet partitioning (Sections 2.6 to 2.8) as a mechanism to cut Σ into smaller alphabets, which can then be handled with GCC. This is in the same line of WTRP (Section 2.9), which also partitions the alphabet. Our techniques deal better with the problem of loss of repetitiveness when the alphabet is partitioned.

The most immediate approach is to generalize WTRP to use a MWT, since now we can use GCC on small alphabets $[1, r]$ to represent the sequences S_v stored at the internal nodes of the MWT. Compared to a binary WT, a MWT takes more advantage of repetitiveness before splitting the alphabet, and reduces the time complexity from $O(\log\sigma \log n)$ to $O(\log_r \sigma \log n)$. The worst-case space is still $O(g\sigma \log n)$ bits. The use of a WM requires only $\log_r \sigma$ grammars, one per level, but still the guarantee on their total size is the same.

A less obvious way to use GCC is to combine it with AP (Section 2.8). Note that the string K is a projection of S, and therefore it retains all its repetitiveness. Further, it contains a small alphabet, of size $\lg\sigma$, and therefore we can use GCC on it. The resulting representation takes at most $O(g \log\sigma \log n)$ bits.

The other important sequences are the S_j, which have alphabets of size 2^{j-1}. For the smallest j, this is small enough to use GCC as well. For larger j, however, we must resort to other representations, like WTRP, GMR, or WT/WM, depending on how compressible they are.

An interesting fact of AP is that it groups symbols of approximately the same frequency. The symbols participating in the most repetitive parts of S have a good chance of having similar frequencies and thus of belonging to the same subalphabet S_j, where their repetitiveness will be preserved. On the other hand, the larger alphabets, where GCC cannot be applied, are likely to contain less frequent symbols, whose representation using faster structures like GMR or WT/WM do not miss very important opportunities to exploit repetitiveness.

Note that, if we do not use WTRP for the larger subalphabets, then the time performance for rsa queries stays within $O(\log n)$, independently of the alphabet size. In exchange, we cannot bound the size of the representation in terms of the size of the grammar that represents S. Instead, if we use WTRP, our worst-case guarantees are the same as for WTRP itself, but in practice our structure will prove to be much better, especially in time.

4.1. AP with GCC in practice

We introduce two new parameters for the combination of AP and GCC. The first parameter, cut, tells that the $2^{\texttt{cut}}$ most frequent symbols will be directly represented in K. This parameter must be set carefully to avoid increasing too much the alphabet of K, since K is represented with GCC.

Our second parameter is $\texttt{cut}_o$, which tells how many of the first S_j classes are to be represented with GCC. For the remaining sequences S_j we consider two options: (a) if S_j is not grammar-compressible, we use GMR [26], which does not compress but is very fast, or (b) if S_j is still grammar-compressible, we use WTRP, which is the grammar-based variant that performed best.

Table 1
Statistics of the datasets. The length is measured in millions of symbols and rounded.

Dataset	$n/10^6$	σ	H_0	RP	LZ	r/n
DNA.1	99	5	2.00	0.819	0.172	0.094
DNA.01	99	5	2.00	0.178	0.042	0.016
DNA.001	99	5	2.00	0.075	0.024	0.007
DNA.0001	99	5	2.00	0.063	0.021	0.006
para	429	5	2.12	0.376	0.191	0.036
influenza	154	15	1.97	0.280	0.132	0.019
escherichia	112	15	2.00	1.048	0.524	0.133
fiwikitags	48	24	3.37	0.110	0.219	0.031
einstein	92	117	5.04	0.019	0.009	0.001
software	210	134	4.69	0.139	0.214	0.009
einstein.words	17	8,046	9.92	0.076	0.003	0.001
fiwiki	86	102,423	11.06	0.235	0.034	0.008
indochina	100	2,576,118	15.39	1.906	0.159	0.076

5. Experimental results

5.1. Setup and datasets

We used an Intel® Xeon® E5620 at 2.40 GHz with 96 GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in C++. The compiler is g++ version 4.7, with -O9 optimization. We implemented our solutions on top of Libcds (github.com/fclaude/libcds) and use Navarro's implementation of RePair (www.dcc.uchile.cl/gnavarro/software/repair.tgz).

Table 1 shows statistics of interest about the datasets used and their compressibility: length (n), alphabet size (σ), zero-order entropy (H_0), bits per symbol (bps) obtained by RePair (RP, assuming $(2(r-\sigma)+c)\lceil \lg r \rceil$ bits, see Section 2.2), bps obtained by p7zip (LZ, www.7-zip.org), a Lempel–Ziv compressor, and finally r/n is the number of runs of the BWT [16] of each dataset divided by n (see Section 6.1).

We use various DNA collections from the Repetitive Corpus of *Pizza&Chili*.[4] On one hand, to study precisely the effect of repetitiveness in the performance of our rsa proposals, we generate four synthetic collections of about 100 MB: DNA 1%, DNA 0.1%, DNA 0.01%, and DNA 0.001%. Each DNA p% text is generated starting from 1 MB of real DNA text, which is copied 100 times, and each copied base is changed to some other value with probability $p/100$. This simulates a genome database with different variability between the genomes. As real genomes, we used collections para, influenza, and escherichia, also obtained from *Pizza&Chili*. From the statistics of Table 1, we see that para and influenza are actually very repetitive, while escherichia is not that much. Collection einstein corresponds to Wikipedia versions of articles about Albert Einstein in German (also available at *Pizza&Chili*) and is the most repetitive dataset we have. Text einstein.words is the same collection but regarded as a sequence of words, instead of characters. Sequence fiwiki is a prefix of a Wikipedia repository in Finnish[5] tokenized as a sequence of words instead of characters. Sequence fiwikitags corresponds to the XML tags extracted from a prefix from the same Finnish Wikipedia repository. Finally, indochina is a subgraph of the Web graph *Indochina2004* available at the WebGraph project[6] containing 2,531,039 nodes and 97,468,933 edges. Each node has an adjacency list of nodes, which is stored as a sequence of integers. Each list is separated from the next with a special separator symbol.

5.2. Parameterizing the data structures

We compare our data structures with several others. The list of structures compared, along with the parameters used, is listed next. These parameter ranges are chosen because they have been proved adequate in previous work, or because we have obtained the best space/time tradeoffs with them.

- GCC.N is our structure for small alphabets where we sample S at regular intervals. We set the sampling rate to $s = \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, the rule sampling to $\delta = \{0, 1, 2, 4\}$, and the superblock sampling to $s' = \{5, 8\}$.
- GCC.C is our structure for small alphabets where we sample C at regular intervals. We set the sampling rate to $s = \{2^6, 2^7, 2^8, 2^9, 2^{10}\}$ and the rule sampling to $\delta = \{0, 1, 2, 4\}$.
- {WT|WM|WTH|WMH}.{CM|RRR} is a wavelet tree, a wavelet matrix, a Huffman-shaped wavelet tree or a Huffman-shaped Wavelet Matrix with bitmaps represented either with CM or RRR. For CM we use the implementation [27] with one level of counters over the plain bitmap, while RRR corresponds to the implementation [19] of the compressed bitmaps of Raman et al. [48]. In both cases, the sampling rate for the counters was set to $\{32, 64, 128\}$.

[4] http://pizzachili.dcc.uchile.cl/repcorpus.

[5] http://www.cs.helsinki.fi/group/suds/rlcsa.

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

- {WT|WM|WTH|WMH}.RP are the WT, WM, WTH or WMH, with the bitmaps compressed with RePair. Therefore, WM.RP is equivalent to WTRP [47], but with our improved implementation using a wavelet matrix and GCC for the bitmaps. As in WTRP, we use several bitmap representations depending on the compressibility of the bitmap: GCC varying the parameters as described above, RRR or CM with sampling set to 32. We choose the one using the least space among these.
- AP is a plain alphabet partitioning implementation [7]. We used parameter values $\texttt{cut} = \{2^3, 2^4, 2^5, 2^6\}$ and $\texttt{cut}_\circ = \{1, 3, 5\}$. The sequence K is represented with WT.RRR with sampling set to 32. The sequences S_j are represented with GMR using the default configuration provided in the libcds tutorial.[7]
- AP.RP.{WMRP|GMR} is our AP-based variant for large alphabets. We use the same values cut and $\texttt{cut}_\circ$ as for AP. The sequence K and the first $\texttt{cut}_\circ$ sequences S_j are represented with GCC. The remaining sequences S_j are represented either with WM.RP or with GMR, using their already described configurations.
- MWTH.RP is a MWTH using RePair-compressed sequences in the nodes. As for AP.RP, we use two different representations for the node sequences. The first $\texttt{cut} = \{2, 3, 4\}$ levels are represented with GCC, and the rest with a WT.RRR with fixed sampling 32. We tested arities in $\{4, 8, 16\}$. We did not try combining with the WM because it is slower (requires more operations) and the overhead of $\sigma/2^b$ nodes is not as large as for σ nodes of the binary case. Also, the Huffman-shaped variants are shown to be always superior.

Among all the data points resulting from the combination of all the parameters, in the experiments we only show those points which are space/time dominant.

Regarding queries, those for access are positions at random in $S[1, n]$. For rank, we used a random position p in $S[1, n]$ and the symbol is $S[p]$. Finally, for select, we took a random position p in $S[1, n]$, using $S[p]$ and a random rank in $[1, \texttt{rank}_{S[p]}(S, n)]$. We generated 10, 000 queries of each type, reporting the average time for each operation.

In Section 3 we proposed two sampling approaches for GCC: GCC.N is regular in S and GCC.C is regular in C. We anticipated that GCC.C should use less space on more repetitive sequences, but it could be slower. Now we compare both sampling methods on the repetitive sequences with smaller alphabets described in Table 1. Fig. 6 shows the results for rank and select (access is equivalent to rank in our algorithms).

While, as said, GCC.C might use less space than GCC.N when the sequence is more repetitive, this occurs in practice only slightly on DNA0001, and spaces become closer as repetitiveness decreases on synthetic datasets (DNA001 to DNA1). Still, the differences are very slight, and instead GCC.N is much faster than GCC.C for the same space usage. The same occurs in the real sequences, where GCC.C uses less space than GCC.N only in fiwikitags. For the remaining experiments, we will use only GCC.N.

5.3. Performance on small alphabets

We compare our GCC.N with WT.RP, WTH.RP, and WM.RP. We also include in the comparison two statistically compressed representations that are the best for small and moderate alphabets: WTH.CM and WTH.RRR.

Fig. 7 shows the results for rank and select on the real collections that have small and moderate alphabets (again, the results for access are very similar to those for rank). It can be seen that WTH.RP generally performs better than WT.RP in space and time, as expected. The variant WM.RP performs slightly better than WT.RP in space, as it represents only one grammar per level and not per node (the difference would be higher on larger alphabets). In exchange, WM.RP is slightly slower than WT.RP because it performs more rank/select operations on the bitmaps represented with GCC. Finally, WMH.RP uses less space than WM.RP only in some cases, but it generally outperforms it for the same space. It performs particularly well on escherichia, the least repetitive of the datasets.

Recall that WM.RP is our improved version of previous work, WTRP [47], and it is now superseded by GCC.N. The space of WM.RP is in most cases similar to that of GCC.N, which means that WM.RP is actually close to the worst-case space estimation, $O(g\sigma \log n)$. In some cases, GCC is significantly smaller. More importantly, GCC.N is 2–15 times faster than WM.RP, and also 2–7 times faster than WTH.RP, the faster of the competitors in this family, which also uses more space than GCC.N. GCC.N handles queries in a few microseconds.

On the other hand, the representations that compress statistically, WTH.CM and WTH.RRR, are about an order of magnitude faster than GCC.N, but also take 5–15 times more space (except on escherichia, which is not repetitive).

5.4. Performance on large alphabets

Now we use the collections einstein (again), software, einstein.words, fiwiki, and indochina from Table 1, to compare the performance on moderate and large alphabets. We compare the two versions of our AP.RP, our MWTH.RP, and all the statistically compressed or compact schemes for large alphabets: WM/WMH with CM/RRR and AP (we only exclude WM.CM, which always loses to others). In the first two collections, whose alphabet size is moderate, we also include GCC.N, to allow comparing its performance with our variants for large alphabets in these intermediate cases.

[7] https://github.com/fclaude/libcds/blob/master/tutorial/tutorial.pdf.

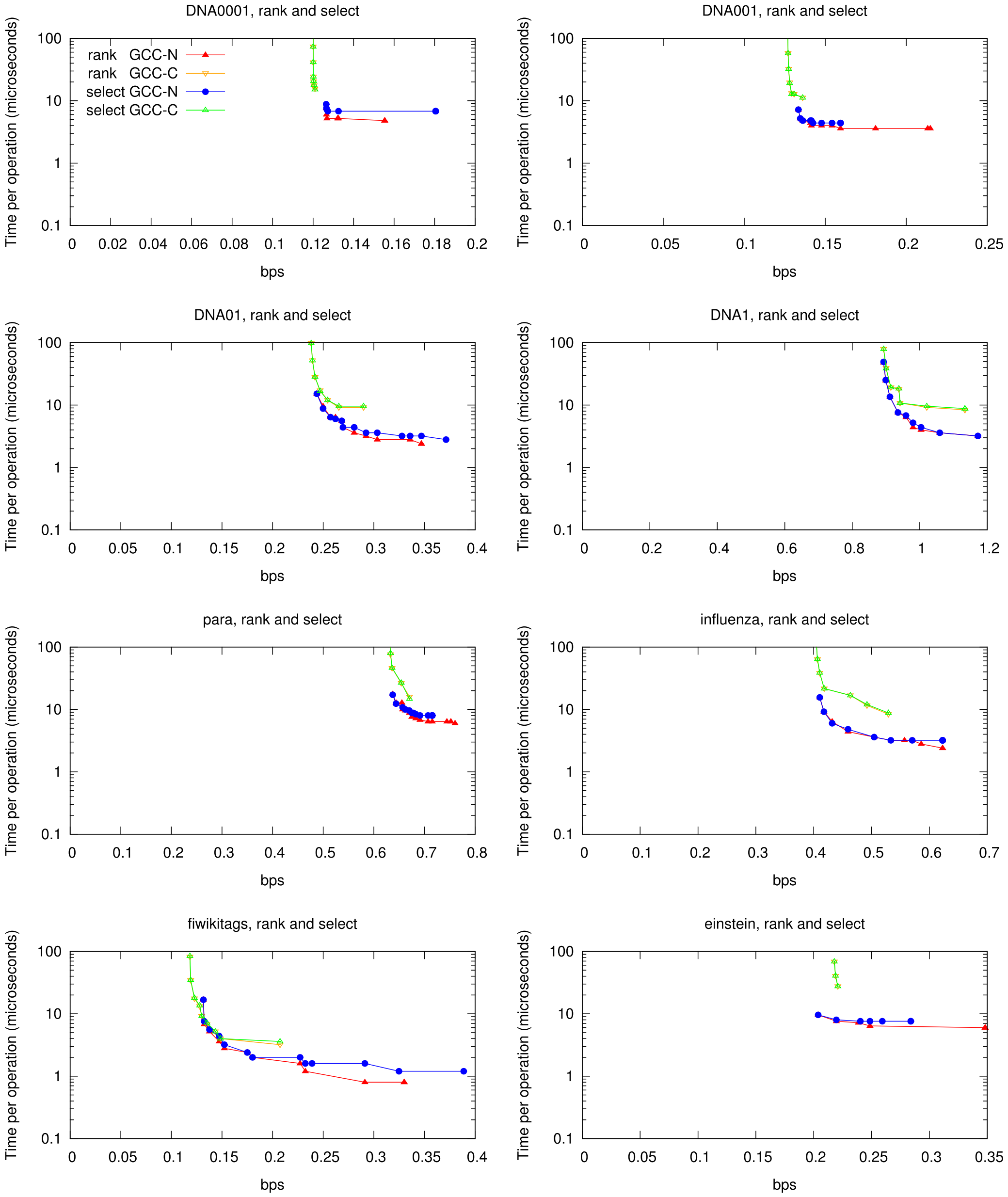


Fig. 6. Comparison of `rank` and `select` performance of GCC.N and GCC.C.

Fig. 8 shows the results for `rank` and `select` queries (once again, `access` is omitted for being very similar to the results of `rank`).

Recall that `WM.RP` is our improvement over the previous work, `WTRP` [47]. The Huffman-shaped variant, `WMH.RP`, outperforms it only slightly in time. Our multi-ary version, `MWTH.RP`, is clearly faster, but not smaller as one could expect. Indeed, it is larger when σ grows, probably due to the use of pointers. What is most interesting, however, is that all those variants are clearly superseded by our `AP.RP.WMRP`, which dominates them all in time (only reached by `MWTH.RP` while using much more space) and in space (only reached by `WM.RP` while using much more time). Compared with previous work [47], `AP.RP.WMRP` is then 2–4 times faster than `WTRP`, while using the same space or less. `AP.RP.WMRP` handles queries in a few tens of microseconds.

Note the particularly bad performance of the Huffman-based versions on `Indochina`. This is because this collection contains inverted lists, which form long increasing sequences that become runs in the wavelet tree; the Huffman rearrangement breaks those runs.

Our second variant, `AP.RP.GMR`, is not so interesting for repetitive collections. On `einstein` and `software` it performs similarly to `AP.RP.WMRP`. On the others, it is 2–5 times faster, but it uses much more space than `AP.RP.WMRP`, not so far

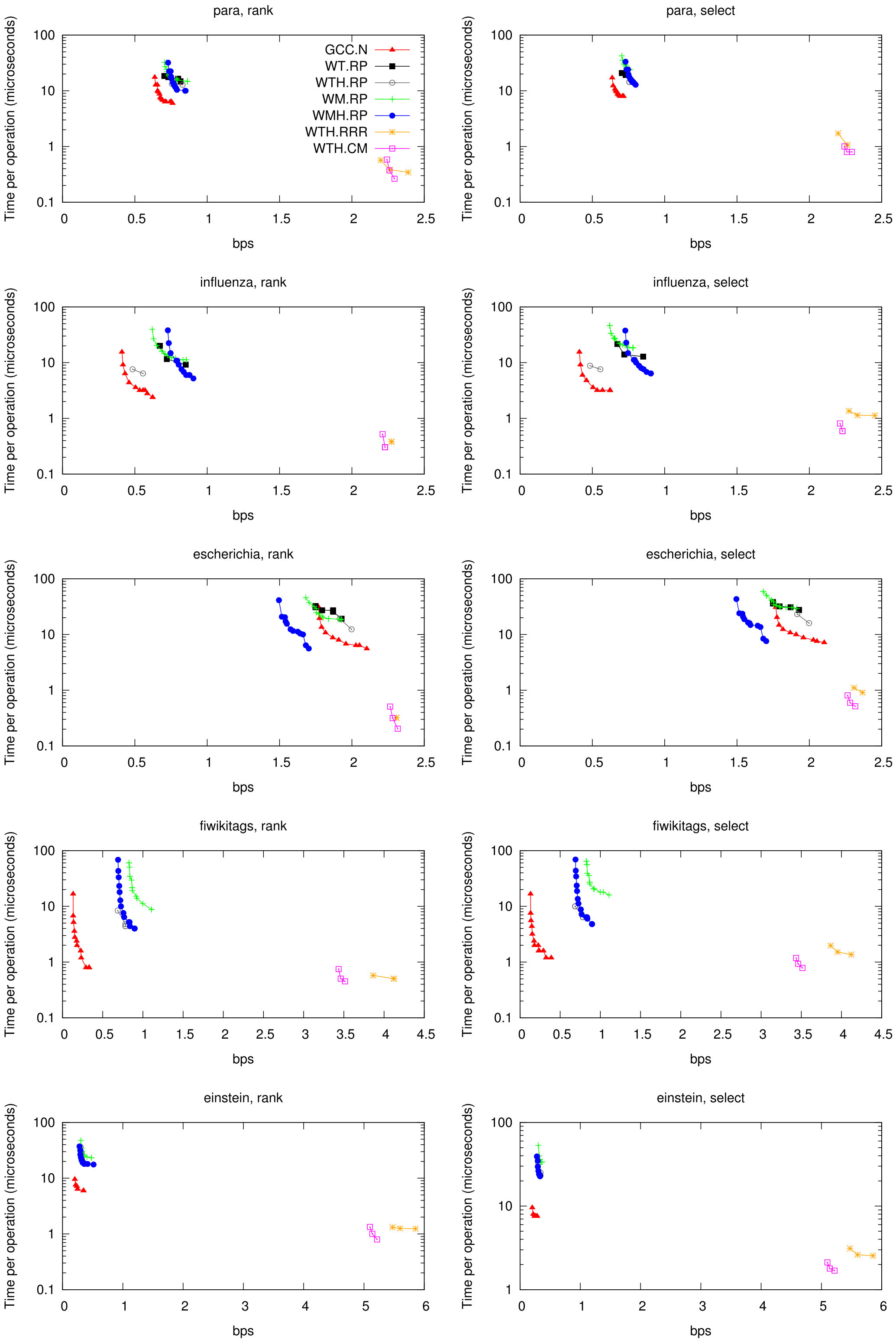


Fig. 7. Space–time tradeoffs for `rank` and `select` queries over small alphabets (time in logscale).

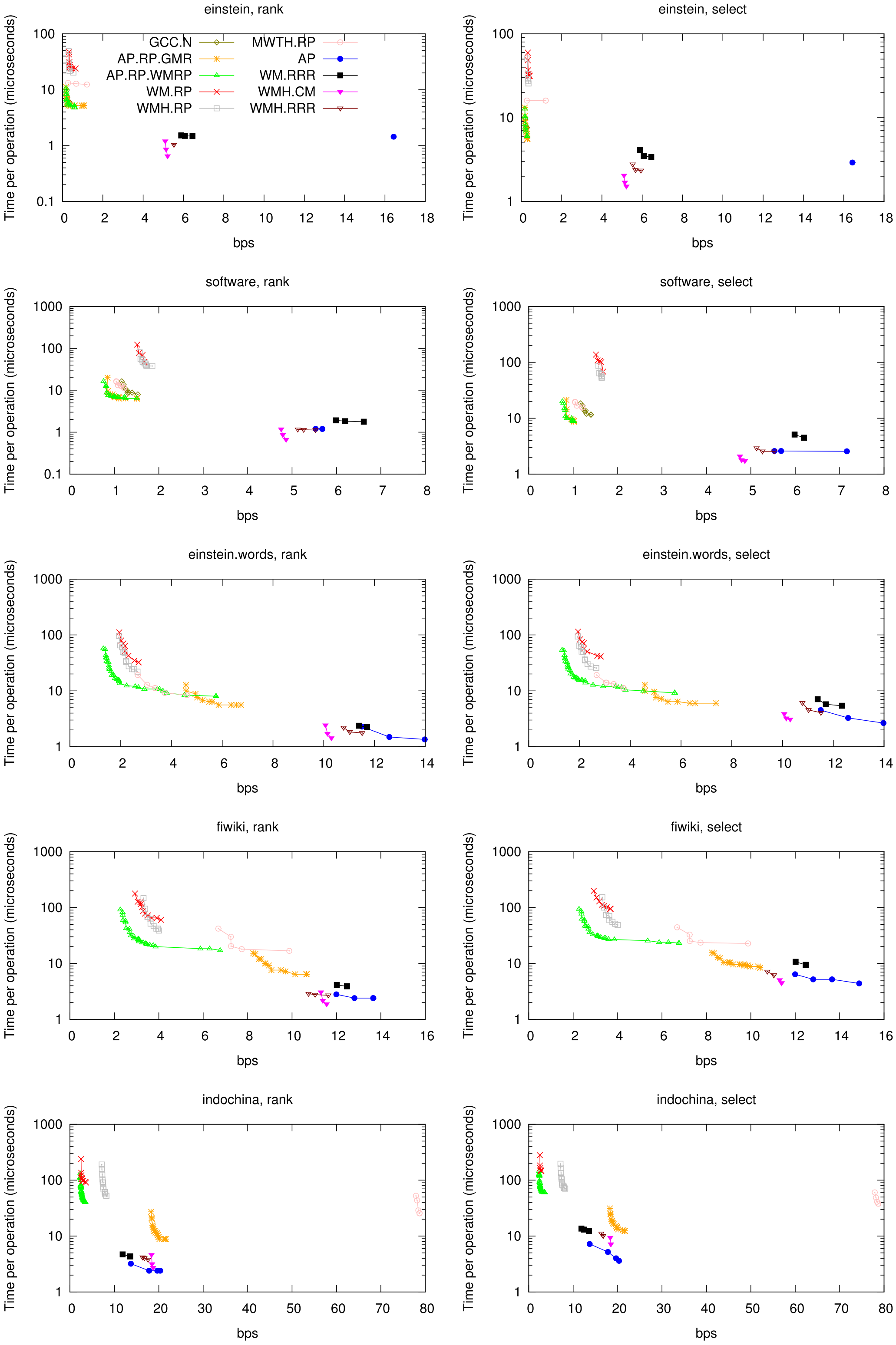


Fig. 8. Space–time tradeoffs for `rank` and `select` queries over moderate and large alphabets (time in logscale).

from that used by statistical representations. Those are, as before, about an order of magnitude faster than AP.RP.WMRP, but also use 3–5 times more space. Also, we can see that GCC.N is competitive on einstein, which is very repetitive, but not so much on software. Both of our new AP.RP versions designed for large alphabets outperform it in space, while they are not slower in time (in some cases they are even faster).

6. Applications

We explore now a couple of text indexing applications, where our new rsa-capable representations can improve the space for repetitive text collections.

6.1. Self-indices

Given a string $S[1,n]$ over alphabet $\Sigma = [1,\sigma]$, a *self-index* is a data structure that represents S and handles operations count(p), which returns the number of occurrences of a string pattern p in S; locate(p), which reports the positions of the occurrences of p in S; and extract(i, j), which retrieves $S[i, j]$.

A well-known family of self-indices are the FM-Indices [23]. Modern FM-Indices [24] build all their functionality on access and rank queries on the BWT (Burrows Wheeler Transform) [16] of S, S_{bwt}. Then, operation count on $p[1,m]$ takes time $O(m \cdot \alpha)$, α being the time to answer access and rank queries on S_{bwt}. The time to answer locate and extract is also proportional to α. Therefore, the time of rsa queries on S_{bwt} directly impacts on the FM-Index performance.

The string S_{bwt} is a reordering of the symbols of S, therefore $H_0(S_{bwt}) = H_0(S)$. Thus, zero-order-compressed representations of S also obtain zero-order compression of S_{bwt}. However, some kinds of zero-order compressors, in particular WT.RRR and WM.RRR, applied on S_{bwt} obtain $nH_k(S)$ bits of space for any $k < \log_\sigma n$ [38]. Further, S_{bwt} is typically formed by a few long *runs* of equal symbols: the number of runs is at most $nH_k(S) + \sigma^k$ for any k [36], and the number is much lower on repetitive sequences [39]. Thus, in a highly repetitive scenario, the runs of S_{bwt} are much longer than $\log_\sigma n$ (see Table 1), and typical k-order statistical compression of S_{bwt} fails to capture its most important regularities.

Run-Length FM-Indices [36,39] aim to capture these regularities. A Run-Length FM-Index stores in S'_{bwt} the first symbol of each run, marking their positions in a bitmap $R[1,n]$ (they also store a bitmap $R'[1,n]$ with a reordering of the bits in R). Compressed Suffix Arrays (CSAs) [30,49] (another family of self-indices) have also been adapted to exploit these runs, in a structure called Run-Length CSA [39]. In general, FM-Indices are preferred over CSAs for sequences over small alphabets, because the cost of rsa operations increases with σ, while the equivalent operations on the CSAs do not depend on it.

An alternative to Run-Length FM-Indices is to grammar-compress S_{bwt} with GCC, our rsa structure for repetitive sequences on small alphabets. To evaluate if grammar compression of S_{bwt} captures more regularities than run-length compression, we compare the following FM-Index implementations:

- FMI-GCC, using the variant GCC.N to represent S_{bwt}.
- FMI-AP.RP.WTRP, using the variant AP.RP.WTRP to represent S_{bwt}.
- FMI-WTH.RRR, which uses WTH.RRR to represent S_{bwt}.
- FMI-WT.RRR, which uses WT.RRR to represent S_{bwt}.
- RLFMI-WTH+DELTA, a Run-Length FM-Index [39] where bitmaps R and R' are compressed with DELTA, while S'_{bwt} is represented with WTH.RRR.
- RLCSA, a Run-Length Compressed Suffix Array [39] setting the sampling rate of its function Ψ to $\{32, 64, 128\}$.

We used the real DNA datasets and fiwikitags, as well as einstein and software to show the case of larger alphabets. We averaged 10,000 queries for patterns picked at random from each dataset. We evaluate the performance of the operation count in the indices, for various pattern lengths. Fig. 9 shows the results for $m = 8$, since all the lengths gave similar results.

As it can be seen, the FMI-GCC obtains the least space on the smaller alphabets. The space of the RLCSA is close, but still larger than that of the FMI-GCC, in collections fiwikitags and influenza. For para and escherichia the differences are larger, our structure using 60%–80% of the RLCSA space. Interestingly, grammar compression of S_{bwt} is stronger than the RLCSA compression especially when the sequence is not so repetitive. In exchange, the RLCSA is about an order of magnitude faster.

Our index also uses half the space, or less, than the RLFMI-WTH+DELTA, which also adapts to repetitiveness but not as well as grammar compression, and performs badly as soon as repetitiveness starts to decrease. Finally, compared with the best statistical approach, the FMI-WTH.RRR, the differences are even larger: our solution needs only 20%–40% of the space in the most repetitive collections, only getting closer in escherichia, which is not so repetitive.

In terms of time performance, the FMI-GCC is in the same order of magnitude of RLFMI-WTH+DELTA, yet it is slower. Compared with FMI-WTH.RRR, our index is about an order of magnitude slower.

On the larger alphabets, instead, the FMI-AP.RP.WTRP outperforms the FMI-GCC and uses about the same space as the RLFMI-WTH+DELTA, while being faster or equally fast. It is only 2–4 times slower than the statistical approaches, while using 10%–20% of their space. However, as expected, the RLCSA outperforms every FM-index on larger alphabets.

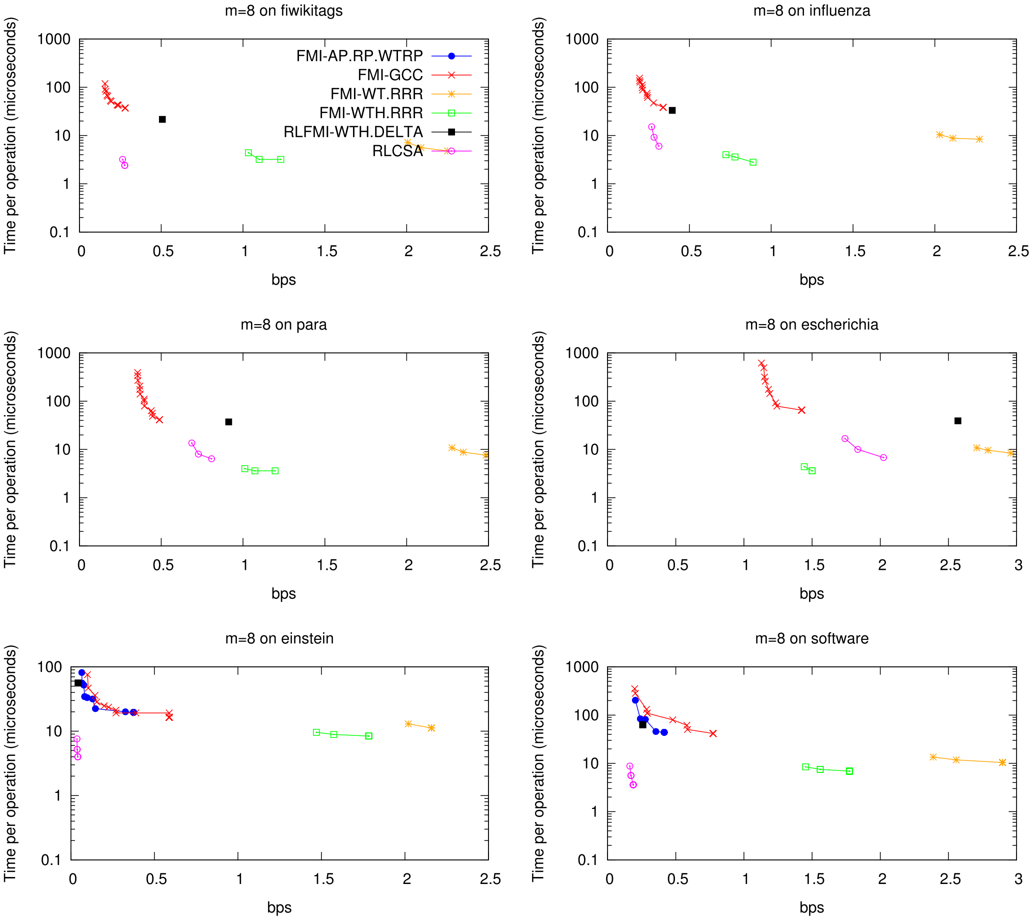


Fig. 9. Space–time tradeoffs for operation `count` with $m = 8$.

In the sequel we call `GFMI` to `FMI-GCC` or `FMI-AP.RP.WTRP`, whichever is better.

6.2. XML and XPath

Now we show the impact of our new representations in the indexing of repetitive XML collections. `SXSI` [2] is a recent system that represents XML datasets in compact form and supports XPath queries on them. Its query processing strategy uses a tree automaton that traverses the XML data, using several queries on the content and structure to speed up navigation towards the points of interest. `SXSI` represents the XML data using three separate components: (1) a text index that represents and carries out pattern searches over the text nodes (any compressed full-text index [44] can be used); (2) a balanced parentheses representation of the XML topology that supports navigation using $2 + o(1)$ bits per node (various alternatives exist [1]); and (3) an `rsa`-capable representation of the sequence of the XML opening and closing tags.

When the XML collection is repetitive (e.g., versioned collections like Wikipedia, versioned software repositories, etc.), one can use the `RLCSA` [39] as the text index for (1), but now we also consider using our new `GFMI`. Components (2) and (3), which are usually less relevant in terms of space, may become dominant if they are represented without exploiting repetitiveness. For (2), we consider `GCT`, a tree representation aimed at repetitive topologies [45], and a classical representation (`FF` [1]). For (3), we will use our new repetition-aware sequence representations, comparing them with the alternative proposed in `SXSI` (`MATRIX`, using one compressed bitmap per tag) and a `WTH` representation.

We use a repetitive data-centric XML collection of 200 MB from a real software repository. Its sequence of XML tags, called `software`, is described in Table 1. As a proof of concept, we run two XPath queries that make intensive use of the sequence of tags and the tree topology: `XQ1=//class[//methods]`, and `XQ2=//class[methods]`.

Table 2 shows the space in bpe (bits per element) of components (2) and (3). An element is an opening or a closing tag, so there are two elements per XML tree node. The space of the `RLCSA` without sampling is always 0.18 bits per character of the XML document, whereas our new `GFMI` uses 0.15 if combined with `AP.RP.WMRP`. The table also shows the impact of each component in the total size of the index, considering this last space. On the rightmost columns, it shows the time to answer both queries.

Table 2
Results on XML. Columns **tags** and **tree** are in bpe. Columns **XQ1** and **XQ2** show query time in microseconds.

Dataset	tags	tree	%tags	%tree	%text	XQ1	XQ2
MATRIX+FF	12.40	1.27	88.89	9.12	1.99	16	35
WTH+FF	2.88	1.27	65.00	28.68	6.32	92	113
GCC+FF	0.37	1.27	19.29	66.17	14.54	184	226
GCC+GCT	0.37	0.19	44.13	22.65	39.74	774	3,066

The original SXSI (MATRIX+FF) is very fast but needs almost 14 bpe, which amounts to 98% of the index space in this repetitive scenario (in non-repetitive text-centric XML, this space is negligible). By replacing the MATRIX by a WTH, the space drops significantly, to slightly over 4 bpe, yet times degrade by a factor of 3–6. By using our GCC for the tags, a new significant space reduction is obtained, to 2.65 bpe, and the times increase by a factor of 2, becoming 6–12 times slower than the original SXSI. Finally, changing FF by GCT [45], we can reach as low as 0.56 bpe, 24 times less than the original SXSI, and using around 60% of the total space. Once again, the price is the time, which becomes 50–90 times slower than the basic SXSI. The price of using the slower GCT is more noticeable on XQ2, which uses more operations on the tree.

While the time penalty is 1–2 orders of magnitude, we note that the gain in space can make the difference between running the index in memory or on disk; in the latter case we can expect it to be up to 6 orders of magnitude slower.

7. Conclusions

We have introduced new sequence representations that take advantage of the repetitiveness of the sequence, by enhancing the output of a grammar compressor with extra information to support efficient direct access, as well as rank and select operation on the sequence. The only previous grammar-compressed representation [47] is 2–15 times slower and uses the same or more space than our new representations. Our structures answer queries in a few tens of microseconds, which is about an order of magnitude slower than the times of statistically compressed representations. However, on repetitive collections, our structures use 2–15 times less space. We have also explored two applications where repetitiveness is a sharp source of compressibility, and have shown how our structures allow one to further exploit that repetitiveness to obtain significantly less space.

An aspect where our structures could possibly be improved is in the clustering of the alphabet symbols used when partitioning the alphabet, both in the simple case of alphabet partitioning and in the hierarchical case of wavelet trees and matrices. In the first case, we obtained a significant space improvement by sorting the symbols by frequency, whereas in the second case none of our attempts performed noticeably better than the original alphabet ordering. While unsuccessful for now, we believe that some clever clustering scheme that avoids separating symbols that appear together in repetitive parts of the sequence could considerably improve the space on large alphabets.

Another future goal is to find ways to improve the time of these grammar compressed representations. We believe this is possible, even if known lower bounds suggest that there must be a price of at least an order of magnitude compared with statistically compressed representations. A more far-fetched goal is to build on Lempel–Ziv compressed representations. Lempel–Ziv is more powerful than grammar compression, but supporting the desired operations on it is thought to be more difficult.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proc. 12th Workshop on Algorithm Engineering and Experiments, ALENEX, 2010, pp. 84–97.
[2] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, N. Välimäki, Fast in-memory XPath search using compressed indexes, Softw. Pract. Exp. 45 (3) (2015) 399–434.
[3] D. Arroyuelo, V. Gil-Costa, S. González, M. Marín, M. Oyarzún, Distributed search based on self-indexed compressed text, Inf. Process. Manag. 48 (5) (2012) 819–827.
[4] D. Arroyuelo, S. González, M. Marín, M. Oyarzún, T. Suel, To index or not to index: time–space trade-offs in search engines with positional ranking functions, in: Proc. 35th International ACM Conference on Research and Development in Information Retrieval, SIGIR, 2012, pp. 255–264.
[5] D. Arroyuelo, S. González, M. Oyarzún, Compressed self-indices supporting conjunctive queries on document collections, in: Proc. 17th International Symposium on String Processing and Information Retrieval, SPIRE, in: LNCS, vol. 6393, 2010, pp. 43–54.
[6] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, 2nd edition, Addison–Wesley, 2011.
[7] J. Barbay, F. Claude, T. Gagie, G. Navarro, Y. Nekrich, Efficient fully-compressed sequence representations, Algorithmica 69 (1) (2014) 232–268.
[8] J. Barbay, F. Claude, G. Navarro, Compact binary relation representations with rich functionality, Inf. Comput. 232 (2013) 19–37.
[9] J. Barbay, M. He, J.I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multilabeled trees, ACM Trans. Algorithms 7 (4) (2011), article 52.
[10] J. Barbay, G. Navarro, On compressing permutations and adaptive sorting, Theor. Comput. Sci. 513 (2013) 109–123.
[11] D. Belazzougui, P.H. Cording, S.J. Puglisi, Y. Tabei, Access, rank, and select in grammar-compressed strings, in: Proc. 23th Annual European Symposium on Algorithms (ESA), in: LNCS, vol. 9294, 2015, pp. 142–154.
[12] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S.J. Puglisi, Y. Tabei, Queries on LZ-bounded encodings, in: Proc. 25th Data Compression Conference, DCC, 2015, pp. 83–92.
[13] D. Belazzougui, G. Navarro, Optimal lower and upper bounds for representing sequences, ACM Trans. Algorithms 11 (4) (2015), article 31.
[14] P. Bille, G.M. Landau, R. Raman, K. Sadakane, S.S. Rao, O. Weimann, Random access to grammar-compressed strings and trees, SIAM J. Comput. 44 (3) (2015) 513–539.

[15] N. Brisaboa, S. Ladra, G. Navarro, DACs: bringing direct access to variable-length codes, Inf. Process. Manag. 49 (1) (2013) 392–404.
[16] M. Burrows, D. Wheeler, A Block Sorting Lossless Data Compression Algorithm, Technical report 124, Digital Equipment Corporation, 1994.
[17] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Trans. Inf. Theory 51 (7) (2005) 2554–2576.
[18] D. Clark, Compact PAT Trees, PhD thesis, University of Waterloo, Canada, 1996.
[19] F. Claude, G. Navarro, Practical rank/select queries over arbitrary sequences, in: Proc. 15th International Symposium on String Processing and Information Retrieval, SPIRE, in: LNCS, vol. 5280, 2008, pp. 176–187.
[20] F. Claude, G. Navarro, Fast and compact Web graph representations, ACM Trans. Web 4 (4) (2010), article 16.
[21] F. Claude, G. Navarro, A. Ordóñez, The wavelet matrix: an efficient wavelet tree for large alphabets, Inf. Sci. 47 (2015) 15–32.
[22] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (1) (2009), article 4.
[23] P. Ferragina, G. Manzini, Indexing compressed texts, J. ACM 52 (4) (2005) 552–581.
[24] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Trans. Algorithms 3 (2) (2007), article 20.
[25] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, S.J. Puglisi, LZ77-based self-indexing with faster pattern matching, in: Proc. 11th Latin American Symposium on Theoretical Informatics, LATIN, in: LNCS, vol. 8392, 2014, pp. 731–742.
[26] A. Golynski, I. Munro, S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2006, pp. 368–373.
[27] R. González, Sz. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms, WEA, 2005, pp. 27–38.
[28] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2003, pp. 841–850.
[29] R. Grossi, A. Orlandi, R. Raman, Optimal trade-offs for succinct string indexes, in: Proc. 37th International Colloquium on Algorithms, Languages and Programming, ICALP, in: LNCS, vol. 6199, 2010, pp. 678–689.
[30] R. Grossi, J. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2006) 378–407.
[31] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proc. IRE 40 (9) (1952) 1098–1101.
[32] J.C. Kieffer, E.-H. Yang, Grammar-based codes: a new class of universal lossless source codes, IEEE Trans. Inf. Theory 46 (3) (2000) 737–754.
[33] S. Kreft, G. Navarro, On compressing and indexing repetitive sequences, Theor. Comput. Sci. 483 (2013) 115–133.
[34] J. Larsson, A. Moffat, Off-line dictionary-based compression, Proc. IEEE 88 (11) (2000) 1722–1732.
[35] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inf. Theory 22 (1) (1976) 75–81.
[36] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nord. J. Comput. 12 (1) (2005) 40–66.
[37] V. Mäkinen, G. Navarro, Position-restricted substring searching, in: Proc. 7th Latin American Symposium on Theoretical Informatics, LATIN, in: LNCS, vol. 3887, 2006, pp. 703–714.
[38] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Trans. Algorithms 4 (3) (2008), article 32.
[39] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, J. Comput. Biol. 17 (3) (2010) 281–308.
[40] J.I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, in: LNCS, vol. 1180, 1996, pp. 37–42.
[41] G. Navarro, Indexing highly repetitive collections, in: Proc. 23rd International Workshop on Combinatorial Algorithms, IWOCA, in: LNCS, vol. 7643, 2012, pp. 274–279.
[42] G. Navarro, Spaces, trees and colors: the algorithmic landscape of document retrieval on sequences, ACM Comput. Surv. 46 (4) (2014), article 52.
[43] G. Navarro, Wavelet trees for all, J. Discret. Algorithms 25 (2014) 2–20.
[44] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007), article 2.
[45] G. Navarro, A. Ordóñez, Faster compressed suffix trees for repetitive text collections, in: Proc. 13th International Symposium on Experimental Algorithms, SEA, in: LNCS, vol. 8504, 2014, pp. 424–435.
[46] G. Navarro, A. Ordóñez, Grammar compressed sequences with rank/select support, in: Proc. 21st International Symposium on String Processing and Information Retrieval, SPIRE, in: LNCS, vol. 8799, 2014, pp. 31–44.
[47] G. Navarro, S.J. Puglisi, D. Valenzuela, General document retrieval in compact space, ACM J. Exp. Algorithmics 19 (2) (2014), article 3.
[48] R. Raman, V. Raman, S. Srinivasa Rao, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (4) (2007), article 43.
[49] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, J. Algorithms 48 (2) (2003) 294–313.
[50] H. Sakamoto, A fully linear-time approximation algorithm for grammar-based compression, J. Discret. Algorithms 3 (2–4) (2005) 416–430.
[51] Y. Tabei, Y. Takabatake, H. Sakamoto, A succinct grammar compression, in: Proc. 24th Annual Symposium on Combinatorial Pattern Matching, CPM, in: LNCS, vol. 7922, 2013, pp. 235–246.
[52] E. Verbin, W. Yu, Data structure lower bounds on random access to grammar-compressed strings, in: Proc. 24th Annual Symposium on Combinatorial Pattern Matching, CPM, in: LNCS, vol. 7922, 2013, pp. 247–258.
[53] H.E. Williams, J. Zobel, Compressing integers for fast file access, Comput. J. 42 (3) (1999) 193–201.
[54] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann, 1999.
[55] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (3) (1977) 337–343.