Grammar Compressed Sequences with Rank/Select Support*

Gonzalo Navarro[1] and Alberto Ordóñez[2]

[1] Dept. of Computer Science, Univ. of Chile, Chile
gnavarro@dcc.uchile.cl
[2] Lab. de Bases de Datos, Univ. da Coruña, Spain
alberto.ordonez@udc.es

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. In several recent applications, the need to represent highly repetitive sequences arises, where statistical compression is ineffective. We introduce grammar-based representations for repetitive sequences, which use up to 10% of the space needed by representations based on statistical compression, and support direct access and rank/select operations within tens of microseconds.

1 Introduction

Given a sequence $S[1,n]$ drawn over an alphabet $\Sigma = [1,\sigma]$, an intensively studied problem in the past few years has been how to represent S space-efficiently while solving operations $\texttt{rank}_b(S,i)$ (number of occurrences of b in $S[1,i]$), $\texttt{select}_b(S,i)$ (i-th occurrence of b in S), and $\texttt{access}(S,i) = S[i]$. The motivation comes from a wide number of applications involving these functionalities: text indexes, document retrieval, data grids, and many others [25].

The most well-known data structure to solve `rank`/`select`/`access` (`rsa`) queries is the *wavelet tree* (`WT`) [18] (with several recent improvements for large alphabets [3,12]). These data structures are able to statistically compress the input sequence while efficiently solving `rsa` queries. However, they are unable to compress S beyond its statistical entropy.

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: compressed software repositories, versioned document collections, DNA datasets of individuals of the same species, and so on, which contain many near-copies of the same source code, document, or genome [24]. In this scenario, statistical compressors, or a

* Funded in part by 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).

E. Moura and M. Crochemore (Eds.): SPIRE 2014, LNCS 8799, pp. 31–44, 2014.

compressed WT, do not take a proper advantage of the repetitiveness [20], which is crucial to reduce the size of those usually huge datasets by orders of magnitude.

Grammar- and Lempel-Ziv-based compressors are very efficient at handling repetitive sequences. However, even supporting operation access is difficult on them. Let $S[1,n]$ be compressible into a grammar of size r, so that a grammar-based compressor uses $r\lg(r+\sigma)$ bits. Bille et al. [5] show how to represent S using $O(r\log n)$ bits so that $\texttt{access}(S,i)$ is solved in $O(\log n)$ time. Let z be the number of phrases into which a Lempel-Ziv parser factors S. Then a Lempel-Ziv compressor achieves $z(\lg n+\lg\sigma)$ bits. Gagie et al. [14] show how to represent S using $O(z\log n\log(n/z))$ bits so that $\texttt{access}(S,i)$ can be supported in time $O(\log n)$. Verbin and Yi [32] show that both times are essentially optimal. Note, however, that the spaces are at best proportional to the size of the compressed string, and that operations rank and select are not supported. This is to be contrasted with, for example, alphabet partitioning techniques [3,4] which obtain asymptotically the same space of a kth-order statistical compression of S, support access in $O(1)$ time, select in almost-constant time (or vice versa), and rank in the optimal $O(\log\frac{\log\sigma}{\log w})$ time on a RAM machine of w bits.

Various scenarios require rsa support on repetitive sequences. Some examples are: document retrieval on repetitive sequence collections, to represent the so-called "document array" [28]; XPath queries on versioned XML data, to represent the sequence of tags [2]; simulating positional inverted indexes on repetitive natural language text collections, by representing the sequence of words [6,15]; and bidirectional navigation of Web graphs, to represent adjacency lists [10].

The only current solution to provide rsa support on repetitive sequences is of practical nature [28]. The key idea is that repetitions in the input sequence S should also induce repetitions in the bitmaps of a WT built on it. This is true at least for the first few levels of the WT, since the WT construction algorithm splits such repetitions as we move downward in the tree. Therefore, if S is grammar-compressible, so are the first bitmaps of the WT. These first levels are compressed with an enhanced Re-Pair (a grammar compressor [21]) representation for bitmaps (RPB [28]) that supports rsa queries in $O(\log n)$ time. The remaining levels, which are not grammar-compressible, are compressed with statistical techniques for bitmaps (RRR [29]) or even not compressed at all (CM [9,23]). Thus, the rsa operations are supported in $O(\log n\log\sigma)$ worst-case time.

This solution, dubbed WTRP, has two main drawbacks: (a) Re-Pair compressed bitmaps RPB [28] are in practice orders of magnitude slower than RRR or CM to support rsa operations ($O(\log n)$ vs $O(1)$ time, in theory), what makes the WTRP significantly slower than a regular WT; (b) the WT construction quickly destroys the repetitiveness of S, and thus the size of the WT can be many times larger than the Re-Pair compressed sequence (there is no theoretical guarantee here).

In this paper we propose two new solutions for rsa queries over grammar compressed sequences. The first one, tailored to sequences over small alphabets, is obtained by enhancing and improving the RPB representation for bitmaps [28]. We dub this solution GCC (Grammar Compression with Counters). This may directly apply, for example, to sequences of XML tags. Our second structure

combines GCC with alphabet partitioning (AP) [3] and is aimed to sequences with large alphabets. AP splits the sequence S into subsequences over smaller alphabets, what lets us apply GCC on them (or a simpler and faster representation on the subsequences that are not grammar-compressible).

Our experiments on various real-life repetitive sequences show that our new representations use significantly less space, and are an order of magnitude faster, than WTRP, the only current solution [28]. They are still an order of magnitude slower than statistically compressed representations, but they also use an order of magnitude less space on repetitive sequences. We show, as a concrete application, the improvement obtained by plugging our structure to represent the sequence of tags within SXSI, a system that supports XPath queries on compactly represented XML data, when the collections are repetitive.

2 Basic Concepts and Related Work

2.1 Grammar compression of Sequences and Re-Pair

Grammar-compressing a sequence S means to find a context-free grammar that generates (only) S. Finding the smallest grammar that generates a given sequence S is NP-complete [8], but heuristics like Re-Pair [21] perform very well in practice, in linear time and space. This will be our compressor of choice.

Re-Pair 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. This process is repeated (X can be involved in future pairs) until the most frequent pair appears only once. The result is a tuple (R, C), where the dictionary R contains $r = |R|$ rules and C, of length $c = |C|$, is the final reduction of S after all the replacements carried out. Note that C is drawn from an alphabet of size $\sigma + r$, not only σ. Thus, the total output size is $(2r+c)\lg r$ bits. By using the technique of Tabei et al. [31], we represent the dictionary in $r \log r + O(r)$ bits, reducing the total space to $(r+c)\log r + O(r)$ bits. Finally, it is possible to force the grammar to be *balanced*, that is, that the grammar tree is of height $O(\log n)$ [30].

2.2 Bitmap Representations and RPB

Several classical solutions represent a binary sequence $B[1, n]$ with rsa support. Clark and Munro [9,23] (CM) use $o(n)$ bits on top of B and solve all the queries in $O(1)$ time. Raman et al. [29] (RRR) also support the operations in $O(1)$ time, but they statistically compress B to $nH_0(B)+o(n)$ bits, where $H_0(B)$ is the empirical zero-order entropy of B: if B has m 1s, then $H_0(B) = \frac{m}{n}\lg\frac{n}{m} + \frac{n-m}{n}\lg\frac{n}{n-m}$.

The only solution that exploits the repetitiveness of the bitmap was proposed by Navarro et al. [28] (RPB). They Re-Pair compress B with a balanced grammar and enhance the output (R, C) with extra information to solve rsa queries: Let $exp(X)$ be the string of terminals X expands to; then they store, for each rule $X \to YZ$, $\ell(X) = |exp(X)|$, the length of $exp(X)$, and $z(X) = rank_0(exp(X), \ell(X))$, the number of 0s in $exp(X)$.

Note that both values can be recursively computed as $\ell(X) = \ell(Y) + \ell(Z)$, with $\ell(0) = \ell(1) = 1$; and $z(X) = z(Y) + z(Z)$, with $z(0) = 1, z(1) = 0$. 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 $rank_1(B_d, r)$. To obtain $\ell(X)$ we check whether $B_d[X] = 1$. If so, then $\ell(X) = S_\ell[rank_1(B_d, X)]$. 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. An array $S_n[0, n/s]$ stores a tuple (p, o, rnk) at $S_n[i]$, where the expansion of $C[p]$ contains $B[i \cdot s]$, that is, $p = \max\{j, L(j) \leq i \cdot s\}$, where $L(j) = 1 + \sum_{k=1}^{j-1} \ell(C[k])$; $o = i \cdot s - L(p)$ is the offset within that symbol; and $rnk = rank_0(B, L(p) - 1)$. Let $S[0] = (0, 0, 0)$.

To solve $rank_0(B, i)$, 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 $l + \ell(C[p]) \leq i$. When $l \leq i < l + \ell(C[p])$, 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. Finally, we return rnk. Obviously, we can also compute $rank_1(B, i) = i - rank_0(B, i)$. Solving $access(B, i)$ is completely equivalent, but instead of returning rnk we return the terminal symbol we reach when $l = i$.

To solve $select_0(B, j)$, we binary search S_n to find $S_n[i] = (p, o, rnk)$ and $S_n[i + 1] = (p', o', rnk')$ such that $rnk < j \leq rnk'$. Then we proceed as for $rank_0$, but iterating as long as $z + z(C[p]) \leq j$, and then traversing by going left (to Y) when $z + z(Y) > j$, and going right (to Z) otherwise. The process for $select_1(B, j)$ is analogous (note 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, the total time for `rsa` is $O(s + \log n)$ and the total space is $O(r \log n + (n/s) \log n) + c \lg(\sigma + r)$ bits. The time is multiplied by δ if we use sampling.

2.3 Sequence Representations

The wavelet tree [18] (`WT`) is a complete balanced binary tree that represents a sequence S on $\Sigma = [1, \sigma]$. It is able to statistically compress the sequence and solves `rsa` queries in $O(\log \sigma)$ time. For large alphabets, a variant called wavelet matrix (`WM`) [12] performs better in practice. Assume we use a plain encoding of symbols in $\lceil \lg \sigma \rceil$ bits, where $a\langle j \rangle$ the jth most significant bit of $a \in \Sigma$. The `WM` construction algorithm starts with $S_l = S$ at level $l = 1$ and proceeds as follows: (1) build a single bitmap $B_l[1, n]$ where $B_l[i] = S_l[i]\langle l \rangle$; 2) compute $\tilde{z}_{l+1} = rank_0(B_l, n)$; (3) build sequence S_{l+1} such that, for $k \leq \tilde{z}_{l+1}$, $S_{l+1}[k] = S_l[select_0(B_l, k)]$, and for $k > \tilde{z}_{l+1}$, $S_{l+1}[k] = S_l[select_1(B_l, k - \tilde{z}_{l+1})]$; (4) repeat the process until $l = \lceil \log \sigma \rceil$. This is actually a reshuffling of the bits

of $S[i]\langle j\rangle$ for all i and j (akin to radix sorting the symbols of S), with $n\lceil \lg \sigma \rceil$ bits in total (plus $\lg n \lg \sigma$ for the $\tilde{z}_l$). The `rsa` operations are carried out with one binary `rsa` operation per level of the `WM`.

By representing the bitmaps B_l with `CM` [9,23], the total space is $n \lg \sigma(1+o(1))$ bits and the `rsa` time is $O(\log \sigma)$. By using `RRR` bitmap representation [29], the time complexity is retained but the space reduces to $nH_0(S) + O(\sigma \log n)$ bits, although the times are higher in practice. Zero-order compression is also obtained, with faster time in practice, by retaining the `CM` representation but using a tree with Huffman [19] shape instead of a balanced one, which gives $n(H_0(S)+1)(1+o(1)) + O(\sigma \log n)$ bits. The results are called `WTH` (Huffman-shaped `WT`) or `WMH` (Huffman-shaped `WM` [13]).

An alternative solution for `rsa` queries over large alphabets is alphabet partitioning (`AP`) [3], which obtains $nH_0(B) + o(n(H_0(B)+1))$ bits and solves `rsa` 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 over Σ_j. A string $K[1,n]$ indicates the sequence each symbol of S belongs. Then `rsa` operations on S are translated into `rsa` operations on K and on some subsequence S_j. Furthermore, the symbols in each Σ_j are of roughly the same frequency, so that using a fast compact (but not compressed) representation of S_j (`GMR`) [16] yields $O(\log \log \sigma)$ time and does not ruin the statistical compression of S. The actual implementation defines Σ_j as the set of the 2^{j-1}th to the (2^j-1)th most frequent symbols, and uses `WT` when this alphabet is small, and `GMR` when it is large.

The mapping to subalphabets is represented in a sequence $M[1,\sigma]$, where $M[a] = j$ iff $a \in \Sigma_j$. In each subsequence S_j, each $a \in \Sigma_j$ is rewritten as $rank_j(M,a)$, so the local alphabet is $[1, 2^{j-1}]$. Now, to find $S[i]$ we compute $j = K[i]$, $v = S_j[rank_j(K,i)]$, and $S[i] = select_j(M,v)$. To find $rank_a(S,i)$, we compute $j = M[a]$, $v = rank_j(M,a)$, $r = rank_j(K,i)$, and $rank_a(S,i) = rank_v(S_j,r)$. Finally, to find $select_a(S,i)$, we compute $j = M[a]$, $v = rank_j(M,a)$, $s = select_v(S_j,i)$, and $select_a(S,i) = select_j(K,s)$.

2.4 Re-Pair Compressed `WT`

As far as we know, `WTRP` [28] (or `WMRP` if implemented on a `WM`) is the only solution to support `rsa` on grammar-compressed sequences. The structure is a `WT` where all the bitmaps at each level l are concatenated, and then the bitmap B_l of each level l is compressed with `RPB` [28]. The rationale is that the repetitiveness of S is reflected in the bitmaps of the `WT`, at least for the first levels. That is because the `WT` construction splits the alphabet at each level, which potentially blurs the repeated substrings into many shorter repetitions.

Therefore, the bitmaps of the first few `WT` levels are likely to be compressible with Re-Pair, while the remaining ones are not. The authors [28] use at each level l the technique to represent B_l that yields the least space, `RPB`, `RRR`, or `CM`. 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 complexities become

$O(\log \sigma(s + \log n))$. On the other hand, as repetitiveness is destroyed at deeper levels, the total space is far from that of a plain Re-Pair compression of S.

3 Efficient `rsa` for Sequences on Small Alphabets

Our first proposal, dubbed `GCC` *(Grammar Compression with Counters)* is aimed at solving `rsa` queries on grammar-compressed sequences with small alphabets. We generalize the existing solution for bitmaps (`RPB`, Section 2.2), to sequences with $\sigma > 2$. Besides, we introduce several enhancements that improve its space usage.

Let (R, C) be the result of a balanced Re-Pair grammar compression of S. We store $S_\ell[X] = \ell(X)$ for each grammar rule $X \in R$. In addition, we store a sequence of counters $S_a[X]$ for each symbol $a \in \Sigma$: $S_a[X] = rank_a(exp(X), \ell(X))$ 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 $lrnk[a] = rank_a(S, L(p) - 1)$ for all $a \in \Sigma$, s being the sampling period.

The `rsa` algorithms stay practically the same as for `RPB`; now we use the symbol counter of a for $rank_a$ and $select_a$. The resulting data structure solves `rsa` in time $O(s + \log n)$ and takes $O(r\sigma \log n + \sigma(n/s) \log n) + c \lg(\sigma + r)$ bits.

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 [7]), 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 increasing. We reduce their space using a two-layer strategy: we sample S_n at regular intervals of length ss. We store $SS_n[j] = S_n[j \cdot ss]$, 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 $SS_n[\lfloor i/ss \rfloor] = (p^*, o^*, lrnk^*[1, \sigma])$. The total space for the p and $lrnk[1, \sigma]$ components is $O(\sigma(n/s) \log(s \cdot ss) + (n/(s \cdot ss)) \log n)$ bits. For example, if we use $ss = \lg n$ and $s = \log^{O(1)} n$ (a larger value would imply an excessively high query time), the space becomes $O(r\sigma \log n + \sigma(n/s) \log \log n) + c \lg(\sigma + r)$ bits. This can be reduced to $O((r\sigma + c) \log n)$ bits by sampling regularly C instead of S and using $s = \Theta(\log n)$, but the described sampling works better in practice.

When σ is small, this data srtucture is very space- and time-efficient. It compress better than `WTRP` [28] since it does not destroy the repetitiveness of S when building the wavelet tree. Besides, it runs faster compared to the $O(\log \sigma \log n)$ time obtained by `WTRP`: we need just one operation on `GCC`, not $\log \sigma$ operations on `RPB`. However, this solution becomes prohibitive when the alphabet becomes large since it has a σ multiplicative term in the space.

4 Efficient rsa for Sequences on Large Alphabets

For large alphabets, our idea is to combine GCC with AP [3] (Section 2.3), which splits $S[1,n]$ into a sequence $K[1,\lg\sigma]$ of classes and $\lg\sigma$ subsequences $S_{[1,\log\sigma]}$. That is, AP partitions the original sequence into subsequences over smaller alphabets, which is the scenario GCC handles well.

Note that, if S is grammar-compressible, then K is grammar-compressible as well, as K consists of a (non-injective) mapping of the symbols of S. It is also reasonable to expect that the subsequences S_j grammar-compress well, at least for the first levels (i.e., the most frequent symbols): If ab is a frequent pair in S, then it is expected that they are frequent individually as well. As a consequence, it is likely that a and b belong to the same first classes. Even for the less frequent symbols, if they appear frequently together, then their individual frequencies are likely to be similar, and thus they have a good chance to be assigned to the same class. If the most frequent pairs of symbols ab are assigned to the same subsequence S_j, then all the space saved by the rule $X \to ab$ is also saved if choosing the same rule when grammar-compressing S_j.

We apply GCC to K and to the first sequences S_j, since they have a small alphabet. For the remaining subsequences we have two choices: (a) represent them using GMR (APRep, recall that subsequences S_j are not statistically compressible [3]); or (b) attempt to grammar-compress them using WMRP (APRep-WMRP). Which is better depends on whether the subsequences on large alphabets (which contain less frequent symbols) are still repetitive or not. While the choice (b) yields higher times than (a), we note that, if queries have the same statistical distribution of the symbols in S, then most queries will refer to more frequent symbols, which will be handled with the fast GCC representation.

5 Experimental Results and Discussion

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB 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 4.6.3, with -O9 optimization. We implemented our solutions inside LIBCDS (github.com/fclaude/libcds) and use Navarro's implementation of Re-Pair (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 Re-Pair (RP, assuming $(2r+c)\lceil\lg(\sigma+r)\rceil$ bits), and bps obtained by p7zip (LZ, www.7-zip.org), a Lempel-Ziv compressor.

5.1 Results for Small Alphabets

To test our structure on small alphabets, we use some DNA datasets (para and escherichia) from *PizzaChili Repetitive Corpus* (pizzachili.dcc.uchile.cl/repcorpus), and influenza from the *SuDS Project* (www.cs.helsinki.fi/group/suds/rlcsa/data/fiwiki.bz2). From *SuDS* we also extract fiwikitags, the sequence of opening and closing tags from a subset of the Finnish Wikipedia.

Table 1. Statistics of the datasets. Length n is measured in millions (and rounded).

dataset	n	σ	H_0	RP	LZ	dataset	n	σ	H_0	RP	LZ
para	429	5	1.12	0.37	0.19	software	37	48	3.23	0.08	0.47
influenza	322	16	1.98	0.23	0.15	einstein	17	8,046	9.91	0.08	0.04
escherichia	113	15	2.00	1.04	0.52	fiwiki	84	99,797	11.04	0.24	0.16
fiwikitags	49	24	3.36	0.11	0.32	indochina	50	685,100	13.94	0.88	0.32

We show results for GCC, using sampling steps $s = 2^{\{10,12,14\}}$ and supersteps $ss = 2^{\{4,6,8\}}$ for C, and $\delta = \{0, 2, 4, 8\}$ for R. We also compare WMRP [28], which takes, for the bitmap of each level, the representation using least space between RPB, RRR (with sampling value 32), and CM (a simple implementation [17] with sampling value 32). We also include in the comparison the WTH (Huffman-compressed WT), as a good statistically compressed solution for rsa. For the WTH bitmaps we use RRR with sampling steps in $\{32, 64, 128\}$.

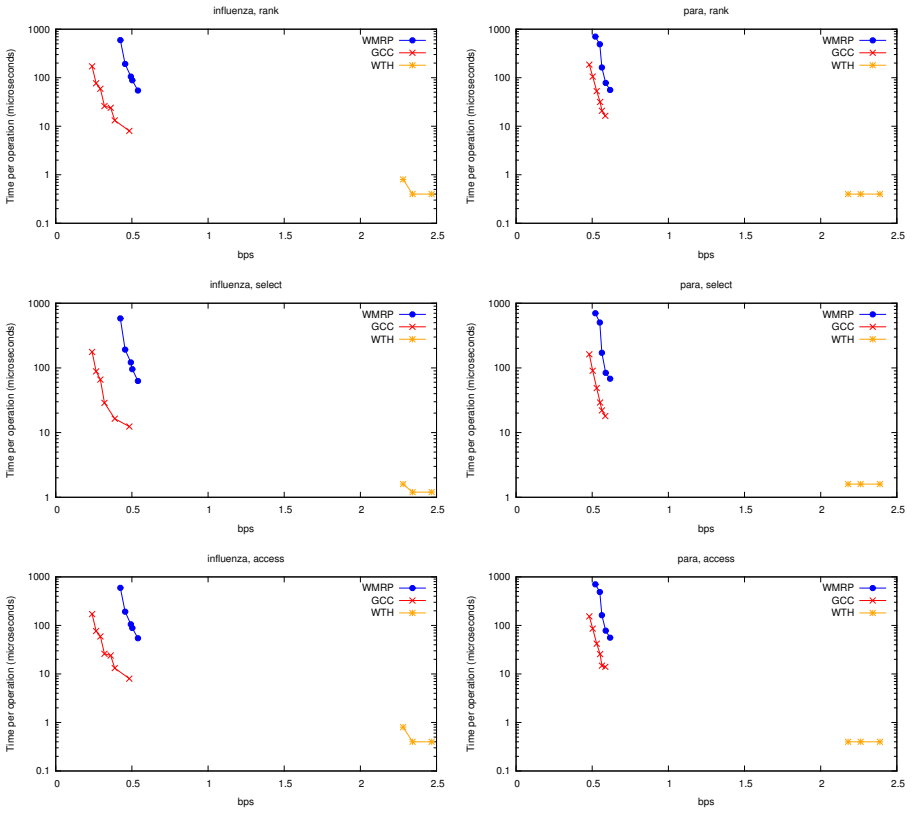

Fig. 1. Space-time tradeoffs for rsa queries over small alphabets: collections influenza and para (note logscale in time)

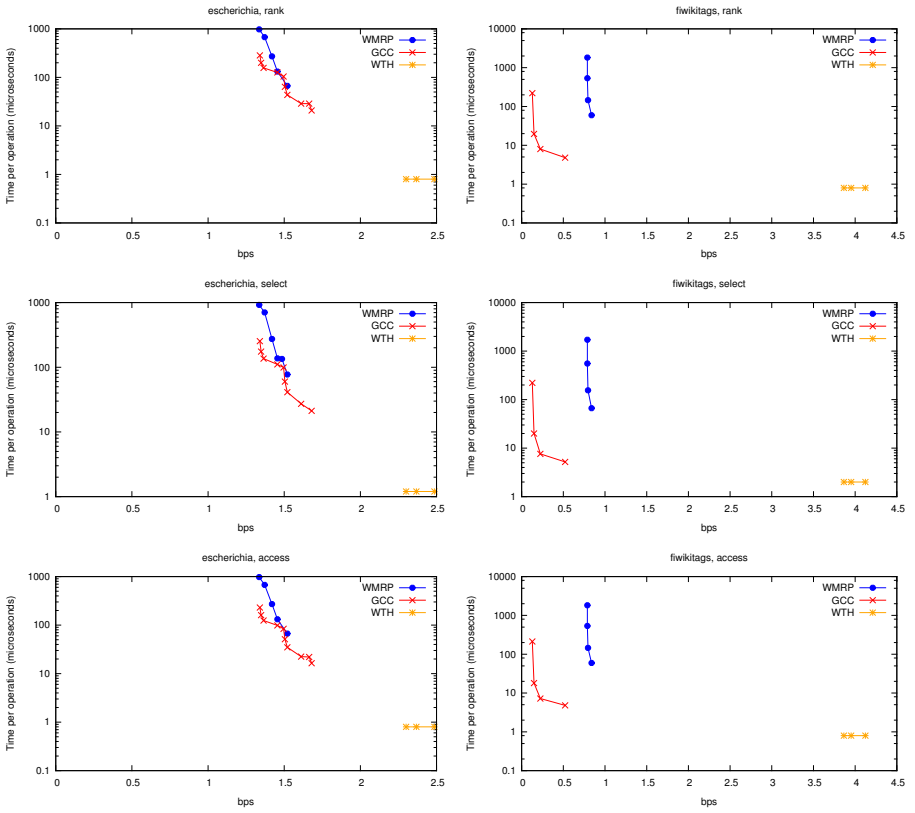

Fig. 2. Space-time tradeoffs for `rsa` queries over small alphabets: collections `escherichia` and `fiwikitags` (note logscale in time)

Figures 1 and 2 show the results for all the operations and collections. Our `GCC` dominates `WMRP` both in space and in `rsa` time. The difference in space with `WMRP` is larger as the sequence is more grammar-compressible (see Table 1). This is because `GCC` preserves all the repetitiveness of S, while paying a price only in terms of the alphabet size. Instead, `WMRP` destroys the repetitiveness after a few wavelet tree levels. In terms of `rsa` performance, `GCC` is up to two orders of magnitude faster than `WMRP` for the same space usage. Note that the collection in which `GCC` and `WMRP` are closest is `escherichia`, the least repetitive one.

On the other hand, the representation that compresses statistically, `WTH`, is about an order of magnitude faster than `GCC`, but it also takes many times more space (up to 10 times in case of `fiwikitags`).

5.2 Results for Large Alphabets

For large alphabets, we use collection `einstein` (also from *PizzaChili*), which contains Wikipedia versions of the article about Einstein in German, and `fiwiki`

(also from *SuDS*), a 400MB prefix of the Finnish Wikipedia. We regard both texts as sequences of words. A third collection is `indochina`, a subset with the first 50 million elements of the adajacency lists of the Web graph `Indochina2004` (available from the *WebGraph Project*, `http://law.dsi.unimi.it`).

We study our two solutions, `APRep` and `APRep-WMRP`. These use `GCC` and `WMRP` internally, for which we use the same configurations as for the case of small alphabets. Besides, we introduce two new parameters: $\beta \in \{2, \ldots, 10\}$, so that the β most frequent symbols are directly stored in K [3], and $f \in \{2, \ldots, 7\}$, so that we use `GCC` on the first f subsequences, $S_1, \ldots, S_f$. We compare these solutions with `WMRP`, parameterized as before, and with `WMH` and `AP`, two good statistically-compressing representations for large alphabets. We use `RRR` [29] for the the `WMH` bitmaps with samplings $\{32, 64, 128\}$. The K sequence of `AP` is represented with a `WT` and each S_j with `GMR`.

Figures 3 and 4 show the results. `APRep-WMRP` obtains the best space, dominating `WMRP` in both space and time by a significant margin. `APRep` takes over when more space is used, being up to twice as fast as `APRep-WMRP` (yet using twice the space). The statistical representations are, as before, up to an order of magnitude faster than our fastest representations, but use much more space, especially on the most repetitive collections. In those, they are two orders of magnitude faster, but use up to 10 times more space, than our most space-efficient representations.

5.3 Application: XPath Queries on Highly Repetitive Collections

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 solves 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 [26] 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, using some sequence representation.

When the XML collection is repetitive (e.g., versioned collections like Wikipedia, versioned software repositories, etc.), one can use the `RLCSA` [22], a full-text index that performs well on a repetitive collection of text nodes, for (1). 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 compare `GCT`, a tree representation aimed at repetitive topologies [27], with 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. All variants will use the `RLCSA` with no text sampling as their text index.

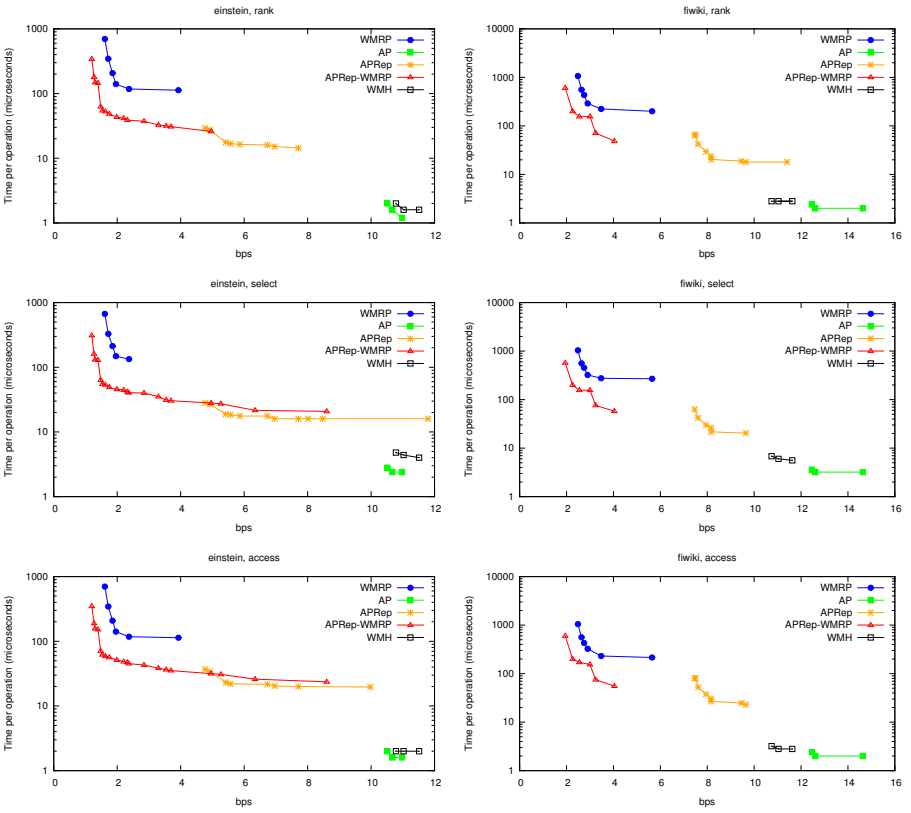

Fig. 3. Space-time tradeoffs for `rsa` queries over large alphabets: collections `einstein` and `fiwiki` (note logscale in time)

We use a repetitive data-centric XML collection of 200MB from a real software repository. Its sequence of XML tags, called `software`, is described in Table 1. 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 here is an opening or a closing tag, so there are two elements per XML tree node. The space of the `RLCSA` is always 2.3 bits per character of the XML document. The table also shows the impact of each component in the total size of the index. Finally, the table shows the time to solve both queries.

The original `SXSI` (`MATRIX+FF`) is very fast but needs almost 14 bpe, which amounts to over 75% 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 4–5, becoming 13–28 times slower than the original `SXSI`. Finally, changing `FF` by `GCT` [27], we can

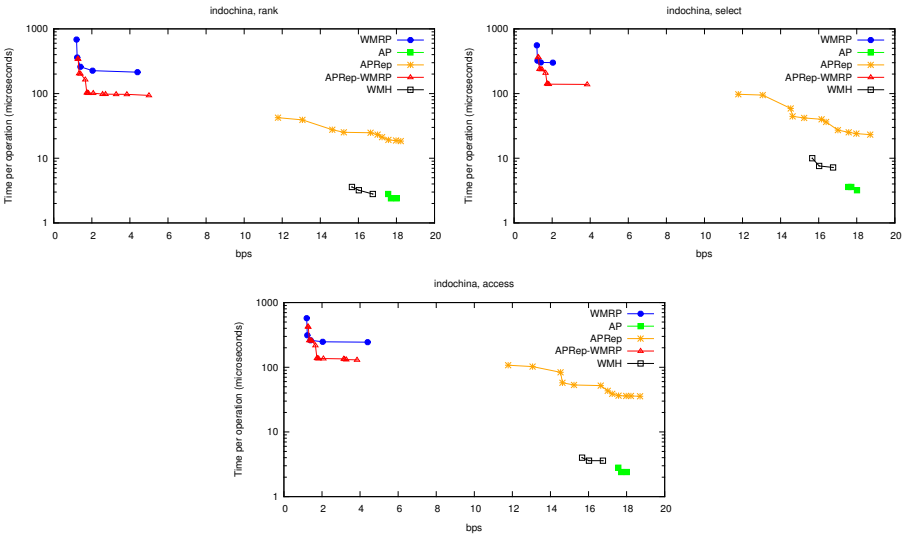

Fig. 4. Space-time tradeoffs for `rsa` queries over large alphabets: collection `indochina` (note logscale in time)

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	69.00	7.19	23.90	16	35
`WTH+FF`	2.88	1.27	34.07	15.09	50.84	92	113
`GCC+FF`	0.37	1.27	6.26	21.45	72.29	442	462
`GCC+GCT`	0.37	0.19	7.66	3.93	88.42	1,032	3,302

reach as low as 0.56 bpe, 24 times less than the original `SXSI`, and using less than 12% of the total space. Once again, the price is the time, which becomes 65–95 times slower than the basic `SXSI`. The price of using the slower `GCT` is more noticeable on XQ2, which requires 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. On the other hand, the time differences will blur on queries that do not only access the tags and the tree, but also involve the text, as these cost the same in all the representations. Finally, we note that the `RLCSA` becomes the space bottleneck in `GCC+GCT`. It is worthwhile to consider even more compressed text representations, for example based on grammars [11] or on LZ77 [20].

6 Final Remarks

Our new ideas permit much more exploration. We have used the same partitioning into sequences given in the alphabet partitioning work [3], with alphabets of doubling sizes. However, other partitionings may be more suitable to our needs, for example building all the subsequences with the same alphabet size ρ, so that alphabet $[1, \rho]$ can be comfortably handled with our basic method for small alphabets. This may induce a hierarchy of classes, instead of two levels as in alphabet partitioning [3]. The result would be indeed a ρ-ary version of the current (2-ary) wavelet-tree based solution [28], which may reduce space and time by increasing the arity. Furthermore, we plan to study heuristics for grouping symbols into classes, aiming to avoid separating symbols that form long repeated substrings, so that fewer repetitions are destroyed when forming the classes.

A more far-fetched goal is to achieve Lempel-Ziv compressed representations that support these operations. Lempel-Ziv is more powerful than grammar compression, but thought to be harder to handle even for supporting direct access.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. ALENEX, pp. 84–97 (2010)
2. D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, and N. Välimäki. Fast in-memory xpath search over compressed text and tree indexes. In: Proc. 26th ICDE, pp. 417–428 (2010)
3. Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. Algorithmica 69(1), 232–268 (2014)
4. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 181–192. Springer, Heidelberg (2012)
5. Bille, P., Landau, G., Raman, R., Sadakane, K., Rao Satti, S., Weimann, O.: Random access to grammar-compressed strings. In: Proc. 22nd SODA, pp. 373–389 (2011)
6. Brisaboa, N., Fariña, A., Ladra, S., Navarro, G.: Implicit indexing of natural language text by reorganizing bytecodes. Inf. Retr. 15(6), 527–557 (2012)
7. Brisaboa, N., Ladra, S., Navarro, G.: DACs: Bringing direct access to variable-length codes. Inf. Proc. Manag. 49(1), 392–404 (2013)
8. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theor. 51(7), 2554–2576 (2005)
9. Clark, D.: Compact Pat trees. PhD thesis, Univ. of Waterloo, Canada (1998)
10. Claude, F., Navarro, G.: Extended compact web graph representations. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Ukkonen Festschrift 2010. LNCS, vol. 6060, pp. 77–91. Springer, Heidelberg (2010)
11. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, LNCS 7608, pages 180–192, 2012.
12. Claude, F., Navarro, G.: The wavelet matrix. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 167–179. Springer, Heidelberg (2012)

13. Claude, F., Navarro, G., Ordóñez, A.: The wavelet matrix: An efficient wavelet tree for large alphabets. Information Systems (to appear, 2014)
14. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: LZ77-based self-indexing with faster pattern matching. In: Pardo, A., Viola, A. (eds.) LATIN 2014. LNCS, vol. 8392, pp. 731–742. Springer, Heidelberg (2014)
15. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comp. Sci. 426-427, 25–41 (2012)
16. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
17. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. 4th WEA, pp. 27–38 (2005)
18. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
19. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the I.R.E. 40(9), 1098–1101 (1952)
20. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comp. Sci. 483, 115–133 (2013)
21. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
22. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comp. Biol. 17(3), 281–308 (2010)
23. Munro, I.: Tables. In: Proc. 16th FSTTCS, pp. 37–42 (1996)
24. Navarro, G.: Indexing highly repetitive collections. In: Smyth, B. (ed.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
25. Navarro, G.: Wavelet trees for all. J. Discr. Alg. 25, 2–20 (2014)
26. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
27. Navarro, G., Ordóñez, A.: Faster compressed suffix trees for repetitive text collections. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 424–435. Springer, Heidelberg (2014)
28. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical compressed document retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)
29. Raman, R., Raman, V., Srinivasa Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), article 43 (2007)
30. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discr. Alg. 3(2-4), 416–430 (2005)
31. Tabei, Y., Takabatake, Y., Sakamoto, H.: A succinct grammar compression. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 235–246. Springer, Heidelberg (2013)
32. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 247–258. Springer, Heidelberg (2013)