HELDA
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

UNIVERSITY OF HELSINKI

https://helda.helsinki.fi

Block trees

Belazzougui, Djamal

2021-05

Belazzougui , D , Caceres , M , Gagie , T , Gawrychowski , P , Kaerkkaeinen , J , Navarro , G , Ordonez , A , Puglisi , S J & Tabei , Y 2021 , ' Block trees ' , Journal of Computer and System Sciences , vol. 117 , pp. 1-22 . https://doi.org/10.1016/j.jcss.2020.11.002

http://hdl.handle.net/10138/350861
https://doi.org/10.1016/j.jcss.2020.11.002

cc_by_nc_nd
acceptedVersion

Downloaded from Helda, University of Helsinki institutional repository.

This is an electronic reprint of the original article.

This reprint may differ from the original in pagination and typographic detail.

Please cite the original version.

Block Trees*

Djamal Belazzougui[1], Manuel Cáceres[2], Travis Gagie[3], Paweł Gawrychowski[4], Juha Kärkkäinen[5], Gonzalo Navarro[2], Alberto Ordóñez[6], Simon J. Puglisi[5], and Yasuo Tabei[7]

[1] DTISI-CERIST, Algeria
[2] Center for Biotechnology and Bioengineering (CeBiB) and Department of Computer Science, University of Chile, Chile
[3] Faculty of Computer Science, Dalhousie University, Canada
[4] University of Wroclaw, Poland
[5] Helsinki Institute for Information Technology (HIIT) and Department of Computer Science, University of Helsinki, Finland
[6] Pinterest Inc., USA
[7] PRESTO, Japan Science and Technology Agency, Japan

Abstract

Let string $S[1..n]$ be parsed into z phrases by the Lempel-Ziv algorithm. The corresponding compression algorithm encodes S in $\mathcal{O}(z)$ space, but it does not support random access to S. We introduce a data structure, the *block tree*, that represents S in $\mathcal{O}(z\log(n/z))$ space and extracts any symbol of S in time $\mathcal{O}(\log(n/z))$, among other space-time tradeoffs. The structure also supports other queries that are useful for building compressed data structures on top of S. Further, block trees can be built in linear time and in a scalable manner. Our experiments show that block trees offer relevant space-time tradeoffs compared to other compressed string representations for highly repetitive strings.

1 Introduction

Much of the fastest-growing data these days is highly repetitive: versioned document and software repositories like Wikipedia and GitHub store tens of versions of each document; whole-genome sequencing projects generate thousands of genomes of individuals of the same species; periodic astronomical surveys regularly scan the same portion of the sky. Such repetitiveness makes those large datasets highly compressible with dictionary methods like grammar-based or Lempel-Ziv compression, whereas typical statistical compression fails to capture the repetitiveness [KN13].

Lempel-Ziv compression [LZ76] of a string $S[1..n]$ parses S into a sequence of z "phrases", where each phrase $S[i..j]$ is a new symbol (and $j = i$) or it appears leftwards in S. Lempel-Ziv compression takes $\mathcal{O}(n)$ time [RPE81] and reduces S to $\mathcal{O}(z)$ space by encoding the phrases. While Lempel-Ziv is the practical method that best exploits repetitiveness, it has the problem that no way is known to access arbitrary substrings of S without decompressing it from the beginning.

All the previous work in the literature [BLR+15, BPT15, BC17, BEGV18, GNP20] resorts to grammar-based compression when it comes to provide direct access to compressed highly repetitive

*Supported in part by Basal Funds FB0001 and Fondecyt Grant 1-200038, Conicyt, Chile, and by the Academy of Finland grants 258308 and 268324. An early partial version of this work appeared in *Proc. DCC'15* [BGG+15].

strings. Grammar-based compression [KY00] of S consists in generating a context-free grammar that generates S and only S. When S is repetitive, the size g of the grammar can be much smaller than n. While finding the smallest grammar that generates S is NP-complete [Ryt03, CLL+05], there are several linear-time approximations that generate grammars of size $g = \mathcal{O}(z \log(n/z))$ [Ryt03, CLL+05, Jeż16]. Grammars, however, do not compress as well as Lempel-Ziv in general: it always holds that $g \geq z$ [Ryt03, CLL+05], and there are string families like $S = a^n$ where $z = \mathcal{O}(1)$ and $g = \Theta(\log n)$. Yet, grammars permit access to arbitrary positions of S in time $\mathcal{O}(\log n)$ and space $\mathcal{O}(g)$; this is not known to be possible within space $\mathcal{O}(z)$.

In this article we introduce the *block tree*, a data structure that uses $\mathcal{O}(z \log(n/z))$ space and supports direct access to any symbol of S in time $\mathcal{O}(\log(n/z))$. It allows other space-time tradeoffs, for example it can use $\mathcal{O}(z^{1-\epsilon} n^{\epsilon})$ space for any constant $\epsilon > 0$ and extract any substring of length m in RAM-optimal time $\mathcal{O}(1 + m/\log_\sigma n)$, where σ is the alphabet size of S. This is the first structure offering constant-time access to S within $\mathcal{O}(z^{1-\epsilon} n^{\epsilon})$ space.

Several more sophisticated objects can be represented in compressed form by resorting to strings [Nav16]. These representations generally need not only direct access to S, but (at least) the following two fundamental operations:

$$S.\text{rank}_a(i) = \text{return the number of occurrences of the character } a \text{ in } S[1..i].$$
$$S.\text{select}_a(j) = \text{return the position of the } j\text{th occurrence of } a \text{ in } S.$$

Such operations can also be supported on grammar-compressed strings in time $\mathcal{O}(\log n)$, by multiplying the space by $\mathcal{O}(\sigma)$ [BPT15, ONB17]. The same holds with block trees, where we can support rank and select in space $\mathcal{O}(z\sigma \log(n/z))$ and time $\mathcal{O}(\log(n/z))$, among other tradeoffs.

Block trees can be built in linear time and space. We also explore other scalable techniques that are relevant when S is very large and we cannot afford such extra space.

Finally, we show experimentally that block trees are a practical alternative to grammar-based representations. As expected from the theoretical results, block-tree-based representations are significantly (typically an order of magnitude) faster than grammar-based ones at the expense of sometimes being larger. On highly repetitive strings, however, block trees still offer an extremely compact representation with very fast direct access and rank/select support, competitive in time with statistically compressed representations, which are many times larger.

This article is a revised and expanded version of its conference publication [BGG+15], where in particular we have optimized the space of the data structure, removed incorrect claims about lowest-common-ancestor functionality, written the results with more detail, and carried out a much more extensive implementation and experimental comparison of different practical variants. The block tree idea derives from the earlier concept of block graph [GGP11, GGP15], which is more complex and has less functionality and worse time complexities.

2 Related Work

Random access to a statistically compressed sequence $S[1..n]$ over alphabet $[1..\sigma]$ can be obtained by adding $o(n \log \sigma)$ bits on top of its high-order empirical entropy [GN06, SG06, FV07], while

allowing the extraction of any substring of length m in the RAM-optimal time $\mathcal{O}(1+m/\log_\sigma n)$.[1] Any of those encodings can be enhanced with $o(n\log\sigma)$ further bits that provide efficient rank and select support [BHMR11]. However, statistical compression is very far from what can be obtained on highly repetitive strings [KN13].

A powerful compression technique to exploit repetitiveness is to find a context-free grammar that generates (only) S [KY00]. If S is parsed by Lempel-Ziv [LZ76] into z phrases, then various algorithms [Ryt03, CLL$^+$05, Jeż16] produce grammars of size $g = \mathcal{O}(z\log(n/z))$. Then, a data structure of size $\mathcal{O}(g)$ allows us to extract any substring of S of length m in time $\mathcal{O}(\log n + m)$ [BLR$^+$15], and even $\mathcal{O}(\log n + m/\log_\sigma n)$ [BPT15]. This $\mathcal{O}(\log n)$ additive penalty is nearly optimal [VY13]. The latter structure [BPT15] also offers $\mathcal{O}(\log_\tau n + m/\log_\sigma n)$ extraction time using $\mathcal{O}(g\tau\log_\tau(n/g))$ space, for any τ. It can also support rank and select operations in $\mathcal{O}(\log n)$ time by increasing the space to $\mathcal{O}(g\sigma)$, or in $\mathcal{O}(\log_\tau n)$ time using space $\mathcal{O}(g\sigma\tau\log_\tau(n/g))$. Belazzougui et al. [BPT15] give some evidence suggesting that the σ factor that multiplies the space cannot be avoided, but clever implementations can reduce its impact [ONB17].

Those results are easier to obtain if we use a balanced grammar, that is, of height $\mathcal{O}(\log n)$. For example, Bille et al. [BEGV18] use a particular grammar of size $\mathcal{O}(z\log(n/z))$ and height $\mathcal{O}(\log(n/z))$ to support extraction of a substring of length m in time $\mathcal{O}(\log(n/z)+m)$. A recent result [GJL19] is that any grammar of size g can be made balanced while retaining $\mathcal{O}(g)$ size.

Another relevant measure of repetitiveness is the number r of equal-letter runs in the Burrows-Wheeler Transform [BW94] of S. Gagie et al. [GNP20] introduce a representation of S using $\mathcal{O}(r\log(n/r))$ space that extracts any substring of S of length m in time $\mathcal{O}(\log(n/r)+m/\log_\sigma n)$. Their result in fact relies on building a grammar of size $\mathcal{O}(r\log(n/r))$ and height $\mathcal{O}(\log(n/r))$.

Yet another measure of repetitiveness is the size e of the smallest compact automaton that recognizes the substrings of S, or CDAWG [BBH$^+$87]. Belazzougui and Cunial [BC17] show how to extract substrings of length m in time $\mathcal{O}(\log n + m/\log_\sigma n)$. Once again, they exploit the fact that the CDAWG induces a balanced grammar of size $\mathcal{O}(e)$.

As it can be seen, every single method for substring extraction and rank/select support on highly repetitive strings builds on context-free grammars. Our block trees, which actually predate many of those results [BGG$^+$15], take an original approach, being the only ones that build directly on the Lempel-Ziv parse of S. They obtain $\mathcal{O}(z\log(n/z))$ space and offer similar time bounds. Interestingly, they do not need to actually build the Lempel-Ziv parse to obtain most of the results. Several other structures then followed the block-tree design; they are mentioned in the Conclusions as a sort of epilogue.

3 Block Trees

Let $S[1..n]$ be a string over alphabet $[1..\sigma]$, and τ and s be integer parameters larger than 1. Assume for simplicity that n is of the form $s\cdot\tau^h$ for some integer h. The block tree is a perfectly balanced tree of height h, which may however have leaves at higher levels. The root has s children and every other internal node has τ children. Leaves in the last level store symbols or short substrings of S; the others store special leftward pointers to nodes or pairs of nodes in the same level.

[1] The minimum time to write the output, m symbols of $\log\sigma$ bits each, even if packing $\log_\sigma n$ consecutive symbols into words of $\log n$ bits, is $\Omega(1+m/\log_\sigma n)$.

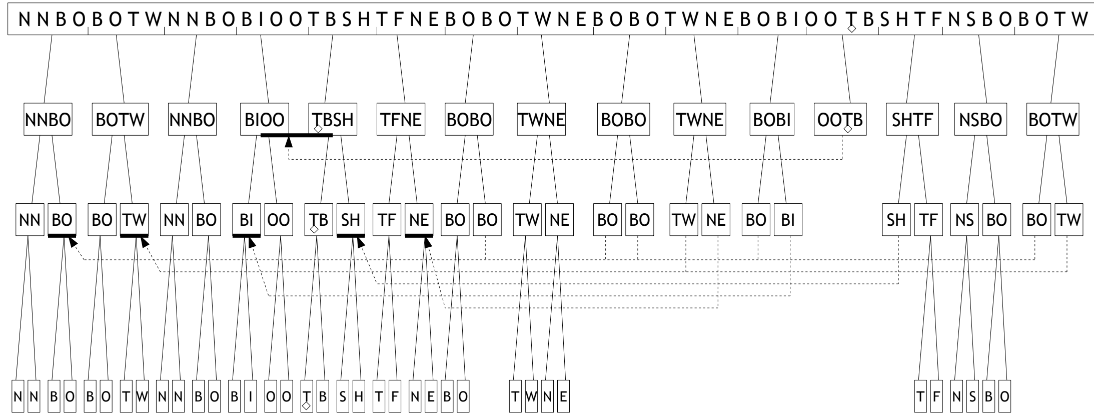

Figure 1: A block tree with $s = 15$, $\tau = 2$, and leaves with size 1, for NNBOBOTW... NSBOBOTW (from "99 Bottles of Beer on the Wall"): boxes are nodes; solid lines are edges and dashed lines are leftward pointers to blocks' first occurrences; only the symbols at the last level are stored. The 'T' symbols decorated with a $\diamond$ illustrate an access. Notice we mark the third child of the root and its children, even though NNBO occurs earlier, because TWNN and BOBI do not. Also, although BISH occurs only once, we do not mark the 22nd and 23rd blocks at the third level, since they are not consecutive in the original string.

Each node u represents a "block" B^u, which is a substring of S with an identified position in S. The root node represents the whole S and has s children representing the consecutive blocks of S of length n/s. Let ℓ_u be the level (or depth) of node u; the level of the root is zero. Nonroot nodes u that are not in the last level can be of two types: an internal node with τ children, or a leaf with leftward pointers. Which is the case depends on a global criterion, defined as follows.

Let $B_1, B_2, \ldots$ be the blocks represented by all the nodes of level ℓ_u, left to right. For any i, if B_i and B_{i+1} are consecutive in S and they form the leftmost occurrence in S of their content, $B_i \cdot B_{i+1}$, — which is the case if $B_i \cdot B_{i+1}$ contains the leftmost occurrence of *any* substring of S — then we say that the two nodes that represent B_i and B_{i+1} are *marked*.

If u turns out not to be marked, then it is an internal node with τ children, which represent the consecutive blocks of length $|B^u|/\tau$ into which B^u can be divided. Otherwise, u is a leaf storing (1) special pointers to the two consecutive nodes v_i and v_{i+1} of the same level whose blocks, B_i and B_{i+1}, cover the leftmost occurrence of B^u in S, and (2) the offset of that occurrence of B^u inside $B_i \cdot B_{i+1}$. Note that the nodes v_i and v_{i+1} must exist and be marked, because they — and their ancestors — contain the leftmost occurrence of B^u.

Note that $|B^u| = n/(s\tau^{\ell_u - 1})$ for nonroot nodes u. The last level is formed when $|B^u|$ is small enough, that is, when the cost of explicitly storing B^u becomes less than that of storing the leftward pointers of a leaf — i.e., when $|B^u| = \Theta(\log_\sigma n)$ and thus B^u can be encoded in $\mathcal{O}(\log n)$ bits. Taking the constant as 1 for simplicity, leaves store exactly $\log n$ bits and the height of the block tree is $h = \log_\tau \frac{n \log \sigma}{s \log n}$. Note that increasing s decreases the height at the cost of adding $\mathcal{O}(s)$ words to the space. Reducing the height h matters because the time complexities of the operations are proportional to h, as seen later.

For strings of arbitrary length, we round their length n up to the next integer of the form $s \cdot \tau^h$ for an integer h, by adding dummy symbols \$ at the end of S and avoiding storing rightmost blocks formed only of \$s. Figure 1 shows an example of a block tree.

4 Space Analysis

We state the key result that relates the size of the block tree to the Lempel-Ziv parse of S.

Lemma 1. *The number of blocks in any level of the block tree (except the first) is at most* $3z\tau$.

Proof. For any level ℓ other than the first, consider the level $\ell - 1$. Any concatenation of three consecutive blocks $C = B_{i-1} \cdot B_i \cdot B_{i+1}$ at level $\ell - 1$ not containing any phrase boundary from the Lempel-Ziv parse has, by the definition of such parsing, a leftward occurrence in S. Therefore, neither the pair $B_{i-1} \cdot B_i$ nor the pair $B_i \cdot B_{i+1}$ will be marked when building the block tree, and then B_i will remain unmarked. Said another way, the only blocks that can be marked are B_{j-1}, B_j, or B_{j+1} whenever a phrase boundary falls inside B_j. Since there are at most z such blocks B_j, and each marked block produces τ blocks in level ℓ, there can be at most $3z\tau$ blocks in level ℓ. □

By Lemma 1, there are $\mathcal{O}(z\tau)$ blocks per level. At any level ℓ except the last, each unmarked block is encoded using $\mathcal{O}(\log n)$ bits of space, for storing pointers of $\log n$ bits into leftward (marked) nodes of level ℓ. Each marked block requires $\mathcal{O}(\tau \log n)$ bits to point to its τ children in level $\ell + 1$, but we can charge each such pointer to the corresponding child. At the last level, each of the $\mathcal{O}(z\tau)$ blocks is simply encoded as plain text, that is, $\log_\sigma n$ symbols of $\log \sigma$ bits each, which yields again $\mathcal{O}(\log n)$ bits per block.

Since the number of levels is $h = \log_\tau \frac{n \log \sigma}{s \log n}$, we need $\mathcal{O}(s)$ pointers to the top-level blocks (even if only $\mathcal{O}(z)$ of them are marked) and $\mathcal{O}(z\tau)$ pointers for the other levels. Rounding up the length n may induce at most one extra block — that is, $\mathcal{O}(\tau)$ pointers — per level.

Theorem 2. *Given a string* $S[1..n]$ *over alphabet* $[1..\sigma]$ *and integers* τ *and* s*, we can build a block tree with* $\log_\tau \frac{n \log \sigma}{s \log n}$ *levels. The block tree occupies* $\mathcal{O}\left(\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right) \log n\right)$ *bits of space, where* z *is the number of phrases in the Lempel-Ziv parsing of* S.

A fairly large value of s that retains the minimum asymptotic space complexity is $s = z$, where the block tree is of height $\log_\tau \frac{n \log \sigma}{z \log n}$ and it uses $\mathcal{O}\left(z\tau \log_\tau \left(\frac{n \log \sigma}{z \log n}\right) \log n\right)$ bits of space. We note that $\frac{n \log \sigma}{z \log n}$ is actually the Lempel-Ziv compression factor. Using a constant value of τ yields the minimum space, $\mathcal{O}\left(z \log \frac{n \log \sigma}{z \log n}\right) \subseteq \mathcal{O}(z \log(n/z))$ (measured in $\Theta(\log n)$-bit words), and a logarithmic number of levels, $\mathcal{O}\left(\log \frac{n \log \sigma}{z \log n}\right) \subseteq \mathcal{O}(\log(n/z))$.

Another interesting setting for the arity is $\tau = \left(\frac{n \log \sigma}{z \log n}\right)^\epsilon$ for some constant $0 < \epsilon < 1$. This makes the space usage to be $(\mathcal{O}\left(\frac{1}{\epsilon}(z \log n)^{1-\epsilon}(n \log \sigma)^\epsilon\right)$ bits (at most $\mathcal{O}\left(\frac{1}{\epsilon} z^{1-\epsilon} n^\epsilon\right)$ words) and the number of levels the constant $\mathcal{O}(1/\epsilon)$. This space usage is a weighted geometric average between $z \log n$ (the space usage in bits achievable by the Lempel-Ziv parsing) and the uncompressed space $n \log \sigma$. The parameter ϵ allows us to give an arbitrarily large weight to the compressed space usage at the cost of increasing the number of levels (and thus query time).

5 Queries

The simplest query to answer with a block tree is to return a character $S[i]$ when given i. To do this, we start at the root and descend to the child whose corresponding block contains $S[i]$, then to the grandchild whose block contains $S[i]$, etc. If we reach a leaf u, then either u stores its block explicitly — we can then return $S[i]$ immediately — or u stores leftward pointers to the (internal) nodes in the same level whose blocks contain the leftmost occurrence in S of B^u, and the offset of that occurrence in those blocks. In the latter case, in $\mathcal{O}(1)$ time we can identify a character $S[i']$ in one of those leftward nodes' blocks such that $S[i'] = S[i]$, then start descending from that leftward node to find $S[i']$. Because we can traverse leftward pointers only once per level, returning $S[i]$ takes a total of $\mathcal{O}\left(\log_\tau \frac{n \log \sigma}{s \log n}\right)$ time, proportional to the height of the tree.

Theorem 3. *Given a string $S[1..n]$ over alphabet $[1..\sigma]$, the block tree with parameters τ and s built on S can extract any symbol $S[i]$ in time $\mathcal{O}\left(\log_\tau \frac{n \log \sigma}{s \log n}\right)$.*

For example, to return $S[47]$ with the block tree shown in Figure 1, we first descend to the twelfth child of the root, which is a leaf; since $S[47]$ is the third character in that child's block $S[45..58]$ and the first occurrence of that block is $S[15..18]$, we know $S[47] = S[17]$. We follow the pointer to the root's fifth child, with block $S[17..20]$; we then descend two more levels and eventually return `T`. The relevant characters are marked with diamonds.

5.1 Extracting substrings

The original paper on block graphs [GGP11] showed how to store S in $\mathcal{O}(z \log(n/z))$ space such that any substring of length m can be extracted in $\mathcal{O}(\log n + m)$ time. In this section we show how to extract an arbitrary substring from a block tree with h levels in $\mathcal{O}(h(1 + m/\log_\sigma n))$ time. Note that this is better than the $\mathcal{O}(hm)$ time we would obtain by simply applying m times the basic traversal procedure just described.

In order to achieve the improved bounds, we store the first and the last $\log_\sigma n$ symbols for every internal node. This adds $\log n$ bits per node and does not increase the space asymptotically. Note that the leaves already store their $\log_\sigma n$ symbols explicitly.

We extract a substring $S[i..i+m-1]$ with $m \le \log_\sigma n$ as follows. Since the smallest blocks in the tree are of length $\log_\sigma n$, $S[i..i+m-1]$ either falls inside the block of a single node or spans the blocks of two consecutive nodes.

At each level, if $S[i..i+m-1]$ spans two consecutive internal blocks, then we can extract the part of the string that lies in the first block in $\mathcal{O}(1)$ time, since we have stored the last $\log_\sigma n$ symbols of the block. The same goes for the part that lies in the second block, since we have stored the first $\log_\sigma n$ symbols of the block. If $S[i..i+m-1]$ is fully contained in a block, then we descend to the next level, either directly if the block is marked, or by following leftward pointers if the block is unmarked. Note that the new substring to extract after following a leftward pointer, $S[i'..i'+m-1] = S[i..i+m-1]$, may now span two blocks.

We continue recursively in this way, stopping either at the first level at which $S[i..i+m-1]$ spans two blocks, or when we reach the last level of the block tree (where the leaf stores its $\log_\sigma n$ symbols explicitly). Overall, the time spent is $\mathcal{O}(h)$.

To extract $S[i..i+m-1]$ when $m > \log_\sigma n$, we simply divide it into pieces of length $\log_\sigma n$ (the last may be shorter) and extract each piece separately.

Theorem 4. *Given a string $S[1..n]$ over alphabet $[1..\sigma]$ and integers τ and s, we can build a data structure occupying $\mathcal{O}\Big(\Big(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\Big) \log n\Big)$ bits of space, where z is the number of phrases in the Lempel-Ziv parsing of S, that allows extraction of any substring of S of length m in time*

$$\mathcal{O}\left(\left(1 + \frac{m}{\log_\sigma n}\right) \cdot \log_\tau \left(\frac{n \log \sigma}{s \log n}\right)\right).$$

Setting $s = z$ and $\tau = \left(\frac{n \log \sigma}{z \log n}\right)^\epsilon$ as before, for any constant $0 < \epsilon < 1$, we obtain RAM-optimal extraction time, $\mathcal{O}\big(\frac{1}{\epsilon}(1 + m/\log_\sigma n)\big)$, within $\mathcal{O}\big(\frac{1}{\epsilon}(z \log n)^{1-\epsilon}(n \log \sigma)^\epsilon\big)$ bits of space.

5.2 Rank

To efficiently support rank queries on S we store at each block tree node u, for each distinct character a, the number $u.\mathrm{pre}(a)$ of occurrences of a in the prefix of S preceding block B^u. This takes $\mathcal{O}\Big(\sigma\left(s + z\tau \log_\tau\left(\frac{n \log \sigma}{s \log n}\right)\right) \log n\Big)$ bits of space. This sample of rank values lets us turn any rank query on S into a rank query on a block in $\mathcal{O}(1)$ time: if $S[i]$ corresponds to $B^u[j]$, then $S.\mathrm{rank}_a(i) = u.\mathrm{pre}(a) + B^u.\mathrm{rank}_a(j)$. The values $u.\mathrm{pre}(a)$ also allow turning a rank query on a marked block for an internal node, into a rank query on one of its children, also in $\mathcal{O}(1)$ time. We also store information to turn any rank query on an unmarked block into a rank query on its pointed marked block in $\mathcal{O}(1)$ time. Finally, we store a classic binary rank data structure [Cla96, Nav16] for the concatenation of the strings in the blocks of the last level, which takes $\mathcal{O}(z\tau \log_\sigma n)$ bits of space per symbol a, so we can answer rank queries on those blocks directly in $\mathcal{O}(1)$ time, and can thus answer rank queries on S in $\mathcal{O}\Big(\log_\tau \frac{n \log \sigma}{s \log n}\Big)$ total time.

Let an unmarked block B^u point to node v_j representing block B_j or to consecutive nodes v_j and v_{j+1} representing blocks B_j and B_{j+1}, starting at offset $g+1$ inside B_j. The information to convert any rank query on B^u into a rank query on B_j or B_{j+1} is, for each character a, the number $B_j.\mathrm{rank}_a(g)$ of occurrences of a in the prefix of B_j that precedes the occurrence of B^u. Let $d = |B_j| - g$ be the length of the prefix of B^u that is a suffix of B_j. Since the number of occurrences of a in B_j is $v_{j+1}.\mathrm{pre}(a) - v_j.\mathrm{pre}(a)$, the number of occurrences of a in $B^u[1..d] = B_j[g+1..|B_j|]$ is $B^u.\mathrm{rank}_a(d) = v_{j+1}.\mathrm{pre}(a) - v_j.\mathrm{pre}(a) - B_j.\mathrm{rank}_a(g)$, which is computed in $\mathcal{O}(1)$ time. Then, if $i \leq d$, it holds that $B^u.\mathrm{rank}_a(i) = B_j.\mathrm{rank}_a(g+i) - B_j.\mathrm{rank}_a(g)$, and if $i > d$, it holds that $B^u.\mathrm{rank}_a(i) = B^u.\mathrm{rank}_a(d) + B_{j+1}.\mathrm{rank}_a(i-d)$. See Figure 2.

5.3 Select

To support select quickly on S we store, for each distinct symbol a, a predecessor data structure on the rank samples $v.\mathrm{pre}(a)$ of the s top-level nodes v. A predecessor query yielding a top-level block $B^v = S[i..i+|B^v|-1]$ then lets us translate the query $S.\mathrm{select}_a(j) = i - 1 + B^v.\mathrm{select}_a(j - v.\mathrm{pre}(a))$. We also store a similar structure on the values $v_k.\mathrm{pre}(a)$ of the τ children $v_1, \ldots, v_\tau$ of internal marked nodes v, to translate the query on v to its proper child in the same way. Analogously as for rank queries, we store a classical constant-time select data structure on the concatenation of

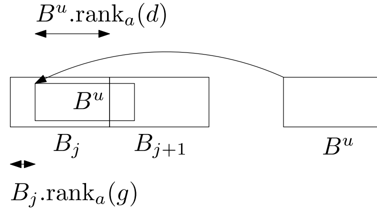

Figure 2: To turn a rank query on the unmarked block B^u into a rank query on one of the consecutive marked blocks B_j and B_{j+1} that contain B^u's first occurrence in S, which starts at relative position $g+1$ of B_i, we store B_i.rank$_a(g)$ in B^u and infer B^u.rank$_a(d)$ from the sampled ranks pre(a).

all the strings in the lowest-level blocks [Cla96, Nav17], so that within $\mathcal{O}(z\tau \log_\sigma n)$ extra bits per symbol a we handle the last level in constant time. Finally, the information we already have stored lets us turn any select query on an unmarked block B^u into a select query on its pointed marked blocks B_i and B_{i+1}: if $j \le B^u.\mathrm{rank}_a(d)$ then $B^u.\mathrm{select}_a(j) = B_i.\mathrm{select}_a(j + B_i.\mathrm{rank}_a(g)) - g$, and if $j > B^u.\mathrm{rank}_a(d)$ then $B^u.\mathrm{select}_a(j) = B_{j+1}.\mathrm{select}_a(j - B^u.\mathrm{rank}_a(d)) + d$.

Theorem 5. *Given a string $S[1..n]$ over alphabet $[1..\sigma]$ and integers τ and s, we can build a data structure occupying $\mathcal{O}\Big(\sigma\left(s + z\tau\log_\tau \frac{n\log\sigma}{s\log n}\right)\log n\Big)$ bits of space, where z is the number of phrases in the Lempel-Ziv parsing of S, that supports* rank *queries on S in time $\mathcal{O}\Big(\log_\tau \frac{n\log\sigma}{s\log n}\Big)$ and* select *queries on S in time $\mathcal{O}\Big(pred(s,n) + \log_\tau\left(\frac{n\log\sigma}{s\log n}\right)\cdot pred(r,n)\Big)$, where $pred(t,n)$ is the time of a predecessor query on t integers in the universe $[1..n]$ with a structure that uses $\mathcal{O}(t)$ space.*

Predecessor structures of size $\mathcal{O}(t)$ can achieve $pred(t,n) = \mathcal{O}(\log\log(n/t))$ [BN15, App. A], for example, but there are many other tradeoffs [PT06].

With constant τ, for example, the predecessor queries on internal nodes take constant time, so the structure uses $\mathcal{O}(\sigma z\log(n/z))$ space and supports rank and select in time $\mathcal{O}(\log(n/z))$.

Once again, if we use $s = z$ and $\tau = \left(\frac{n\log\sigma}{z\log n}\right)^\epsilon$ for any constant $0 < \epsilon < 1$, the structure requires $\mathcal{O}\big(\frac{1}{\epsilon}\sigma(z\log n)^{1-\epsilon}(n\log\sigma)^\epsilon\big)$ bits of space (i.e., $\mathcal{O}\big(\frac{1}{\epsilon}z^{1-\epsilon}n^\epsilon\big)$ words) and supports rank in constant time, $\mathcal{O}\big(\frac{1}{\epsilon}\big)$, and select in time $\mathcal{O}\big(\frac{1}{\epsilon}\log\log n\big)$. To obtain $\mathcal{O}\big(\frac{1}{\epsilon}\big)$ time also on select queries we can use $\tau = (n/z)^{\epsilon/2}$ and implement the predecessor queries as a trie with branching factor τ, so that the space is still $\mathcal{O}\big(\frac{1}{\epsilon}\sigma z^{1-\epsilon}n^\epsilon\big)$ words. Yet another interesting tradeoff is obtained with $s = \tau = \log^\epsilon n$ and using atomic heaps to solve predecessor queries in constant time. In this case, the space is $\mathcal{O}\big(\sigma z\log^{1+\epsilon} n\big)$ words and both rank and select are solved in time $\mathcal{O}\Big(\frac{\log n}{\epsilon\log\log n}\Big)$.

6 Construction

We now describe different strategies to build the block tree data structure on a string $S[1..n]$ in order to support the simple access of an isolated symbol. Adding the construction of the extra data to support faster access, as well as rank and select operations, is an easy exercise.

The core strategy is to identify all the marked blocks with a single pass over the text, and then the targets of the unmarked blocks with a second pass. The successive passes on the next levels process geometrically decreasing texts, so linear-time construction is obtained when $s = \Theta(z)$. To ensure worst-case times, an Aho-Corasick automaton can be used, though it requires $\mathcal{O}(n)$ extra space. This is a problem on very large strings, even if their final block tree is small. Instead, Karp-Rabin hashing yields the same expected times, with just $\mathcal{O}(s + zr)$ extra space. Further, this second construction can maintain the original string on disk and perform only a logarithmic number of passes over it. It then builds the block tree essentially in-place.

6.1 Worst-case-time construction using $\mathcal{O}(n)$ working space

We first partition S into s blocks $B_1, \ldots, B_s$ of lengths n/s, and create in $\mathcal{O}(n)$ time and space an Aho-Corasick automaton [AC75] that recognizes all the $s-1$ consecutive pairs $B_i \cdot B_{i+1}$. This automaton searches for all the patterns simultaneously in $\mathcal{O}(1)$ amortized time per scanned symbol.

We set a counter to zero for each block B_i. We then traverse S with the automaton. Every time we find an occurrence of a pair $B_i \cdot B_{i+1}$, if it is the first time we find the pair in S, we increase the counters of B_i and of B_{i+1}. After we scan S, all the blocks B_i with counter equal to 2 (and B_s, if its counter is equal to 1) are the unmarked blocks (because both $B_{i-1} \cdot B_i$ and $B_i \cdot B_{i+1}$ have earlier occurrences in S).

In the second stage, we destroy the first Aho-Corasick automaton and create a new one with the contents of the unmarked blocks B^u, and scan S again. The first time we find each B^u to be equal to a block B_i or contained in two consecutive blocks $B_i \cdot B_{i+1}$, we set pointers from B^u to the node of B_i and, if appropriate, to that of B_{i+1}, and also store in B^u the offset g where the copy of B^u starts within B_i. Recall that B_i and B_{i+1} are guaranteed to be marked because they contain the first occurrence of the string B^u, so they must also be the first occurrence of $B_i \cdot B_{i+1}$.

We now delete the second Aho-Corasick automaton and move on to the second level, where each marked block is split into τ sub-blocks, all of size $n/(s\tau)$. We repeat the process, but build the automata considering only the existing sub-blocks in this level. Sub-blocks that are surrounded by two sub-blocks need to reach the counter value 2 to be unmarked; the others need to reach the counter value 1. When we reach the final level, with blocks of length $\log_\sigma n$, we simply store the text corresponding to each block.

To analyze the construction time complexity, consider the following. The top-level blocks are of length n/s. If $s < 3z\tau$, then after level $\ell = 1 + \lfloor \log_\tau(3z/s) \rfloor$ the block lengths are below $n/(3z\tau)$. Up to level ℓ, the lengths of the existing blocks may add up to n per level, but after level ℓ, since there exist at most $3z\tau$ blocks per level (Lemma 1) and their lengths decrease exponentially, the sum of all the existing block lengths is $\mathcal{O}(n)$. In total, since each symbol is processed in constant amortized time, the process requires $\mathcal{O}(n(1 + \log_\tau(z/s))$ time and $\mathcal{O}(n)$ space. To this, we must add the time to generate the block tree, $\mathcal{O}\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right) \subseteq \mathcal{O}(n + z\tau \log_\tau(n/s))$. This is subsumed by our construction time, $\mathcal{O}(n \log_\tau(z\tau/s))$, as long as $z\tau = \mathcal{O}(n)$. Note that if $\tau > n/z$ then the block tree uses $\Omega(n \log n)$ bits (Theorem 2) and then it is better to represent S in plain form without building the block tree at all (i.e., we can stop building the block tree as soon as generating it exceeds our time budget; we know it will not compress).

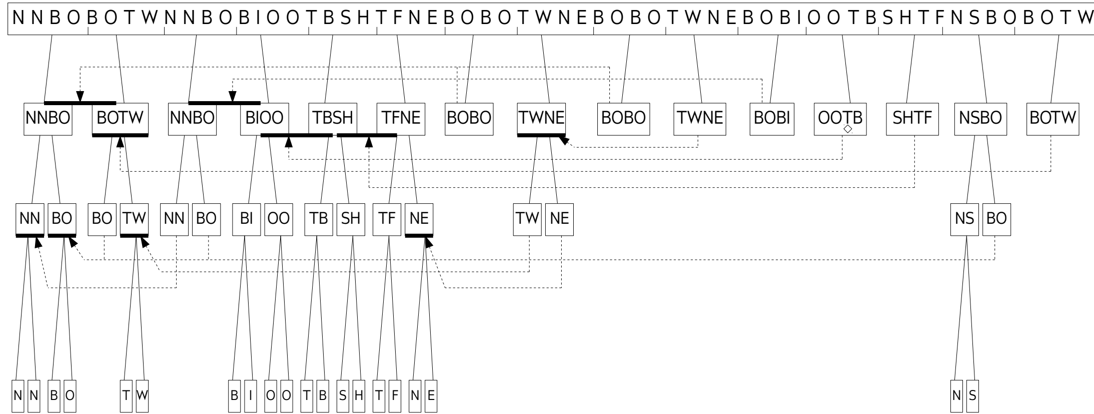

Figure 3: The pruned version of the block tree of Figure 1.

Pruning. While the structure we built satisfies the space guarantees of Theorem 2, it might hold more blocks than necessary. We mark the first occurrence of each pair of consecutive blocks $B_i \cdot B_{i+1}$, which ensures that any rightward block B^u appearing inside them can point to them without having unmarked blocks pointing to unmarked blocks. However, it might be that no rightward block B^u points to any of them, and that either or both B_i and B_{i+1} appear earlier in S. In this case, they could have been replaced by leftward pointers, thereby reducing space.

We then modify our construction and also carry out a postprocessing space optimization once the block tree is built. The modification is that, when we build the second automaton, we insert all the blocks of the level, not only the unmarked ones. For the marked blocks we also remember their leftmost occurrence found by the automaton, even if we do not yet replace them. The postprocessing consists of traversing the block tree in post-order, with the children in reverse order and the parent after the children. Initially, all the nodes have counter value zero. For each unmarked block B^u we reach in our traversal that points leftward to blocks B_i and (possibly) B_{i+1}, we increment the counters of B_i and (possibly) B_{i+1}. For each marked block we reach that has zero counter, its τ children are unmarked or last-level leaves storing explicit strings, and its leftmost occurrence in S does not overlap itself, we remove its children and make the block unmarked, creating the leftward pointer (those pointed blocks must then have their counter incremented, and the blocks pointed by the removed children must be decremented).

Figure 3 shows how the block tree of Figure 1 is pruned by this procedure. In Figure 1, we set all counters to zero and start at the rightmost child, representing BOTW. We then visit its children, TW and BO, which are unmarked, and increment the counters of their pointed blocks, the second and fourth of the third level. When returning to the parent BOTW, we note that its counter is zero, its two children are unmarked, and the leftmost occurrence of BOTW does not overlap the last block. We then remove its two children, decrease the counters we had just incremented, remove them, make the parent BOTW unmarked, make it point to its leftmost occurrence (the second block in the second level), and increment the counter of that newly pointed block. The process continues until obtaining the pruned block tree of Figure 3.

This postprocessing may remove a large number of unnecessary blocks from the tree. Another

obvious improvement is to remove the initial levels one by one as long as they do not have sufficient leftward pointers to reduce the space, since otherwise they only increase query times.

The postprocessing time, $\mathcal{O}\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right)$, is proportional to the size of the block tree, and thus it does not affect the construction complexity.

Theorem 6. *The block tree of any string $S[1..n]$ over alphabet $[1..\sigma]$, with integer parameters τ and s, can be built and optimized in $\mathcal{O}(n(1+\log_\tau(z/s)))$ worst-case time and $\mathcal{O}(n)$ working space.*

If we want to use the value $s = z$ in order to build the block tree in $\mathcal{O}(n)$ time, we can first determine z by carrying out the Lempel-Ziv parsing of S. This takes $\mathcal{O}(n)$ time and space [RPE81].

6.2 Expected-time construction with $\mathcal{O}(s + z\tau)$ working space

A problem with the previous construction is that using $\mathcal{O}(n)$ working space may be unfeasible on very large sequences. Such space is needed to build the Aho-Corasick automata, and we can avoid it by replacing Aho-Corasick searching with Karp-Rabin signatures [KR87]. We store in a universal hash table the signatures of all the pairs of blocks (first stage), or of the blocks (second stage), and then scan the text computing the fingerprint of all the windows, each in constant time. We must verify the strings with matching signatures, but with high probability (w.h.p.)[2] the strings match, and a matching block is never verified again because we have already found its leftmost occurrence. Therefore, at each level we spend $\mathcal{O}(n)$ time for scanning and $\mathcal{O}(n)$ expected time for hashing and verifications. The expected time becomes $\mathcal{O}(n)$ w.h.p. if we use a proper hashing scheme [Wil00]. The space per level is now proportional to the number of blocks in that level.

Overall, we build the block tree level by level, using only $\mathcal{O}(s)$ working space in the first level and $\mathcal{O}(z\tau)$ in the others, apart from storing S and the final block tree (without pruning), in $\mathcal{O}(n(1+\log_\tau(z/s)))$ expected time (and w.h.p.). We can then apply the pruning without requiring extra asymptotic space. The total construction space is $\mathcal{O}\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right)$ plus the $n \log \sigma$ bits to store S, that is, asymptotically dominated by the input plus the output upper bound.

Theorem 7. *The block tree of any string $S[1..n]$ over alphabet $[1..\sigma]$, with integer parameters τ and s, can be built in $\mathcal{O}(n(1+\log_\tau(z/s)))$ expected time (and w.h.p.) and $\mathcal{O}\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right)$ working space, having read-access to S itself.*

Note that, if we cannot compute z in low space, we can simply choose $s = 1$, so that we first build all the levels and then remove the useless top levels; we can also apply pruning at this point. In this case, however, the construction time becomes $\mathcal{O}(n \log_\tau z)$.

6.3 Construction in external memory

In large and very repetitive texts, where $z \ll n$, the bulk of the working space in the preceding construction will be the $n \log \sigma$ bits to maintain S, not the space proportional to the block tree size. We now show that, if S is maintained on disk, we can use $\mathcal{O}\left(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\right)$ main memory construction space, $\mathcal{O}(n(1+\log_\tau(z/s)))$ expected CPU time, and perform only $\mathcal{O}\left(\log_\tau \frac{n \log \sigma}{s \log n}\right)$ sequential passes over S on disk.

[2]That is, with probability $1 - \mathcal{O}(n^{-c})$ for any desired constant c.

For each level, we traverse S once to compute the Karp-Rabin signatures of the left-to-right blocks (or block pairs) in that level. A second traversal on S slides the window and finds leftmost occurrences of the blocks or block pairs. The only important difference is that this time we cannot afford comparing the strings with matching signatures, because the string contents are on disk.

Considering that the strings will be equal w.h.p., we can do as follows. We assume that matching signatures imply matching strings and go on with the construction. Note that the resulting block tree, if incorrect, must be smaller than the correct block tree, because it does not mark some blocks that should have been marked. Therefore, the main memory space is still in $\mathcal{O}\Big(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\Big)$. Once the block tree is built, we use Theorem 4 to retrieve its corresponding string $S'[1..n]$ by chunks of $\log_\sigma n$ symbols, in total time $\mathcal{O}(h \cdot n / \log_\sigma n) \subseteq \mathcal{O}(n \log_\tau \sigma) \subseteq \mathcal{O}(n \log_\tau z)$, because $z \geq \sigma$. As we extract the symbols of S', we scan S one final time, in order to verify that $S = S'$. If they are distinct, we discard everything and run the construction again. This Las Vegas procedure iterates a constant expected number of times (once, w.h.p.), which retains the time complexities.

Theorem 8. *The block tree of any string $S[1..n]$ over alphabet $[1..\sigma]$, with integer parameters τ and s, can be built in $\mathcal{O}(n(1 + \log_\tau z))$ expected time (and w.h.p.) and $\mathcal{O}\Big(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}\Big)$ working space, plus $\mathcal{O}\Big(\log_\tau \frac{n \log \sigma}{s \log n}\Big)$ sequential read-only passes on S (in particular, S may reside on disk).*

As before, we can simply use $s = 1$ and have a construction I/O complexity of $\mathcal{O}((n/B) \log_\tau n)$, where B is the disk block size. If we want to use $s = z$, an external memory algorithm computes z within sorting I/O complexity [KKP14]. We can prune the tree in main memory after building it.

7 Implementation

Our implementation of the block tree data structure is built on top of the `sdsl` library[3] and is publicly available at `https://github.com/elarielcl/BlockTrees`.

We implemented the construction described in Section 6.2, which is the most practical if the string fits in main memory. We include the pruning algorithm. We choose parameter $s = 1$, so we build all the block tree levels. Then, we remove the top levels that do not contain any unmarked block. We retain parameter τ and use a parameter b telling the length of the strings that are stored explicitly at the last-level leaves.

The blocks at each level form a conceptual sequence. A bitvector aligned to that sequence, in each level, indicates with a 1 which are marked. Since for each marked block there are exactly τ blocks in the next level, we easily traverse the tree downwards by computing rank_1 on these bitvectors. The leftward pointers from unmarked blocks are packed in an array, which is addressed using rank_0 on the same bitvector. All the arrays of integers are stored using the minimum number of bits required to store their maximum value (class `sdsl::int_vector`), and the bitvector is indexed to support constant-time rank queries (class `sdsl::bit_vector` with `sdsl::rank_support_v<1>`).

As in the definition, we extend the string up to the next power of τ with a special terminator symbol, but omit the blocks that are formed only by terminators, as these will never be accessed. Since we do not store explicit pointers to children, this imposes a space overhead of $\mathcal{O}(\log_\tau(n/b))$.

[3] `https://github.com/simongog/sdsl-lite`

Access. We store no extra information in order to support access to S. In particular, to save space, we do not store the $\log_\sigma n$ explicit leftmost and rightmost symbols described in Section 5.1, but rather use the simpler procedure of Theorem 3. For unmarked blocks pointing to B_i or to B_i and B_{i+1}, we store the integer offset of B_i, since B_{i+1} must be next to it in case it exists. We also store the offset where the block copy starts inside B_i. The last-level blocks are also stored in packed form, concatenating all the explicit strings.

Therefore, it takes $\mathcal{O}(m \log_\tau(n/b))$ time to extract m consecutive symbols in the worst case. In practice, if we keep track of the segment to extract as we descend, it is likely that we arrive at a leaf with a range longer than 1, and then can access all those symbols together in optimal time.

Rank and select. To support rank queries, we store additional fields. For the top-level blocks, we store the rank of every symbol a just before starting the current block. For the other blocks (which are children of marked blocks), in order to represent smaller numbers, we store the rank of every symbol a inside the blocks. Thus, when we go down, we must accumulate the ranks of the siblings preceding the desired child, but the effect of this overhead was negligible for the small τ values we used. For unmarked blocks B^u pointing to $B_i \cdot B_{i+1}$, we also store $B^u.\mathrm{rank}_a(d)$ for every symbol a; recall Figure 2. For binary sequences we only store the rank of one of the two symbols, since the other can be inferred.

To implement select we perform binary search on the rank values of the first level, and then sequential search on the children.

On the explicit strings at the leaves, we store no rank/select information, thus we have to scan the b symbols of one leaf to complete the operations, but these accesses are cache-friendly.

As a result, rank takes $\mathcal{O}(\tau \log_\tau(n/b) + b)$ and select takes $\mathcal{O}(\log n + \tau \log_\tau(n/b) + b)$ time.

8 Experiments

We report practical performance of our block trees compared to other state-of-the art sequence representations solving access, rank, and select queries, on different kinds of repetitive sequences.

8.1 Sequences

Table 1 lists our sequences with their basic statistics, including Lempel-Ziv compressibility using `p7zip`[4]. The base of our sources are repetitive text collections obtained from the Repetitive Corpus of the *Pizza&Chili* platform[5]. We consider the following real texts:

`influenza`**:** A collection composed of 78,041 DNA sequences of *Haemophilus Influenzae*, obtained from the NCBI[6].

`escherichia`**:** A collection of DNA sequences of different *Escherichia Coli* individuals, also obtained from the NCBI.

[4] `http://p7zip.sourceforge.net`
[5] `http://pizzachili.dcc.uchile.cl/repcorpus`
[6] `https://www.ncbi.nlm.nih.gov`

Collection	Size	σ	p7zip
influenza	148 MB	4	1.69%
escherichia	107 MB	4	7.76%
einstein	89 MB	117	0.11%
kernel	247 MB	160	2.56%
dna.0001	100 MB	4	0.52%
dna.001	100 MB	4	0.54%
dna.01	100 MB	4	0.76%
dna.1	100 MB	4	2.35%
dna0.001.par	399 MB	2	2.63%
dna0.01.par	399 MB	2	4.19%

Collection	Size	σ	p7zip
dna.0001.wt.bv	200 MB	2	0.26%
dna.001.wt.bv	200 MB	2	0.31%
dna.01.wt.bv	200 MB	2	0.73%
dna.1.wt.bv	200 MB	2	3.62%
escherichia.wt.bv	214 MB	2	5.34%
influenza.wt.bv	296 MB	2	2.25%
dna.0001.bwt.wt.bv	200 MB	2	0.65%
dna.001.bwt.wt.bv	200 MB	2	0.87%
dna.01.bwt.wt.bv	200 MB	2	1.53%
dna.1.bwt.wt.bv	200 MB	2	4.38%
escherichia.bwt.wt.bv	214 MB	2	7.23%
influenza.bwt.wt.bv	296 MB	2	1.34%

Table 1: The repetitive collections we use, with their size n (in MB, assuming one byte per symbol is used), their alphabet size σ, and the compression achieved with p7zip (a hint on $\frac{z \log n}{n \log \sigma}$, again assuming one byte per symbol is used).

einstein: A collection of all the versions (up to January 12, 2010) of the German Wikipedia Article of Albert Einstein.

kernel: A collection of 36 versions of the Linux Kernel.

We also consider so-called "pseudo-real" texts. These are obtained by creating, from a real nonrepetitive text, a collection with controlled repetitiveness. This is useful to study how the structures evolve as repetitiveness increases. We use the pseudo-real texts dna0.001, dna0.01, dna0.1, and dna1.0, where each dna$p$ is built by taking a 1MB prefix of the DNA sequence in the corpus, copying it 100 times, and mutating (substituting by another) each symbol with probability $p/100$. The 1MB prefix is obtained from the DNA sequence of the Gutenberg Project[7].

We are particularly interested in binary sequences, because block trees have a σ blowup factor in their space when supporting rank and select. We then generate various binary sequences from our base texts, where these operations are useful in applications:

***.par:** These are obtained from the respective text, by building its suffix tree [Wei73, CR02] and then representing it as a sequence of balanced parentheses [MR01, Nav16], which is used for example to represent suffix trees of repetitive collections [NO16, CN19].

***.wt.bv:** These are the concatenation of the bitvectors of the wavelet trees [GGV03, Nav16] of the sequences. Wavelet trees are sequence representations that support access, rank, and select. Since deeper levels of the wavelet tree lose the original repetitiveness, we use only the DNA sequences, whose wavelet trees have two levels only.

[7] http://www.gutenberg.org

`*.bwt.wt.bv`: The concatenations of the bitvectors of the wavelet trees of the Burrows-Wheeler Transform (BWT) [BW94, Nav16] of the sequences. The BWT transform is used to implement text indexes by using the operations access, rank, and select.

8.2 Data structures

We compare our block trees with structures specialized for each kind of sequence, binary and non-binary. The space for operations rank and select is included only in the corresponding plots.

8.2.1 Binary alphabet

BT Our block trees. We vary the arity $\tau \in \{2, 4, 8\}$ and the length of the blocks at the leaves, $b \in \{16, 32, 64, 128, 256, 512\}$.

GCCN,GCCC The most efficient implemented grammar-compressed sequence representation supporting the operations [ONB17]. We tested both regular sampling on text (GCCN), and sampling on the start rule (GCCC). We vary their parameters as the authors suggest in their original publication.

CM An uncompressed bitvector representation [Cla96, Nav16], using an implementation [GGMN05] with one level of counters over the plain bitvectors. We vary the sampling length for the counters, $s \in \{32, 64, 128\}$.

RRR A statistically compressed bitvector representation [RRR07, Nav16], using an efficient implementation [CN08] where we vary the sampling length for counters, $s \in \{32, 64, 128\}$.

8.2.2 Non-binary alphabet

BT Our block trees, varying $\tau \in \{2, 4, 8\}$ and $b \in \{2, 4, 8, 16, 32, 64\}$.

GCCN,GCCC: The grammar-compressed sequence representation [ONB17], with the same considerations as for the binary alphabet.

WT.{GCCN,GCCC}: The wavelet tree [GGV03, Nav16] using different representations for the bitvectors, with the same parameterizations as for binary alphabets.

HSWT.{GCCN,GCCC,CM,RRR} The Huffman-shaped wavelet tree [GGV03, Nav16] using different representations of their bitvectors, with the same parameterizations as for binary alphabets.

We omit some combinations that are never Pareto-optimal, such as WT.CM and WT.RRR, to avoid cluttering the plots. Except for BT, the code of the structures comes from the benchmark of the GCC article [ONB17].

8.3 Setup and Plots

Our experiments ran on an isolated Intel(R) Xeon(R) CPU E5620 @ 2.40GHz with 96GB of RAM and 10MB of L3 cache. The operating system is GNU/Linux, Debian 2, with kernel 4.9.0-6-amd64. The implementations use a single thread and all of them are coded in C++. The compiler is gcc version 6.3.0, with -O9 optimization flag set.

For every input, operation, and tested structure, we ran 10,000 random queries, and obtained the average time. The plots show only the Pareto-optimal points. In the case of access we chose 10,000 random positions of the input. For rank we also chose 10,000 random letters from the input alphabet. For select, we chose 10,000 random characters, and 10,000 random ranks between 1 and the number of times the character occurs.

Points are shown in 2D-graphs, where the x-coordinate represents the space in bps (bits per symbol) and the y-coordinate is the time per operation in microseconds and logarithmic scale; the input and operation analyzed are indicated in the title of every graph. The results are shown in Figures 4 to 12.

8.4 Results

Access. Figures 4 to 6 show the results on operation access. On general sequences (Figure 4) block trees (BT) excel. In all the datasets, they outperform GCC, the grammar-based alternative, by more than an order of magnitude in time, while using nearly the same space. On the other hand, BT is several times smaller and as fast as the structures that use statistical compression (HSWT.CM, HSWT.RRR, and WT.RRR) or no compression at all (WT.CM; some are omitted in the plots). The alternatives that combine GCC with wavelet trees ({WT,HSWT}.{GCCN,GCCC}) are dominated by GCC, and sharply dominated by BT. Note that access times in uncompressed and statistically compressed structures is $\mathcal{O}(\log \sigma)$ (0.1–1 μsec in practice), whereas grammars require $\mathcal{O}(\log n)$ time (around 10 μsec in practice) and block trees require time $\mathcal{O}(\log(n/s))$ (0.1–1 μsec in practice), where s is the number of blocks in the first represented level.

In the case of binary alphabets, Figure 5 shows the case of parenthesis sequences and of wavelet tree bitvectors, whereas Figure 6 shows the bitvectors of wavelet trees of BWTs. On parentheses, BT offers a relevant tradeoff: it is more than an order of magnitude faster than GCC, though 3–4 times larger. BT is as fast as statistically compressed bitvectors (RRR) and 3–5 times smaller. In fact, statistical compression is useless in these sequences: the plain representation (CM) uses the same space (i.e., 3–5 times larger than BT), though it is nearly an order of magnitude faster than BT (its operation times are $\mathcal{O}(1)$, nearly 0.01 μsec in practice, whereas BT and GCC require the same time as for general alphabets).

On wavelet-tree bitvectors, BT is always better than GCC, using less or similar space and more than an order of magnitude less time. Uncompressed bitvectors (CM) are an order of magnitude faster than BT, but they use many times more space (twice in the least repetitive sequences, and many more in the others). Statistically-compressed bitvectors (RRR) are again dominated by uncompressed representations.

The situation changes somehow on the BWTs. BT still outperforms GCC by an order of magnitude in time, but it loses in space on the less repetitive sequences. On the other hand, RRR becomes much more competitive because it reaches high-order statistical compression, and it even

dominates BT on the least repetitive sequences.

Rank and Select. Figures 7 to 9 show the results for rank queries. In the case of general sequences (Figure 7), the need to store σ counters per block makes BT significantly larger, especially on the less repetitive sequences or on large alphabets. Instead, GCC has sampling mechanisms that diminish this impact [ONB17], although it also grows. As a result, although the time comparisons are as for operation access, BT is competitive in space only on the most repetitive DNA sequences. Statistically compressed structures do not have the same problem, because their representation already supports rank and select with no extra space. As a consequence, BT is dominated by statistically compressed representations on `dna1.0` and `escherichia`.

On binary alphabets (Figures 8 and 9), BT and GCC need only to store one extra field, which diminishes the impact on their space. Still, the impact is much larger on BT than on GCC, and as a consequence BT uses much more space than GCC on various less repetitive sequences. The uncompressed representation (CM) is nearly an order of magnitude faster and still generally smaller, but by a smaller factor. On the BWT-transformed sequences, BT is still about an order of magnitude faster than GCC, but always larger by a significant factor. The statistically compressed representation (RRR) is several times faster than BT, and smaller on the less repetitive sequences.

As a rough figure, rank on general sequences takes around 10 μsec on GCC ($\mathcal{O}(\log n)$ complexity) and 1 μsec on BT ($\mathcal{O}(\log(n/s))$ complexity). The statistically compressed and uncompressed representations take around 1 μsec on general alphabets ($\mathcal{O}(\log \sigma)$ complexity) and 0.1 μsec on binary alphabets ($\mathcal{O}(1)$ complexity).

Figures 10 to 12 show the results for select, which are similar to those of rank, except that the practical implementation of this operation on uncompressed and statistically compressed bitvectors takes time $\mathcal{O}(\log n)$, and therefore they do not outperform BT and GCC as for rank. While the times of GCC and BT are nearly unaffected, the wavelet-tree based representations now take 1–10 μsec, being noticeably worse on the larger alphabets. Similarly, on bitvectors, CM and RRR are at most twice as fast as BT, lying in the range of 0.5–1 μsec.

9 Conclusions

We have introduced the block tree, a data structure that represents sequences in compressed form while supporting logarithmic-time access and rank/select queries on them. In particular, if a string $S[1..n]$ is parsed into z phrases (and thus compressed to $\mathcal{O}(z)$ space) by Lempel-Ziv [LZ76], then the block tree can represent S in $\mathcal{O}(z \log(n/z))$ space and support the queries in $\mathcal{O}(\log(n/z))$ time. Our experiments demonstrate that block trees are very competitive compressed representations of highly repetitive sequences: they use about the same space of grammar-compressed representations [ONB17] but they can be orders of magnitude faster, and they are not much slower than the statistically-compressed representations [Cla96, RRR07, GGV03], which are several times larger.

9.1 Epilogue

Since our conference publication [BGG+15], block trees have had a significant impact on compressed data structures. In particular, its basic idea of marked and unmarked blocks has been applied to

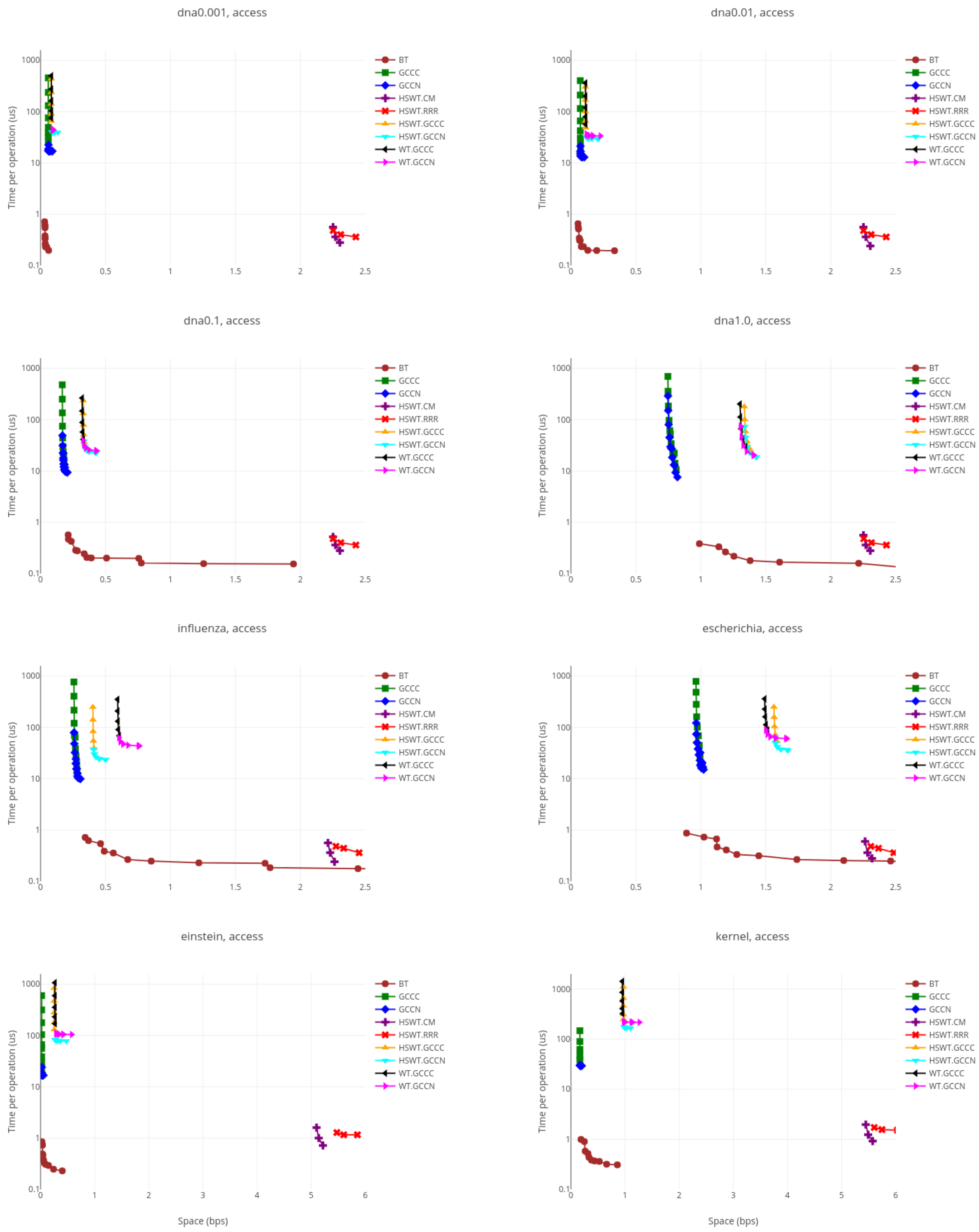

Figure 4: Access on non-binary alphabets.

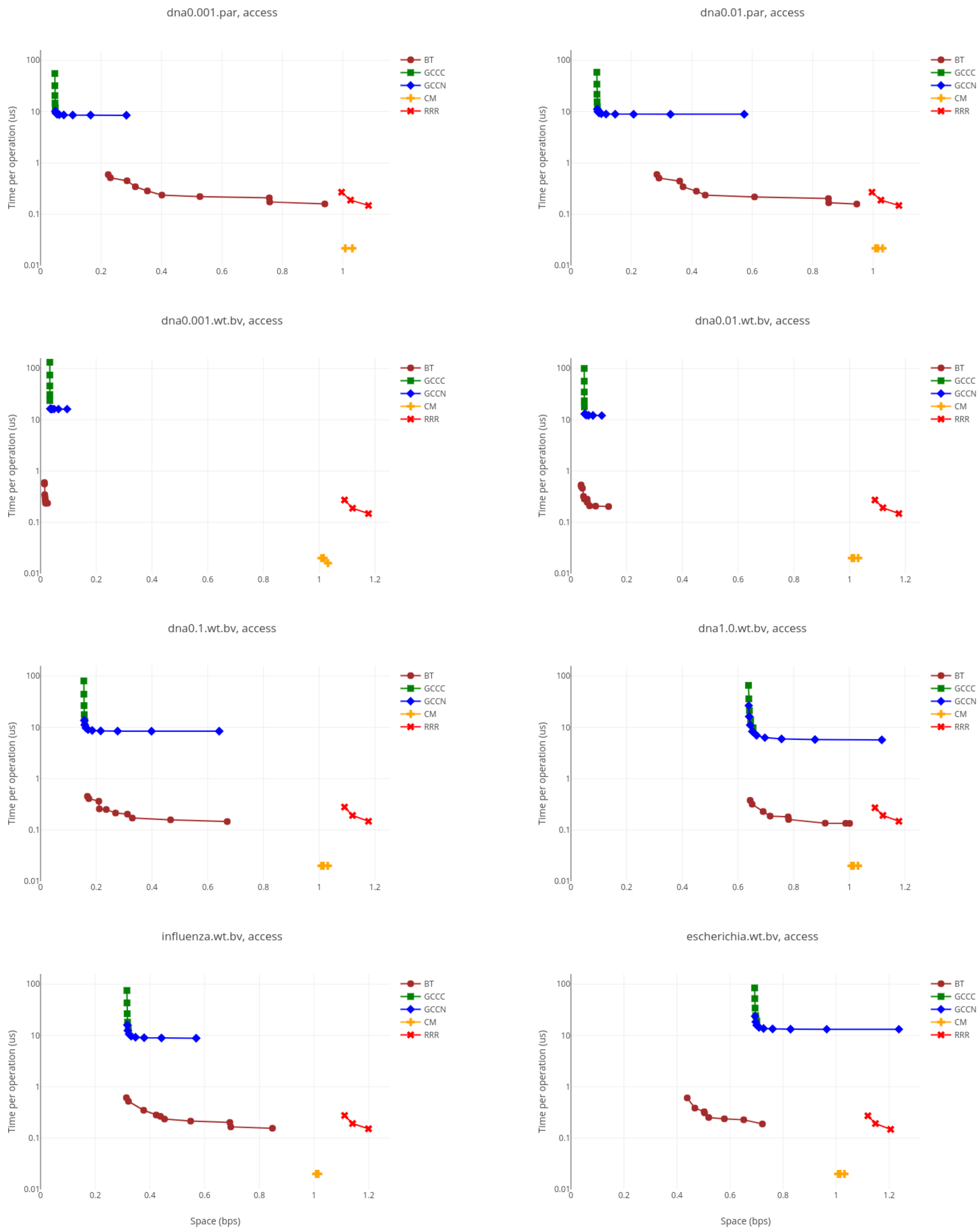

Figure 5: Access on binary alphabets: suffix tree topologies and wavelet tree bitvectors.

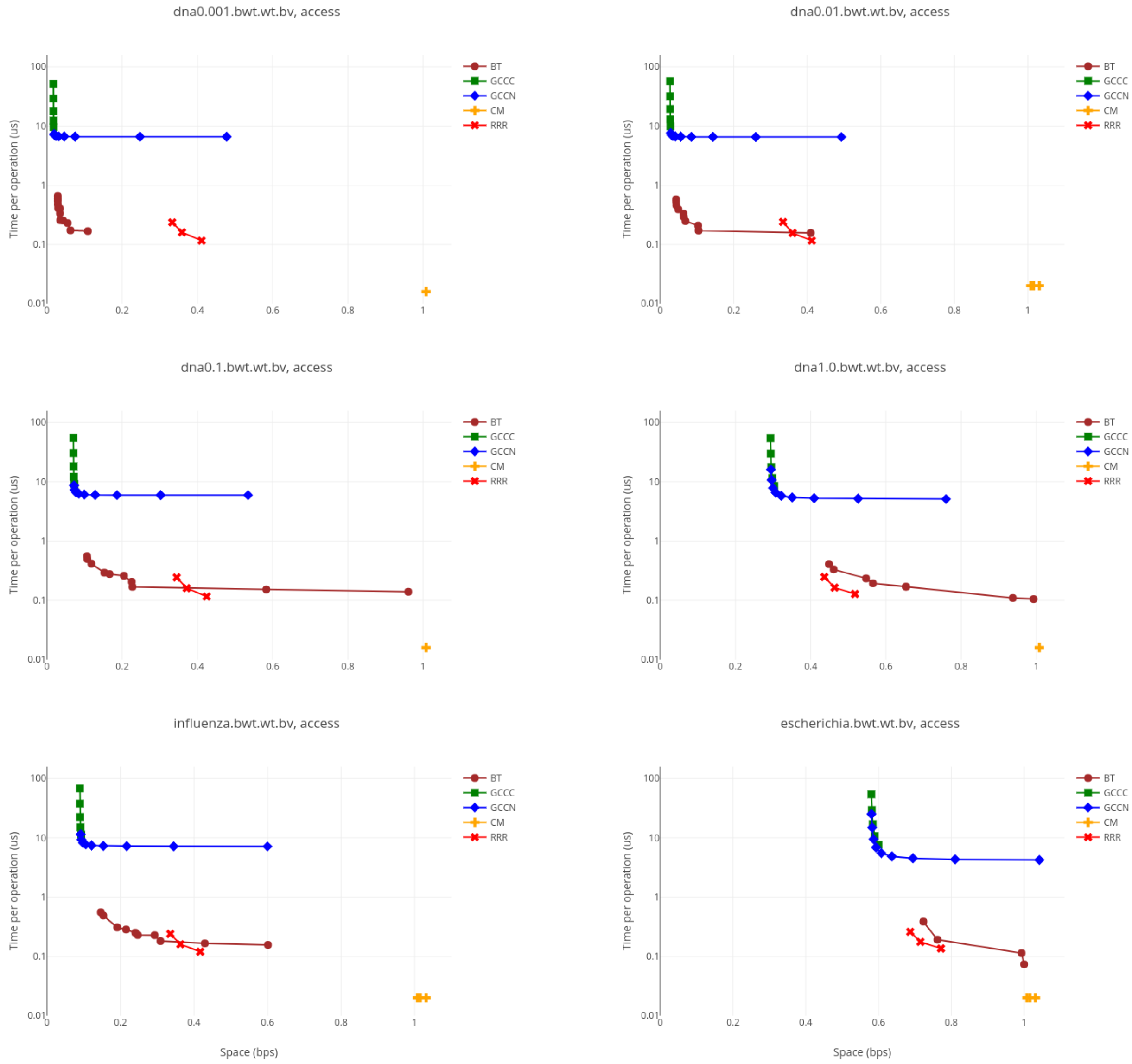

Figure 6: Access on binary alphabets: wavelet trees of BWTs.

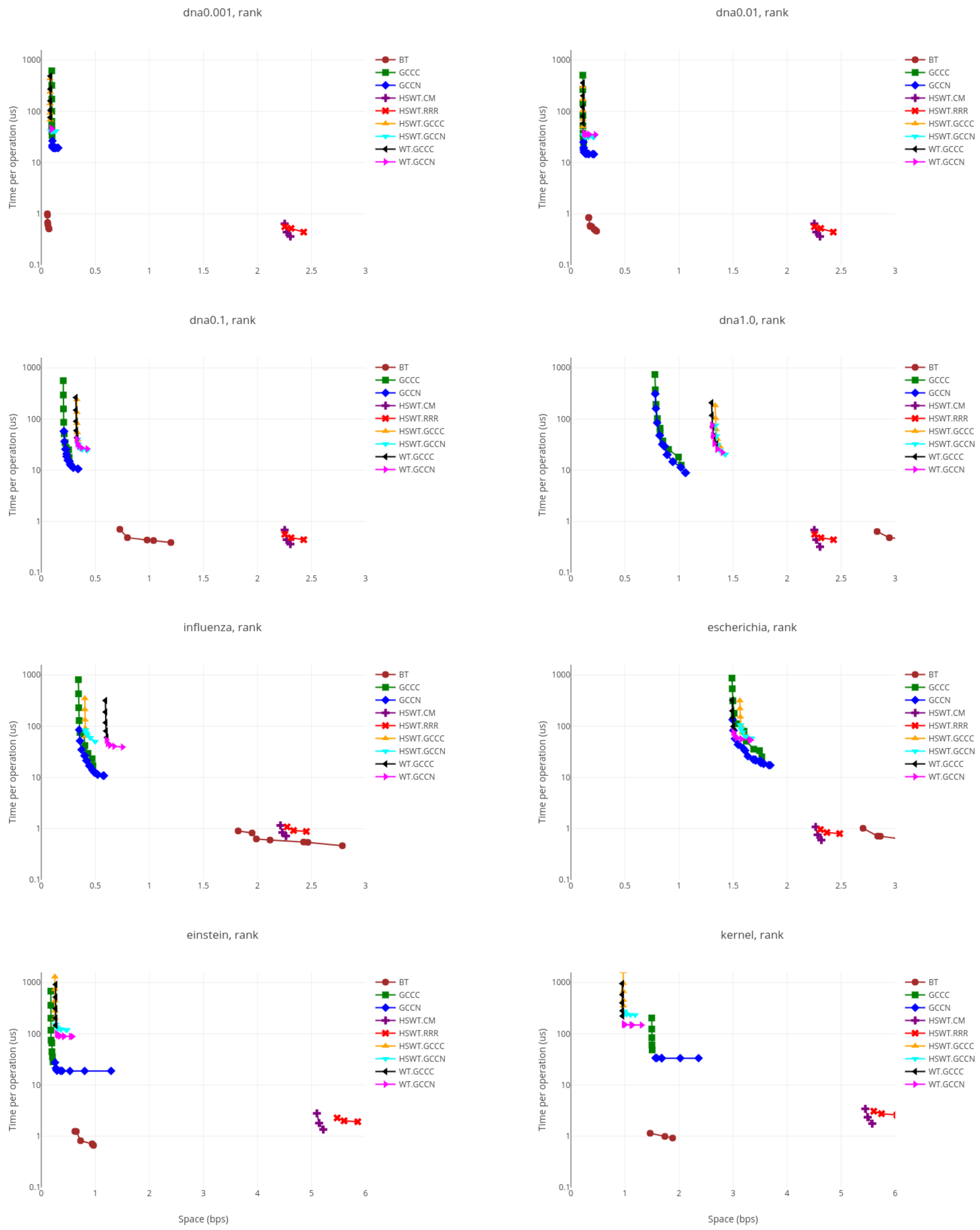

Figure 7: Rank on general sequences.

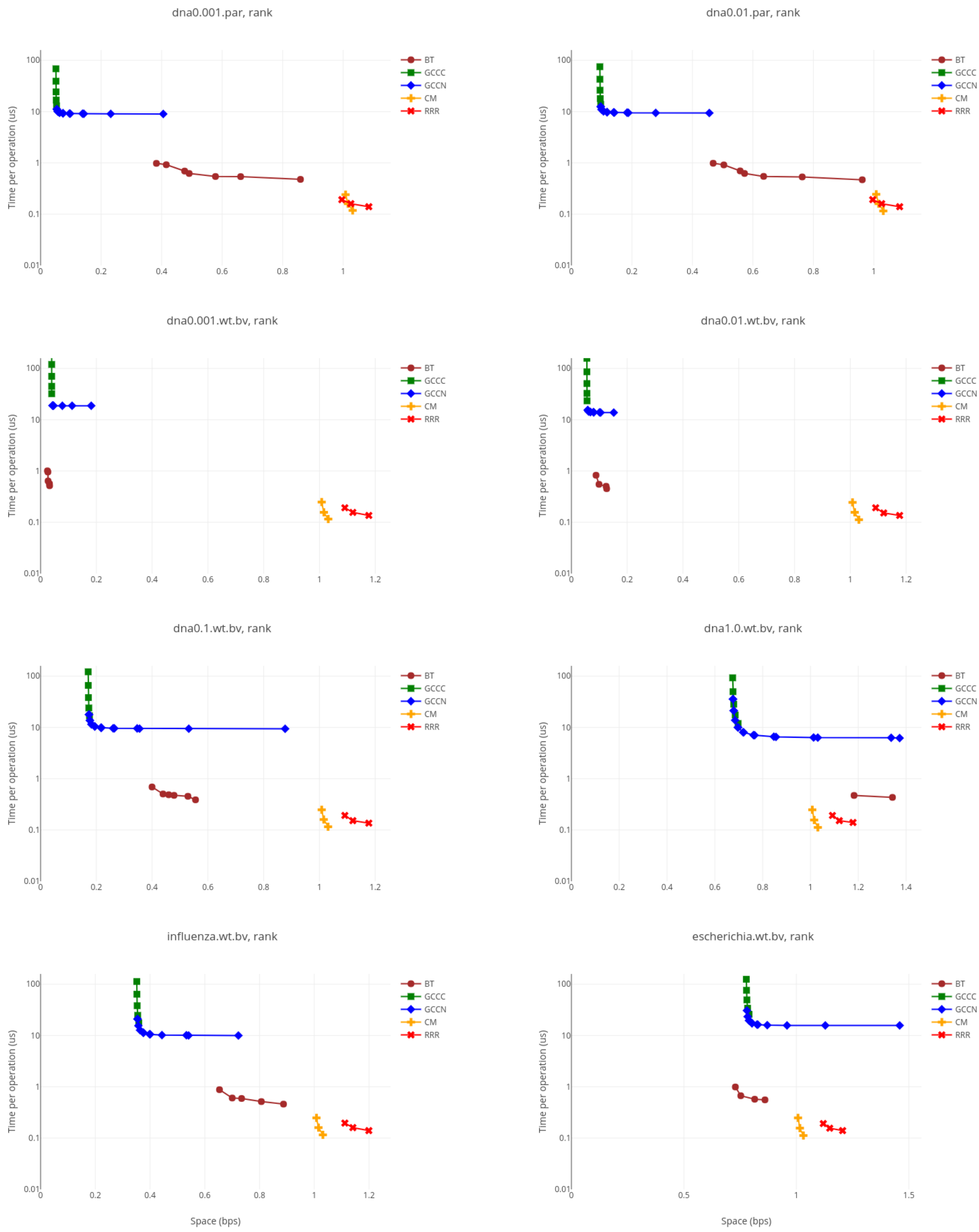

Figure 8: Rank on binary alphabets: suffix tree topologies and wavelet tree bitvectors.

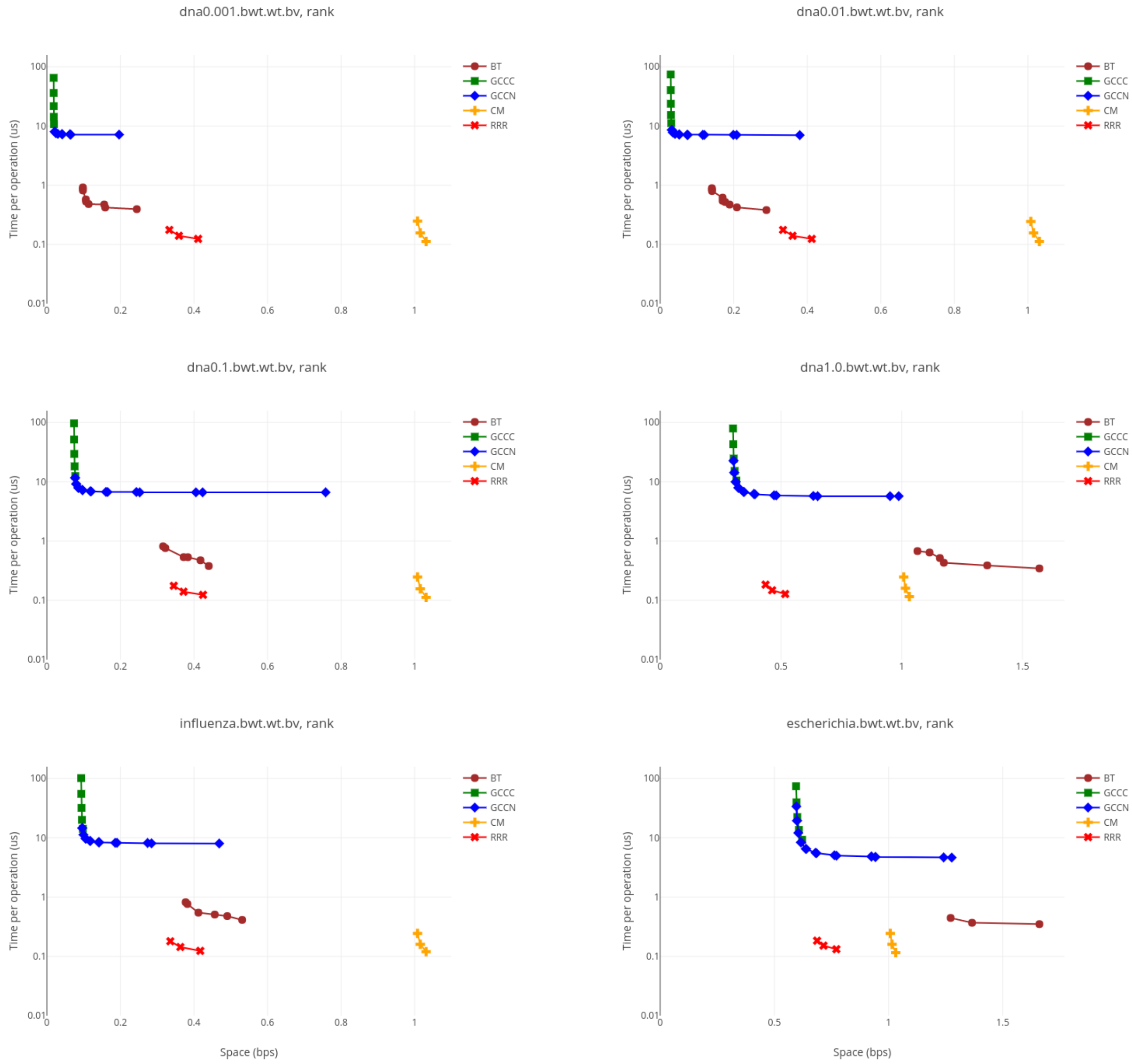

Figure 9: Rank on binary alphabets: wavelet trees of BWTs.

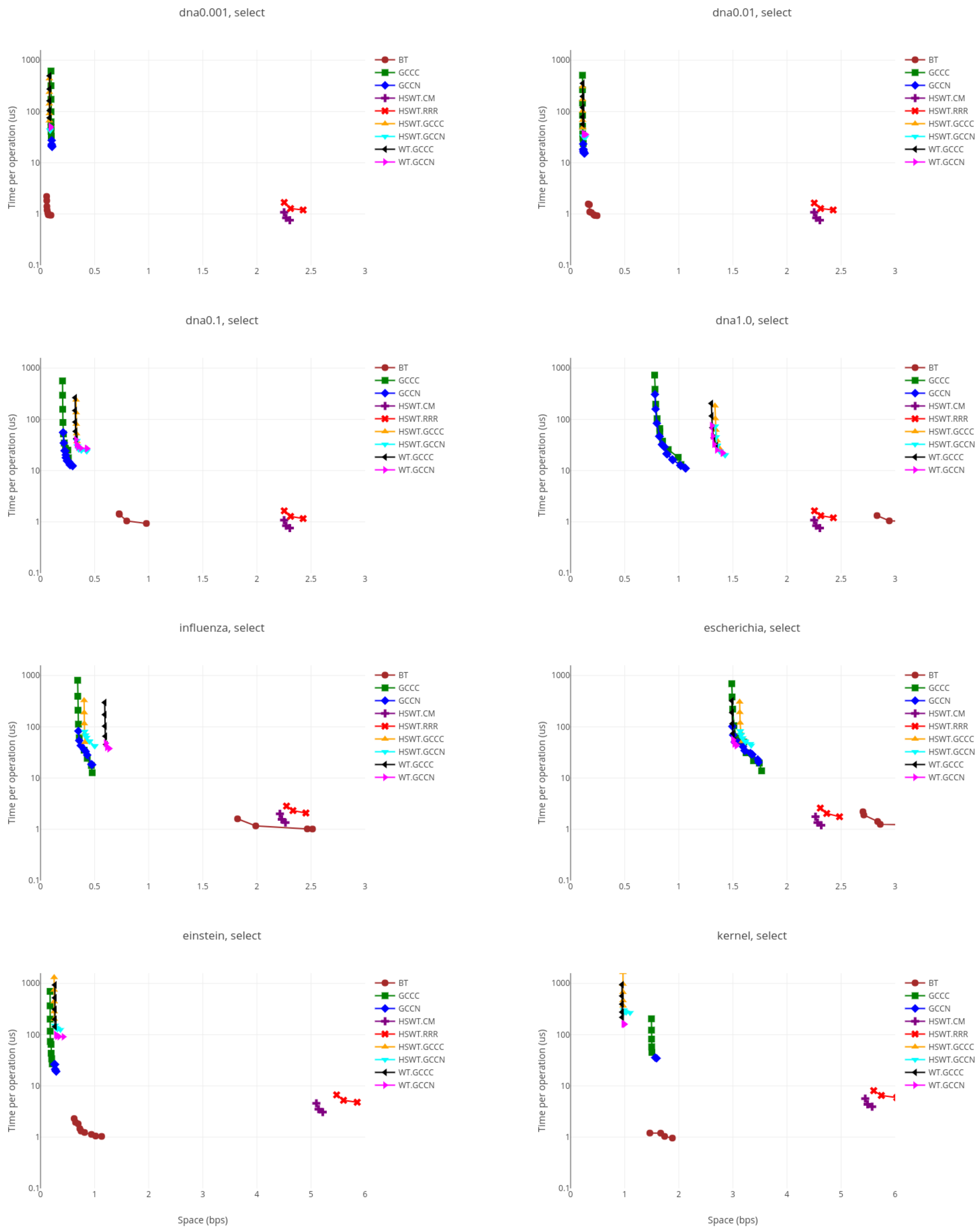

Figure 10: Select on general sequences.

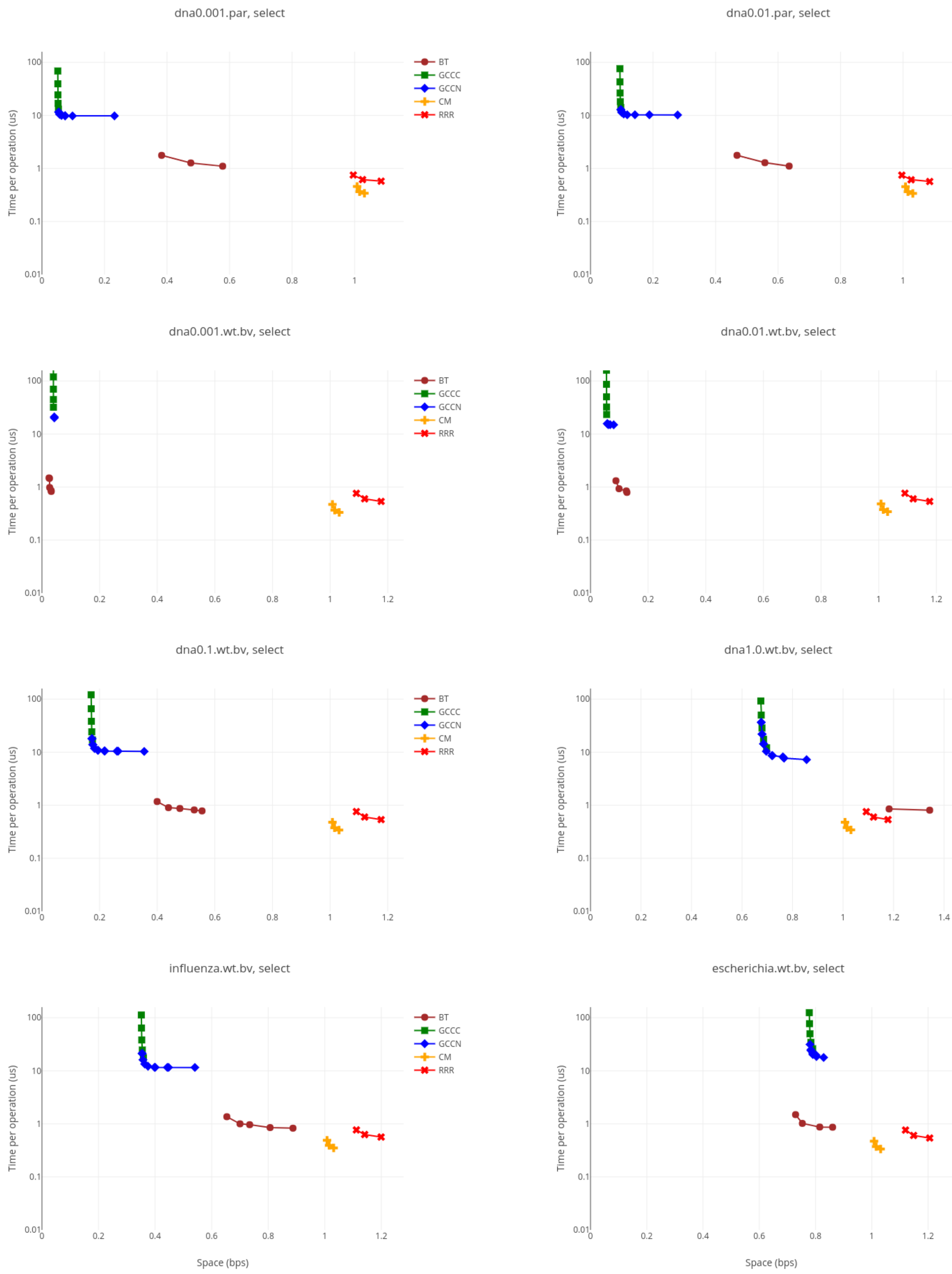

Figure 11: Select on binary alphabets: suffix tree topologies and wavelet tree bitvectors.

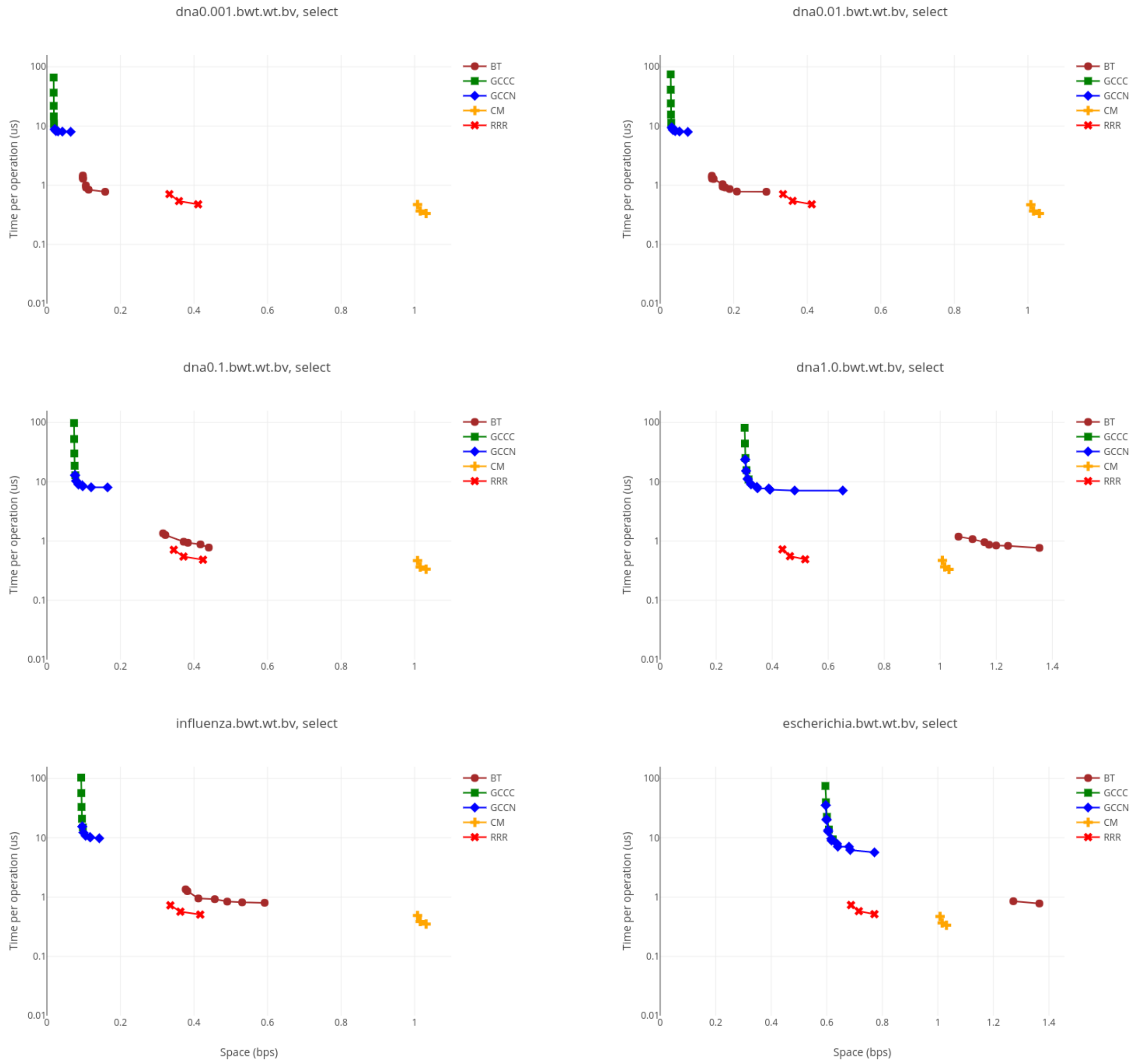

Figure 12: Select on binary alphabets: wavelet trees of BWTs.

analogous situations. Kempa and Prezza [KP18] defined *attractors*, which are combinatorial objects that describe the repetitiveness of a string $S[1..n]$. An attractor Γ is a set of positions so that any substring of S has a copy including a position in Γ. The size γ of the smallest attractor lower-bounds z, $\gamma \leq z$, and a block-tree-like data structure is shown to provide access on S in time $\mathcal{O}(\log(n/\gamma))$ and space $\mathcal{O}(\gamma \log(n/\gamma))$ [KP18]. They also extract a substring of length m in time $\mathcal{O}(\log(n/\gamma) + m/\log_\sigma n)$. Such block-tree-like structure can also be enhanced to support rank and select operations in time $\mathcal{O}(\log_\tau(n/\gamma))$ by using space $\mathcal{O}(\gamma\sigma\tau \log_\tau(n/\gamma))$ [Pre19].

Navarro and Prezza [NP19] used a variant, called Γ-tree, that is closer to an actual block tree with parameters $s = \gamma$ and $r = 2$. The main difference of both variants [KP18, NP19] is that the marked blocks in each level are those close to an attractor position. It can then be shown, much like our Lemma 1, that since any unmarked block has a copy including an attractor position, such copy lies within the concatenation of two marked blocks. Navarro and Prezza [NP19] add fields in the Γ-tree blocks to support another operation: given a range (i, j), compute the Karp-Rabin fingerprint of $S[i..j]$ in time $\mathcal{O}(\log(n/\gamma))$. This structure is then used as the basis of a compressed text index, which uses $\mathcal{O}(\gamma \log(n/\gamma))$ space and finds the *occ* occurrences of a pattern of length m in S in $\mathcal{O}(m \log n + occ \log^\epsilon n)$ time for any constant $\epsilon > 0$. Their fingerprint calculation algorithm works almost verbatim on the original block trees.

Such block tree variants based on attractors have an important conceptual interest because many other dictionary compressors can be expressed in terms of attractors, for example collage systems [KMS+03] and bidirectional macro schemes [SS82], in addition to Lempel-Ziv parsing: if any such method compresses $S[1..n]$ into space p, then one can build an attractor of size $\gamma = \mathcal{O}(p)$ and thus a block-tree-like structure built on S uses $\mathcal{O}(p \log(n/p))$ space and supports direct access (and other functions) in $\mathcal{O}(\log(n/p))$ time. Finding the smallest attractor size γ is NP-complete [KP18], however, and among all those compression methods, Lempel-Ziv parsing provides the best approximation to the smallest γ that can be computed in polynomial time (actually, in linear time). Therefore, this generalization boils down, in practice, to our original Lempel-Ziv-bounded block trees (i.e., marked blocks are those around phrase boundaries).

Even more remarkably, Kociumaka et al. [KNP20] worked on an even stricter measure of repetitiveness, $\delta \leq \gamma$, which like z can be computed in linear time. They show that it is sufficient to choose $s = \delta$ in our block trees to obtain space $\mathcal{O}(\delta \log(n/\delta))$ while retaining the time complexities in terms of $\log(n/z)$. This then yields a practical construction of our block trees that is even smaller than those based on attractors, of size $\mathcal{O}(\gamma \log(n/\gamma))$.

Other generalizations of the block tree in the literature include two-dimensional block trees to compress adjacency matrices of Web graphs (and possibly repetitive images) [BGGBN18], self-indexes on the original block trees [Nav17], extensions to compute suffix array cells [Kem19], and extensions to support sequences of parentheses with applications to suffix trees of repetitive text collections [CN19]. The latter application is interesting because it exposes some possible limitations of the block-tree functionality: a sequence of parentheses that represents a tree topology needs to support further operations, such as finding a given "excess" forwards or backwards from a given position and giving the position with minimum "excess" in a range [NS14]. The excess of a position is the number of opening minus closing parentheses up to it. Despite what is incorrectly claimed in the conference version of this article [BGG+15], we know no way to compute these operations in polylogarithmic time using block trees (some heuristics perform well in practice

[CN19]). This is instead possible using context-free grammars [BLR+15, NO16], but block trees permit a block to be an arbitrary substring of two consecutive blocks, which context-free grammars do not support. Block trees are, instead, a particular case of collage systems [KMS+03], for which no polylogarithmic-time access algorithms are known. Delimiting which operations can be efficiently computed on block trees is an interesting open problem.

Acknowledgements

We thank Meg Gagie for correcting our grammar, and the reviewers for their thorough work.

References

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[BBH+87] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[BC17] D. Belazzougui and F. Cunial. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:13, 2017.

[BEGV18] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theoretical Computer Science*, 713:66–77, 2018.

[BGG+15] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S. J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. 25th Data Compression Conference (DCC)*, pages 83–92, 2015.

[BGGBN18] N. Brisaboa, T. Gagie, A. Gómez-Brandón, and G. Navarro. Two-dimensional block trees. In *Proc. 28th Data Compression Conference (DCC)*, pages 229–238, 2018.

[BHMR11] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):article 52, 2011.

[BLR+15] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[BN15] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

[BPT15] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, pages 142–154, 2015.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[Cla96] D. R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[CLL+05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 176–187, 2008.

[CN19] M. Cáceres and G. Navarro. Faster repetition-aware compressed suffix trees based on block trees. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2019. To appear.

[CR02] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[FV07] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 371(1):115–121, 2007.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. Posters of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[GGP11] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proc. 22nd Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 653–662, 2011.

[GGP15] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Approximate pattern matching in LZ77-compressed texts. *Journal of Discrete Algorithms*, 32:64–68, 2015.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GJL19] M. Ganardi, A. Jeż, and M. Lohrey. Balancing straight-line programs. In *Proc. 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1169–1183, 2019.

[GN06] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 295–306, 2006.

[GNP20] T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

[Jeż16] A. Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[Kem19] D. Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1344–1357, 2019.

[KKP14] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th Data Compression Conference (DCC)*, pages 153–162, 2014.

[KMS+03] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[KN13] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[KNP20] T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. In *Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN)*, 2020. To appear.

[KP18] D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 827–840, 2018.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 2:249–260, 1987.

[KY00] J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[Nav16] G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

[Nav17] G. Navarro. A self-index on block trees. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 278–289, 2017.

[NO16] G. Navarro and A. Ordóñez. Faster compressed suffix trees for repetitive text collections. *Journal of Experimental Algorithmics*, 21(1):article 1.8, 2016.

[NP19] G. Navarro and N. Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

[NS14] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[ONB17] A. Ordóñez, G. Navarro, and N. Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017.

[Pre19] N. Prezza. Optimal rank and select queries on dictionary-compressed text. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 4:1–4:12, 2019.

[PT06] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[RPE81] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

[RRR07] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

[Ryt03] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[SG06] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[SS82] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[VY13] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.

[Wil00] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.