A Fast and Small Subsampled R-Index

Dustin Cobas
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dept. of Computer Science, University of Chile, Santiago, Chile

Travis Gagie
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dalhousie University, Halifax, Canada

Gonzalo Navarro
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dept. of Computer Science, University of Chile, Santiago, Chile

Abstract

The *r*-index (Gagie et al., JACM 2020) represented a breakthrough in compressed indexing of repetitive text collections, outperforming its alternatives by orders of magnitude. Its space usage, $\mathcal{O}(r)$ where r is the number of runs in the Burrows–Wheeler Transform of the text, is however larger than Lempel–Ziv and grammar-based indexes, and makes it uninteresting in various real-life scenarios of milder repetitiveness. In this paper we introduce the *sr*-index, a variant that limits a large fraction of the space to $\mathcal{O}(\min(r, n/s))$ for a text of length n and a given parameter s, at the expense of multiplying by s the time per occurrence reported. The *sr*-index is obtained by carefully subsampling the text positions indexed by the *r*-index, in a way that we prove is still able to support pattern matching with guaranteed performance. Our experiments demonstrate that the *sr*-index sharply outperforms virtually every other compressed index on repetitive texts, both in time and space, even matching the performance of the *r*-index while using 1.5–3.0 times less space. Only some Lempel–Ziv-based indexes achieve better compression than the *sr*-index, using about half the space, but they are an order of magnitude slower.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern matching, r-index, compressed text indexing, repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.13

Supplementary Material *Software (Source Code)*: `https://github.com/duscob/sr-index` archived at `swh:1:dir:edbc683faf60b1c3604e211cb464d6183822bb0a`

Funding *Dustin Cobas*: ANID/Scholarship Program/DOCTORADO BECAS CHILE/2020-21200906, Chile. Basal Funds FB0001, ANID, Chile.
Travis Gagie: NSERC Discovery Grant RGPIN-07185-2020. Basal Funds FB0001, ANID, Chile.
Gonzalo Navarro: Fondecyt grant 1-200038, ANID, Chile. Basal Funds FB0001, ANID, Chile.

1 Introduction

The rapid surge of massive repetitive text collections, like genome and sequence read sets and versioned document and software repositories, has raised the interest in text indexing techniques that exploit repetitiveness to obtain orders-of-magnitude space reductions, while supporting pattern matching directly on the compressed text representations [10, 21].

Traditional compressed indexes rely on statistical compression [22], but this is ineffective to capture repetitiveness [15]. A new wave of repetitiveness-aware indexes [21] build on other compression mechanisms like Lempel–Ziv [16] or grammar compression [14]. A particularly useful index of this kind is the rlfm-index [18, 19], because it emulates the classical suffix array [20] and this simplifies translating suffix-array based algorithms to run on it [17].

The rlfm-index represents the Burrows–Wheeler Transform (BWT) [3] of the text in run-length compressed form, because the number r of maximal equal-letter runs in the BWT is known to be small on repetitive texts. A problem with the rlfm-index is that, although it can count the number of occurrences of a pattern using $\mathcal{O}(r)$ space, it needs to sample the text at every sth position, for a parameter s, in order to locate each of those occurrences in time proportional to s. The $\mathcal{O}(n/s)$ additional space incurred on a text of length n ruins the compression on very repetitive collections, where $r \ll n$. The recent *r*-index [11] closed the long-standing problem of efficiently locating the occurrences within $\mathcal{O}(r)$ space, offering pattern matching time orders of magnitude faster than previous repetitiveness-aware indexes.

In terms of space, however, the *r*-index is considerably larger than Lempel–Ziv based indexes of size $\mathcal{O}(z)$, where z is the number of phrases in the Lempel–Ziv parse. Gagie et al. [11] show that, on extremely repetitive text collections where $n/r = 500$–$10{,}000$, r is around $3z$ and the *r*-index size is 0.06–0.2 bits per symbol (bps), about twice that of the lz-index [15], a baseline Lempel–Ziv based index. However, r degrades faster than z as repetitiveness drops: in an experiment on bacterial genomes in the same article, where $n/r \approx 100$, the *r*-index space approaches 0.9 bps, 4 times that of the lz-index; r also approaches $4z$. Experiments on other datasets show that the *r*-index tends to be considerably larger [23, 5, 6, 1].Indeed, in some realistic cases n/r can be over 1,500, but in most cases it is well below: 40–160 on versioned software and document collections and fully assembled human chromosomes, 7.5–50 on virus and bacterial genomes (with r in the range $4z$–$7z$), and 4–9 on sequencing reads; see Section 5. An *r*-index on such a small n/r ratio easily becomes larger than the plain sequence data.

In this paper we tackle the problem of the (relatively) large space usage of the *r*-index. This index manages to locate the pattern occurrences by sampling r text positions (corresponding to the ends of BWT runs). We show that one can remove some carefully chosen samples so that, given a parameter s, the index stores only $\mathcal{O}(\min(r, n/s))$ samples while its locating machinery can still be used to guarantee that every pattern occurrence is located within $\mathcal{O}(s)$ steps. We call the resulting index the *subsampled* r-*index*, or *sr*-index. The worst-case time to locate the occ occurrences of a pattern of length m on an alphabet of size σ then rises from $\mathcal{O}((m + occ)\log(\sigma + n/r))$ in the implemented *r*-index to $\mathcal{O}((m + s \cdot occ)\log(\sigma + n/r))$ in the *sr*-index, which matches the search cost of the rlfm-index.

The *sr*-index can then be seen as a hybrid between the *r*-index (matching it when $s = 1$) and the rlfm-index (obtaining its time with less space; the spaces become similar when repetitiveness drops). In practice, however, the *sr*-index performs much better than both on repetitive texts, sharply dominating the rlfm-index, the best grammar-based index [5], and in most cases the lz-index, both in space and time. The *sr*-index can also get as fast as the *r*-index while using 1.5–4.0 times less space. Its only remaining competitor is a hybrid between a Lempel–Ziv based and a statistical index [7]. This index can use up to half the space of the *sr*-index, but it is an order of magnitude slower. Overall, the *sr*-index stays orders of magnitude faster than all the alternatives while using practical amounts of space in a wide range of repetitiveness scenarios.

2 Background

The *suffix array* [20] $\mathsf{SA}[1..n]$ of a string $\mathcal{T}[1..n]$ over alphabet $[1..\sigma]$ is a permutation of the starting positions of all the suffixes of $\mathcal{T}$ in lexicographic order, $\mathcal{T}[\mathsf{SA}[i]..n] < \mathcal{T}[\mathsf{SA}[i+1]..n]$ for all $1 \le i < n$. The suffix array can be binary searched in time $\mathcal{O}(m \log n)$ to obtain the range $\mathsf{SA}[sp..ep]$ of all the suffixes prefixed by a search pattern $P[1..m]$ (which then occurs $occ = ep - sp + 1$ times in $\mathcal{T}$). Once they are *counted* (i.e., their suffix array range is

determined), those occurrences are *located* in time $\mathcal{O}(occ)$ by simply listing their starting positions, $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$. The suffix array can then be stored in $n\lceil \lg n \rceil$ bits (plus the $n\lceil \lg \sigma \rceil$ bits to store $\mathcal{T}$) and searches for P in $\mathcal{T}$ in total time $\mathcal{O}(m \log n + occ)$.

Compressed suffix arrays (CSAs) [22] are space-efficient representations of both the suffix array (SA) and the text ($\mathcal{T}$). They can find the interval $\mathsf{SA}[sp..ep]$ corresponding to $P[1..m]$ in time $\mathrm{t}_{\mathrm{search}}(m)$ and access any cell $\mathsf{SA}[j]$ in time $\mathrm{t}_{\mathrm{lookup}}(n)$, so they can be used to search for P in time $\mathcal{O}(\mathrm{t}_{\mathrm{search}}(m) + occ\, \mathrm{t}_{\mathrm{lookup}}(n))$. Most CSAs need to store sampled SA values to compute any $\mathsf{SA}[j]$ in order to support the locate operation, inducing the tradeoff of using $\mathcal{O}((n/s)\log n)$ extra bits to obtain time $\mathrm{t}_{\mathrm{lookup}}(n)$ proportional to a parameter s.

The *Burrows–Wheeler Transform* [3] of $\mathcal{T}$ is a permutation $\mathsf{BWT}[1..n]$ of $\mathcal{T}[1..n]$ defined as $\mathsf{BWT}[i] = \mathcal{T}[\mathsf{SA}[i] - 1]$ (and $\mathcal{T}[n]$ if $\mathsf{SA}[i] = 1$), which boosts the compressibility of $\mathcal{T}$. The fm-index [8, 9] is a CSA that represents SA and $\mathcal{T}$ within the *statistical entropy* of $\mathcal{T}$, by exploiting the connection between the BWT and SA. For counting, the fm-index resorts to *backward search*, which successively finds the suffix array ranges $\mathsf{SA}[sp_i..ep_i]$ of $P[i..m]$, for $i = m$ to 1, starting from $\mathsf{SA}[sp_{m+1}..ep_{m+1}] = [1..n]$ and then

$$\begin{aligned} sp_i &= C[c] + \mathtt{rank}_c(\mathsf{BWT}, sp_{i+1} - 1) + 1, \\ ep_i &= C[c] + \mathtt{rank}_c(\mathsf{BWT}, ep_{i+1}), \end{aligned}$$

where $c = P[i]$, $C[c]$ is the number of occurrences of symbols smaller than c in $\mathcal{T}$, and $\mathtt{rank}_c(\mathsf{BWT}, j)$ is the number of times c occurs in $\mathsf{BWT}[1..j]$. Thus, $[sp, ep] = [sp_1, ep_1]$ if $sp_i \le ep_i$ holds for all $1 \le i \le m$.

For locating the occurrences $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$, the fm-index uses SA sampling as described: it stores sampled values of SA at regularly spaced text positions, say multiples of s. This is done via the so-called *LF-steps*: The BWT allows one to efficiently compute, given j such that $\mathsf{SA}[j] = i$, the value j' such that $\mathsf{SA}[j'] = i - 1$, called $j' = \mathsf{LF}(j)$. The formula is

$$\mathsf{LF}(i) = C[c] + \mathtt{rank}_c(\mathsf{BWT}, i),$$

where $c = \mathsf{BWT}[i]$. Note that the LF-steps virtually traverse the text backwards. By marking with 1s in a bitvector $B[1..n]$ the positions j^* such that $\mathsf{SA}[j^*]$ is a multiple of s, we can start from any j and, in $k < s$ LF-steps, find some sampled position $j^* = \mathsf{LF}^k(j)$ where $B[j^*] = 1$. By storing those values $\mathsf{SA}[j^*]$ explicitly, we have $\mathsf{SA}[j] = \mathsf{SA}[j^*] + k$.

By implementing BWT with a wavelet tree, for example, access and $\mathtt{rank}_c$ on BWT can be supported in time $\mathcal{O}(\log \sigma)$, and the fm-index searches in time $\mathcal{O}((m + s \cdot occ)\log \sigma)$ [9].

Since the statistical entropy is insensitive to repetitiveness [15], however, the fm-index is not adequate for repetitive datasets. The *Run-Length FM-index*, rlfm-index (and its variant rlcsa) [18, 19], is a modification of the fm-index aimed at repetitive texts. Say that the $\mathsf{BWT}[1..n]$ is formed by r maximal runs of equal symbols, then r is relatively small in repetitive collections (in particular, $r = \mathcal{O}(z \log^2 n)$, where z is the number of phrases of the Lempel–Ziv parse of $\mathcal{T}$ [13]). The rlfm-index supports counting within $\mathcal{O}(r \log n)$ bits, by implementing the backward search over alternative data structures. In particular, it marks in a bitvector $\mathtt{Start}[1..n]$ with 1s the positions j starting BWT runs, that is, where $j = 1$ or $\mathsf{BWT}[j] \neq \mathsf{BWT}[j-1]$. The first letter of each run is collected in an array $\mathtt{Letter}[1..r]$. Since $\mathtt{Start}$ has only r 1s, it can be represented within $r \lg(n/r) + \mathcal{O}(r)$ bits. Within this space, one can access any bit $\mathtt{Start}[j]$ and support operation $\mathtt{rank}_1(\mathtt{Start}, j)$, which counts the number of 1s in $\mathtt{Start}[1..j]$, in time $\mathcal{O}(\log(n/r))$ [25]. Therefore, we simulate $\mathsf{BWT}[j] = \mathtt{Letter}[\mathtt{rank}_1(\mathtt{Start}, j)]$ in $\mathcal{O}(r \log n)$ bits. The backward search formula can be efficiently simulated as well, leading to $\mathcal{O}((m + s \cdot occ)\log(\sigma + n/r))$ search time. However, the rlfm-index still uses SA samples to locate, and when $r \ll n$ (i.e., on repetitive texts), the $\mathcal{O}((n/s)\log n)$ added bits ruin the $\mathcal{O}(r \log n)$-bit space (s is typically $\mathcal{O}(\log n)$ or close).

The *r*-index [11] closed the long-standing problem of efficiently locating the occurrences of a pattern in a text using $\mathcal{O}(r \log n)$-bit space. The experiments showed that the *r*-index outperforms all the other implemented indexes by orders of magnitude in space or in time to locate pattern occurrences on highly repetitive datasets. However, other experiments on more typical repetitiveness scenarios [23, 5, 6, 1] showed that the space of the *r*-index degrades very quickly as repetitiveness decreases. For example, a grammar-based index (which can be of size $g = \mathcal{O}(z \log(n/z))$) is usually slower but significantly smaller [5], and an even slower Lempel–Ziv based index of size $O(z)$ [15] is even smaller. Some later proposals [24] further speed up the *r*-index by increasing the constant accompanying the $\mathcal{O}(r \log n)$-bit space. The unmatched time performance of the *r*-index comes then with a very high price in space on all but the most highly repetitive text collections, which makes it of little use in many relevant application scenarios. This is the problem we address in this paper.

3 The *r*-index Sampling Mechanism

Gagie et al. [11] provide an $\mathcal{O}(r \log n)$-bits data structure that not only finds the range $\mathsf{SA}[sp..ep]$ of the occurrences of P in $\mathcal{T}$, but also gives the value $\mathsf{SA}[ep]$, that is, the text position of the last occurrence in the range. They then provide a second $\mathcal{O}(r \log n)$-bits data structure that, given $\mathsf{SA}[j]$, efficiently finds $\mathsf{SA}[j-1]$. This suffices to efficiently find all the occurrences of P, in time $\mathcal{O}((m + occ) \log \log(\sigma + n/r))$ in their theoretical version.

In addition to the theoretical design, Gagie et al. and Boucher et al. [11, 2] provided a carefully engineered *r*-index implementation. The counting data structures (which find the range $\mathsf{SA}[sp..ep]$) require, for any small constant $\epsilon > 0$, $r \cdot ((1+\epsilon)\lg(n/r) + \lg\sigma + \mathcal{O}(1))$ bits (largely dominated by the described arrays `Start` and `Letter`), whereas the locating data structures (which obtain $\mathsf{SA}[ep]$, and $\mathsf{SA}[j-1]$ given $\mathsf{SA}[j]$), require $r \cdot (2\lg n + \mathcal{O}(1))$ further bits. The locating structures are then significantly heavier in practice, especially when n/r is not that large. Together, the structures use $r \cdot ((1+\epsilon)\lg(n/r) + 2\lg n + \lg\sigma + \mathcal{O}(1))$ bits of space and perform backward search steps and LF-steps in time $\mathcal{O}(\frac{1}{\epsilon}\log(\sigma + n/r))$, so they search for P in time $\mathcal{O}(\frac{1}{\epsilon}(m + occ)\log(\sigma + n/r))$.

For conciseness we do not describe the counting data structures of the *r*-index, which are the same of the rlfm-index and which we do not modify in our index. The *r*-index locating structures, which we do modify, are formed by the following components:

- $\texttt{First}[\texttt{1}..n]$: a bitvector marking with 1s the *text* positions of the letters that are the first in a BWT run. That is, if $j = 1$ or $\mathsf{BWT}[j] \neq \mathsf{BWT}[j-1]$, then $\texttt{First}[\mathsf{SA}[j]-1] = 1$. Since `First` has only r 1s, it is represented in compressed form using $r\lg(n/r) + \mathcal{O}(r)$ bits, while supporting $\texttt{rank}_1$ in time $\mathcal{O}(\log(n/r))$ and, in $\mathcal{O}(1)$ time, the operation $\texttt{select}_1(\texttt{First}, j)$ (the position of the jth 1 in `First`) [25]. This allows one find the rightmost 1 up to position i, $\texttt{pred}(\texttt{First}, i) = \texttt{select}_1(\texttt{First}, \texttt{rank}_1(\texttt{First}, i))$.
- $\texttt{FirstToRun}[\texttt{1}..r]$: a vector of integers (using $r\lceil\lg r\rceil$ bits) mapping each letter marked in `First` to the BWT run where it lies. That is, if the pth BWT run starts at $\mathsf{BWT}[j]$, and $\texttt{First}[i] = 1$ for $i = \mathsf{SA}[j] - 1$, then $\texttt{FirstToRun}[\texttt{rank}_1(\texttt{First}, i)] = p$.
- $\texttt{Samples}[\texttt{1}..r]$: a vector of $\lceil\lg n\rceil$-bit integers storing samples of SA, so that $\texttt{Samples}[p]$ is the text position $\mathsf{SA}[j]-1$ corresponding to the last letter $\mathsf{BWT}[j]$ in the pth BWT run.

These structures are used in the following way in the *r*-index implementation [11]:

Problem 1: When computing the ranges $\mathsf{SA}[sp..ep]$ along the backward search, we must also produce the value $\mathsf{SA}[ep]$. They actually compute all the values $\mathsf{SA}[ep_i]$. This is stored for $\mathsf{SA}[ep_{m+1}] = \mathsf{SA}[n]$ and then, if $\mathsf{BWT}[ep_{i+1}] = P[i]$, we know that $ep_i = \mathsf{LF}(ep_{i+1})$ and thus $\mathsf{SA}[ep_i] = \mathsf{SA}[ep_{i+1}] - 1$. Otherwise, $ep_i = \mathsf{LF}(j)$ and $\mathsf{SA}[ep_i] = \mathsf{SA}[j] - 1$, where

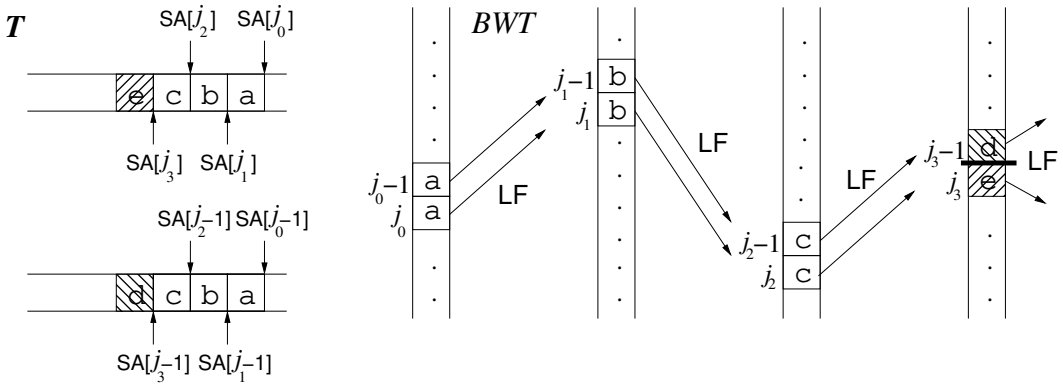

Figure 1 Schematic example of the sampling mechanism of the *r*-index. There is a run border between $j_3 - 1$ and j_3.

$j \in [sp_{i+1}..ep_{i+1}]$ is the largest position with $\mathsf{BWT}[j] = P[i]$. The position j is efficiently found with their counting data structures, and the remaining problem is how to compute $\mathsf{SA}[j]$. Since j must be an end of run, however, this is simply computed as $\texttt{Samples}[p] + 1$, where $p = \texttt{rank}_1(\texttt{Start}, j)$ is the run where j belongs.

Problem 2: When locating we must find $\mathsf{SA}[j-1]$ from $i = \mathsf{SA}[j] - 1$. There are two cases:

- $j-1$ ends a BWT run, that is, $\texttt{Start}[j] = 1$, and then $\mathsf{SA}[j-1] = \texttt{Samples}[p-1] + 1$, where p is as in Problem 1;
- $j-1$ is in the same BWT run of j, in which case they compute $\mathsf{SA}[j-1] = \phi(i)$, where[1]

$$\phi(i) = \texttt{Samples}[\texttt{FirstToRun}[\texttt{rank}_1(\texttt{First}, i)] - 1] + 1 + (i - \texttt{pred}(\texttt{First}, i)). \quad (1)$$

This formula works because, when j and $j-1$ are in the same BWT run, it holds that $\mathsf{LF}(j-1) = \mathsf{LF}(j) - 1$ [8]. Figure 1 explains why this property makes the formula work. Consider two BWT positions, $j = j_0$ and $j' = j - 1 = j_0 - 1$, that belong to the same run. The LF formula will map them to consecutive positions, j_1 and $j'_1 = j_1 - 1$. If j_1 and $j_1 - 1$ still belong to the same run, LF will map them to consecutive positions again, j_2 and $j'_2 = j_2 - 1$, and once again, j_3 and $j'_3 = j_3 - 1$. Say that j_3 and $j_3 - 1$ do not belong to the same run. This means that $j_3 - 1$ ends a run (and thus it is stored in `Samples`) and j_3 starts a run (and thus $\mathsf{SA}[j_3] - 1$ is marked in `First`). To the left of the BWT positions we show the areas of $\mathcal{T}$ virtually traversed as we perform consecutive LF-steps. Therefore, if we know $i = \mathsf{SA}[j] - 1 = \mathsf{SA}[j_0] - 1$, the nearest 1 in `First` to the left is at $\texttt{pred}(\texttt{First}, i) = \mathsf{SA}[j_3] - 1$ (where there is an `e` in $\mathcal{T}$) and $p = \texttt{FirstToRun}[\texttt{rank}(i)]$ is the number of the BWT run that starts at j_3. If we subtract 1, we have the BWT run ending at $j_3 - 1$, and then $\texttt{Samples}[p-1]$ is the position preceding $\mathsf{SA}[j_3 - 1]$ (where there is a `d` in $\mathcal{T}$). We add $1 + (i - \texttt{pred}(\texttt{First}, i)) = 4$ to obtain $\mathsf{SA}[j_0 - 1] = \mathsf{SA}[j-1]$.

These components make up, effectively, a sampling mechanism of $\mathcal{O}(r \log n)$ bits (i.e., sampling the end of runs), instead of the traditional one of $\mathcal{O}((n/s) \log n)$ bits (i.e., sampling every sth text position).

[1] The special case where $\texttt{rank}_1(\texttt{First}, i) = 0$ is handled separately.

4 Our Subsampled *r*-index

Despite its good performance on highly repetitive texts, the sampling mechanism introduced by the *r*-index is excessive in areas where the BWT runs are short, because those induce oversampled ranges on the text. In this section we describe an *r*-index variant we dub *subsampled* r-*index*, or sr-*index*, which can be seen as a hybrid between the *r*-index and the rlfm-index. The *sr*-index samples the text at end of runs (like the *r*-index), but in oversampled areas it removes some samples to ensure that no three consecutive samples lie within distance s (roughly as in the rlfm-index). It then handles text areas with denser and sparser sampling in different ways.

4.1 Subsampling

The *sr*-index subsampling process removes *r*-index samples in oversampled areas. Concretely, let $t'_1 < \cdots < t'_r$ be the text positions of the last letters in BWT runs, that is, the sorted values in array `Samples`. For any $1 < i < r$, we remove the sample t'_i if $t'_{i+1} - t'_{i-1} \leq s$, where s is a parameter. This condition is tested and applied sequentially for $i = 2, \ldots, r-1$ (that is, if we removed t'_2 because $t'_3 - t'_1 \leq s$, then we next remove t'_3 if $t'_4 - t'_1 \leq s$; otherwise we remove t'_3 if $t'_4 - t'_2 \leq s$). Let us call $t_1, t_2, \ldots$ the sequence of the remaining samples.

The arrays `First`, `FirstToRun`, and `Samples` are built on the samples t_i only. That is, if we remove the sample $\texttt{Samples}[p] = t'$, we also remove the 1 in `First` corresponding to the first letter of the $(p+1)$th BWT run, which is the one Eq. (1) would have handled with $\texttt{Samples}[p]$. We also remove the corresponding entry of `FirstToRun`. Note that, if j is the first position of the $(p+1)$th run and $j-1$ the last of the pth run, then if we remove $\texttt{Samples}[p] = \mathsf{SA}[j-1] - 1$, we remove the corresponding 1 at position $\mathsf{SA}[j] - 1$ in `First`. Finally, note that `FirstToRun` must be adapted to point to the corresponding entry of `Samples`, once some entries of the latter are removed.

It is not hard to see that subsampling avoids the excessive space usage when r is not small enough, reducing it from $\mathcal{O}(r)$ to $\mathcal{O}(\min(r, n/s))$ entries for the locating structures.

▶ **Lemma 1.** *The subsampled structures* `First`*,* `FirstToRun`*, and* `Samples` *use* $\min(r, 2\lceil n/(s+1)\rceil) \cdot (2 \lg n + \mathcal{O}(1))$ *bits of space.*

Proof. This is the same space as in the implemented *r*-index, with the number of samples reduced from r to $\min(r, 2\lceil n/(s+1)\rceil)$. We start with r samples and remove some, so there are at most r. By construction, any remaining sample t_i satisfies $t_{i+1} - t_{i-1} > s$, so if we cut the text into blocks of length $s+1$, no block can contain more than 2 samples. ◀

Our index adds the following small structure on top of the above ones, so as to mark the removed samples:

`Removed`$[1..r]$: A bitvector telling which of the original samples have been removed, that is, $\texttt{Removed}[p] = 1$ iff the sample at the end of the pth BWT run was removed. We can compute any $\texttt{rank}_1(\texttt{Removed}, p)$ in constant time using $r + o(r)$ bits [4].

It is easy to see that, once the *r*-index structures are built, the *sr*-index subsampling, as well as building and updating the associated structures, are lightweight tasks, easily carried out in $\mathcal{O}(r)$ space and $\mathcal{O}(r \log r)$ time. It is also possible to build the subsampled structures directly without building the full *sr*-index sampling first, in $\mathcal{O}(n \log(\sigma + n/r))$ time: we simulate a backward text traversal using LF-steps, so that we can build bitvector `Removed`. A second similar traversal fills the 1s in `First` and the entries in `FirstToRun` and `Samples` for the runs whose sample was not removed.

4.2 Solving Problem 1

For Problem 1, we must compute $\mathsf{SA}[j]$, where j is the end of the pth run, with $p = \mathtt{rank}_1(\mathtt{Start}, j)$. This position is sampled in the *r*-index, where the problem is thus trivial: $\mathsf{SA}[j] = \mathtt{Samples}[p] + 1$. However, in the *sr*-index it might be that $\mathtt{Removed}[p] = 1$, which means that the subsampling process removed $\mathsf{SA}[j]$. In this case, we compute $j_k = \mathsf{LF}^k(j)$ for $k = 1, 2, \ldots$ until finding a sampled value $\mathsf{SA}[j_k]$ (i.e., $j_k = n$ or $\mathtt{Start}[j_k + 1] = 1$) that is not removed (i.e., $q = \mathtt{rank}_1(\mathtt{Start}, j_k)$ and $\mathtt{Removed}[q] = 0$). We then compute $q' = q - \mathtt{rank}_1(\mathtt{Removed}, q)$, and $\mathsf{SA}[j] = \mathtt{Samples}[q'] + k + 1$.

The next lemma shows that we find a nonremoved sample for some $k < s$.

▶ **Lemma 2.** *If there is a removed sample t'_j such that $t_i < t'_j < t_{i+1}$, then $t_{i+1} - t_i \leq s$.*

Proof. Since our subsampling process removes samples left to right, by the time we removed t'_j, the current sample t_i was already the nearest remaining sample to the left of t'_j. If the sample following t'_j was the current t_{i+1}, then we removed t'_j because $t_{i+1} - t_i \leq s$, and we are done. Otherwise, there were other samples to the right of t'_j, say $t'_{j+1}, t'_{j+2}, \ldots, t'_{j+k}$, that were consecutively removed until reaching the current sample t_{i+1}. We removed t'_j because $t'_{j+1} - t_i \leq s$. Then, for $1 \leq l < k$, we removed t'_{j+l} (after having removed $t'_j, t'_{j+1}, \ldots, t'_{j+l-1}$) because $t'_{j+l+1} - t_i \leq s$. Finally, we removed t'_{j+k} because $t_{i+1} - t_i \leq s$. ◀

This implies that, from a removed sample $\mathtt{Samples}[p] = t'$, surrounded by the remaining samples $t_i < t' < t_{i+1}$, we can perform only $k = t' - t_i < s$ LF-steps until $j_k = \mathsf{LF}^{(k)}(j)$ satisfies $\mathsf{SA}[j_k] - 1 = t_i$ and thus it is stored in $\mathtt{Samples}[q]$ and not removed.

If we followed verbatim the modified backward search of the *r*-index, finding every $\mathsf{SA}[ep_i]$, we would perform $\mathcal{O}(m \cdot s)$ steps on the *sr*-index. We now reduce this to $\mathcal{O}(m + s)$ steps by noting that the only value we need is $\mathsf{SA}[ep] = \mathsf{SA}[ep_1]$. Further, we need to know $\mathsf{SA}[ep_{i+1}]$ to compute $\mathsf{SA}[ep_i]$ only in the easy case where $\mathsf{BWT}[ep_{i+1}] = P[i]$ and so $\mathsf{SA}[ep_i] = \mathsf{SA}[ep_{i+1}] - 1$. Otherwise, the value $\mathsf{SA}[ep_i]$ is computed afresh.

We then proceed as follows. We do not compute any value $\mathsf{SA}[ep_i]$ during backward search; we only remember the last (i.e., smallest) value i' of i where the computation was not easy, that is, where $\mathsf{BWT}[ep_{i'+1}] \neq P[i']$. Then, $\mathsf{SA}[ep_1] = \mathsf{SA}[ep_{i'}] - (i' - 1)$ and we need to apply the procedure described above only once: we compute $\mathsf{SA}[j]$, where j is the largest position in $[sp_{i'+1}..ep_{i'+1}]$ where $\mathsf{BWT}[j] = P[i']$, and then $\mathsf{SA}[ep_{i'}] = \mathsf{SA}[j] - 1$.

Algorithm 1 gives the complete pseudocode that solves Problem 1. Note that, if P does not occur in $\mathcal{T}$ (i.e., $occ = 0$) we realize this after the $\mathcal{O}(m)$ backward steps because some $sp_i > ep_i$, and thus we do not spend the $\mathcal{O}(s)$ extra steps.

4.3 Solving Problem 2

For Problem 2, finding $\mathsf{SA}[j-1]$ from $i = \mathsf{SA}[j] - 1$, we first proceed as in Problem 1, from $j-1$. We compute $j'_k = \mathsf{LF}^k(j-1)$ for $k = 0, \ldots, s-1$. If any of those j'_k is the last symbol of its run (i.e., $j'_k = n$ or $\mathtt{Start}[j'_k + 1] = 1$), and the sample corresponding to this run was not removed (i.e., $\mathtt{Removed}[q] = 0$, with $q = \mathtt{rank}_1(\mathtt{Start}, j'_k)$), then we can obtain immediately $\mathsf{SA}[j'_k] = \mathtt{Samples}[q'] + 1$, where $q' = q - \mathtt{rank}_1(\mathtt{Removed}, q)$, and thus $\mathsf{SA}[j-1] = \mathsf{SA}[j'_k] + k$.

Unlike in Problem 1, $\mathsf{SA}[j-1]$ is not necessarily an end of run, and therefore we are not guaranteed to find a solution for $0 \leq k < s$. However, the following property shows that, if there were some end of runs j'_k, it is not possible that all were removed from $\mathtt{Samples}$.

▶ **Lemma 3.** *If there are no remaining samples in $\mathsf{SA}[j-1] - s, \ldots, \mathsf{SA}[j-1] - 1$, then no sample was removed between $\mathsf{SA}[j-1] - 1$ and its preceding remaining sample.*

Algorithm 1 Counting pattern occurrences on the *sr*-index.

Input : Search pattern $P[1..m]$.
Output : Returns suffix array range $[sp, ep]$ for P and $\mathsf{SA}[ep]$.

$sp \leftarrow 1$; $ep \leftarrow n+1$
$i \leftarrow m$; $i' \leftarrow m+1$
while $i \geq 1$ **and** $sp \leq ep$ **do**
  $p \leftarrow \mathtt{rank}_1(\mathtt{Start}, ep)$
  if $\mathtt{Letter}[p] \neq P[i]$ **then**
    $i' \leftarrow i$; $p' \leftarrow p$
  $c \leftarrow P[i]$
  $sp \leftarrow C[c] + \mathtt{rank}_c(\mathsf{BWT}, sp-1) + 1$
  $ep \leftarrow C[c] + \mathtt{rank}_c(\mathsf{BWT}, ep)$
if $sp > ep$ **then return** "P does not occur in $\mathcal{T}$"
if $i' = m+1$ **then return** $[sp, ep]$ and $\mathsf{SA}[ep] = \mathsf{SA}[n] - m$ ($\mathsf{SA}[n]$ is stored)
$c \leftarrow P[i']$
$q \leftarrow \mathtt{select}_c(\mathtt{Letter}, \mathtt{rank}_c(\mathtt{Letter}, p'))$ (supported by the rlfm-index/*r*-index)
$j \leftarrow \mathtt{select}_1(\mathtt{Start}, q+1) - 1$
$k \leftarrow 0$
while $(j < n$ **and** $\mathtt{Start}[j+1] = 0)$ **or** $\mathtt{Removed}[q] = 1$ **do**
  $j \leftarrow \mathsf{LF}(j)$
  $q \leftarrow \mathtt{rank}_1(\mathtt{Start}, j)$
  $k \leftarrow k+1$
return $[sp, ep]$ and $\mathsf{SA}[ep] = \mathtt{Samples}[q - \mathtt{rank}_1(\mathtt{Removed}, q)] + k + 1 - (i'-1)$

Proof. Let $t_i < \mathsf{SA}[j-1] - 1 < t_{i+1}$ be the samples surrounding $\mathsf{SA}[j-1]-1$, so the remaining sample preceding $\mathsf{SA}[j-1]-1$ is t_i. Since $t_i < \mathsf{SA}[j-1] - s$, it follows that $t_{i+1} - t_i > s$ and thus, by Lemma 2, no samples were removed between t_i and t_{i+1}. ◀

This means that, if the process above fails to find an answer, then we can directly use Eq. (1), as we prove next.

▶ **Lemma 4.** *If there are no remaining samples in* $\mathsf{SA}[j-1]-s, \ldots, \mathsf{SA}[j-1]-1$*, then subsampling removed no* 1*s in* $\mathtt{First}$ *between positions* $i = \mathsf{SA}[j]-1$ *and* $\mathtt{pred}(\mathtt{First}, i)$.

Proof. Let $t_i < \mathsf{SA}[j-1] - 1 < t_{i+1}$ be the samples surrounding $\mathsf{SA}[j-1]-1$, and $k = \mathsf{SA}[j-1] - 1 - t_i$. Lemma 3 implies that no sample existed between $\mathsf{SA}[j-1]-1$ and $\mathsf{SA}[j-1]-k = t_i+1$, and there exists one at t_i. Consequently, no 1 existed in $\mathtt{First}$ between positions $\mathsf{SA}[j]-1$ and $\mathsf{SA}[j]-k$ (inclusively), and there exists one in $\mathsf{SA}[j]-1-k$. Indeed, $\mathtt{pred}(\mathtt{First}, i) = \mathsf{SA}[j]-1-k$. ◀

A final twist, which does not change the worst-case complexity but improves performance in practice, is to reuse work among successive occurrences. Let $\mathsf{BWT}[sm..em]$ be a maximal run inside $\mathsf{BWT}[sp..ep]$. For every $sm \leq j \leq em$, the first LF-step will lead us to $\mathsf{LF}(j) = \mathsf{LF}(sm) + (j - sm)$; therefore we can obtain them all with only one computation of LF. Therefore, instead of finding $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$ one by one, we report $\mathsf{SA}[ep]$ (which we know) and cut $\mathsf{BWT}[sp..ep-1]$ into maximal runs using bitvector $\mathtt{Start}$. Then, for each maximal run $\mathsf{BWT}[sm..em]$, if the end of run $\mathsf{BWT}[em]$ is sampled, we report its position and continue

Algorithm 2 Locating pattern occurrences on the *sr*-index.

Input : Global array $Res[1..occ]$ of results, range $[sp, ep]$ to report, $\mathsf{SA}[ep]$.
Output : Fills $Res[i] = \mathsf{SA}[sp-1+i]$ for all $1 \leq i \leq occ$.

```
1  Res[ep − sp + 1] ← SA[ep] (known from backward search)
2  if sp < ep then locate(sp, ep − 1, 0)
3  Proc locate(sm, em, k)
4      if k = s then
5          for im = em, ..., sm do
6              i ← Res[im − sp + 2] − 1
7              Res[im − sp + 1] ← φ(i) (Eq. (1))
8      else
9          if Start[em + 1] = 1 then
10             q ← rank_1(Start, em)
11             if Removed[q] = 0 then
12                 Res[em − sp + 1] ← Samples[q − rank_1(Removed, q)] + 1 + k
13                 em ← em − 1
14         q ← rank_1(Start, sm)
15         while sm ≤ em do
16             im ← select_1(Start, q + 1)
17             if im − 1 > em then im ← em + 1
18             locate(sm, im − 1, k + 1)
19             sm ← im
20             q ← q + 1
```

recursively reporting $\mathsf{SA}[\mathsf{LF}(sm)..\mathsf{LF}(sm)+(em-sm)-1]$; otherwise we continue recursively reporting $\mathsf{SA}[\mathsf{LF}(sm)..\mathsf{LF}(sm)+(em-sm)]$. Note that we must add k to the results reported at level k of the recursion. By Lemma 2, every end of run found in the way has been reported before level $k = s$. When $k = s$, then, we use Eq. (1) to obtain $\mathsf{SA}[em], \ldots, \mathsf{SA}[sm]$ consecutively from $\mathsf{SA}[em+1]$, which must have been reported because it is ep or was an end of run at some level of the recursion.

Algorithm 2 gives the complete procedure to solve Problem 2.

4.4 The basic index, *sr*-index$_0$

We have just described our most space-efficient index, which we call *sr*-index$_0$. Its space and time complexity is established in the next theorem.

▶ **Theorem 5.** *The* sr-*index*$_0$ *uses* $r\cdot((1+\epsilon)\lg(n/r)+\lg\sigma+\mathcal{O}(1))+\min(r, 2\lceil n/(s+1)\rceil)\cdot 2\lg n$ *bits of space, for any constant* $\epsilon > 0$*, and finds all the* occ *occurrences of* $P[1..m]$ *in* $\mathcal{T}$ *in time* $\mathcal{O}(\frac{1}{\epsilon}(m + s\cdot occ)\log(\sigma + n/r))$.

Proof. The space is the sum of the counting structures of the *r*-index and our modified locating structures, according to Lemma 1. The space of bitvector `Removed` is $\mathcal{O}(r)$ bits, which is accounted for in the formula.

As for the time, we have seen that the modified backward search requires $\mathcal{O}(m)$ steps if $occ = 0$ and $\mathcal{O}(m + s)$ otherwise (Problem 1). Each occurrence is then located in $\mathcal{O}(s)$ steps (Problem 2). In total, we complete the search with $\mathcal{O}(m + s\cdot occ)$ steps.

Each step involves $\mathcal{O}(\frac{1}{\epsilon}\log(\sigma + n/r))$ time in the basic *r*-index implementation, including Eq. (1). Our index includes additional `rank`s on `Start` and other constant-time operations, which are all in $\mathcal{O}(\log(n/r))$. Since the `First` now has $\mathcal{O}(\min(r, n/s))$ 1s, however, operation $\texttt{rank}_1$ on it takes time $\mathcal{O}(\log(n/\min(r, n/s))) = \mathcal{O}(\log\max(n/r, s)) = \mathcal{O}(\log(n/r + s))$. Yet, this `rank` is computed only once per occurrence reported, when using Eq. (1), so the total time per occurrence is still $\mathcal{O}(\log(n/r + s) + s \cdot \log(\sigma + n/r)) = \mathcal{O}(s \cdot \log(\sigma + n/r))$. ◀

Note that, in asymptotic terms, the *sr*-index is never worse than the rlfm-index with the same value of s and, with $s = 1$, it boils down to the *r*-index. Using predecessor data structures of the same asymptotic space of our lighter sparse bitvectors, the logarithmic times can be reduced to loglogarithmic [11], but our focus is on low practical space usage.

Note also that this theorem can be obtained by simply choosing the smallest between the *r*-index and the rlfm-index. In practice, however, the *sr*-index performs much better than both extremes, providing a smooth transition that retains sparsely indexed areas of $\mathcal{T}$ while removing redundancy in oversampled areas. This will be demonstrated in Section 5.

4.5 A faster and larger index, *sr*-index$_1$

The *sr*-index$_0$ guarantees locating time proportional to s and uses almost no extra space. On the other hand, on Problem 2 it performs up to s LF-steps for *every* occurrence, even when this turns out to be useless. The variant *sr*-index$_1$ adds a new component, also small, to speed up some cases:

`Valid`: a bitvector storing one bit per (remaining) sample in text order, so that $\texttt{Valid}[q] = 0$ iff there were removed samples between the qth and the $(q+1)$th 1s of `First`.

With this bitvector, if we have $i = \mathsf{SA}[j] - 1$ and $\texttt{Valid}[\texttt{rank}_1(\texttt{First}, i)] = 1$, we know that there were no removed samples between i and $\texttt{pred}(\texttt{First}, i)$ (even if they are less than s positions apart). In this case we can skip the computation of $\mathsf{LF}^k(j-1)$ of *sr*-index$_0$, and directly use Eq. (1). Otherwise, we must proceed exactly as in *sr*-index$_0$ (where it is still possible that we compute all the LF-steps unnecessarily). More precisely, this can be tested for every value between sm and em so as to report some further cells before recursing on the remaining ones, in lines 14–19 of Algorithm 2.

The space and worst-case complexities of Theorem 5 are preserved in *sr*-index$_1$.

4.6 Even faster and larger, *sr*-index$_2$

Our final variant, *sr*-index$_2$, adds a second and significantly larger structure:

`ValidArea`: an array whose cells are associated with the 0s in `Valid`. If $\texttt{Valid}[q] = 0$, then $d = \texttt{ValidArea}[q - \texttt{rank}_1(\texttt{Valid}, q)]$ is the distance from the qth 1 in `First` to the next removed sample. Each entry in `ValidArea` requires $\lceil \lg s \rceil$ bits, because removed samples must be at distance less than s from their preceding sample, by Lemma 2.

If $\texttt{Valid}[\texttt{rank}_1(\texttt{First}, i)] = 0$, then there was a removed sample at $\texttt{pred}(\texttt{First}, i) + d$, but not before. So, if $i < \texttt{pred}(\texttt{First}, i) + d$, we can still use Eq. (1); otherwise we must compute the LF-steps $\mathsf{LF}^k(j-1)$ and we are guaranteed to succeed in less than s steps. This improves performance considerably in practice, though the worst-case time complexity stays as in Theorem 5 and the space increases by at most $r \lg s$ bits.

5 Experimental Results

We implemented the *sr*-index in C++14, on top of the SDSL library[2], and made it available at `https://github.com/duscob/sr-index`.

We benchmarked the *sr*-index against available implementations for the *r*-index, the rlfm-index, and several other indexes for repetitive text collections.

Our experiments ran on a hardware with two Intel(R) Xeon(R) CPU E5-2407 processors at 2.40 GHz and 250 GB RAM. The operating system was Debian Linux kernel `4.9.0-14-amd64`. We compiled with full optimization and no multithreading.

Our reported times are the average user time over 1000 searches for patterns of length $m = 10$ obtained at random from the texts. We give space in bits per symbol (bps) and times in microseconds per occurrence (μs/occ). Indexes that could not be built on some collection, or that are out of scale in space or time, are omitted in the corresponding plots.

5.1 Tested indexes

We included the following indexes in our benchmark; their space decrease as s grows:

sr-index: Our index, including the three variants, with sampling values $s = 4, 8, 16, 32, 64$.
r-index: The *r*-index implementation we build on.[3]
rlcsa: An implementation of the run-length CSA [19], which outperforms the actual rlfm-index implementation.[4] We use text sampling values $s = n/r \times f/8$, with $f = 8, 10, 12, 14, 16$.
csa: An implementation of the CSA [28], which outperforms in practice the fm-index [8, 9]. This index, obtained from SDSL, acts as a control baseline that is not designed for repetitive collections.We use text sampling parameter $s = 16, 32, 64, 128$.
g-index: The best grammar-based index implementation we are aware of [5].[5] We use Patricia trees sampling values $s = 4, 16, 64$.
lz-index and lze-index: Two variants of the Lempel–Ziv based index [15].[6]
hyb-index: A hybrid between a Lempel–Ziv and a BWT-based index [7].[7] We build it with parameters $M = 8, 16$, the best for this case.

5.2 Collections

We benchmark various repetitive text collections; Table 1 gives some basic measures on them.

PizzaChili: A generic collection of real-life texts of various sorts and repetitiveness levels, which we use to obtain a general idea of how the indexes compare. We use 4 collections of microorganism genomes (influenza, cere, para, and escherichia) and 4 versioned document collections (the English version of einstein, kernel, worldleaders, coreutils).[8]
Synthetic DNA: A 100KB DNA text from PizzaChili, replicated 1,000 times and each copied symbol mutated with a probability from 0.001 (DNA-001, analogous to human assembled genomes) to 0.03 (DNA-030, analogous to sequence reads). We use this collection to study how the indexes evolve as repetitiveness decreases.

[2] From `https://github.com/simongog/sdsl-lite`.
[3] From `https://github.com/nicolaprezza/r-index`.
[4] From `https://github.com/adamnovak/rlcsa`.
[5] From `https://github.com/apachecom/grammar_improved_index`.
[6] From `https://github.com/migumar2/uiHRDC`.
[7] From `https://github.com/hferrada/HydridSelfIndex`.
[8] From `http://pizzachili.dcc.uchile.cl/repcorpus/real`.

Table 1 Basic characteristics of the repetitive texts used in our benchmark. Size is given in MB.

Collection	Size	n/r	Collection	Size	n/r
influenza	147.6	51.2	DNA-001	100.0	142.4
cere	439.9	39.9	DNA-003	100.0	58.3
para	409.4	27.4	DNA-010	100.0	26.0
escherichia	107.5	7.5	DNA-030	100.0	11.6
einstein	447.7	1611.2	HLA	53.7	161.4
kernel	238.0	92.4	Chr19	2,819.3	89.2
worldleaders	44.7	81.9	Salmonella	3,840.5	43.9
coreutils	195.8	43.8	Reads	2,565.5	8.9

Real DNA: Some real DNA collections to study other aspects:

HLA: A dataset with copies of the short arm (p arm) of human chromosome 6 [27].[9] This arm contains about 60 million base pairs (Mbp) and it includes the 3 Mbp HLA region. That region is known to be highly variable, so the *r*-index sampling should be sparse for most of the arm and oversample the HLA region.

Chr19 and Salmonella: Human and bacterial assembled genome collections, respectively, of a few billion base pairs. We include them to study how the indexes behave on more massive data. Chr19 is the set of 50 human chromosome 19 genomes taken from the 1000 Genomes Project [30], whereas Salmonella is the set of 815 Salmonella genomes from the GenomeTrakr project [29].

Reads: A large collection of sequence reads, which tend to be considerably less repetitive than assembled genomes.[10] We include this collection to study the behavior of the indexes on a popular kind of bioinformatic collection with mild repetitiveness. In Reads the sequencing errors have been corrected, and thus its $n/r \approx 9$ is higher than the $n/r \approx 4$ reported on crude reads [6].

5.3 Results

Figures 2 and 3 show the space taken by all the indexes and their search time.

A first conclusion is that *sr*-index$_2$ always dominates *sr*-index$_0$ and *sr*-index$_1$, so we will refer to it simply as *sr*-index from now on. The plots show that the extra information we associate to the samples makes a modest difference in space, while time improves considerably. This *sr*-index can be almost as fast as the *r*-index, and an order of magnitude faster than all the others, while using 1.5–4.0 less space than the *r*-index. Therefore, as promised, we are able to remove a significant degree of redundancy in the *r*-index without affecting its outstanding time performance.

In all the PizzaChili collections, the *sr*-index dominates almost every other index, outperforming them both in time and space. The only other index on the Pareto curve is the hyb-index, which can use as little as a half of the space of the sweet spot of the *sr*-index, but still at the price of being an order of magnitude slower. This holds even on escherichia, where n/r is well below 10, and both the rlcsa and the csa become closer to the *sr*-index.

In general, in all the collections with sufficient repetitiveness, say n/r over 25, the *sr*-index sharply dominates as described. As repetitiveness decreases, with n/r reaching around 10, the rlcsa and the csa approach the *sr*-index and outperform every other repetitiveness-aware index, as expected. This happens on escherichia (as mentioned) and Reads (where the *sr*-index,

[9] From `ftp://ftp.ebi.ac.uk/pub/databases/ipd/imgt/hla/fasta/hla_gen.fasta`.
[10] From `https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=ERR008613`.

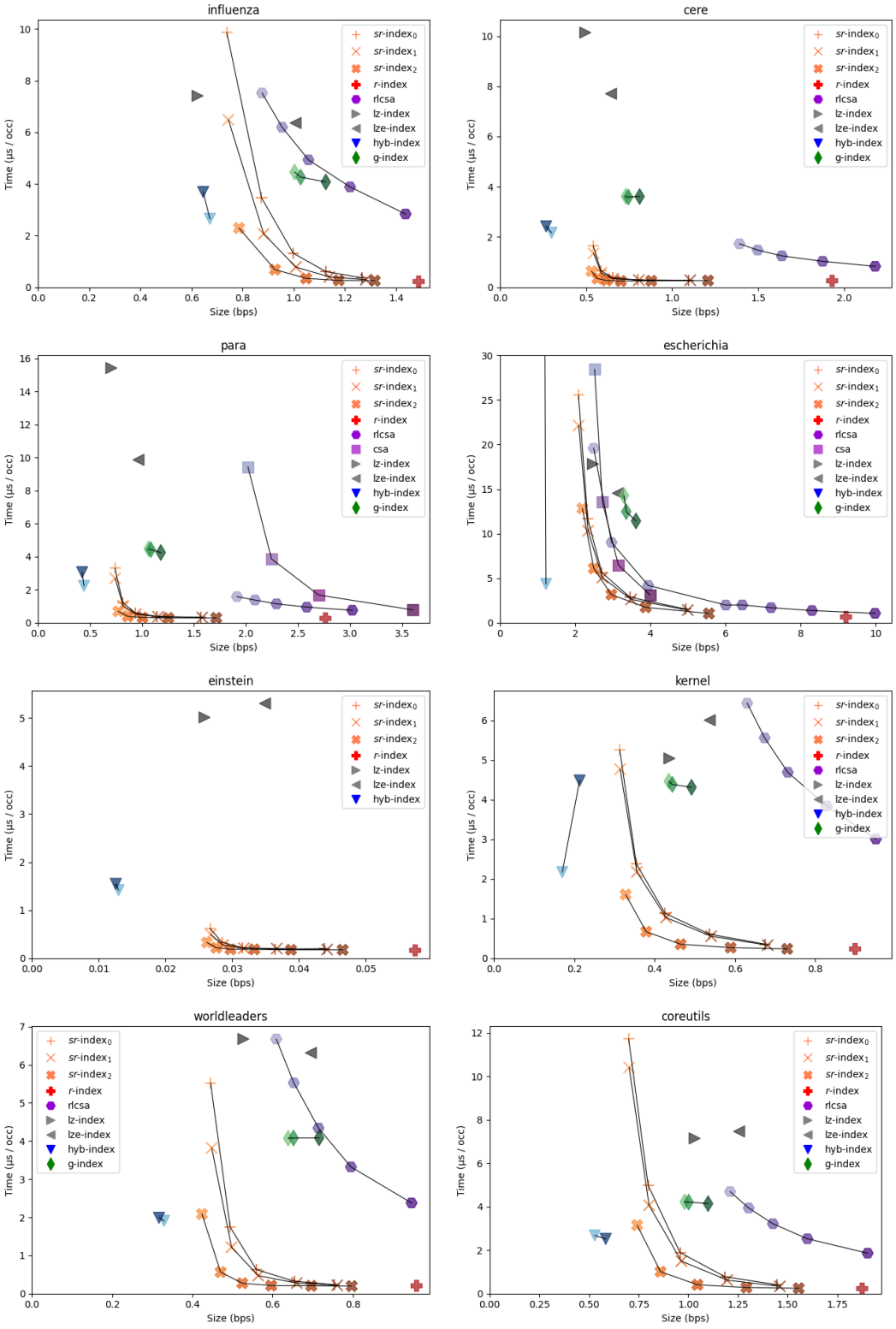

Figure 2 Space-time tradeoffs for the PizzaChili collections.

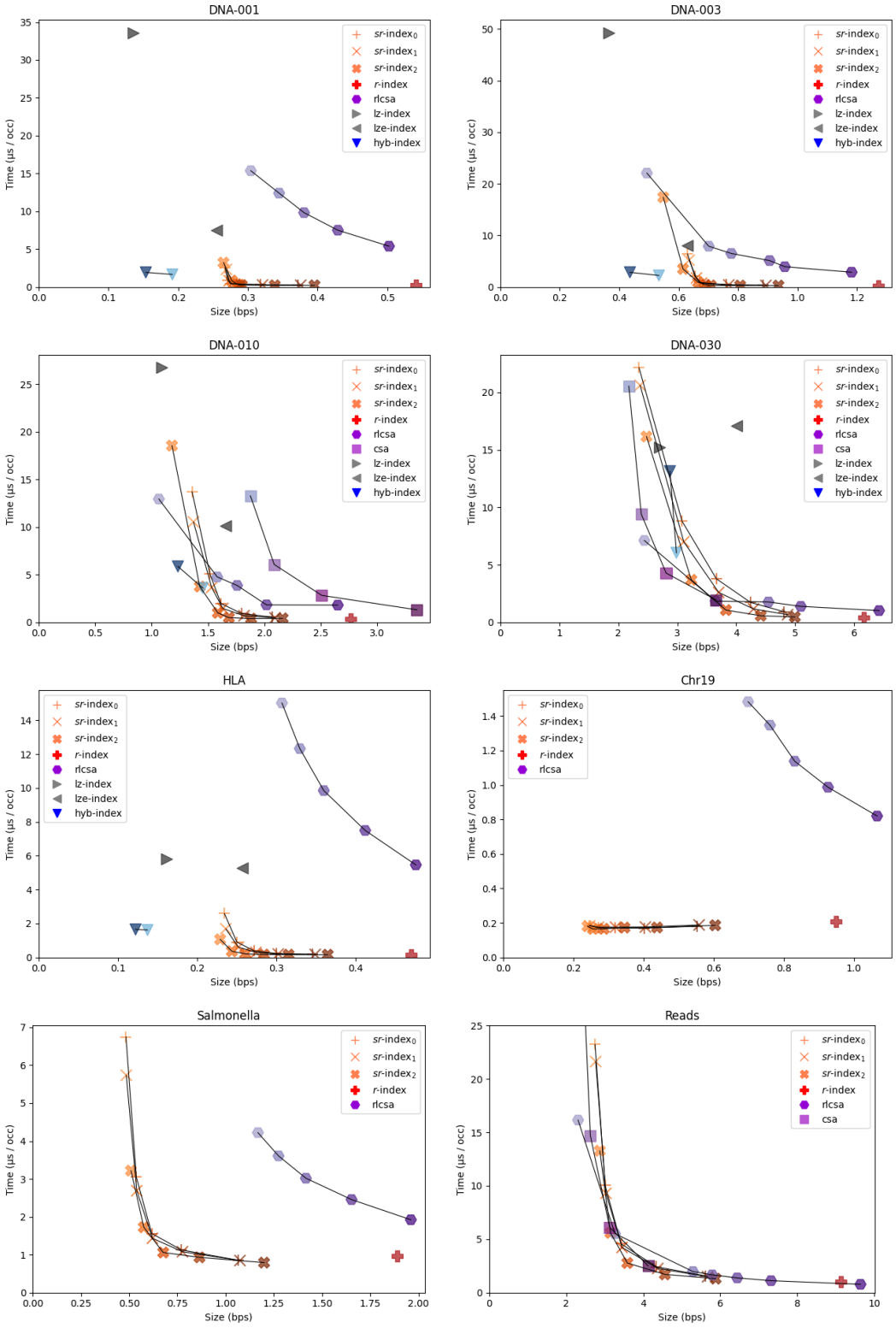

Figure 3 Space-time tradeoffs for the synthetic and real DNA datasets.

the rlcsa, and the csa behave similarly). This is also the case on the least repetitive synthetic DNA collection, DNA-030, where the mutation rate reaches 3%. In this collection, the repetitiveness-unaware csa largely dominates all the space-time map.

We expected the *sr*-index to have a bigger advantage over the *r*-index on the HLA dataset because its oversampling is concentrated, but the results are similar to those on randomly mutated DNA with about the same n/r value (DNA-001). In general, the bps used by the *sr*-index can be roughly predicted from n/r; for example the sweet spot often uses around $40r$ total bits, although it takes $20r$–$30r$ bits in some cases. The *r*-index uses $70r$–$90r$ bits.

The bigger collections (Chr19, Salmonella, Reads), on which we could build the BWT-related indexes only, show that the same observed trends scale to gigabyte-sized collections of various repetitiveness levels.

6 Conclusions

We have introduced the *sr*-index, an *r*-index variant that solves the problem of its relatively bloated space while retaining its high search performance. The *sr*-index is orders of magnitude faster than the other repetitiveness-aware indexes, while outperforming most of them in space as well. It matches the time performance of the *r*-index while using 1.5–4.0 less space.

Unlike the *r*-index, the *sr*-index uses little space even in milder repetitiveness scenarios, which makes it usable in a wider range of bioinformatic applications. For example, it uses 0.25–0.60 bits per symbol (bps) while reporting each occurrence within a microsecond on gigabyte-sized human and bacterial genomes, where the original *r*-index uses 0.95–1.90 bps. In general, the *sr*-index outperforms classic compressed indexes on collections with repetitiveness levels n/r over as little as 7 in some cases, though in general it is reached by repetitiveness-unaware indexes when n/r approaches 10, which is equivalent to a DNA mutation rate around 3%.

Compared to the rlfm-index, which for pattern searching is dominated by the *sr*-index, the former can use its regular text sampling to compute any entry of the suffix array or its inverse in time proportional to the sampling step s. Obtaining an analogous result on the *sr*-index, for example to implement compressed suffix trees, is still a challenge. Other proposals for accessing the suffix array faster than the rlfm-index [12, 26] illustrate this difficulty: they require even more space than the *r*-index.

References

1 Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–72, 2021.

2 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):13:1–13:15, 2019.

3 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

4 David R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

5 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

6 Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31st Data Compression Conference (DCC)*, pages 93–102, 2021.

7 Héctor Ferrada, Dominik Kempa, and Simon J. Puglisi. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8, 2018.

8 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

9 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed Representations of Sequences and Full-text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

10 Travis Gagie and Gonzalo Navarro. *Compressed Indexes for Repetitive Textual Datasets*, pages 475–480. Springer, 2019.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

12 Rodrigo González, Gonzalo Navarro, and Héctor Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.

13 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In *Proc. 61st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020.

14 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

15 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

16 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

17 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

18 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

19 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

20 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

21 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

22 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

23 Gonzalo Navarro and Víctor Sepúlveda. Practical indexing of repetitive collections using Relative Lempel–Ziv. In *Proc. 29th Data Compression Conference (DCC)*, pages 201–210, 2019.

24 Takaaki Nishimoto and Yasuo Tabei. Faster queries on BWT-runs compressed indexes. *CoRR*, 2006.05104, 2020. `arXiv:2006.05104`.

25 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

26 Simon J. Puglisi and Bella Zhukova. Relative Lempel–Ziv compression of suffix arrays. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 89–96, 2020.

27 James Robinson, Dominic J. Barker, Xenia Georgiou, Michael A. Cooper, Paul Flicek, and Steven G. E. Marsh. IPD-IMGT/HLA Database. *Nucleic Acids Research*, 48(D1):D948–D955, 10 2019.

28 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

29 Eric L. Stevens, Ruth Timme, Eric W. Brown, Marc W. Allard, Errol Strain, Kelly Bunning, and Steven Musser. The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, 8:808, 2017.

30 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.