Fast and Small Subsampled R-indexes

DUSTIN COBAS, CeBiB — Center for Biotechnology and Bioengineering, Chile and University of Chile, Chile

TRAVIS GAGIE, CeBiB — Center for Biotechnology and Bioengineering, Chile and Dalhousie University, Canada

GONZALO NAVARRO, CeBiB — Center for Biotechnology and Bioengineering, Chile and University of Chile, Chile

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 in query time. Its space usage, $O(r)$ where r is the number of runs in the Burrows–Wheeler Transform of the text, is however higher 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 $O(\min(r, n/s))$ 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 theoretical analysis falls short in describing the practical advantages of the *sr*-index, because it performs much better on real texts than on synthetic ones: the *sr*-index retains the performance of the *r*-index while using 1.5–4.0 times less space, sharply outperforming *virtually every other* compressed index on repetitive texts in both time and space. Only a particular Lempel–Ziv-based index uses less space—about half—than the *sr*-index, but it is an order of magnitude slower.

Our second contribution are the *r*-csa and *sr*-csa indexes. Just like the *r*-index adapts the well-known FM-Index to repetitive texts, the *r*-csa adapts Sadakane's Compressed Suffix Array (CSA) to this case. We show that the principles used on the *r*-index turn out to fit naturally and efficiently in the CSA framework. The *sr*-csa is the corresponding subsampled version of the *r*-csa. While the CSA performs better than the FM-Index on classic texts with alphabets larger than DNA, our experiments show that the *sr*-csa outperforms the *sr*-index on repetitive texts not only over those larger alphabets, but on some DNA texts as well.

Overall, our new subsampled indexes sweep the table of the existing indexes for highly repetitive text collection, by combining the exceptional speed of the *r*-index with drastically reduced storage use.

Authors' addresses: Dustin Cobas, CeBiB — Center for Biotechnology and Bioengineering, Chile and University of Chile, Dept. of Computer Science, Chile, dustin.cobas@gmail.com; Travis Gagie, CeBiB — Center for Biotechnology and Bioengineering, Chile and Dalhousie University, Canada, travis.gagie@gmail.com; Gonzalo Navarro, CeBiB — Center for Biotechnology and Bioengineering, Chile and University of Chile, Dept. of Computer Science, Chile, gnavarro@dcc.uchile.cl.

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 [15, 35].

Traditional compressed indexes rely on statistical compression [36], but this is ineffective to capture repetitiveness [26]. A new wave of repetitiveness-aware indexes [35] build on other compression mechanisms like Lempel–Ziv [27] or grammar compression [25]. A particularly useful index of this kind is the rlfm-index [31, 33], because it emulates the classical suffix array [34] and this simplifies translating suffix-array based algorithms to run on it [30].

The rlfm-index represents the Burrows–Wheeler Transform (BWT) [4] 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 [24]. A problem with the rlfm-index is that, although it can count the number of occurrences of a pattern using $O(r)$ 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 $O(n/s)$ additional space incurred on a text of length n ruins the compression on very repetitive collections, where $r \ll n$. The recent r-index [16] closed the long-standing problem of efficiently locating the occurrences within $O(r)$ 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 $O(z)$, where z is the number of phrases in the Lempel–Ziv parse. Gagie et al. [16] 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 [26], a baseline Lempel–Ziv 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 [3, 7, 10, 38]. Indeed, while in some realistic cases n/r can be over 1,500, 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 just 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 $O(\min(r, n/s))$ samples while its locating machinery can still be

used to guarantee that every pattern occurrence is located within $O(s)$ 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 $O((m + occ)\log(\sigma + n/r))$ in the implemented *r*-index to $O((m + s \cdot occ)\log(\sigma + n/r))$ 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, retaining the time performance of the *r*-index while using 1.5–4.0 times less space, and sharply dominating the rlfm-index, the best grammar-based index [7], and the lz-index, both in space and time. Its only remaining competitor is a hybrid between a Lempel–Ziv based and a statistical index [11]. 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 small space—generally less—in a wide range of repetitiveness scenarios.

For historical reasons, the *r*-index was developed on top of the rlfm-index, which performs best on small alphabets like DNA. Another well-known alternative to the rlfm-index, the rlcsa [31, 33], performs better on larger alphabets but suffers from the same space-time tradeoff: one needs to spend $O(n/s)$ space in order to report each occurrence in time proportional to s. Our second contribution is to adapt the *r*-index mechanisms to run on the rlcsa, to obtain what we dub *r-csa*. It turns out that the techniques used on the *r*-index apply naturally and efficiently to the rlcsa data structures, leading to a space- and time-efficient index. We further apply the subsampling mechanism of the *sr*-index to the *r*-csa to obtain the *subsampled r-csa*, or *sr*-csa. Our experiments show that the *sr*-csa outperforms the *sr*-index on texts over large alphabets, as well as on some repetitive DNA collections.

Overall, the development of the *sr*-index and the *sr*-csa represents a major improvement in the state of the art of compressed indexes for highly repetitive text collections. By combining the speed of the *r*-index, which was by far the fastest index but used significantly more space than others, with a drastic reduction in space that does not sacrifice time, our new subsampled indexes sharply dominate most of the existing actors in compressed text indexing.

A conference version of this paper appeared in *Proc. CPM 2021* [8]. This article contains more detailed explanations, more extensive experiments, improved implementations, and the full development of the *r*-csa and *sr*-csa indexes.

2 BACKGROUND

2.1 Suffix arrays

The *suffix array* [34] $\mathsf{SA}[1..n]$ 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 $1 \leq i < n$. For technical convenience we assume that $\mathcal{T}[n] = \$$, a special terminator symbol that is smaller than every other symbol in $\mathcal{T}$. The suffix array can be binary searched in time $O(m \log n)$ to obtain the range $\mathsf{SA}[sp..ep]$ of all the suffixes prefixed by a search pattern $P[1..m]$. Once this range is determined, the occurrences of P can be *counted* (i.e., return their number $occ = ep - sp + 1$ of occurrences in $\mathcal{T}$), and also *located* (i.e., returning their positions in $\mathcal{T}$) in time $O(occ)$ by simply listing their starting positions, $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$. The suffix array can then be stored in $n\lceil \log n \rceil$ bits[1] (plus the $n\lceil \log \sigma \rceil$ bits to store $\mathcal{T}$) and we say it *searches* (i.e., counts and locates) for P in $\mathcal{T}$ in total time $O(m \log n + occ)$. Its main drawback is that its space usage is too high to maintain it in main memory for current text collections.

2.2 Compressed suffix arrays

Compressed suffix arrays (CSAs) [36] 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 $\mathsf{SA}[i]$ in time $\mathrm{t}_{\mathrm{lookup}}(n)$, so they can be used to search for P in time $O(\mathrm{t}_{\mathrm{search}}(m) + occ\, \mathrm{t}_{\mathrm{lookup}}(n))$.

2.2.1 Ψ–based CSAs. Grossi and Vitter [21] and Sadakane [45] introduced CSAs based on another permutation, $\Psi[1..n]$, related to the suffix array:

$$\Psi(i) = \mathsf{SA}^{-1}[(\mathsf{SA}[i] \bmod n) + 1],$$

that is, $\Psi(i) = j$ such that $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[j] = \mathsf{SA}[i] + 1$. Ψ is then a permutation of $[1..n]$ where $\Psi(i)$ holds the position of $\mathsf{SA}[i] + 1$ in SA, which allows us virtually move forward in $\mathcal{T}$ from SA: if $\mathsf{SA}[i]$ points to $\mathcal{T}[j]$, then $\mathsf{SA}[\Psi(i)]$ points to $\mathcal{T}[j+1]$. Sadakane's CSA [45], which we call simply csa, adds a bitvector $D[1..n]$ that marks with $D[i] = 1$ the σ positions $\mathsf{SA}[i]$ where the first symbol of the suffixes changes, and an $o(n)$-space data structure [6] that computes in constant time $\mathsf{rank}(D, i)$, the number of 1s in $D[1..i]$. This allows us reading any string pointed from $\mathsf{SA}[j]$: the consecutive symbols are $\mathsf{rank}(D, i)$, $\mathsf{rank}(D, \Psi(i))$, $\mathsf{rank}(D, \Psi(\Psi(i)))$, ... so we can compare P with the suffix pointed from any suffix array position along the binary search in $O(m)$ time, and thus find the suffix array range $\mathsf{SA}[sp..ep]$ in time $\mathrm{t}_{\mathrm{search}}(m) = O(m \log n)$.

For locating, we must be able to compute any $\mathsf{SA}[i]$. The csa stores the SA values that point to text positions that are a multiple of s, for a space-time tradeoff parameter s. A bitvector $B[1..n]$

[1]We use log to denote the binary logarithm.

with rank support is used to mark with $B[i] = 1$ the sampled positions $\mathsf{SA}[i]$, so a sampled entry $\mathsf{SA}[i]$ is stored at position $\mathsf{rank}(B, i)$ of a sampled array. If $\mathsf{SA}[i]$ is not sampled, the csa tries $\mathsf{SA}[\Psi(i)]$, $\mathsf{SA}[\Psi^2(i)]$, and so on, until it finds a sampled $\mathsf{SA}[\Psi^k(i)]$ (i.e., $B[\Psi^k(i)] = 1$), for some $k < s$. It then holds that $\mathsf{SA}[i] = \mathsf{SA}[\Psi^k(i)] - k$, so the csa supports $\mathrm{t}_{\mathrm{lookup}}(n) = O(s)$. The csa then searches in time $O(m \log n + s \cdot occ)$.

Compression is obtained thanks to the regularities of permutation Ψ. For example, because Ψ is increasing in the area of the suffixes starting with the same symbol, it can be represented within the zero-order statistical entropy of $\mathcal{T}$, while supporting constant-time access [45] (more complex Ψ-based CSAs obtain higher-order entropy space [21]). To this space, we must add the $O(n)$ bits for bitvectors D and B, and the $\lfloor n/s \rfloor \log n$ bits for the samples of SA. The text $\mathcal{T}$ is *not* stored.

Mäkinen and Navarro [31] observed another regularity of Ψ: it features *runs* of consecutive values, that is, $\Psi(i+1) = \Psi(i) + 1$. They designed the so-called *Run-Length CSA*, or rlcsa, which aimed to use $O(r_\Psi)$ space, where r_Ψ is the number of maximal runs in Ψ. It was soon noted that r_Ψ is particularly small on repetitive text collections, which enabled space reductions that are much more significant than those obtained via statistical entropy [33].

Function Ψ was represented in $O(r_\Psi \log n)$ bits by encoding the runs $\Psi(i..i+l) = \Psi(i), \Psi(i)+1, \ldots, \Psi(i)+l$ as the pair $\langle \Psi(i), l \rangle$. The time to access Ψ increases, using modern predecessor data structures [2], to $O(\log \log_w(n/r_\Psi))$, where w is the size in bits of the computer word (we give the details in Section 3.1). By also representing bitvectors D and B with predecessor data structures, the rlcsa searches in time $O((m \log n + s \cdot occ) \log \log_w(\sigma + s + n/r_\Psi))$. The total space of the rlcsa is then $O((r_\Psi + n/s) \log n)$ bits. In highly repetitive text collections, the term n/s overshadows r_Ψ and ruins the high compression achieved by collapsing the runs in Ψ.

2.2.2 BWT*–based CSAs.* The *Burrows–Wheeler Transform* [4] of $\mathcal{T}$ is a permutation $\mathsf{BWT}[1..n]$ of the symbols 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 [12, 13] is a CSA that represents SA and $\mathcal{T}$ within the high-order 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

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

where $c = P[i]$, $C[c]$ is the number of occurrences of symbols smaller than c in $\mathcal{T}$, and $\mathsf{rank}_c(\mathsf{BWT}, j)$ is the number of times c occurs in $\mathsf{BWT}[1..j]$. Thus, $[sp, ep] = [sp_1, ep_1]$ if $sp_i \leq ep_i$ holds for all $1 \leq i \leq m$, otherwise P does not occur in $\mathcal{T}$.

For locating the occurrences $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$, the fm-index samples SA just like the csa. The function used to traverse the text towards a sampled position is the so-called LF-*step*, which simulates a backward traversal of $\mathcal{T}$: if $\mathsf{SA}[i] = j$, the value i' such that $\mathsf{SA}[i'] = j - 1$ is $\mathsf{LF}(i)$, where

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

where $c = \mathsf{BWT}[i]$. Note that $\mathsf{LF}(i)$ is the inverse function of $\Psi(i)$. Starting from $\mathsf{SA}[i]$, we compute LF successively until, for some $0 \leq k < s$, we find a sampled entry $\mathsf{SA}[\mathsf{LF}^k(i)]$, which is stored explicitly. It then holds $\mathsf{SA}[i] = \mathsf{SA}[\mathsf{LF}^k(i)] + k$.

By implementing BWT with a wavelet tree [20], for example, access and rank_c on BWT can be supported in time $O(\log \sigma)$, and the fm-index searches in time $O((m + s \cdot occ) \log \sigma)$ [13]. With more sophisticated wavelet tree representations [23, 32], the space of the fm-index is the high-order entropy of $\mathcal{T}$ plus the $O((n/s) \log n)$ bits for the sampling of SA.

The *Run-Length FM-index*, rlfm-index [31, 33] is an adaptation of the fm-index aimed at repetitive texts, just like the rlcsa is to the csa. Say that the $\mathsf{BWT}[1..n]$ is formed by r maximal *runs of equal symbols*, then it holds that r is small in repetitive collections (in particular, it holds $r_\Psi \leq r \leq r_\Psi + \sigma$ [31]). For example, it is now known that $r = O(z \log^2 n)$, where z is the number of phrases of the Lempel–Ziv parse of $\mathcal{T}$ [24].

The rlfm-index supports counting within $O(r \log n)$ bits, by implementing the backward search over data structures that use space proportional to the number of BWT- runs. It marks in a bitvector $\mathtt{Start}[1..n]$ with 1s the positions i starting BWT-runs, that is, where $i = 1$ or $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$. 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 $r \log(n/r) + O(r)$ bits, so that any bit $\mathtt{Start}[i]$ and $\mathtt{rank}(\mathtt{Start}, i)$ are computed in time $O(\log(n/r))$ [40]. We then simulate $\mathsf{BWT}[j] = \mathtt{Letter}[\mathtt{rank}(\mathtt{Start}, j)]$ in $O(r \log n)$ bits. The backward search formula can be efficiently simulated as well, by adding another bitvector that records the run lengths in lexicographic order. Overall, the search time becomes $O((m + s \cdot occ) \log(\sigma + n/r))$ (by replacing the sparse bitvectors with predecessor data structures and using an alternative to wavelet trees [18], one can reach $O((m + s \cdot occ) \log\log(\sigma + n/r))$). The rlfm-index still uses SA samples to locate, however, and when $r \ll n$ (i.e., on repetitive texts), the $O((n/s) \log n)$ added bits ruin the $O(r \log n)$-bit space (unless one accepts high locating times by setting $s \approx r$).

The *r*-index [16] closed the long-standing problem of efficiently locating the pattern occurrences using $O(r \log n)$-bit space. The experiments showed that the *r*-index outperforms all the other

implemented indexes by orders of magnitude in space or in search time on highly repetitive datasets. However, other experiments on more typical repetitiveness scenarios [3, 7, 10, 38] 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 $g = O(z \log(n/z))$) is usually slower but significantly smaller [7], and an even slower Lempel–Ziv based index of size $O(z)$ [26] is even smaller. Some later proposals [39] further speed up the r-index by increasing the constant accompanying the $O(r \log n)$-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 *R*-CSA: A Ψ-BASED INDEX FOR REPETITIVE TEXTS

In this section we introduce the r-csa, an equivalent to the r-index based on the rlcsa. We first describe the counting algorithm and data structures, which is just a modern version of those given in the original rlcsa [33]. We then show how to locate within $O(r)$ space, by translating the techniques of the r-index [16] to this scenario.

3.1 Counting in $O(r)$ space

Let $\Psi_c = \Psi[i..j]$ be the range in Ψ corresponding to each symbol $c \in [1..\sigma]$, such that all the suffixes of $\mathcal{T}$ starting with c are in the range $\mathsf{SA}[i..j]$. As said, Ψ_c is strictly increasing over $[1..n]$; let us say that it contains r_c maximal runs of consecutive values. By definitions of Ψ and BWT, if $\Psi_c(i) = k$ then $\mathsf{BWT}[k] = \mathcal{T}[\mathsf{SA}[i]] = c$. Consequently, a one-to-one relation exists between the pth Ψ_c run and the pth BWT-run of symbol c, for $1 \leq p \leq r_c$. It then follows that $r = \sum_{c=1}^{\sigma} r_c$.[2] We call Ψ-runs those r maximal runs in Ψ that are inside some range Ψ_c.

A backward search process was also devised for the csa [44] and used in the rlcsa [33]. Once the range $\mathsf{SA}[sp_{i+1}..ep_{i+1}]$ for $P[i+1..m]$ is known, we obtain $\mathsf{SA}[sp_i..ep_i]$ by binary searching within Ψ_c, for $c = P[i]$, the maximal range of positions j such that $\mathsf{SA}[j] \in [sp_{i+1}..ep_{i+1}]$. Indeed, those are the suffixes that start with $c = P[i]$ and follow with $P[i+1..m]$. This technique yields the same $O(m \log n)$ counting time, but has better locality of reference.

As in the the rlcsa, we represent each Ψ-run $\Psi(i..i+l)$ as a pair $\langle \Psi(i), l \rangle$. Unlike the original rlcsa, we construct a predecessor data structure $\mathcal{P}_\Psi$ on the r Ψ-run heads within the universe σn by concatenating all the σ ranges Ψ_c. Specifically, the predecessor $\mathcal{P}_\Psi$ stores $x + (c-1)n$ if $x = \Psi(i)$ is a Ψ-run head in Ψ_c, allowing us to compute the Ψ-run that contains or precedes a given value $y \in [1..n]$. Thus, the predecessor operation pred_c on Ψ_c is defined in terms of a classic predecessor

[2] Some sequences of consecutive values in Ψ can extend beyond the limit of the corresponding range Ψ_c. This is why it holds $r_\Psi \leq r$, which becomes an equality if we split those runs of Ψ by allowing only runs inside each Ψ_c.

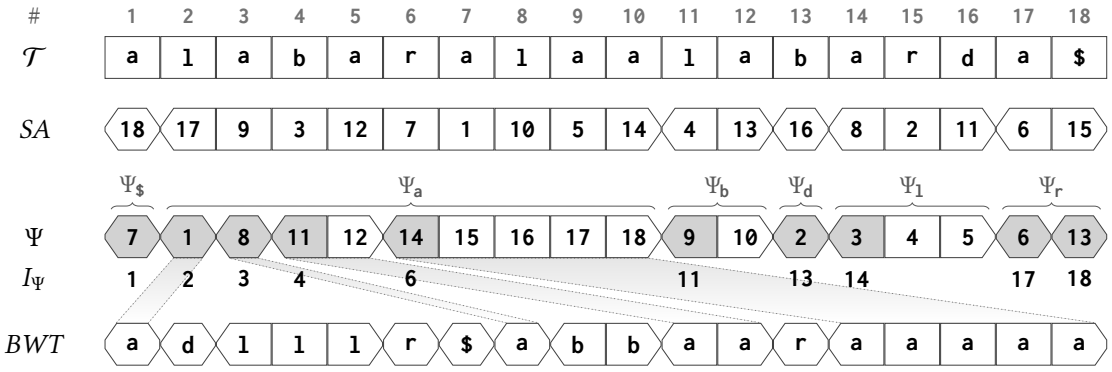

Fig. 1. Data structures for the counting mechanism of the r-csa on an example text. The blocks in SA cover the suffixes starting with the same symbol, which align with the areas of each Ψ_c. The blocks in Ψ and BWT represent runs. The gray cells in Ψ are the r run heads. The stripes show the relation between each run of Ψ_a and its corresponding BWT-run.

function pred on the Ψ-run heads, as

$$\mathsf{pred}_c(y) = \mathsf{pred}(\mathcal{P}_\Psi, y + (c-1)n) = \langle x, k \rangle$$

where x is the actual predecessor value in $\mathcal{P}_\Psi$, and k is its Ψ-run rank (i.e., the number of Ψ-runs up to the one that starts with value x). Using recent predecessor structures [2, Thm. A.1] to represent $\mathcal{P}_\Psi$, we use $O(r \log(n/r))$ bits of space and answer queries in $O(\log\log_w(\sigma n/r))$ time.

In addition, we associate each Ψ-run head $x = \Psi(i)$ with its global position i in Ψ in an array $I_\Psi[1..r]$, where $I_\Psi[j] = i$ iff $\Psi(i)$ is the first item of the jth Ψ-run. I_Ψ is used to support the backward search, computing the length of Ψ-runs and of each new range in SA. Structure $I_\Psi[1..r]$ replaces the array $C[1..\sigma]$ of the original rlcsa [33]. Figure 1 illustrates definitions and relations.

Algorithm 1 computes the suffix array range $A[sp..ep]$ of the occurrences of a pattern P in the text $\mathcal{T}$, using the predecessor structure $\mathcal{P}_\Psi$ and the array I_Ψ. Note that it must consider the cases where the answer is within a Ψ-run or not. This yields the following result.

Theorem 3.1 (Counting with r-csa). *The r-csa of a text $\mathcal{T}[1..n]$ over alphabet $[1..\sigma]$, with r Ψ-runs, can be represented using $O(r \log n)$ bits of space and count the occurrences of any pattern $P[1..m]$ in $O(m \log\log_w(\sigma n/r))$ time, where w is the computer word size.*

3.2 Locating in $O(r)$ space

Following the r-index [16], we reduce the problem of locating the occurrences of pattern P with the r-csa to two subproblems: (1) maintaining the text position of $\mathrm{SA}[sp_i]$ along the backward

Algorithm 1: Counting pattern occurrences with r-csa.

Input : Query pattern $P[1..m]$.
Output: Range $\langle sp, ep\rangle$ on SA for P.

```
1  function count(P[1..m])
2      ⟨sp, ep⟩ ← ⟨1, n⟩
3      for i ← m downto 1 do
4          c ← P[i]
5          sp ← findNextStartPos(sp, c)
6          ep ← findNextEndPos(ep, c)
7          if sp > ep then
8              return "P is not in T"
9      return ⟨sp, ep⟩

10 function findNextStartPos(sp, c)
11     sp' ← sp + (c − 1) · n
12     ⟨x, k⟩ ← pred(P_Ψ, sp')
13     if sp' < x + (I_Ψ[k + 1] − I_Ψ[k]) then
14         return I_Ψ[k] + (sp' − x)
15     return I_Ψ[k + 1]

16 function findNextEndPos(ep, c)
17     ep' ← ep + (c − 1) · n
18     ⟨x, k⟩ ← pred(P_Ψ, ep')
19     if ep' < x + (I_Ψ[k + 1] − I_Ψ[k]) then
20         return I_Ψ[k] + (ep' − x)
21     return I_Ψ[k + 1] − 1
```

search, (2) finding $\mathsf{SA}[j+1]$ given $\mathsf{SA}[j]$. After the backward search, then, we know $\mathsf{SA}[sp]$ by (1) and then find $\mathsf{SA}[sp+1], \mathsf{SA}[sp+2], \ldots, \mathsf{SA}[ep]$ with (2).

3.2.1 Counting with toehold. In the same vein as the r-index [16, Lem. 3.2], we show how to enhance the backward search so that we always know $\mathsf{SA}[sp_i]$ (called the "toehold"). We give a proof that this can be done that is better suited for practical Ψ-based indexes.

Lemma 3.2. *The rlcsa backward search process on Ψ can be enhanced to retrieve, along with the range $\langle sp, ep\rangle$ on* SA *for the pattern $P[1..m]$, the toehold value* $\mathsf{SA}[sp]$ *in $O(1)$ additional time per backward step and with $O(r \log n)$ additional bits of space.*

Proof. We store in a new array $F_{\mathsf{SA}}[1..r]$ the text positions of the Ψ-run heads, that is, $F_{\mathsf{SA}}[j] = \mathsf{SA}[i]$ iff the jth Ψ-run begins at position i. The backward search initiates with the entire interval $\langle sp_{m+1}, ep_{m+1}\rangle = \langle 1, n\rangle$ of SA. The initial toehold is then $\mathsf{SA}[sp_{m+1}] = \mathsf{SA}[1] = F_{\mathsf{SA}}[1] = n$.

Let $\langle sp_{i+1}, ep_{i+1}\rangle$ be the range on SA for the occurrences of $P[i+1..m]$, with $\mathsf{SA}[sp_{i+1}]$ being a known value. As described in Algorithm 1, the function `findNextStartPos` relies on the operation $\mathsf{pred}(\mathcal{P}_\Psi, sp_{i+1}, P[i]) = \langle x, k\rangle$ to compute the starting position sp_i of the interval for $P[i..m]$.

Extending `findNextStartPos` to additionally obtain the value $\mathsf{SA}[sp_i]$ when the pattern occurs in $\mathcal{T}$ results in two possible cases. If the kth Ψ-run contains the value sp_{i+1}, then $sp_{i+1} = \Psi(sp_i)$ because the first value of the range $\mathsf{SA}[sp_{i+1}..ep_{i+1}]$ is preceded by $P[i]$. Thus the next toehold is

Algorithm 2: Counting pattern occurrences and finding value SA[sp] with r-csa.

Input : Query pattern $P[1..m]$.
Output : Range $\langle sp, ep\rangle$ on SA for P; Value SA[sp].

```
1  function count(P[1..m])
2      ⟨sp, ep, v⟩ ← ⟨1, n, n⟩
3      for i ← m downto 1 do
4          c ← P[i]
5          ⟨v, sp⟩ ← findStartToehold(v, sp, c)
6          ep ← findNextEndPos(ep, c)
7          if sp > ep then
8              return "P is not in T"
9      return ⟨sp, ep, v⟩
10 function findStartToehold(v, sp, c)
11     sp' ← sp + (c − 1) · n
12     ⟨x, k⟩ ← pred(P_Ψ, sp')
13     if sp' < x + (I_Ψ[k + 1] − I_Ψ[k]) then
14         return ⟨v − 1, I_Ψ[k] + (sp' − x)⟩
15     return ⟨F_SA[k + 1], I_Ψ[k + 1]⟩
```

straightforwardly calculated as $\mathsf{SA}[sp_i] = \mathsf{SA}[sp_{i+1}] - 1$. If, instead, sp_{i+1} does not belong to the kth Ψ-run, then sp_i is the head of the $(k+1)$th Ψ-run. Using the F_{SA} array, the next toehold is computed as $\mathsf{SA}[sp_i] = F_{\mathsf{SA}}[k+1]$, also in constant time. □

Algorithm 2 gives the corresponding pseudocode.

3.2.2 Locating from toehold. While it is possible to employ a sampling scheme nearly identical to that utilized by the r-index, we opt for an alternative one that virtually moves forwards in the text, rather than backwards. This choice is influenced by the nature of the Ψ function, which enables more efficient forward than backward traversal.

Lemma 3.3. *Given a text position* $\mathsf{SA}[i]$*, let* $\Psi[l]$ *be the tail of the* Ψ*-run with the smallest text position* $\mathsf{SA}[l] \geq \mathsf{SA}[i]$*. Then,*

$$\mathsf{SA}[i+1] = \mathsf{SA}[l+1] + (\mathsf{SA}[l] - \mathsf{SA}[i]). \tag{1}$$

Proof. There are two possible cases. The first case, where $\Psi[i]$ is the last symbol of a Ψ-run, is trivial because $i = l$. For the second case, where $\mathsf{SA}[i]$ is not the last symbol of a Ψ-run, let $\Delta = \mathsf{SA}[l] - \mathsf{SA}[i]$, that is, $l = \Psi^{\Delta}(i)$. By the definition of l, it holds for all $0 \leq p < \Delta$ that $\Psi^p(i)$ is *not* the last element of a Ψ-run. Consequently, for all $0 \leq p \leq \Delta$, it holds that

$$\Psi^p(i) + 1 = \Psi^p(i+1).$$

This shows that each pair $\Psi^p(i)$ and $\Psi^p(i+1)$ are adjacent positions within a Ψ-run until the position $l = \Psi^{\Delta}(i)$ is reached. That is, text positions $\mathsf{SA}[\Psi^p(i)]$ and $\mathsf{SA}[\Psi^p(i+1)]$ traverse forward

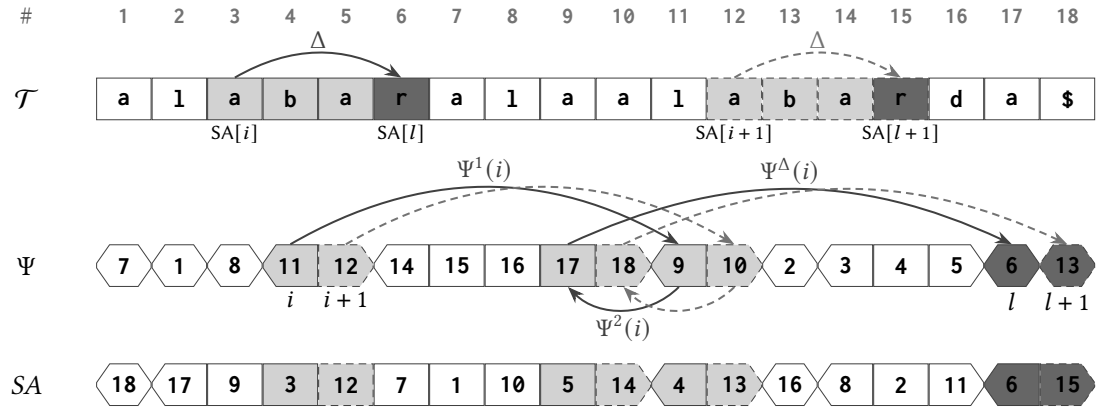

Fig. 2. Example of the sampling mechanism of the *r*-csa. The arrows in $\mathcal{T}$ show the Δ distance from the given $\mathsf{SA}[i]$ to its Ψ-run last element successor $\Psi[l]$. In Ψ, each run is represented by a block; solid arrows are Ψ steps for i; and dashed arrows are Ψ steps for $i+1$.

together in the suffix array for Δ steps, so

$$\mathsf{SA}[i+1] = \mathsf{SA}[\Psi^{\Delta}(i+1)] - \Delta = \mathsf{SA}[\Psi^{\Delta}(i)+1] - \Delta = \mathsf{SA}[l+1] + (\mathsf{SA}[l] - \mathsf{SA}[i]),$$

where the first equality holds just by definition of Ψ. Figure 2 illustrates the proof. □

Kärkkäinen et al. [22] defined the function Φ that returns $\mathsf{SA}[i-1]$ for the given text position $\mathsf{SA}[i]$. Gagie et al. [16] later added the function Φ^{-1} returning $\mathsf{SA}[i+1]$. This last function is formally defined below.

Definition 3.4 (Gagie et al. [16]). Function Φ^{-1} is a permutation of $[1..n]$ such that, for any text position j and its related position i in the suffix array (i.e., $j = \mathsf{SA}[i]$), is defined as

$$\Phi^{-1}(j) = \begin{cases} \mathsf{SA}[\mathsf{SA}^{-1}[j]+1] = \mathsf{SA}[i+1], & \text{if } i < n \\ \mathsf{SA}[1] = n, & \text{if } i = n \end{cases} \tag{2}$$

Gagie et al. [16] show how to store the permutations Φ and Φ^{-1} in $O(r \log n)$ bits of space using a predecessor data structure. We achieve a similar result based on Lemma 3.3, yet using a successor function over the text positions of Ψ-run tails.

Lemma 3.5. *The function Φ^{-1} can be evaluated in $O(\log \log_w(n/r))$ time using an $O(r \log n)$-bits successor data structure.*

Proof. Let L be the set of r text positions such as $x = \mathsf{SA}[l] \in L$ iff $\Psi[l]$ is the last element in its Ψ-run, and $\mathcal{S}_L$ be a successor function over the values of L. Also, each value $\mathsf{SA}[l] \in \mathcal{S}_L$

is paired with the text position $y = \mathsf{SA}[l+1]$ of its next Ψ-run head. Given a value $j = \mathsf{SA}[i]$, if $\langle x, y\rangle = \mathcal{S}_L(j)$, then $\mathsf{SA}[i+1] = y + (x - j)$ by Lemma 3.3.

A recent predecessor data structure [2, Thm. A.1] represents $\mathcal{S}_L$ within $O(r \log n)$ bits and answers successor queries in time $O(\log\log_w(n/r))$. □

Algorithm 3: Locating pattern occurrences with r-csa.

Input : Query pattern $P[1..m]$.
Output: Occurrences of P: $V[1..occ] = \mathsf{SA}[sp..ep]$.

```
1 function locate(P[1..m])
2     ⟨sp, ep, v⟩ ← count(P)
3     V[1] ← v
4     for i ← 2 to ep − sp + 1 do
5         V[i] ← Φ^{-1}(V[i − 1])
6     return V
```

Algorithm 3 shows how Φ^{-1} is used to compute all the occurrences of P given the first one. We have now arrived at the primary outcome of this section, which is stated in the following form for compatibility with the r-index, using that $O(\log(\sigma + n/r)) = O(\log(\sigma n/r))$.

Theorem 3.6 (Locating with r-csa). *The r-csa of a text $\mathcal{T}[1..n]$ over alphabet $[1..\sigma]$, with r Ψ-runs, can be represented within $O(r \log n)$ bits and locate the occ occurrences of a pattern $P[1..m]$ in $O(m \log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$ time, where w is the computer word size.*

3.3 Practical design

While the theoretical result yields $O(r \log n)$ space without full details on the constants, a finer design is needed in order to obtain a space-competitive data structure, even if it does not yield the same time complexities.

Following Mäkinen et al. [33] (and Sadakane [45]), we decompose Ψ into σ partial functions Ψ_c, one per symbol c. Because each Ψ_c is strictly increasing, we differentially encode the first and last values of the Ψ-runs, using δ-codes to represent the differences. To accelerate access to the function Ψ, we sample every B-th absolute value, creating a reduced sequence $\widehat{\Psi_c}$. Parameter B yields a tradeoff between space and time to access Ψ: one spends $\lfloor r/B \rfloor \log n$ bits on the samples and accesses any cell of Ψ in time $O(B)$, by accessing the preceding cell of $\widehat{\Psi_c}$ and then decoding up to B δ-codes. Further, function $\mathsf{pred}(\mathcal{P}_\Psi, i)$ can be computed in time $O(\log(r/B) + B) = O(\log r + B)$, by binary searching the samples $\widehat{\Psi_c}$ and then decoding up to B values.

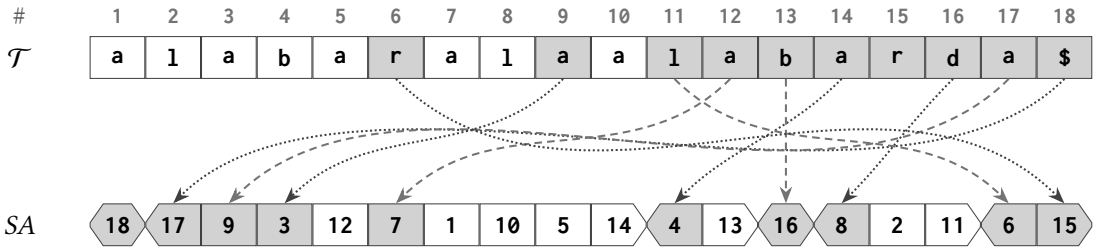

Fig. 3. Practical data structures used for locating mechanism of the r-csa. The gray cells in $\mathcal{T}$ are the r elements marked in the bitvector $\mathsf{L}_{\mathcal{T}}$. The gray cells in SA are the r samples stored in the array F_{SA}. The arrows show the mapping $\mathsf{Map}_{\mathsf{LF}}$ (different arrow styles are used to improve visualization).

This compressed representation of Ψ requires

$$r \cdot (\log(\sigma n/r) + \log(n/r) + O(\log\log(\sigma n/r))) + O((r/B)\log n) + O(\sigma \log n)$$

bits of space. The initial term is the worst-case size of the run-length encoding of Ψ, using δ-codes to store the length of each Ψ-run and the gap between them (i.e., the distance between the first value of a Ψ-run and the last value of the preceding one). The second term covers the first value $\Psi(i)$ of every Bth run, and their absolute ranks. The last term represents the array $C[1..\sigma]$ (used instead of array $I_\Psi[1..r]$ in our practical proposal), and additional samples of Ψ for the first element in each partial Ψ_c.

The second aspect to consider is the practical implementation of function Φ^{-1}. This relies on several components; Figure 3 illustrates their relation.

$\mathsf{L}_{\mathcal{T}}[1..n]$**:** a bitvector marking with $\mathsf{L}_{\mathcal{T}}[j] = 1$ the text positions $j = \mathsf{SA}[i]$ where $\Psi(i)$ is the last symbol of a Ψ-run. Since $\mathsf{L}_{\mathcal{T}}$ has only r 1s, it is represented in compressed form using $r\log(n/r) + O(r)$ bits, while supporting $\mathsf{rank}(\mathsf{L}_{\mathcal{T}}, i)$ in time $O(\log(n/r))$ and, in $O(1)$ time, the operation $\mathsf{select}(\mathsf{L}_{\mathcal{T}}, p)$ (the position of the pth 1 in $\mathsf{L}_{\mathcal{T}}$) [40]. This allows one to find the leftmost 1 from position i as

$$\mathsf{succ}(\mathsf{L}_{\mathcal{T}}, i) = \mathsf{select}_1(\mathsf{L}_{\mathcal{T}}, \mathsf{rank}_1(\mathsf{L}_{\mathcal{T}}, i-1) + 1).$$

$\mathsf{Map}_{\mathsf{LF}}[1..r]$**:** an array of integers (using $r\lceil \log r \rceil$ bits) mapping each text position marked in $\mathsf{L}_{\mathcal{T}}$ to the related sample in F_{SA}. Note that, if $\mathsf{L}_{\mathcal{T}}[j] = 1$ with $j = \mathsf{SA}[l]$, then there exists p such that $\mathsf{F}_{\mathsf{SA}}[p] = \mathsf{SA}[l+1]$, because $\Psi(l)$ is the last symbol in a Ψ-run.[3] We find it with

$$p = \mathsf{map}(\mathsf{L}_{\mathcal{T}}, j) = \mathsf{Map}_{\mathsf{LF}}[\mathsf{rank}_1(\mathsf{L}_{\mathcal{T}}, j-1) + 1].$$

[3]In the particular case where $l = n$ (that is, $j = \mathsf{SA}[l]$ is the last symbol of the final Ψ-run), the associated sample in F_{SA} is $p = \mathsf{map}(\mathsf{L}_{\mathcal{T}}, j) = \mathsf{Map}_{\mathsf{LF}}[r] = 1$, which corresponds to the Ψ-run of the special symbol \$.

If we apply this formula on a text position j that does not correspond to the end of a run, it will return the run number p of the next text position that corresponds to the end of a run. Using also the F_{SA} array in addition to the components mentioned above, we can then compute the function Φ^{-1} as follows:

$$\Phi^{-1}(j) = \mathsf{F}_{\mathsf{SA}}[\mathsf{map}(\mathsf{L}_{\mathcal{T}}, j)] - (\mathsf{succ}(\mathsf{L}_{\mathcal{T}}, j) - j). \tag{3}$$

A straightforward examination of the preceding data structures reveals that they collectively yield the following result.

Theorem 3.7 (Practical r-csa). *The practical r-csa of a text $\mathcal{T}[1..n]$ over alphabet $[1..\sigma]$, with r Ψ-runs, is represented using*

$$r \cdot \left(2\log n + 2\log(n/r) + \log\sigma + O(\log\log(\sigma n/r))\right) + O((r/B)\log n) + O(\sigma \log n)$$

bits and can locate the occ occurrences of a pattern $P[1..m]$ in $O(m(\log r + B) + occ\log(n/r))$ time, where B is the block size for the representation of Ψ.

4 SUBSAMPLED BWT/Ψ-BASED INDEXES

While the r-index and r-csa sampling mechanisms perform very well on highly repetitive texts, they can be less efficient in areas where the run heads or tails split the text into many short blocks. The text is sampled too frequently in such areas, creating unnecessary redundancy in the indexes. Figure 4 illustrates an analysis of commonly used datasets confirming that these oversampled areas indeed arise in various types of text.

The existence of those short blocks in the texts is to be expected. Consider, in the particular case of DNA sequences, the site of a single-nucleotide polymorphism, where some genomes have A and the others have G, in all cases followed by the same string α and preceded by the same symbols $\beta_m \cdots \beta_1$. Those As and Gs are likely to be intermingled in the BWT area of the suffixes that start with α, but the characters β_1 will be separated into two BWT areas: those of the suffixes $\mathtt{A}\alpha$ and $\mathtt{G}\alpha$. The same will happen to the characters β_2 (separated in the areas of the suffixes $\beta_1\mathtt{A}\alpha$ and $\beta_1\mathtt{G}\alpha$), β_3, and so on. But for large enough m, $\beta_m \cdots \beta_1$ will be unique in the collection, and the suffix array areas of $\beta_m \cdots \beta_1\mathtt{A}\alpha$ and $\beta_m \cdots \beta_1\mathtt{G}\alpha$ will be merged again.

The character β_1 preceding (in the genomes) the first/last (in the BWT) of those As and of those Gs is quite likely to be the start/end of a run in the BWT, and so are the corresponding characters β_2, β_3, and so on, until the two areas merge. It follows that, if a character in the genomes is at a boundary between runs in the BWT, then the character immediately to its left is quite likely to be as well. In other words, the characters at boundaries between runs in the BWT, will tend to cluster in the genomes.

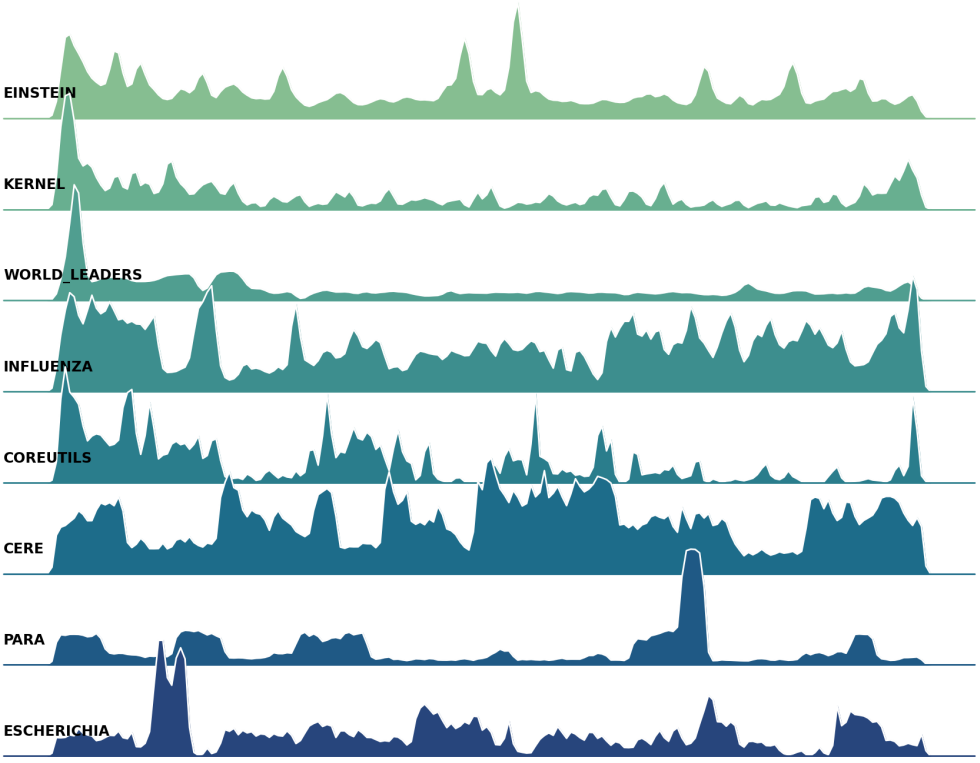

Fig. 4. Distribution of BWT-run heads within the Pizza&Chili repetitive texts [14]. The x-axis represents the positions of these run heads along the text. The y-axis shows the density, which indicates how frequently BWT-run heads appear at different locations. The smooth curve is obtained using the statistical method Kernel Density Estimation (KDE).

This section introduces two new indexing schemes for repetitive texts: the *subsampled r-index* (*sr*-index) and the *subsampled r-csa* (*sr*-csa). Both are built by combining aspects of existing methods. The *sr*-index is a hybrid between the *r*-index and the rlfm-index. The *sr*-csa, on the other hand, combines the *r*-csa with the rlcsa.

For the sake of clarity, we will employ the designation *sr-indexes* to refer to our two subsampled solutions (*sr*-index and *sr*-csa), *r-indexes* to mean both BWT/Ψ-runs based indexes (*r*-index and *r*-csa), and *rl-indexes* to represent the run-length based indexes rlfm-index and rlcsa.

Similarly to their corresponding *r*-indexes, *sr*-indexes take text position samples at the beginning or end of each run. Yet, they remove samples from text areas where there are too many of them, ensuring that no three samples in a row are closer together than a certain distance (defined by a parameter s). This approach is a relaxation of the regular text sampling used in the *rl*-indexes,

where consecutive samples are separated exactly by s text positions. Unlike *rl*-indexes, some consecutive samples can be very far apart in some areas of the text, but unlike *r*-indexes, *sr*-indexes ensure that they are never too close to each other. To achieve this goal, the *sr*-indexes face various challenges related to maintaining correctness and efficiency upon removal of samples, in particular ensuring, like *rl*-indexes, that they never require more than s steps to simulate a backward step or computation of an entry of SA.

Our *sr*-index and *sr*-csa are based on a similar design, with the primary distinction being the use of the Φ or Φ^{-1} function, respectively. The Φ function employs a predecessor data structure to locate the remaining values in $\mathsf{SA}[sp..ep-1]$, whereas the Φ^{-1} function relies on the successor to compute $\mathsf{SA}[sp+1..ep]$ (recall that the LF function of the *r*-index is the inverse of the Ψ function of the *r*-csa). To avoid redundant explanations, we will focus our attention on the *sr*-index, making pertinent remarks when differences with the *sr*-csa are significant and require further explanation. We will directly present the practical data structures that implement the subsampled indexes.

4.1 Subsampling

The *r*-index locating structures are formed by the following components, analogous to those of the *r*-csa we described in Section 3.3. Our subsampling solutions will later modify them.

- $\mathsf{L}_{\mathsf{SA}}[1..r]$**:** an array of r sampled text positions, where $\mathsf{L}_{\mathsf{SA}}[p] = \mathsf{SA}[i]-1$ iff $\mathsf{BWT}[i]$ is the last letter in the pth BWT-run.
- $\mathsf{F}_{\mathcal{T}}[1..n]$**:** a bitvector marking with 1s the *text* positions of the letters that are the first in a BWT-run. That is, if $i=1$ or $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$, then $\mathsf{F}_{\mathcal{T}}[\mathsf{SA}[i]-1]=1$. This allows one to find the rightmost 1 up to position j,
$$\mathsf{pred}(\mathsf{F}_{\mathcal{T}}, j) = \mathsf{select}(\mathsf{F}_{\mathcal{T}}, \mathsf{rank}(\mathsf{F}_{\mathcal{T}}, j)),$$
with $\mathsf{select}(\mathsf{F}_{\mathcal{T}}, 0) = 0$.
- $\mathsf{Map}_{\mathsf{FL}}[1..r]$**:** an array mapping each letter marked in $\mathsf{F}_{\mathcal{T}}$ to the BWT-run preceding the one in which it is located. If $\mathsf{F}_{\mathcal{T}}[j]=1$ with $j=\mathsf{SA}[i]-1$, then there exists p such that $\mathsf{L}_{\mathsf{SA}}[p] = \mathsf{SA}[i-1]-1$, because $\mathsf{BWT}[i]$ is the first letter in a BWT-run,[4] and
$$p = \mathsf{map}(\mathsf{F}_{\mathcal{T}}, j) = \mathsf{Map}_{\mathsf{FL}}[\mathsf{rank}(\mathsf{F}_{\mathcal{T}}, j)],$$
where $\mathsf{Map}_{\mathsf{FL}}[0]$ yields the BWT-run preceding to the run of the special symbol \$.

The *r*-index computes the Φ function as
$$\Phi(j) = \mathsf{L}_{\mathsf{SA}}[\mathsf{map}(\mathsf{F}_{\mathcal{T}}, j-1)] + (j - \mathsf{pred}(\mathsf{F}_{\mathcal{T}}, j-1)). \tag{4}$$

[4]Note that i cannot belong to the first run (i.e., $i=1$), as then we would be on the suffix $\mathcal{T}[\mathsf{SA}[1]] = \$$.

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 L_{SA}. For any $1 < i < r$, the sample t'_i is removed if $t'_{i+1} - t'_{i-1} \leq s$, where s is a parameter. This condition is tested and applied sequentially for $i = 2, \ldots, r-1$. If, for example, we removed t'_2 because $t'_3 - t'_1 \leq s$, then we next remove t'_3 if $t'_4 - t'_1 \leq s$; if we had not removed t'_2, then we remove t'_3 if $t'_4 - t'_2 \leq s$. Let us designate $t_1, t_2, \ldots$ as the sequence of the *remaining* samples.

The structures $\mathsf{F}_{\mathcal{T}}$, $\mathsf{Map}_{\mathsf{FL}}$, and L_{SA} are constructed exclusively on the remaining subsamples t_i. Consequently, the removal of the sample $\mathsf{L}_{\mathsf{SA}}[p] = t'$ also entails the removal of the 1 in $\mathsf{F}_{\mathcal{T}}$ corresponding to the first letter of the $(p+1)$th BWT-run, which is the very instance that Eq. (4) would have addressed with $\mathsf{L}_{\mathsf{SA}}[p]$. In other words, if i is the first position of the $(p+1)$th run and $i-1$ the last of the pth run, then if we remove $\mathsf{L}_{\mathsf{SA}}[p] = \mathsf{SA}[i-1]-1$, we remove the corresponding 1 at position $\mathsf{SA}[i]-1$ in $\mathsf{F}_{\mathcal{T}}$. In addition, the corresponding entry of $\mathsf{Map}_{\mathsf{FL}}$ is also removed. Finally, note that $\mathsf{Map}_{\mathsf{FL}}$ must be adapted to point to the corresponding entry of L_{SA}, once some entries of the latter are removed.

Subsampling proves to be an effective method for avoiding the excessive space required to store the locating structures. This is particularly beneficial when the number of BWT-runs r is a relatively large value. In such cases, subsampling can reduce the entries in those data structures from $O(r)$ to $O(\min(r, n/s))$ in the worst case (the reduction is much higher in practice because the samples are not uniformly distributed, as seen in Figure 4).

Lemma 4.1. *The subsampled structures* L_{SA}, $\mathsf{F}_{\mathcal{T}}$ *and* $\mathsf{Map}_{\mathsf{FL}}$ *use* $\min(r, 2\lceil n/(s+1)\rceil) \cdot (2\log n + O(1))$ *bits of space.*

Proof. If x is the number of remaining samples, then for each remaining sample array L_{SA} uses $\lceil \log n \rceil$ bits, bitvector $\mathsf{F}_{\mathcal{T}}$ uses $\log(n/x) + O(1)$ bits [40], and $\mathsf{Map}_{\mathsf{FL}}$ uses $\lceil \log x \rceil$ bits. The combined size of the three arrays is then $x \cdot (2\log n + O(1))$ bits. This is the same space as in the implemented *r*-index [16], with the number of samples reduced from r to x.

Our subsampling process begins with r samples and subsequently removes a subset of them, thus ensuring that the number of samples never exceeds r. By construction, any remaining sample t_i is guaranteed to satisfy $t_{i+1} - t_{i-1} > s$, so if we cut the text into blocks of length $s+1$, no block can contain more than 2 samples. Therefore, $x \leq \min(r, 2\lceil n/(s+1)\rceil)$. □

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

`Del`$[1..r]$**:** a bitvector telling which of the original samples have been removed. Specifically, `Del`$[p] = 1$ iff the sample at the end of the pth BWT-run was removed. We can compute

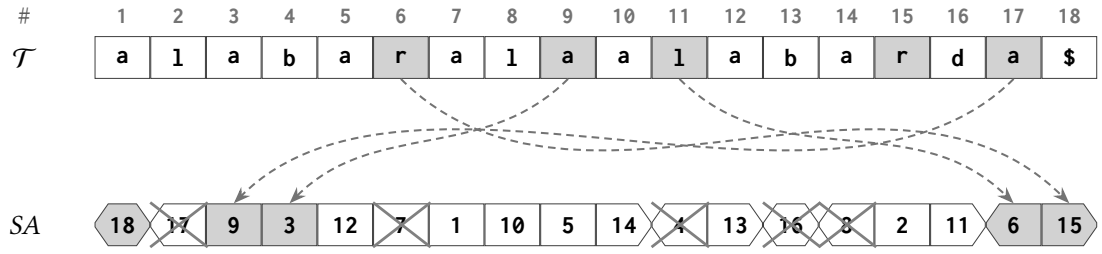

Fig. 5. Subsampled data structures used for the locating mechanism of the *sr*-csa using a sampling factor $s = 4$ (compare with the full sampling in Figure 3). The gray cells in $\mathcal{T}$ are the subsampled elements marked in the bitvector $\mathsf{L}_{\mathcal{T}}$. The gray cells in SA are the remaining samples stored in the array F_{SA}. The crossed cells in SA are the removed samples marked in the bitvector $\mathtt{Del}$. The arrows show the mapping $\mathsf{Map}_{\mathsf{LF}}$.

any $\mathtt{rank}(\mathtt{Del}, p)$ in constant time using $r + o(r)$ bits [6], as well as $\mathtt{rank}_0(\mathtt{Del}, p) = p - \mathtt{rank}(\mathtt{Del}, p)$ (which counts the 0s in $\mathtt{Del}[1..p]$).

Figure 5 illustrates the sampling scheme on our running example.

Construction. Once the basic structures of the *r*-index able to perform LF-steps are built, we can create the additional structures $\mathsf{F}_{\mathcal{T}}$, L_{SA}, $\mathsf{Map}_{\mathsf{FL}}$, and $\mathtt{Del}$ in two additional virtual backward passes over the text, in $O(n \log(\sigma + n/r))$ time and without extra space. A first pass performs the subsampling, in text order, thereby defining the bitvector $\mathtt{Del}$. Once we know the number of remaining samples, a second traversal fills the 1s in $\mathsf{F}_{\mathcal{T}}$ and the entries in $\mathsf{Map}_{\mathsf{FL}}$ and L_{SA} for the runs whose sample was not removed.

Running on the sr-csa. A nearly identical sampling process is used on the *sr*-csa, using the samples in the F_{SA} array. Recall that, unlike the *r*-index, these samples are the text positions of the first element in each Ψ-run. Given this condition along with the nature of the *sr*-csa, which relies on the Ψ and $\mathtt{succ}(\mathsf{L}_{\mathcal{T}}, i)$ functions, we have opted to implement a slight variation. Specifically, the subsampling mechanism employs a backward iteration over the samples $i = r - 1, \ldots, 2$ in F_{SA}. It is straightforward to verify that this change retains the results given above.

The construction process is also similar, using two forward text traversals simulated using the Ψ function. The construction time is $O(n \log B)$.

4.2 Counting with Toehold

The counting algorithm of the practical *r*-index is based on the data structures of a rlfm-index variant called *sparse RLBWT* [41, Thm. 28]. By applying a sparsification strategy primarily to the $\mathtt{Start}$ bitvector, this structure requires only $r \cdot ((1 + \epsilon) \log(n/r) + \log \sigma + O(1))$ bits (largely

dominated by the described arrays Start and Letter) for any small constant $\epsilon > 0$. The time complexity for backward search steps and LF-steps is $O((1/\epsilon)\log(\sigma + n/r))$.

To obtain the necessary text position or *toehold* along the backward search, the practical *r*-index maintains the value $\mathsf{SA}[ep_i]$ along each interval $[sp_i..ep_i]$, for $1 \leq i \leq m$. In the non-trivial cases where $P[i] \neq \mathsf{BWT}[ep_{i+1}]$, the end of interval is $ep_i = \mathsf{LF}(j)$, where $j \in [sp_{i+1}..ep_{i+1}]$ is the largest position with $\mathsf{BWT}[j] = P[i]$. It is easy to see that j must be the end of a BWT-run, in particular of the pth run, with $p = \mathsf{rank}_1(\mathtt{Start}, j)$. As we do not know j, this run can be computed as $p = \mathsf{select}_c(\mathtt{Letter}, \mathsf{rank}_c(\mathtt{Letter}, \mathsf{rank}_1(\mathtt{Start}, ep_{i+1})))$. Finding $\mathsf{SA}[ep_i]$ then requires a straightforward lookup process in the *r*-index, since it is precomputed and stored in array L_{SA}, where it holds $\mathsf{SA}[ep_i] = \mathsf{SA}[j] - 1 = \mathsf{L}_{\mathsf{SA}}[p]$.

However, the *sr*-index might have removed the sample $\mathsf{L}_{\mathsf{SA}}[p] = \mathsf{SA}[ep_i]$ during its subsampling process, which is indicated by the flag $\mathtt{Del}[p] = 1$. When this happens, we use an iterative search, computing $j_k = \mathsf{LF}^k(j)$ for $k = 1, 2, \ldots$ and $j = \mathsf{LF}^{-1}(ep_i)$, until a remaining sampled $\mathsf{SA}[j_k]$ is found. This is identified because j_k is the last position in a BWT-run (i.e., $j_k = n$ or $\mathtt{Start}[j_k{+}1] = 1$) and $\mathtt{Del}[q] = 0$ for $q = \mathsf{rank}_1(\mathtt{Start}, j_k)$. When we find such j_k, we can compute the final $\mathsf{SA}[ep_i]$ by adjusting the sample found, $q' = \mathsf{rank}_0(\mathtt{Del}, q)$, based on the k steps in the search, obtaining $\mathsf{SA}[ep_i] = \mathsf{L}_{\mathsf{SA}}[q'] + k$.

The number of LF-steps in each backward search iteration of the *sr*-index is bounded by the sampling factor s: the next lemma shows that for some $k < s$ we will find a non-removed sample.

Lemma 4.2. *If there is a removed sample t'_j such that t_i and t_{i+1} are remaining samples satisfying $t_i < t'_j < t_{i+1}$, then $t_{i+1} - t_i \leq s$.*

Proof. Since our subsampling process removes samples from 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 t'_j was removed because $t_{i+1} - t_i \leq s$. Therefore, the lemma holds in this case.

Otherwise, there were other samples to the right of t'_j, say $t'_{j+1}, t'_{j+2}, \ldots, t'_{j+k}$, which were consecutively removed until the current sample t_{i+1} was reached. First, 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 $t'_j, t'_{j+1}, \ldots, t'_{j+l-1}$) as $t'_{j+l+1} - t_i \leq s$. Finally, we removed t'_{j+k} since $t_{i+1} - t_i \leq s$, thus that the lemma holds in this case as well. □

This implies a fixed bound on the search, beginning from the position j of the removed sample $t' = \mathsf{SA}[j] - 1$ and extending to the surrounding remaining samples $t_i < t' < t_{i+1}$. It is sufficient to perform $k = t' - t_i < s$ LF-steps until position $j_k = \mathsf{LF}^k(j)$ satisfies $\mathsf{SA}[j_k] - 1 = t_i$, which is stored in $\mathsf{L}_{\mathsf{SA}}[q']$ and not removed.

If we followed verbatim the modified backward search of the *r*-index, finding every $\mathsf{SA}[ep_i]$, we would perform $O(m \cdot s)$ steps on the *sr*-index. We now reduce this to $O(m+s)$ steps by noting that

Algorithm 4: Counting pattern occurrences and finding value SA[ep] with the *sr*-index.

Input : Query pattern $P[1..m]$.
Output: Range $\langle sp, ep\rangle$ on SA for P; Value SA[ep].

```
1  function count(P[1..m])
2      ⟨sp, ep⟩ ← ⟨1, n⟩
3      ⟨i_v, p_v⟩ ← ⟨−1, −1⟩
4      for i ← m downto 1 do
5          c ← P[i]
6          p ← rank_1(Start, ep)
7          if c ≠ Letter[p] then
8              ⟨i_v, p_v⟩ ← ⟨i, p⟩
9          sp ← C[c] + rank_c(BWT, sp−1)+1
10         ep ← C[c] + rank_c(BWT, ep)
11         if sp > ep then
12             return "P is not in T"
13     v ← findToehold(i_v, p_v)
14     return ⟨sp, ep, v⟩
15 function findToehold(i_v, p_v)
16     if i_v = −1 then
           // SA[n] is stored
17         return SA[n] − m
18     c ← P[i_v]
19     q ← select_c(Letter, rank_c(Letter, p_v))
20     j ← select_1(Start, q + 1) − 1
21     k ← 0
22     while (j < n and Start[j + 1] = 0) or
        Del[q] = 1 do
23         j ← LF(j)
24         q ← rank_1(Start, j)
25         k ← k + 1
26     return L_SA[rank_0(Del, q)]+k−(i_v−1)
```

the only value we require is SA[ep] = SA[ep_1]. Further, we need to know SA[ep_{i+1}] to compute SA[ep_i] only in the easy case where BWT[ep_{i+1}] = $P[i]$ and so SA[ep_i] = SA[ep_{i+1}] − 1. Otherwise, the value SA[ep_i] is computed afresh.

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

Algorithm 4 gives the complete pseudocode that counts while finding a toehold. Note that, if P does not occur in $\mathcal{T}$ (i.e., $occ = 0$) we realize this after the $O(m)$ backward steps because some $sp_i > ep_i$, and thus we do not spend the $O(s)$ extra steps.

Running on the sr-csa. Although it is possible to simulate the LF-steps via the Ψ function, a more intuitive and efficient approach is to search the text forwards for a sampled position, using Ψ directly. Consequently, starting from the eliminated sample t', the *sr*-csa searches for the next remaining sample, t_{i+1}, rather than for the preceding one, t_i. Lemma 4.2 also shows that the subsampling factor s bounds the number of steps in this case. Therefore, for some $k < s$, sample

$t_{i+1} = \mathsf{SA}[\Psi^k(j)]$ is a Ψ-run head stored in F_{SA}, with $t' = \mathsf{SA}[j]$ (note that we use F_{SA} instead of L_{SA} because on the *sr*-csa we obtain $\mathsf{SA}[sp]$, not $\mathsf{SA}[ep]$).

4.3 Locating from Toehold

We now focus on the problem of finding $\mathsf{SA}[j-1]$ from $i = \mathsf{SA}[j] - 1$. If we just apply the *r*-index procedure based on Φ, we will end up at an incorrect predecessor if the correct one has been removed during subsampling. To circumvent this problem, the *sr*-index will start with a procedure similar to the one used to identify the toehold, that is, iteratively searching for a remaining sample. We will show that, when this procedure fails, we can safely use the Φ function like the *r*-index.

In this context, we first calculate $j'_k = \mathsf{LF}^k(j-1)$ for $k = 0, \ldots, s-1$. For each of those j'_k we verify whether it 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 has not been removed (i.e., $\mathtt{Del}[q] = 0$, with $q = \mathsf{rank}_1(\mathtt{Start}, j'_k)$). If these conditions are met, then it can be immediately derived that $\mathsf{SA}[j'_k] = \mathsf{L}_{\mathsf{SA}}[q'] + 1$, where $q' = \mathsf{rank}_0(\mathtt{Del}, q)$. Consequently, we can also obtain $\mathsf{SA}[j-1] = \mathsf{SA}[j'_k] + k$.

Unlike in Section 4.2, the symbol $\mathsf{BWT}[j-1]$ is not necessarily an end of run. Therefore, there is no guarantee that a solution will be found for some $0 \le k < s$. However, the following property shows that, if there were some ends of runs j'_k, it is not possible that all were removed from L_{SA}.

Lemma 4.3. *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.*

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 4.2, no samples were removed between t_i and t_{i+1}. □

This means that, if the above process fails to yield an answer, it is possible to employ Eq. (4) directly, as proved next.

Lemma 4.4. *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* $\mathsf{F}_{\mathcal{T}}$ *between positions* $i = \mathsf{SA}[j]-1$ *and* $\mathsf{pred}(\mathsf{F}_{\mathcal{T}}, i)$.

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 4.3 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 $\mathsf{F}_{\mathcal{T}}$ between positions $\mathsf{SA}[j]-1$ and $\mathsf{SA}[j]-k$ (both included), and there exists one in $\mathsf{SA}[j]-1-k$. Indeed, $\mathsf{pred}(\mathsf{F}_{\mathcal{T}}, i) = \mathsf{SA}[j]-1-k$. □

An additional optimization, which does not alter the worst-case complexity but enhances performance in practice, is to reuse work across successive occurrences. Let $\mathsf{BWT}[sm..em]$ be a maximal run inside $\mathsf{BWT}[sp..ep]$. For every $sm \le j \le em$, the first LF-step will result in

Algorithm 5: Locating pattern occurrences with the *sr*-index.

Input : Query pattern $P[1..m]$.
Output: Occurrences of P: $V[1..occ] = \mathsf{SA}[sp..ep]$.

```
1  function locate(P[1..m])
2      ⟨sp, ep, v⟩ ← count(P)
3      V[ep − sp + 1] ← v
4      if sp < ep then
5          locateRec (sp, ep − 1, 0)
6      return V
7  function checkSample(em, k)
8      p ← rank_1(Start, em)
9      if Del[p] = 1 then
10         return em
11     V[em − sp + 1] ←
12             L_SA[rank_0(Del, p)] + 1 + k
13     return em − 1
14 function locateRec(sm, em, k)
15     if k = s then
16         for j ← em downto sm do
17             V[j − sp + 1] ← Φ(V[j − sp + 2])
18         return
19     p ← rank_1(Start, em)
20     q ← rank_1(Start, sm)
21     for i ← p downto q do
22         if i < p or Start[em + 1] = 1 then
23             em ←checkSample (em, k)
24         im ← max(select_1(Start, i), sm)
25         if im > em then
26             continue
27         locateRec (LF(im), LF(em), k + 1)
28         em ← im − 1
```

$\mathsf{LF}(j) = \mathsf{LF}(sm) + (j - sm)$. Therefore, the entire sequence of values $\mathsf{LF}(j)$ can be computed through a single iteration of the LF function.

Consequently, rather than locating $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$ one by one, we first report $\mathsf{SA}[ep]$ (which has been previously identified), and then partition $\mathsf{BWT}[sp..ep-1]$ into maximal runs using bitvector Start. We will traverse those maximal runs $\mathsf{BWT}[sm..em]$, from largest to smallest sm, with the invariant that $\mathsf{SA}[em+1]$ is known. We first check that the run end $\mathsf{BWT}[em]$ is sampled, in which case we report its position and decrement em (note that the offset k must be added to all the results reported at level k of the recursion). We then continue recursively with $\mathsf{SA}[\mathsf{LF}(sm)..\mathsf{LF}(sm) + (em - sm)]$. By Lemma 4.2, every non-removed sample found during the traversal has been duly reported prior to level $k = s$. Upon reaching the final recursion level ($k = s$), we use Eq. (4) to obtain $\mathsf{SA}[em], \ldots, \mathsf{SA}[sm]$ consecutively from $\mathsf{SA}[em+1]$. Algorithm 5 gives the complete procedure to locate the occurrences.

Running on the sr-csa. The aforementioned results are directly applicable to the *r*-csa to obtain the *sr*-csa. On this structure, the problem is to compute the value $\mathsf{SA}[j+1]$ based on the previously determined value $i = \mathsf{SA}[j]$. It is a straightforward exercise to prove the symmetric version of

Lemmas 4.3 and 4.4 needed for the *sr*-csa. The *sr*-csa probes the range $\mathsf{SA}[j+1], \ldots, \mathsf{SA}[j+1]+s-1$ using $\Psi^k(j+1)$ for $0 \le k < s$, looking for a non-removed sample. If this fails, it makes use of the Φ^{-1} function of the *r*-csa. Finally, a similar recursive process can be employed to avoid computing $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$ individually. In this case, the procedure uses the maximal runs within $\Psi[sp..ep]$.

4.4 The basic *sr*-indexes, *sr*-index$_0$ and *sr*-csa$_0$

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

PROOF. The space is the sum of the counting structures of the *r*-index and our modified locating structures, according to Lemma 4.1. The space of bitvector `Del` is $O(r)$ bits, which is accounted for in the formula.

As for the time, we have seen that the modified backward search requires $O(m)$ steps if $occ = 0$ and $O(m+s)$ otherwise (Section 4.2). Each occurrence is then located in $O(s)$ steps (Section 4.3). In total, we complete the search with $O(m + s \cdot occ)$ steps.

Each step involves $O((1/\epsilon)\log(\sigma + n/r))$ time in the basic *r*-index implementation, including Eq. (4). Our index includes additional `ranks` on `Start` and other constant-time operations, which are all in $O(\log(n/r))$. Since $\mathsf{F}_{\mathcal{T}}$ now has $O(\min(r, n/s))$ 1s, however, operation rank_1 on it takes time $O(\log(n/\min(r, n/s))) = O(\log\max(n/r, s)) = O(\log(n/r + s))$. Yet, this `rank` is computed only once per occurrence reported, when using Eq. (4), so the total time per occurrence is still $O(\log(n/r+s) + s \cdot \log(\sigma + n/r)) = O(s \cdot \log(\sigma + n/r))$. □

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 [16], but our focus is on low practical space usage.

A similar analysis, combined with Theorem 3.7, yields the analogous result for the *sr*-csa.

THEOREM 4.6. *The sr-csa$_0$ uses*

$$r \cdot \big((2\log(n/r) + \log\sigma)(1+\epsilon) + O((\log n)/B) + O(1)\big) + \min(r, 2\lceil n/(s+1)\rceil) \cdot 2\log n$$

bits of space, for any constant $\epsilon, B > 0$*, and finds all the occ occurrences of* $P[1..m]$ *in* $\mathcal{T}$ *in time* $O(m(\log r + B) + occ(s(\log r + B) + \log n))$.

Proof. We carry out $O(m + s \cdot occ)$ steps, each of which computes Ψ at cost $O(\log r + B)$. This yields total time $O((m + s \cdot occ)(\log r + B) + occ \log(n/r + s))$, which is simplified to the one given once we put together all the terms that are multiplied by occ. □

It should be noted that Theorems 4.5 and 4.6 can be obtained by simply choosing the smallest between the r-index and the rlfm-index, or r-csa and rlcsa, respectively. In practice, however, the sr-indexes perform significantly 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 Faster and larger *sr*-indexes, *sr*-index$_1$ and *sr*-csa$_1$

The sr-index$_0$ and the sr-csa$_0$ guarantee locating time proportional to s. To do this, however, they perform up to s LF-steps or Ψ-steps to locate *every* occurrence, even when this turns out to be useless. The sr-index$_1$ variant adds a new small component to speed up some cases:

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

With this bitvector, if we have $i = \texttt{SA}[j] - 1$ and $\texttt{Valid}_\texttt{F}[\texttt{rank}_1(\texttt{F}_\mathcal{T}, i)] = 1$, we know that there were no removed values between i and $\texttt{pred}(\texttt{F}_\mathcal{T}, i)$ (even if they are less than s positions apart). In this case we can skip the computation of $\texttt{LF}^k(j-1)$ of sr-index$_0$, and directly use Eq. (4). 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 line 27 of Algorithm 5.

The sr-csa$_1$ index employs the analogous bitvector $\texttt{Valid}_\texttt{L}$ to ascertain whether values between i and $\texttt{succ}(\texttt{L}_\mathcal{T}, i)$ have been removed. In this instance, $\texttt{Valid}_\texttt{L}[q] = 0$ indicates that one or more elements were removed between the $(q-1)$th and qth 1s of $\texttt{L}_\mathcal{T}$.

The space and worst-case complexities of Theorems 4.5 and 4.6 are preserved in sr-index$_1$ and sr-csa$_1$.

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

Our second variants, sr-index$_2$ and sr-csa$_2$, add a second and significantly larger structure:

$\texttt{ValidArea}_\texttt{F}$: an array whose cells are associated with the 0s in $\texttt{Valid}_\texttt{F}$. If $\texttt{Valid}_\texttt{F}[q] = 0$, then $d = \texttt{ValidArea}_\texttt{F}[\texttt{rank}_0(\texttt{Valid}, q)]$ is the distance from the qth 1 in $\texttt{F}_\mathcal{T}$ to the next removed value. Each entry in $\texttt{ValidArea}_\texttt{F}$ requires $\log n$ bits. [5]

[5]The subsampling process guarantees that for each BWT-run tail that is removed, there is a preceding tail that remains at a distance no greater than s in the text. It should be noted that this condition is not guaranteed for the heads of the BWT-runs.

If $\mathtt{Valid_F}[\mathtt{rank}_1(\mathtt{F}_{\mathcal{T}}, i)] = 0$, then there was a removed sample at $\mathtt{pred}(\mathtt{F}_{\mathcal{T}}, i) + d$, but not before. So, if $i < \mathtt{pred}(\mathtt{F}_{\mathcal{T}}, i)+d$, we can still use Eq. (4); 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 4.5 (and 4.6) and the space increases by at most $r \log n$ bits.

The *sr*-csa$_2$ is based on the same fundamental concept but employs the array $\mathtt{ValidArea_L}$. Let $\mathtt{Valid_L}[q] = 0$ and $\mathtt{ValidArea_L}[\mathtt{rank}_0(\mathtt{Valid_L}, q)] = d$, for any text position i such that $q = \mathtt{rank}_1(\mathtt{L}_{\mathcal{T}}, \mathtt{succ}(\mathtt{L}_{\mathcal{T}}, i))$: If $i > \mathtt{succ}(\mathtt{L}_{\mathcal{T}}, i) - d$, the Φ^{-1} function (Eq. 3) can be employed; otherwise, up to s Ψ-steps must be executed from the corresponding SA position $j + 1$.

5 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed indexing schemes: *r*-csa, *sr*-csa, and *sr*-index. These algorithms are implemented using C++17 and the SDSL library[6]. The code is publicly available on GitHub (https://github.com/duscob/sr-index).

We compare our proposed solutions against existing implementations of the *r*-index, rlcsa, and other popular indexing methods designed for repetitive text collections. We aim to assess the effectiveness of the *sr*-csa and *sr*-index approaches, particularly in terms of space usage and search speed, across various datasets.

The tests were conducted on a computer with two Intel Xeon processors (Silver 4110 at 2.10 GHz) and 736 GB of RAM. The operating system was Debian Linux (version `5.10.0-0.deb10.16-amd64`). We compiled the code with the highest optimization settings and disabled multithreading for consistency.

To ensure reliable results, we measured the average user time needed to perform several searches for random patterns of lengths 10, 20, and 30 characters in different text collections. We report space usage in bits per symbol (bps) and search times in microseconds per occurrence (μs/occ). We note that some indexing methods might not be suitable for all text collections or might require excessive space or time to build. Those methods are excluded from the corresponding graphs in the section of results.

5.1 Tested indexes

We include the following indexes in our benchmark; their space decrease as s grows. We chose the parameter range so as to cover the interesting space-time tradeoffs. In particular, larger values of s for the *sr*-indexes only increase the time without significantly reducing the space any further.

***r*-csa:** Our index implementation, using block size $B = 64$ (also for *sr*-csa).

[6]Available at https://github.com/simongog/sdsl-lite.

***sr*-csa:** Our index, including the three variants, with sampling values $s = 4, 8, 16, 32, 64$.

***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.[7]

rlcsa: An implementation of the run-length CSA [33], which outperforms the actual rlfm-index implementation.[8] We use text sampling values $s = n/r \cdot f/8$, with $f = 8, 10, 12, 14, 16$.

csa: An implementation of the CSA [45], which outperforms in practice the fm-index [12, 13]. 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 [7].[9] We use Patricia tree sampling values $s = 4, 16, 64$.

lz-index and lze-index: Two variants of the Lempel–Ziv based index [26].[10]

hyb-index: A hybrid between a Lempel–Ziv and a BWT-based index [11].[11] We build it with parameters $M = 8, 16$, which are the best for this case.

5.2 Collections

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

Pizza&Chili: 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).[12]

Synthetic DNA: A 100KB DNA text from Pizza&Chili, 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.

Real DNA: Some real DNA collections to study the performance on more massive data:

- **Chr19:** Human assembled genome collections of about 55 billion base pairs, concretely the set of 1,000 chromosome 19 genomes taken from the 1000 Genomes Project [47].
- **Salmonella:** Bacterial assembled genome collections of about 70 billion base pairs, concretely the set of 14,609 genomes from the GenomeTrakr project [46].

For each dataset, we randomly selected 500 patterns of three different sizes (10, 20, and 30 characters), mixed in a single set. To ensure the stability and reliability of our results, we identified

[7] From https://github.com/nicolaprezza/r-index.
[8] From https://github.com/adamnovak/rlcsa.
[9] From https://github.com/apachecom/grammar_improved_index.
[10] From https://github.com/migumar2/uiHRDC.
[11] From https://github.com/hferrada/HydridSelfIndex.
[12] From http://pizzachili.dcc.uchile.cl/repcorpus/real.

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	1,611.2	Chr19	56,386.1	1,287.4
kernel	238.0	92.4	Salmonella	70,242.6	47.0
worldleaders	44.7	81.9			
coreutils	195.8	43.8			

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

and removed outliers from the collected patterns using the interquartile range (IQR) method. This process resulted in a final set of 404 to 465 patterns per dataset.

5.3 Results

To simplify our analysis and later comparisons between the indexing methods, we first investigated the effectiveness of different variants within our proposed *sr*-index and *sr*-csa methods.

Our initial experiments focus on synthetic DNA datasets; the results are shown in Figure 6. The largest variant stands out as the best choice for both *sr*-index and *sr*-csa in terms of balancing space efficiency and search speed. When the sampling factor s is small, all three variants within each method exhibit similar performance. However, as we increase the parameter s, the number of samples decreases at a regular rate, but a significant difference in locating speed emerges: *sr*-index$_2$ and *sr*-csa$_2$ demonstrate a substantial improvement in search time compared to the other two, while the required space remains comparable across all variants.

These findings show that the extra data we associate with samples in the third variant has a minor impact in space. Yet, this additional sample validity information plays a crucial role in how fast the index can locate pattern occurrences. Given these results, we will simply refer to the third variants as *sr*-index and *sr*-csa, and use them for the following comparisons.

We also note that, while the *r*-index and the *r*-csa are almost indistinguishable, there is some difference in the subsampled variants: the *sr*-index performs better with higher repetitiveness and the *sr*-csa stands out with lower repetitiveness, meeting at a mutation probability of 0.003.

Figure 7 includes the other state-of-the-art solutions in the comparison. It can be seen that the *r*-index and our *r*-csa are significantly faster than the others on highly repetitive scenarios. They are only matched by the fm-index and the rlcsa, which are not designed for repetitive texts, when the mutation probability reaches 0.01 (and the average run length n/r approaches 25).

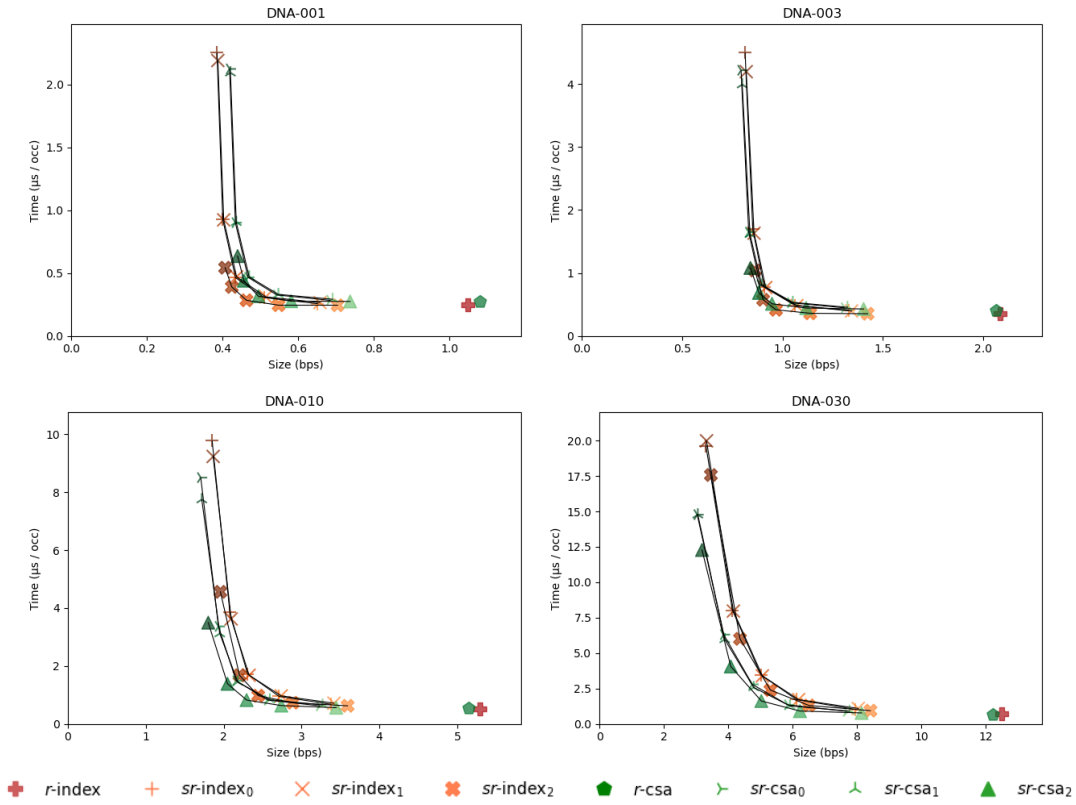

Fig. 6. Space-time tradeoffs of *sr*-index and *sr*-csa variants on the synthetic DNA collections.

The dominant indexes, however, are our main contributions: the *sr*-index and the *sr*-csa. The figures show that they can be almost as fast as the *r*-index and *r*-csa, while using less than half their space. They sharply dominate the space-time tradeoff map, even with mutation probability as high as 0.01, sweeping out all previous solutions based on the BWT, on Ψ, on grammars, and on Lempel-Ziv. The only alternative that stays in the Pareto curve is the hyb-index, which in the most repetitive texts can use up to half the space, yet at the price of being an order of magnitude slower. Only when the mutation rate reaches 0.03 and the average run length approaches 10, our *sr*-indexes finally yield to the csa.

Therefore, as promised, we are able to remove a significant degree of redundancy in the *r*-indexes without sacrificing their outstanding time performance.

Figure 8 shows the performance on the real-life genome collections of Pizza&Chili. The situation is now more varied, for example the *sr*-indexes now retain the performance of their corresponding *r*-indexes while using 1.5–4.0 times their space. The *sr*-index outperforms the *sr*-csa in some

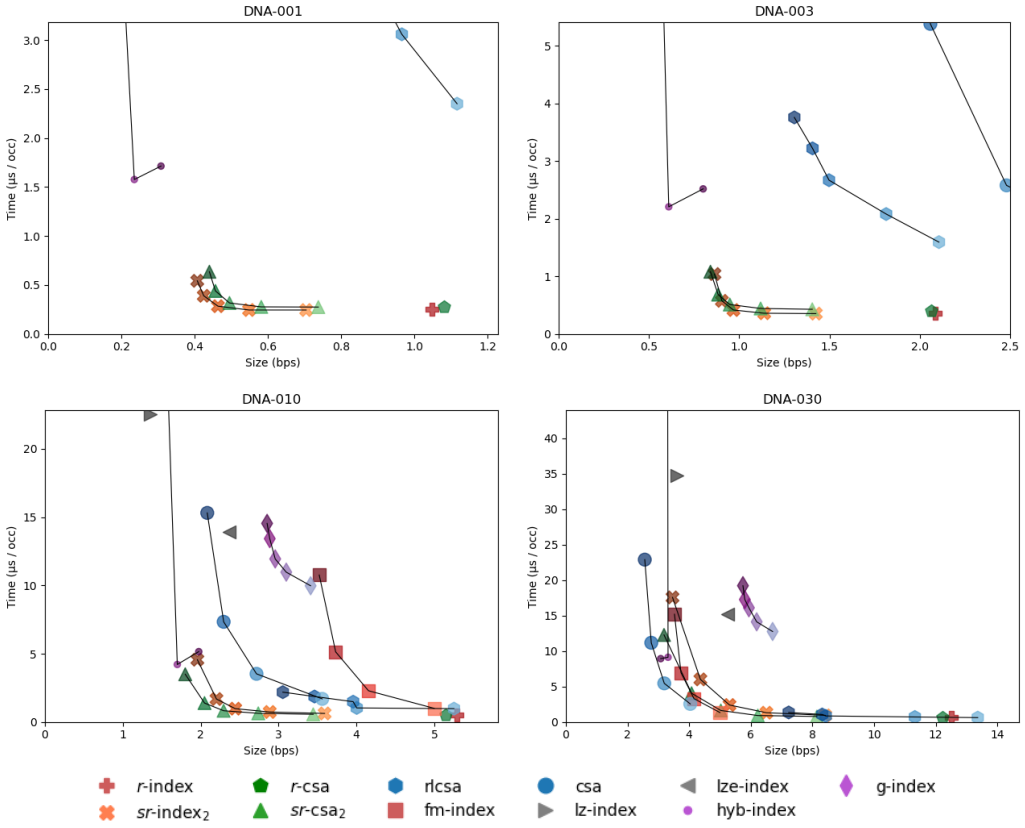

Fig. 7. Space-time tradeoffs on the synthetic DNA collections.

texts and loses to it on others, independently of the average run length of the collections. Most interestingly, our *sr*-indexes perform even *better* than their corresponding *r*-indexes and *rl*-indexes in those real texts, arguably because they exploit better the non-uniform distributions of the samples in the text. For example, none of our preceding indexes based on the BWT or Ψ matches our *sr*-indexes even on Escherichia, where the average run length is around 7.5, while the csa outperformed our subsampled indexes on the synthetic DNA collections with mutation probability 0.03 and average run length around 11. The hyb-index still belongs to the Pareto curve and also seems to benefit from non-uniformity: it now clearly outperforms our *sr*-indexes on Escherichia, the text with the least repetitiveness.

Figure 9 illustrates this phenomenon. It shows the distribution of run heads in text order on some synthetic and real texts, and how many samples survive for increasing values of s. Note how the distribution of samples on the synthetic texts (even if coming from random mutations over an actual DNA text) are uniform and very different from the distributions on real texts. Note also how,

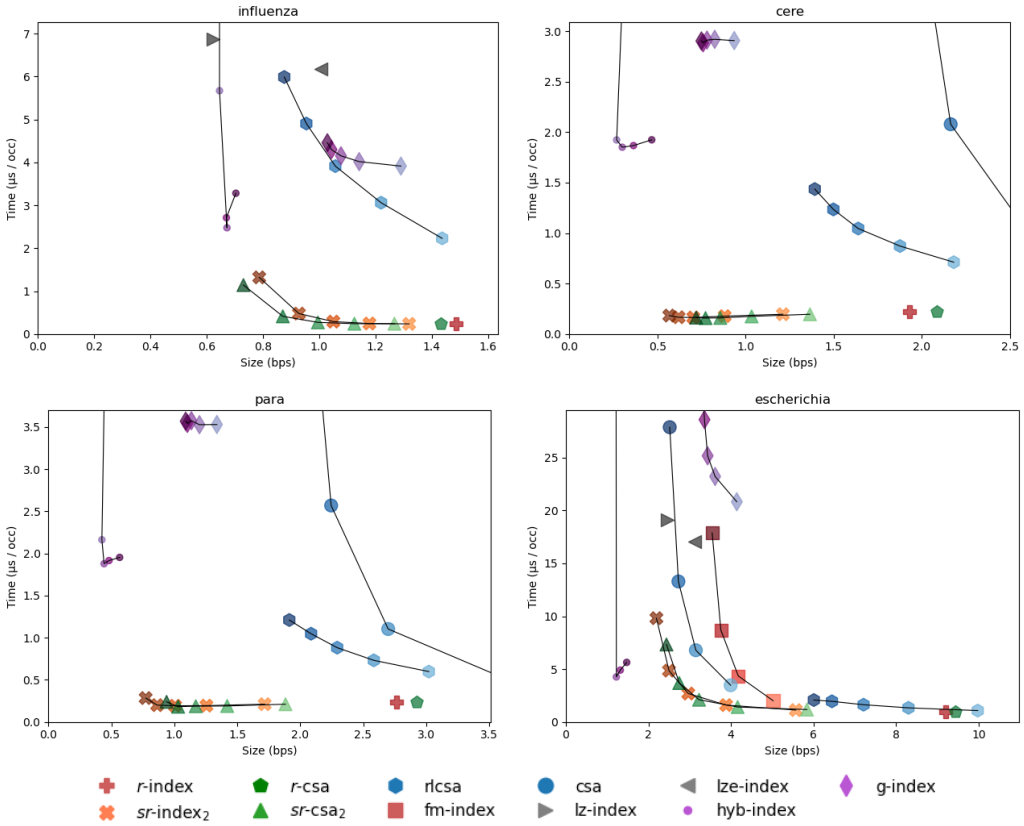

Fig. 8. Space-time tradeoffs on the genome Pizza&Chili collections.

as s grows, the samples decrease much faster on the denser areas and make the distributions tend to uniform (see the darkest areas on the real texts). For example, for $s = 4$, the number of samples roughly reduce to $1/4$ on the synthetic texts, while they reduce to about a $1/7$ in the densest areas of the real texts. Our worst-case analysis better reflects the uniform case, but the subsampling is much more effective on the non-uniform histograms.

Figure 10 shows the case of other repetitive collections in Pizza&Chili. In these collections with larger alphabets, the *sr*-csa generally outperforms the *sr*-index, as the latter has an $O(\log \sigma)$ time penalty its operations. Otherwise, the conclusions do not differ from those obtained on DNA.

Overall, we conclude that our *sr*-indexes are sharply dominant when the average run length n/r exceeds 10. At this point, depending on the type of text, they may be matched by other indexes, particularly the csa and the hyb-index. Texts with those average run lengths are arguably non-repetitive anymore: even the classic indexes exceed the 2 bits per symbol used by a plain representation of the data! Finally, our *sr*-indexes perform better on real than on synthetic data.

Fig. 9. Distribution of original and subsampled BWT-run heads within the synthetic DNA datasets and Pizza&Chili repetitive texts. The x-axis represents the positions of these run heads along the text. The y-axis indicates how frequently BWT-run heads appear at different locations. The original distributions are shown with the lightest color, and darker colors are used for subsampling with larger values of s.

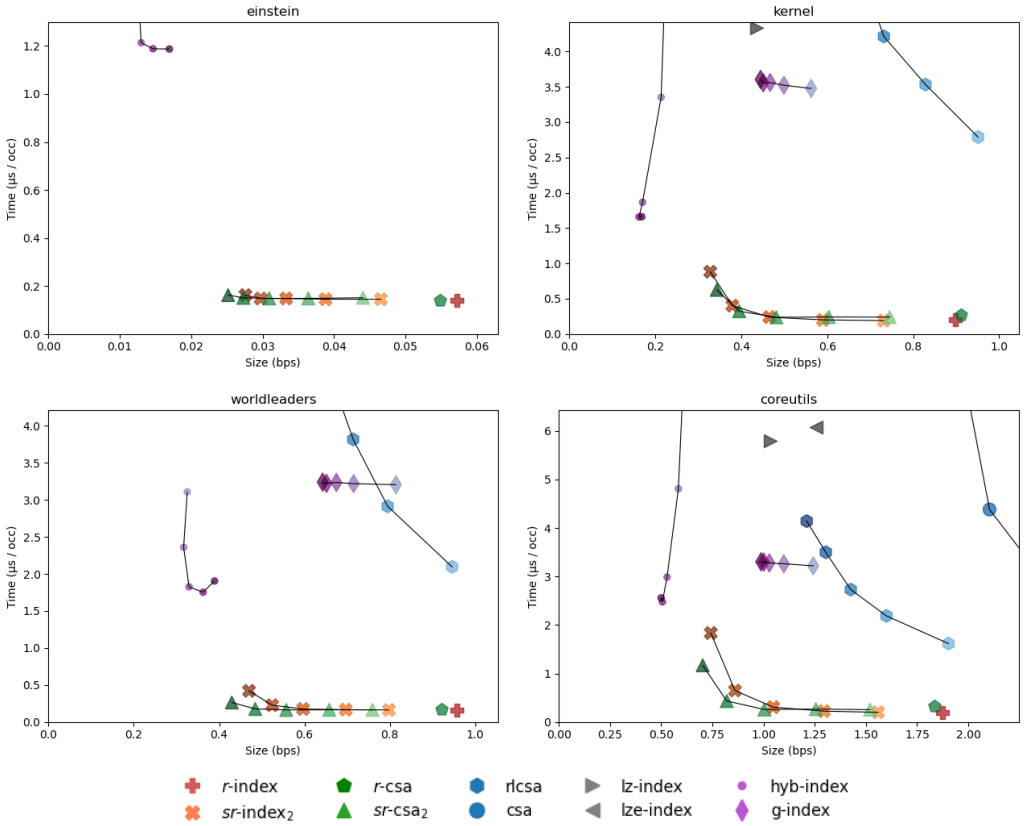

Fig. 10. Space-time tradeoffs on the document Pizza&Chili collections.

In general, the bits per symbol used by the *sr*-indexes 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 and *r*-csa use $70r$–$90r$ bits.

Finally, Figure 11 shows the results on the largest collections, Chr19 (56 GB) and Salmonella (70 GB). We were only able to construct the BWT/Ψ-related indices (except rlcsa) for both, hyb-index for Chr19, and lz-index and lze-index for Salmonella. Our benchmarks show that the same observed trends scale to gigabyte-sized collections with different repetitiveness levels. Specifically, for Chr19, the *sr*-index shows the best performance. With a sampling factor of $s = 64$, it reduces the space required by the *r*-index from 0.076 to 0.032 bps, and even reduces the search time from 0.69 to 0.58 μs. The hyb-index requires 57% of the *r*-index space, but it is 4.4 times slower. In the case of Salmonella, the *sr*-csa excels in the time-space tradeoff. It reduces the *r*-csa space by 4.2 times (from 1.89 to 0.45 bps), while the query time increases only by 4.7%, from 1.71 to 1.79 μs.

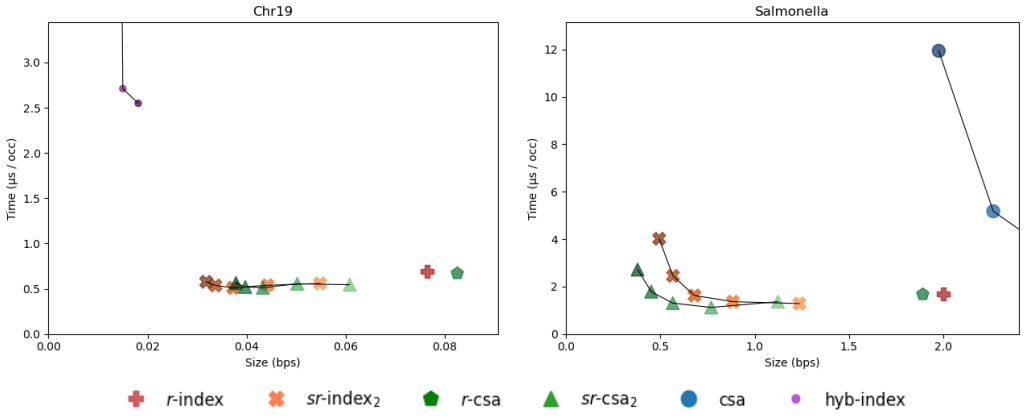

Fig. 11. Space-time tradeoffs on the real DNA collections.

6 CONCLUSIONS

We have introduced the *sr*-index, an *r*-index variant that solves the problem of its relatively bloated space while retaining its search speed. We have also designed the *r*-csa, an equivalent to the *r*-index that builds on the rlcsa instead of on the rlfm-index, and also its reduced-space version, the *sr*-csa. The *sr*-index and *sr*-csa match the time performance of their large-space versions, *r*-index and *r*-csa, while using 1.5–4.0 times less space. Further, they sweep the table of compressed indexes for highly repetitive text collections: they are orders of magnitude faster than the others while sharply outperforming most of them in space as well.

Unlike the *r*-index and the *r*-csa, the *sr*-index and the *sr*-csa use little space even in scenarios of mild repetitiveness, which makes them usable in a wider range of bioinformatic applications. For example, the *sr*-index uses 0.03 bits per symbol (bps) while reporting each occurrence within half a microsecond on a very repetitive gigabyte-sized collection of human genomes, where the original *r*-index and *r*-csa use around 0.08 bps and take the same time. On a similarly sized but less repetitive collection of Salmonella genomes, the *sr*-csa takes 0.45 bps to answer queries in less than 2 microseconds, matching the time of the *r*-index and *r*-csa, which take about 2 bps.

On synthetic texts with random mutations, the *sr*-index and the *sr*-csa outperform classic compressed indexes on collections with mutation rates under as much as 1%. Classic indexes only match them when the mutation rate reaches 3%. At this rate the text is arguably no longer repetitive, since no index can reach the 2 bps required to store the data in plain form. Further, our indexes perform much better on *real* texts than on synthetic ones, as they precisely exploit the uneven coverage of text samples that arise with the *r*-index and *r*-csa.

The conference version of this article had already an impact on Bioinformatic research. Goga et al. [17] simplified and reduced the *sr*-index subsample by abandoning Φ and Φ^{-1}, and used that subsample in a version of Rossi et al.'s [43] tool MONI. Goga et al.'s version finds maximal exact matches (MEMs) between a pattern and an indexed text using longest common prefix (LCP) queries between suffixes of the pattern and suffixes starting immediately after characters at boundaries of runs in the BWT of the text. They observed that in their experiment with 1000 human chromosome 19s, "with s = 5 the index took less than three quarters as much space as without subsampling and used only 6% more query time".

The *sr*-index subsampling may be useful even when we have no interest in the suffix array, as in Depuydt et al.'s [9] index for metagenomic classification. They use Li's [28] forward-backward algorithm (see a recent discussion [29]) to find the MEMs between a DNA long read and a large collection of genomes from several different species. Forward-backward returns the BWT intervals for the MEMs (without using the suffix array), and Depuydt et al. check the corresponding intervals in a run-length compressed tag array [1] indicating the species of the genome containing each character in the BWT. If there are enough sufficiently long MEMs in a read and they all occur only in genomes of the same species, Depuydt et al. guess the read comes from an individual of that species.

When only a few distinct species are represented in the dataset, the number of runs in the tag array may be even smaller than the number of runs in the BWT. When there are many related species, however, the BWT tends to be much more run-length compressible. In such cases, Depuydt et al. store a bitvector marking the boundaries between runs in the tag array, so they can check that all the occurrences of a MEM are in genomes of one species (without finding out which species that is). They also store as a "toehold" the tag for the first character in each run in the BWT. When all the occurrences of a MEM are in genomes of one species, the last toehold tag they found while computing that MEM, is that species. Those toehold tags take the place of suffix-array entries, and they can be *sr*-index subsampled as well.

Another relevant line of future work is to support direct access to arbitrary entries of the suffix array and its inverse, for example to implement compressed suffix trees [16] The rlfm-index and rlcsa [33], which for pattern searching are dominated by the *sr*-index and the *sr*-csa, use their regular text sampling to compute any such entry in time proportional to the sampling step s. Obtaining an analogous result on the *sr*-index or the *r*-csa would lead to practical compressed suffix trees for highly repetitive text collections, which to date hardly reach the barrier of 2 bps on real bioinformatic collections [3, 5, 37]. Other proposals for accessing the suffix array faster than the rlfm-index [19, 42] illustrate this difficulty: they require even more space than the *r*-index.

ACKNOWLEDGEMENTS

Funded in part by Basal Funds FB0001 and AFB240001, ANID, Chile. D.C. also funded by ANID/Scholarship Program/DOCTORADO BECAS CHILE/2020-21200906, Chile. T.G. funded by NSERC Discovery Grant RGPIN-07185-2020. G.N. also funded by Fondecyt Grant 1-230755, ANID, Chile.

REFERENCES

[1] Andrej Baláž, Travis Gagie, Adrián Goga, Simon Heumos, Gonzalo Navarro, Alessia Petescia, and Jouni Sirén. 2024. Wheeler maps. In *Proc. Latin American Symposium on Theoretical Informatics (LATIN)*. 178–192.

[2] Djamal Belazzougui and Gonzalo Navarro. 2015. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms* 11, 4, Article 31 (2015).

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

[4] Michael Burrows and David J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report 124. Digital Equipment Corporation.

[5] Manuel Cáceres and Gonzalo Navarro. 2022. Faster repetition-aware compressed suffix trees based on Block Trees. *Information and Computation* 285B (2022), article 104749.

[6] David R. Clark. 1996. *Compact PAT Trees.* Ph. D. Dissertation. University of Waterloo, Canada.

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

[8] Dustin Cobas, Travis Gagie, and Gonzalo Navarro. 2021. A fast and small subsampled r-index. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*. Article 13.

[9] Lore Depuydt, Omar Ahmed, Jan Fostier, Travis Gagie, and Ben Langmead. In preparation. Metagenomic classification via MEM-finding and tagging.

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

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

[12] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *Journal of the ACM* 52, 4 (2005), 552–581.

[13] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms* 3, 2 (2007).

[14] Paolo Ferragina and Gonzalo Navarro. Accessed Sept 2024. The Pizza&Chili Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

[15] Travis Gagie and Gonzalo Navarro. 2019. Compressed indexes for repetitive textual datasets. In *Encyclopedia of Big Data Technologies*. Springer.

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

[17] Adrián Goga, Lore Depuydt, Nathaniel K. Brown, Jan Fostier, Travis Gagie, and Gonzalo Navarro. 2024. Faster maximal exact matches with lazy LCP evaluation. In *Proc. 34th Data Compression Conference (DCC)*. 123–132.

[18] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. 2006. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*. 368–373.

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

[20] Robrtyo Grossi, Ankur Gupta, and Jeffrey S. Vitter. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

[21] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing* 35, 2 (2005), 378–407.

[22] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. 2009. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

[23] Juha Kärkkäinen and Simon J. Puglisi. 2011. Fixed block compression boosting in FM-Indexes. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*. 174–184.

[24] Dominik Kempa and Tomasz Kociumaka. 2020. Resolution of the Burrows–Wheeler Transform conjecture. In *Proc. 61st IEEE Symposium on Foundations of Computer Science (FOCS)*. 1002–1013.

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

[26] Sebastian Kreft and Gonzalo Navarro. 2013. On compressing and indexing repetitive sequences. *Theoretical Computer Science* 483 (2013), 115–133.

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

[28] Heng Li. 2012. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics* 28, 14 (2012), 1838–1844.

[29] Heng Li. 2024. BWT construction and search at the terabase scale. *CoRR* 2409.00613 (2024).

[30] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-Scale Algorithm Design*. Cambridge University Press.

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

[32] Veli Mäkinen and Gonzalo Navarro. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms* 4, 3 (2008), article 32.

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

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

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

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

[37] Gonzalo Navarro and Alberto Ordóñez. 2016. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics* 21, 1 (2016), article 1.8.

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

[39] Takaaki Nishimoto and Yasuo Tabei. 2020. Faster queries on BWT-runs compressed indexes. *CoRR* 2006.05104 (2020).

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

[41] Nicola Prezza. 2017. *Compressed Computation for Text Indexing*. Ph. D. Dissertation. University of Udine, Italy.

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

[43] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. 2022. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology* 29, 2 (2022), 169–187.

[44] Kunihiko Sadakane. 2002. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 225–232.

[45] Kunihiko Sadakane. 2003. New text indexing functionalities of the Compressed Suffix Arrays. *Journal of Algorithms* 48, 2 (2003), 294–313.

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

[47] The 1000 Genomes Project Consortium. 2015. A global reference for human genetic variation. *Nature* 526 (2015), 68–74.