Information and Computation 298 (2024) 105155

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

r-indexing the eBWT

Christina Boucher [a], Davide Cenzato [b], Zsuzsanna Lipták [c], Massimiliano Rossi [a], Marinella Sciortino [d,*]

[a] University of Florida, Gainesville, USA
[b] University Ca' Foscari, Venice, Italy
[c] University of Verona, Italy
[d] University of Palermo, Italy

ARTICLE INFO

Article history:
Received 7 March 2023
Received in revised form 26 February 2024
Accepted 29 February 2024
Available online 5 March 2024

Keywords:
Burrows–Wheeler transform
Extended BWT
Prefix-free parsing
r-index
MEMs
Circular strings

ABSTRACT

The extended Burrows-Wheeler Transform (eBWT) [Mantaci et al. TCS 2007] is a variant of the BWT, introduced for collections of strings. In this paper, we present the *extended r-index*, an analogous data structure to the r-index [Gagie et al. JACM 2020]. It occupies $\mathcal{O}(r)$ words, with r the number of runs of the eBWT, and offers the same functionalities as the r-index. We also show how to efficiently support finding maximal exact matches (MEMs). We implemented the extended r-index and tested it on circular bacterial genomes and plasmids, comparing it to five state-of-the-art compressed text indexes. While our data structure maintains similar time and memory requirements for answering pattern matching queries as the original r-index, it is the only index in the literature that can naturally be used for both circular and linear input collections. This is an extended version of [Boucher et al., *r-indexing the* eBWT, SPIRE 2021].

© 2024 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

There exist many sequencing projects that aim to identify the biological variation of individuals of a given species, such as the 100K Human Genome Project [51], the 1001 Arabidopsis Project [50], and the 3,000 Rice Genomes Project (3K RGP) [49]. Although biological variation is common and necessary among cultivars or individuals of these sequencing projects, a large portion of sequencing data is shared—leading to repetition within the dataset. Given the thousands of individual sequences within these sequencing projects, indexing them in a manner that allows variation to be identified and compared is challenging. The FM-index [17] has been the cornerstone of this indexing, as most standard read alignment algorithms (e.g., BWA [33] and Bowtie [31]) build and use the FM-index of the set of one or more reference genomes. More specifically, these read alignment algorithms build the Burrows-Wheeler Transform (BWT) and suffix array (SA) or a sample of the SA to find alignments between the sequence reads and the database of genomes. However, as the number of genomes increases, there has been an aim to build the BWT and SA-samples in space that is linear in the number of runs of the BWT typically denoted as r. This is because the number of runs is likely to be much smaller than the size of the text in the case of massive and repetitive collections.

* Corresponding author.
E-mail addresses: christinaboucher@ufl.edu (C. Boucher), davide.cenzato@unive.it (D. Cenzato), zsuzsanna.liptak@univr.it (Zs. Lipták), rossi.m@ufl.edu (M. Rossi), marinella.sciortino@unipa.it (M. Sciortino).

https://doi.org/10.1016/j.ic.2024.105155
0890-5401/© 2024 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Mäkinen and Navarro [35] introduced the *run-length FM-index* (RLFM), which is built on the run-length compressed BWT (RLBWT), and showed how to locate all occurrences of a query string $P[1..p]$ in text $T[1..n]$ in $\mathcal{O}\big((p + s \cdot occ)(\frac{\log\sigma}{\log\log r} + (\log\log n)^2)\big)$ time, where occ is the number of occurrences of P, σ is the alphabet size, and s is a parameter. The downside is that they require $\mathcal{O}(r + n/s)$ space for the RLBWT and the SA-samples. Given a query string P and the RLBWT of a string T, Policriti and Prezza [44] showed how to find one SA entry in the interval in RLBWT containing P in $\mathcal{O}(r)$ space. Later, Gagie et al. [19,20] showed how to fully support locate queries, i.e., locate all occ occurrences of P in $\mathcal{O}(r)$ space. The resulting data structure is referred to as the *r-index*. Recently, Nishimoto and Tabei [42] improved the previous running times for counting to $\mathcal{O}(p \log\log_w \sigma)$ and locating to $\mathcal{O}(p \log\log_w \sigma + occ)$, where w is the machine word size.

We note that Gagie et al. [19,20] defined the r-index but did not give an algorithm to construct it. An algorithm to construct the r-index was described later by Boucher et al. [6] and Kuhnle et al. [30]—both are based on a preprocessing technique called *Prefix-Free Parsing* (PFP). PFP produces two temporary structures called the *dictionary* and *parse*, from which the r-index can be built.

In a further development, Bannai et al. [3] showed how finding *maximal exact matches* (MEMs) with the r-index can be supported by computing the *matching statistics* of a query string P with respect to the text T. In order to compute the matching statistics, they described the addition of a small data structure to the r-index, referred to as *thresholds*. Rossi et al. [47] then showed how to compute the thresholds efficiently using the PFP based construction algorithm of the r-index.

1.1. Indexing string collections with the extended BWT

During the development of these data structures, however, little attention was paid to the fact that the input is typically a *string collection* (i.e., a multiset of strings), rather than an individual sequence. Indeed, this type of input is effectively just treated as if it was a single sequence: most tools use one of two existing methods, both of which concatenate the input strings, adding string separator symbols to mark string boundaries. In other words, they turn the input collection into one sequence, and then proceed to treat it as such, only making sure that no artificial matches (substrings spanning two consecutive input strings) are produced. Cenzato and Lipták [10,11] showed that these different methods are not equivalent; the output can vary significantly, extending to the number r of runs of the BWT. Moreover, the concatenation methods are input order dependent: if the same string collection is presented in a different order to the same tool, the resulting transform may be different, including the number of runs. For more details on the exact nature of these differences, see [10, 11]. The difference in the number of runs r has been shown to be as much as a multiplicative factor of 31 on real biological data (using the *same* method on the *same* input collection, varying only the input order), see Cenzato et al. [12].

While the fact that different methods co-exist does not compromise correctness of algorithms on the indexes, such as pattern matching or text access, it does have at least two serious implications: (1) Presenting the same string collection to different tools, or to the same tool but in different order, may result in a big variation in memory requirement of the resulting data structures (which depend on r). (2) The parameter r is increasingly being used as a repetitiveness measure *of the string collection itself*, similar to z (the number of phrases in the Lempel-Ziv 77 compression), or g (the size of the smallest grammar producing the text) [40]. But this measure is not well-defined if it can vary with the method used, as well as with the order in which the input sequences are presented.

Another issue is that many real-life strings, especially in biology, are not linear but circular. For example, bacterial genomes, human mitochondrial DNA, and many viral genomes are circular. This is of primary importance when analyzing metagenomes, since they typically contain very large bacterial communities [15]. Modelling circular strings as linear strings necessarily brings with it extra work, for example for accommodating substrings that span the end and the beginning of the linearization chosen.

Mantaci et al. in 2007 [38] proposed a mathematically sound extension of the BWT to string collections, which they called *extended BWT* (eBWT). The eBWT is independent of the input order, and thus does not suffer from the problem of different outputs on different permutations of the collection. Moreover, it can handle both circular and linear string collections: by default the input strings are regarded as circular, but if an end-of-string character (a \$) is added to each input string, then the result is equivalent to computing the classical BWT on the linear strings, input in lexicographical order [10,11].

Until recently, no efficient construction algorithm of the eBWT was known. In 2021, Bannai et al. presented such an algorithm [4]. Their algorithm, an adaptation of the well-known SAIS algorithm [43], was originally introduced for computing in linear time the *bijective BWT* [23], a BWT-variant for single texts. After additional linear-time preprocessing, this algorithm can also be used to construct the eBWT of a collection of strings, in total time linear in the size of the input collection. We simplified this algorithm in [8], removing the preprocessing step and modifying other details. We also presented a full implementation, and further combined our original algorithm with prefix-free parsing (PFP) [6]. Thus, for the first time, there exists the possibility of efficiently constructing the original eBWT for very large string collections.

1.2. Our contribution

In this paper, we extend our eBWT-construction of [8] to computing the analog of the r-index for the eBWT. We refer to our new data structure as *extended r-index*. We show how to adapt the PFP to the new problem, thus allowing us to handle very large string collections. The extended r-index has similar properties to the r-index of Gagie et al. [20]. First, its memory requirement is $\mathcal{O}(r)$ words, where r is the number of runs of the eBWT; in our case, this is independent of the input order. Next, it can return the number *occ* of occurrences of a pattern $P[1..p]$ (i.e., answer count queries) in $\mathcal{O}(p \log\log_w(\sigma + N/r))$ time, where N is the total length of the input collection, σ the size of the alphabet, and w the computer word size. Lastly, it can return all *occ* occurrences of P (i.e., answer locate queries) in $\mathcal{O}(p \log\log_w(\sigma + N/r) + occ \log\log_w(N/r))$ time. We also give details of how the data structure can be further extended to enable finding MEMs.

As the eBWT can handle both circular and linear strings, all of these results refer either to circular string collections, or to linear string collections. In the first case, *occ* denotes the number of circular (cyclic) occurrences of pattern P, i.e. including those that span the end and the beginning of some string in the collection. If, on the other hand, the input strings all have a final \$ appended, then the eBWT returns only linear occurrences of the input pattern, similar to the original r-index. Cyclic occurrences have been computed before, in the context of indexing the bijective BWT [2]. However, there an effort has to be made to avoid those occurrences that span the end and the beginning of the string (i.e., those that are not linear occurrences). In the current context, on the other hand, being able to find these occurrences is desired. This is something that the eBWT is able to do in a natural way, while the original r-index is not.

We implemented our extended r-index and compared it to five state-of-the-art text indexes ranging through the most popular strategies for indexing texts. These include a very efficient implementation of the original r-index; one of its latest improvements, the subsampled r-index [14]; an improved version of the LZ-parsing index of Kreft and Navarro [28]; an implementation of one of the most efficient grammar-based indexes by Claude et al. [13]; and CHICO [52], an improvement of the hybrid index of Ferrada et al. [16]. We tested the five competing indexes and our index by comparing the running time and memory usage to perform circular count and locate queries on three biological datasets containing bacterial genomes. We selected these datasets since indexing circular genomes is the most interesting scenario for performing circular count and locating queries.

The experimental results show that our text index behaves similarly to the r-index both regarding time and space requirements, while reporting all circular occurrences. Compared to the dictionary-based indexes, it is faster on all data sets but needs significantly more space.

As the other indexes do not naturally accept circular input sequences, one of two strategies have to be applied to handle circular string collections. Either one duplicates each input string and filters out duplicate linear occurrences in a post-processing step, thus paying both in space and time; or one just regards the input strings as linear, thus losing those occurrences that span the end and beginning of some string. To see how big this latter effect is, we compared the number of cyclic and linear occurrences of patterns of varying lengths, sampled from the input collections. We show that the difference between these numbers can be significant, especially when considering long patterns: for patterns of length 10,000, the number of linear occurrences is, on average, 50% less than the total number of (cyclic) occurrences.

Throughout the paper, we assume the word-RAM model, with machine word size $w = \Omega(\log N)$; thus, all arithmetic and logical operations on a machine word are performed in constant time.

1.3. Overview of the paper

In the next section, we introduce our notation and terminology (Section 2). In Section 3, we present and analyze our data structure, the extended r-index, and in Section 4 detail its construction using PFP. In Section 5, we explain how to construct the thresholds data structure along with the extended r-index to answer MEM queries, and in Section 6, we present our experimental results. We close with an outlook in Section 7.

2. Preliminaries

A string $T = T[1..n]$ is a sequence of characters $T[1] \cdots T[n]$ drawn from an ordered alphabet Σ of size σ. We denote by $|T|$ the length n of T. The empty string ϵ is the unique string of length 0. Given a multiset of m strings $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$, we denote the total length of the strings in $\mathcal{M}$ as $||\mathcal{M}||$, i.e., $||\mathcal{M}|| = |T_1| + \ldots + |T_m|$. We write $T[i..j]$ for the *substring* $T[i] \cdots T[j]$, where by convention, $T[i..j] = \epsilon$ if $i > j$. A *prefix* is a substring of the form $T[1..i]$ for some i, and a *suffix* one of the form $T[i..n]$, which we also write as $T[i..]$ for brevity. Given a positive integer $i \le n$ and a symbol $c \in \Sigma$, $rank_c(T, i)$ is defined as the number of occurrences of the character c in the prefix $T[1..i]$, while $select_c(T, i)$ is the position of the ith occurrence of c in T. Given two strings S and T, we denote by $\texttt{lcp}(S, T)$ the length of the *longest common prefix* (LCP) of S and T, i.e., $\texttt{lcp}(S, T) = \max\{i \mid S[1..i] = T[1..i]\}$.

Given a string $T = T[1..n]$ and an integer $k \ge 1$, the kn-length string $T^k = TT \cdots T$ is the k-fold concatenation of T, while $T^\omega = TT \cdots$ is the infinite string obtained by concatenating an infinite number of copies of T. A string T is called *primitive* if $T = S^k$ implies $T = S$ and $k = 1$. For any string T, there exists a unique primitive word S and a unique integer k such that $T = S^k$. We refer to $S = T[1..\frac{|T|}{k}]$ as $\text{root}(T)$ and to k as $\exp(T)$. Thus, $T = \text{root}(T)^{\exp(T)}$.

2.1. Suffix array and Burrows-Wheeler transform

We denote by $<_{\text{lex}}$ the lexicographic order: for two strings $S[1..m]$ and $T[1..n]$, $S <_{\text{lex}} T$ if S is a proper prefix of T, or there exists an index $1 \le i \le n, m$ such that $S[1..i-1] = T[1..i-1]$ and $S[i] < T[i]$. Given a string $T[1..n]$, the *suffix array* [37], denoted by $\text{SA} = \text{SA}_T$, is the permutation of $\{1, \ldots, n\}$ such that $T[\text{SA}[i]..]$ is the suffix which is the ith in lexicographic order among all non-empty suffixes of T. For an example, see Example 1. Given a string $T[1..n]$ and the SA of T, we denote the *inverse suffix array* as ISA, and define it as $\mathrm{ISA}[\mathrm{SA}[i]] = i$ for all $i = 1, \dots, n$. In several data structures (such as the FM-index [17] or the r-index [20]), instead of storing the entire array SA, only a proper subset is stored; these will be referred to as SA-*samples*.

A basic property of the SA is that, given a substring P of T, the set of occurrences of P appear as an interval in SA. Thus, P defines a unique interval $[s_P, e_P]$ of SA, namely $\mathrm{SA}[s_P]$ is the lexicographically least suffix and $\mathrm{SA}[e_P]$ the lexicographically largest suffix which have P as a prefix.

The string S is a *conjugate* of the string T if $S = T[i..n]T[1..i-1]$, for some $i \in \{1, \dots, n\}$; S is then also called the *ith rotation* of T and denoted $\mathrm{conj}_i(T)$. For a string T, the *conjugate array*[1] $\mathrm{CA} = \mathrm{CA}_T$ of T is the permutation of $\{1, \dots, n\}$ such that $\mathrm{CA}[i] = j$ if $\mathrm{conj}_j(T)$ is the ith conjugate of T with respect to the lexicographic order, with ties broken according to string order. In other words, if $\mathrm{CA}[i] = j$ and $\mathrm{CA}[i'] = j'$ for some $i < i'$, then either $\mathrm{conj}_j(T) <_{\mathrm{lex}} \mathrm{conj}_{j'}(T)$, or $\mathrm{conj}_j(T) = \mathrm{conj}_{j'}(T)$ and $j < j'$.

A string U is a *cyclic* or *circular substring* of string T if it is a substring of TT of length at most $|T|$, or equivalently, if it is the prefix of some conjugate of T. For example, ATA is a cyclic substring of AGCAT. It is sometimes also convenient to regard a given string $T[1..n]$ itself as *cyclic* (or *circular*); hence, we set $T[0] = T[n]$ and $T[n+1] = T[1]$.

Given a string T, the *Burrows-Wheeler Transform* [9], $\mathrm{BWT}(T)$, is a permutation of the characters of T which equals the last column of the matrix of the lexicographically sorted conjugates of T. The construction is reversible (up to rotation), allowing the original string T to be reconstructed [9]. The BWT itself can be computed from the conjugate array, since for all $i = 1, \dots, n$, $\mathrm{BWT}(T)[i] = T[\mathrm{CA}[i] - 1]$, where T is considered to be cyclic.

In many applications, it is assumed that an end-of-string-character (usually denoted \$), which is not element of Σ, is appended to the string. This character is assumed to be smaller than all characters from Σ. In this case, computing the conjugate array becomes equivalent to computing the suffix array, since $\mathrm{CA}_{T\$}[i] = \mathrm{SA}_{T\$}[i]$. Thus, applying one of the linear-time suffix array computation algorithms [45,39] leads to linear-time computation of $\mathrm{BWT}(T\$)$. In general, the two arrays CA and SA are not equal. However, there exist linear time algorithms for computing these two arrays also when no end-of-string marker is appended to the string. [22,8].

The LF-*mapping* (last-to-first mapping) [9] plays a central role in BWT-based algorithms. Given a string S of length n, the LF-mapping is a permutation of $\{1, \dots, n\}$ defined as: $\mathrm{LF}(j) < \mathrm{LF}(j')$ if $S[j] < S[j']$ or $S[j] = S[j']$ and $j < j'$. This permutation is also called *standard permutation of* S [34]. When S is the BWT of some primitive string, then the lexicographically smallest such T can be constructed using the LF-mapping, as follows [9]: $T[n] = S[1]$ and for $i \geq 1$, $T[n-i] = T[\mathrm{LF}^i(1)]$. In particular, the LF-mapping allows to walk through the original string T, in a back-to-front direction, by mapping the lexicographically jth conjugate $\mathrm{conj}_j(T)$ to the lexicographic rank of $\mathrm{conj}_{j-1}(T)$. The same mapping is fundamental for *backward search* [17], the efficient pattern matching algorithm on BWT-based data structures. For more details, see [41].

Another fundamental permutation is the ϕ-*mapping* introduced in [26]. This permutation is defined as $\phi(i) = \mathrm{SA}[\mathrm{ISA}[i] - 1]$ if $\mathrm{ISA}[i] > 1$, and $\phi(i) = \mathrm{SA}[n]$ otherwise. In other words, ϕ maps the ith suffix $T[i..]$ to the lexicographically next smaller suffix. In terms of the suffix array, this can be expressed as: $\phi(\mathrm{SA}[1]) = \mathrm{SA}[n]$ and $\phi(\mathrm{SA}[j]) = \mathrm{SA}[j-1]$, for $j > 1$. See Example 1 for an example. We note that the ϕ-function can be similarly defined on the basis of the conjugate array CA, and the LF-mapping on that of the SA (if an end-of-string marker is present).

In a certain sense, the LF-mapping and the permutation ϕ are analogous: LF gives the SA-index (or CA-index) of the previous suffix (or conjugate), where *previous* refers to text order; ϕ gives the text index of the previous suffix (or conjugate), where *previous* refers to SA-order (or CA-order). In other words, LF walks through the text, back-to-front, using suffix array (or conjugate array) indices, and ϕ walks through the suffix array (or conjugate array), back-to-front, using text indices.

Example 1. Let $T = \mathtt{GATAT}$, and $T' = \mathtt{GATAT\$}$. In Fig. 1, for each string, we list its BWT, LF-mapping, and ϕ. We also give the CA and the SA for T and T', respectively.

2.2. *r-index*

A common parameter in the context of the BWT is the number r of the equal-letter runs of the BWT of a string. For example, as shown in Fig. 1, $r = 3$ for the string $T = \mathtt{GATAT}$ since $BWT(T) = \mathtt{GTTAA}$. The memory required by the run-length encoded version of the BWT is $\mathcal{O}(r)$. The r-index [20], building on the run-length encoded FM-index of Mäkinen and Navarro [35,36] and the Toehold Lemma by Policriti and Prezza [44], is a data structure that not only requires $\mathcal{O}(r)$ memory only, but supports pattern matching queries in time which depends almost entirely on the size of the pattern. In detail, let w be the size of a computer word. Given a pattern P of length p, the r-index returns the number of occurrences of P (denoted as *occ*) in $\mathcal{O}(p \log\log_w(\sigma + n/r))$ time (a *count query*), and, after having answered the count query, returns all *occ* occurrences of P in $\mathcal{O}(\log\log_w(n/r))$ time per occurrence. In other words, it answers *locate queries* in total time $\mathcal{O}(p \log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$.

[1] The conjugate array CA is called *circular suffix array* and denoted $\mathrm{SA}_\circ$ in [25,4], and *BW-array* in [29], but in both cases defined for primitive strings only.

$T = \texttt{GATAT}$

i	ϕ_T	CA_T	LF_T	BWT_T	
1	4	2	3	G	ATATG
2	5	4	4	T	ATGAT
3	1	1	5	T	GATAT
4	2	3	1	A	TATGA
5	3	5	2	A	TGATA

$T' = \texttt{GATAT\$}$

i	$\phi_{T'}$	$\mathrm{SA}_{T'}$	$\mathrm{LF}_{T'}$	$\mathrm{BWT}_{T'}$	
1	2	6	5	T	$GATAT
2	4	4	6	T	AT$GAT
3	5	2	4	G	ATAT$G
4	6	1	1	$	GATAT$
5	1	5	2	A	T$GATA
6	3	3	3	A	TAT$GA

Fig. 1. Figure for Example 1.

Two ideas are crucial in how pattern matching with the r-index works: (1) the Toehold Lemma [44] allows to produce *one* occurrence along with answering a count query (the *toehold value*), and (2) repeated application of the ϕ-function returns all occurrences contained in the SA-interval of P.

The r-index consists of three main components: (1) a data structure that stores the run-length encoded BWT supporting LF-mapping queries, (2) an SA-sample for each of the r runs, and (3) a data structure supporting ϕ operations. In particular, the data structure (1) combines the RLFM index [36] with the data structures for rank and predecessor queries described in [5], while (2) is an array storing the SA-samples at the end of each run. Finally, (3) is implemented as a predecessor data structure built on SA-samples at the beginning of each run, with the corresponding SA-sample at the end of the previous run as satellite information.

2.3. Maximal exact matches with the r-index

Maximal exact matches (MEMs) are exact matches between a text T and a pattern P that can neither be extended to the left nor to the right. More formally, $P[i..i+\ell-1]$ of length ℓ is a MEM of P in T if (1) $P[i..i+\ell-1]$ occurs in T and (2) either $i=1$ or $P[i-1..i+\ell-1]$ does not occur in T, and (3) either $i+\ell-1=|P|$ or $P[i..i+\ell]$ does not occur in T.

A classic application for MEMs is finding seeds between a short sequence and a genome for computing multiple sequence alignments of both short and long reads [32]. As previously noted, the main components of the r-index, namely the run-length BWT and the SA-samples, are not sufficient to find MEMs efficiently. Bannai et al. [3] showed that MEM-finding can be supported by computing the matching statistics (MS) of a query string P with respect to a text T. The *matching statistics* of $P[1..p]$ with respect to $T[1..n]$ are defined as an array of pairs $MS[1..p]$ whose ith entry consists of the length and an occurrence of the longest prefix of $P[i..p]$ which occurs in T. More formally, for $1 \le i \le p$, $MS[i] = (\ell_i, j_i)$, where (1) $T[j_i..j_i+\ell_i-1] = P[i..i+\ell_i-1]$ and (2) either $i+\ell_i > p$ or $P[i..i+\ell_i]$ does not occur in T. The next lemma is a slight variation of Lemma 1 from [47]; it can be used to compute MEMs from the matching statistics with a left-to-right scan.

Lemma 1. *Given an input text $T[1..n]$ and a query string $P[1..p]$, let $MS[1..p] = [(\ell_1, j_1), \ldots, (\ell_p, j_p)]$ be the matching statistics of P with respect to T. For all positions $i = 1, \ldots, p$, $P[i..i+\ell-1]$ is a MEM of length ℓ in T if and only if $\ell_i = \ell$, and either $i = 1$ or $\ell_{i-1} \le \ell_i$ (in the case $i > 1$).*

In order to compute the matching statistics with the r-index, Bannai et al. [3] described a two-pass algorithm. First, working right to left, each suffix $P[i..p]$ of P is processed to find a position in the text T which corresponds to the longest prefix of $P[i..p]$ occurring in T, in this way populating the values of j_i. Then, working left to right, one uses random access to T to determine the length of those matches.

The first pass requires the addition of a small data structure to the r-index referred to as *thresholds*; these serve to compute the position of a longest prefix of the ith suffix of P and T when a mismatch is found while backward searching for P on $\mathrm{BWT}(T)$. Thresholds are defined for each two consecutive runs of the same character. The original definition of [3] was later modified by Rossi et al. [47]. We use the latter definition.

Given a text T and four integers $1 \le j' \le j < k \le k' \le n$ such that $\mathrm{BWT}(T)[j'..j]$ and $\mathrm{BWT}(T)[k..k']$ are two consecutive runs of the same character in $\mathrm{BWT}(T)$. The *threshold* p for the pair of runs $\mathrm{BWT}(T)[j'..j]$ and $\mathrm{BWT}(T)[k..k']$ of character $\mathrm{BWT}(T)[j]$ is defined as the smallest index i in the interval $[j+1..k]$ such that $\mathrm{LCP}[i] = \min \mathrm{LCP}[j+1..k]$.

$\mathcal{M} = \{\texttt{AAT}, \texttt{AATAT}, \texttt{GATAATAA}, \texttt{AGA}\}$

i	$\mathrm{GCA}_{\mathcal{M}}$	$\mathrm{LF}_{\mathcal{M}}$	$\mathrm{eBWT}_{\mathcal{M}}$	
1	(3,4)	13	G	AAG
2	(7,3)	15	T	AAGATAAT
3	(4,3)	16	T	AATAAGAT
4	(1,1)	17	T	AAT
5	(1,2)	18	T	AATAT
6	(1,4)	1	A	AGA
7	(8,3)	2	A	AGATAATA
8	(5,3)	3	A	ATAAGATA
9	(2,3)	14	G	ATAATAAG
10	(2,1)	4	A	ATA
11	(4,2)	19	T	ATAAT
12	(2,2)	5	A	ATATA
13	(2,4)	6	A	GAA
14	(1,3)	7	A	GATAATAA
15	(6,3)	8	A	TAAGATAA
16	(3,3)	9	A	TAATAAGA
17	(3,1)	10	A	TAA
18	(5,2)	11	A	TAATA
19	(3,2)	12	A	TATAA

Fig. 2. An illustration of the eBWT for the multiset of strings $\mathcal{M}$. We report, from left to right, the index i, the generalized conjugate array GCA for $\mathcal{M}$, the LF permutation, the eBWT, and the conjugates of $\mathcal{M}$ sorted according to the ω-order. Highlighted in red: the GCA-samples at the beginning and at the end of eBWT runs. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

2.4. *Generalized conjugate array and* eBWT

The *ω-order* [21,38] (denoted by $\prec_\omega$) is defined as follows: for two strings S and T, $S \prec_\omega T$ if $\mathrm{root}(S) = \mathrm{root}(T)$ and $\exp(S) < \exp(T)$, or $S^\omega <_{\text{lex}} T^\omega$ (this implies $\mathrm{root}(S) \neq \mathrm{root}(T)$). It can be verified that the ω-order relation is different from the lexicographic order. For instance, $\texttt{CG} <_{\text{lex}} \texttt{CGA}$ but $\texttt{CGA} \prec_\omega \texttt{CG}$.

Given a multiset of strings $\mathcal{M} = \{T_1, \dots, T_m\}$, the *generalized conjugate array* $\mathrm{GCA} = \mathrm{GCA}_{\mathcal{M}}$ is an array of length $||\mathcal{M}||$ whose kth entry $\mathrm{GCA}[k]$ equals (j, d) if $\mathrm{conj}_j(T_d)$ is the kth string in ω-sorted list of the conjugates of all strings of $\mathcal{M}$, with ties broken first w.r.t. the index of the string (in case of identical strings) in the multiset, and then w.r.t. the index in the string itself. The *text order* for the entries of the generalized conjugate array $\mathrm{GCA}_{\mathcal{M}}$ is defined as follows: we say that (j, d) precedes (j', d') in *text order* if either $d < d'$ or $d = d'$ and $j < j'$.

Given a string $P[1..p]$, referred to as *pattern*, we say that P *occurs* in $\mathcal{M} = \{T_1, T_2, \dots, T_m\}$ if P occurs as a cyclic substring of one of $T_1, T_2, \dots, T_m$. More formally, we define a *cyclic occurrence* (or simply *occurrence*) of P in $\mathcal{M}$ as a pair (j, d) such that P is a prefix of $\mathrm{conj}_j(T_d)$. For clarity, we call an occurrence (j, d) of pattern P *linear* if $1 \le j \le |T_d| - p$.

The *extended Burrows–Wheeler Transform* [38] (eBWT) is a generalization of the BWT to a multiset $\mathcal{M} = \{T_1, \dots, T_m\}$ of strings that is independent of the order in which the strings appear in the multiset. The $\mathrm{eBWT}(\mathcal{M})$ is a permutation of the characters of the strings in $\mathcal{M}$ which is based on the ω-order between the conjugates of the strings in $\mathcal{M}$. Such a permutation is obtained by sorting all the conjugates of the strings in the multiset according to the ω-order and by concatenating the last character of each conjugate in the sorted list. The $\mathrm{eBWT}(\mathcal{M})$ can be computed by using the generalized conjugate array, since for all $k = 1, \dots, ||\mathcal{M}||$, $\mathrm{eBWT}(\mathcal{M})[k] = T_d[j-1]$ if $\mathrm{GCA}[k] = (j, d)$, where the string $T_d \in \mathcal{M}$ is considered to be cyclic.

In [8], we showed how to compute the eBWT and the generalized conjugate array $\mathrm{GCA}_{\mathcal{M}}$ in time linear in the total size of $\mathcal{M}$, also when non-primitive strings are included in $\mathcal{M}$.

Similarly to the BWT, the eBWT is a reversible transformation, i.e., starting from $\mathrm{eBWT}(\mathcal{M})$ the original strings of the multiset $\mathcal{M}$ can be recovered (up to rotation). Such a recovering can be realized by using the LF-mapping denoted by $\mathrm{LF}_{\mathcal{M}}$ and defined as the standard permutation of the string $\mathrm{eBWT}(\mathcal{M})$. In particular, for each $d = 1, \dots, m$, the lexicographically smallest rotation of the string T_d is constructed as follows: $T_d[|T_d|] = \mathrm{eBWT}(\mathcal{M})[k_d]$, with $\mathrm{GCA}[k_d] = (1, d)$, and for $i \ge 1$, $T_d[|T_d| - i] = T_d[\mathrm{LF}^i_{\mathcal{M}}(k_d)]$.

Example 2. In Fig. 2, we show the eBWT, GCA, and LF-mapping for the multiset of strings $\mathcal{M} = \{\texttt{AAT}, \texttt{AATAT}, \texttt{GATAATAA}, \texttt{AGA}\}$. We highlight the difference between the ω-order and lexicographic order in the GCA by noting that $\mathrm{GCA}_{\mathcal{M}}[9] = (2, 3)$ and $\mathrm{GCA}_{\mathcal{M}}[10] = (2, 1)$ corresponding to conjugates $\texttt{ATAATAAG}$ and $\texttt{ATA}$ respectively.

The definition of the ϕ-mapping can be naturally extended to the generalized conjugate array $\mathrm{GCA}_{\mathcal{M}}$ as follows: $\phi_{\mathcal{M}}(\mathrm{GCA}[1]) = \mathrm{GCA}[||\mathcal{M}||]$ and for each $h > 1$ $\phi_{\mathcal{M}}(\mathrm{GCA}[h]) = \mathrm{GCA}[h-1]$.

3. The extended *r*-index

In this section, we show how to extend the r-index to the eBWT. Throughout the rest of the paper, we assume that all strings in $\mathcal{M}$ are primitive, and that the multiset $\mathcal{M}$ is *conjugate-free*, i.e. no two strings are conjugates. The first assumption is justified, since we showed in our previous work how, given a multiset of strings $\mathcal{M} = \{T_1, \ldots, T_m\}$, the $\text{GCA}_{\mathcal{M}}$ can be computed from the GCA of the roots of the strings in $\mathcal{M}$ [8]. In addition, the ϕ-mapping can be adapted to work with non-primitive strings by storing some additional information, such as the exponents of the non-primitive strings in $\mathcal{M}$. The assumption that $\mathcal{M}$ does not contain two strings which are conjugate is more restrictive. In Section 6 we detail how we deal with input collections for which this condition does not hold. Note in particular that the fact that $\mathcal{M}$ is conjugate-free implies that $\mathcal{M}$ is a set.

Finally, given a pattern $P[1..p]$, we assume the pattern length to be $p \leq \min\{|s| : s \in \mathcal{M}\}$. This assumption is realistic in most practical scenarios. In the rest of the paper, we will denote the total length of strings in $\mathcal{M}$ by N.

3.1. The data structures for the extended r-index

The *extended r-index* consists of three main components: (1) a data structure supporting backward search queries on the run-length encoded eBWT, (2) a data structure computing the toehold value during the backward search, and (3) a data structure computing $\phi_{\mathcal{M}}$ operations on the GCA.

We construct (1) by extending the RLFM-index by Mäkinen and Navarro [35] to the eBWT, and describe it using the definitions and notation of Gagie et al. [20]. It consists of four elements:

(i) a data structure $E[1..r]$ which supports predecessor search queries and stores the first position of each eBWT run (*run heads*). The ith entry $E[i]$ is the beginning of the ith run;
(ii) a sequence $L[1..r]$ storing the symbol of the eBWT runs such that $L[i]$ is the symbol of the ith run in the eBWT (from left to right). L supports rank and select operations;
(iii) a data structure $D[1..r]$ which stores the cumulative lengths of the eBWT runs of the same symbol after stably sorting them according to the lexicographic order of their characters;
(iv) two vectors $C[1..\sigma]$ and $C'[1..\sigma]$: entry $C[c]$ contains the number of characters smaller than c in the eBWT, and $C'[c]$ contains the number of characters smaller than c in L.

In Example 3, we show the content of these data structures when the set $\mathcal{M} = \{\texttt{AAT}, \texttt{AATAT}, \texttt{GATAATAA}, \texttt{AGA}\}$ is considered.

We implement the backward search procedure to search a pattern $P[1..p]$ in the eBWT using these four data structures. In particular, we compute the interval $\text{GCA}[i..j]$ for P by using $2p$ rank queries on the run-length encoded eBWT. Each rank query counts the occurrences of a character c in an eBWT prefix $\text{eBWT}[1..i]$ and can be computed efficiently using the E, L, D and C' data structures as shown in [20, Section 2.5] (see Example 3). Then, once the results of the rank queries have been computed, the C vector is used to get the correct eBWT interval. All these data structures can be stored in $\mathcal{O}(r)$ words with up-to-date implementations (see Section 6.1). Note that also the select queries are computed using the same four data structures. We recall that rank and select queries can be formally defined as follows: given an array S of length n and a symbol c, $rank_c(S, i)$ returns the number of times in which c appears in the prefix $S[1, i]$; $select_c(S, j)$ returns the position i of the jth occurrence of c in $S[1, n]$. Moreover, given an ordered array S and a symbol c, where S contains elements from a totally ordered set U and $c \in U$, the predecessor query $pred(S, c)$ returns the position of the largest element in S smaller than or equal to c. In Example 3, we also show how these operations work on the previously defined data structures, and how to apply them for the computation described above.

Example 3. In Fig. 2, we have the eBWT and the GCA of a string collection $\mathcal{M}$. Following the above list, (i) $E = [1, 2, 6, 9, 10, 11, 12]$ and answers predecessor queries like $pred(E, 7) = 3$, where $E[3] = 6$ is the predecessor of 7 in E; (ii) $L = [\texttt{G}, \texttt{T}, \texttt{A}, \texttt{G}, \texttt{A}, \texttt{T}, \texttt{A}]$ and answers rank and select queries like $rank_{\texttt{A}}(L, 5) = 2$, and $select_{\texttt{A}}(L, 2) = 5$; (iii) $D = [3, 4, 12, 1, 2, 4, 5]$ corresponding to the sorted runs $(\texttt{A}, 3)(\texttt{A}, 1)(\texttt{A}, 8)(\texttt{G}, 1)(\texttt{G}, 1)(\texttt{T}, 4)(\texttt{T}, 1)$; (iv) $C = [0, 12, 14]$, and $C' = [0, 3, 5]$ for the $\texttt{A}, \texttt{G}, \texttt{T}$ characters.

We compute the rank query $rank_{\texttt{A}}(\text{eBWT}, 13)$ as follows. The predecessor of 13 in E is 12 in position 7, and $\text{eBWT}[13]$ is in a run of $\texttt{A}$ since $L[7] = \texttt{A}$. With $rank_{\texttt{A}}(L, 7) = 3$, we detect that $\text{eBWT}[13]$ is within the third run of $\texttt{A}$. We now access D to get $D[C'[A] + (3 - 1)] = 4$, and compute the final result $rank_{\texttt{A}}(\text{eBWT}, 13) = 4 + (13 - 12) + 1 = 6$.

We construct (2) by augmenting the backward search procedure to find a toehold [20, Lemma 3.2] in the GCA, rather than in the SA, and (3) by extending the locate machinery in [20, Section 3] to compute ϕ operations on the GCA rather than on the SA. We do this by including the following data structures:

(v) a data structure $G[1..r]$ which stores the GCA-samples at the end of the eBWT runs. The ith entry $G[i]$ is the GCA-sample at the end of the ith run,
(vi) a data structure $F[1..r]$ which supports circular predecessor search queries and stores the GCA-samples at the beginning of the eBWT runs in text order. The ith entry $F[i]$ is the ith GCA-sample at the beginning of an eBWT run in text order;

(vii) a data structure $FirstToRun[1..r]$ which stores the mapping between the GCA-samples in F and G. The ith entry $FirstToRun[i] = i'$ indicates that $F[i]$ and $G[i']$ are the GCA-samples at the beginning and end of the i'th eBWT run, respectively;

(viii) a data structure $B[1..m]$ which stores the length of each input string. The ith entry $B[i]$ is the length of the ith string in $\mathcal{M}$.

The content of these data structures for the input string collection $\mathcal{M} = \{\mathtt{AAT}, \mathtt{AATAT}, \mathtt{GATAATAA}, \mathtt{AGA}\}$ is described in Example 4. Notice that since we are working with collections of circular strings, the predecessor search queries on F have to work circularly. Given a GCA-sample, $s = (j, d)$, the circular predecessor query $pred_{circ}(F, s)$ returns the position of the GCA-sample (j', d) in F such that j' is the largest positive integer smaller or equal than j. If there is no such a GCA-sample in F, we ensure the circular behavior by searching the predecessor of $(B[d], d)$, i.e., the sample (j', d) such that j' is the largest integer in $\{j + 1, .., B[d]\}$.

Example 4. Continuing Example 3, the data structure given in (v) stores the GCA-samples at the end of the eBWT runs $G = [(3, 4), (1, 2), (5, 3), (2, 3), (2, 1), (4, 2), (3, 2)]$, the one in (vi) stores the GCA-samples at the beginning of the eBWT runs sorted in text order $F = [(2, 1), (2, 2), (4, 2), (2, 3), (7, 3), (1, 4), (3, 4)]$. The data structure of (vii) stores the mapping between G and F, $FirstToRun = [5, 7, 6, 4, 2, 3, 1]$. Finally, (viii) stores the input string lengths $B = [3, 5, 8, 3]$.

We now describe how to compute the toehold value for the GCA during the backward search procedure. First, we need to augment the backward search in a way that, at each step, in addition to the interval $[i..j]$ of GCA, it also returns GCA$[j]$. We always store the toehold value as the last value of the interval $[i..j]$. Assume that we are searching for a pattern $P[1..p]$ and we matched it up to position p'. The interval of GCA corresponding to $P[p'..p]$ is $[i..j]$; the toehold for this interval is GCA$[j]$. We update the toehold for the next backward search step as follows. Let $[i'..j']$ be the interval of GCA for $P[p' - 1..p]$ and $c = P[p' - 1]$. We need to distinguish between two cases. If eBWT$[j] \neq c$ then the last occurrence of c in eBWT$[i..j]$ is the last character of a run. Therefore, we can compute the new toehold value GCA$[j']$ by finding the position of this character c in the input collection as follows. We use rank and select queries on eBWT to compute the position k of this character c in eBWT, thus, $i \leq k \leq j$. Then we use the predecessor query $pred(E, k)$ to find the position h in E of the first character of the corresponding eBWT run containing this c. So, accessing $G[h] = (x, d)$ allows us to compute the new toehold value GCA$[j'] = (x - 1, d)$. Otherwise, if eBWT$[j] = c$, we simply update the current toehold value (x, d) to $(x - 1, d)$. Notice that also the update of the toehold value has to work circularly. If $x = 1$ then we compute the updated toehold as $(x - 1, d) = (B[d], d)$. We keep updating the toehold with this procedure until we have the final GCA-interval for $P[1..p]$.

Example 5. Continuing Example 3, assume that we are locating pattern $P = \mathtt{AAG}$. The GCA-interval for the empty string is $[1..19]$ and the toehold value is $(3, 2)$. Using the LF-mapping, we compute the new GCA-interval for $P[3..3] = \mathtt{G}$, $[13..14]$. Since eBWT$[19] \neq \mathtt{G}$, we compute the new toehold GCA$[14]$ as follows: $rank_{\mathtt{G}}(\text{eBWT}, 19) = 2$, thus, $select_{\mathtt{G}}(\text{eBWT}, 2) = 9$, meaning that the last occurrence of $\mathtt{G}$ is in position 9. Finally, we compute $pred(E, 9) = 4$ and $G[4] = (2, 3)$. So, the updated toehold value is $(2 - 1, 3) = (1, 3)$. Then we compute the GCA-interval for $P[2..3] = \mathtt{AG}$, which is $[6..7]$. In this case, eBWT$[14] = \mathtt{A}$. Thus, we compute the position preceding $(1, 3)$ (circularly, using the length of T_3, stored in $B[3]$), i.e., $(8, 3)$. Finally, we compute the GCA-interval $[1..2]$ for $P[1..3]$ with toehold GCA$[2] = (7, 3)$.

The last element we need to implement in our locate machinery is a data structure able to compute ϕ operations on the GCA. Given GCA$[i] = (j, d)$, we compute $\phi_{\mathcal{M}}(\text{GCA}[i]) = \text{GCA}[i - 1]$ by using a predecessor search query on F and one access to $FirstToRun$ and to G each, as follows. First, we obtain the predecessor (j', d) of (j, d) in F, by a predecessor query on F, $h = pred_{circ}(F, (j, d))$, and $F[h] = (j', d)$. Then we compute the distance (number of LF-steps) of position (j, d) from position (j', d), namely $\Delta = j - j'$. Note that if $j' > j$ then the distance is computed in a circular way as $\Delta = j + B[d] - j'$. We get the final result by adding Δ to the sample in $G[FirstToRun[h] - 1] = (x, d')$, i.e., $\phi_{\mathcal{M}}(\text{GCA}[i]) = (((x + \Delta - 1) \bmod B[d']) + 1, d')$.

Example 6. Continuing Example 3, say we have already computed the GCA-interval $[3..5]$ of the pattern $P = \mathtt{AAT}$ with toehold GCA$[5] = (1, 2)$. We get $\phi_{\mathcal{M}}(\text{GCA}[5])$ as follows. We compute a circular predecessor search query $pred_{circ}(F, (1, 2)) = 3$, and find that $F[3] = (4, 2)$. We compute the number of LF-steps separating the two GCA samples, $\Delta = (1, 2) - (4, 2) = 1 + (5 - 4) = 2$. Now we get the GCA-sample at the end of the previous run $G[FirstToRun[3] - 1] = G[6 - 1] = (2, 1)$, and compute the final result GCA$[4] = (1 + ((2 + \Delta - 1) \bmod 3), 1) = (1, 1)$.

We store G, F, B, and $FirstToRun$ in $\mathcal{O}(r)$ words (Proposition 5), and give details of the implementation in Section 6.1.

3.2. Analysis of the extended r-index

We will now show that the memory requirement of our data structure is $\mathcal{O}(r)$ words, and that it can answer count and locate queries in time analogous to the original r-index. Recall also that all strings in $\mathcal{M}$ are assumed to be primitive and that $\mathcal{M}$ is a conjugate-free set.

We first need a technical lemma:

Lemma 2. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N,* GCA *its generalized conjugate array, with* $\mathrm{GCA}[h] = (j_h, d_h)$ *for* $1 \le h \le N$. *Let k be such that* $\mathrm{eBWT}[k] = \mathrm{eBWT}[k-1]$, *and let* $k' = \mathrm{LF}(k)$. *Then* $\mathrm{LF}(k-1) = k' - 1$. *In particular,* $\mathrm{GCA}[k'-1] = (j_{k-1} - 1, d_{k-1})$.

Proof. This follows directly from the fact that the eBWT has the LF-property, i.e., that same characters appear in the same order in the last and the first column of the eBWT-matrix, see Proposition 10 in [38]. □

Example 7. Fig. 2 illustrates an example of the property in Lemma 2. Let $k = 13$. As $\mathrm{eBWT}(13) = \mathtt{A} = \mathrm{eBWT}(12)$, we have $\mathrm{LF}(12) = 5 = 6 - 1 = \mathrm{LF}(13) - 1$, as claimed. Moreover, $\mathrm{GCA}(5) = (1, 2) = (j_{12} - 1, d_{12})$, since $k - 1 = 12$ and $\mathrm{GCA}(12) = (2, 2)$.

To analyze the space required by the extended r-index, we show that every string in $\mathcal{M}$ is sampled at least once among the run-beginning samples and at least once among the run-end samples.

Proposition 3. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N,* GCA *its generalized conjugate array, with* $\mathrm{GCA}[h] = (j_h, d_h)$ *for $h = 1, \ldots, N$. Then, for each i, $1 \le i \le m$, there exist integers k and k', such that $d_k = d_{k'} = i$ and*

1. *either $k = 1$ or* $\mathrm{eBWT}[k] \neq \mathrm{eBWT}[k-1]$, *and*
2. *either $k' = N$ or* $\mathrm{eBWT}[k'] \neq \mathrm{eBWT}[k'+1]$.

Proof. We assume (for contradiction) that there exists an integer i, with $1 \le i \le m$, such that at least one of the two following conditions holds:

1. $d_1 \neq i$ and, for all $1 < h \le N$ such that $d_h = i$, $\mathrm{eBWT}_{\mathcal{M}}[h] = \mathrm{eBWT}_{\mathcal{M}}[h-1]$;
2. $d_N \neq i$ and, for all $1 \le h < N$ such that $d_h = i$, $\mathrm{eBWT}_{\mathcal{M}}[h] = \mathrm{eBWT}_{\mathcal{M}}[h+1]$.

We assume w.l.o.g. that 1 holds. By iteratively applying Lemma 2, we see that if T_i is never at a run beginning, then every conjugate of T_i is preceded by a conjugate of the same string in $\mathrm{GCA}_{\mathcal{M}}$. Formally: there exists an integer $1 \le i' \le m$, with $i' \neq i$, such that for all $1 < h \le N$, if $d_h = i$, then $d_{h-1} = i'$. Let $u = \mathrm{conj}_{j_{h-1}}(T_{d_{h-1}})$ and $v = \mathrm{conj}_{j_h}(T_{d_h})$. Since we are assuming that the respective eBWT-characters are always the same (as no conjugate of T_i is at a run beginning), we conclude that $\mathrm{root}(u) = \mathrm{root}(v)$. In fact, every time we make an LF-step, the conjugate index in $\mathrm{GCA}_{\mathcal{M}}$ is decreased by 1, and after $|v|$ steps, the index of the conjugate of u must be j_{h-1}. This implies that $\mathrm{root}(u) = \mathrm{root}(v)$. Since all strings in $\mathcal{M}$ are primitive, this implies that $u = v$, and thus, T_i and $T_{i'}$ are conjugate. This is a contradiction to $\mathcal{M}$ being a conjugate-free set. □

Proposition 3 allows us to deduce the following relationship between the number r of runs and the number m of strings.

Corollary 4. *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N, r the number of runs of its* eBWT. *Then $m \le r$. Moreover, if all strings T_i have the same length ℓ, then $N/r \le \ell$.*

Proof. Both statements follow directly from Proposition 3. □

We are ready to state the memory requirement of our data structure:

Proposition 5. *The extended r-index of a conjugate-free set of primitive strings requires $\mathcal{O}(r)$ words of storage, where r is the number of runs of the* eBWT *of $\mathcal{M}$.*

Proof. The data structures E, L, D, G, F, and $FirstToRun$, as presented in Section 3.1, occupy $\mathcal{O}(r)$ words each, C and C' occupy $\mathcal{O}(\sigma)$ words, and B occupies $\mathcal{O}(m)$ words. Since $m \le r$ by Corollary 4, altogether we have $\mathcal{O}(r)$ space. □

Count queries are performed analogously to the r-index.

Proposition 6. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, and r the number of runs of* $\mathrm{eBWT}(\mathcal{M})$. *We can build an index of $\mathcal{O}(r)$ words such that, later, given a pattern $P[1..p]$, we can return the number of cyclic occurrences of P in $\mathcal{M}$ in $\mathcal{O}(p \log\log_w(\sigma + N/r))$ time.*

Proof. Count queries are answered by applying p backward search steps, i.e., $2p$ rank queries, starting from the complete interval $[1, N]$ and computing the interval $[s_P, e_P]$ which contains the conjugates of which the pattern P is a prefix. The number of occurrences is then $occ = e_P - s_P + 1$. Since our strategy and data structures are exactly analogous to the ones used for the r-index, the time complexity follows by using similar arguments as in [20, Lemma 2.1]. □

Next, we give the analog of the Toehold Lemma [44,20].

Proposition 7. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N and r the number of runs of* eBWT($\mathcal{M}$). *We can build an index of $\mathcal{O}(r)$ words such that, later, given a pattern $P[1..p]$, we can return the interval $[s_P, e_P]$ of cyclic occurrences of P in $\mathcal{M}$, along with one position q and the content* GCA[q], *in $\mathcal{O}(p \log\log_w(\sigma + N/r))$ time.*

Proof. As described in Section 3.1, we compute the toehold value similarly to the r-index, employing the additional data structures for the extended r-index, which allow us to handle circular predecessor queries and navigation between different strings in the string collection. The running time follows again analogously to [20, Lemma 3.2]. □

The next lemma, the analog of Lemma 3.5 in [20], states that one $\phi_{\mathcal{M}}$-value can be computed in $\mathcal{O}(\log\log_w(N/r))$ time. Together with Proposition 7, this will allow us, analogously to the r-index, to output all *occ* occurrences of a pattern P.

Lemma 8. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N and r the number of runs of* eBWT($\mathcal{M}$). *Using the extended r-index of $\mathcal{M}$, we can evaluate $\phi_{\mathcal{M}}$ in $\mathcal{O}(\log\log_w(N/r))$ time.*

Proof. As described in Section 3.1, the evaluation of $\phi_{\mathcal{M}}$ is realized analogously to the r-index. Our additional data structures that allow the predecessor data structure E to work circularly, do not affect the $\mathcal{O}(\log\log_w(N/r))$ query time proved in [5]. □

We summarize the properties of the extended r-index in the following theorem.

Theorem 9. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, and let r be the number of runs of* eBWT($\mathcal{M}$). *We can build an index of $\mathcal{O}(r)$ words such that, later, given a pattern $P[1..p]$, we can return the number of cyclic occurrences of P in $\mathcal{M}$ in $\mathcal{O}(p \log\log_w(\sigma + N/r))$ time, and after counting, return all occ cyclic occurrences of P in $\mathcal{M}$ in $\mathcal{O}(occ \log\log_w(N/r))$ time.*

Proof. The extended r-index can be stored in $\mathcal{O}(r)$ words by Proposition 5. Given a query pattern P, in $\mathcal{O}(p \log\log_w(\sigma + N/r))$ time the GCA-interval of its occurrences $[s_P, e_P]$ can be computed, along with one occurrence in $\mathcal{M}$ (the toehold value) by Proposition 7. By our construction, this toehold value is always the last in the interval, i.e. GCA(e_P). Then, by repeated applications of $\phi_{\mathcal{M}}$, we can compute all *occ* occurrences of P, each in time $\mathcal{O}(\log\log_w(N/r))$ by Lemma 8. □

4. Efficient construction of the extended *r*-index

In [8], we showed how to efficiently construct the eBWT of large string collections in a way that preserves the original definition of Mantaci et al. [38]. We do this by combining two algorithms: a modified version of the Suffix Array Induced Sorting (SAIS) algorithm of Nong et al. [43] with a variant of the prefix-free parsing algorithm (PFP) of Boucher et al. [6]. In this section, we augment the PFP algorithm to compute the GCA-samples as well as the eBWT. The PFP is a preprocessing technique originally introduced to construct the BWT of large and repetitive string collections. We now give a brief overview of the algorithm. With one scan, the PFP divides the input in overlapping substrings of variable length, called *phrases*, which are used to construct what is referred to as the *dictionary* $\mathcal{D}$ and *parse* $\mathcal{P}$ of the input collection. This is done based on a set of *trigger strings* $\mathcal{E}$ of fixed (short) length t: a phrase is defined as a substring starting and ending with some trigger string, and no internal occurrence of any trigger string; therefore, two consecutive phrases in the text will overlap by t characters. Then a separate procedure constructs the BWT of the input directly from $\mathcal{D}$ and $\mathcal{P}$; thus using space proportional to the combined size of these two data structures. The key of the PFP is that if the input collection is repetitive enough, the combined size of $\mathcal{D}$ and $\mathcal{P}$ will be much smaller than the size of the original input. In [8] we presented a variant of the PFP, called *cyclic PFP*, to compute the eBWT of string collections. The cyclic PFP is designed to process circular strings; in particular, it constructs a dictionary that also contains phrases spanning the beginning of an input string.

In [30] Kuhnle et al. showed how to construct the SA, and the SA-samples along with the run-length encoded BWT using the PFP data structures. This is performed by computing for each BWT entry, BWT[i], its corresponding SA value SA[i]. On the other hand, the SA-samples are computed by checking at each step whether BWT[i] $\neq$ BWT[$i-1$]; if this is the case we store both the sample at the end, SA[$i-1$], and at the beginning, SA[i] of the previous and current BWT run respectively. The SA entries are computed by storing $||\mathcal{P}||$ additional integers, one for each phrase, representing the offsets of the last characters of the PFP phrases in the original text. Later in the algorithm, when processing each sorted suffix s of one of the phrases in $\mathcal{D}$, we use the offset of the corresponding phrase and the length of s to output the SA value together with the BWT character.

In this work, we extend our previously described strategy from [8] to compute the GCA-samples rather than the SA-samples. We do this by storing $||\mathcal{P}||$ additional offsets $o = [(j_1, d_1), ..., (j_{||\mathcal{P}||}, d_{||\mathcal{P}||})]$ representing the last positions of the cyclic PFP phrases in $\mathcal{M}$ and implementing an algorithm similar to the one in [30]. Briefly, every time we process a suffix s of some phrase in $\mathcal{D}$, we get the offset of the corresponding phrase (j, d) and use the suffix length to compute the correct GCA-sample $(j - |s|, d)$. The only exception we need to handle is the case where we process a phrase spanning the beginning of a string, i.e., $|s| \geq j$. Here, in order to ensure the correct circular GCA-sample computation, we need to maintain only m additional integers storing the length of each input string. In particular, when processing a suffix s of some phrase in $\mathcal{D}$, if s spans the beginning of a string T_i, we retrieve the string length and compute the correct circular GCA-sample for s. As a result, we compute the run-length eBWT as well as the GCA samples using the augmented cyclic PFP algorithm in time linear in the size of the input collection $\mathcal{M}$ and space linear in the combined size of $\mathcal{D}$ and $\mathcal{P}$.

Finally, given these two data structures, we construct the extended r-index using the procedure described in [20, Appendix A].

As for the RLFM-index, we compute the run heads, the cumulative lengths of the sorted runs, and the content of C and C' while scanning the run-length encoded eBWT in $\mathcal{O}(r)$ time. Moreover, we compute the E and D predecessor search data structures in $\mathcal{O}(r)$ time and $\mathcal{O}(r)$ words of space using up-to-date data structures (see Section 6.1). As for the locate machinery, we sort the GCA samples at the beginning of the runs and store them in F in $\mathcal{O}(sort(r))$ time and $\mathcal{O}(r)$ words of space, where $sort(r)$ is the time required to sort r integers. Since we have already sorted the samples in F in text order, we compute the $FirstToRun$ in $\mathcal{O}(r)$ time by storing the original positions of those samples in F. Note that the values in B are computed during the first pass of the cyclic PFP algorithm by maintaining a counter for the length of the input strings.

Proposition 10. *Given a conjugate-free set of primitive strings $\mathcal{M}$ of total length N, we compute the run-length encoded eBWT and the* GCA*-samples in $\mathcal{O}(N)$ time and $\mathcal{O}(||\mathcal{D}|| + ||\mathcal{P}||)$ words of space, where $\mathcal{D}$ and $\mathcal{P}$ are the dictionary and parse defined by the cylic* PFP*, and construct the extended r-index in $\mathcal{O}(sort(r))$ time and $\mathcal{O}(r)$ words of space.*

Proof. In the previous discussion, we described how to augment the PFP algorithm introduced in [8] to compute the GCA-samples along with the run-length eBWT. This requires, in addition to the PFP data structures, to store $||\mathcal{P}||$ additional phrase offsets and m additional integers for the end-of-string indexes. However, since $\mathcal{P}$ contains at least one phrase for each input string, both are stored in $O(||\mathcal{P}||)$ words of space. In addition, we need to execute $\mathcal{O}(r)$ additional rank and select queries on a bitvector to compute the GCA-samples. Thus, altogether we maintain the same $\mathcal{O}(N)$ time and $\mathcal{O}(||\mathcal{D}|| + ||\mathcal{P}||)$ space complexity of the original algorithm [8]. Finally, it follows from the previous discussion that given the run-length eBWT and the GCA-samples, we construct the extended r-index in $\mathcal{O}(sort(r))$ time and $\mathcal{O}(r)$ space by extending the procedure described in [20, Appendix A]. $\square$

5. Computing MEMs with the extended r-index

The first step for computing MEMs with the extended r-index is to define both matching statistics and MEMs in the context of the eBWT. Given a multiset $\mathcal{M} = \{T_1, \dots, T_m\}$ of input strings and a query string P, the *matching statistics* (MS) of P w.r.t. $\mathcal{M}$ is defined as an array of size $|P|$, with $MS[i] = (\ell, (j, d))$, where ℓ is the length of the longest substring starting at position i in P which occurs in $\mathcal{M}$, and (j, d) is one of its occurrences, i.e. $\text{conj}_j(T_d)[1..\ell] = P[i..i + \ell - 1]$. A substring $P[i..i + \ell - 1]$ of length ℓ is a MEM of P in $\mathcal{M}$ if (1) $P[i..i + \ell - 1]$ occurs in $\mathcal{M}$ and (2) either $i = 1$ or $P[i - 1..i + \ell - 1]$ does not occur in $\mathcal{M}$, and (3) either $i + \ell - 1 = |P|$ or $P[i..i + \ell]$ does not occur in $\mathcal{M}$. Note that in both definitions, *occurrence* stands for circular occurrence.

We now extend Lemma 1 to the eBWT:

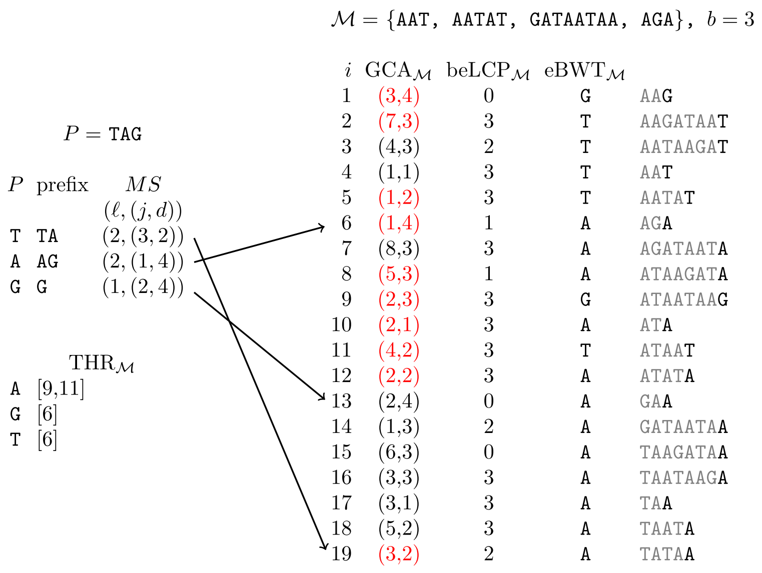
Lemma 11. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N and a query string $P[1..p]$, let $MS[1..p] = [(\ell_1, (j_1, d_1)), \dots, (\ell_p, (j_p, d_p))]$ be the matching statistics of P with respect to $\mathcal{M}$. For all positions $i = 1, \dots, p$, $P[i..i + \ell - 1]$ is a maximal exact match of length ℓ in $\mathcal{M}$ if and only if $\ell_i = \ell$, and either $i = 1$ or $\ell_{i-1} \leq \ell_i$ (in the case $i > 1$).*

Proof. For $P[i..i + \ell - 1]$ to be a MEM, it has to be shown that all of its occurrences in $\mathcal{M}$ are left- and right-maximal. That they are right-maximal, follows from the definition of MS. On the other hand, if an occurrence $i \neq 1$ was not left-maximal then this would imply that $\ell_{i-1} = \ell_i + 1$. $\square$

Next we extend the definition of LCP to the eBWT where the length of query patterns is bounded by the length of the smallest string in the multiset $\mathcal{M}$. To this end, we define the *bounded extended* LCP (beLCP) as an array of length N such that element i contains the length of the longest common prefix of the ith and $(i-1)$st smallest conjugate in $\mathcal{M}$ w.r.t. the ω-order, bounded by the length of the shortest string in $\mathcal{M}$. Formally, let b be the length of the shortest string in $\mathcal{M}$. We set $\text{beLCP}[1] = 0$, and for $i = 2, \dots, N$, if $\text{GCA}[i] = (j_i, d_i)$ then $\text{beLCP}[i] = \min\{b, \texttt{lcp}(\text{conj}_{j_i}(T_{d_i}), \text{conj}_{j_{i-1}}(T_{d_{i-1}}))\}$, where the $\texttt{lcp}$ is w.r.t. the linear strings $\text{conj}_{j_i}(T_{d_i})$ and $\text{conj}_{j_{i-1}}(T_{d_{i-1}})$.

The data structure beLCP can be built with a straightforward generalization of the algorithm of Kasai et al. [27] for LCP-array construction, in time $\mathcal{O}(N)$.

We are now ready to modify the definition of thresholds from Rossi et al. [47] to the extended r-index, substituting the LCP by beLCP. Given a multiset $\mathcal{M} = \{T_1, \dots, T_m\}$ of input strings and four integers $1 \leq j' \leq j < k \leq k' \leq n$ such that $\text{eBWT}(\mathcal{M})[j'..j]$ and $\text{eBWT}(\mathcal{M})[k..k']$ are two consecutive runs of the same character in $\text{eBWT}(\mathcal{M})$. The *threshold* p for the run pair $\text{eBWT}(\mathcal{M})[j'..j]$ and $\text{eBWT}(\mathcal{M})[k..k']$ of character $\text{eBWT}(\mathcal{M})[j]$ is defined as the smallest index i in the interval $\text{beLCP}[j + 1..k]$ such that $\text{beLCP}[i] = \min \text{beLCP}[j + 1..k]$.



$\mathcal{M} = \{$AAT, AATAT, GATAATAA, AGA$\}$, $b = 3$

$P =$ TAG

P	prefix	MS $(\ell,(j,d))$
T	TA	$(2,(3,2))$
A	AG	$(2,(1,4))$
G	G	$(1,(2,4))$

	THR$_{\mathcal{M}}$
A	[9,11]
G	[6]
T	[6]

i	GCA$_{\mathcal{M}}$	beLCP$_{\mathcal{M}}$	eBWT$_{\mathcal{M}}$	
1	(3,4)	0	G	AAG
2	(7,3)	3	T	AAGATAAT
3	(4,3)	2	T	AATAAGAT
4	(1,1)	3	T	AAT
5	(1,2)	3	T	AATAT
6	(1,4)	1	A	AGA
7	(8,3)	3	A	AGATAATA
8	(5,3)	1	A	ATAAGATA
9	(2,3)	3	G	ATAATAAG
10	(2,1)	3	A	ATA
11	(4,2)	3	T	ATAAT
12	(2,2)	3	A	ATATA
13	(2,4)	0	A	GAA
14	(1,3)	2	A	GATAATAA
15	(6,3)	0	A	TAAGATAA
16	(3,3)	3	A	TAATAAGA
17	(3,1)	3	A	TAA
18	(5,2)	3	A	TAATA
19	(3,2)	2	A	TATAA

Fig. 3. An illustration of the thresholds for calculating the matching statistics of a query string $P[1..p]$ in a set of strings $\mathcal{M}$. Shown on the left is P, the longest prefix of the current suffix of P that occurs in $\mathcal{M}$, and the matching statistics MS with their components $(\ell,(j,d))$. Below the pattern P, we show the thresholds THR$_{\mathcal{M}}$ for the characters A, G, and T; e.g., the threshold between the first and the second run of A is in position 9, and the threshold between the second and third run of A is in position 11. Shown on the right, the GCA$_{\mathcal{M}}$ of $\mathcal{M}$ with the entries corresponding to the beginning or to the end of some run highlighted in red, the beLCP$_{\mathcal{M}}$ computed with $b = 3$, the eBWT$_{\mathcal{M}}$, and the corresponding rotations of strings in $\mathcal{M}$. The arrows illustrate the position in GCA$_{\mathcal{M}}$ which corresponds to the prefix on the left. See also Example 8.

Fig. 3 depicts an example of matching statistics query of the pattern $P =$ TAG against the collection of strings $\mathcal{M} = \{$AAT, AATAT, AAGATAAT, AGA$\}$, which is described in the following.

Example 8. We begin the query matching the empty suffix ϵ of the pattern with the prefix of the first conjugate AAG at index 1 of the GCA.

To extend the matching we check whether the eBWT character at index 1 (G) matches the character at index 3 of P (G). In this case the characters match, therefore we perform one LF step to index 13 corresponding to the conjugate GAA and we store the corresponding GCA value (2,4) in the MS array in position 3. We now check if we can extend the match by checking whether the eBWT character at index 13 matches the character at index 2 of P (A). The characters match, therefore we perform one LF step to index 6 corresponding to the conjugate AGA, and we store the corresponding GCA value (1,4) in the MS array in position 2. We now check if we can extend the match by checking whether the eBWT character at index 6 matches the character at index 1 of P (T). The characters do not match. Therefore, we use the thresholds of the character T to determine the index, in GCA, of the conjugate preceded by T sharing the longest prefix with the suffix AG of P. To do this we count the number of runs of T in the eBWT up to index 6, which is 1, and check the threshold between the first and second run of T, which also points to index 6. Since those values are equals, we continue the matching from the index of the beginning of the second run of Ts in the eBWT, which in this case is at index 11 corresponding to the conjugate ATAAT. We perform one LF step from index 11 to index 19 corresponding to the conjugate TATAA and we store the corresponding GCA value (3,2) in the MS array in position 1.

5.1. Computing the thresholds using prefix-free parsing

Rossi et al. in [47] showed how to compute the thresholds efficiently using the PFP-based construction algorithm of the r-index. In this work, we show that such a modification can also be made for the extended r-index, allowing the thresholds for the eBWT to be constructed along with the GCA-samples.

Given the dictionary $\mathcal{D}$ and the parse $\mathcal{P}$ of the cyclic PFP of a multiset $\mathcal{M} = \{T_1, \ldots, T_m\}$ of input strings, in addition to the data structures needed to build the eBWT and the GCA-samples, we build the following data structures:

1) the LCP array of the concatenation of the phrases of the dictionary $\mathcal{D}$ separated by different end-of-string symbols, with an additional data structure supporting range minimum queries; and
2) the beLCP array of the parse $\mathcal{P}$ where the longest common prefixes are measured in terms of original characters, not symbols of $\mathcal{P}$, with trigger strings counted only once. We refer to this data structure as beLCP of $\mathcal{P}$ *associated to* eBWT($\mathcal{M}$). This data structure is also enriched with an additional RMQ-data structure.

Lemma 12. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, and $\mathcal{P}$ and $\mathcal{D}$ its parse and dictionary given by the cyclic PFP. We can build the LCP-array of $\mathcal{D}$ and the* beLCP*-array of $\mathcal{P}$ associated to* eBWT$(\mathcal{M})$ *in $\mathcal{O}(||\mathcal{P}|| + ||\mathcal{D}||)$ time.*

Proof. First, we compute the LCP-array of the string consisting of the concatenated phrases of the dictionary $\mathcal{D}$, separated by distinct separators, using the algorithm of Kasai et al. [27], in $\mathcal{O}(||\mathcal{D}||)$ time. We build on this an RMQ-data structure that allows constant time range-minimum queries. Such a data structure can be constructed in $\mathcal{O}(||\mathcal{D}||)$ time and requires $2||\mathcal{D}|| + o(||\mathcal{D}||)$ bits [18]. This will give us the possibility of giving, for any two distinct suffixes of phrases from $\mathcal{D}$, the length of the longest common prefix, in $\mathcal{O}(1)$ time.

For constructing the beLCP of $\mathcal{P}$, we will use a modification of the Kasai et al. algorithm [27]. The generalization to cyclic strings, as well as from one string to a set of strings, is straightforward. However, additional complexity is added by the fact that the entries of the beLCP contain the length of the LCP in terms of the number of characters in the original strings in $\mathcal{M}$, rather than in terms of the number of meta-characters of $\mathcal{P}$.

In the following, we assume familiarity with the algorithm by Kasai et al. [27], and only sketch our modification. Note that the algorithm fills in the beLCP array in string order, rather than in GCA-order. In our case, it fills the beLCP-values corresponding to each parse string $P \in \mathcal{P}$ individually. The basic idea of the original algorithm is that for any text position i, if LCP[ISA[i]] $= k$ then LCP[ISA[$i+1$]] $\geq k-1$, and therefore, the comparisons for LCP[ISA[$i+1$]] can start at position $i+k$ in the text. As a result, when updating i to $i+1$, if the current value k of the lcp is greater than 0, then it has to be decremented by 1, and further comparisons taken from there. In our modified algorithm, we will carry forward two values: the current length k of the lcp in terms of number of meta-characters, and the current length of the same lcp in terms of characters of the underlying strings in $\mathcal{M}$, which we refer to as κ. We will update both values, and when filling in the corresponding entry, we cut it off at b, i.e. $\text{LCP}_{\mathcal{P}}[j] = \min(b, \kappa)$.

Let P be some conjugate of some parse string in $\mathcal{P}$ with $\text{GCA}_{\mathcal{P}}[j] = P$, and Q the conjugate that immediately precedes P in $\text{GCA}_{\mathcal{P}}$, i.e. $\text{GCA}_{\mathcal{P}}[j-1] = Q$. If $\text{LCP}_{\mathcal{P}}[j] = k$, then we can compute $\text{beLCP}_{\mathcal{P}}[j]$ as follows. First, $\text{LCP}_{\mathcal{P}}[j] = k$ implies that the k-length prefix of P, $P[1..k]$, is common to P and Q; therefore, the corresponding prefix is also common to the underlying strings in $\mathcal{M}$. The length of this prefix is given by the sum of the phrase lengths of $P[1..k]$; however, the overlapping trigger strings, of length t, must be counted only once, yielding $\sum_{\iota=1}^{k}(|P[\iota]| - t)$ as total length of this prefix. Further, if $|P|, |Q| \geq k$, then we also need to add the lcp of the subsequent phrases $P[k+1]$ and $Q[k+1]$. Let $P[k+1] = D[i_1]$ and $Q[k+1] = D[i_2]$, and let $d = \texttt{lcp}(D[i_1], D[i_2])$. Observe that $d \geq t$, since these two phrases necessarily start with the same trigger string. Note further that d can be computed in $\mathcal{O}(1)$ time using the RMQ-data structure on $\text{LCP}_{\mathcal{D}}$. The total length of the lcp of the string conjugates in $\mathcal{M}$ corresponding to P and Q is therefore $\kappa = \sum_{\iota=1}^{k}(|P[\iota]| - t) + d$, and we get $\text{beLCP}_{\mathcal{P}}[j] = \min(b, \kappa)$.

Now when passing to the next conjugate $P' = P[2..|P|]P[1]$, we need to decrement the value κ by d, the lcp of the first mismatching character pair $P[k+1], Q[k+1]$. If $k \geq 1$, we decrement κ further by the length of the first meta-character, $|P[1]| - t$; in this case, k is decremented by 1. Both k and κ are then incremented according to the possible new matching meta-characters between the following characters of P' and those of the immediately preceding conjugate in the GCA of $\mathcal{P}$. Finally, we build an RMQ-data structure on $\text{beLCP}_{\mathcal{P}}$ in time $\mathcal{O}(||\mathcal{P}||)$.

By amortized analysis it can be seen that the total time taken is $\mathcal{O}(||\mathcal{P}||)$ for computing $\text{beLCP}_{\mathcal{P}}$, since each value $|P[i]|$, corresponding to the character $P[i]$, is only added once. Therefore, for computing both data structures, we get altogether $\mathcal{O}(||\mathcal{D}|| + ||\mathcal{P}||)$ time, as claimed. □

We now show how to use the data structures of Lemma 12 to compute the thresholds. We build on the algorithm in Section 4. The high-level idea is that given the multiset $\mathcal{M} = \{T_1, \ldots, T_m\}$ and its cyclic PFP parse $\mathcal{P}$ and dictionary $\mathcal{D}$, we can iterate sequentially through the values of eBWT[i], GCA[i], and beLCP[i], for $i = 1, \ldots, N$. We already know from Section 3 how to compute the first two. We will now show how to compute also the beLCP-array (of $\mathcal{M}$) using the data structures 1) and 2).

We recall that each conjugate in GCA can be uniquely identified by a proper phrase suffix in $\mathcal{D}$ and the conjugate in $\mathcal{P}$ starting in correspondence to the first complete phrase following the proper phrase suffix in $\mathcal{D}$ of the conjugate.

When iterating through the conjugates in GCA order using $\mathcal{P}$ and $\mathcal{D}$, in order to compute beLCP[i] we need to compare two consecutive conjugates in GCA order, i.e., GCA[$i-1$] and GCA[i]. We need to consider two cases. In the first case, the conjugates corresponding to GCA[$i-1$] and GCA[i] are identified by different proper phrase suffixes in $\mathcal{D}$. Therefore, beLCP[i] is given by the length of the longest common prefix between the two proper phrase suffixes in $\mathcal{D}$ (bounded by b). This can be computed with an RMQ-query in $\mathcal{O}(1)$ time using the data structure 1). In the second case, the conjugates corresponding to GCA[$i-1$] and GCA[i] are identified by the same proper phrase suffix in $\mathcal{D}$, therefore their conjugates in $\mathcal{P}$ will be different. In this case, we can compute beLCP[i] as the length of the common proper phrase suffixes in $\mathcal{D}$ plus the longest common prefix of the conjugates in $\mathcal{P}$, in terms of characters of the underlying strings in $\mathcal{M}$, bounded by b. This can be computed with an RMQ-query in $\mathcal{O}(1)$ time using the data structure 2).[2]

[2] Note that this computation needs to be adjusted to take into account the overlap between phrases that is introduced by the prefix-free parsing. See Rossi et al. [46] for additional details.

Proposition 13. *Let $\mathcal{M}$ be a conjugate-free set of primitive strings of total length N, we can build the thresholds in addition to the extended r-index in $\mathcal{O}(N)$ time and $\mathcal{O}(||\mathcal{D}|| + ||\mathcal{P}||)$ space, where $\mathcal{D}$ and $\mathcal{P}$ are the dictionary and parse defined by the* PFP.

Proof. In the previous discussion, we described how to extend the algorithm presented in Section 4 to compute the entries of beLCP of $\mathcal{M}$ alongside the eBWT and GCA. To compute the thresholds, we need to store, for each character of the alphabet, the minimum value of the beLCP computed from the last occurrence of the character run in the eBWT, and reporting it when we encounter a new run of the same character. Therefore, computing the thresholds requires additional $\mathcal{O}(N\sigma)$ time and $\mathcal{O}(\sigma)$ space, in addition to the computation of the runs of the eBWT and the GCA-samples. Under the assumption of constant size alphabet, we can thus build the thresholds in addition to the extended r-index in $\mathcal{O}(N)$ time and $\mathcal{O}(||\mathcal{D}|| + ||\mathcal{P}||)$ space. □

6. Experimental results

We implemented the extended r-index in C++ by adapting the code of the r-index (https://github.com/nicolaprezza/r-index), and made it available at https://github.com/davidecenzato/extended_r-index. We compute all data structures necessary to construct the extended r-index via an adaptation of `pfpebwt` [8] to enable our construction algorithm to scale to large string collections. We compare the performance of our text index with the r-index and other four available text indexes for highly repetitive biological data. Note that, unlike the r-index and the other indexes, our implementation supports circular pattern matching, i.e., we report occurrences that span the end and beginning of a string in addition to linear occurrences.

6.1. Implementation

We now describe how we implemented the data structures described in Section 3.1. We construct the eBWT RLFM-index that computes the GCA-interval $[i..j]$ for a pattern P by using a similar implementation as the one described by Gagie et al. in [20, Section 7.1]. We compute L by constructing the wavelet tree of the characters in the run-length encoded eBWT using the Huffman-encoded wavelet tree implementation of `sdsl` (`wt_huff`) supporting rank and select queries. We compute E and D by encoding the run heads and the cumulative lengths using the unary encoding and storing the final binary strings in two `sdsl` Elias-Fano compressed bitvectors (`sd_vector`) that support rank and select queries. We do not store C' explicitly since we split D in σ different substrings, enabling direct access to D. Finally, we store the C vector using the `sdsl` (`int_vector`) implementation. Altogether, these data structures take $(1+\epsilon)r(\log(N/r)+2) + (rH_0 + 2\sigma \log r) + \sigma \log N$ bits of space (lower-order terms omitted), where $\epsilon > 0$ is a constant defined at construction time ($\epsilon = 0.5$ in our case), N denotes the total length of the input collection, and H_0 is the zero-order entropy.[3]

We extend the locate machinery described by Gagie et al. [20] to support ϕ operations on the GCA rather than on the SA. We construct G, the data structure that stores the GCA-samples at the end of the runs, as well as the $FirstToRun$ vector using the `sdsl` (`int_vector`) implementation [24]. Further, we compute F by storing the GCA-samples at the beginning of a run using a gap-encoded bitvector. In particular, we construct a bitvector $B_f[1..N]$ for F such that the ith entry $B_f[i] = 1$ if $i \in F$. We construct the B data structure that we additionally need in the extended r-index in a similar way. More specifically, we define a bitvector $B_b[1..N]$ for B such that the ith entry $B_b[i] = 1$ if i is the position of the first character of one of the input strings. In our implementation, we store the input collection $\mathcal{M} = \{T_1, \ldots, T_m\}$ as the concatenation of the input strings $T_1 \cdots T_m$. Thus, the indexes encoded by B_f refer to positions in the concatenation, and B_b stores the string boundaries in the concatenation. We implement both B_f and B_b using the Elias-Fano compressed bitvector of `sdsl` (`sd_vector`) implementation. Altogether these data structures implementing the locate machinery take $r \log N + r \log r + r(\log(N/r)+2) + m(\log(N/m)+2)$ bits of space (lower-order terms omitted).

6.2. Handling equal conjugates

Proposition 3 ensures that an input collection $\mathcal{M}$ where no two strings have the same set of conjugates, has at least two GCA-samples for each string, one at the beginning and one at the end of an eBWT run. However, when working with string collections containing circular genomes of the same species, this condition may not be fulfilled since we may find two sequences such that one is a rotation of the other. In this case, we will have some sequences with no GCA-samples. We solve this problem by sampling some additional GCA values corresponding to the first rotation of each string in $\mathcal{M}$. We added this functionality to the algorithm described in Section 4: while computing the eBWT, every time we process the first rotation of a string, we check whether the corresponding character eBWT$[i]$ is the first character of a new run. If not, we instantiate a new run and store two GCA-samples, one at the beginning of the new run GCA$[i]$, and one at the end of the previous run GCA$[i-1]$. At the end of this procedure, we will have sampled at most $2m$ additional GCA-samples.

[3] The Huffman-shaped wavelet tree, first introduced in [35], in the `sdsl` implementation uses $(rH_0 + 2\sigma \log r)$ bits of space, see github.com/simongog/sdsl-lite/blob/master/include/sdsl/wt_huff.hpp.

We set this functionality as our software's default mode since it allows our index to handle any genomic collection containing primitive strings, thus meeting the majority of real scenarios. In addition, the space required to store and process the additional GCA-samples is negligible when working with whole long genome sequences. We ran all our experiments using this default mode. We allow deactivating this functionality using a flag if the user knows that no input string is a conjugate of another.

Example 9. In Fig. 2, assume we want to sample the GCA positions corresponding to the first rotation of each string in $\mathcal{M}$. In this case, we also need to store $(1,1)$, $(1,2)$ and $(1,3)$, which are the samples corresponding to the `AAT`, `AATAT` and `GATAATAA` rotation, respectively. Note that $(1,4)$ was already at the beginning of the third eBWT run, thus, we do not need to sample it again.

6.3. Handling non-primitive strings

As for handling non-primitive strings, we need two additional functionalities: (i) computing the GCA and eBWT of string collections containing non-primitive strings and (ii) supporting $\phi_{\mathcal{M}}$ operations on such a GCA. We obtain (i) by modifying the PFP algorithm as described in [7]; this includes running the algorithm on the roots $\{root(T_1), \ldots, root(T_m)\}$ of the strings in $\mathcal{M}$ and using the exponents $\{exp(T_1), \ldots, exp(T_m)\}$ to output the correct eBWT and GCA-samples. Without loss of generality we can assume that the collection of the roots is a conjugate-free set. Otherwise, we use the strategy described in the previous subsection, where strings with conjugate roots are considered according to an order given by the exponents. As for (ii) we implement it again by using the exponents and root lengths; in particular, given $k_d = exp(T_d)$ and $r_d = |root(T_d)|$, when computing $\phi_{\mathcal{M}}(\text{GCA}[i])$, where $\text{GCA}[i] = (j, d)$, if $j > r_d$ then the query will report $(j - r_d, d)$ as a result, otherwise we apply the procedure described in Subsection 3.1 on the set $\mathcal{M}' = \{root(T_1), \ldots, root(T_m)\}$ to get $\phi_{\mathcal{M}'}((j, d)) = (j', d')$. If $k_{d'} = 1$, then $\phi_{\mathcal{M}}((j, d)) = (j', d')$. Otherwise, if $k_{d'} > 1$, we additionally set $j' = j' + r_{d'} \cdot (k_{d'} - 1)$.

Example 10. Let $\mathcal{M} = \{\texttt{ACGACG}, \texttt{GCGCGC}\}$ be a string collection containing two non-primitive strings, $\text{GCA}(\mathcal{M}) = [(1,1), (4,1), (2,1), (5,1), (2,2), (4,2), (6,2), (3,1), (6,1), (1,2), (3,2), (5,2)]$, $r_1 = 3$, $k_1 = 2$ and $r_2 = 2$, $k_2 = 3$. We evaluate $\phi_{\mathcal{M}}((5,2))$ and $\phi_{\mathcal{M}}((1,2))$ as follows. Since $5 > r_2$, the answer is $\phi_{\mathcal{M}}((5,2)) = (5 - r_2, 2) = (3,2)$. As for the second query, $1 < r_2$, thus we need to evaluate $\phi_{\mathcal{M}}((1,2))$ by using the procedure described in Subsection 3.1 on the set of the roots $\{\texttt{ACG}, \texttt{GC}\}$. The result is $(3,1)$. Since $k_1 > 1$, we finally obtain $\phi_{\mathcal{M}}((1,2)) = (3 + r_1(k_1 - 1), 1) = (6,1)$.

6.4. Text indexes tested

We included in the comparison six text indexes using different strategies to encode repetitiveness in biological data.

- ***extended*** ***r*-*index:*** Our text index, constructed using a variant of the `pfpebwt` algorithm and sampling the beginning of the eBWT runs.[4]
- ***r*-*index:*** The r-index implementation using the `Big-BWT` algorithm [6] to enable the index construction for large datasets.[5]
- ***subsampled*** ***r*-*index:*** The subsampled r-index implementation (sr-index) by Cobas et al. [14].[6] We computed it using the sampling value $s = 64$.
- ***lz-index:*** An optimized implementation of the Lempel-Ziv based text index by Kreft and Navarro [28].[7]
- ***hybrid-index:*** The `CHICO` implementation [52],[8] which improves the original hybrid index by Ferrada et al. [16]. We computed it by setting the M parameter to the three pattern lengths.
- ***grammar-index:*** The grammar-based text index implementation by Claude et al. [13].[9] We computed it with Patricia trees sampling value $s = 64$.

6.5. Datasets

We evaluated the extended r-index and the other indexes using three genomic datasets: the first one contains circular assembled genomes of *Salmonella enterica* (`Salmonella`); the second one contains circular assembled genomes of *Escherichia coli* (`E.coli`) strains; the last dataset contains a set of plasmid (`Plasmids`) genomes. We downloaded the `Salmonella` and `E.coli` datasets from NCBI using the accession ids of the reference genomes from the ZymoBIOMICS

[4] https://github.com/davidecenzato/extended_r-index.
[5] https://github.com/alshai/r-index.
[6] https://github.com/duscob/sr-index.
[7] https://github.com/migumar2/uiHRDC/tree/master/uiHRDC/self-indexes/LZ.
[8] https://www.cs.helsinki.fi/u/dvalenzu/software.
[9] https://github.com/apachecom/grammar_improved_index.

Table 1
Table summarizing the main parameters of the three datasets. From left to right, we report the dataset name, the number of sequences, the total length, the average, minimum, and maximum sequence length, and the average run-length of the eBWT (n/r).

dataset	no. seq	total length	avg. length	min. length	max. length	n/r (eBWT)
`Salmonella`	846	4,121,587,394	4,871,853	4,482,093	5,700,307	70.574
`E.coli`	1,362	7,024,773,608	5,157,690	4,456,672	6,162,417	88.011
`Plasmids`	25,916	3,458,859,947	133,464	10,005	4,605,385	4.220

High-Molecular-Weight DNA Mock Microbial community used by Ahmed et al. [1]. We downloaded the most recent assemblies of this data, and removed the ones marked as anomalous. In addition, we only kept the reference genomes and removed additional contigs. This resulted in 846 *Salmonella enterica* and 1,362 *Escherichia coli* assembled genomes. As for `Plasmids`, we downloaded the sequences from the PLSDB database containing a collection of plasmid sequences gathered from NCBI and INSDC [48]. Since our implementation does not include the functionalities described in Section 6.3, we removed non-primitive strings, which resulted in 25,916 genomic sequences. Features of the three datasets are summarized in Table 1.

Some of the competing methods are not designed to work on large datasets. Thus, for each dataset, we derived a smaller string collection to make it possible to evaluate a broader range of methods. This resulted in 210 *Salmonella enterica*, 200 *Escherichia coli*, and 8,000 plasmid assembled genomes.

6.6. Experimental setup

We performed the experiments on a server with Intel(R) Core(TM) CPU i9-11900 @ 2.50GHz with 8 cores and 64 Gigabytes of RAM running Ubuntu 22.04 LTS 64-bit. The compiler was g++ version 11.3.0 using C++ 17 standard and `-Ofast -fstrict-aliasing -march=native -DNDEBUG` options. We recorded the memory usage using the maximum resident set size retrieved from `/usr/bin/time`.

From each dataset, we extracted four pattern sets containing 1 million sequences each, of length 10, 100, $1,000$, and $10,000$, respectively. Patterns were extracted by randomly sampling substrings from the input strings. In particular, for each pattern length p, we select a string $T_d \in \mathcal{M}$ at random, select a random offset $1 \le i \le |T_d|$, and extract the circular substring beginning at the i-th position, $T_d[i..(i+p-1) \bmod |T_d|]$. Notice that this procedure samples both patterns occurring as linear substrings and patterns spanning the end and the beginning of a string. For each set of patterns, we computed both count and locate queries on all patterns and collected the number of occurrences, the total memory usage, and the average time to process a pattern. Note that only the three indexes based on the BWT naturally support count queries; in particular, they allow computing the number of occurrences of a pattern in the input without locating explicitly where it occurs.

In our experiments, we built and queried the extended r-index and the five competing text indexes. In order to make our experimental results comparable, we needed all indexes to compute the same circular count and locate queries supported by our index. Given a collection of strings $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$, we fulfilled this requirement by constructing the five competing indexes for the two strings $S = T_1\$ \cdots T_m\$$, and $S' = T_1T_1\$ \cdots T_mT_m\$$. This allowed searching for circular patterns with minimal implementation effort and without changing the software specifications.

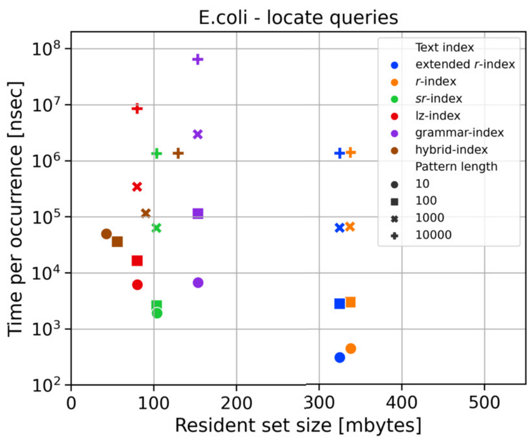
Next we explain how to answer circular pattern matching queries with the competing indexes. Given a pattern P, circular count queries are performed by computing a count query on the indexes of S', which return twice the number of linear occurrences of P in $\mathcal{M}$, and once the number of occurrences of P that span the end and the beginning of some string in $\mathcal{M}$; then we subtract from this value the result of a count query on the indexes of S. Thus, every count query computed by one of the competing indexes consists of two count queries. Finally, the locate queries are computed as a locate query on the indexes of S', where all occurrences located in the second copy of some string are discarded.

6.7. Results

We first report the running time and space comparison with competing indexes, on reduced versions of the three datasets (see Sec. 6.5). This is followed by an analysis of the number of linear and circular occurrences of patterns on the complete datasets.

6.7.1. Comparison with competing indexes using reduced datasets

We start the experimental comparison between our index and the five competing indexes by discussing their running times. In Fig. 4, 5 and 6, we illustrate the time and the memory usage to perform locate queries on the small versions of `Salmonella`, `E.coli`, and `Plamids` datasets and a subset of $100,000$ patterns, as explained at the end of Sec. 6.5. In particular, we report the resident set size on the x-axis and the average time to process a pattern occurrence on the y-axis. We note that the differences are largest for the shortest pattern length, where the extended r-index is always faster than the other indexes. In particular, for $p = 10$, our index has a 160x maximum and 1.4x minimum speedup on `E.coli` dataset, a 137x maximum and 1.4x minimum speedup on `Salmonella` dataset, and a 536x maximum and 1.6x minimum speedup on `Plasmids` dataset, compared to hybrid-index and r-index respectively. On the other hand, our index always uses more



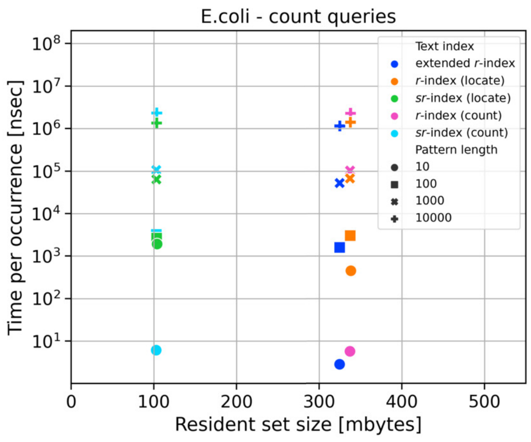


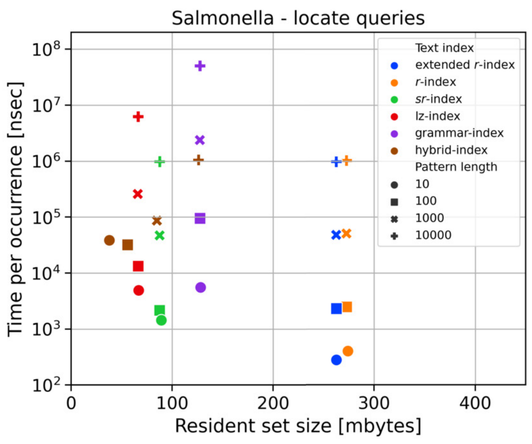
Fig. 4. Time and space to perform locate and count queries on `E.coli` small datasets. Each point in the figures represents a pair consisting of the average time to find a pattern occurrence and the peak memory usage of one of the indexes.

memory than the competitors except for the *r*-index. These performances for short patterns are justified by considering that the competing indexes, unlike our index, have to find and filter a lot of duplicated occurrences, thus slowing down the computation.

For the other pattern lengths $p \geq 100$, there are more differences to take into consideration. As for the BWT-based indexes, the runtime and memory requirements for the extended *r*-index are similar to the *r*-index for all datasets and pattern lengths. The extended *r*-index was slightly faster for locate queries on `Salmonella` and `E.coli`, with a maximum 1.10x speedup factor. In contrast, the extended *r*-index was slightly slower for $p = 10,000$ on the `Plasmids` dataset. The same holds for memory usage, where the extended *r*-index uses at most 1.04x less memory than the *r*-index. As for the *sr*-index, it always uses between 3x to 6x less memory and has similar running times as our index. As for the other three indexes, the grammar-index and lz-index are always slower but more memory efficient than our index. In particular, the extended *r*-index uses between 3x and 5x more memory; however, it is always between 5x and 58x faster than the two indexes. Finally, the hybrid-index is always slower and more memory efficient than our index for all p values except the largest one, where the running times of the two indexes are similar. This suggests that lz- and grammar-based indexes are more memory efficient than our method, however, they are much slower in most scenarios.

We also measured the running time to perform circular count queries with the BWT-based indexes. Since the competing indexes need to run two queries for each circular count query, the extended *r*-index is always faster than the competitors with a 2.2x maximum speedup. In addition, we realized that in all cases except for $p = 10$, it is more efficient to implement the circular count query in the competitor tools by performing one locate query on S' rather than perform count queries on S and S', as explained at the end of Sec. 6.6. That's why we also report, for the *r*-index and the *sr*-index, the average running time for both types of query, locate and count.

In summary, we showed that the extended *r*-index has better or similar running times compared to the other indexes for all combinations of dataset and pattern length. Notwithstanding this performance, our index uses more memory than the competing indexes except for the original *r*-index. However, we noticed that by adapting the results of the subsampled *r*-index to the eBWT, we may be able to obtain an index that is both fast and very memory efficient. Finally, the extended



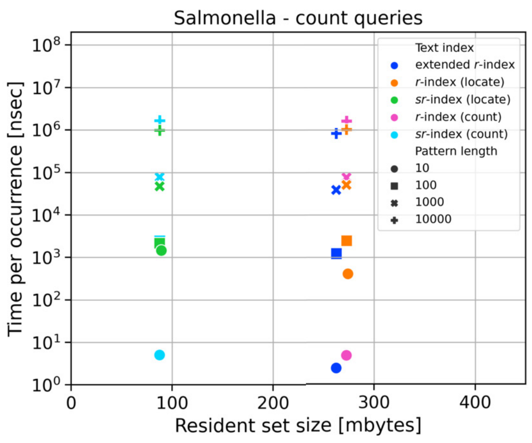


Fig. 5. Time and space to perform locate and count queries on `Salmonella` small dataset. Each point in the figures represents a pair consisting of the average time to find a pattern occurrence and the peak memory usage of one of the indexes.

r-index can handle large datasets and is also simpler to construct than the competitors since we do not need to double the size of the dataset.

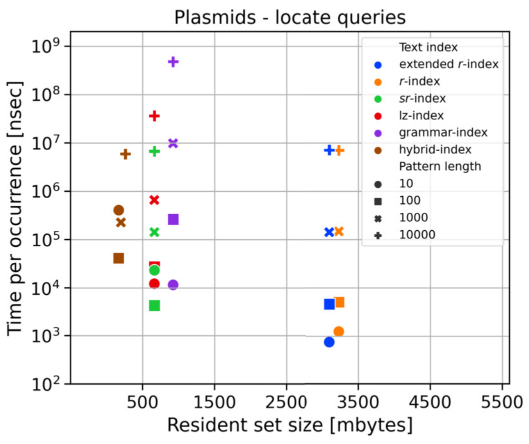
6.7.2. *Analysis of the number of occurrences*

In this section, we analyze the differences between circular and linear pattern matching queries. For each complete dataset of `Salmonella`, `E. coli`, and `Plasmids`, we ran count queries for three pattern sets, each containing 1 million randomly extracted substrings of length 100, 1, 000, and 10, 000. For each pattern, we recorded the number of linear and circular (or cyclic) occurrences.

The difference in the number of linear and cyclic occurrences depends on two factors: the lengths of the indexed sequences and the lengths of the patterns. With long indexed sequences and short patterns, it is less likely to extract a pattern that spans the beginning of a string; thus, most patterns have the same number of linear and cyclic occurrences. On `Salmonella`, and `E.coli` datasets, only 0.02%, 0.04%, 0.30% and 0.04%, 0.06%, 0.27% of the patterns show a different number of linear and cyclic occurrences for the 100, 1,000, and 10,000 pattern length, respectively. On the other hand, when the dataset contains short sequences, we are more likely to extract patterns that span the beginning of a string. On the `Plasmids` dataset, 3.59%, 9.44%, and 24.77% of the patterns show a different number of linear vs. cyclic occurrences for the three pattern lengths, respectively (see Fig. 7).

We continue this analysis by studying the difference in the number of occurrences, only for those patterns where linear and circular occurrence numbers do not coincide, i.e. the patterns spanning the beginning of a string. In Fig. 8, we show a dispersion plot where each dot corresponds to the number of linear and cyclic occurrences of one pattern: the farther the dots are from the main diagonal, the higher the difference. Again, we notice the largest variations for the longest pattern length and `Plasmids`, which contains the shortest sequences among the three datasets.

We now report a more in-depth analysis of the number of matches we lose when using the r-index on circular strings, on which *all* cyclic occurrences should be reported. Given a string P, we defined the percentage of matches lost as $(1 - c(P)/c'(P)) \cdot 100$, where $c(P)$ and $c'(P)$ are the number of cyclic and linear occurrences of P in the input, respectively. Thus, we analyze the average number of matches lost on patterns for which $c(P) \neq c'(P)$. For patterns of length 10,000, we lose



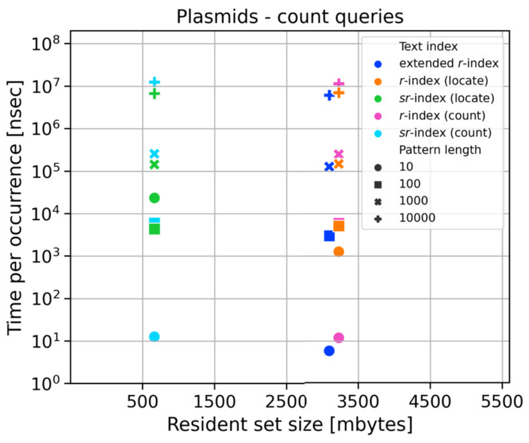


Fig. 6. Time and space to perform locate and count queries on `Plasmids` small dataset. Each point in the figures represents a pair consisting of the average time to find a pattern occurrence and the peak memory usage of one of the indexes.

on average more than 50% of its matches when using the *r*-index. In particular, we report 68.0%, 71.9%, and 78.4% average matches lost for the `Salmonella`, `E.coli`, and `Plasmids` dataset, respectively. For patterns of length 1,000, the average match loss is 42.5%, 32.7%, and 20.0% for the three datasets. Finally, the reported average match loss is 6.0%, 3.7%, and 5.1% on patterns of length 100. In Fig. 9, we summarize these results showing a histogram that reports the number of patterns associated with each percentage of matches lost.

7. Conclusion

In this paper, we described how the fundamentals of the *r*-index can be transferred to the context of the eBWT. We note that the eBWT has the advantage over other BWT-based data structures for string collections that it is independent of the order of the input strings. The *r*-index based on the eBWT, which we call *extended r-index*, inherits this important property. We showed that it presents the most convenient framework for circular strings, among the different options present in the literature and in existing implementations. To substantiate this claim, we compared our extended *r*-index to five state-of-the-art text indexes, including the classical *r*-index and several grammar-based indexes. We tested the performance of these indexes on three biological data sets containing circular bacterial genomes and plasmids.

Applied as given, all competitor indexes would miss cyclic occurrences that span the end of the input strings; but constructing the indexes on two copies of each input string introduces unnecessary increase in space and post-processing work. For this reason, when the input collection consists of circular strings, the extended *r*-index should be chosen rather than the original *r*-index, given the very similar performance both in space and query time. On the other hand, grammar-based compressed self-indexes are superior to our data structure with respect to space usage but they are slow and do not scale well for big collections. We note that the subsampled *r*-index, a variant of the original *r*-index, uses an amount of memory similar to the LZ-index, grammar-index, and hybrid-index. Due to this, extending the improvements of the subsampled *r*-index to the eBWT could improve the space requirements of our index and close the memory gap with the dictionary-based indexes.

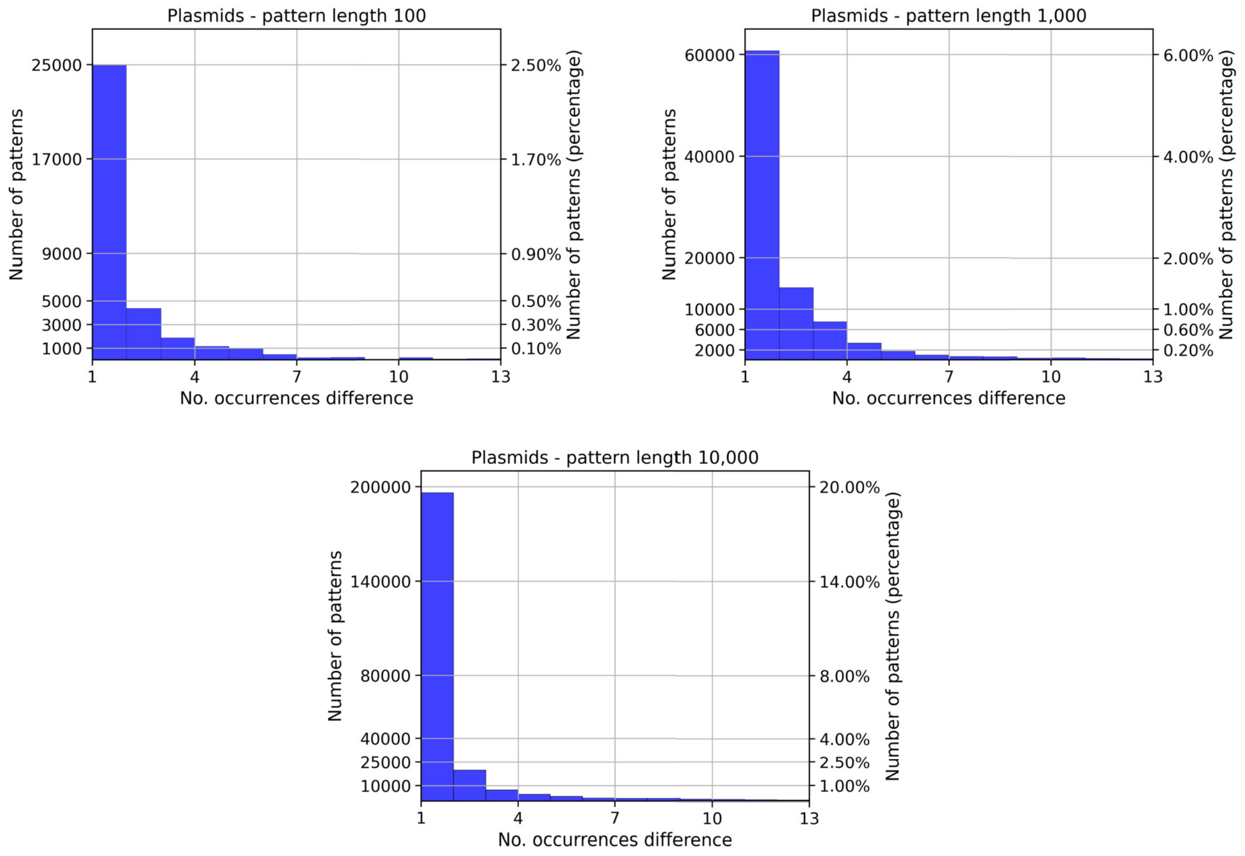


Fig. 7. Histograms summarizing the difference between the number of cyclic and linear occurrences on the `Plasmids` dataset and three sets of patterns of different lengths, containing 1 million sequences each. Each bar reports the percentage of patterns with a given difference. For ease of presentation, we cut the x-axis at 13 since all other bars have a value on the x-axis below 1,000. We omitted the first bin containing the number of patterns whose difference is zero (the percentage of patterns included is reported in the text).

Note also that the use of the extended r-index is not restricted to circular strings; it can be applied to string collections of linear strings, by simply appending an end-of-string character to each input string.

Throughout we assumed that the pattern length is bounded by the length of the shortest string in $\mathcal{M}$. This is to avoid finding occurrences which are substrings of ω-powers of input strings, but not of the strings themselves. We pose it as an open problem to give efficient locate and count algorithms for the extended r-index that do not necessitate this assumption. We note further that recasting some of the more recent results—including the results of Nishimoto and Tabei [42], Bannai et al. [3], and Cobas et al. [14]—regarding the r-index to the context of eBWT merits attention since it would allow to further improve time and space requirements of our extended r-index.

CRediT authorship contribution statement

Christina Boucher: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Davide Cenzato:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Zsuzsanna Lipták:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Massimiliano Rossi:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Marinella Sciortino:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

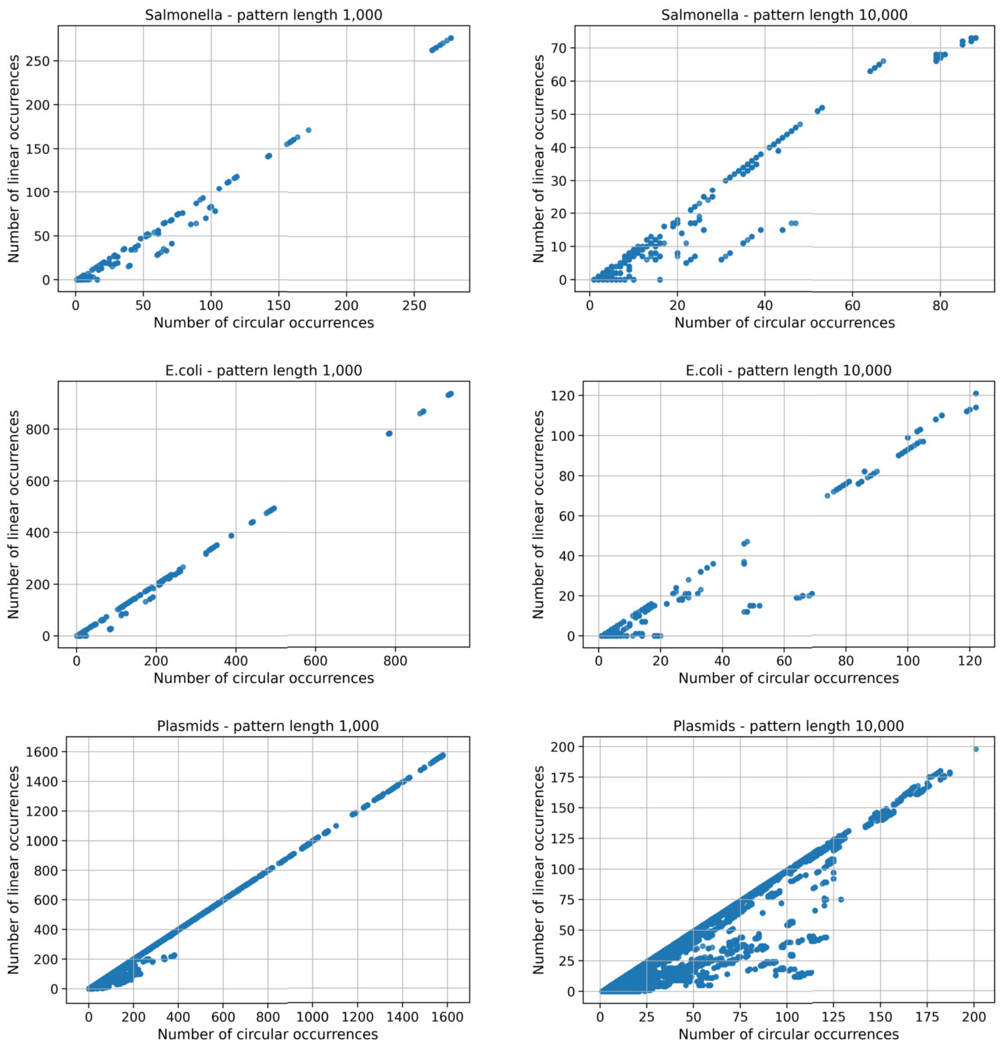


Fig. 8. Dispersion plots between the number of linear vs. cyclic occurrences on three sets of patterns of different lengths, containing 1 million sequences each. Each point in the figures represents a specific pattern. We only included the patterns having different number of cyclic and linear occurrences (the percentage of patterns included is reported in the text).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank Enno Ohlebusch and two anonymous reviewers for their insightful comments, and Dustin Cobas for providing a working version of the grammar index tool. CB and MR are funded by National Science Foundation NSF IIBR (Grant No. 2029552), NSF SCH (Grant No. 2013998), National Institutes of Health (NIH) NIAID (Grant No. HG011392) and NIH NIAID (Grant No. R01AI141810). ZsL and MS are partially funded by the MUR PRIN Project “PINC, Pangenome INformatiCs: from Theory to Applications” (Grant No. 2022YRB97K). MS is partially funded by the Istituto Nazionale di Alta Matematica - Gruppo Nazionale Calcolo Scientifico (INdAM-GNCS) Project (CUP_E53c22001930001), and ZsL is member of INdAM-GNCS. DC is funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

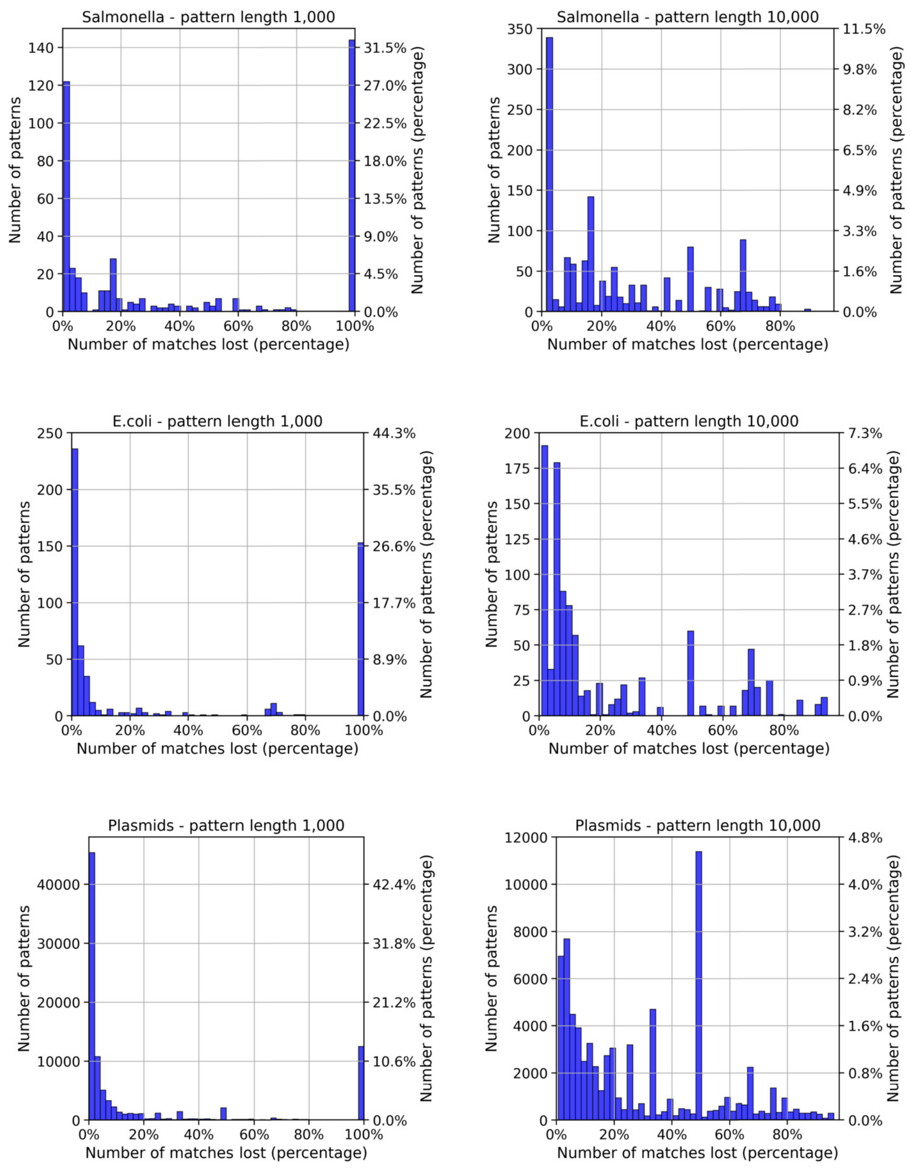


Fig. 9. Histograms summarizing the number of matches lost between the extended *r*-index and the *r*-index on the three datasets. Each bar shows the percentage of the number of patterns with a given percentage of matches lost. For this plot, we only include the patterns whose number of occurrences is different between the two indexes. We omitted the 100% bins containing the patterns with zero matches using the *r*-index for length 10,000. The heights of these bins are 63.9% for `E.coli`, 56.4% for `Salmonella` and 69.6% for `Plasmids`.

References

[1] Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C. Schatz, Travis Gagie, Christina Boucher, Ben Langmead, Pan-genomic matching statistics for targeted Nanopore sequencing, iScience 24 (6) (2021) 102696.

[2] Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, Marcin Piatkowski, Indexing the bijective BWT, in: 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, in: LIPIcs, vol. 128, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 17:1–17:14.

[3] Hideo Bannai, Travis Gagie, I. Tomohiro, Refining the r-index, Theor. Comput. Sci. 812 (2020) 96–108.

[4] Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, Marcin Piatkowski, Constructing the bijective and the extended Burrows-Wheeler transform in linear time, in: Proc. of the 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, in: LIPIcs, vol. 191, 2021, pp. 7:1–7:16.

[5] Djamal Belazzougui, Gonzalo Navarro, Optimal lower and upper bounds for representing sequences, ACM Trans. Algorithms 11 (4) (2015) 31:1–31:21.

[6] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, Taher Mun, Prefix-free parsing for building big BWTs, Algorithms Mol. Biol. 14 (1) (2019) 13:1–13:15.

[7] Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, Marinella Sciortino, r-Indexing the eBWT, in: Proc. of the 28th International Symposium on String Processing and Information Retrieval, SPIRE 2021, in: LNCS, vol. 12944, 2021, pp. 3–12.

[8] Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, Marinella Sciortino, Computing the original eBWT faster, simpler, and with less memory, in: Proc. of the 28th International Symposium on String Processing and Information Retrieval, SPIRE 2021, in: LNCS, vol. 12944, 2021, pp. 129–142.

[9] Michael Burrows, David J. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.
[10] Davide Cenzato, Zsuzsanna Lipták, A theoretical and experimental analysis of BWT variants for string collections, in: Proc. of the 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, in: LIPIcs, vol. 223, 2022, pp. 25:1–25:18.
[11] Davide Cenzato, Zsuzsanna Lipták, A survey of BWT variants for string collections, Submitted.
[12] Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, Giovanna Rosone, Computing the optimal BWT of very large string collections, in: Proc. of the 33rd Data Compression Conference, DCC 2023, 2023, pp. 71–80.
[13] Francisco Claude, Gonzalo Navarro, Alejandro Pacheco, Grammar-compressed indexes with logarithmic search time, J. Comput. Syst. Sci. 118 (2021) 53–74.
[14] Dustin Cobas, Travis Gagie, Gonzalo Navarro, A fast and small subsampled R-index, in: Proc. of the 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, in: LIPIcs, vol. 191, 2021, pp. 13:1–13:16.
[15] Jason Lloyd-Price, et al., Strains, functions and dynamics in the expanded human microbiome project, Nature 550 (7674) (2017) 61–66, https://doi.org/10.1038/nature23889.
[16] Héctor Ferrada, Dominik Kempa, Simon J. Puglisi, Hybrid indexing revisited, in: Proc. of the 20th Workshop on Algorithm Engineering and Experiments, ALENEX 2018, 2018, pp. 1–8.
[17] Paolo Ferragina, Giovanni Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.
[18] Johannes Fischer, Volker Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[19] Travis Gagie, Gonzalo Navarro, Nicola Prezza, Optimal-time text indexing in BWT-runs bounded space, in: Proc. of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, 2018, pp. 1459–1477.
[20] Travis Gagie, Gonzalo Navarro, Nicola Prezza, Fully functional suffix trees and optimal text searching in BWT-runs bounded space, J. ACM 67 (1) (2020) 2:1–2:54.
[21] Ira M. Gessel, Christophe Reutenauer, Counting permutations with given cycle structure and descent set, J. Comb. Theory, Ser. A 64 (2) (1993) 189–215.
[22] Raffaele Giancarlo, Antonio Restivo, Marinella Sciortino, From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization, Theor. Comput. Sci. 387 (3) (2007) 236–248.
[23] Joseph Yossi Gil, David Allen Scott, A bijective string sorting transform, CoRR, arXiv:1201.3077 [abs], 2012.
[24] Simon Gog, Timo Beller, Alistair Moffat, Matthias Petri, From theory to practice: plug and play with succinct data structures, in: 13th International Symposium on Experimental Algorithms, (SEA), 2014, pp. 326–337.
[25] Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, Sharma V. Thankachan, Efficient algorithm for circular Burrows-Wheeler transform, in: Proc. of the 23rd Annual Symposium on Combinatorial Pattern Matching, CPM 2012, in: LNCS, vol. 7354, 2012, pp. 257–268.
[26] Juha Kärkkäinen, Giovanni Manzini, Simon J. Puglisi, Permuted longest-common-prefix array, in: Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching CPM 2009, in: LNCS, vol. 5577, Springer, 2009, pp. 181–192.
[27] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, Kunsoo Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001, in: LNCS, vol. 2089, 2001, pp. 181–192.
[28] Sebastian Kreft, Gonzalo Navarro, On compressing and indexing repetitive sequences, Theor. Comput. Sci. 483 (2013) 115–133.
[29] Gregory Kucherov, Lilla Tóthmérész, Stéphane Vialette, On the combinatorics of suffix arrays, Inf. Process. Lett. 113 (22–24) (2013) 915–920.
[30] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, Giovanni Manzini, Efficient construction of a complete index for pan-genomics read alignment, J. Comput. Biol. 27 (4) (2020) 500–513.
[31] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biol. 10 (3) (2009) R25.
[32] Heng Li, Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, arXiv, 2013.
[33] Heng Li, Richard Durbin, Fast and accurate short read alignment with Burrows-Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[34] M. Lothaire, Algebraic Combinatorics on Words, Cambridge University Press, 2002.
[35] Veli Mäkinen, Gonzalo Navarro, Succinct suffix arrays based on run-length encoding, Nord. J. Comput. 12 (1) (2005) 40–66.
[36] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, Niko Välimäki, Storage and retrieval of highly repetitive sequence collections, J. Comput. Biol. 17 (3) (2010) 281–308.
[37] Udi Manber, Eugene W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[38] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, An extension of the Burrows-Wheeler transform, Theor. Comput. Sci. 387 (3) (2007) 298–312.
[39] Gonzalo Navarro, Compact Data Structures: A Practical Approach, Cambridge University Press, 2016.
[40] Gonzalo Navarro, Indexing highly repetitive string collections, part I: repetitiveness measures, ACM Comput. Surv. 54 (2) (2022) 29:1–29:31.
[41] Gonzalo Navarro, Veli Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007) 2.
[42] Takaaki Nishimoto, Yasuo Tabei, Optimal-time queries on BWT-runs compressed indexes, in: Proc. of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, in: LIPIcs, vol. 198, 2021, pp. 101:1–101:15.
[43] Ge Nong, Sen Zhang, Wai Hong Chan, Two efficient algorithms for linear time suffix array construction, IEEE Trans. Comput. 60 (10) (2011) 1471–1484.
[44] Alberto Policriti, Nicola Prezza, LZ77 computation based on the run-length encoded BWT, Algorithmica 80 (2017) 1986–2011.
[45] Simon J. Puglisi, William F. Smyth, Andrew Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2) (2007) 4.
[46] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, Christina Boucher, MONI: a pangenomics index for finding MEMs, in: Proc. of the 25th Annual International Conference on Research in Computational Molecular Biology, RECOMB 2021, 2021.
[47] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, Christina Boucher, MONI: a pangenomic index for finding maximal exact matches, J. Comput. Biol. 29 (2) (2022) 169–187.
[48] Georges P. Schmartz, Anna Hartung, Pascal Hirsch, Fabian Kern, Tobias Fehlmann, Rolf Müller, Andreas Keller, PLSDB: advancing a comprehensive database of bacterial plasmids, Nucleic Acids Res. 50 (D1) (2021) D273–D278.
[49] Chen Sun, Zhiqiang Hu, Tianqing Zheng, Kuangchen Lu, Yue Zhao, Wensheng Wang, Jianxin Shi, Chunchao Wang, Jinyuan Lu, Dabing Zhang, Zhikang Li, Chaochun Wei, RPAN: rice pan-genome browser for 3000 rice genomes, Nucleic Acids Res. 45 (2) (2017) 597–605.
[50] The 1001 Genomes Consortium. Epigenomic Diversity in a Global Collection of Arabidopsis Thaliana Accessions, Cell 166 (2) (2016) 492–505.
[51] Clare Turnbull, et al., The 100,000 genomes project: bringing whole genome sequencing to the NHS, Br. Med. J. 361 (2018).
[52] Daniel Valenzuela, CHICO: a compressed hybrid index for repetitive collections, in: Andrew V. Goldberg, Alexander S. Kulikov (Eds.), Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 9685, Springer, 2016, pp. 326–338.