Bi-Directional r-Indexes

Yuma Arakawa
Department of Mathematical Informatics, The University of Tokyo, Japan

Gonzalo Navarro
CeBiB and Department of Computer Science, University of Chile, Santiago, Chile

Kunihiko Sadakane
Department of Mathematical Informatics, The University of Tokyo, Japan

Abstract

Indexing highly repetitive texts is important in fields such as bioinformatics and versioned repositories. The run-length compression of the Burrows-Wheeler transform (BWT) provides a compressed representation particularly well-suited to text indexing. The r-index is one such index. It enables fast locating of occurrences of a pattern within $O(r)$ words of space, where r is the number of equal-letter runs in the BWT. Its mechanism of locating is to maintain one suffix array sample along the backward-search of the pattern, and to compute all the pattern positions from that sample once the backward-search is complete. In this paper we develop this algorithm further, and propose a new bi-directional text index called the br-index, which supports extending the matched pattern both in forward and backward directions, and locating the occurrences of the pattern at any step of the search, within $O(r + r_R)$ words of space, where r_R is the number of equal-letter runs in the BWT of the reversed text. Our experiments show that the br-index captures the long repetitions of the text, and outperforms the existing indexes in text searching allowing some mismatches except in an internal part.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Compressed text indexes, Burrows-Wheeler Transform, highly repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.11

Supplementary Material *Software (Source Code)*: `https://github.com/U-Ar/br-index`
archived at `swh:1:dir:988f1c8381e90fa759b316908113e0c5cf92f228`

Funding *Gonzalo Navarro*: Funded in part by Basal Funds FB0001 and Fondecyt Grant 1-200038, ANID, Chile.

1 Introduction

A text index is a data structure equipped with search operations on a text string. The *suffix tree* [23], which is the compacted trie whose paths to the leaves spell out the suffixes of the text, enables various complex operations useful in bioinformatics [8]. The *suffix array* [14] is a simplified variant of the suffix tree with less space usage but also less functionality. It still supports the most basic searches, *counting* and *locating* the occurrences of a pattern in the text, among more sophisticated ones [11]. Compressed suffix arrays are suffix array representations that retain its functionality within further compressed space. One of those, the *FM-index* [3], is based on the *Burrows-Wheeler transform (BWT)* [2], which searches for the pattern by starting from its last character and extends the match leftwards. The *bi-directional BWT* [10] also supports rightward extension by constructing FM-indexes on both the text and the reversed text, thus using roughly twice the space of the FM-index. This extended functionality allows retrieving some of the lost suffix tree functionality.

Classical compressed suffix arrays are based on statistical compression. This cannot capture repetitions of long text substrings when indexing highly repetitive texts, so the index sizes grow proportionally to the input sizes. Large highly repetitive texts are arising in bioinformatic applications and versioned document and software stores. For those texts,

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).
Editors: Hideo Bannai and Jan Holub; Article No. 11; pp. 11:1–11:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Comparison of space and time with the existing compressed bi-directional indexes. H is the length of the longest maximal repeat in the text. *right-extension*(*contraction*) is symmetric to *left-extension*(*contraction*). Here w is the number of bits in the computer word.

index	space	*left-extension*
bi-directional BWT [10]	$\mathrm{O}(nH_k(T)) + o(n \log \sigma)$ bits	$\mathrm{O}(\frac{\log \sigma}{\log \log n})$
Belazzougui and Cunial [1]	$\mathrm{O}(r + r_R)$ words	$\mathrm{O}(H^2 \log \log n)$
br-index (Theorem 1)	$\mathrm{O}(r + r_R)$ words	$\mathrm{O}(\sigma + \log \log_w (n/r))$
br-index (Theorem 2)	$\mathrm{O}(r + r_R)$ words	$\mathrm{O}(\frac{1}{\epsilon} \log^{2+\epsilon} r)$

index	*left-contraction*	*locate*
bi-directional BWT [10]	not supported	$\mathrm{O}(occ \cdot \log^{1+\epsilon} n)$
Belazzougui and Cunial [1]	$\mathrm{O}(H^2 \log \log n)$	not supported
br-index (Theorem 1)	not supported	$\mathrm{O}(occ)$
br-index (Theorem 2)	not supported	$\mathrm{O}(occ)$

indexes based on compression methods such as Lempel-Ziv and grammar compression have been proposed [17]. While those indexes can locate, and in some cases count, the pattern occurrences, they are not based on suffix arrays and therefore lack the potential to enable other more sophisticated suffix array functionalities. The *r-index* [5, 6] is the first compressed suffix array suitable for highly repetitive texts. It is based on the run-length compression of the BWT and uses $O(r)$ space, where r, the number of equal-letter runs in the BWT, stays low on repetitive texts. The r-index enables efficient *count* and *locate* queries within that space, but more complex operations that are supported on classical suffix arrays are yet to be studied. In particular, an index supporting bi-directional extensions based on this compression method has been proposed [1], but it does not support the key *locate* operation.

Our contribution. We introduce the *br-index*, an r-index extension that supports bi-directional extensions along the pattern search process, within $\mathrm{O}(r + r_R)$ words of space, where r_R is the number of equal-letter runs in the BWT of the reversed text. The simpler version of Theorem 1 is easily built on top of the r-index of both of the text and its reverse. The refined version of Theorem 2 reduces the σ term in the computation time of *left-extension* and *right-extension* (where σ is the alphabet size), and is more advantageous when σ is large. Compared to the bi-directional BWT [10], the br-index captures long repetitions in the text and thus compresses highly repetitive text collections. Compared to the index proposed by Belazzougui and Cunial [1], the br-index enables *locate* in efficient time and is easier to implement, though it does not support contractions (i.e., the inverses of expansions). See Table 1 for a detailed comparison. We also implemented the version of Theorem 1 and compared its practical performance with the bi-directional BWT and the r-index.

This paper is organized as follows. In Section 2 we describe the needed concepts to present our results. In Section 3 we introduce the algorithmic details of the br-index. Section 4 shows the experimental results. We conclude in Section 5.

2 Preliminaries

2.1 Basic notions

In this paper, we call a sequence of characters $T = T[1]T[2] \cdots T[n]$ a *string* of length n. Each character $T[i]$ $(i = 1, \ldots, n)$ is an element of an ordered *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. Here we assume Σ is the *effective alphabet*, which means that each character in Σ appears at least once in T. For convenience, we assume $T[n] = 1$ and $T[i] \neq 1$ $(i = 1, \ldots, n-1)$, that is, the last character is a unique endmarker with the minimum lexicographic rank. In addition, we call the sequence of characters $T^R = T[n-1]T[n-2]\cdots T[1].1$ the *reversed string*. In other words, we obtain T^R by reversing the meaningful content of the string and attaching the character 1 at the end.

We define two queries on T, where P is a sequence of m characters:

***count*(*P*)** returns the number of the occurrences of the pattern P in T.
***locate*(*P*)** returns the starting positions of the occurrences of the pattern P in T.

We write $[l, r]$ for the set of integers $\{l, l+1, \ldots, r\}$ ($\emptyset$ if $l > r$). This notation is used to describe substrings and subsequences as well; $T[l, r]$ is the substring $T[l]T[l+1]\cdots T[r]$, which is the empty string ε if $l > r$.

A *bitvector* B is an array whose elements are 0 or 1. We define two queries on a bitvector, $\mathit{rank}_1(B, j)$ returns the number of 1-bits in $B[1, j]$ and $\mathit{select}_1(B, i)$ returns the position of the i-th 1-bit in B.

A *predecessor data structure* on the totally ordered set S supports the query $\mathit{pred}(S, i)$, which returns the maximum element that is smaller than or equal to i, $\max\{s \in S \mid s \leq i\}$.

2.2 Suffix array, Burrows-Wheeler transform, and LCP array

The *suffix array* [14] of T is an array of integers $SA[1, n]$, where $SA[i]$ is the starting position in T of the i-th lexicographically smallest suffix of T, that is, the lexicographic rank of the suffix $T[SA[i], n]$ is i. We also denote the inverse of the suffix array by ISA, that is, $SA[\mathit{ISA}[i]] = i$ $(i = 1, \ldots, n)$.

The *Burrows-Wheeler transform* (BWT) [2] of T is a sequence $L[1, n]$ of characters that satisfies

$$L[i] = \begin{cases} T[SA[i]-1] & (SA[i] \neq 1) \\ 1 & (SA[i] = 1) \end{cases}$$

Note that $L[i]$ is the character preceding the i-th suffix in lexicographic order. Exceptionally $L[i] = 1$ when the i-th suffix is the whole string T. We also define a function *rank* on L: $\mathit{rank}_c(L, i)$ is the number occurrences of the character c in $L[1, i]$. It is 0 if $i = 0$.

The *longest common prefix array* (LCP) of T is an array $\mathit{LCP}[1, n]$ of integers satisfying

$$\mathit{LCP}[i] = \begin{cases} \mathit{lcp}(T[SA[i-1], n], T[SA[i], n]) & (i \neq 1) \\ 0 & (i = 1) \end{cases}$$

where $\mathit{lcp}(P, P')$ is the length of the longest common prefix between strings P and P'.

2.3 Backward search

The suffix array SA and the BWT L are useful for computing *count* and *locate* of a pattern $P[1, m]$ [3]. Given P, there exists a unique range $[s, e]$ on SA corresponding to the occurrences of P (the range is empty when P does not occur in T). In this case, $SA[s, e]$ is the list of the starting positions of P in T. We can then represent (the occurrences of) P by the range $[s, e]$. With *rank* on L we can extend the current pattern leftwards. Specifically, we can compute the range $[s', e']$ corresponding to the pattern cP, from the character c and $[s, e]$ corresponding to P, with the following formula. We call this a *left-extension*.

$$\begin{cases} s' = C[c] + \mathit{rank}_c(L, s-1) + 1 \\ e' = C[c] + \mathit{rank}_c(L, e) \end{cases}$$

Here, $C[1, \sigma]$ is the array of integers where $C[c]$ is the number of occurrences of the characters c' satisfying $c' < c$ in T. When cP does not occur in T, the formula yields $e' > s'$.

The *FM-index* [3] is a statistically compressed suffix array. When it computes *count*(P) and *locate*(P), it starts from the end of P and extends leftwards with the formula above. It starts with the empty string ε, whose SA range is $[1, n]$. Then, from the range $[s_{i+1}, e_{i+1}]$ corresponding to $P[i+1, m]$ $(1 \leq i \leq m)$, it obtains $[s_i, e_i]$ with

$$\begin{cases} s_i = C[P[i]] + \mathit{rank}_{P[i]}(L, s_{i+1} - 1) + 1 \\ e_i = C[P[i]] + \mathit{rank}_{P[i]}(L, e_{i+1}) \end{cases}$$

ending if $s_i > e_i$ or $i = 1$ holds. In the first case, *count*(P) is zero, otherwise it is $e_1 - s_1 + 1$, and the results of *locate*(P) are in $SA[s_1, e_1]$. This searching algorithm is called the *backward search*. We denote the time to compute *left-extension* by t_{LF}, whose name comes from *LF-mapping* $LF(i) = C[L[i]] + \mathit{rank}_{L[i]}(L, i)$. Similarly, the time to access an element of SA is denoted by t_{SA}.

With the backward search algorithm, *count* takes $\mathrm{O}(m \cdot t_{LF})$ time and *locate* takes $\mathrm{O}(m \cdot t_{LF} + \mathit{occ} \cdot t_{SA})$ time, where *occ* is the number of the occurrences of P in T. On an alphabet of size σ, the FM-index achieved $t_{LF} = \mathrm{O}(\frac{\log \sigma}{\log \log n})$ and $t_{SA} = \mathrm{O}(\log^{1+\epsilon} n)$ with $nH_k(T) + o(n \log \sigma)$ bits of space for any constant $0 < \epsilon < 1$, where $H_k(T)$ is the k-th empirical entropy of T [4].

2.4 Run-length compression of BWT and r-index

The size of the representation of L grows linearly with the input size n even if we use statistical compression as in the FM-index. To handle large repetitive text collections we need to capture the repetitions in T and compress them.

Mäkinen and Navarro [12] focused on equal-letter runs in L to capture the repetitiveness. A *run* of the BWT is a maximal substring of L whose characters are equal. Since the suffixes are ordered lexicographically, the sequence of their preceding characters, L, is expected to have long runs if T is highly repetitive. They showed that the number r of such runs is sensitive to the statistical entropy of T, $r \leq nH_k(T) + \sigma^k$ for any $k \geq 0$. In particular, $r \leq nH_k(T) + \mathrm{o}(n)$ for any $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. It was later realized that r was sensitive to the repetitiveness of T, and the run-length-based FM-index (RLFM-index), which compressed the BWT by run-length encoding, was designed [13]. The RLFM-index achieved $t_{LF} = \mathrm{O}(\frac{\log \sigma}{\log \log r} + (\log \log n)^2)$ in $\mathrm{O}(r)$ words of space by emulating access and *rank* on L. From this, we can compute *count* within $\mathrm{O}(r)$ words with the RLFM-index, but *locate* is not supported in the same space. To do that, additional $\mathrm{O}(n/s)$ words of space, where s is a sampling parameter, is required to store samples of SA at regularly spaced intervals. Since this method yields $t_{SA} = \mathrm{O}(s \cdot t_{LF})$, saving spaces with larger s in turn worsens the time complexity.

The *r-index* [5, 6] made it possible to compute *locate* in $\mathrm{O}(m \cdot (t_{LF} + \log \log_w(n/r)) + \mathit{occ} \cdot t_\phi)$ time within $\mathrm{O}(r)$ words of space, without the SA samplings at regular intervals. To compute *rank* on L, it uses an updated version of the RLFM-index, which achieves $t_{LF} = \mathrm{O}(\log \log_w(\sigma + n/r))$. The removal of SA samplings is achieved by maintaining one SA sample during the backward search and designing inverse functions ϕ and ϕ^{-1}, whose computation time is denoted by t_ϕ:

$$\phi(i) = \begin{cases} SA[ISA[i]-1] & (ISA[i] \neq 1) \\ SA[n] & (ISA[i] = 1) \end{cases} \qquad \phi^{-1}(i) = \begin{cases} SA[ISA[i]+1] & (ISA[i] \neq n) \\ SA[1] & (ISA[i] = n) \end{cases}$$

These functions enable us to compute neighboring SA values from an SA sample. From a sample $SA[i]$, we obtain $SA[i-1]$ by applying ϕ and $SA[i+1]$ by applying ϕ^{-1}. They compute those functions in time $t_\phi = \mathrm{O}(\log\log_w(n/r))$. To explain our results later, we describe next the algorithm to maintain an SA sample during the backward search.

We say character $T[i]$ is *sampled* if and only if $i = 1$ or $T[i]$ is the first or last character of a BWT run. The number of the sampled characters is $\mathrm{O}(r)$. In addition to the RLFM-index, we store a predecessor data structure R_c for each c, with the BWT positions of all the sampled characters equal to c. We associate each BWT position $q \in R_c$ with the pair $\langle q, SA[q]-1\rangle$. During the backward search, we know an SA sample $(p, SA[p])$ in the current SA range $[s, e]$ and update it using R_c. Assume we are extending $P[i+1, m]$ to $P[i, m]$ during the backward search. We want to compute the SA range $[s_i, e_i]$ corresponding to $P[i, m]$ and the new sample $p', SA[p']$ $(s_i \leq p' \leq e_i)$, from the range $[s_{i+1}, e_{i+1}]$ corresponding to $P[i+1, m]$ and the current sample $(p, SA[p])$ $(s_{i+1} \leq p \leq e_{i+1})$. $[s_i, e_i]$ is computed using the RLFM-index. If $L[p] = P[i]$, $LF(p) \in [s_i, e_i]$ holds, so the sample can be updated to $(p' = LF(p), SA[p'] = SA[p]-1)$. In the other case, where $L[p] \neq P[i]$ but $P[i]$ still occurs somewhere else, we obtain a predecessor $\langle q, SA[q]-1\rangle$ by querying $pred(R_{P[i]}, e_{i+1})$. Since $L[q] = P[i]$ holds, the sample is updated to $(p' = LF(q), SA[p'] = SA[q]-1)$.

Nishimoto and Tabei [19] recently managed to improve the times of the operations to $t_{LF} = \mathrm{O}(1)$ and $t_\phi = \mathrm{O}(1)$, still within $\mathrm{O}(r)$ words, by avoiding predecessor queries.

3 Bi-directional r-index

With the r-index, we can compute *left-extension* and locate all the occurrences of the current pattern at any step of the extensions. However, the extension is unidirectional; *right-extension* cannot be carried out. The text index we propose, br-index, enables us to extend in both directions and compute *locate* at an arbitrary step, as shown in the following theorem.

▶ **Theorem 1.** *We can store $\mathrm{O}(r) + \mathrm{O}(r_R)$ words such that, at an arbitrary step of the search, we can execute left-extension in $\mathrm{O}(\sigma t_{LF} + \log\log_w(n/r))$ time, right-extension in $\mathrm{O}(\sigma t_{LF^R} + \log\log_w(n/r_R))$ time, compute count of the current pattern in $\mathrm{O}(1)$ time, and compute locate of the current pattern in $\mathrm{O}(occ)$ time, where occ is the number of the occurrences of the current pattern in the string, w is the number of bits in the computer word, and r_R is the number of runs in the BWT L^R of the reversed string T^R.*

▶ Remark. The best known upper bound of r_R by r is $r_R = \mathrm{O}(r \log r \max(1, \log \frac{n}{r \log r}))$ [9]. In practice, their values are very close; see Section 4.

In Sections 3.1 and 3.2 we prove Theorem 1. In Section 3.3, we propose a variant using the wavelet tree [7], which achieves the improved time bounds of *left-extension* and *right-extension*, as seen in Theorem 2.

▶ **Theorem 2.** *For any $\epsilon > 0$, we can store $\mathrm{O}(r) + \mathrm{O}(r_R)$ words such that, at any arbitrary step of the search, we can execute left-extension in $\mathrm{O}(\frac{1}{\epsilon}\log^{2+\epsilon} r)$ time, right-extension in $\mathrm{O}(\frac{1}{\epsilon}\log^{2+\epsilon} r_R)$ time, compute count of the current pattern in $\mathrm{O}(1)$ time, and compute locate of the current pattern in $\mathrm{O}(occ)$ time, where occ is the number of the occurrences of the current pattern in the string.*

The key idea of the br-index is to compute *locate* efficiently by maintaining one SA sample and one SA^R sample at the same time. These samples are not necessarily starting or ending positions of the current pattern. Instead, we also maintain their offsets towards both ends, and the length of the current pattern.

3.1 Left-extension and right-extension

Updating the ranges on SA and SA^R

Let $[s, e]$ be the range on SA corresponding to the current pattern P. Similarly, let $[s_R, e_R]$ be the range on SA^R corresponding to P^R.

When we compute *left-extension* $P \to cP$, we update $[s, e]$ by $s \leftarrow C[c] + rank_c(L, s - 1) + 1, e \leftarrow C[c] + rank_c(L, e)$. To update $[s_R, e_R]$, we use another idea [10]. We count the total number acc of occurrences of patterns aP for all $a < c$, by applying LF iteratively for each such a. Since the size of the range of any pattern is equal on SA and SA^R, we can update $[s_R, e_R]$ by $s_R \leftarrow s_R + acc, e_R \leftarrow s_R + acc + e - s$. *right-extension* is symmetric. In this case, we apply LF^R, which is LF-mapping on the BWT of T^R, instead of LF.

The required structures to update the ranges are just the RLFM-indexes on T and T^R. The space used is $O(r + r_R)$ words, the time complexity is $O(\sigma t_{LF})$ when we extend leftward, and $O(\sigma t_{LF^R})$ when we extend rightward, where t_{LF^R} is the time to compute LF^R.

Updating the sample

In addition to the SA range $[s, e]$ and the SA^R range $[s_R, e_R]$, we maintain seven variables during the search: $p, j, d, p_R, j_R, d_R, len$. We call the tuple of these variables the *sample*: p is the position of the sample in SA, j is the value of $SA[p]$, and d is the offset of j to the starting position of the current pattern. That is, it holds $j = SA[p]$ and $T[j - d, j - d + |P| - 1] = P$. The corresponding values for the reversed direction are $j_R = SA^R[p_R]$ and $T^R[j_R - d_R, j_R - d_R + |P| - 1] = P^R$. Finally, len is the length of the pattern.

We note, however, that we will not be able to maintain p and p_R in all cases; we will manage without them. We still speak of those variables for reasoning about correctness.

Assume we are computing *left-extension* $P \to cP$. If the size of the range $[s, e]$ on SA corresponding to the pattern does not change, only the character c precedes P in T. In this case, we simply increment d and len. Otherwise, we compute the predecessor $pred(R_c, e)$, to obtain $\langle q, SA[q] - 1\rangle$. We then update $j \leftarrow SA[q] - 1$ and $j_R \leftarrow n - j$. Also, offsets are updated to $d \leftarrow 0, d_R \leftarrow len$, and $len \leftarrow len + 1$. The case of *right-extension* is symmetric.

The details are shown in Algorithms 1 and 2. In the following lemma, we prove the invariant conditions that hold during the extensions. These conditions are important for the correctness of the *locate* algorithm presented in the next section.

▶ **Lemma 3.** *Assume we are computing left-extension and right-extension, and the current pattern is P. Then the following conditions are invariant, except when P is empty.*
(1) $len = |P|$
(2) $d + d_R + 1 = len$
(3) *Let $j = SA[p]$ and $j_R = SA^R[p_R]$, then $s \le LF^d(p) \le e$ and $s_R \le (LF^R)^{d_R}(p_R) \le e_R$*

Proof. When we start with an empty pattern $P = \varepsilon$, we initialize the ranges and the sample with $s = s_R = 1, e = e_R = n, len = d = d_R = 0$. We then obtain an arbitrary predecessor $\langle q, SA[q] - 1\rangle$ and set $j = y$ and $j_R = n - y$. We now prove that the invariants are maintained by *left-extension*; *right-extension* is symmetric.

Algorithm 1 Left-extension $P \to cP$.

Input: A character c and values corresponding to $P : [s, e],\ [s_R, e_R],\ j,\ d, len$
Output: Values corresponding to $cP : [s', e'],\ [s'_R, e'_R],\ j',\ j'_R,\ d',\ d'_R,\ len'$
1: $s' \leftarrow C[c] + rank_c(L, s-1) + 1$
2: $e' \leftarrow C[c] + rank_c(L, e)$
3: **if** $s' > e'$ **then**
4: $\quad cP$ does not occur.
5: **else**
6: $\quad acc \leftarrow 0$
7: $\quad$ **for** $a = 1$ **to** $c - 1$ **do**
8: $\quad\quad acc \leftarrow acc + rank_a(L, e) - rank_a(L, s-1)$
9: $\quad$ **end for**
10: $\quad [s'_R, e'_R] \leftarrow [s_R + acc, s_R + acc + e' - s']$
11: $\quad$ **if** $e' - s' \neq e - s$ (cP and $c'P$ occur for some $c' \neq c$) **then**
12: $\quad\quad (q, j') \leftarrow pred(R_c, e),\ d' \leftarrow 0$
13: $\quad$ **else**
14: $\quad\quad j' \leftarrow j,\ d' \leftarrow d + 1$
15: $\quad$ **end if**
16: $\quad j'_R \leftarrow n - j',\ d'_R \leftarrow len - d'$
17: $\quad len' \leftarrow len + 1$
18: **end if**

First, consider the case where $e' - s' \neq e - s$ in line 11 of Algorithm 1. (1) Since len' is incremented from len, $len' = |cP|$ holds. (2) $d' + d'_R + 1 = 0 + len + 1 = len'$ holds. (3) From the definition of R_c, $j' = SA[q] - 1$, so the new value for p is $p' = LF(q)$. Also, since $j'_R = n - j' = n - (SA[q] - 1) = SA^R[ISA^R[n - SA[q] + 1]]$, it holds that the new value for p_R is $p'_R = ISA^R[n - SA[q] + 1]$. Now, cP and $c'P$ ($c' \neq c$) occur in this case, which means an end of a BWT run of the character c exists in $[s, e]$. Thus, $s \leq q \leq e$ and $L[q] = c$ holds, which in turn implies $s' \leq LF(q) = p' \leq e'$. On the other hand, $SA^R[(LF^R)^{d'_R}(p'_R)] = SA^R[p'_R] - d'_R = j'_R - d'_R = (n - j') - d'_R = n - (j' + d'_R)$ holds. This position in T^R corresponds to the position $j' + d'_R = j' + len' - d' - 1 = SA[LF^{d'}(p')] + len' - 1$ in T. This is the ending position of the pattern cP in T, and the starting position of P^Rc in T^R. Therefore $s'_R \leq (LF^R)^{d'_R}(p'_R) \leq e'_R$ holds.

Second, consider the other case, where $e' - s' = e - s$ in line 13 of Algorithm 1. This case does not happen when P is empty since T contains at least two distinct characters. Thus, the inductive assumption can be used. That is, we assume that the three conditions hold before the execution of *left-extension*. (1) Same as the former case. (2) $d' + d'_R + 1 = d + 1 + d_R + 1 = len + 1 = len'$ holds from the inductive assumption. (3) Note that j and j_R do not change, so $p' = p$ and $p'_R = p_R$. In this case c precedes all the occurrences of P. Thus, $s'_R = s_R$ and $e'_R = e_R$, and since we also maintain $d'_R = d_R$, the relation $s_R = s'_R \leq (LF^R)^{d'_R}(p'_R) = (LF^R)^{d_R}(p_R) \leq e'_R = e_R$ stays true by induction. On the other hand, $s' = C[c] + rank_c(L, s-1) + 1 = C[c] + rank_c(L, s)$, $e' = C[c] + rank_c(L, e)$, and $LF^{d'}(p') = LF(LF^d(p)) = C[c] + rank_c(L, LF^d(p))$ holds since $L[s] = L[LF^d(p)] = c$. Therefore, $s' \leq LF^{d'}(p') \leq e'$ holds from the inductive assumption. ◀

Algorithm 2 Right-extension $P \to Pc$.

Input: A character c and values corresponding to $P : [s, e],\ [s_R, e_R],\ j_R,\ d_R,\ len$
Output: Values corresponding to $Pc : [s', e'],\ [s'_R, e'_R],\ j',\ j'_R,\ d',\ d'_R,\ len'$

$s'_R \leftarrow C[c] + rank_c(L^R, s_R - 1) + 1$
$e'_R \leftarrow C[c] + rank_c(L^R, e_R)$
if $s'_R > e'_R$ **then**
  Pc does not occur.
else
  $acc \leftarrow 0$
  for $a = 1$ **to** $c - 1$ **do**
    $acc \leftarrow acc + rank_a(L^R, e_R) - rank_a(L^R, s_R - 1)$
  end for
  $[s', e'] \leftarrow [s + acc, s + acc + e'_R - s'_R]$
  if $e'_R - s'_R \neq e_R - s_R$ (Pc and Pc' occur for some $c' \neq c$) **then**
    $(q_R, j'_R) \leftarrow pred(R^R_c, e_R),\ d'_R \leftarrow 0$
  else
    $j'_R \leftarrow j_R,\ d'_R \leftarrow d_R + 1$
  end if
  $j' \leftarrow n - j'_R,\ d' \leftarrow len - d'_R$
  $len' \leftarrow len + 1$
end if

3.2 Determining the end of *locate* with run-length compressed PLCP

We now present the algorithm for *locate*. We can obtain the values $SA[i-1], SA[i+1]$ from $SA[i]$, using just the functions ϕ and ϕ^{-1} of the r-index. Therefore, neighboring SA values are obtained sequentially from component j, d of the sample. However, because we do not know $p' = LF^d(p)$, we cannot determine how many values $i < p'$ and $i > p'$ are within the range $[s, e]$ corresponding to the current pattern P.

In order to determine the ends of the iterative computations of ϕ and ϕ^{-1}, we make use of the permuted LCP array $PLCP[1, n]$, which satisfies $PLCP[i] = LCP[ISA[i]]$ $(i = 1, \ldots, n)$. Let the current position in SA be $p' \in [s, e]$. When we are computing the value of $SA[p' - 1]$ from $SA[p']$, we compare $PLCP[SA[p']]$ with $|P|$. If $PLCP[SA[p']]$ is smaller than $|P|$, $SA[p' - 1]$ does not correspond to an occurrence of the whole pattern P. Thus, $p' = s$ holds in this case. Otherwise we go on and compute ϕ. Similarly, when we compute $SA[p' + 1]$ from $SA[p']$, we compare $PLCP[SA[p' + 1]]$ with $|P|$.

The details are shown in Algorithm 3. In the following lemma, we prove that Algorithm 3 runs properly if the invariant conditions hold. Combining Lemmas 3 and 4, we obtain the correctness of *locate*.

▶ **Lemma 4.** *Let $[s, e]$ be the range on SA that corresponds to the current pattern P. Assume the input of Algorithm 3 satisfies $j = SA[p], s \leq LF^d(p) \leq e, len = |P|$. Then Algorithm 3 correctly outputs all the positions of the occurrences of P.*

Proof. The correctness of ϕ, ϕ^{-1} is proved in [6, Lem. 3.5]. Since $j = SA[p]$, $j' = j - d$ is equal to $SA[p']$ $(p' = LF^d(p))$. Provided $s \leq p' \leq e$, we have to prove

- $PLCP[SA[p']] \geq |P| \Rightarrow p' > s$
- $PLCP[SA[p']] < |P| \Rightarrow p' = s$

In the case where $PLCP[SA[p']] \geq |P|$, $PLCP[SA[p']] = LCP[ISA[SA[p']]] = LCP[p'] = lcp(T[SA[p'], n], T[SA[p' - 1], n]) \geq |P|$ holds. Since the first $|P|$ characters of $T[SA[p'], n]$ are identical to P from the assumption, the first $|P|$ characters of $T[SA[p' - 1], n]$ are also

Algorithm 3 Locate the current pattern P.

Input: $p,\ j(= SA[p]),\ d,\ len(= |P|)$
Output: All the starting positions of the occurrences of P in T

$j' \leftarrow j - d\ (= SA[LF^d(p)])$
$pos \leftarrow j'$
output pos
while $PLCP[pos] \geq len$ **do**
  $pos \leftarrow \phi(pos)$
  output pos
end while
$pos \leftarrow j'$
while true **do**
  if $pos = SA[n]$ **then return**
  $pos \leftarrow \phi^{-1}(pos)$
  if $PLCP[pos] < len$ **then return**
  output pos
end while

the same as P. Thus, $p' - 1$ is also within the range $[s, e]$, which means $p' > s$. On the other hand, when $PLCP[SA[p']] < |P|$, $lcp(T[SA[p'], n], T[SA[p' - 1], n]) < |P|$ holds. In this case, at least one character among the first $|P|$ characters of $T[SA[p'], n]$ and $T[SA[p' - 1], n]$ differ. Since the first $|P|$ characters of $T[SA[p'], n]$ are identical to P, the first $|P|$ characters of $T[SA[p' - 1], n]$ are not the same as P. Thus, $p' - 1$ is out of the range $[s, e]$, which means $p' = s$. Similarly,

- $PLCP[SA[p' + 1]] \geq |P| \Rightarrow p' < e$
- $PLCP[SA[p' + 1]] < |P| \Rightarrow p' = e$

holds when $p' \leq n - 1$, so we can correctly decide whether $s \leq p' \leq e$ holds.

From the above arguments, we can locate all the occurrences of P using Algorithm 3. ◀

If we use a predecessor data structure to store $PLCP$ in $O(r)$ words of space, we can access one value of $PLCP$ in $O(\log \log_w(n/r))$ time [6, Lem. 3.8.]. As a more sophisticated solution, ϕ, ϕ^{-1} and $PLCP$ can be computed simultaneously in $O(1)$ time within $O(r)$ words of space, with a *move data structure* [19]. The algorithm to compute ϕ^{-1} is explained in [19]. ϕ is symmetric. We integrate a procedure to compute $PLCP$ into the algorithm. In addition to the values of ϕ and ϕ^{-1} stored in the structure, we store the values of $PLCP$ at the same sampled positions. We compute the predecessor by a *move* query, obtain its $PLCP$ value, and subtract the offset between the current position and the predecessor from the value. Therefore we obtain Theorem 1.

3.3 Improving the *extend* time with wavelet tree

In lines 7-9 of Algorithm 1, *rank* on L is computed for $O(\sigma)$ times in order to calculate the accumulated number of occurrences of $c'P$ ($c' < c$). These computations are costly when σ is large. We could easily compute the accumulated number in $O(\log \sigma)$ time on the wavelet tree of the BWT, since it is a range-counting problem [15]. This is not that simple, however, on the run-length BWT representation. We now show that polylogarithmic time is still possible, however.

Consider the sequence $L'[1,r]$ of the *run heads* in the BWT, that is, the first characters of the BWT runs. Regard L' as the 2-dimensional grid G of size $r \times \sigma$ which has r points, whose x-coordinates are the positions in L' and y-coordinates are the characters. That is, if $L'[i] = c$, there is a grid point at (i, c). Give to that point a *weight*, equal to the length of the corresponding run in L. We can apply the following theorem on that grid (simplified for our purpose).

▶ **Theorem 5** ([16]). *Let a grid of size $r \times r$ store r points with associated non-negative integers whose values are at most n. For any $\epsilon > 0$, a structure of $\mathrm{O}(\frac{1}{\epsilon} r \log n)$ bits can compute the sum of the integers in any rectangular range in time $\mathrm{O}(\frac{1}{\epsilon} \log^{2+\epsilon} r)$.*

Since the shape of the grid is required to be $r \times r$ in Theorem 5, we extend the $r \times \sigma$ grid with an empty area. We also need a way to determine, given a position $L[i]$, the run it belongs to, and the start/end positions of that run in L. This is already supported by the r-index structures, in time $O(\log\log_w(n/r))$.

With these structures, we count the number of symbols $< c$ in $L[l, r]$ as follows. (1) Compute the runs x_1 and x_2 where l and r belong, respectively, the ending position l' of the x_1-th run and the starting position r' of the x_2-th run. (2) Compute, using Theorem 5, the sum of the weights of the points falling in $[x_1 + 1, x_2 - 1] \times [1, c - 1]$. (3) Add $l' - l + 1$ if $L[l] < c$, and $r - r' + 1$ if $L[r] < c$.

We thus construct the structure of Theorem 5 on L and on L^R. We obtain Theorem 2 by noting that all the times of the form $O(\log\log_w(n/r))$ come from predecessor queries, which can also be done in time $O(\log r)$ by resorting to binary search.

4 Experiments

4.1 Experimental setup

In order to test the practical performance of the index, we experimented on repetitive datasets taken from the Pizza&Chili Repetitive Corpus.[1] Their characteristics are shown in Table 2. We compared the br-index with the r-index and the bi-directional FM-index (2BWT) built on the same datasets. For the br-index, we implemented the differentially encoded *PLCP* with a sparse bitmap [22, 20]. For the 2BWT, we tested $s = 16, 32, 64, 128$ as the sampling parameter of *SA*. Also, as the components of the 2BWT, we used the wavelet trees implemented with RRR bitvectors [21].

We evaluated all the experiments in a machine with Intel Xeon CPU E5-2650 v2 clocked at 2.60GHz and the 128GB memory. The compiler was gcc 4.8.5 and the compiler options were `-std=c++11 -Ofast -march=native`.

In addition to comparing the spaces used by the indexes, we demonstrate the power of the extended primitives on a simplified variant of a popular bioinformatics query, the so-called *seed-and-extend* approach used in BLAST. In the query, we consider a pattern divided into three parts, $P = P_1 P_2 P_3$. We locate all the occurrences of P allowing up to k mismatches in P_1 and P_3, while P_2 is matched exactly. Note that we do not locate the occurrences of P with mismatches in P_2, even if the total number of mismatches in P is within k. On the 2BWT and the br-index, we execute the query by first searching for P_2 in exact form.

[1] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

Table 2 The statistics for the datasets. The lexicographically minimum character attached to the end is included.

datasets	n	σ	r	r_R	r/n
cere	461,286,644	6	11,574,641	11,575,583	0.0251
coreutils	205,281,778	237	4,684,460	4,732,795	0.0228
einstein.de	92,758,441	118	101,370	99,834	0.0011
einstein.en	467,626,544	140	290,239	286,698	0.0006
escherichia	112,689,515	16	15,044,487	15,045,278	0.1335
influenza	154,808,555	16	3,022,822	3,018,825	0.0195
kernel	258,961,616	161	2,791,368	2,780,096	0.0108
para	429,265,758	6	15,636,740	15,635,178	0.0364
world-leaders	46,968,181	90	573,487	583,397	0.0122

Table 3 The sizes (bits/symbol) of the indexes on the repetitive datasets. s is the sampling parameter for SA.

	2BWT				r-index	br-index
	$s = 16$	$s = 32$	$s = 64$	$s = 128$		
cere	8.44	6.33	5.27	4.73	1.93	5.63
coreutils	12.80	10.68	9.61	9.07	1.87	4.92
einstein.de	11.08	8.96	7.90	7.36	0.099	0.276
einstein.en	11.97	9.86	8.79	8.24	0.057	0.162
escherichia	10.18	8.07	7.00	6.46	9.20	26.89
influenza	8.80	6.69	5.62	5.09	1.49	4.32
kernel	12.32	10.20	9.14	8.60	0.90	2.54
para	8.61	6.50	5.43	4.90	2.76	8.07
world-leaders	11.38	9.26	8.20	7.66	0.96	2.74

Then we extend the match leftwards to any $P_1'P_2$, where P_1' has $0 \leq k' \leq k$ mismatches with respect to P_1. This is done with the usual backtracking mechanism starting from the range of P_2, using *left-extension* on every possible symbol as long as the error threshold permits. Finally, we extend each resulting range rightwards using *right-extension*, finding P_3 with at most $k - k'$ mismatches, and report all the occurrences found.

This strategy cannot be used on the r-index, because it cannot extend rightwards. In this case, we tested two different algorithms. The first algorithm, which we call *match-first*, searches for the pattern from the end to the beginning using *left-extension*, allowing up to k mismatches when matching P_3 and P_1. This is likely to be considerably slower because it does not restrict the matches to P_2 before starting to allow errors. The second algorithm, which we call *locate-first*, finds all the occurrences of P_2 with just the r-index, and extracts the text around each occurrence to check if the number of mismatches in P_1' and P_3' is within k. This algorithm is similar to the approach of BLAST, although we extract the characters around P_2 using LF and FL (the inverse function of LF) because we were not storing the plain text. This approach can work well if P_2 is long enough, although it scales linearly with the text size.

We extracted 100 random substrings of length $16, 32, 64$ as the target patterns from influenza, and computed *seed-and-extend* for each pattern. P_2 is set at the middle of P, with length $\lceil |P|/3 \rceil$. The number of allowed mismatches was between 0 and 10.

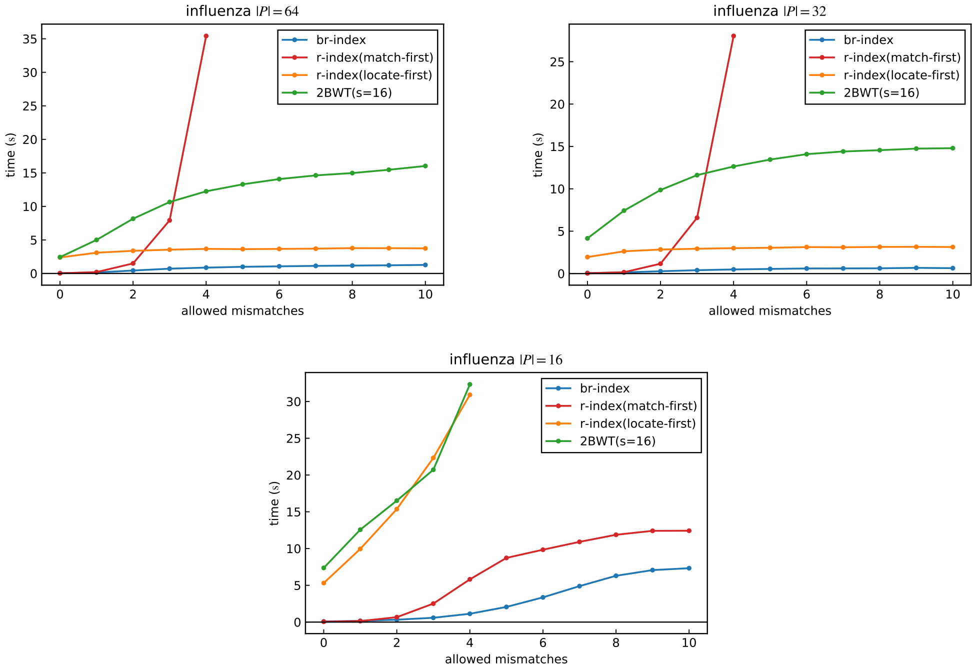

Figure 1 The total computation times of *seed-and-extend* query for all the target patterns on influenza with the number of allowed mismatches between 0 and 10. The 2BWT sometimes mistakenly locates positions for unknown reasons, but the number of reported patterns is very close to that of other indexes.

4.2 Experimental results

The index sizes are shown in Table 3. The br-index is smaller than the 2BWT in many cases. Exceptionally, the br-index is larger when built on escherichia, where r/n is relatively large. The br-index is about 3 times larger than the r-index in all cases. This is expected because we store L, L^R, $PLCP$, and the structures to compute ϕ^{-1} (in practice the r-index works with only ϕ).

Figure 1 shows the computation times of *seed-and-extend*. As it can be seen, the br-index and the 2BWT yield curves with similar shape, though the br-index is an order of magnitude faster. The match-first algorithm we use on the r-index, instead, is sharply outperformed as soon as we allow a few mismatches, as expected. When the pattern is short, the approach manages to outperform the 2BWT, but still the br-index is considerably faster. The br-index is also faster than the locate-first algorithm on the r-index in all cases, and is robust to the increase of allowed mismatches when the pattern is long. The locate-first approach, instead, worsens significantly on short patterns, because in that case P_2 has too many occurrences to verify.

5 Conclusions

We introduced the br-index, which supports the bi-directional extension of the currently searched pattern while efficiently locating all of its occurrences within $O(r + r_R)$ words, by maintaining an SA sample and its offset to the current pattern, and determining the end of the *locate* area using the run-length compressed $PLCP$. In practice, the size of the br-index was observed to be around 3 times as large as that of the r-index [6], and comparable to that of the 2BWT [1], on repetitive datasets. Also, as an application of interleaving *left-extension* and *right-extension*, we tested the *seed-and-extend* query, which finds a pattern allowing some mismatches except in an internal part. The br-index is shown to sharply outperform the r-index on this query, and the gap is likely to grow when allowing more mismatches.

Our work can be seen as a first step towards a fully-functional compressed suffix tree whose size is as close to $O(r + r_R)$ words as possible. The br-index can serve as a component of such a suffix tree, since we can compute *child* and *weiner-link* with it: these operations correspond to *right-extension* and *left-extension*, respectively. On the other hand, *suffix-link* and *parent* are not supported because they need bi-directional pattern contraction. These operations can be carried out with the representation of the suffix tree topology or the random access to *LCP*, both of which require some queries on it. From the perspective of the computation time, the former is more promising in practice [18], while the latter is guaranteed to use $O(r \log \frac{n}{r})$ words [6]. We wonder if the functionality can be supported in $O(r + r_R)$ words, or if another reasonable repetitiveness measure can be defined within which we can represent, for example, the compressed suffix tree topology.

References

1. Djamal Belazzougui and Fabio Cunial. Smaller fully-functional bidirectional BWT indexes. In *International Symposium on String Processing and Information Retrieval*, pages 42–59, 2020.
2. Michael. Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. *Digital SRC Research Report*, 1994.
3. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
4. Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
5. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477, 2018.
6. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):1–54, 2020.
7. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.
8. Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
9. Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In *IEEE 61st Annual Symposium on Foundations of Computer Science*, pages 1002–1013, 2020.
10. T. W. Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and S. M. Yiu. High throughput short read alignment via bi-directional BWT. *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 31–36, 2009.
11. Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
12. Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56, 2005.
13. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

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

15 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

16 Gonzalo Navarro. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science*, 772:58–72, June 2019.

17 Gonzalo Navarro. Indexing highly repetitive string collections, part i. *ACM Computing Surveys*, 54(2), 2021.

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

19 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Leibniz International Proceedings in Informatics*, pages 101:1–101:15, 2021.

20 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pages 60–70, 2007.

21 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.

22 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

23 Peter Weiner. Linear pattern matching algorithms. *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.