Optimal-Time Queries on BWT-Runs Compressed Indexes

Takaaki Nishimoto ✉
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Yasuo Tabei ✉
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Abstract

Indexing highly repetitive strings (i.e., strings with many repetitions) for fast queries has become a central research topic in string processing, because it has a wide variety of applications in bioinformatics and natural language processing. Although a substantial number of indexes for highly repetitive strings have been proposed thus far, developing compressed indexes that support various queries remains a challenge. The *run-length Burrows-Wheeler transform* (RLBWT) is a lossless data compression by a reversible permutation of an input string and run-length encoding, and it has received interest for indexing highly repetitive strings. LF and ϕ^{-1} are two key functions for building indexes on RLBWT, and the best previous result computes LF and ϕ^{-1} in $O(\log \log n)$ time with $O(r)$ words of space for the string length n and the number r of runs in RLBWT. In this paper, we improve LF and ϕ^{-1} so that they can be computed in a constant time with $O(r)$ words of space. Subsequently, we present *OptBWTR (optimal-time queries on BWT-runs compressed indexes)*, the first string index that supports various queries including locate, count, extract queries in optimal time and $O(r)$ words of space.

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.ICALP.2021.101

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2006.05104`

1 Introduction

A string index represents a string in a compressed format that supports locate queries (i.e., computing all the positions at which a given pattern appears in a string). The *FM-index* [10, 11] is an efficient string index on a lossless data compression called the *Burrows-Wheeler transform (BWT)* [5], which is a reversible permutation of an input string. In particular, locate queries can be efficiently computed on an FM-index by performing a *backward search*, which is an iterative algorithm for computing an interval corresponding to the query on a *suffix array* (SA) [19] storing all the suffixes of an input string in lexicographical order. The FM-index performs locate queries in $O(m + occ)$ time with $O(n(\frac{\log \sigma}{\log n} + \frac{1}{s}))$ words of space for a string T of length n, query string of length m, alphabet size σ, parameter s, and number occ of occurrences of a query in T [3].

A *highly repetitive string* is a string including many repetitions. Examples include the human genome, version-controlled documents, and source code in repositories. A significant number of string indexes on various compressed formats for highly repetitive strings have been proposed thus far (e.g., SLP-index [8], LZ-indexes [6, 12], BT-indexes [7, 21]). For a large collection of highly repetitive strings, the most powerful and efficient compressed format is the run-length (RL) Burrows Wheeler transform (RLBWT) [5], which is a BWT compressed by run-length encoding. Mäkinen et al. [18] presented an RLBWT-based string index, named the RLFM-index, that solves locate queries by executing a backward search algorithm on RLBWT. While the RLFM-index can solve locate queries in $O(r + n/s)$ words of space in $O((m + s \cdot occ)(\frac{\log \sigma}{\log \log r} + (\log \log n)^2))$ time for the number r of runs in the RLBWT of T and parameter $s \geq 1$, the size of the index depends on the string length. Recently, Gagie et al. [13] presented the r-index, which can reduce the space usage of the RLFM-index to one linearly proportional to the number of runs in RLBWT. The r-index can solve locate queries space efficiently with only $O(r)$ words of space and in $O(m \log \log_w(\sigma + (n/r)) + occ \log \log_w(n/r))$ time for a machine word size $w = \Theta(\log n)$. If the r-index allows $O(r \log \log_w(\sigma + n/r))$ words of space to be used, it can solve locate queries in the optimal time, $O(m + occ)$. Although there are other important queries including count query, extract query, decompression and prefix search for various applications to string processing, no previous string index can support various queries in addition to locate queries based on RLBWT in an optimal time with only $O(r)$ words of space. That is, developing a string index for various queries in an optimal time with $O(r)$ words of space remains a challenge.

Contribution. In this paper, we present *OptBWTR (optimal-time queries on BWT-runs compressed indexes)*, the first string index that supports various queries including locate, count, extract queries in optimal time and $O(r)$ words of space for the number r of runs in RLBWT. LF and ϕ^{-1} are important functions for string indexes on RLBWT. The best previous data structure computes LF and ϕ^{-1} in $O(\log \log_w(n/r))$ time with $O(r)$ words of space [13]. In this paper, we present a novel data structure that can compute LF and ϕ^{-1} in constant time and $O(r)$ words of space. Subsequently, we present OptBWTR that supports the following five queries in optimal time and $O(r)$ words of space.

- **Locate query:** OptBWTR can solve a locate query on an input string in $O(r)$ words of space and $O(m \log \log_w \sigma + occ)$ time, which is optimal for strings with polylogarithmic alphabets (i.e., $\sigma = O(\text{polylog}\, n)$).
- **Count query:** OptBWTR can return the number of occurrences of a query string on an input string in $O(r)$ words of space and $O(m \log \log_w \sigma)$ time, which is optimal for polylogarithmic alphabets.
- **Extract query:** OptBWTR can return substrings starting at a given position bookmarked beforehand in a string in $O(1)$ time per character and $O(r + b)$ words of space, where b is the number of bookmarked positions. Resolving extract queries is sometimes called the *bookmarking problem* [12, 9].
- **Decompression:** OptBWTR decompresses the original string of length n in optimal time (i.e., $O(n)$). This is the first linear-time decompression algorithm for RLBWT in $O(r)$ words of working space.
- **Prefix search:** OptBWTR can return the strings in a set D that include a given pattern as their prefixes in optimal time (i.e., $O(m + occ')$) and $O(r')$ words of space, where occ' is the number of output strings and r' is the number of runs in the RLBWT of a string made by concatenating the strings in D.

The state-of-the-art string indexes for each type of query are summarized in Table 1.

This paper is organized as follows. In Section 2, we introduce the important notions used in this paper. Section 3 presents novel data structures for computing LF and ϕ^{-1} in constant time. Section 4 presents a data structure supporting a modified version of a backward search on RLBWT. The backward search leverages the two data structures introduced in Section 3. Sections 5 and 6 present OptBWTR that supports all five queries mentioned above by leveraging the modified backward search, LF, and ϕ^{-1}.

Table 1 Summary of space and time for (i) locate and (ii) count queries, (iii) extract queries (a.k.a the bookmarking problem), (iv) decompression of BWT or RLBWT and (v) prefix searches for each query, where n is the length of the input string T, m is the length of a given string P, occ is the number of occurrences of P in T, σ is the alphabet size of T, $w = \Theta(\log n)$ is the machine word size, r is the number of runs in the RLBWT of T, s is a parameter, g is the size of a compressed grammar deriving T, b is the number of input positions for the bookmarking problem, $G = \max\{1, \log^* g - \log^*(\frac{g}{b} - \frac{b}{g})\}$, D is a set of strings of total length n, occ' is the number of strings in D such that each string has P as a prefix and r' is the number of runs in the RLBWT of a string made by concatenating the strings in D.

(i) Locate query	Space (words)	Time
RLFM-index [18]	$O(r + n/s)$	$O((m + s \cdot occ)(\frac{\log \sigma}{\log\log r} + (\log\log n)^2))$
r-index [13]	$O(r)$	$O(m \log\log_w(\sigma + (n/r)) + occ \log\log_w(n/r))$
	$O(r \log\log_w(\sigma + (n/r)))$	$O(m + occ)$
	$O(rw \log_\sigma \log_w n)$	$O(\lceil m \log(\sigma)/w \rceil + occ)$
OptBWTR	$O(r)$	$O(m \log\log_w \sigma + occ)$

(ii) Count query	Space (words)	Time
RLFM-index [18]	$O(r)$	$O(m(\frac{\log \sigma}{\log\log r} + (\log\log n)^2))$
r-index [13]	$O(r)$	$O(m \log\log_w(\sigma + (n/r)))$
	$O(r \log\log_w(\sigma + (n/r)))$	$O(m)$
	$O(rw \log_\sigma \log_w n)$	$O(\lceil m \log(\sigma)/w \rceil)$
OptBWTR	$O(r)$	$O(m \log\log_w \sigma)$

(iii) Extract query	Space (words)	Time per character	Overhead
Gagie et al. [12]	$O(g + b \log^* n)$	$O(1)$	-
Gagie et al. [13]	$O(r \log(n/r))$	$O(\log(\sigma)/w)$	$O(\log(n/r))$
Cording et al. [9]	$O((g + b)G)$	$O(1)$	-
OptBWTR	$O(r + b)$	$O(1)$	-

(iv) Decompression	Space (words)	Time
Lauther and Lukovszki [17]	$O(n(\log\log n + \log \sigma)/w)$	$O(n)$
Golynski et al. [14]	$O((n \log \sigma)/w)$	$O(n \log\log \sigma)$
Predecessor queries [4]	$O(r)$	$O(n \log\log_w(n/r))$
OptBWTR	$O(r)$	$O(n)$

(v) Prefix search	Space (words)	Time
Compact trie [20]	$(n \log \sigma)/w + O(\lvert D \rvert)$	$O(m + occ')$
Z-fast trie [2]	$(n \log \sigma)/w + O(\lvert D \rvert)$	expected $O(\lceil \frac{m \log(\sigma)}{w} \rceil + \log m + \log\log \sigma + occ')$
Packed c-trie [25]	$(n \log \sigma)/w + O(\lvert D \rvert)$	expected $O(\lceil \frac{m \log(\sigma)}{w} \rceil + \log\log n + occ')$
c-trie++ [26]	$(n \log \sigma)/w + O(\lvert D \rvert)$	expected $O(\lceil \frac{m \log(\sigma)}{w} \rceil + \log\log_\sigma w + occ')$
OptBWTR	$O(r' + \lvert D \rvert)$	$O(m + occ')$

2 Preliminaries

Let $\Sigma = \{1, 2, \ldots, \sigma\}$ be an ordered alphabet of size σ, T be a string of length n over Σ, and $|T|$ be the length of T. Let $T[i]$ be the i-th character of T (i.e., $T = T[1], T[2], \ldots, T[n]$) and $T[i..j]$ be the substring of T that begins at position i and ends at position j. For two strings, T and P, $T \prec P$ means that T is lexicographically smaller than P. Let ε be the empty string, i.e., $|\varepsilon| = 0$. We assume that (i) $\sigma = n^{O(1)}$ and (ii) the last character of string T is a special character \$ not occurring on substring $T[1..n-1]$ such that $\$ \prec c$ holds for any character $c \in \Sigma \setminus \{\$\}$. For two integers, b and e ($b \leq e$), *interval* $[b, e]$ is the set $\{b, b+1, \ldots, e\}$. $Occ(T, P)$ denotes all the occurrence positions of a string P in a string T, i.e., $Occ(T, P) = \{i \mid i \in [1, n-|P|+1] \text{ s.t. } P = T[i..(i+|P|-1)]\}$. A *count query* on a string T returns the number of occurrences of a given string P in T, i.e., $|Occ(T, P)|$. Similarly, a *locate query* on string T returns all the starting positions of P in T, i.e., $Occ(T, P)$.

A *rank* query $\mathsf{rank}(T, c, i)$ on a string T returns the number of occurrences of a character c in $T[1..i]$, i.e., $\mathsf{rank}(T, c, i) = |Occ(T[1..i], c)|$. A *select* query $\mathsf{select}(T, c, i)$ on a string T returns the i-th occurrence of c in T (i.e., it returns the smallest integer $j \geq 1$ such that $|Occ(T[1..j], c)| = i$) if T contains c; otherwise it returns -1. Assume that $T[b..e]$ contains a character c for an interval $[b, e] \subseteq [1, n]$. Let $\hat{b}$ and $\hat{e}$ be the first and last occurrences of a character c in $T[b..e]$ (i.e., $\hat{b} = \min\{i \mid i \in [b, e] \text{ s.t. } T[i] = c\}$ and $\hat{e} = \max\{i \mid i \in [b, e] \text{ s.t. } T[i] = c\}$). Then, we can compute $\hat{b}$ and $\hat{e}$ by the following lemma.

▶ **Lemma 1.** *The following statements hold: (i) $T[b..e]$ contains a character c if and only if $\mathsf{rank}(T, c, e) - \mathsf{rank}(T, c, b - 1) \geq 1$ holds. (ii) $\hat{b} = \mathsf{select}(T, c, \mathsf{rank}(T, c, b - 1) + 1)$ and $\hat{e} = \mathsf{select}(T, c, \mathsf{rank}(T, c, e))$ hold if $T[b..e]$ contains c.*

A suffix array (SA) of a string T is an integer array of size n such that $\mathsf{SA}[i]$ stores the starting position of the i-th suffix of T in lexicographical order. Formally, SA is a permutation of $[1, n]$ such that $T[\mathsf{SA}[1]..n] \prec \cdots \prec T[\mathsf{SA}[n]..n]$ holds. Each value in SA is called an *sa-value.*

The *suffix array interval* (*sa-interval*) of a string P is an interval $[b, e] \subseteq [1, n]$ such that $\mathsf{SA}[b..e]$ represents all the occurrence positions of P in string T, i.e., $Occ(T, P) = \{\mathsf{SA}[b], \mathsf{SA}[b + 1], \ldots, \mathsf{SA}[e]\}$. The sa-interval of the empty string ε is defined as $[1, n]$.

LF is a function that returns the position with sa-value $\mathsf{SA}[i] - 1$ on SA (i.e., $\mathsf{SA}[\mathsf{LF}(i)] = \mathsf{SA}[i] - 1$) for a given integer $i \in [1, n]$ if $\mathsf{SA}[i] \neq 1$; otherwise, it returns the position with sa-value n (i.e., $\mathsf{SA}[\mathsf{LF}(i)] = n$). ϕ^{-1} [15] is a function that returns $\mathsf{SA}[i + 1]$ for a given sa-value $\mathsf{SA}[i] \in [1, n]$ (i.e., $\phi^{-1}(\mathsf{SA}[i]) = \mathsf{SA}[i + 1]$) if $i \neq n$; otherwise, it returns $\mathsf{SA}[1]$.

We will use base-2 logarithms throughout this paper unless indicated otherwise. Our computation model is a unit-cost word RAM with a machine word size of $w = \Theta(\log n)$ bits. We evaluate the space complexity in terms of the number of machine words. A bitwise evaluation of space complexity can be obtained with a $\log n$ multiplicative factor.

2.1 Rank-select data structure

We describe a set $\{c_1, c_2, \ldots, c_{\sigma'}\}$ and function γ for a string T. $c_1, c_2, \ldots, c_{\sigma'}$ are all the distinct characters in T, i.e., $\{c_1, c_2, \ldots, c_{\sigma'}\} = \{T[i] \mid i \in [1, |T|]\}$ $(c_1 < c_2 < \cdots < c_{\sigma'})$. The function γ returns the rank of a given character $c \in \Sigma$ in a string T; i.e., $\gamma(T, c) = j$ if there exists an integer j such that $c = c_j$ holds; otherwise $\gamma(T, c) = -1$.

A rank-select data structure $R(T)$ consists of three data structures R_{rank}, R_{select}, and R_{map}. R_{rank} is a *rank data structure* for solving a rank query on a string T in $O(\log\log_w \sigma)$ time and with $O(|T|)$ words of space [4]. R_{select} consists of σ' arrays $H_1, H_2, \ldots, H_{\sigma'}$. The size of H_j is $|Occ(T, c_j)|$ for each $j \in \{1, 2, \ldots, \sigma'\}$, and $H_j[i]$ stores $\mathsf{select}(T, c_j, i)$ for each $i \in [1, |Occ(T, c_j)|]$. R_{map} is a *deterministic dictionary* [24] storing the mapping function γ for T. The deterministic dictionary can compute $\gamma(T, c)$ for a given character c in constant time, and its space usage is $O(\sigma')$ words. The space usage of the rank-select data structure is $O(|T|)$ words in total, because $\sigma' \leq |T|$ holds. We can compute a given select query $\mathsf{select}(T, c, i)$ in two steps: (i) compute $j = \gamma(T, c)$; and (ii) return -1 if $j = -1$ or $|H_j| < i$; otherwise, return $H_j[i]$. Hence, the rank-select data structure can support rank and select queries on T in $O(\log\log_w \sigma)$ and $O(1)$ time, respectively.

2.2 BWT and run-length BWT (RLBWT)

The BWT [5] of a string T is a string L of length n built by permuting T as follows: (i) all n circular strings of T (i.e., $T[1..n]$, $T[2..n]T[1]$, $T[3..n]T[1..2]$, $\ldots$, $T[n]T[2..n-1]$) are sorted in lexicographical order; (ii) $L[i]$ is the last character at the i-th circular string in

			Sorted circular strings		
i	SA	LF	F		L
1	15	10	\$	baababaabaaba	b
2	7	11	a	abaabab\$baaba	b
3	10	12	a	abab\$baababaa	b
4	2	13	a	ababaabaabab\$	b
5	13	14	a	b\$baababaabaa	b
6	5	15	a	baabaabab\$baa	b
7	8	2	a	baabab\$baabab	a
8	11	3	a	bab\$baababaab	a
9	3	4	a	babaabaabab\$b	a
10	14	5	b	\$baababaabaab	a
11	6	6	b	aabaabab\$baab	a
12	9	7	b	aabab\$baababa	a
13	1	1	b	aababaabaabab	\$
14	12	8	b	ab\$baababaaba	a
15	4	9	b	abaabaabab\$ba	a

Figure 1 Table illustrating the BWT (L), SA, LF function, F, and the sorted circular strings of $T = baababaabaabab\$$.

the sorted order for $i \in [1, n]$. Similarly, F is a string of length n such that $F[i]$ is the first character at the i-th circular string in the sorted order. Formally, let $L[i] = T[\mathsf{SA}[\mathsf{LF}(i)]]$ and $F[i] = T[\mathsf{SA}[i]]$.

Let C be an array of size σ such that $C[c]$ is the number of occurrences of characters lexicographically smaller than $c \in \Sigma$ in string T i.e., $C[c] = |\{i \mid i \in [1, n] \text{ s.t. } T[i] \prec c\}|$. The BWT has the following property. For any integer $i \in [1, n]$, $\mathsf{LF}(i)$ is equal to the number of characters that are lexicographically smaller than the character $L[i]$ plus the rank of $L[i]$ on the BWT. Thus, $\mathsf{LF}(i) = C[c] + \mathsf{rank}(L, c, i)$ holds for $c = L[i]$. This is because $\mathsf{LF}(i) < \mathsf{LF}(j)$ if and only if either of the following conditions holds: (i) $L[i] \prec L[j]$ or (ii) $L[i] = L[j]$ and $i < j$ for two integers $1 \leq i < j \leq n$.

Let $[b, e]$ be the sa-interval of a string P and $[b', e']$ be the sa-interval of cP for a character c. Then, the following relation holds between $[b, e]$ and $[b', e']$ on the BWT L.

▶ **Lemma 2** (e.g., [10]). *Let $\hat{b}$ and $\hat{e}$ be the first and last occurrences of c in $L[b..e]$ (i.e., $\hat{b} = \min\{i \mid i \in [b, e] \text{ s.t. } L[i] = c\}$ and $\hat{e} = \max\{i \mid i \in [b, e] \text{ s.t. } L[i] = c\}$). Then, $b' = \mathsf{LF}(\hat{b})$, $e' = \mathsf{LF}(\hat{e})$, and $\mathsf{SA}[b'] = \mathsf{SA}[\hat{b}] - 1$ hold if P and cP are substrings of T.*

Figure 1 illustrates the BWT, SA, LF function, F, L and sorted circular strings of a string $T = baababaabaabab\$$. For example, let $P = ab$, $c = b$. Then $[b, e] = [5, 9]$, $[b', e'] = [14, 15]$, $\hat{b} = 5$, and $\hat{e} = 6$ (see also Figure 1). Moreover, $b' = \mathsf{LF}(\hat{b})$ and $e' = \mathsf{LF}(\hat{e})$ hold by Lemma 2.

The RLBWT of T is a BWT encoded by run-length encoding; i.e., it is a partition of L into r substrings $\mathsf{rlbwt}(L) = L_1, L_2, \ldots, L_r$ such that each substring L_i is a maximal repetition of the same character in L (i.e., $L_i[1] = L_i[2] = \cdots = L_i[|L_i|]$ and $L_{i-1}[1] \neq L_i[1] \neq L_{i+1}[1]$). Each L_i is called a *run*. Let ℓ_i be the starting position of the i-th run of BWT L, i.e., $\ell_1 = 1$, $\ell_i = \ell_{i-1} + |L_{i-1}|$ for $i \in [2, r]$. Let $\ell_{r+1} = n + 1$. The RLBWT is represented as r pairs $(L_1[1], \ell_1)$, $(L_2[1], \ell_2)$, $\ldots$, $(L_r[1], \ell_r)$ using $2r$ words. For example, $\mathsf{rlbwt}(L) = bbbbbb, aaaaaa, \$, aa$ for BWT L illustrated in Figure 1. The RLBWT is represented as $(b, 1), (a, 7), (\$, 13)$, and $(a, 14)$.

Let δ be a permutation of $[1, r]$ satisfying $\mathsf{LF}(\ell_{\delta[1]}) < \mathsf{LF}(\ell_{\delta[2]}) < \cdots < \mathsf{LF}(\ell_{\delta[r]})$. The LF function has the following properties on RLBWT.

▶ **Lemma 3** (e.g., Lemma 2.1 in [16]). *The following two statements hold: (i) Let x be the integer satisfying $\ell_x \leq i < \ell_{x+1}$ for some $i \in [1, n]$. Then,* $\mathsf{LF}(i) = \mathsf{LF}(\ell_x) + (i - \ell_x)$*; (ii)* $\mathsf{LF}(\ell_{\delta[1]}) = 1$ *and* $\mathsf{LF}(\ell_{\delta[i]}) = \mathsf{LF}(\ell_{\delta[i-1]}) + |L_{\delta[i-1]}|$ *for all $i \in [2, r]$.*

Proof. (i) Let $y = (i - \ell_x)$ and $c = L[\ell_x + (i - \ell_x)]$. $\mathsf{LF}(\ell_x + y) = C[c] + \mathsf{rank}(L, c, \ell_x + y)$ holds by the BWT property. $\mathsf{rank}(L, c, \ell_x + y) = \mathsf{rank}(L, c, \ell_x) + y$ holds because the x-th run L_x is a repetition of the character c. Hence $\mathsf{LF}(\ell_x + y) = C[c] + \mathsf{rank}(L, c, \ell_x + y) = C[c] + \mathsf{rank}(L, c, \ell_x) + y = \mathsf{LF}(\ell_x) + y$ holds. By $i = \ell_x + y$, $\mathsf{LF}(i) = \mathsf{LF}(\ell_x) + (i - \ell_x)$ holds.

(ii) Clearly, $\mathsf{LF}(\ell_{\delta[1]}) = 1$. Next, $\mathsf{LF}(\ell_{\delta[i]}) = \mathsf{LF}(\ell_{\delta[i-1]}) + |L_{\delta[i-1]}|$ holds for any $i \in [2, r]$, because (a) the LF function maps the interval $[\ell_{\delta[i]}, \ell_{\delta[i]} + |L_{\delta[i]}| - 1]$ into the interval $[\mathsf{LF}(\ell_{\delta[i]}), \mathsf{LF}(\ell_{\delta[i]}) + |L_{\delta[i]}| - 1]$ by Lemma 3(i) for any $i \in [1, r]$, (b) LF is a bijection from $[1, n]$ to $[1, n]$, and (c) $\mathsf{LF}(\ell_{\delta[1]}) < \mathsf{LF}(\ell_{\delta[2]}) < \cdots < \mathsf{LF}(\ell_{\delta[r]})$ holds. ◀

The sequence $u_1, u_2, \ldots, u_{r+1}$ consists of sa-values such that (i) $\{u_1, u_2, \ldots, u_r\} = \{\mathsf{SA}[\ell_1 + |L_1| - 1], \mathsf{SA}[\ell_2 + |L_2| - 1], \ldots, \mathsf{SA}[\ell_r + |L_r| - 1]\}$, and (ii) $u_1 < u_2 < \cdots < u_r$. Let δ' be a permutation of $[1, r]$ satisfying $\phi^{-1}(u_{\delta'[1]}) < \phi^{-1}(u_{\delta'[2]}) < \cdots < \phi^{-1}(u_{\delta'[r]})$, and let $u_{r+1} = n + 1$. ϕ^{-1} has the following properties on RLBWT.

▶ **Lemma 4** (Lemma 3.5 in [13]). *The following three statements hold: (i) Let x be the integer satisfying $u_x \leq i < u_{x+1}$ for some integer $i \in [1, n]$. Then $\phi^{-1}(i) = \phi^{-1}(u_x) + (i - u_x)$; (ii) $\phi^{-1}(u_{\delta'[1]}) = 1$ and $\phi^{-1}(u_{\delta'[i]}) = \phi^{-1}(u_{\delta'[i-1]}) + d$ for all $i \in [2, r]$, where $d = u_{\delta'[i-1]+1} - u_{\delta'[i-1]}$; (iii) $u_1 = 1$.*

Proof. (i) Lemma 4(i) clearly holds for $i = u_x$. We show that Lemma 4(i) holds for $i \neq u_x$ (i.e., $i > u_x$). Let s_t be the position with sa-value $u_x + t$ for an integer $t \in [1, y]$ (i.e., $\mathsf{SA}[s_t] = u_x + t$), where $y = i - u_x$. s_t is not the ending position of a run (i.e., $(u_x + t) \notin \{u_1, u_2, \ldots, u_r\}$), and thus, two adjacent positions s_t and $s_t + 1$ are contained in an interval $[\ell_v, \ell_v + |L_v| - 1]$ on SA (i.e., $s_t, s_t + 1 \in [\ell_v, \ell_v + |L_v| - 1]$), which corresponds to the v-th run L_v of L. The LF function maps s_t into s_{t-1}, where s_0 is the position with sa-value u_x. LF also maps $s_t + 1$ into $s_{t-1} + 1$ by Lemma 3(i). The two mapping relationships established by LF produce y equalities $\phi^{-1}(\mathsf{SA}[s_1]) = \phi^{-1}(\mathsf{SA}[s_0]) + 1$, $\phi^{-1}(\mathsf{SA}[s_2]) = \phi^{-1}(\mathsf{SA}[s_1]) + 1$, ..., $\phi^{-1}(\mathsf{SA}[s_y]) = \phi^{-1}(\mathsf{SA}[s_{y-1}]) + 1$. The equalities lead to $\phi^{-1}(\mathsf{SA}[s_y]) = \phi^{-1}(\mathsf{SA}[s_0]) + y$, which represents $\phi^{-1}(i) = \phi^{-1}(u_x) + (i - u_x)$ by $\mathsf{SA}[s_y] = i$, $\mathsf{SA}[s_0] = u_x$, and $y = i - u_x$.

(ii) Clearly, $\phi^{-1}(u_{\delta'[1]}) = 1$. $\phi^{-1}(u_{\delta'[i]}) = \phi^{-1}(u_{\delta'[i-1]}) + d$ holds for any $i \in [2, r]$, because (a) ϕ^{-1} maps the interval $[u_{\delta'[i]}, u_{\delta'[i]} + d - 1]$ into the interval $[\phi^{-1}(u_{\delta'[i]}), \phi^{-1}(u_{\delta'[i]}) + d - 1]$ by Lemma 4(i) for any $i \in [1, r]$, (b) ϕ^{-1} is a bijection from $[1, n]$ to $[1, n]$, and (c) $\phi^{-1}(u_{\delta'[1]}) < \phi^{-1}(u_{\delta'[2]}) < \cdots < \phi^{-1}(u_{\delta'[r]})$ holds.

(iii) Let p be the integer satisfying $L_p = \$$. Then there exists an integer q' such that $u_{q'}$ is the sa-value at position ℓ_p, because the length of L_p is 1. Hence, $u_1 = u_{q'} = 1$ holds. ◀

Here, we give an example of Lemma 3. In Figure 1, $(\ell_1, \ell_2, \ell_3, \ell_4) = (1, 7, 13, 14)$ and $(\mathsf{LF}(\ell_1), \mathsf{LF}(\ell_2), \mathsf{LF}(\ell_3), \mathsf{LF}(\ell_4)) = (10, 2, 1, 8)$. Hence, $\mathsf{LF}(3) = \mathsf{LF}(\ell_1) + (3 - \ell_1) = 12$ and $\mathsf{LF}(8) = \mathsf{LF}(\ell_2) + (8 - \ell_2) = 3$ hold by Lemma 3(i).

Next, we give an example of Lemma 4. In Figure 1, $(u_1, u_2, u_3, u_4) = (1, 4, 5, 9)$ and $(\phi^{-1}(u_1), \phi^{-1}(u_2), \phi^{-1}(u_3), \phi^{-1}(u_4)) = (12, 15, 8, 1)$. Hence $\phi^{-1}(3) = \phi^{-1}(u_1) + (3 - u_1) = 14$ and $\phi^{-1}(8) = \phi^{-1}(u_3) + (8 - u_3) = 11$ hold by Lemma 4(i).

3 Novel data structures for computing LF and ϕ^{-1} functions

In this section, we present two new data structures for computing LF and ϕ^{-1} functions in constant time with $O(r)$ words of space. Our key idea is to (i) divide the domains and ranges of two functions into at least r non-overlapping intervals on RLBWT and (ii) compute two

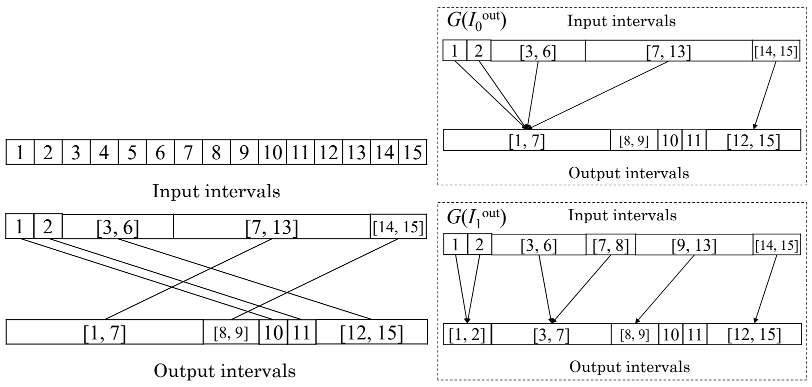

Figure 2 Left figure illustrates input and output intervals created by $I = (1, 10)$, $(2, 11)$, $(3, 12)$, $(7, 1)$, $(14, 8)$. The i-th input and output intervals are connected by a black line. Right figure illustrates two permutation graphs $G(I_0^{\mathsf{out}})$ and $G(I_1^{\mathsf{out}})$ for I.

functions for each domain and range by a linear search in constant time. First, we introduce a notion named *disjoint interval sequence* that is used for a function with non-overlapping intervals for its domain and range. Then, we present a *move query* for computing a function on each disjoint interval sequence and a novel data structure for efficiently computing move queries. Finally, we show that LF and ϕ^{-1} can be computed on two disjoint interval sequences using move queries.

3.1 Disjoint interval sequence and move query

Let $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ be a sequence of k pairs of integers. We introduce a permutation π of $[1, k]$ and sequence $d_1, d_2, \ldots, d_k$ for I. π satisfies $q_{\pi[1]} \leq q_{\pi[2]} \leq \cdots \leq q_{\pi[k]}$, and $d_i = p_{i+1} - p_i$ for $i \in [1, k]$, where $p_{k+1} = n + 1$. We call the sequence I a *disjoint interval sequence* if it satisfies the following three conditions: (i) $p_1 = 1 < p_2 < \cdots < p_k \leq n$ holds, (ii) $q_{\pi[1]} = 1$, and (iii) $q_{\pi[i]} = q_{\pi[i-1]} + d_{\pi[i-1]}$ holds for each $i \in [2, k]$.

We call the two intervals $[p_i, p_i + d_i - 1]$ and $[q_i, q_i + d_i - 1]$ the i-th *input and output intervals* of the disjoint interval sequence I, respectively, for each $i \in [1, k]$. The input intervals $[p_1, p_1 + d_1 - 1]$, $[p_2, p_2 + d_2 - 1]$, …, $[p_k, p_k + d_k - 1]$ do not overlap, i.e., $[p_i, p_i + d_i - 1] \cap [p_j, p_j + d_j - 1] = \emptyset$ holds for any pair of two distinct integers $i, j \in [1, k]$. Hence, the union of the input intervals is equal to the interval $[1, n]$, i.e., $\bigcup_{i=1}^{k} [p_i, p_i + d_i - 1] = [1, n]$. Similarly, the output intervals $[q_1, q_1 + d_1 - 1]$, $[q_2, q_2 + d_2 - 1]$, …, $[q_k, q_k + d_k - 1]$) do not overlap, and their union is equal to $[1, n]$.

A move query $\mathsf{Move}(I, i, x)$ returns a pair (i', x') on a disjoint interval sequence I for a position $i \in [1, n]$ and the index x of the input interval of I containing the position i (i.e., x is the integer satisfying $i \in [p_x, p_x + d_x - 1]$). Here, $i' = q_x + (i - p_x)$ and x' is the index of the input interval of I containing i'. We can represent a bijective function using a disjoint interval sequence and move query. Formally, let $f_I(i) = i'$ for an integer $i \in [1, n]$, where i' is the first value of the pair outputted by $\mathsf{Move}(I, i, x)$. f_I maps the j-th input interval into the j-th output interval (i.e., $f_I(i) = q_j + (i - p_j)$ for $i \in [p_j, p_j + d_j - 1]$). Hence, f_I is a bijective function from $[1, n]$ to $[1, n]$.

In Figure 2, the left figure illustrates the input and output intervals of the disjoint interval sequence $I = (1, 10)$, $(2, 11)$, $(3, 12)$, $(7, 1)$, $(14, 8)$, where $n = 15$. The input intervals created by I are $[1, 1], [2, 2], [3, 6], [7, 13]$, and $[14, 15]$. The output intervals created by I are $[10, 10], [11, 11], [12, 15], [1, 7]$, and $[8, 9]$. For example, $\mathsf{Move}(I, 3, 3) = (12, 4)$, $\mathsf{Move}(I, 5, 3) = (14, 5)$, and $\mathsf{Move}(I, 8, 4) = (2, 2)$.

3.2 Move data structure

In this section, we present a data structure called *move data structure* for computing move queries in constant time. To do so, we introduce three notions, i.e., the *permutation graph*, *split interval sequence*, and *balanced interval sequence.* A permutation graph $G(I)$ is a directed graph for a disjoint interval sequence I. The number of nodes in $G(I)$ is $2k$, and the nodes correspond one-by-one with the input and output intervals of I. Each input interval $[p_i, p_i + d_i - 1]$ has a single outgoing edge pointing to the output interval $[q_j, q_j + d_j - 1]$ containing p_i; i.e., j is the integer satisfying $p_i \in [q_j, q_j + d_j - 1]$. Hence, $G(I)$ has k edges. We say that I is *out-balanced* if every output interval has at most three incoming edges.

A split interval sequence I^{out}_t is a disjoint interval sequence for a disjoint interval sequence I and an integer $t \geq 0$. Let $I^{\mathsf{out}}_0 = I$. For $t \geq 1$, we define I^{out}_t using I^{out}_{t-1} and two integers j, d if I^{out}_{t-1} is not out-balanced. Let (i) $I^{\mathsf{out}}_{t-1} = (p'_1, q'_1), (p'_2, q'_2), \ldots, (p'_{k'}, q'_{k'})$, (ii) j be the smallest integer such that the j-th output interval of I^{out}_{t-1} has at least four incoming edges in $G(I^{\mathsf{out}}_{t-1})$, and (iii) d be the largest integer satisfying $|[q_j, q_j + d - 1] \cap \{p_1, p_2, \ldots, p_{k'}\}| = 2$. Then, I^{out}_t is defined as (p'_1, q'_1), (p'_2, q'_2), …, (p'_{j-1}, q'_{j-1}), (p'_j, q'_j), $(p'_j + d, q'_j + d)$, …, $(p'_{k'}, q'_{k'})$. In other words, I^{out}_t is created by splitting the j-th pair (p'_j, q'_j) of I^{out}_{t-1} into two pairs (p'_j, q'_j) and $(p'_j + d, q'_j + d)$. Let $\tau \geq 0$ be the smallest integer such that I^{out}_τ is out-balanced.

In Figure 2, the right figure illustrates two permutation graphs $G(I^{\mathsf{out}}_0)$ and $G(I^{\mathsf{out}}_1)$, where I is the disjoint interval sequence illustrated in the left figure, i.e., $I = (1, 10)$, $(2, 11)$, $(3, 12)$, $(7, 1)$, $(14, 8)$. The fourth output interval $[1, 7]$ of I^{out}_0 has four incoming edges, and the other output intervals have at most one incoming edge in $G(I^{\mathsf{out}}_0)$. Hence, $I^{\mathsf{out}}_1 = (1, 10)$, $(2, 11)$, $(3, 12)$, $(7, 1)$, $(9, 3)$, $(14, 8)$ holds by $j = 4$ and $d = 2$. I^{out}_1 is out-balanced, and hence $\tau = 1$ holds.

The split interval sequence has the following four properties for each $t \in [0, \tau]$: (i) I^{out}_t consists of $k + t$ pairs. (ii) I^{out}_t consists of at least $2t$ pairs. (iii) Let $d'_i = p'_{i+1} - p'_i$ for $i \in [1, k']$ and $p'_{k'+1} = n + 1$. Both output intervals $[q'_j, q'_j + d - 1]$ and $[q'_j + d, q'_j + d'_j - 1]$ have at least two incoming edges in $G(I^{\mathsf{out}}_t)$. (iv) Let f_I and f^t_I be the two bijective functions represented by I and I^{out}_t, respectively. Then, $f_I(i) = f^t_I(i)$ holds for $i \in [1, n]$. Formally, we obtain the second property from the following lemma.

▶ **Lemma 5.** *$|I^{\mathsf{out}}_t| \geq 2t$ holds for any $t \in [0, \tau]$.*

Proof. Let $\mathcal{Q}(I^{\mathsf{out}}_{t-1})$ be the set of the starting positions of input intervals in $G(I^{\mathsf{out}}_{t-1})$ (i.e., $\mathcal{Q}(I^{\mathsf{out}}_{t-1}) = \{p'_1, p'_2, \ldots, p'_{k'}\}$). Then $\mathcal{Q}(I^{\mathsf{out}}_t) = \mathcal{Q}(I^{\mathsf{out}}_{t-1}) \cup \{p'_j + d\}$ holds from the definition of I^{out}_t. Next, let $\mathsf{Edge}_2(I^{\mathsf{out}}_{t-1})$ be the set of output intervals such that each output interval has at least two incoming edges in $G(I^{\mathsf{out}}_{t-1})$, i.e., $\mathsf{Edge}_2(I^{\mathsf{out}}_{t-1}) = \{[q'_i, q'_i + d'_i - 1] \mid i \in [1, k'] \text{ s.t. } |[q'_i, q'_i + d'_i - 1] \cap \mathcal{Q}(I^{\mathsf{out}}_{t-1})| \geq 2\}$, where $d'_i = p'_{i+1} - p'_i$. $[q'_i, q'_i + d'_i - 1] \in \mathsf{Edge}_2(I^{\mathsf{out}}_t)$ holds if $[q'_i, q'_i + d'_i - 1] \in \mathsf{Edge}_2(I^{\mathsf{out}}_{t-1})$ for any integer $i \in [1, k'] \setminus \{j\}$. This is because (i) $[q'_i, q'_i + d'_i - 1]$ is also an output interval of I^{out}_t, and (ii) $([q'_i, q'_i + d'_i - 1] \cap \mathcal{Q}(I^{\mathsf{out}}_{t-1})) \subseteq ([q'_i, q'_i + d'_i - 1] \cap \mathcal{Q}(I^{\mathsf{out}}_t))$ holds by $\mathcal{Q}(I^{\mathsf{out}}_{t-1}) \subseteq \mathcal{Q}(I^{\mathsf{out}}_t)$. $[q'_j, q'_j + d - 1], [q'_j + d, q'_{j+1} - 1] \in \mathsf{Edge}_2(I^{\mathsf{out}}_t)$ also holds by the third property of I^{out}_t. Hence, we obtain an inequality $|\mathsf{Edge}_2(I^{\mathsf{out}}_t)| \geq |\mathsf{Edge}_2(I^{\mathsf{out}}_{t-1})| + 1$ for any integer $t \in [1, \tau]$. The inequality $|\mathsf{Edge}_2(I^{\mathsf{out}}_t)| \geq |\mathsf{Edge}_2(I^{\mathsf{out}}_{t-1})| + 1$ guarantees that $|\mathsf{Edge}_2(I^{\mathsf{out}}_t)| \geq t$ holds for any integer $t \in [0, \tau]$. The inequality $|\mathsf{Edge}_2(I^{\mathsf{out}}_t)| \geq t$ indicates that I^{out}_t consists of at least $2t$ pairs, because each output interval in $\mathsf{Edge}_2(I^{\mathsf{out}}_t)$ has at least two incoming edges from distinct input intervals. Hence, Lemma 5 holds. ◀

A balanced interval sequence $B(I)$ is defined as I^{out}_τ for a disjoint interval sequence I. We obtain the lemma below from the four properties of I^{out}_τ.

▶ **Lemma 6.** *Let f_I and $f_{B(I)}$ be the two bijective functions represented by I and $B(I)$, respectively for a disjoint interval sequence I of length k. The following three statements hold: (i) $|B(I)| \leq 2k$, (ii) $B(I)$ is out-balanced, and (iii) the two disjoint interval sequences I and $B(I)$ represent the same bijective function, i.e., $f_I(i) = f_{B(I)}(i)$ for $i \in [1, n]$.*

Proof. (i) We obtain an inequality $\tau \leq k$ from the first and second properties of I^{out}_t, because $k + t \geq 2t$ must hold for any $t \in [0, \tau]$. Hence, I^{out}_τ consists of at most $2k$ pairs; i.e., $|B(I)| \leq 2k$ holds. (ii) I^{out}_τ is out-balanced, and thus, $B(I)$ is out-balanced. (iii) $f_I(i) = f^0_I(i) = f^1_I(i) = \cdots = f^\tau_I(i) = f_{B(I)}(i)$, and thus, $f_I(i) = f_{B(I)}(i)$ for $i \in [1, n]$. ◀

The move data structure $F(I)$ is built on a balanced interval sequence $B(I) = (p_1, q_1)$, (p_2, q_2), …, $(p_{k'}, q_{k'})$ for a disjoint interval sequence I, and it supports move queries on $B(I)$. The move data structure consists of two arrays D_{pair} and D_{index} of size k'. $D_{\mathsf{pair}}[i]$ stores the i-th pair (p_i, q_i) of $B(I)$ for each $i \in [1, k']$. $D_{\mathsf{index}}[i]$ stores the index j of the input interval containing q_i. Hence, the space usage is $O(k')$ words in total.

Now let us describe an algorithm for solving a move query $\mathsf{Move}(B(I), i, x) = (i', x')$ on $B(I)$, where x and x' are the indexes of the two input intervals of $B(I)$ containing i and i', respectively, and $i' = q_x + (i - p_x)$. The algorithm consists of three steps. In the first step, the algorithm computes $i' = q_x + (i - p_x)$. In the second step, the algorithm finds the x'-th input interval by a linear search on the input intervals of $B(I)$. Let $b = D_{\mathsf{index}}[x]$. The linear search starts at the b-th input interval $[p_b, p_{b+1} - 1]$, reads the input intervals in the left-to-right order, and stops if the input interval containing position i' is found (i.e., the x'-th input interval). The linear search is always successful (i.e., $x' \geq b$), because $i' \geq q_x$ holds. In the third step, the algorithm returns the pair (i', x'). The running time of the algorithm is $O(x' - b + 1)$ in total.

The running time is computed as follows. Let i_{beg} and i_{end} be the indexes of the first and last input intervals that are connected to the x-th output interval in $G(B(I))$. The x-th output interval has at most three incoming edges, and hence, $i_{\mathsf{end}} - i_{\mathsf{beg}} + 1 \leq 3$ holds. Since b is the index of an input interval that overlaps the x-th output interval, $i_{\mathsf{beg}} - 1 \leq b \leq i_{\mathsf{end}}$. Similarly, $i_{\mathsf{beg}} - 1 \leq x' \leq i_{\mathsf{end}}$. Therefore, $x' - b \leq 3$ and we can solve the move query in constant time.

3.3 Computing LF and ϕ^{-1} functions using move data structures

Here, we show that we can compute the LF function using a move data structure. Recall that ℓ_i is the starting position of the i-th run on BWT L for $i \in [1, r]$, and δ is the permutation of $[1, r]$ introduced in Section 2.2. The sequence I_{LF} is defined as r pairs $(\ell_1, \mathsf{LF}(\ell_1))$, $(\ell_2, \mathsf{LF}(\ell_2))$, …, $(\ell_r, \mathsf{LF}(\ell_r))$. I_{LF} satisfies the three conditions of a disjoint interval sequence by Lemma 3, i.e., (i) $\ell_1 = 1 < \ell_2 < \cdots < \ell_r \leq n$, (ii) $\mathsf{LF}(\ell_{\delta[1]}) = 1$, and (iii) $\mathsf{LF}(\ell_{\delta[i]}) = \mathsf{LF}(\ell_{\delta[i-1]}) + |L_{\delta[i-1]}|$ holds for each $i \in [2, r]$. Hence I_{LF} is a disjoint interval sequence.

Let f_{LF} be the bijective function represented by the disjoint interval sequence I_{LF}. Then, $f_{\mathsf{LF}}(i) = \mathsf{LF}(\ell_x) + (i - \ell_x)$ holds, where x is the integer such that $\ell_x \leq i < \ell_{x+1}$ holds. On the other hand, we have $\mathsf{LF}(i) = \mathsf{LF}(\ell_x) + (i - \ell_x)$ by Lemma 3(i). Hence, f_{LF} and LF are the same function, i.e., $\mathsf{LF}(i) = f_{\mathsf{LF}}(i)$ for $i \in [1, n]$.

Let $F(I_{\mathsf{LF}})$ be the move data structure built on the balanced interval sequence $B(I_{\mathsf{LF}})$ for I_{LF}. By Lemma 6, the move data structure requires $O(r)$ words of space, and $\mathsf{LF}(i) = i'$ holds for a move query $\mathsf{Move}(B(I), i, x) = (i', x')$ on $B(I_{\mathsf{LF}})$. Hence, we have proven the following theorem.

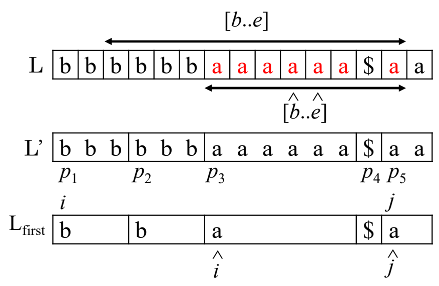

Figure 3 Example of modified toehold lemma.

▶ **Theorem 7.** *Let x and x' be the indexes of the two input intervals of $B(I_{\mathsf{LF}})$ containing an integer $i \in [1, n]$ and $\mathsf{LF}(i)$, respectively. We can compute $\mathsf{LF}(i)$ and x' in constant time by using $F(I_{\mathsf{LF}})$ and (i, x).*

Similarly, we can show that we can compute ϕ^{-1} by using a move data structure. A sequence I_{SA} consists of r pairs $(u_1, \phi^{-1}(u_1))$, $(u_2, \phi^{-1}(u_2))$, $\ldots$, $(u_r, \phi^{-1}(u_r))$, where $u_1, u_2, \ldots, u_r$ are the integers introduced in Section 2.2. I_{SA} has the following three properties: (i) $u_1 = 1 < u_2 < \cdots < u_r \leq n$ holds by Lemma 4(iii), (ii) $\phi^{-1}(u_{\delta'[1]}) = 1$ by Lemma 4(ii), and (iii) $\phi^{-1}(u_{\delta'[i]}) = \phi^{-1}(u_{\delta'[i-1]}) + (u_{\delta'[i-1]+1} - u_{\delta'[i-1]})$ holds by Lemma 4(ii) for each $i \in [2, r]$, where δ' is the permutation of $[1, r]$ introduced in Section 2.2. Hence, I_{SA} satisfies the three conditions of a disjoint interval sequence by Lemma 4.

Let f_{SA} be the bijective function represented by the disjoint interval sequence I_{SA}. Then $f_{\mathsf{SA}}(i) = \phi^{-1}(u_x) + (i - u_x)$ holds, where x is the integer such that $u_x \leq i < u_{x+1}$ holds. On the other hand, $\phi^{-1}(i) = \phi^{-1}(u_x) + (i - u_x)$ holds by Lemma 4(i). Hence f_{SA} and $\phi^{-1}(i)$ are the same function.

Let $F(I_{\mathsf{SA}})$ be the move data structure built on the balanced interval sequence $B(I_{\mathsf{SA}})$ for I_{SA}. Then, the result of a move query on $B(I_{\mathsf{SA}})$ contains $\phi^{-1}(i)$ for $i \in [1, n]$, and hence, we have proven (i) of the following theorem.

▶ **Theorem 8.** *Let x, x', $\hat{x}$ be the indexes of the three input intervals of $B(I_{\mathsf{SA}})$ containing an integer $i \in [1, n]$, $\phi^{-1}(i)$, and $i - 1$, respectively. Then, the following two statements hold: (i) We can compute $\phi^{-1}(i)$ and x' in constant time using data structure $F(I_{\mathsf{SA}})$ and the pair (i, x). (ii) We can compute the index $\hat{x}$ using $F(I_{\mathsf{SA}})$ and (i, x).*

Proof. (ii) Let $B(I_{\mathsf{SA}}) = (p_1, q_1), (p_2, q_2), \ldots, (p_{k'}, q_{k'})$. The x-th input interval is $[p_x, p_{x+1} - 1]$, which contains i. $\hat{x} = x$ holds if $p_x \neq i$; otherwise, $\hat{x} = x - 1$. We can verify $p_x \neq i$ holds in constant time by using $F(I_{\mathsf{SA}})$. ◀

Theorems 7 and 8 indicate that we can compute the position obtained by recursively applying LF and ϕ^{-1} to a position $i \in [1, n]$ t times in $O(t)$ time if we know the index of the input interval containing i. For example, let x, x', x'', and x''' be the indexes of the four input intervals of $B(I_{\mathsf{LF}})$ containing i, $\mathsf{LF}(i)$, $\mathsf{LF}(\mathsf{LF}(i))$, and $\mathsf{LF}(\mathsf{LF}(\mathsf{LF}(i)))$, respectively. $\mathsf{LF}(\mathsf{LF}(\mathsf{LF}(i)))$ can be computed by computing three move queries $\mathsf{Move}(B(I_{\mathsf{LF}}), i, x) = (\mathsf{LF}(i), x')$, $\mathsf{Move}(B(I_{\mathsf{LF}}), \mathsf{LF}(i), x') = (\mathsf{LF}(\mathsf{LF}(i)), x'')$, and $\mathsf{Move}(B(I_{\mathsf{LF}}), \mathsf{LF}(\mathsf{LF}(i)), x'') = (\mathsf{LF}(\mathsf{LF}(\mathsf{LF}(i))), x''')$.

4 New data structure for backward searches

Here, we present a modified version of the *backward search* [10, 1], which we call *backward search query for OptBWTR* (BSR query), for computing the sa-interval of cP for a given string P and character c. To define the BSR query, we will introduce a new tuple: a *balanced* *sa-interval* of a string P is a 6-tuple $(b, e, \mathsf{SA}[b], i, j, v)$. Here, (i) $[b, e]$ is the sa-interval of P; (ii) i and j are the indexes of the two input intervals of $B(I_{\mathsf{LF}})$ containing b and e, respectively; (iii) v is the index of the input interval of $B(I_{\mathsf{SA}})$ containing $\mathsf{SA}[b]$. The balanced sa-interval of P is undefined if the sa-interval of P is $\emptyset$ (i.e., P is not a substring of T). The input of the BSR query is the balanced sa-interval $(b, e, \mathsf{SA}[b], i, j, v)$ of a string P and a character c. The output of the BSR query is the balanced sa-interval $(b', e', \mathsf{SA}[b'], i', j', v')$ of string cP if the sa-interval of cP is not the empty set; otherwise BSR outputs a mark $\perp$.

Now, we will present a data structure called the *BSR data structure*. The BSR data structure supports BSR queries in $O(\log\log_w \sigma)$ time. It consists of five data structures $F(I_{\mathsf{LF}})$, $F(I_{\mathsf{SA}})$, $R(L_{\mathsf{first}})$, SA^+, and $\mathsf{SA}^+_{\mathsf{index}}$. Here, $F(I_{\mathsf{LF}})$ and $F(I_{\mathsf{SA}})$ are the two move data structures introduced in Section 3.3. Let $B(I_{\mathsf{LF}}) = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$. Then L_{first} is the string satisfying $L_{\mathsf{first}} = L[p_1], L[p_2], \ldots, L[p_k]$. $R(L_{\mathsf{first}})$ is a rank-select data structure built on L_{first}, which is defined in Section 2. $R(L_{\mathsf{first}})$ requires $O(|L_{\mathsf{first}}|)$ words of space, and it supports rank and select queries on L_{first} in $O(\log\log_w \sigma)$ and $O(1)$ time, respectively. SA^+ is an array of size k such that $\mathsf{SA}^+[x]$ stores the sa-value at the starting position of the x-th input interval of $B(I_{\mathsf{LF}})$ for each $x \in [1, k]$ (i.e., $\mathsf{SA}^+[x] = \mathsf{SA}[p_x]$). Let $B(I_{\mathsf{SA}}) = (p'_1, q'_1), (p'_2, q'_2), \ldots, (p'_{k'}, q'_{k'})$. $\mathsf{SA}^+_{\mathsf{index}}$ is an array of size k such that $\mathsf{SA}^+_{\mathsf{index}}[x]$ stores the index y of the input interval of $B(I_{\mathsf{SA}})$ containing the position $\mathsf{SA}^+[x]$ (i.e., y is the integer satisfying $\mathsf{SA}^+[x] \in [p'_y, p'_{y+1} - 1]$). The space usage of the five data structures is $O(|B(I_{\mathsf{LF}})| + |B(I_{\mathsf{SA}})|)$ words, and $|B(I_{\mathsf{LF}})|, |B(I_{\mathsf{SA}})| = O(r)$ holds by Lemma 6(i).

Next, we will present a key observation on BSR queries, which is based on the toehold lemma (see, e.g., [23, 13, 1]). Let L' be a sequence of k substrings $L[p_1..p_2 - 1], L[p_2..p_3 - 1], \ldots, L[p_k..p_{k+1} - 1]$ of BWT L, where $p_{k+1} = n + 1$. Then, L' has the following properties: (i) L' represents a partition of L. (ii) Each string of L' consists of a repetition of the same character. (iii) Each character $L_{\mathsf{first}}[t]$ corresponds to the first character of the t-th string of L'. (iv) The i-th and j-th strings of L' contain the b-th and e-th characters of BWT L, respectively. (v) Let $\hat{b}$ and $\hat{e}$ be the first and last occurrences of c in $L[b..e]$ (i.e., $\hat{b} = \min\{t \mid t \in [b, e] \text{ s.t. } L[t] = c\}$ and $\hat{e} = \max\{t \mid t \in [b, e] \text{ s.t. } L[t] = c\}$). Similarly, let $\hat{i}$ and $\hat{j}$ be the indexes of the two strings of L' containing the $\hat{b}$-th and $\hat{e}$-th characters of BWT L, respectively. Then $\hat{i}$ and $\hat{j}$ are equal to the first and last occurrences of c in $L_{\mathsf{first}}[i..j]$. We obtain the following relations among the four positions b, $\hat{b}$, e, and $\hat{e}$ by using the above five properties: (i) $\hat{b} = b$ if $L_{\mathsf{first}}[i] = c$; otherwise, $\hat{b} = p_{\hat{i}}$. (ii) Similarly, $\hat{e} = e$ if $L_{\mathsf{first}}[j] = c$; otherwise $\hat{b} = p_{\hat{j}+1} - 1$. We call these two relations the *modified toehold lemma.*

Let $\hat{v}$ be the index of the input interval of $B(I_{\mathsf{SA}})$ containing position $\mathsf{SA}[\hat{b}]$. $\hat{v} = v$ and $\mathsf{SA}[\hat{b}] = \mathsf{SA}[b]$ hold if $\hat{b} = b$; otherwise, $\hat{v} = \mathsf{SA}^+_{\mathsf{index}}[\hat{i}]$ and $\mathsf{SA}[\hat{b}] = \mathsf{SA}^+[\hat{i}]$ by the modified toehold lemma. We can compute the balanced sa-interval of cP by using $F(I_{\mathsf{LF}})$ and $F(I_{\mathsf{SA}})$ after computing the six integers $\hat{b}, \hat{e}, \hat{i}, \hat{j}, \hat{v}, \mathsf{SA}[\hat{b}]$, because $b' = \mathsf{LF}(\hat{b})$, $e' = \mathsf{LF}(\hat{e})$, and $\mathsf{SA}[b'] = \mathsf{SA}[\hat{b}] - 1$ hold by Lemma 2.

Figure 3 illustrates an example of the modified toehold lemma for a BWT $L = bbbbbb\ aaaaaa\$aa$. In this example, $c = a$ and $L' = bbb, bbb, aaaaaa, \$, aa$. (i) $k = 5$, (ii) $(p_1, p_2, p_3, p_4, p_5) = (1, 4, 7, 13, 14)$, (ii) $L_{\mathsf{first}} = bba\$a$, (iii) $(b, e) = (3, 14)$, (iv) $(\hat{b}, \hat{e}) = (7, 14)$, (v) $(i, j) = (1, 5)$, and (vi) $(\hat{i}, \hat{j}) = (3, 5)$. The i-th string of L' is not a repetition of the character c, and the $\hat{i}$-th string of L' contains the $\hat{b}$-th character of L. Hence $\hat{b} = p_{\hat{i}} = 7$ holds by the modified toehold lemma. Similarly, the j-th string of L' is a repetition of c, and hence $\hat{e} = e$ holds by the modified toehold lemma.

We solve a BSR query in four steps. In the first step, we verify whether $L_{\mathsf{first}}[i..j]$ contains character c by computing two rank queries $\mathsf{rank}(L_{\mathsf{first}}, c, j)$ and $\mathsf{rank}(L_{\mathsf{first}}, c, i)$. By Lemma 1(i), $L_{\mathsf{first}}[i..j]$ contains c if $\mathsf{rank}(L_{\mathsf{first}}, c, j) - \mathsf{rank}(L_{\mathsf{first}}, c, i) \geq 1$; otherwise, cP is not a substring of T, and hence BSR outputs a mark $\perp$. In the second step, we compute two integers $\hat{\imath}$ and $\hat{\jmath}$ using rank and select queries on the string L_{first}. $\hat{\imath} = \mathsf{select}(L_{\mathsf{first}}, c, \mathsf{rank}(L_{\mathsf{first}}, c, i-1)+1)$ and $\hat{\jmath} = \mathsf{select}(L_{\mathsf{first}}, c, \mathsf{rank}(L_{\mathsf{first}}, c, j))$ hold by Lemma 1(ii). In the third step, we compute $\hat{b}$, $\hat{e}$, $\hat{v}$, and $\mathsf{SA}[\hat{b}]$ by the modified toehold lemma. In the fourth step, we compute the balanced sa-interval of cP by processing the six integers $\hat{b}, \hat{e}, \hat{\imath}, \hat{\jmath}, \hat{v}, \mathsf{SA}[\hat{b}]$, i.e., we compute (i) the pair (b', i') using a move query on $B(I_{\mathsf{LF}})$ for the pair $(\hat{b}, \hat{\imath})$, (ii) the pair (e', j') using a move query on $B(I_{\mathsf{LF}})$ for the pair $(\hat{e}, \hat{\jmath})$, and (iii) the pair $(\mathsf{SA}[v'], v')$ by Theorem 8(ii). The running time is $O(\log\log_w \sigma)$ in total.

5 OptBWTR

Here, we present OptBWTR, which supports optimal-time queries for polylogarithmic alphabets by leveraging data structures for computing LF and ϕ^{-1} functions. Let P be a string of length m in a count or locate query and $occ = |Occ(T, P)|$. The goal of this section is to prove the following theorem.

▶ **Theorem 9.** *OptBWTR requires $O(r)$ words, and it supports count and locate queries on a string T in $O(m \log\log_w \sigma)$ and $O(m \log\log_w \sigma + occ)$ time, respectively. We can construct OptBWTR in $O(n + r \log r)$ time and $O(r)$ words by processing the RLBWT of T.*

Proof. See the full version of this paper [22] for the proof of the construction time and working space in Theorem 9. ◀

OptBWTR consists of the five data structures composing the BSR data structure, i.e., $F(I_{\mathsf{LF}})$, $F(I_{\mathsf{SA}})$, $R(L_{\mathsf{first}})$, SA^+, and $\mathsf{SA}^+_{\mathsf{index}}$. First, we present an algorithm for a count query using OptBWTR that consists of two phases. In the first phase, the algorithm computes the balanced sa-interval of P by iterating BSR query m times. The input of the i-th BSR query is the $(m-i+1)$-th character of P (i.e., $P[m-i+1]$) and the balanced sa-interval of $P[m-i+2..m]$ for each $i \in [1, m]$. Here, $P[m+1..m]$ is defined as the empty string ε. The balanced sa-interval of ε is $(1, n, n, 1, |B(I_{\mathsf{LF}})|, |B(I_{\mathsf{SA}})|)$, because (i) the sa-interval of the empty string is $[1, n]$, and (ii) $\mathsf{SA}[1] = n$. The i-th BSR query outputs the balanced sa-interval of $P[m-i+1..m]$ if $P[m-i+1..m]$ is a substring of T; otherwise it outputs a mark $\perp$. If a BSR query outputs $\perp$, the pattern P does not occur in T. In this case, the algorithm stops and returns 0 as the solution for the count query. In the second phase, the algorithm returns the length of the sa-interval $[b, e]$ of P (i.e., $e-b+1$) as the solution for the count query, because $occ = e-b+1$ holds. The sa-interval of P is contained in the balanced sa-interval of P; hence, the running time is $O(m \log\log_w \sigma)$ in total.

Next, we present an algorithm for a locate query using OptBWTR. Assume that we already computed the balanced sa-interval of P by the algorithm for the count query. Let v_t be the index of the input interval of $B(I_{\mathsf{SA}})$ containing $\mathsf{SA}[b+t]$ for $t \in [0, e-b]$. Then $\mathsf{SA}[b+1..e]$ can be computed by computing $(e-b)$ move queries $\mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[b], v_0) = (\mathsf{SA}[b+1], v_1)$, $\mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[b+1], v_1) = (\mathsf{SA}[b+2], v_2), \ldots, \mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[e-1], v_{e-b}) = (\mathsf{SA}[e], v_{e-b+1})$ on $B(I_{\mathsf{SA}})$. The first sa-value $\mathsf{SA}[b]$ and the index v_0 are stored in the balanced sa-interval of P.

The algorithm for a locate query also consists of two phases. In the first phase, the algorithm computes the balanced sa-interval of P by iterating BSR query m times. In the second phase, it computes $(e-b)$ move queries $\mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[b], v_0)$, $\mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[b+1], v_1)$, ..., $\mathsf{Move}(B(I_{\mathsf{SA}}), \mathsf{SA}[e-1], v_{e-b})$ by using the move data structure $F(I_{\mathsf{SA}})$, and outputs $\mathsf{SA}[b..e]$. Hence, we can solve a locate query in $O(m \log\log_w \sigma + occ)$ time.

6 Applications

In this section, we show that OptBWTR can support extract, decompression, and prefix search queries in optimal time.

Extract query. Let a string T of length n have b marked positions $i_1, i_2, \ldots, i_b \in [1, n]$. An extract query (also called the bookmarking problem) is to return substring $T[i_j..i_j + d - 1]$ for a given integer $j \in [1, b]$ and $d \in [1, n - i_j + 1]$.

We will use *FL function* to solve extract queries. FL is the inverse function of LF function, i.e., $\mathsf{FL}(\mathsf{LF}(i)) = i$ holds for $i \in [1, n]$. We will also use the function FL_x and integers $h_1, h_2, \ldots, h_b$. $\mathsf{FL}_x(i)$ returns the position obtained by recursively applying the FL function to a given integer i x times, i.e., $\mathsf{FL}_0(i) = i$ and $\mathsf{FL}_x(i) = \mathsf{FL}_{x-1}(\mathsf{FL}(i))$ for $x \geq 1$. h_j is the position with sa-value i_j on SA (i.e., $\mathsf{SA}[h_j] = i_j$). The FL function returns the position with the sa-value $y + 1$ on SA for a given position with sa-value y, and hence $T[i_j..i_j + d - 1] = F[\mathsf{FL}_0(h_j)], F[\mathsf{FL}_1(h_j)], \ldots, F[\mathsf{FL}_{d-1}(h_j)]$ holds for $j \in [1, b]$, where F is the string described in Section 2.2. We can construct a data structure of $O(r)$ words to compute FL function in constant time by modifying Theorem 7 and can solve an extract query in linear time by using the data structure. See the full version of this paper [22] for details of our data structure for solving extract queries.

▶ **Theorem 10.** *There exists a data structure of $O(r + b)$ words that solves the bookmarking problem for a string T and b positions $i_1, i_2, \ldots, i_b$ ($1 \leq i_1 < i_2 < \cdots < i_b \leq n$). This data structure supports an exact query in constant time per character. We can construct the data structure in $O(n)$ time and $O(r + b)$ words of space by processing the RLBWT and positions $i_1, i_2, \ldots, i_b$.*

Proof. See the full version of this paper [22]. ◀

Decompression of RLBWT. We apply Theorem 10 to $T[1..n]$ with marked position 1. Then, our data structure for extract queries can return the string T in $O(n)$ time (i.e., the data structure can recover T from the RLBWT of T in linear time to n). The $O(n)$ time decompression is the fastest among other decompression algorithms on compressed indexes in $O(r)$ words of space, as the following theorem shows.

▶ **Theorem 11.** *We can compute the characters of T in left-to-right order (i.e., $T[1], T[2], \ldots, T[n]$) in $O(n)$ time and $O(r)$ words of space by processing the RLBWT of string T.*

Prefix search. The prefix search for a set of strings $D = \{T_1, T_2, \ldots, T_d\}$ returns the indexes of the strings in D that include a given string P as their prefixes (i.e., $\{i \mid i \in [1, d] \text{ s.t. } T_i[1..|P|] = P\}$). We can construct a data structure supporting the prefix search by combining Theorem 10 with *compact trie* [20].

A compact trie for a set of strings D is a trie for D such that all unary paths are collapsed, and each node represents the string by concatenating labels on the path from the root to the node. For simplicity, we assume that the set D is *prefix-free*, i.e., T_i is not a prefix of $T_{i'}$ for any pair of two strings T_i and $T_{i'}$ in D. Each leaf in the compact trie represents a distinct string in D by the assumption. Let v be the node such that (i) P is a prefix of the string represented by the node and (ii) P is not a prefix of the string represented by its parent. Then, the leaves under v are the output of the prefix search query for P.

To find v, we decode the string on the path from the root to the node v in linear time using exact queries for the path. After we find v, we traverse the subtree rooted at v and output all the leaves in the subtree. This procedure runs in $O(|P| + occ')$ time, where occ' is the number of leaves under the lowest node. See the full version of this paper [22] for the details of our data structure for solving prefix search queries.

▶ **Theorem 12.** *Let r' be the number of runs in the RLBWT of a string T containing all the strings in $D = \{T_1, T_2, \dots, T_d\}$. There exists a data structure that supports a prefix search on D in $O(|P| + occ')$ time and $O(r' + d)$ words of space for a string P. The data structure also returns the number of the strings in D that include P as their prefixes in $O(|P|)$ time.*

Proof. See the full version of this paper [22]. ◀

7 Conclusion

We presented OptBWTR, the first string index that can support count and locate queries on RLBWT in optimal time with $O(r)$ words of space for polylogarithmic alphabets. OptBWTR also supports extract queries and prefix searches on RLBWT in optimal time for any alphabet size. In addition, we presented the first decompression algorithm working in optimal time and $O(r)$ words of working space. This is the first optimal-time decompression algorithm working in $O(r)$ words of space.

We presented a new data structure of $O(r)$ words for computing LF and ϕ^{-1} functions in constant time by using a new data structure named move data structure, provided that we use an additional input. We also showed that the backward search works in optimal time for polylogarithmic alphabets with $O(r)$ words of space using the data structure. The two functions and the backward search are general and applicable to various queries on RLBWT.

The following problems remain open: Does there exist a string index of $O(r)$ words supporting locate queries in optimal time for any alphabet size? We assume $\sigma = O(\text{polylog}\, n)$ for supporting locate queries in optimal time with $O(r)$ words. As mentioned in Section 1, a faster version of r-index can support locate queries in optimal time with $O(r \log\log_w(\sigma + (n/r)))$ words. Thus, improving OptBWTR so that it can support locate queries in optimal time with $O(r)$ words for any alphabet size is an important future work. For this goal, one needs to solve a rank query on a string of length $\Theta(r)$ in constant time and $O(r)$ words of space. However, this seems impossible because any data structure of $O(r)$ words requires $\Omega(\log\log_w \sigma)$ time to compute a rank query on a string of length r [4]. Perhaps, we may be able to compute the sa-interval of a given pattern in $O(m)$ time and $O(r)$ words of space without using rank queries. After computing the sa-interval of the pattern, we can solve the locate query in optimal time by using our data structure for the ϕ^{-1} function.

References

1 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

2 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *Proceedings of SPIRE*, pages 159–172, 2010.

3 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10:23:1–23:19, 2014.

4 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11:31:1–31:21, 2015.

5 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical report*, 1994.

6 Anders Roy Christiansen and Mikko Berggren Ettienne. Compressed indexing with signature grammars. In *Proceedings of LATIN*, pages 331–345, 2018.

7 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17:8:1–8:39, 2021.

8 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proceedings of SPIRE*, pages 180–192, 2012.

9 Patrick Hagge Cording, Pawel Gawrychowski, and Oren Weimann. Bookmarks in grammar-compressed strings. In *Proceedings of SPIRE*, pages 153–159, 2016.

10 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52:552–581, 2005.

11 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3:20, 2007.

12 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proceedings of LATIN*, pages 731–742, 2014.

13 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, 2020.

14 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of SODA*, pages 368–373, 2006.

15 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of CPM*, pages 181–192, 2009.

16 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of SODA*, pages 1344–1357, 2019.

17 Ulrich Lauther and Tamás Lukovszki. Space efficient algorithms for the Burrows-Wheeler backtransformation. *Algorithmica*, 58:339–351, 2010.

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

19 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, 1993.

20 Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

21 Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

22 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. *CoRR*, abs/2006.05104, 2021. `arXiv:2006.05104`.

23 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80:1986–2011, 2018.

24 Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In *Proceedings of ICALP*, pages 84–95, 2008.

25 Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100-A:1785–1793, 2017.

26 Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. c-trie++: A dynamic trie tailored for fast prefix searches. In *Proceedings of DCC*, pages 243–252, 2020.