An Optimal-Time RLBWT Construction in BWT-Runs Bounded Space

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

Shunsuke Kanda
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

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

Abstract

The compression of highly repetitive strings (i.e., strings with many repetitions) has been a central research topic in string processing, and quite a few compression methods for these strings have been proposed thus far. Among them, an efficient compression format gathering increasing attention is the run-length Burrows–Wheeler transform (RLBWT), which is a run-length encoded BWT as a reversible permutation of an input string on the lexicographical order of suffixes. State-of-the-art construction algorithms of RLBWT have a serious issue with respect to (i) non-optimal computation time or (ii) a working space that is linearly proportional to the length of an input string. In this paper, we present *r-comp*, the first optimal-time construction algorithm of RLBWT in BWT-runs bounded space. That is, the computational complexity of r-comp is $O(n + r \log r)$ time and $O(r \log n)$ bits of working space for the length n of an input string and the number r of equal-letter runs in BWT. The computation time is optimal (i.e., $O(n)$) for strings with the property $r = O(n/\log n)$, which holds for most highly repetitive strings. Experiments using a real-world dataset of highly repetitive strings show the effectiveness of r-comp with respect to computation time and space.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases lossless data compression, Burrows–Wheeler transform, highly repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.99

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2202.07885`

Supplementary Material *Software (Source Code)*: `https://github.com/kampersanda/rcomp`

1 Introduction

Highly repetitive strings (i.e., strings including many repetitions) have become common in research and industry. For instance, the 1000 Genomes Project [32] was established for the purpose of building a detailed catalogue of human genetic variation, and it has sequenced a large number of human genomes. Nowadays, approximately 60 billion pages are said to exist on the Internet, and large sections of those pages (e.g., version-controlled documents) are highly repetitive. There is therefore a growing demand to develop scalable data compression for efficiently storing, processing, and analyzing a gigantic number of highly repetitive strings.

To fulfill this demand, quite a few data compression methods for highly repetitive strings have been developed. Examples are LZ77 [34], grammar compression [19, 12, 31, 14], block trees [2], and many others [25, 22, 10]. Among them, an efficient compression format gathering increased attention is the *run-length Burrows–Wheeler transform* (RLBWT), which is a run-length encoded BWT [6] as a reversible permutation of an input string on the lexicographical order of suffixes. Recently, researchers have focused on developing string

Table 1 Summary of state-of-the-art RLBWT construction algorithms. The update time in the rightmost column is the time needed to construct a new RLBWT from the current RLBWT for a character newly added to the string. The update time of r-comp is amortized. T is an input string of alphabet size σ and length n; $|\mathsf{PFP}|$ is the size of the dictionary and factorization created by the prefix-free parsing of T [5].

Method	Type	Running time	Working space (bits)	Update time
D. Belazzougui+ [4]	indirect	$O(n)$	$O(n \log \sigma)$	Unsupported
J. Munro+ [20]	indirect	$O(n)$	$O(n \log \sigma)$	Unsupported
D.Kempa [15]	indirect	$O(n/ \log_\sigma n + r \log^7 n)$	$O(n \log \sigma + r \log^6 n)$	Unsupported
D.Kempa+[16]	indirect	$O(n \log \sigma / \sqrt{\log n})$	$O(n \log \sigma)$	Unsupported
Big-BWT [5]	indirect	$O(n)$	$O(\lvert\mathsf{PFP}\rvert \log n)$	Unsupported
KK method [17, 23]	direct	$O(n(\log \log n)^2 + r \log^8 n)$	$O(r \operatorname{polylog} n)$	Unsupported
PP method [30]	direct	$O(n \log r)$	$O(r \log n)$	$O(\log r)$
Faster-PP method [28]	direct	$O(n \log r)$	$O(r \log n)$	$O(\log r)$
r-comp (this study)	direct	$O(n + r \log r)$	$O(r \log n)$	$O(1 + (r \log r)/n)$

processing methods such as locate query [13, 1, 3, 26], document listing [7], and substring enumeration [27] on RLBWT. Although several algorithms for constructing the RLBWT from an input string have been proposed thus far, there is no prior work that achieves the computational complexity of optimal time (i.e., time linearly proportional to the length of the input string) and BWT-runs bounded space (i.e., a working space linearly proportional to the number of equal-letter runs in the BWT and logarithmically proportional to the length of the input string).

Contribution. We present *r-comp*, the first construction algorithm of RLBWT that achieves optimal time and BWT-runs bounded space. R-comp directly constructs the RLBWT of an input string. It reads one character of an input string at a time from the reversed string and gradually builds the RLBWT corresponding to the suffixes read so far. The state-of-the-art online construction methods [30, 28] use inefficient data structures such as dynamic wavelet trees and B-trees for inserting each character into the current RLBWT at an insertion position, which is the most time-consuming part in an RLBWT construction. We present a new *divided BWT (DBWT)* representation of BWT and a new bipartite graph representation on DBWT called *LF-interval graph* to speed up the construction of RLBWT. The DBWT and LF-interval graph are efficiently built while reading each character one by one, and they enable us to quickly compute an appropriate position for inserting each character into the current RLBWT of the string. Another remarkable property of r-comp is the ability to extend the RLBWT for a newly added character without rebuilding the data structures used in r-comp from the beginning.

As a result, the computational complexity of r-comp is $O(n + r \log r)$ time and $O(r \log n)$ bits of working space for the length n of an input string and the number r of equal-letter runs in BWT. In particular, the computational complexity is optimal (i.e., $O(n)$) for strings with the property $r = O(n/ \log n)$, which holds for most highly repetitive strings. We experimentally tested the ability of r-comp to compress various highly repetitive strings, and we show that r-comp performs better than other methods with respect to computation time and space.

2 Related work

There are two types of methods for indirectly or directly constructing the RLBWT of a string (see Table 1 for a summary of state-of-the-art construction algorithms of RLBWT). In the indirect constructions of RLBWT, the BWT of an input string is first built and then the

BWT is encoded into the RLBWT by run-length encoding. Several efficient algorithms for constructing the BWT of a given string have been proposed [4, 16, 21, 8, 16, 20]. Let T be a string of length n with an alphabet of size σ, and let r be the number of equal-letter runs in its BWT. Kempa [15] proposed a RAM-optimal time construction of the BWT of string T with compression ratio $n/r = \Omega(\mathrm{polylog}\, n)$. The algorithm runs in $O(n/\log_\sigma n)$ time with $O(n \log \sigma)$ bits of working space. Kempa and Kociumaka also proposed a BWT construction in $O(n \log \sigma)$ bits of working space [16]. This algorithm runs in $O(n \log \sigma/\sqrt{\log n})$ time, which is bounded by $o(n)$ time for a string with $\log \sigma = o(\sqrt{\log n})$. These algorithms are not space efficient for highly repetitive strings in that their working space is linearly proportional to the length of the input string.

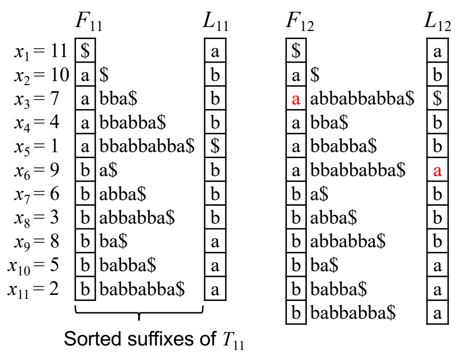
Big-BWT [5] is a practical algorithm for constructing the BWT of a huge string using *prefix-free parsing*, which constructs a dictionary of strings and a factorization from string T. Although Big-BWT runs in optimal time (i.e., $O(n)$) with $O(|\mathsf{PFP}| \log n)$ bits of working space for the sum $|\mathsf{PFP}|$ of (i) the lengths of all the strings in the dictionary and (ii) the number of strings in the factorization, Big-BWT is not space efficient for highly repetitive strings in the worst case, because $|\mathsf{PFP}|$ can be $\sqrt{n}$ times larger than r, resulting in $\Omega(r\sqrt{n} \log n)$ bits of working space (see the full version of the paper [24] for the proof). Even worse, several data structures used in these indirect constructions cannot be updated. Thus, one needs to rebuild the data structures from scratch for a newly added character, which reduces the usability of indirect constructions of RLBWT.

In the direct constructions of RLBWT, Policriti and Prezza [30] proposed an algorithm for the construction of RLBWT, which we call *PP method*. The PP method reads an input string in reverse by one character, and it gradually builds the RLBWT corresponding to the suffix that was just read, where an inefficient dynamic wavelet tree is used for inserting a character into the RLBWT at an appropriate position, limiting the scalability of the PP method in practice. Ohno et al. [28] proposed a faster method, which we call *Faster-PP method*, by replacing the dynamic wavelet tree used in the PP method by a B-tree. Whereas both the PP method and Faster-PP method run with the same time and space complexities – $O(n \log r)$ time and $O(r \log n)$ bits of working space – the time complexity is not the optimal time for most highly repetitive strings.

Kempa and Kociumaka [17] proposed a conversion algorithm, which is referred to as *KK method*, from the LZ77 parsing [34] of T to the RLBWT in $O(z \log^7 n)$ time with $O(z\, \mathrm{polylog}\, n)$ bits of space, where z is the number of phrases in the parsing. Theoretically, we can compute the RLBWT of an input string by combining the KK method with an algorithm for computing the LZ77 parsing (e.g., [23]), and the working space of their conversion is bounded by $O(r\, \mathrm{polylog}\, n)$ bits because $z = O(r \log n)$ [22]. Kempa and Langmead [18] proposed a practical algorithm for constructing a compressed grammar from an input string in $\Omega(n)$ time using an approximate LZ77 parsing. Because these methods use several static data structures that cannot be updated, the data structures must be rebuilt from scratch when a new character is added.

Although there are several algorithms for indirectly or directly constructing the RLBWT, no previous work has been able to achieve optimal time (i.e., $O(n)$ time) with BWT-runs bounded space (i.e., $O(r \log n)$ bits). We present *r-comp*, the first direct construction of RLBWT that achieves optimal time with BWT-runs bounded space for most highly repetitive strings. Details of r-comp are presented in the following sections.

This paper is organized as follows. Section 3 introduces basic notions used in this paper, and a DBWT representation of BWT is presented in Section 4. Section 5 presents an LF-interval graph representation of DBWT and a fast update operation on LF-interval



■ **Figure 1** (Left) Sorted suffixes of $T_{11} = abbabbabba\$$, F_{11}, and L_{11}. (Right) Sorted suffixes of $T_{12} = aabbabbabba\$$, F_{12}, and L_{12}.

graphs. The r-comp algorithm is presented in Section 6. Section 7 presents the experimental results using the r-comp algorithm on benchmark and real-world datasets of highly repetitive strings.

3 Preliminaries

Basic notation. An *interval* $[b, e]$ for two integers b and e ($b \leq e$) represents the set $\{b, b+1, \ldots, e\}$. Let T be a string of length n over an alphabet $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$ of size σ, and $|T|$ be the length of T (i.e., $|T| = n$). 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. Let T_δ be the suffix of T of length δ ($1 \leq \delta \leq n$), i.e., $T_\delta = T[(n - \delta + 1)..n]$. A *rank query* $\mathsf{rank}(T, c, i)$ on a string T returns the number of occurrences of character c in $T[1..i]$, i.e., $\mathsf{rank}(T, c, i) = |\{j \mid T[j] = c, 1 \leq j \leq i\}|$.

For a string P, $P[i] < P[j]$ means that the i-th character of P is smaller than the j-th character of P. Moreover, $T \prec P$ means that T is lexicographically smaller than P. Formally, $T \prec P$ if and only if either of the following two conditions holds: (i) there exists an integer i such that $T[1..i-1] = P[1..i-1]$ and $T[i] < P[i]$; (ii) T is a prefix of P (i.e., $T = P[1..|T|]$) and $|T| < |P|$. Here, $\mathsf{occ}_<(T, c)$ denotes the number of characters smaller than character c in string T (i.e., $\mathsf{occ}_<(T, c) = |\{j \mid j \in \{1, 2, \ldots, n\} \text{ s.t. } T[j] < c\}|$). Special character \$ is the smallest character in Σ. Throughout this paper, we assume that special character \$ only appears at the end of T (i.e., $T[n] = \$$ and $T[i] \neq \$$ for all $\{1, 2, \ldots, n-1\}$).

A *run* is defined as the maximal repetition of the same character. Formally, a substring $T[i..j]$ of T is a *run* of the same character c if it satisfies the following three conditions: (i) $T[i..j]$ is a repetition of the same character c (i.e., $T[i] = T[i+1] = \cdots = T[j] = c$); (ii) $i = 1$ or $T[i-1] \neq c$; (iii) $j = n$ or $T[j+1] \neq c$.

We use base-2 logarithm throughout this paper. Our computation model is a unit-cost word RAM with a machine word size of $\Theta(\log n)$ 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.

BWT, LF function, and RLBWT. The BWT [6] of a suffix T_δ is a permuted string L_δ of T_δ, and it is constructed as follows: all the suffixes of T_δ are sorted in the lexicographical order and the character preceding each suffix is taken. Formally, let $x_1, x_2, \ldots, x_\delta$ be the starting positions of the sorted suffixes of T_δ (i.e., $x_1, x_2, \ldots, x_\delta$ are a permutation

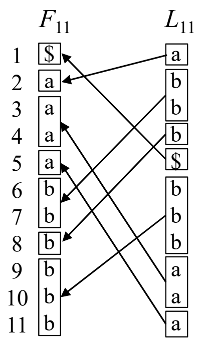


Figure 2 DBWT-repetitions and their corresponding F-intervals on F_{11} for a DBWT $D_{11} = a, bb, b, \$, bbb, aa, a$ of BWT L_{11} in Figure 1. Each rectangle on L_{11} represents a DBWT-repetition, and each rectangle on F_{11} represents an F-interval on F_{11}. Each directed arrow indicates the F-interval corresponding to the DBWT-repetition on L_{11}.

of sequence $1, 2, \ldots, \delta$ such that $T_\delta[x_1..\delta] \prec T_\delta[x_2..\delta] \prec \cdots \prec T_\delta[x_n..\delta]$). Then, $L_\delta = T_\delta[x_1 - 1], T_\delta[x_2 - 1], \ldots, T_\delta[x_\delta - 1]$), where $T_\delta[0]$ is defined as the last character of T_δ (i.e., $T_\delta[0] = T_\delta[\delta] = \$$). Similarly, the permuted string F_δ of suffix T_δ consists of the first characters of the sorted suffixes of T_δ, i.e., $F_\delta = T_\delta[x_1], T_\delta[x_2], \ldots, T_\delta[x_\delta]$.

Figure 1 illustrates the sorted suffixes of T_{11} and T_{12} for $T = aabbabbabba\$$. Here, $x_1, x_2, \ldots, x_{11}$ are the starting positions of the sorted suffixes of T_{11}. Moreover, $L_{11} = abbb\$bbbaaa$ and $F_{11} = \$aaaabbbbbb$. The BWT of T is $L_{12} = ab\$bbabbbaaa$.

There is a one-to-one correspondence between L_δ and F_δ because the two strings are permutations of T_δ. Formally, for two integers $i, j \in \{1, 2, \ldots, \delta\}$, $L_\delta[i]$ corresponds to $F_\delta[j]$ if and only if either of the following two conditions holds: (i) $x_i - 1 = x_j$ or (ii) $x_i = 1$ and $x_j = \delta$. *LF function* LF_δ is a bijective function from L_δ to F_δ [11]. Function $\mathsf{LF}_\delta(i) = j$ for two integers $i, j \in \{1, 2, \ldots, \delta\}$ if and only if $L_\delta[i]$ corresponds to $F_\delta[j]$. *LF formula* [11] is a well-known property of LF function, and it enables us to compute the corresponding position in F_δ from a position in L_δ. Namely, $\mathsf{LF}_\delta(i)$ is equal to the summation of (i) the number of characters in L_δ smaller than $L_\delta[i]$ and (ii) the number of $L_\delta[i]$ in the prefix $L_\delta[1..i]$, i.e., $\mathsf{LF}_\delta(i) = \mathsf{occ}_<(L_\delta, L_\delta[i]) + \mathsf{rank}(L_\delta, i, L_\delta[i])$.

BWT can be separated into all the runs of the same character. We call each run BWT-run. For BWT L_δ, r BWT-runs $P_1, P_2, \ldots, P_r$ satisfy (i) $L_\delta = P_1, P_2, \ldots, P_r$ and (ii) each P_i ($i = 1, 2, \ldots, r$) is a run of the same character in L_δ. The RLBWT of a suffix T_δ is defined as a sequence of r pairs $(P_1[1], |P_1|)$, $(P_2[1], |P_2|)$, $\ldots$, $(P_r[1], |P_r|)$. The RLBWT can be stored in $r(\log n + \log \sigma)$ bits, and we can recover T_δ from the RLBWT using LF function (e.g., [26]). Throughout this paper, r denotes the number of BWT-runs in the BWT of T.

In Figure 1, the BWT-runs in the BWT L_{11} of T_{11} are $a, bbb, \$, bbb$, and aaa. The RLBWT of T_{11} is $(a, 1)$, $(b, 3)$, $(\$, 1)$, $(b, 3)$, and $(a, 3)$.

4 DBWT

The divided BWT (DBWT) is a general concept in the RLBWT and is the foundation of the LF-interval graph. Formally, the DBWT D_δ of BWT L_δ is defined as a sequence $L_\delta[p_1..(p_2 - 1)], L_\delta[p_2..(p_3 - 1)], \ldots, L_\delta[p_k..(p_{k+1} - 1)]$ for $p_1 = 1 < p_2 < \cdots < p_k < p_{k+1} = n + 1$, where $L_\delta[p_i..(p_{i+1} - 1)]$ for each $i = 1, 2, \ldots, k$ is a repetition of the same character. We call each repetition of the same character in the DBWT *DBWT-repetition*. A DBWT-repetition is not necessarily a run. DBWT D_δ is equal to the RLBWT of T_δ if and only if $L_\delta[p_i..(p_i - 1)]$ for each $i \in \{1, 2, \ldots, k\}$ is a run.

In Figure 2, sequence $D_{11} = a, bb, b, \$, bbb, aa, a$ of equal-letter repetitions is a DBWT for BWT L_{11} of string T_{11}. The DBWT-repetitions in D_{11} are the strings enclosed by the rectangles on L_{11}.

The DBWT of a BWT is not unique because a BWT can be divided by various criteria. We present a criterion for DBWT in the following in order to efficiently build RLBWT. The LF function maps each DBWT-repetition $L_\delta[p_i..(p_{i+1}-1)]$ into the consecutive characters on interval $[\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$, which is called an *F-interval* on F_δ. The LF formula enables us to compute $\mathsf{LF}_\delta(j)$ for each position $j \in [p_i, (p_{i+1}-1)]$ on the i-th DBWT-repetition in $O(1)$ time using the starting position p_i of the DBWT-repetition and the F-interval $[\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$ corresponding to the DBWT-repetition as follows: $\mathsf{LF}_\delta(j) = \mathsf{LF}_\delta(p_i) + j - p_i$.

In Figure 2, each F-interval on F_{11} corresponding to a DBWT-repetition on L_{11} is enclosed by a rectangle. The F-intervals on F_{11} are $[1, 1]$, $[2, 2]$, $[3, 4]$, $[5, 5]$, $[6, 7]$, $[8, 8]$, and $[9, 11]$. The F-interval corresponding to the second DBWT-repetition bb is $[6, 7]$.

Let α be a user-defined parameter no less than 2 (i.e., $\alpha \geq 2$). DBWT-repetition $L_\delta[p_i..(p_{i+1}-1)]$ is said to *cover* the starting position $\mathsf{LF}_\delta(p_j)$ of an F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1})]$ on F_δ if interval $[p_i, (p_{i+1}-1)]$ on F_δ contains the position $\mathsf{LF}_\delta(p_j)$ (i.e., $\mathsf{LF}_\delta(p_j) \in [p_i, (p_{i+1}-1)]$). The DBWT-repetition is said to be *α-heavy* if it covers at least α starting positions of the F-intervals on F_δ for parameter $\alpha \geq 2$. Similarly, F-interval $[\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$ on F_δ is said to cover the starting position p_j of a DBWT-repetition $L_\delta[p_j..(p_{j+1}-1)]$ if interval $[\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$ on L_δ contains the position p_j (i.e., $p_j \in [\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$). An F-interval is said to be α-heavy if it covers at least α starting positions of DBWT-repetitions for parameter $\alpha \geq 2$. A DBWT is said to be *α-balanced* if the DBWT includes neither α-heavy DBWT-repetitions nor F-intervals, and D^α_δ denotes an α-balanced DBWT of BWT L_δ.

In Figure 2 with $\alpha = 3$, the fifth DBWT-repetition bbb of D_{11} covers two starting positions of F-intervals $[6, 7]$ and $[8, 8]$ on F_{11}, and the DBWT-repetition is not 3-heavy. The F-interval $[9, 11]$ of the fifth DBWT-repetition covers the starting positions of two DBWT-repetitions aa and a. Moreover, the F-interval of the fifth DBWT-repetition is not 3-heavy. Thus, D_{11} is 3-balanced because D_{11} includes neither 3-heavy DBWT-repetitions nor F-intervals.

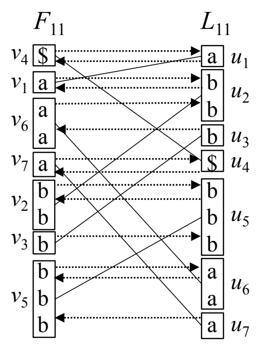
In the next section, the α-balanced DBWT is used to derive the time needed to update an LF-interval graph.

5 LF-interval graph

An LF-interval graph is a bipartite graph that represents both (i) the correspondence between each pair of elements in L_δ and F_δ according to the LF function and (ii) a covering relationship between DBWT-repetitions and F-intervals on a DBWT. The LF-interval graph $\mathsf{Grp}(D_\delta)$ for DBWT D_δ of k DBWT-repetitions $L_\delta[p_1..(p_2-1)], L_\delta[p_2..(p_3-1)], \ldots, L_\delta[p_k..(p_{k+1}-1)]$ is defined as 4-tuple $(U \cup V,\ E_{LF} \cup E_L \cup E_F,\ B_U \cup B_V,\ B_L \cup B_F)$, as detailed in the following.

Set $U = \{u_1, u_2, \ldots, u_k\}$ is a set of nodes, and u_i for each $i \in \{1, 2, \ldots, k\}$ represents the i-th DBWT-repetition $L_\delta[p_i..(p_{i+1}-1)]$ on DBWT D_δ. Moreover, set $V = \{v_1, v_2, \ldots, v_k\}$ is a set of nodes, and v_i for each $i \in \{1, 2, \ldots, k\}$ represents the F-interval $[\mathsf{LF}_\delta(p_i), \mathsf{LF}_\delta(p_{i+1}-1)]$ mapped from the i-th DBWT-repetition represented as u_i on DBWT D_δ by the LF function.

The set E_{LF} of undirected edges in LF-interval graph $\mathsf{Grp}(D_\delta)$ represents the correspondence between DBWT-repetitions on DBWT D_δ and F-intervals on F_δ according to the LF function. Formally, $E_{LF} \subseteq (U \times V)$ is a set of undirected edges between U and V, and $(u_i, v_j) \in E_{LF}$ holds if and only if the i-th DBWT-repetition $L_\delta[p_i..(p_{i+1}-1)]$ represented as u_i is mapped to the j-th F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1}-1)]$ represented as v_j. Namely, $E_{LF} = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$.



$B_U(u_1) = B_V(v_1) = (\text{a}, 1)$	$B_L(u_1, v_4) = 0$	$B_F(v_1, u_2) = 0$
$B_U(u_2) = B_V(v_2) = (\text{b}, 2)$	$B_L(u_2, v_1) = 0$	$B_F(v_2, u_5) = 0$
$B_U(u_3) = B_V(v_3) = (\text{b}, 1)$	$B_L(u_3, v_6) = 1$	$B_F(v_3, u_5) = 2$
$B_U(u_4) = B_V(v_4) = (\$, 1)$	$B_L(u_4, v_7) = 0$	$B_F(v_4, u_1) = 0$
$B_U(u_5) = B_V(v_5) = (\text{b}, 3)$	$B_L(u_5, v_2) = 0$	$B_F(v_5, u_6) = 0$
$B_U(u_6) = B_V(v_6) = (\text{a}, 2)$	$B_L(u_6, v_5) = 0$	$B_F(v_6, u_2) = 1$
$B_U(u_7) = B_V(v_7) = (\text{a}, 1)$	$B_L(u_7, v_5) = 2$	$B_F(v_7, u_4) = 0$

Figure 3 LF-interval graph $\mathsf{Grp}(D_{11})$ for DBWT D_{11} in Figure 2.

Two sets E_L and E_F of directed edges represent the covering relationship between DBWT-repetitions and F-intervals on DBWT D_δ. Set $E_L \subseteq (U \times V)$ is a set of directed edges from U to V, and $(u_i, v_j) \in E_L$ holds if and only if F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1} - 1)]$, represented as v_j, covers the starting position p_i of DBWT-repetition $L_\delta[p_i..(p_{i+1} - 1)]$, represented as u_i. Formally, $E_L = \{(u_i, v_j) \mid 1 \leq i, j \leq k \text{ s.t. } p_i \in [\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1} - 1)]\}$. Similarly, $E_F \subseteq (V \times U)$ is a set of directed edges from V to U, and $(v_j, u_i) \in E_F$ holds if and only if DBWT-repetition $L_\delta[p_i..(p_{i+1} - 1)]$, represented as u_i, covers the starting position $\mathsf{LF}_\delta(p_j)$ of F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1} - 1)]$, represented as v_j. Formally, $E_F = \{(v_j, u_i) \mid 1 \leq i, j \leq k \text{ s.t. } \mathsf{LF}_\delta(p_j) \in [p_i, (p_{i+1} - 1)]\}$.

Function $B_U : U \to (\Sigma, \mathbb{N})$ is a label function for the set U of nodes, and it maps each node $u_i \in U$ to a pair consisting of the character in Σ and the length in $\mathbb{N} = \{1, 2, \ldots\}$ for the i-th DBWT-repetition represented by u_i. Namely, $B_U(u_i) = (L_\delta[p_i], p_{i+1} - p_i)$. Similarly, $B_V : V \to (\Sigma, \mathbb{N})$ is a label function for the set V of nodes, and it maps each node $v_i \in V$ to a pair consisting of the character in Σ and the length in $\mathbb{N}$ for the repetition $F_\delta[\mathsf{LF}_\delta(p_i)..\mathsf{LF}_\delta(p_{i+1} - 1)]$ of the same character on the F-interval represented by v_i. Namely, $B_V(v_i) = (F_\delta[\mathsf{LF}_\delta(p_i)], \mathsf{LF}_\delta(p_{i+1} - 1) - \mathsf{LF}_\delta(p_i) + 1)$. For all $i \in \{1, 2, \ldots, k\}$, $B_U(u_i) = B_V(v_i)$ holds by the LF formula.

Function $B_L : E_L \to \mathbb{N}$ is a label function for the set E_L of directed edges, and it maps each edge $(u_i, v_j) \in E_L$ to an integer value representing the difference between the starting position p_i of DBWT-repetition $L_\delta[p_i..(p_{i+1} - 1)]$, represented as u_i, and the starting position $\mathsf{LF}_\delta(p_j)$ of F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1} - 1)]$, represented as v_j. Namely, $B_L(u_i, v_j) = p_i - \mathsf{LF}_\delta(p_j)$.

Similarly, $B_F : E_F \to \mathbb{N}$ is a label function for the set E_F of directed edges, and it maps each edge $(v_j, u_i) \in E_F$ to an integer value representing the difference between the starting position $\mathsf{LF}_\delta(p_j)$ of F-interval $[\mathsf{LF}_\delta(p_j), \mathsf{LF}_\delta(p_{j+1} - 1)]$, represented as v_j, and the starting position p_i of DBWT-repetition $L_\delta[p_i..(p_{i+1} - 1)]$, represented as u_i. Namely, $B_F(v_j, u_i) = \mathsf{LF}_\delta(p_j) - p_i$.

Figure 3 illustrates LF-interval graph $\mathsf{Grp}(D_{11})$ for DBWT D_{11} in Figure 2. For $U = \{u_1, u_2, \ldots, u_7\}$ and $V = \{v_1, v_2, \ldots, v_7\}$, each node $u_i \in U$ (respectively, $v_j \in V$) is enclosed by a rectangle on L_{11} (respectively, F_{11}). We have $E_{LF} = \{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), (u_6, v_6), (u_7, v_7)\}$. We depict each undirected edge in set E_{LF} by solid lines. Moreover, $E_L = \{(u_1, v_4), (u_2, v_1), (u_3, v_6), (u_4, v_7), (u_5, v_2), (u_6, v_5), (u_7, v_5)\}$, and $E_F = \{(v_1, u_2), (v_2, u_5), (v_3, u_5), (v_4, u_1), (v_5, u_6), (v_6, u_2), (v_7, u_4)\}$. Each directed edge in the two sets E_L and E_F is depicted by a dotted arrow. The four label functions B_U, B_V, B_L, and B_F are listed in Figure 3.

5.1 Dynamic data structures for the LF-interval graph

Several dynamic data structures are used for efficiently updating LF-interval graph $\mathsf{Grp}(D_\delta)$. Two doubly linked lists are used for supporting the insertions and deletions of nodes in U and V. The nodes in U should be totally ordered with respect to the starting position of the DBWT-repetition, which is represented as a node in U. Namely, $u_1 < u_2 < \cdots < u_k$. The nodes in set U are stored in a doubly linked list, where each node $u_i \in U$ has previous and next pointers connecting to the previous and next nodes, respectively, in the total order of nodes in U. Similarly, the nodes in V should be totally ordered with respect to the starting position of the F-interval, which is represented as a node in V. Namely, $v_{\pi_1} < v_{\pi_2} < \ldots < v_{\pi_k}$ holds for permutation $\pi_1, \pi_2, \ldots, \pi_k$ of sequence $1, 2, \ldots, k$ such that $\mathsf{LF}_\delta(p_{\pi_1}) < \mathsf{LF}_\delta(p_{\pi_2}) < \ldots < \mathsf{LF}_\delta(p_{\pi_k})$. The nodes in V are stored in another doubly linked list, where each node $v_i \in V$ has previous and next pointers connecting to the previous and next nodes in the increasing order of nodes in V, respectively. The space of two doubly linked lists storing nodes in U and V is $O(k \log n)$ bits.

All the nodes corresponding to α-heavy DBWT-repetitions in U are stored in an array data structure in any order. Similarly, all the nodes corresponding to α-heavy F-intervals in V are stored in another array data structure in any order. The two arrays take $O(k \log n)$ bits of space. Each array stores nothing if D_δ is α-balanced.

An *order maintenance data structure* [9] is used for comparing two nodes in U with the total order of U, and the data structure supports the following three operations: (i) the order operation determines whether or not node $u_i \in U$ precedes node $u_j \in U$ in the total order of U; (ii) the insertion operation inserts node $u_i \in U$ right after node $u_j \in U$ in the total order of U; (iii) the deletion operation deletes node $u_i \in U$ from U. The data structure supports these three operations in $O(1)$ time with $O(k \log n)$ bits of space, and it is used with a B-tree that stores the nodes in V, as explained below.

A B-tree (a type of self-balancing search tree) is built on the set V of nodes using the combination of the order maintenance data structure, where each node v_i in V is totally ordered with respect to (i) the total order of the node u_i in U that is connected to v_i by an edge in E_{LF} (i.e., $(u_i, v_i) \in E_{LF}$) and (ii) the first character $L_\delta[p_i]$ of the DBWT-repetition is represented as u_i. For a node $v_i \in V$, the B-tree stores a pair $(u_i, L_\delta[p_i])$ as the key of the node v_i. Nodes in V are totally ordered using the key, and $v_i \in V$ precedes $v_j \in V$ if and only if either of the following conditions holds: (i) $L_\delta[p_i] < L_\delta[p_j]$ or (ii) $L_\delta[p_i] = L_\delta[p_j]$ and u_i precedes u_j in the total order of U (i.e., $i < j$). Condition (ii) is efficiently computed in $O(1)$ time by the order maintenance data structure of U. According to the following lemma, the order of keys in the B-tree is the same as that of the nodes stored in the doubly linked list of V (i.e., the order of the nodes in the B-tree is $v_{\pi_1} < v_{\pi_2} < \ldots < v_{\pi_k}$).

▶ **Lemma 1.** *For two distinct nodes $v_i, v_j \in V$, the key of v_i precedes that of v_j in the B-tree of V if and only if v_i precedes v_j in the doubly linked list of V (i.e., $\mathsf{LF}_\delta(p_i) < \mathsf{LF}_\delta(p_j)$).*

Proof. See the full version of the paper [24]. ◀

The B-tree with the order maintenance data structure supports the three operations of search, insertion, and deletion for any node in V in $O(\log k)$ time with $O(k \log n)$ bits of space.

5.2 Extension of BWT ([30, 28])

The BWT of a suffix can be extended from the BWT of a shorter suffix [30, 28]. In this section, we review the extension of BWT, which is used for updating the LF-interval graph. The BWT $L_{\delta+1}$ of a suffix $T_{\delta+1}$ of length $\delta + 1$ can be computed from the BWT L_δ of the

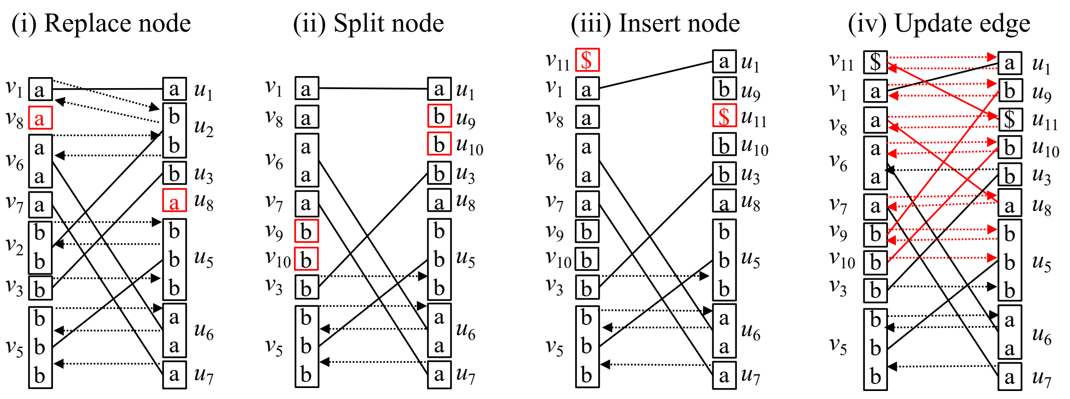

(i) Replace node
(ii) Split node
(iii) Insert node
(iv) Update edge
v_1 v_8 v_6 v_7 v_2 v_3 v_5 v_9 v_{10} v_{11}
u_1 u_2 u_3 u_8 u_5 u_6 u_7 u_9 u_{10} u_{11}
a b $

Figure 4 Each step in the update operation for the LF-interval graph in Figure 3. New nodes and edges created in each step are colored in red.

suffix T_δ of length δ using the following two steps: (i) special character \$ in L_δ is replaced with the first character c of $T_{\delta+1}$ (i.e., $c = T[n-\delta]$); (ii) special character \$ is inserted into L_δ at a position ins. Here, ins is computed by the LF formula as follows: for the position rep of special character \$ in L_δ (i.e., $L_\delta[\mathsf{rep}] = \$$), $\mathsf{ins} = \mathsf{occ}_<(L_\delta, c) + \mathsf{rank}(L_\delta, \mathsf{rep}, c) + 1$.

In Figure 1, BWT L_{12} of suffix T_{12} can be extended from L_{11} of suffix T_{11}. The first character c of T_{12} is a, and special character \$ is replaced with a at $\mathsf{rep} = 5$ on L_{11}. The insertion position ins for L_{11} is 3 because $\mathsf{occ}_<(L_{11}, a) + \mathsf{rank}(L_{11}, \mathsf{rep}, a) + 1 = 3$.

5.3 Foundation of updates of the LF-interval graph

Given the first character c in suffix $T_{\delta+1}$ of length $\delta + 1$, an update operation of LF-interval graph $\mathsf{Grp}(D^\alpha_\delta)$ for an α-balanced DBWT $D^\alpha_\delta = L_\delta[p_1..(p_2-1)], L_\delta[p_2..(p_3-1)], \ldots, L_\delta[p_k..(p_{k+1}-1)]$ of T_δ updates $\mathsf{Grp}(D^\alpha_\delta)$ to $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1})$ for a $(2\alpha+1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of $T_{\delta+1}$. The update operation updates the given LF-interval graph according to the extension of BWT. This operation consists of four main steps: (I) *replace node*, (II) *split node*, (III) *insert node*, and (IV) *update edge*. Note that the update operation presented in this section is a foundation for the ones presented in the following two subsections, where several modifications are made to the foundation for faster operation.

(I) Replace node. This step replaces the node $u_i \in U$ labeled $(\$, 1)$ with a new one $u_{i'}$ labeled $(c, 1)$, and it updates V according to the replacement of the node in U. Node u_i can be found in $O(1)$ time by keeping track of it on U. The doubly linked list of U is updated according to the replacement. The node $v_i \in V$ connected to u_i by edge $(u_i, v_i) \in E_{LF}$ is removed from V, and a new node $v_{i'}$ labeled $(c, 1)$ is inserted into V at the position next to the most backward node v_g of the nodes whose keys are smaller than key (u_i, c). Node v_g can be found in $O(\log k)$ time using the B-tree of V. This step takes $O(\log k)$ time in total.

Figure 4-(I) shows an example of the replace-node step for the LF-interval graph $\mathsf{Grp}(D_{11})$ in Figure 3. Node $u_4 \in U$ labeled $(\$, 1)$ on $\mathsf{Grp}(D_{11})$ is replaced with node u_8 labeled $(a, 1)$. Node $v_4 \in V$, which is connected to u_4 by edge $(v_4, u_4) \in E_{LF}$, is removed from V, and edge (v_4, u_4) is removed from E_{LF}. A new node v_8 with label $(a, 1)$ is inserted into V. This node is inserted into the doubly linked list of V at the position next to v_1 (i.e., $v_g = v_1$).

(II) Split node. The insert-node step (as the next step) inserts a new node representing special character \$ into U. However, before the insert-node step, the split-node step splits a node $u_j \in U$ into two new nodes at an appropriate position on the doubly linked list of

U. This step is executed for inserting the new node representing special character \$ into U at a appropriate position in the insert-node step. Following the extension of BWT in Section 5.2, node $u_j \in U$ has label $(L_\delta[p_j], p_{j+1} - p_j)$ for the two starting positions p_j and p_{j+1} satisfying $p_j < \mathsf{ins} < p_{j+1}$ for the insertion position ins of special character \$. Such a node u_j exists if and only if (i) the BWT $L_{\delta+1}$ of $T_{\delta+1}$ does not have special character \$ as the last character (i.e., $\mathsf{ins} \neq \delta + 1$) and (ii) $p_i \neq \mathsf{ins}$ for all $i \in \{1, 2, \ldots, k\}$. This is because $p_1 < p_2 < \ldots < p_{k+1}$ and $p_{k+1} = \delta + 1$ hold.

If node u_j does not exist in U, this step does not split nodes. Otherwise, u_j is replaced with two new nodes $u_{j'}$ and $u_{j'+1}$ in the doubly linked list of U, where $u_{j'}$ is previous to $u_{j'+1}$. The new nodes $u_{j'}$ and $u_{j'+1}$ are labeled as $(L_\delta[p_j], \mathsf{ins} - p_j)$ and $(L_\delta[p_j], p_{j+1} - \mathsf{ins})$ using insertion position ins, respectively.

Although we do not know position ins in the split-node step, we can find node u_j. This is because (i) set V contains node v_{gnext} representing the F-interval starting at position ins unless $\mathsf{ins} = \delta + 1$, and (ii) v_{gnext} is next to v_g in the doubly linked list of V for the node v_g searched for in the replace-node step. The following lemma ensures that we can find u_j and compute the labels of the new nodes in $O(1)$ time.

▶ **Lemma 2.** *The following two statements hold after executing the replace-node step: (i) we can check whether u_j exists or not in $O(1)$ time; (ii) we can find u_j and compute the labels of two nodes $u_{j'}$ and $u_{j'+1}$ in $O(1)$ time.*

Proof. See the full version of the paper [24]. ◀

Next, set V is updated according to the replacement of nodes in U, i.e., for undirected edge $(u_j, v_j) \in E_{LF}$, node v_j is replaced with two new nodes $v_{j'}$ and $v_{j'+1}$ in the doubly linked list of V, where $v_{j'}$ is previous to $v_{j'+1}$. Nodes $v_{j'}$ and $v_{j'+1}$ have the same labels of $u_{j'}$ and $u_{j'+1}$, respectively. Therefore, this step takes $O(1)$ time.

Figure 4-(II) illustrates an example of the split-node step. In this example, $\mathsf{ins} = 3$, $v_{\mathsf{gnext}} = v_6$, $v_j = v_2$, $v_{j'} = v_9$, and $v_{j'+1} = v_{10}$ hold. In Figure 3, the directed edge starting at node v_6 is labeled as integer 1 by function $B_F(v_6, u_2)$, and the directed edge points to node u_2 with label $(b, 2)$. Hence, node u_2 is replaced with two nodes u_9 and u_{10}. Two nodes u_9 and u_{10} are labeled with pairs $(b, 1)$ and $(b, 1)$, respectively. Node v_2 is connected to u_2 by edge $(v_2, u_2) \in E_{LF}$, and v_2 is replaced with two nodes v_9 and v_{10}. Here, v_9 and v_{10} are labeled with pairs $(b, 1)$ and $(b, 1)$, respectively.

(III) Insert node. This step inserts a new node $u_{x'}$ labeled $(\$, 1)$ into U, and it updates V according to the insertion of U. Analogous to the extension of BWT described in Section 5.2, the position for inserting the new node in the doubly linked list of U is determined according to the following three cases: (i) Node $u_j \in U$ was found and it was split into two nodes $u_{j'}$ and $u_{j'+1}$ in the split-node step. In this case, node $u_{x'}$ is inserted at the position next to $u_{j'} \in U$ on the doubly linked list of U. (ii) Node u_j was not found, and new node $v_{i'}$ is inserted at the position next to the last element on the doubly linked list of V in the replace-node step. In this case, $u_{x'}$ is inserted at the position next to the last element on the doubly linked list of U. (iii) Node u_j was not found, and $v_{i'}$ is inserted at the position previous to a node $v_{\mathsf{gnext}} \in V$ on the doubly linked list of V. In this case, v_{gnext} is connected to a node $u_x \in U$ by a directed edge in E_F, and $u_{x'}$ is inserted into the doubly linked list of U at the position previous to u_x.

Next, this step creates a new node $v_{x'}$ labeled $(\$, 1)$, and it is inserted into the doubly linked list of V. The new node is inserted at the top of the list, because the new label includes special character \$. This step takes $O(1)$ time.

Figure 4-(III) illustrates an example of the insert-node step. Because the split-node step replaced node u_2 with two nodes u_9 and u_{10}, the insert-node step inserts node u_{11} labeled $(\$, 1)$ at the position next to u_9 on the doubly linked list of U. On the other hand, the step inserts node v_{11} labeled $(\$, 1)$ in the doubly linked list of V at the position previous to node v_1.

(IV) Update edge. This step updates the set E_{LF} of undirected edges and two sets E_L and E_F of directed edges according to the two sets U and V, which were updated in the previous steps. This step consists of two phases: (i) for the nodes of u_i, v_i, u_j, and v_j removed in the replace-node and split-node steps, the edges connected to these nodes are removed from E_{LF}, E_L, and E_F; (ii) new edges connecting new nodes (i.e., $u_{j'}$, $u_{j'+1}$, $v_{i'}$, and $v_{j'+1}$) are added to E_{LF}, E_L, and E_F. The labels of new directed edges are computed at the second phase. The number of removed edges and new edges can be bounded by $O(\alpha)$ because (i) every node in the LF-interval graph for an $O(\alpha)$-balanced DBWT is connected to $O(\alpha)$ edges, (ii) the DBWT represented by the given LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ is α-balanced, and (iii) the LF-interval graph $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1})$ outputted by this update operation represents a $(2\alpha + 1)$-balanced DBWT. Because the number of updated edges is small, this step can be performed in $O(\alpha)$ time. See the full version of the paper [24] for the details of the update-edge step. Formally, we obtain the following lemma.

▶ **Lemma 3.** *The update-edge step takes $O(\alpha)$ time.*

Proof. See the full version of the paper [24]. ◀

In Figure 4-(IV), four edges (u_8, v_8), (u_9, v_9), (u_{10}, v_{10}), and (u_{11}, v_{11}) connecting new nodes are added to E_{LF}. Six directed edges (u_1, v_{11}), (u_9, v_1), (u_{11}, v_8), (u_{10}, v_6), (u_8, v_7), and (u_5, v_9) are added to E_L. Similarly, seven directed edges (v_{11}, u_1), (v_1, u_9), (v_8, u_{11}), (v_6, u_{10}), (v_7, u_8), (v_9, u_5), and (v_{10}, u_5) are added to E_F.

Update of the data structures. Similar to the update-edge step, the four data structures in the LF-interval graph (i.e., the order maintenance data structure, the B-tree of set V, and two arrays that store nodes representing α-heavy DBWT-repetitions and F-intervals) are updated according to the removed nodes and new nodes.

See the full version of the paper [24] for the details of the algorithm updating the four data structures. The following lemma concerning the update time of the four data structures.

▶ **Lemma 4.** *Updating the four data structures takes $O(\alpha + \log k)$ time.*

Proof. See the full version of the paper [24]. ◀

The update operation takes $O(\alpha + \log k)$ time in total. The following lemma concerning the theoretical results on this update operation holds.

▶ **Theorem 5.** *The following two statements hold: (i) the update operation takes $O(\alpha+\log k)$ time; (ii) the update operation takes as input the LF-interval graph for an α-balanced DBWT D^{α}_{δ} of BWT L_{δ}, and it outputs the LF-interval graph for a $(2\alpha+1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$ with at most two α-heavy DBWT-repetitions and at most two α-heavy F-intervals.*

Proof. See the full version of the paper [24]. ◀

From Theorem 5, the update operation outputs the LF-interval graph for a $(2\alpha+1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$, which is not α-balanced. The output DBWT $D^{2\alpha+1}_{\delta+1}$ is balanced into an α-balanced DBWT $D^{\alpha}_{\delta+1}$ by the balancing operation presented in Section 5.6. The update of the LF-interval graph presented in this section takes $O(\alpha + \log k)$ time, which results in an $O(\alpha n + n \log k)$-time construction of the RLBWT from an input string of length n. The following two sections (Section 5.4 and Section 5.5) present two modified updates of the LF-interval graph in $O(\alpha + \log k)$-time and $O(\alpha)$-time, respectively, in order to achieve the $O(\alpha n + r \log r)$-time construction of an RLBWT with $O(r \log n)$ bits of working space.

5.4 $O(\alpha + \log k)$-time update of LF-interval graph

This section presents the update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ taking an LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ and the first character c of suffix $T_{\delta+1}$ as input and running in $O(\alpha + \log k)$ time by modifying the foundation of the update operation presented in Section 5.3. For the node u_{i-1} previous to the node u_i representing special character \$ in the doubly linked list of U and the node u_{i+1} next to u_i, update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ is applied if neither u_{i-1} nor u_{i+1} has labels including character c.

We first present a modified update of the B-tree of V in $O(\log k)$ time. This update replaces the original update of the B-tree. The following lemma holds with respect to nodes searched for using the B-tree of V in the replace-node step of this update operation.

▶ **Lemma 6.** *For the node $v_g \in V$ searched for using the B-tree of V in the replace-node step of the update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$, v_g satisfies any one of the following three properties: (i) for undirected edge $(u_g, v_g) \in E_{LF}$ and node $u_{g+1} \in U$ next to u_g in the doubly linked list of U, the two consecutive nodes u_g and u_{g+1} have labels including different characters, and the label of u_{g+1} does not include special character \$; (ii) for the node $u_{g+2} \in U$ next to u_{g+1} in the doubly linked list of U, the two nodes u_g and u_{g+2} have labels including different characters, and the label of u_{g+1} includes special character \$; (iii) the label of u_g includes special character \$.*

Proof. See the full version of the paper [24]. ◀

Thus, the B-tree of V stores only the nodes in V satisfying one of the three conditions of Lemma 6, because Lemma 6 ensures only such nodes are searched for in the replace-node step of this update operation.

The target nodes inserted into the B-tree of V (respectively, the target nodes deleted from the B-tree of V) are limited to four (respectively, three) according to the following lemma.

▶ **Lemma 7.** *Nodes $u_i \in U$ and $v_i \in V$ are the nodes removed from U and V by the replace-node steps, respectively. Node $u_{i-1} \in U$ is the node previous to node u_i in the doubly linked list of U, and $v_{i-1} \in V$ is the node connected to u_{i-1} by undirected edge $(u_{i-1}, v_{i-1}) \in E_{LF}$. Node $v_j \in V$ is the node removed from V by the split-node step, and $v_{j'}, v_{j'+1} \in V$ are the nodes newly created by the same step. Similarly, $v_{i'} \in V$ and $v_{x'} \in V$ are the nodes created by the replace-node and insert-node steps, respectively. Then, the following two statements hold for the update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$: (i) targets inserted into the B-tree of V can be limited to only four nodes $v_{i-1}, v_{i'}, v_{x'}$, and $v_{j'+1}$; (ii) the targets deleted from the B-tree of V can be limited to only three nodes, v_{i-1}, v_i, and v_j.*

Proof. See the full version of the paper [24]. ◀

Thus, all the nodes inserted into the B-tree of V can be found by searching for only four nodes $v_{i-1}, v_{i'}, v_{x'}$, and $v_{j'+1}$ and checking whether or not each one, $v_{i-1}, v_{i'}, v_{x'}$, and $v_{j'+1}$, satisfies one of the three conditions presented in Lemma 6. Similarly, all the nodes deleted from the B-tree of V can be found by searching for only three nodes v_{i-1}, v_i, and v_j and checking whether or not each of these nodes satisfies one of the three conditions. This modified update of the B-tree takes $O(\log k)$ time in total.

The algorithm of update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ is the same as that of the original update operation presented in Section 5.3 except for the algorithm updating the B-tree of V. Hence, update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ takes $O(\alpha + \log k)$ time in total. The following lemma concerning the theoretical results on this update operation holds.

▶ **Lemma 8.** *Assume that the B-tree of V in LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ contains only nodes satisfying one of the three conditions of Lemma 6: (i) update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ runs in $O(\alpha + \log k)$ time; (ii) the update operation outputs the LF-interval graph $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1})$ for a $(2\alpha + 1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$ with at most two α-heavy DBWT-repetitions and at most two α-heavy F-intervals; (iii) the B-tree of V in the outputted LF-interval graph contains only nodes satisfying one of the three conditions of Lemma 6.*

In the next subsection, the second modified update operation of LF-interval graph achieves $O(\alpha)$ time using the B-tree of V containing only nodes satisfying one of the three conditions of Lemma 6.

5.5 $O(\alpha)$-time update of LF-interval graph

This section presents fast update operation $\mathsf{fastUpdate}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$, which takes an LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ and the first character c of suffix $T_{\delta+1}$ as input and runs in $O(\alpha)$ time. This time is achieved by modifying the foundation of the update operation presented in Section 5.3, and the B-tree of V needs to contain only nodes of satisfying one of the three conditions of Lemma 6 in the input and output LF-interval graphs, similar to the update operation in Section 5.4 (Lemma 8).

For node $u_{i-1} \in U$ previous to node $u_i \in U$ that represents special character \$ in the doubly linked list of U and node $u_{i+1} \in U$ next to u_i, the fast update operation is applied if either or both u_{i-1} and u_{i+1} have a label including character c; the update operation in Section 5.4 is applied otherwise. The large computational demand of the update operation on LF-interval graphs presented in Section 5.3 derives from the access and update of the B-tree of V in $O(\log k)$ time, resulting in an $O(\alpha + \log k)$ time update of LF-interval graphs. We present two improvements to the foundation of the update operation: (i) deletion and insertion operations of the B-tree of V in $O(1)$ time and (ii) the replace-node step in $O(1)$ time without using the B-tree of V. The details of the fast operation are presented in the full version of the paper [24].

Deletion and insertion operations of B-tree in constant time. Generally, inserting/deleting a key into/from the B-tree of V takes $O(\log k)$ time. We present $O(1)$-time deletion and insertion operations of a specific node in the B-tree of V without the need for heavyweight operations to maintain the balance of the B-tree.

Recall that (i) $u_{i'} \in U$ and $v_{i'} \in V$ are the nodes created by the replace-node step, (ii) $v_j \in V$ is the node removed from V by the split-node step, and (iii) $u_{x'} \in U$ and $v_{x'} \in V$ are the nodes created by the insert-node step. Let $u_{i'-1} \in U$ (respectively, $u_{i'+1} \in U$) be the node previous to node $u_{i'}$ (respectively, the node next to node $u_{i'}$) in the doubly linked list of U after the insert-node step has been executed. Then, there exist two nodes $v_{i'-1}$ and $v_{i'+1} \in V$ such that $(u_{i'-1}, v_{i'-1}), (u_{i'+1}, v_{i'+1}) \in E_{LF}$.

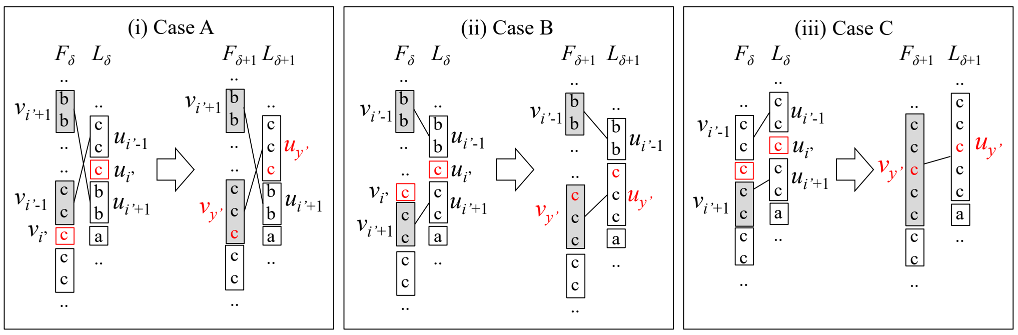


Figure 5 Three cases A, B, and C for preprocessing deletions/insertions in the B-tree. Gray nodes are stored in the B-tree.

In the fast update operation, the target nodes deleted from the B-tree are limited to at most four nodes $v_i, v_j, v_{i'-1}$, and $v_{i'+1}$, which is similar to Lemma 7-(ii). In the doubly linked list of V, node v_i is replaced with the new node $v_{x'}$ created in the insert-node step of the update operation. Because the two nodes v_i and $v_{x'}$ represent special character \$, both satisfy the third condition of Lemma 6. Thus, both nodes v_i and $v_{x'}$ are placed on the root of the B-tree. Hence, v_i can be deleted and $v_{x'}$ can be inserted into the B-tree in $O(1)$ time without balancing the B-tree.

Next, in the doubly linked list of V, node v_j is replaced with the two nodes $v_{j'}$ and $v_{j'+1} \in V$ that were created in the split-node step of the update operation. Node $v_{j'}$ satisfies none of the three conditions of Lemma 6. Thus, v_j is deleted from the B-tree of V, and $v_{j'+1}$ (that is, not $v_{j'}$) is only inserted into the B-tree of V if v_j is contained in the B-tree of V because of the next lemma.

▶ **Lemma 9.** *The following two statements hold: (i) $v_{j'}$ satisfies none of the three conditions of Lemma 6; (ii) for the node $u_j (\neq u_{i-1})$ connected to v_j by undirected edge $(u_j, v_j) \in E_{LF}$, $v_{j'+1}$ satisfies any one of the three conditions of Lemma 6 if and only if v_j satisfies any one of the three conditions.*

Proof. See the full version of the paper [24]. ◀

Because v_j is replaced with $v_{j'+1}$ in the doubly linked list of V, the element representing v_j can be replaced with the element representing $v_{j'+1}$ in the B-tree from Lemma 1. Hence, v_j is deleted from the B-tree of V and $v_{j'+1}$ is inserted into the B-tree of V in $O(1)$ time without balancing the B-tree. Illustrations of the replacement of two nodes v_i and v_j in the doubly linked list of V can be found in the full version of the paper [24].

The above procedure inserts $v_{j'+1}$ into the B-tree of V even if the node satisfies none of the three conditions of Lemma 6. This is because Lemma 9-(ii) assumes that $u_j \neq u_{i-1}$ holds, but $u_j \neq u_{i-1}$ is not always true. If the assumption holds, $v_{j'+1}$ is appropriately inserted into the B-tree of V. Otherwise (i.e., $v_{j'+1} = v_{i'-1}$ holds), node $v_{j'+1}$ is appropriately deleted from the B-tree of V in $O(1)$ time, which is explained next.

Two nodes $v_{i'-1}$ and $v_{i'+1}$ are appropriately deleted from the B-tree of V, and new nodes are inserted into the B-tree of V in the update operation of the LF-interval graph. For updating the B-tree of V in $O(1)$ time, we merge at most three nodes $v_{i'}$, $v_{i'-1}$, and $v_{i'+1}$ into a new node in a prepossessing step. The merging of nodes and update of the B-tree are performed according to the following three cases.

Case A: $v_{i'-1}$ has a label including character c and $v_{i'+1}$ does not have a label including character c (Figure 5-(i)). First, the two consecutive nodes $u_{i'-1}$ with label (c, ℓ) and $u_{i'}$ with label $(c, 1)$ are merged into a new one $u_{y'}$ with label $(c, \ell + 1)$ in the doubly linked list of U. Then, set V is updated according to the merge of the two nodes in U, i.e., the two consecutive nodes $v_{i'-1}$ and $v_{i'}$ are merged into a new one $v_{y'}$ with label $(c, \ell + 1)$ in the doubly linked list of V.

Node $v_{i'+1}$ is kept in the B-tree of V (if the node is contained in the B-tree). Node $v_{i'-1}$ is deleted from the B-tree of V and new node $v_{y'}$ is inserted only if $v_{i'-1}$ is contained in the B-tree of V. Because $v_{i'-1}$ is replaced by $v_{y'}$ in the doubly linked list of V, $v_{i'-1}$ can be deleted from the B-tree of V and $v_{y'}$ can be inserted in $O(1)$ time from Lemma 1.

Case B: $v_{i'-1}$ does not have a label including character c, and $v_{i'+1}$ has a label including character c (Figure 5-(ii)). First, the two consecutive nodes $u_{i'}$ and $u_{i'+1}$ with label (c, ℓ') are merged into a new node $u_{y'}$ with label $(c, \ell' + 1)$ in the doubly linked list of U. Then, set V is updated according to the merge of the two nodes in U.

In this case, $v_{i'-1}$ is contained in the B-tree of V, and the node is kept. Node $v_{i'+1}$ is deleted from the B-tree of V and new node $v_{y'}$ is inserted only if $v_{i'+1}$ is contained in the B-tree of V. Because $v_{i'+1}$ is replaced with $v_{y'}$ in the doubly linked list of V, $v_{i'+1}$ is deleted from the B-tree of V and $v_{y'}$ is inserted in $O(1)$ time from Lemma 1.

Case C: both $v_{i'-1}$ and $v_{i'+1}$ have labels including the same character c (Figure 5-(iii)). First, the three consecutive nodes $u_{i'-1}$, $u_{i'}$ and $u_{i'+1}$ are merged into a new node $u_{y'}$ with label $(c, \ell + \ell' + 1)$ including the same character c in the doubly linked list of U. Then, set V is updated according to the merge of the three nodes in U.

In this case, $v_{i'-1}$ is not contained in the B-tree of V. Node $v_{i'+1}$ is deleted from the B-tree and the new node $v_{y'}$ is inserted if $v_{i'+1}$ is contained in the tree; otherwise, $v_{y'}$ is not inserted into the tree. Because $v_{i'+1}$ is replaced by $v_{y'}$ in the doubly linked list of V, $v_{i'+1}$ is deleted and $v_{y'}$ is inserted into the B-tree of V in $O(1)$ time from Lemma 1.

One of the three cases always holds (see the full version of the paper [24]). The preprocessing of the B-tree of V (i.e., the merging of nodes) does not affect Lemma 9. Hence, updating the B-tree takes $O(1)$ time.

Replace-node step in constant time. The replace-node step is performed in $O(1)$ time by finding the position for inserting a new node $v_{i'}$ with label $(c, 1)$ into the doubly linked list of V without accessing the B-tree of set V. This is made possible if either or both u_{i-1} and u_{i+1} have labels including the same character c. The following lemma holds.

▶ **Lemma 10.** *If either or both u_{i-1} and u_{i+1} have labels including the same character c, the position for inserting the new node $v_{i'}$ into the doubly linked list of V can be found in $O(1)$ time.*

Proof. See the full version of the paper [24]. ◀

The following lemma concerning the conclusion of the fast update operation holds.

▶ **Lemma 11.** *Assume that the B-tree of V in LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ contains only the nodes that satisfy one of the three conditions of Lemma 6: (i) fast update operation $\mathsf{fastUpdate}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ runs in $O(\alpha)$ time; (ii) the fast update operation outputs the LF-interval graph for a $(2\alpha+1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$ with at most two α-heavy DBWT-repetitions and at most two α-heavy F-intervals; (iii) the B-tree of V in the outputted LF-interval graph contains only nodes satisfying one of the three conditions of Lemma 6.*

Proof. See the full version of the paper [24]. ◀

Both update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ and fast update operation $\mathsf{fastUpdate}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ output the LF-interval graph for a $(2\alpha + 1)$-balanced DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$ with at most two α-heavy DBWT-repetitions and at most two α-heavy F-intervals. The outputs are balanced by the balancing operation presented in the next subsection such that the LF-interval graph represents an α-balanced DBWT $D^{\alpha}_{\delta+1}$ of BWT $L_{\delta+1}$.

5.6 Balancing operation of the LF-interval graph

Balancing operation $\mathsf{balance}(\mathsf{Grp}(D_{\delta}))$ takes the LF-interval graph $\mathsf{Grp}(D_{\delta})$ for a DBWT D_{δ} of BWT L_{δ} as input, and it outputs the LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ for an α-balanced DBWT D^{α}_{δ} of the same BWT. The basic idea behind the balancing operation is to iteratively remove each of nodes representing α-heavy DBWT repetitions and α-heavy F-intervals from the given LF-interval graph by splitting the chosen node into two nodes. The balancing operation repeats this process until it obtains the LF-interval graph for an α-balanced DBWT.

We explain the algorithm of the balancing operation. At each iteration, the balancing operation processes the LF-interval graph for an $O(\alpha)$-balanced DBWT with $O(\alpha)$ α-heavy DBWT-repetitions and $O(\alpha)$ α-heavy F-intervals. Let $(u_i, v_i) \in E_{LF}$ be an undirected edge such that node $u_i \in U$ represents an α-heavy DBWT-repetition, or node $v_i \in V$ represents an α-heavy F-interval. At the iteration, node u_i is split into two nodes, and v_i is split into two nodes according to the splitting of u_i. The LF-interval graph $\mathsf{Grp}(D^{O(\alpha)}_{\delta})$ is updated according to the splitting of u_i and v_i. One iteration of the balancing operation takes $O(\alpha)$ time. See the full version of the paper [24] for the details of the balancing operation.

The following lemma concerning the theoretical results on the balancing operation holds.

▶ **Lemma 12.** *For the LF-interval graph $\mathsf{Grp}(D^{2\alpha+1}_{\delta})$ for a $(2\alpha+1)$-balanced DBWT including at most two α-heavy DBWT-repetitions and at most two α-heavy F-intervals, we assume that the B-tree of V in the LF-interval graph contains only nodes satisfying one of the three conditions of Lemma 6. Then, balancing operation $\mathsf{balance}(\mathsf{Grp}(D^{2\alpha+1}_{\delta}))$ takes $O(\alpha)$ time per iteration for all $\alpha \geq 4$, and the B-tree of V in the outputted LF-interval graph contains only nodes satisfying one of the three conditions of Lemma 6.*

Proof. See the full version of the paper [24]. ◀

6 R-comp algorithm

In this section, we present the r-comp algorithm, and we also present its space and time complexities. The r-comp algorithm takes input string T and parameter $\alpha \geq 2$, and it outputs the RLBWT of T. The pseudo-code of r-comp is given in Algorithm 1.

The algorithm reads T from its end to its beginning (i.e., $T[n], T[n-1], \ldots, T[1]$), and it gradually builds the LF-interval graph $\mathsf{Grp}(D^{\alpha}_{n})$ for an α-balanced DBWT D^{α}_{n} of BWT L_n of T. At each $\delta \in \{1, 2, \ldots, n-1\}$, LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$ for an α-balanced DBWT D^{α}_{δ} of BWT L_{δ} of $T[(n-\delta+1)..n]$ (i.e., suffix T_{δ}) is built. For the node $u_i \in U$ representing special character \$ in the doubly linked list of set U of nodes in LF-interval graph $\mathsf{Grp}(D^{\alpha}_{\delta})$, two nodes $u_{i-1} \in U$ and $u_{i+1} \in U$ are previous and next to node u_i, respectively, in the list. If neither the label of u_{i-1} nor the label of u_{i+1} includes the $(n-\delta)$-th character c of T (i.e., the first character c of $T_{\delta+1}$), the update operation $\mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ described in Section 5.4 is applied; otherwise the fast update operation $\mathsf{fastUpdate}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ described in Section 5.5

Algorithm 1 R-comp algorithm. The algorithm takes string T and parameter $\alpha \geq 2$ as input, and it outputs the RLBWT of T. n : length of T; D^{α}_{δ} : α-balanced DBWT of BWT L_{δ} of $T[(n-\delta+1)..n]$; $\mathsf{Grp}(D^{\alpha}_{\delta})$: the LF-interval graph of α-balanced DBWT D^{α}_{δ}.

```
1:  function R-COMP($T$, $\alpha$)
2:      $D^{\alpha}_{1} \leftarrow \$$                                  ▷ Initialize $D^{\alpha}_{1}$ as special character $\$$
3:      build $\mathsf{Grp}(D^{\alpha}_{1})$
4:      for $\delta = 1, 2, \ldots, n-1$ do
5:          $c \leftarrow T[n-\delta]$                             ▷ Read character $c$ from $T[n-\delta]$
6:          if neither the label of $u_{i-1}$ nor the label of $u_{i+1}$ includes character $c$ then
7:              $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1}) \leftarrow \mathsf{update}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$          ▷ The update operation in Sec. 5.4
8:          else
9:              $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1}) \leftarrow \mathsf{fastUpdate}(\mathsf{Grp}(D^{\alpha}_{\delta}), c)$ ▷ The fast update operation in Sec. 5.5
10:         end if
11:         $\mathsf{Grp}(D^{\alpha}_{\delta+1}) \leftarrow \mathsf{balance}(\mathsf{Grp}(D^{2\alpha+1}_{\delta+1}))$         ▷ The balancing operation in Sec. 5.6
12:     end for
13:     Recover $D^{\alpha}_{n}$ from $\mathsf{Grp}(D^{\alpha}_{n})$
14:     Convert $D^{\alpha}_{n}$ into the RLBWT of $T$
15:     Return the RLBWT
16: end function
```

is applied. Both the update operation and the fast update operation output an LF-interval graph $\mathsf{Grp}(D^{2\alpha+1}_{\delta+1})$ for DBWT $D^{2\alpha+1}_{\delta+1}$ of BWT $L_{\delta+1}$ that is not α-balanced. Thus, the r-comp algorithm balances the LF-interval graph such that it represents an α-balanced DBWT $D^{\alpha}_{\delta+1}$ using the balancing operation in Section 5.6 as $\mathsf{Grp}(D^{\alpha}_{\delta+1})$. After $(n-1)$ iterations of those steps, the LF-interval graph $\mathsf{Grp}(D^{\alpha}_{n})$ for α-balanced DBWT D^{α}_{n} of BWT L_n is obtained; it is then converted into D^{α}_{n} using the doubly linked list of U. Finally, D^{α}_{n} is converted into the RLBWT of T.

6.1 Space and time complexities

Space complexity. The r-comp algorithm requires $O(k \log n)$ bits of space for k DBWT-repetitions in the DBWT D^{α}_{n} because the LF-interval graph $\mathsf{Grp}(D^{\alpha}_{n})$ for the DBWT requires $O(k \log n)$ bits of space. The value of k depends on the number of executions of update, fast update, and balancing operations executed by the r-comp algorithm. The following lemma ensures that k can be bounded by $O(r)$.

▶ **Lemma 13.** *We modify the fast update operation. Then, the following three statements hold: (i) this modification does not affect Lemma 11; (ii) $k \leq r + k_{\mathsf{split}}$; (iii) $k_{\mathsf{split}} \leq \frac{2r}{\lceil \alpha/2 \rceil - 7}$ holds for any constant $\alpha \geq 16$.*

Proof. See the full version of the paper [24]. ◀

Because $k = O(r)$ by Lemma 13, the following theorem is obtained.

▶ **Theorem 14.** *The r-comp algorithm takes $O(r \log n)$ bits of working space for $\alpha \geq 16$.*

Time complexity. The bottleneck of r-comp is the update operation of LF-interval graph with $O(\alpha + \log k)$ time in Section 5.4. The number of executions of the update operation can be bounded by $O(r)$. This fact indicates that we can bound the running time of r-comp by $O(\alpha n + r \log k)$, i.e., $O(\alpha n + r \log r)$. Finally, we obtain the following theorem.

▶ **Theorem 15.** *R-comp runs in $O(\alpha n + r \log r)$ time for $\alpha \geq 16$.*

Proof. See the full version of the paper [24]. ◀

7 Experiments

Setup. We empirically tested the performance of the r-comp algorithm on strings from four datasets with different types of highly repetitive strings: (i) nine strings from the Pizza&Chili repetitive corpus [29]; (ii) three strings (boost, samtools, and sdsl) of the latest revisions of Git repositories, each of which is 1GB in size; (iii) a 37GB string (enwiki) of English Wikipedia articles with a complete edit history [33]; and (iv) a 59GB string (chr19.1000) obtained by concatenating chromosome 19 from 1,000 human genomes in the 1000 Genomes Project [32]. See the full version of the paper [24] for the relevant statistics in the datasets. The ratio n/r of string length n to the number r of BWT-runs in BWT is the compression ratio of each string. The strings with high compression ratios are versions of Wikipedia articles (einstein.de.txt and einstein.en.txt), revisions of Git repositories (boost, samtools, and sdsl), and 1000 human genomes (chr19.1000).

We implemented two versions of the r-comp algorithm (i.e., r-comp and r-comp$_{\text{saving}}$). Here, r-comp is the straightforward implementation of the r-comp algorithm presented in Section 6; r-comp$_{\text{saving}}$ is a space-saving implementation of the r-comp algorithm. This version is more space efficient than r-comp because it uses a grouping technique with parameter $g = 16$. We compared r-comp and r-comp$_{\text{saving}}$ with three state-of-the-art algorithms, one indirect construction algorithm of RLBWT (Big-BWT) and two direct construction algorithms of RLBWT (the PP and Faster-PP methods), which were reviewed in Section 2 and summarized in Table 1. The comparisons were performed using a single thread on one core of a CPU. The source code of the r-comp algorithm is available at `https://github.com/kampersanda/rcomp`. See the full version of the paper [24] for the details of the setup.

Results. A comparison of the r-comp variants (r-comp and r-comp$_{\text{saving}}$) shows that the working space of r-comp$_{\text{saving}}$ was 3.8–4.4 times smaller than that of r-comp, whereas the construction time of r-comp$_{\text{saving}}$ was at most only 2.0 times slower and at most 2.0 times faster on world_leaders and samtools. These results show r-comp$_{\text{saving}}$ has a high compression performance when compared with r-comp. Comparisons of the experimental results of r-comp$_{\text{saving}}$ and the other methods are presented below.

r-comp$_{\text{saving}}$ was the fastest in the comparison to the direct RLBWT constructions of the PP and Faster-PP methods. Especially for strings with a large ratio n/r, r-comp$_{\text{saving}}$ was 81–118 times faster than the PP method and 3.8–5.1 times faster than the Faster-PP method; the working space of r-comp$_{\text{saving}}$ was 2.4–6.1 times larger than that of the PP method and 2.8–2.9 times larger than that of the Faster-PP method, which shows that the working space of r-comp$_{\text{saving}}$ is reasonable considering its construction time. For the large dataset enwiki, r-comp$_{\text{saving}}$ finished the construction in 6.8 hours, whereas the Faster-PP method took 15.4 hours. In addition, the PP method did not finish within 24 hours. For the 1000 human genomes chr19.1000, r-comp$_{\text{saving}}$ finished the construction in 11 hours, whereas the PP and Faster-PP methods did not finish within 24 hours.

In the comparison between r-comp and Big-BWT, r-comp was more space efficient than Big-BWT on most strings. Because r was much smaller than $|\mathsf{PFP}|$, the results are consistent with the theoretical bound $O(r \log n)$ of the working space of the r-comp algorithm (Theorem 14). On strings with large values of $|\mathsf{PFP}|/r$, whereas r-comp$_{\text{saving}}$ was slower than Big-BWT, the difference in the construction times between r-comp$_{\text{saving}}$ and Big-BWT were reasonable if one considers the space efficiency of r-comp$_{\text{saving}}$. For example, for boost, r-comp was 26 times more space efficient and only 2.7 times slower; for world_leaders, r-comp was 3.9 times more space efficient and only 1.5 times slower.

On the large datasets (i.e., enwiki and chr19.1000), r-comp was more space-efficient but slower than Big-BWT; r-comp was 2.3 times space-efficient but 10 times slower than Big-BWT on enwiki; r-comp was 2.5 times space-efficient but 11 times slower than Big-BWT on chr19.1000.

Overall, r-comp$_{\text{saving}}$ was the fastest RLBWT construction in $O(r \log n)$ bits of space. Although Big-BWT was faster than r-comp$_{\text{saving}}$, it was not space efficient for several strings (e.g., boost). On most datasets, r-comp$_{\text{saving}}$ achieved a better tradeoff between construction time and working space. Furthermore, r-comp has a huge advantage in that it supports the insertion of a new character into RLBWT, while BigBWT does not support such an insertion operation.

8 Conclusion

We presented r-comp, the first optimal-time construction algorithm of RLBWT in $O(n)$ time with $O(r \log n)$ bits of working space for highly repetitive strings with $r = O(n/\log n)$. Experimental results using benchmark and real-world datasets of highly repetitive strings demonstrated the superior performance of the r-comp algorithm.

The idea behind the DBWT presented in this paper has a wide variety of applications, and it is applicable to the construction of various types of data structures. Therefore, a future task is to develop optimal-time constructions of various data structures for fast queries.

References

1 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theor. Comput. Sci.*, 812:96–108, 2020.

2 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021.

3 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proceedings of CPM*, pages 26–39, 2015.

4 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algor.*, 16:17:1–17:54, 2020.

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

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

7 Dustin Cobas, Veli Mäkinen, and Massimiliano Rossi. Tailoring r-index for document listing towards metagenomics applications. In *Proceedings of SPIRE*, pages 291–306, 2020.

8 Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the Burrows-Wheeler transform in place and in small space. *J. Disc. Algor.*, pages 44–52, 2015.

9 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of STOC*, pages 365–372, 1987.

10 Patrick Dinklage, Jonas Ellert, Johannes Fischer, Dominik Köppl, and Manuel Penschuck. Bidirectional text compression in external memory. In *Proceedings of ESA*, pages 41:1–41:16, 2019.

11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of FOCS*, pages 390–398, 2000.

12 Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. Practical grammar compression based on maximal repeats. *Algorithms*, 13:103, 2020.

13 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67, 2020.

14 Artur Jez. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016.

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

16 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of STOC*, pages 756–767, 2019.

17 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *Proceedings of FOCS*, pages 1002–1013, 2020.

18 Dominik Kempa and Ben Langmead. Fast and space-efficient construction of AVL grammars from the LZ77 parsing. In *Proceedings of ESA*, 2021.

19 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of DCC*, pages 296–305, 1999.

20 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of SODA*, pages 408–424, 2017.

21 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43:1781–1806, 2014.

22 Gonzalo Navarro, Carlos Ochoa, and Nicola Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inform. Theory*, 67:1008–1026, 2021.

23 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discret. Appl. Math.*, 274:116–129, 2020.

24 Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An optimal-time RLBWT construction in BWT-runs bounded space. *CoRR*, abs/2202.07885, 2022.

25 Takaaki Nishimoto and Yasuo Tabei. LZRR: LZ77 parsing with right reference. *Inf. Comput.*, page 104859, 2021.

26 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs bounded space. In *Proceedings of ICALP*, pages 101:1–101:15, 2021.

27 Takaaki Nishimoto and Yasuo Tabei. R-enum: Enumeration of characteristic substrings in BWT-runs bounded space. In *Proceedings of CPM*, pages 21:1–21:21, 2021.

28 Tatsuya Ohno, Kensuke Sakai, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online RLBWT and its application to LZ77 parsing. *J. Disc. Algor.*, 52-53:18–28, 2018.

29 Pizza&Chili repetitive corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

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

31 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302:211–222, 2003.

32 The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491:56–65, 2012.

33 Enwiki dump progress on 20210401: All pages with complete edit history (`xml-p1p873`). `https://dumps.wikimedia.org/enwiki/`.

34 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23:337–343, 1977.