Conversion from RLBWT to LZ77

Takaaki Nishimoto
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan
takaaki.nishimoto@riken.jp

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan
yasuo.tabei@riken.jp

Abstract

Converting a compressed format of a string into another compressed format without an explicit decompression is one of the central research topics in string processing. We discuss the problem of converting the run-length Burrows-Wheeler Transform (RLBWT) of a string into Lempel-Ziv 77 (LZ77) phrases of the reversed string. The first results with Policriti and Prezza's conversion algorithm [Algorithmica 2018] were $O(n \log r)$ time and $O(r)$ working space for length of the string n, number of runs r in the RLBWT, and number of LZ77 phrases z. Recent results with Kempa's conversion algorithm [SODA 2019] are $O(n/\log n + r \log^9 n + z \log^9 n)$ time and $O(n/\log_\sigma n + r \log^8 n)$ working space for the alphabet size σ of the RLBWT. In this paper, we present a new conversion algorithm by improving Policriti and Prezza's conversion algorithm where dynamic data structures for general purpose are used. We argue that these dynamic data structures can be replaced and present new data structures for faster conversion. The time and working space of our conversion algorithm with new data structures are $O(n \min\{\log \log n, \sqrt{\frac{\log r}{\log \log r}}\})$ and $O(r)$, respectively.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Burrows-Wheeler Transform, Lempel-Ziv Parsing, Lossless Data Compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.9

1 Introduction

Converting a compressed format of a string into another compressed format without an explicit decompression is one of the central research topics in string processing. Examples are conversions from the Lempel-Ziv 77 (LZ77) Phrases of a string into a grammar-based encoding [10, 16], from a grammar-based encoding of a string into LZ78 phrases [2, 1] and from a grammar-based encoding of a string into another grammar-based encoding [17]. Such conversion is beneficial when one intends to process a compressed string in a different compressed format without decompressing it.

LZ77 parsing, proposed in 1976 [13], is one of the most popular lossless data compression algorithms and is a greedy partition of a string such that each phrase is a previous occurrence of a substring or a character not occurring previously in the string. The *run-length Burrows-Wheeler transform (RLBWT)* [5] is a recent popular lossless data compression algorithm with a run-length encoded permutation of a string.

Policriti and Prezza [15] proposed the first conversion algorithm from the RLBWT of an input string into the LZ77 phrases of the reversed string. The basic idea with this algorithm is to carry out backward searches on the RLBWT and find a previous occurrence of each phrase using red-black trees storing a sampled suffix array and dynamic data structure for solving *the searchable partial sums with the indels problem* (e.g., [4, 9]). Since these data structures are updated frequently for scanning the string in the RLBWT format, the running time and working space are $O(n \log r)$ and $O(r)$ words, respectively, for string length n and number of runs r (i.e., the number of continuous occurrences of the same characters).

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis; Article No. 9; pp. 9:1–9:12
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Summary of conversion algorithms from the RLBWT to LZ77.

Algorithm	Conversion time	Working space (words)
Policriti and Prezza (Thm. 7) [15]	$O(n\sqrt{\frac{\log r}{\log\log r}})$ or expected $O(n \log\log n)$	$O(r)$
Kempa (Thm. 7.3) [12]	$O(n/\log n + r\log^9 n + z\log^9 n)$	$O(n/\log_\sigma n + r\log^8 n)$
This study	$O(n \min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$	$O(r)$

Their running time can be improved using a more faster dynamic predecessor instead of the red-black trees. As a result, we can achieve expected $O(n \log\log n)$ time and $O(n\sqrt{\frac{\log r}{\log\log r}})$ time using *y-fast trie* [18] and Beame and Fich's dynamic predecessor [3], respectively.

Kempa [12] recently presented a conversion algorithm from the RLBWT to LZ77, which runs in $O(n/\log n + r\log^9 n + z\log^9 n)$ time and $O(n/\log_\sigma n + r\log^8 n)$ working space for the number z of LZ77 phrases and the alphabet size σ of the string in the RLBWT format. While the algorithm runs in $o(n)$ time and working space, especially when r and z are small (e.g., $r, z = O(n/\log^9 n)$), the working space of the algorithm is larger than that of Policriti and Prezza's algorithm in many cases.

In this paper, we present a new conversion algorithm from the RLBWT to LZ77 by improving Policriti and Prezza's algorithm. Their algorithm adopts dynamic data structures for four queries comprising backward search, LF function, access queries on the RLBWT, and so-called *range more than query (RMTQ)*. We argue that these dynamic data structures can be replaced for answering those queries and present new data structures for faster conversion. Our algorithm runs in $O(n \min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$ deterministic time and $O(r)$ working space, which improves their algorithm (see Table 1 for a summary of conversion algorithms).

2 Preliminaries

Let Σ 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 and $T[i..j]$ be the substring of T that begins at position i and ends at position j. The $T[i..]$ denotes the suffix of T beginning at position i, i.e., $T[i..n]$. Let T^R be the reversed string of T, i.e., $T^R = T[n]T[n-1]\cdots T[1]$. For two integers i and j $(i \leq j)$, $[i, j]$ represents $\{i, i+1, \ldots, j\}$. For two strings T and P, $T \prec P$ is that T is lexicographically smaller than P. $Occ(T, P)$ denotes all the occurrence positions of string P in string T, i.e., $Occ(T, P) = \{i \mid P = T[i..(i + |P| - 1)], i \in [1, n - |P| + 1]\}$. *Right occurrence* p of substring $T[i..j]$ is a subsequent occurrence position of $T[i..j]$ in T, i.e., any $p \in Occ(T[i+1..], T[i..j])$.

For a string T, character c, and integer i, $\mathsf{rank}(T, c, i)$ returns the number of a character c in $T[..i]$, i.e., $\mathsf{rank}(T, c, i) = |Occ(T[..i], c)|$. $\mathsf{access}(T, i)$ returns $T[i]$. $\mathsf{select}(T, c, i)$ returns the position of the i-th occurrence of a character c in T. If the number of occurrences of c in T is smaller than i, it returns $n+1$, i.e., $\mathsf{select}(T, c, i) = \min(\{j \mid |Occ(T[..j], c)| \geq i, j \in [1, n]\} \cup \{n+1\})$ where $\min\{S\}$ returns the minimum value in a given set S.

For an integer x and set S of integers, a *predecessor* query $\mathsf{pred}(S, x)$ returns the number of elements that are no more than x in S, i.e., $\mathsf{pred}(S, x) = |\{y \mid y \leq x, y \in S\}|$. A *predecessor data structure* of S supports predecessor queries on S. For an integer array D and two positions i, j $(i \leq j)$ on D, a *range maximum query* (RMQ) $\mathsf{RMQ}(D, i, j, k)$ returns the maximum value in $D[i..j]$, i.e., $\mathsf{RMQ}(D, i, j) = \max D[i..j]$, where $\max\{S\}$ returns the maximum value in a given set S.

SA-interval of *si*

i	SA	Suffix
1	12	\$
2	11	i\$
3	8	ippi\$
4	5	issippi\$
5	2	ississippi\$
6	1	mississippi\$
7	10	pi\$
8	9	ppi\$
9	7	sippi\$
10	4	sissippi\$
11	6	ssippi\$
12	3	ssissippi\$

F		L
$\$_1$	$m_1 i_4 s_4 s_2 i_3 s_3 s_1 i_2 p_2 p_1$	i_1
i_1	$\$_1 m_1 i_4 s_4 s_2 i_3 s_3 s_1 i_2 p_2$	p_1
i_2	$p_2 p_1 i_1 \$_1 m_1 i_4 s_4 s_2 i_3 s_3$	s_1
i_3	$s_3 s_1 i_2 p_2 p_1 i_1 \$_1 m_1 i_4 s_4$	s_2
i_4	$s_4 s_2 i_3 s_3 s_1 i_2 p_2 p_1 i_1 \$_1$	m_1
m_1	$i_4 s_4 s_2 i_3 s_3 s_1 i_2 p_2 p_1 i_1$	$\$_1$
p_1	$i_1 \$_1 m_1 i_4 s_4 s_2 i_3 s_3 s_1 i_2$	p_2
p_2	$p_1 i_1 \$_1 m_1 i_4 s_4 s_2 i_3 s_3 s_1$	i_2
s_1	$i_2 p_2 p_1 i_1 \$_1 m_1 i_4 s_4 s_2 i_3$	s_3
s_2	$i_3 s_3 s_1 i_2 p_2 p_1 i_1 \$_1 m_1 i_4$	s_4
s_3	$s_1 i_2 p_2 p_1 i_1 \$_1 m_1 i_4 s_4 s_2$	i_3
s_4	$s_2 i_3 s_3 s_1 i_2 p_2 p_1 i_1 \$_1 m_1$	i_4

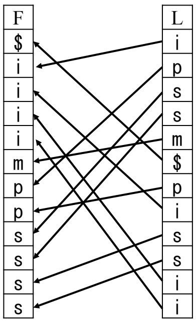

Figure 1 Example for SA (left), F, L (center), and LF function (right) of $T = mississippi\$$.

Our computation model is a unit-cost word RAM with a machine word size of $\Omega(\log_2 n)$ 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 $\log_2 n$ multiplicative factor. Throughout this paper, logarithm to base 2 is used if the logarithmic base is not indicated.

2.1 Suffix Array (SA) and SA interval

The suffix array (SA) [14] of string T is an integer array of size n such that $\mathsf{SA}[i]$ stores the starting position of i-th suffix of T in lexicographical order. Formally, SA is the permutation of $[1..n]$ such that $T[SA[1]..] \prec \cdots \prec T[SA[n]..]$ holds. For example, $T =$ **mississippi\$** and $\mathsf{SA} = 12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3$. The left figure in Figure 1 depicts the sorted suffixes of T and SA of T.

Since suffixes in the suffix array are sorted in the lexicographical order, suffixes with prefix Y occur continuously on an interval in the suffix array. We call this interval *SA interval* of Y. Formally, the SA interval of string Y is interval $[b, e]$ such that $p \in \mathsf{SA}[b..e]$ holds for all $p \in Occ(T, Y)$. For the above example, the SA intervals of **si** and **p** are $[9, 10]$ and $[7, 8]$, respectively.

2.2 BWT and backward search

The Burrows Wheeler Transform (BWT) [5] of string T is a permutation of T obtained as follows. We sort all the n rotations of T in lexicographical order and take the last character at each rotation in sorted order. L is the permutation of T such that for all $i \in [1..n]$, $L[i] = T[\mathsf{SA}[i] - 1]$ holds if $\mathsf{SA}[i] \neq 1$ and $L[i] = T[n]$ holds otherwise. Similarly, let F be a permutation of T that consists of the first characters in rotations in sorted order, i.e., $F[i] = T[\mathsf{SA}[i]]$ for all $i \in [1..n]$. The middle table in Figure 1 represents F, L, and sorted rotations of $T =$ **mississippi\$**.

A property of BWT is that the i-th occurrence of character c in F corresponds to the i-th occurrence of c in L. In other words, let x and y be the positions of the i-th occurrence of c in F and L, respectively. Then $F[x]$ is a character of position p in T when $L[y]$ is a character of the same position in T. The LF function receives a position y in L as input and returns such corresponding position x in F. Since F consists of the sorted characters, $\mathsf{LF}(y) = C[L[y]] + \mathsf{rank}(L, L[y], i)$ holds for all $y \in [1, n]$, where $C[c]$ is the number of occurrences of characters lexicographically less than $c \in \Sigma$ in L.

Backward search computes the SA interval of cY for a given SA interval of Y and character c using the BWT of T. Function $\mathsf{backward\_search}(T, b, e, c)$ takes SA interval $[b, e]$ of Y and character $c \in \Sigma$ as input and returns SA interval $[b', e']$ of cY. We compute backward search using LF, rank, and select queries on L. The LF function receives the first and last occurrences of c in $L[b..e]$ and returns the first and last positions of the SA interval of cY because $L[i]$ represents the character preceding $F[i]$ on T for an integer $i \in [2..n]$. We can compute the first and last occurrences of c using rank and select queries on L.

Formally, let $x = \mathsf{rank}(L, c, b-1)$ and $y = \mathsf{rank}(L, c, e)$; $b' = \mathsf{LF}(\mathsf{select}(L, c, x+1))$ and $e' = \mathsf{LF}(\mathsf{select}(L, c, y))$ hold if the length of the SA interval for cY is not zero.

2.2.1 Run-length encoding and RLBWT

For a string T, *Run-length encoding* $\mathsf{RLE}(T)$ is a partition of T into substrings $f_1, f_2, \ldots, f_r$ such that each f_i is a maximal repetition of the same character in T. We call each f_i a *run*.

The RLBWT of T is the BWT encoded by the run-length encoding, i.e., $\mathsf{RLE}(L)$. The RLBWT is stored in $O(r)$ space because each run in the RLBWT can be encoded into a pair of integers c and ℓ, where c is the character and ℓ is the length of the run. We call such a representation the *compressed form* of the RLBWT.

2.3 LZ77

For a string T, LZ77 parsing [13] of the reversed T greedily partitions T into substrings (phrases) $f_z, f_{z-1}, \ldots, f_1$ in right-to-left order such that each phrase is either (i) copied from a subsequent substring in T (target phrase) or (ii) an explicit character (character phrase). We denote LZ77 phrases of the reversed T as $\mathsf{LZ}(T^R)$.

Formally, let i' be the ending position of f_i for $i \in [1, z]$, i.e., $i' = |f_{i'-1} \cdots f_1| + 1$. Then $f_1 = T[n]$, and f_i is $T[i']$ for $i \in [2, z]$ if $T[i']$ is a new character (i.e., $Occ(T[i'+1..], T[i']) = \emptyset$); otherwise, f_i is the longest suffix P of $T[..i']$, which has right occurrences in T (i.e., $|P| = \max\{\ell \mid Occ(T[i' - \ell + 2..], T[i' - \ell + 1..i']) \neq \emptyset, \ell \in [1, i']\}$).

We can store LZ77 phrases in $O(z)$ space because we encode a target phrase into the pair $\langle p, \ell \rangle$ of the right occurrence p and length ℓ of the phrase. We call such representation the *compressed form* of LZ77. For example, let $T = cbbbbbabaababa$. Then $\mathsf{LZ}(T^R) = c, bbbb, baba, aba, b, a$, and the compressed form of $\mathsf{LZ}(T^R)$ is $c, \langle 3, 4 \rangle, \langle 11, 4 \rangle, \langle 12, 3 \rangle, b, a$.

3 Policriti and Prezza's conversion algorithm

Policriti and Prezza's conversion algorithm [15] converts a compressed string of T in the RLBWT format to another compressed string of T^R in the LZ77 format while using data structures in $O(r)$ space. The data structures support four queries: backward search, the LF function, access queries on the RLBWT L, and RMTQ on the suffix array of T. The $\mathsf{RMTQ}(D, i, j, k)$ takes value k, interval $[b, e]$, and array D as inputs and returns a value larger than k on interval $[b, e]$ in D.

The algorithm extracts the original string from L in right-to-left order using the LF function and access queries on L and computes LZ77 phrases sequentially using backward search and RMTQ. In each step, the algorithm extracts a suffix of the original string (i.e., current extracted string) and it outputs the LZ77 phrase called *current pattern* in the suffix. In each step, the following two conditions for the current pattern are guaranteed: (i) the current pattern has at least one right occurrence or is the string of length 0; (ii) the length of the current pattern is no less than that of the following current pattern (i.e., the next computed LZ77 phrase).

For computing the current extracted string in each step, the algorithm computes (i) the next character preceding the current extracted string, (ii) computes the SA interval corresponding to the current pattern using the backward search, and (iii) finds any right

```
Algorithm 1: Policriti and Prezza's conversion algorithm.
  Data: RLBWT $L$ of $T$ and position $y$ of $T[n]$ on $L$
  Result: $\mathsf{LZ}(T^R)$
  $(b, e, p, \ell, x) \leftarrow (1, n, -1, 1, n)$;                          /* Initialization */
  while $x \geq 1$ do
      /* Let P be T[x..x+l-1]                                                  */
      $c \leftarrow \mathsf{access}(L, y)$;                                   /* Access T[x] */
      $[b', e'] \leftarrow \mathsf{backward\_search}(T, b, e, c)$;   /* Compute the SA interval of P */
      $p' \leftarrow \mathsf{RMTQ}(\mathsf{SA}, b', e', x + 1)$;
      if $p' = -1$ then                        /* Any right occurrence of P was not found */
          if $\ell > 1$ then
              output $\langle p, \ell - 1 \rangle$;
          else
              output $c$;
              $(x, y) = (x - 1, \mathsf{LF}(y))$;
          $(b, e, p, \ell) \leftarrow (1, n, p', 1)$;
      else
          $(b, e, p, \ell, x, y) \leftarrow (b', e', p', \ell + 1, x - 1, \mathsf{LF}(y))$;
```

occurrence of the current pattern in the SA interval using RMTQ. If such right occurrence of the extended pattern does not exist, the algorithm outputs the current pattern as the next LZ77 phrase. When the length of the current pattern is zero, the next phrase is the next character. The algorithm repeats the above step until it extracts the whole string and outputs LZ77 phrases of T^R. Algorithm 1 shows a pseudo code of the algorithm.

The algorithm uses two dynamic data structures: one for supporting backward search, the LF function, and access queries on L in $O(\log r)$ time and $O(r)$ space; the other for supporting RMTQ in $O(\log r)$ time and $O(r)$ space, which is detailed in the next subsection.

3.1 RMTQ data structure

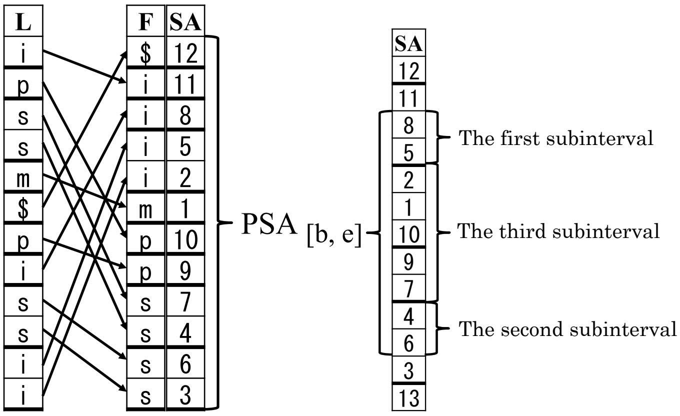

Figure 2 Left figure illustrates partitioned suffix array (PSA). Bold horizontal lines on suffix array represent partitions on PSA; hence, PSA is $(12), (11), (8), (5, 2), (1), (10), (9), (7, 4), (6, 3)$. Right figure illustrates three subintervals used in Section 4.1.

Policriti and Prezza presented an RMTQ data structure for fixed k, which can be updated when k is decremented. The construction for RMTQ data structure partitions the suffix array of T into subarrays for every run in L by the LF function, resulting in r subarrays in total. Since the i-th occurrence of any character c in F corresponds to the i-th occurrence of c in L, the characters on every run in L also occur continuously on F. The F can be partitioned into r substrings such that each substring corresponds to a run in L. We call such subarrays *partitioned suffix arrays* (PSAs). The left in Figure 2 shows an example for the PSA of T in Figure 1.

The data structure does not store the whole PSA. Instead, it stores only the first and last values larger than k on each subarray of the PSA and their corresponding positions on it. The red-black tree is used to store those positions. We call such a first position (respectively, last position) on the i-th subarray for fixed k a *k-open position* (respectively, *k-close position*) on the i-th subarray, which is denoted as $\mathsf{open}(i,k)$ (respectively, $\mathsf{close}(i,k)$).

$\mathsf{RMTQ}(D,b,e,k)$ using the data structure is divided into two cases according to the relationship between the query interval $[b,e]$ and PSA: (A) there exists a subarray $\mathsf{SA}[p..q]$ of the PSA completely including interval $[b,e]$ (i.e., $p < b \leq e < q$ holds); and (B) such a subarray does not exist. Both cases are detailed as follows.

For case (B), the query interval is partitioned into several subarrays of the PSA and contains either a prefix or suffix of each subarray. Therefore, the interval contains at least one of the k-open and k-close positions if and only if the interval contains a value larger than k. We select an answer of RMTQ from the k-open and k-close positions on subarrays in the PSA in $O(\log r)$ time using the red-black tree storing the set of k-open and k-close positions.

For case (A), $\mathsf{RMTQ}(D,i,j,k)$ is computed using its computation result in the previous iteration of Algorithm 1. The query interval $[b,e]$ represents the SA interval of string P, and the length of P is at least two because the SA interval of a character does not satisfy case (A). This means that the query interval in the previous iteration represents the SA interval of $P[2..]$, and Algorithm 1 computes the answer p ($\neq -1$) in the previous iteration. Thus, in case (A), $(p-1)$ is the answer for P because $P[1]$ is the character on L such that p is at the same position on the suffix array.

Formally, let $\mathsf{RLE}(L) = L_1, L_2, \ldots, L_r$, $p(i)$ be the starting position of L_i in L (i.e., $p(i) = |L_1 \cdots L_i| - |L_i| + 1$), and X be the permutation of $[1..r]$ such that $\mathsf{LF}(p(X[1])) < \ldots < \mathsf{LF}(p(X[r]))$ holds. Then the PSA of T is r subarrays $s_1, \ldots, s_r$ such that $s_i = \mathsf{SA}[\mathsf{LF}(p(X[i]))..\mathsf{LF}(p(X[i])) + |L_{X[i]}| - 1]$ holds for all $i \in [1,r]$. Let $\mathsf{open}(i,k) = \min(\{n+1\} \cup \{j \mid \mathsf{SA}[j] \geq k, j \in [s_i..s_{i+1}-1]\})$ and $\mathsf{close}(i,k) = \max(\{0\} \cup \{j \mid \mathsf{SA}[j] \geq k, j \in [s_i..s_{i+1}-1]\})$ for $k \in [1,n]$ and $i \in [1,r]$. Let $\mathcal{T}_k$ be the set of k-open and close positions in the suffix array of T, i.e, $\mathcal{T}_k = \{\mathsf{open}(1,k), \mathsf{close}(1,k) \ldots, \mathsf{open}(r,k), \mathsf{close}(r,k)\}$. Then the following lemma holds.

▶ **Lemma 1** ([15]). *Let P be a substring of T starting at a position k and $\mathsf{SA}[b..e]$ be the SA interval of P. (1) In case (A), the length of P is at least 2 and $\mathsf{RMTQ}(SA,b,e,k)$ can return $\mathsf{RMTQ}(SA,b',e',k+1)-1$, where $\mathsf{SA}[b..e]$ is the SA interval of $P[2..]$. (2) In case (B), if $\mathcal{T}_k \cap [b..e] \neq \emptyset$ holds, then $\mathsf{RMTQ}(SA,b,e,k)$ can return the value at any position in $\mathcal{T}_k \cap [b..e]$; otherwise $\mathsf{RMTQ}(\mathsf{SA},b,e,k) = -1$ holds.*

4 Data structures for faster conversion

We improve the query time in the data structure for backward search, the LF function, access query on L, and RMTQ for case (B) by presenting two novel data structures: one supports the RMTQ for case (B); and the other supports backward search, the LF function, and access

Algorithm 2: Our $\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$ algorithm for case (B).

```
Result: $\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$
$\hat{b} \leftarrow \mathsf{pred}(Z, b)$;    /* Get the index $\hat{b}$ of the subarray on the position b */
$\hat{e} \leftarrow \mathsf{pred}(Z, e)$;    /* Get the index $\hat{e}$ of the subarray on the position e */
$(x^{close}, v^{close}) \leftarrow M_k^{close}[\hat{b}]$;
$(x^{open}, v^{open}) \leftarrow M_k^{open}[\hat{e}]$;
$v^{\circ} \leftarrow \mathsf{RMQ}(M, \hat{b}+1, \hat{e}-1)$;
if $x^{close} \in [b, e]$ then
    return $v^{close}$;
else if $x^{open} \in [b, e]$ then
    return $v^{open}$;
else if $v^{\circ} > k$ and $\hat{b}+1 \leq \hat{e}-1$ then
    return $v^{\circ}$;
else
    return $-1$;
```

query on L. Our data structures use static predecessor data structures internally and improve four query times by choosing the predecessor data structure with the fastest (estimated) query time. Finally, we show the results of our data structures, which are summarized in Table 1.

4.1 RMTQ data structure in case (B)

Our RMTQ data structure for fixed k consists of four arrays of length r: *k-open array* M_k^{open}, *k-close array* M_k^{close}, *max value array* M, and *starting position array* Z. Data structures for the RMQ and predecessor query are built on M and Z, respectively.

The i-th element of the k-open array (respectively, k-close array) stores the pair of k-open position (respectively, k-close position) and its corresponding value on the i-th subarray of the PSA. The k-open and k-close arrays can be updated when k is decremented. The i-th element of the max value array M stores the maximum value on the i-th subarray of the PSA. The i-th element of the starting position array stores the starting position of the i-th subarray on the PSA (i.e., $Z[i] = p(X[i])$).

Our algorithm for RMTQ ($\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$ algorithm) consists of three parts: (i) it divides a given query interval into at most three subintervals; (ii) computes the RMTQ for each subinterval; and (iii) returns the final answer of the RMTQ for a given interval using answers for subintervals. Those three subintervals are defined as follows: the first subinterval is on the first subarray of the PSA in the query interval, the second subinterval is on the last subarray of the PSA in the query interval, and the third subinterval is on the remaining subarrays. The right figure in Figure 2 illustrates those three subintervals. We compute the three subintervals using predecessor queries on Z for a given query interval.

The first subinterval is on a suffix of the first subarray; hence, the first subinterval has a value larger than k if and only if the subinterval contains the k-close position of the first subarray. Similarly, the second subinterval has a value larger than k if and only if the subinterval contains the k-open position of the last subarray. The third subinterval is on middle subarrays; hence, the third subinterval has a value larger than k if and only if the maximal value is larger than k on the subarrays. Therefore, we compute the RMTQ for the first subinterval (respectively, the second subinterval) by accessing the element of the first subarray on the k-close array (respectively, the element of the last subarray on the k-open array). We also compute the RMTQ for the third subinterval using the RMQ whose query interval covers the third subinterval on M.

Algorithm 2 shows a pseudo code of our RMTQ algorithm. Since we can compute the RMQ in constant time (e.g., [7]), the query time of our RMTQ algorithm depends on the performance of predecessor queries on Z. We can also convert k-open and close arrays into $(k-1)$-open and close arrays by changing at most two elements because the conversion can change the only element of the subarray containing k.

Formally, let $\hat{b}$ and $\hat{e}$ be the ranks of the subarray of the PSA of T which contains the positions b and e, respectively, i.e., $\hat{b} = \mathsf{pred}(Z, b)$ and $\hat{e} = \mathsf{pred}(Z, e)$. For $k \in [1, n]$, let M_k^{open} be the array of size r such that for $i \in [1, r]$, $M_k^{open}[i] = (\mathsf{SA}[\mathsf{open}(i, k)], \mathsf{open}(i, k))$ if $\mathsf{open}(i, k) \neq n + 1$ holds; otherwise $M_k^{open}[i] = (-1, \mathsf{open}(i, k))$. Similarly, let M_k^{close} be the array of size r such that for $i \in [1, r]$, $M_k^{close}[i] = (\mathsf{SA}[\mathsf{close}(i, k)], \mathsf{close}(i, k))$ if $\mathsf{close}(i, k) \neq 0$; otherwise $M_k^{close}[i] = (-1, \mathsf{close}(i, k))$. Let M be the integer array of length r such that $M[i]$ stores the maximal value in the i-th subarray of the PSA of T, i.e., $M[i] = \mathsf{RMQ}(\mathsf{SA}, p(i), p(i + 1) - 1)$ for all $i \in [1, r]$. Let $Q(m, u)$ be the query time of the predecessor query on a set S of size m from a universe $[1, u]$ by a predecessor data structure of $O(m)$ space. Then the following lemmas hold.

▶ **Lemma 2.** *In case (B),* $\mathsf{RMTQ}(\mathsf{SA}, b, e, k) \neq -1$ *holds if and only if there exists an answer of* $\mathsf{RMTQ}(\mathsf{SA}, b, e, k)$ *in* $M_k^{open}[\hat{b}], M_k^{close}[\hat{e}]$*, or* $\mathsf{RMQ}(M, \hat{b} + 1, \hat{e} - 1, k)$*.*

▶ **Lemma 3.** *Our data structure for case (B) can compute* $\mathsf{RMTQ}(SA, b, e, k)$ *in* $O(1+Q(r, n))$ *time. The space usage is* $O(r)$ *space.*

Proof. We construct the predecessor data structure for Z, which supports predecessor queries in $Q(r, n)$ time, and the RMQ data structure for M which supports the RMQ in $O(1)$ time using [7]. Then Lemma 3 holds by Algorithm 2. ◀

▶ **Lemma 4.** *For a given position x of k on the suffix array of T (i.e.,* $k = \mathsf{SA}[x]$*), we can convert M_k^{open} and M_k^{close} into M_{k-1}^{open} and M_{k-1}^{close} in $O(1 + Q(r, n))$ time using the predecessor data structure for Z.*

Proof. $M_k^{open}[i] = M_{k-1}^{open}[i]$ and $M_k^{close}[i] = M_{k-1}^{close}[i]$ hold for all $i \in ([1, r] \setminus \{p\})$, where $p = \mathsf{pred}(Z, x)$. Therefore, we appropriately update $M_k^{open}[p]$ and $M_k^{open}[p]$ for $k - 1$. ◀

4.2 Data structure for backward search, LF, and access queries

We leverage the static data structures presented by Gagie et al. [8] for backward search, the LF function, and access queries on L instead of Policriti and Prezza's dynamic data structure. The static data structures compute three queries by executing only a constant number of predecessor queries, and the three queries using the static data structures can be faster than those using Policriti and Prezza's dynamic data structure.

Since the time for the predecessor query on the static data structures is proportional to the alphabet size of the input RLBWT, the alphabet size slightly increases the query times. For faster queries, we replace one of the static data structures for the predecessor queries depending on the alphabet size with an array of size σ. We obtain the following lemma.

▶ **Lemma 5.** *Let $C(m,u)$ be the construction time for the predecessor data structure of $O(m)$ space, which supports predecessor queries in $Q(m,u)$ time. We can construct the data structure of $O(r+\sigma)$ space, which supports* backward_search, LF, *and* access *queries for L in $O(1+Q(r,n))$ time, by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. See Appendix. ◀

4.3 Improved Policriti and Prezza's algorithm

Algorithm 1 using our data structures requires $O(r+\sigma)$ space, which can be $\omega(r)$ when $\sigma \geq r$, e.g., $\sigma = n^2$. To bound the space usage to $O(r)$, we reduce the alphabet size of the RLBWT L to at most r by renumbering characters in L.

We modify Algorithm 1 as follows: (i) We replace each character c in L with the rank of c in L (i.e., $|\{L[i] \mid L[i] \leq c, i \in [1,n]\}|$) and construct the new RLBWT L' over the alphabet of at most r. We call the converted string the *shrunk string* of L. (ii) We convert L' into LZ77 phrases of the string T'^R recovered from L' using Algorithm 1. (iii) We recover the LZ77 phrases of T^R from that of T'^R using the *inverse array* W, where the i-th element of the array stores the original character of rank i (i.e., $W[L'[i]] = L[i]$ holds for any $i \in [1,n]$). The modified algorithm works correctly because (1) our backward search queries receive only characters that appear in the RLBWT, (2) L' is the RLBWT of the shrunk string of T, and (3) the form of LZ77 phrases is independent of the alphabet, i.e., we obtain the i-th LZ77 phrase of T^R by mapping characters in the i-th LZ77 phrase of T'^R into the original characters.

Finally, we obtain a conversion algorithm from RLBWT into LZ77 in $O(n(1+Q(r,n)) + C(r,n))$ time and $O(r)$ space, and the algorithm depends on the performance of the static predecessor data structure. There exist two predecessor data structures such that (1) $Q(r,n) = O(\sqrt{\log r/\log\log r})$ and $C(r,n) = O(r\sqrt{\log r/\log\log r})$ hold [3] and (2) $Q(r,n) = O(\log\log n)$ and $C(r,n) = O(r)$ hold [6]. Since we can compute r and n by processing the RLBWT in $O(r)$ time, we choose the faster predecessor data structure between those predecessor data structures. Therefore, we obtain the result of our data structures, is listed in Table 1.

Formally, the following lemmas and theorem hold.

▶ **Lemma 6.** *The following statements hold. (1) We can compute the shrunk string L' of L and the inverse array W in $O(r)$ time and working space. (2) The L' is the RLBWT of the shrunk string T' of T. (3) The $|\mathsf{LZ}(T'^R)| = |\mathsf{LZ}(T^R)|$ and $\mathsf{LZ}(T^R)[i][j] = W[\mathsf{LZ}(T'^R)[i][j]]$ hold for $i \in [1,z]$ and $j \in [1, |\mathsf{LZ}(T^R)[i]|]$, where z is the number of LZ77 phrase of T^R. (4) We can convert the compressed form of $\mathsf{LZ}(T'^R)[i]$ into that of $\mathsf{LZ}(T^R)[i]$ in constant time using W for all $i \in [1,z]$.*

Proof. (1) We construct the string E that consists of r first characters in the run-length encoding of L (i.e., $E = \mathsf{RLE}(L)[1][1], \mathsf{RLE}(L)[2][1], \ldots, \mathsf{RLE}(L)[r][1]$) and construct the suffix array of E in $O(r)$ time and working space [11]. We construct the shrunk string E' of E and W using the suffix array and construct L' using E'. (2) Let SA and SA' be the suffix arrays of T and T'. Then $\mathsf{SA}[i] = \mathsf{SA}'[i]$ holds for all $i \in [1,n]$. Therefore, the RLBWT of T' is L'. (3) This holds because $Occ(T, T[x..y]) = Occ(T', T'[x..y])$ holds for two integers $1 \leq x \leq y \leq n$. (4) If $\mathsf{LZ}(T'^R)[i]$ is a target phrase, we return the phrase as $\mathsf{LZ}(T^R)[i]$. Otherwise, we return $W[\mathsf{LZ}(T'^R)[i]]$ as $\mathsf{LZ}(T^R)[i]$. ◀

▶ **Lemma 7.** *We can construct the data structure of Lemma 3 in $O(n(1+Q(r,n))+C(r,n)+\sigma)$ time and $O(r+\sigma)$ working space using the RLBWT data structure of Lemma 5.*

Proof. See Appendix. ◀

▶ **Theorem 8.** *There exists a conversion algorithm from RLBWT to LZ77 in $O(n(1+Q(r,n))+C(r,n))$ time and $O(r)$ space.*

Proof. We already have described our algorithm in Section 4.3. Each step of Algorithm 1 additionally needs to update M_k^{open}, M_k^{close} and determine either case (A) or (B) for a given query interval. The total time is $O(1+Q(r,n))$ using predecessor queries on L and Lemma 4. Therefore, Theorem 8 holds by Lemmas 3, 5, 6, and 7. ◀

5 Conclusion

We presented a new conversion algorithm from RLBWT into LZ77 in $O(n(1+Q(r,n))+C(r,n))$ time and $O(r)$ space. By leveraging the fastest static predecessor data structure using $O(r)$ space, we obtain the conversion algorithm that runs in $O(n\min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$ time. This result improves the previous result in $O(n\log r)$ time and $O(r)$ space.

We have the following open problem: can we achieve the conversion from RLBWT into LZ77 in $O(n)$ time and $O(r)$ space? It is difficult to achieve the $O(n)$ time complexity with our approach because any predecessor data structure for a set using $m^{O(1)}$ words of $(\log|U|)^{O(1)}$ bits requires $\Omega(\sqrt{\log m/\log\log m})$ query time in the worst case [3], where m is the number of elements in the set and U is the universe of elements. Kempa's conversion algorithm can run faster than our algorithm, but it requires $\omega(r)$ working space in the worst case. Thus, a new approach is required to solve this open problem.

References

1 Hideo Bannai, Pawel Gawrychowski, Shunsuke Inenaga, and Masayuki Takeda. Converting SLP to LZ78 in almost Linear Time. In *Proceedings of CPM*, pages 38–49, 2013.

2 Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 Factorization of Grammar Compressed Text. In *Proceedings of SPIRE*, pages 86–98, 2012.

3 Paul Beame and Faith E. Fich. Optimal Bounds for the Predecessor Problem and Related Problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

4 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

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

6 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proceedings of CPM*, pages 160–171, 2015.

7 Johannes Fischer and Volker Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

8 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-Time Text Indexing in BWT-runs Bounded Space. In *Proceedings of SODA*, pages 1459–1477, 2018.

9 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for Searchable Partial Sums with optimal worst-case performance. *Theor. Comput. Sci.*, 412(39):5176–5186, 2011.

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

11 Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of ICALP*, pages 943–955, 2003.

12 Dominik Kempa. Optimal Construction of Compressed Indexes for Highly Repetitive Texts. In *Proceedings of SODA*, pages 1344–1357, 2019.

13 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

14 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

15 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

16 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

17 Kensuke Sakai, Tatsuya Ohno, Keisuke Goto, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. RePair in Compressed Space and Time. *CoRR*, abs/1811.01472, 2018.

18 Dan E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983.

A The proof of Lemma 5

We can compute LF and $\mathsf{backward\_search}$ queries using C, rank, select, access queries for L and construct C by processing the RLBWT of T in $O(r+\sigma)$ time and working space. Therefore, we give the data structures for rank, select, and access queries for L by the following lemmas.

▶ **Lemma 9.** *We can construct the data structure of $O(r+\sigma)$ space that supports* rank *queries for L in $O(1+Q(r,n))$ time by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. We compute rank queries for a character c by the constant number of accessing elements on two arrays B_c and V_c and the constant number of predecessor queries on B_c. Array B_c is the array storing sorted starting positions of runs of c in L, and V_c is the integer array such that $V_c[i]$ stores the rank of the first character of the i-th run of character c. We can construct $B_1, \ldots, B_\sigma$ and $V_1, \ldots, V_\sigma$ in $O(r+\sigma)$ time and working space by processing the RLBWT of T and predecessor data structures for rank queries in $O(1+C(r,n))$ time. Therefore Lemma 9 holds.

Formally, let $\mathsf{run}(c)$ be the number of runs of character c in L (i.e., $\mathsf{run}(c) = |\{i \mid L_i[1] = c, i \in [1,r]\}|$). Array $B_c[i]$ stores the starting position of the i-th run of character c for all $c \in \Sigma$ and $i \in [1, \mathsf{run}(c)]$, $V_c[i] = \mathsf{rank}(L, c, B_c[i])$ for $c \in \Sigma$ and $i \in [1, \mathsf{run}(c)]$ and $V_c[\mathsf{run}(c)+1] = \mathsf{rank}(L, c, n) + 1$. If $L[x] = c$ holds, then $\mathsf{rank}(L, c, x) = V_c[t] + x - B_c[t]$ holds; otherwise, $\mathsf{rank}(L, c, x) = V_c[t+1] - 1$ holds, where $t = \mathsf{pred}(B_c, x)$. Since $V_c[i+1] - V_c[i]$ represents the length of the i-th run of character c, we can compute $L[x] = c$ using predecessor queries, i.e., $x - B_c[t] + 1 \leq \ell$ holds if and only if $L[x] = c$ holds, where $\ell = V_c[\mathsf{pred}(B_c, x) + 1] - V_c[\mathsf{pred}(B_c, x)]$. ◀

▶ **Lemma 10.** *We can construct the data structure of $O(r+\sigma)$ space that supports* select *queries for L in $O(1+Q(r,n))$ time by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. We compute select queries for a character c using three arrays C, V_c, and B_c and the predecessor on V_c. When $\mathsf{select}(L, c, x) \neq n+1$ holds, $\mathsf{select}(L, c, x) = B_c[\mathsf{pred}(V_c, x)] + x - V_c[\mathsf{pred}(V_c, x)]$ holds. Since $C[c+1] - C[c]$ represents the number of cs in L, we can compute

$\mathsf{select}(L, c, x) \neq n + 1$ using C. We can construct $B_1, \ldots, B_\sigma$ and $V_1, \ldots, V_\sigma$ in $O(r + \sigma)$ time and working space by processing the RLBWT of T and predecessor data structures for select queries in $O(1 + C(r, n))$ time. Therefore Lemma 10 holds. ◀

▶ **Lemma 11.** *We can construct the data structure of $O(r)$ space that supports access queries for L in $O(1 + Q(r, n))$ time by processing the RLBWT of T in $O(C(r, n) + r)$ time and $O(r)$ working space.*

Proof. We compute access queries using two arrays B and E and the predecessor on B. Array B is the sorted starting positions of runs in L (i.e., $B = p(1), p(2), \ldots, p(r)$), and E is the first characters of runs in L (i.e., $E = \mathsf{RLE}(L)[1][1], \mathsf{RLE}(L)[2][1], \ldots, \mathsf{RLE}(L)[r][1]$). We can construct B and E in $O(r)$ time by processing the RLBWT. Since we can compute $L[i]$ by $E[\mathsf{pred}(B, i)]$ for a given integer i, Lemma 11 holds. ◀

Therefore Lemma 5 holds by Lemmas 9, 10, and 11.

B The proof of Lemma 7

Proof. Our data structure for RMTQ consists of $M, M_n^{open}, M_n^{close}, Z$, the RMQ data structure for M, and the predecessor data structure for Z. We construct Z using *c-integer sequence*. The c-integer sequence is the subarray of Z such that subarray $Z[b..e]$ is on the run of a character c in F, i.e., $b = \min\{X[u] \mid L[p(u)] = c, u \in [1, r]\}$ and $e = \max\{X[u] \mid L[p(u)] = c, u \in [1, r]\}$. We construct 1-integer sequence, …, σ-integer sequence in $O(r(1 + Q(r, n)) + \sigma)$ using the LF function since $\mathsf{LF}(p(i)) < \mathsf{LF}(p(j))$ holds for two positions i and j such that $L[i] = L[j]$ and $i < j$ hold. We obtain Z by concatenating the sequences. The total time is $O(r(1 + Q(r, n)) + \sigma)$ and the working space is $O(r + \sigma)$.

We construct M using the LF function and predecessor on Z in $O(1 + Q(r, n))$ time since $\mathsf{LF}(i)$ represents a position on the suffix array of T for $i \in [1, n]$. We construct M_n^{open} and M_n^{close} in $O(r)$ time since $M_n^{open} = (-1, n + 1), \ldots, (-1, n + 1)$ and $M_n^{close} = (-1, 0), \ldots, (-1, 0)$. We construct the RMQ data structure for M in $O(r)$ time and working space using [7]. Therefore Lemma 7 holds since $r \leq n$. ◀