Abstract

Lossless data compression has been widely studied in computer science. One of the most widely used lossless data compressions is Lempel-Zip (LZ) 77 parsing, which achieves a high compression ratio. Bidirectional (a.k.a. macro) parsing is a lossless data compression and computes a sequence of phrases copied from another substring (target phrase) on either the left or right position in an input string. Gagie et al. (LATIN 2018) recently showed that a large gap exists between the number of smallest bidirectional phrases of a given string and that of LZ77 phrases. In addition, finding the smallest bidirectional parse of a given text is NP-complete. Several variants of bidirectional parsing have been proposed thus far, but no prior work for bidirectional parsing has achieved high compression that is smaller than that of LZ77 phrasing for any string. In this paper, we present the first practical bidirectional parsing named LZ77 parsing with right reference (LZRR), in which the number of LZRR phrases is theoretically guaranteed to be smaller than the number of LZ77 phrases. Experimental results using benchmark strings show the number of LZRR phrases is approximately five percent smaller than that of LZ77 phrases.

1 Introduction

Lossless data compression has been widely studied in computer science. One of the most widely used lossless data compressions is Lempel-Zip (LZ) 77 parsing, which compresses a given string by computing a sequence of phrases copied from the longest substring on the left position in an input string. LZ77 parsing has a long research history, with the first paper on it published in 1976 [8]. Many LZ77’s extensions have since been proposed (e.g., [7, 3, 10]), and LZ77 parsing achieves the smallest compression ratio among them.

Bidirectional (a.k.a. macro) parsing [11] is a lossless data compression and computes a sequence of phrases copied from another substring (target phrase) on either the left or right position in an input string. Each set of LZ77 phrases is convertible into a set of bidirectional phrases, and the number of phrases in the smallest bidirectional parsing is less than that of LZ77 phrases. Gagie et al. [2] recently showed the number of LZ77 phrases $z$ representing an input string of length $n$ can be tightly bounded by the smallest number of bidirectional phrases $b^*$ representing the same string as $z = O(b^* \log(n/b^*))$, which suggests that a large gap exists between $b^*$ and $z$. In addition, finding the smallest bidirectional parse of a given text is NP-complete [11]. Thus, an important open challenge is to develop a polynomial time bidirectional parsing such that the number of bidirectional phrases is smaller than that of LZ77 phrases.

Several variants of bidirectional parsing have been proposed thus far. Lex-parsing [9] is a bidirectional parsing that computes a sequence of bidirectional phrases that each occurred previously on a suffix array of a string. The number of phrases $v$ in the lex-parsing is bounded by $v = O(b^* \log(n/b^*))$ [2]. Although the lex-parsing is effective for most benchmark strings (i.e., phrases $v$ is very close to $z$) in practice, it can fail to compress some strings (i.e., $v$ is much larger than $z$) [9]. Lcpcomp [1] and a bidirectional parsing using Burrows-Wheeler transform (BWT) [2] have also been proposed, and they never have fewer phrases than lex-parse [9]. Kempa and Prezza proposed a parsing algorithm for computing the bidirectional parse of an input string for a given string attractor of the string [6]. The number of the bidirectional phrases is bounded by $O(\gamma \log(n/\gamma))$, where $\gamma$ is the size of the string attractor. Let $\gamma^*$ be the size of the smallest string attractor for a given string. Then $b^* = O(\gamma^* \log(n/\gamma^*))$ holds [6]. In addition, finding the smallest string attractor of a given string is also NP-complete [6]. In summary, no prior bidirectional parsing achieves high compression that is smaller than that of LZ77 phrasing for any string.
In this paper, we present the first practical bidirectional parsing named \textit{LZ77 parsing with right reference (LZRR)} in which the number of LZRR phrases is always smaller than the number of LZ77 phrases by a large margin. LZRR is a polynomial time algorithm that greedily computes phrases from a string in the left-to-right order the same as LZ77. The main difference between LZRR and LZ77 is the way to compute their phrases. Whereas LZ77 parsing chooses the longest substring occurring previously as a phrase, LZRR parsing uses not only previous occurrences of each phrase but also subsequent occurrences (i.e., it chooses the longest substring occurring previously or subsequently as a phrase). For this reason, the number of LZRR phrases is theoretically guaranteed to be no more than that of LZ77 phrases. Experimental results using benchmark datasets show the number of LZRR phrases is approximately five percent smaller than that of LZ77 phrases.

## 2 Preliminaries

Let $\Sigma$ be an ordered alphabet of size $\sigma$, $T$ be a string of length $n$ over $\Sigma$ 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 $j$. $T[i..]$ 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., $T^R = T[n]T[n-1]\cdots T[1]$.

$Occ(T,s)$ denotes all the occurrence positions of string $s$ in string $T$, i.e., $Occ(T,s) = \{i \mid s = T[i,i+|s|-1], 1 \leq i \leq n - |s| + 1\}$. Let $lcp(i,j)$ be the length of the longest common prefix (LCP) of $T[i..]$ and $T[j..]$. For two strings $x$ and $y$, $x < y$ represents that $x$ is lexicographically smaller than $y$, we write $x \prec y$. Similarly, for a string $z$, $z \leq y$ represents that the LCP of $x$ and $z$ is equal to or longer than that of $y$ and $z$. For example, $aab \leq_{aa} ab$.

Our model of computation 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.

### 2.1 Arrays

**Suffix array** SA, **inverse suffix array** ISA, **LCP array** LCP, **longest previous factor array** LPF, and **sorted suffix array** SA are integer arrays of length $n$ for a string $T$, respectively. SA is the permutation of $[1..n]$ such that $T[SA[1..]] \prec \cdots \prec T[SA[n..]]$ holds. ISA is the permutation of $[1..n]$ such that $SA[ISA[i]] = i$ holds for any $i \in \{1, 2, \ldots, n\}$. $LCP[1] = 0$ and $LCP[i] = lcp(SA[i], SA[i-1])$ for $i \in \{2, 3, \ldots, n\}$. LPF[i] stores the length of the longest prefix of $T[i..]$ occurring previously; that is $LPF[i] = \max\{\text{lcp}(i, p) \mid p \in \{1..i-1\}\}$, where max returns the maximal element of a given set. SA is the sorted starting positions of suffixes in decreasing order for the length of the LCP with $T[k..]$. Formally, for an integer $k \in \{1, 2, \ldots, n\}$, $SA_k$ is a permutation of $[1..n]$ such that $T[SA_k[1..]] \leq_{T[k..]} \cdots \leq_{T[k..]} T[SA_k[n..]]$. $SA_k$ is not unique when there exist two positions $i$ and $j$ such that $\text{lcp}(k, i) = \text{lcp}(k, j)$.

For $T = abababaabbb$, $SA = 7, 5, 3, 1, 8, 10, 6, 4, 2, 9$, $ISA = 4, 9, 3, 8, 2, 7, 1, 5, 10, 6$, $LCP = 0, 1, 3, 5, 2, 0, 1, 2, 4, 1$, $LPF = 0, 0, 5, 4, 3, 2, 1, 2, 1, 1$, and $SA_1 = 1, 3, 5, 8, 7, 10, 6, 4, 2, 9$.

### 2.2 Union-find data structure

**Union-find** is a data structure for disjoint sets and supports the following operations for disjoint set $D$: MakeSet, Union, Find. MakeSet adds element $\{m + 1\}$ into $D$ and returns the integer where $m$ is the cardinality of $D$. Union($x$, $y$) merges two sets $X, Y \in D$ containing $x$ and $y$, respectively; it adds a new set $X \cup Y$ into $D$; it removes $X$ and $Y$ from $D$. The Find($x$) returns the id of the set containing $x$ in $D$. The union-find data structure performs MakeSet, Union, Find operations in $O(m + p + q \alpha_q(p))$ time, while using $O(m)$ space [12], where $p$ and $q$ are the numbers of Union and Find operations, respectively, and $\alpha_k$ is the inverse of the $k$-th row of Ackermann function.

### 2.3 Bidirectional phrases and partial bidirectional phrases

**Bidirectional phrases** (BP) [11] of string $T$ is a partition of $T$ as substrings (phrases) $B = f_1, f_2, \ldots, f_k$ such that each $f_i = T[s_i..s_i + \ell - 1]$ is (i) either copied from another substring $T[t_i..t_i + \ell - 1]$ (target phrase) with $s_i \neq t_i$, which can overlap $T[s_i..s_i + \ell - 1]$, or (ii) an explicit character (character phrase), i.e., $f_i = T[s_i]$. Target phrase $f_i$ is denoted as a pair $(t_i, [f_i])$ of the reference position $t_i$ and the length $|f_i|$ of $f_i$. The substring $T[t_i..t_i + |f_i| - 1]$ is called the reference string of $f_i$. 

2
The original string $T$ can be recovered from BP $B$ by referring to a finite number of phrases from each $f_i$ in $B$. If an infinite loop of phrases referred from any $f_i$ exists, the original string $T$ cannot be recovered from $B$. If $T$ can be recovered from $B$, $B$ is said to be a valid BP of $T$; otherwise, $B$ is said to be invalid BP of $T$.

The value of the phrase reached from position $x$ in $k$ iterations of references is formally defined as $g^k : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\} \cup \Sigma$. For $k = 0$, if $T(x)$ is a character phrase, $g^0(x) = T(x)$; otherwise $g^0(x) = t_p + (x - s_p)$, where $p$ is the integer such that $s_p \leq x < s_{p+1}$ holds for $s_{b+1} = n + 1$. For $k \geq 1$, we define $g^k$ as follows:

$$g^k(x) = \begin{cases} g^{k-1}(x) & \text{if } g^{k-1}(x) \in \Sigma, \\ g^0(g^{k-1}(x)) & \text{otherwise.} \end{cases}$$

(1)

If $g^n[x] \in \Sigma$ holds, then there are no infinite loops of references containing $x$. Therefore, $B$ is valid BP of $T$ if $B$ has no infinite loops of references, i.e., $g^n[x] \in \Sigma$ holds for all $x$.

For example, let $B = (3, 2), a, b, \langle 2, 3 \rangle$ and $B' = (3, 2), \{1, 2\}, b, a, b$ be BPs of $T = ababbab$. Then $B$ is valid since $g^4(1), \ldots, g^7(7) = a, b, a, b, 4, a, b$ and $g^7(5) = b$. On the other hand, $B'$ is invalid since $g^2(1), \ldots, g^7(7) = 1, 2, 3, 4, b, a, b$.

LZ77 phrases are a specialization of BP and defined as the bidirectional phrases that are all selected from previously seen substrings. Since there is no infinite loops of references on phrases, LZ77 phrases of $T$ are always valid BP of $T$. Formally, let $LZ(T) = f_1, f_2, \ldots, f_z$ of $T$ be valid BP of $T$ such that $|f_i| = \max\{1, \lfloor s_i \rfloor\}$ for each $i \in \{1, \ldots, z\}$.

LZRR parsing gradually builds the valid BP from the start position of $T$ in the left-to-right order. A subsequence of the valid BP is called partial bidirectional phrases (PBP) and is defined as a BP $P = f_1, f_2, \ldots, f_k$ for a prefix of $T$ that can be copied from any substring of $T$, i.e., $l_i \in \{1, \ldots, n\} \setminus \{s_i\}$ for all $i \in \{1, \ldots, k\}$ for a target phrase $f_i$, which avoids a self copy.

The concatenation of such PBP $P$ and every character phrase referred from $P$ can recover the prefix of $T$ with a finite number of references. Such PBP are called valid PBP, and other PBP are called invalid PBP. Formally, let $B_P = f_1, f_2, \ldots, f_k$ be the concatenation of PBP $P$ and the remaining character phrases $f_1, f_2, \ldots, f_k$ equivalent to suffix $T[(n-k+1)\ldots]$. $P$ is valid if $B_P$ is valid; otherwise $P$ is invalid. For example, let $P = (3, 2), (6, 2)$ be a PBP of $T = ababbab$. Then $B_P = (3, 2), (6, 2), b, a, b$.

The original string of a PBP can be recovered by iteratively referring to phrases starting from each target phrase in a finite number of times until the character phrase is found. Thus, the position of each character phrase can be seen as the source for positions of target/character phrases. Formally, for a PBP $P$ and position $x \in \{1, \ldots, n\}$ on $T$, source$(P, x)$ returns source $y \in \mathcal{N}$ of $x$ in $B_P$, i.e., position $y$ satisfying either (i) $g^k(x) = y$ and $g^{k+1}(x) \in \Sigma$ for an integer $k$ or (ii) $x = y$ and $g^0(x) \in \Sigma$. For the above example, the source of the position $6$ is the position $6$ in $P$ since $g^0(1) = 3, g^1(1) = 6$ and $g^2(1) = a$.

## 3 LZRR

A key idea of LZRR parsing is to compute the valid BP from an input text $T$ by gradually computing the valid PBP from the head of $T$ in the left-to-right order. LZRR parsing computes whole LZRR phrases initialized as zero phrase for an input string in two steps: (i) it computes candidates of the reference positions of the longest valid phrase following the current LZRR phrase; and (ii) it computes the valid (possibly character) phrase with the maximum length among extensions starting from those candidates. Steps (i) and (ii) are iterated until whole LZRR phrases are computed.

LZRR parsing uses two major functions of LP and LF for steps (i) and (ii), respectively. Given a valid PBP $P$ of $T$, LP function $LP(P)$ returns the longest valid phrase following $P$, i.e., the longest phrase $f$ such that $P \cdot f$ is a valid PBP of $T$. Given a valid PBP $P$ of $T$ and reference position $j \in \{1, \ldots, n\}$, LF function $LF(P, j)$ returns the length of the longest valid phrase having reference position $j$ and following $P$, i.e., $LF(P, j) = \max\{0 \mid \ell \in \{1, 2, \ldots, \text{lp}(i, j)\} : P \cdot (j, \ell) \text{ is valid}\}$ where $i$ is the starting position of the phrase following $P$. LZRR parsing computes LZRR phrases as the valid BP $LZRR(T) = LP(P_0), \ldots, LP(P_{b \cdot 1})$ of $T$ where $P_0$ is the first $p$ LZRR phrases for each $p \in \{0, 1, \ldots, b\}$ and $b$ is the number of LZRR phrases of $T$. The LZRR phrases of $T$ are not unique.

For example, let $P_1 = (3, 5)$ be the first LZRR phrase of $T = abababaababa$. $LF(P_1, 1), \ldots, LF(P_1, 12) = 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0$. LZRR parsing chooses phrase (9, 2) or (11, 2) as the next one.

This paper shows the following two theorems.
Algorithm 1: The LP algorithm.

Input: $P = f_1, \ldots, f_p$, Output: LP($P$);

$(k, j) \leftarrow (1, SA[i][1])$;

$(k, j_{\text{max}}, \ell_{\text{max}}) \leftarrow (1, \text{NIL}, 0)$;

while true do

\[ j \leftarrow SA[i][k++] \] // $i = |f_1 \cdots f_p| + 1$.

if lcp($i, j$) $\leq \ell_{\text{max}}$ then break;

$\ell \leftarrow LF(P, j)$;

if $\ell > \ell_{\text{max}}$ then $(j_{\text{max}}, \ell_{\text{max}}) \leftarrow (j, \ell)$;

if $\ell_{\text{max}} > 0$ then return $(j_{\text{max}}, \ell_{\text{max}})$ else return $T[i]$;

Theorem 1. For a given string $T$, LZRR parsing computes LZRR($T$) in $O(n^2 \alpha_n(n^2))$ time using $O(n)$ working space.

Theorem 2. $|\text{LZRR}(T)| \leq |\text{LZ}(T^R)|$ holds.

The LZRR parsing algorithm is presented in Section 3. Theorems 1 and 2 are shown in Section 4.

3.1 LP algorithm

A straightforward computation of LP($P$) is to compute reference position $j_{\text{max}}$ such that $LF(P, j_{\text{max}}) = \text{max}\{LF(P, 1), \ldots, LF(P, n)\}$ and then compute $\ell_{\text{max}} = LF(P, j_{\text{max}})$, which results in LZRR phrase $(j_{\text{max}}, \ell_{\text{max}})$. This method takes $\Omega(n)$ time even if $LF(P, j)$ can be computed in constant time for each position $j$. Instead, we reduce the computation time of $LF$ functions by leveraging the following fact: the length of the longest valid phrase of starting position $i$ and reference position $j$ is not larger than that of the LCP of $T[i..]$ and $T[j..]$. This fact suggests that after we find a phrase of length $\ell'$, we do not need to compute LF functions for any reference position $j$ such that the LCP of $T[i..]$ and $T[j..]$ is not longer than $\ell'$. For an efficient computation, we sort reference positions in descending order with respect to the length of the LCP for $T[i..]$ and maintain those positions in the sorted suffix array $SA_i$ of $i$. Then, we omit computing LF functions of reference positions on $SA_i[k_{\text{max}} + 1..]$ for the left-most position $k_{\text{max}}$ on $SA_i$ such that the longest valid phrase starting at a reference position in $SA_i[1..k_{\text{max}}]$ is at least as long as the LCP of $T[i..]$ and $T[SA_i[k_{\text{max}}]..]$. This is because $j_{\text{max}}$ exists on $SA_i[1..k_{\text{max}}]$. Thus, the following lemma holds.

Lemma 3. Let $\ell_k = \text{max}\{LF(P, SA_i[1]), \ldots, LF(P, SA_i[k])\}$ and $k_{\text{max}}$ be the left-most position on the $SA_i$ such that $\ell_{k_{\text{max}}} \geq \text{lcp}(i, SA_i[k_{\text{max}}])$ holds. Then $\ell_{\text{max}} = \ell_{k_{\text{max}}}$ holds and $SA_i[..k_{\text{max}}]$ contains $j_{\text{max}}$.

Proof. See Appendix.

Algorithm 3 shows the algorithm for computing LP($P$) function and computes each LF function from the head of $SA_i$. When Algorithm 3 finds $k_{\text{max}}$, it returns the current longest valid phrase.

3.2 LF algorithm

LF algorithm LF($P, j$) finds the longest valid target phrase with reference position $j$ and following the PBP $P$ of $T$ by gradually extending the target phrase of length 1 until it cannot find any reference string copying the target phrase. When PBP $P \cdot (j, \ell)$ for $P$ and the target phrase $(j, \ell)$ is computed one-by-one, it can include an infinite loop of references by a mutual reference of phrases. This is because PBP as a target phrase can be copied from the left and right reference strings. This can happen when for computing the extension $P \cdot (j, \ell)$ the position of a target phrase in $P$ and $(j, \ell)$ can be mutually reached with a finite number of references. The LP algorithm avoids such cases by using the union-find data structure built from PBP $P$.

Each disjoint set in the union-find data structure includes string positions with the same source (character phrase) for PBP $P$. The union-find data structure is initialized as $n$ disjoint sets that all contain the unique position of the input string of length $n$. If the union-find data structure for PBP $P \cdot (j, \ell - 1)$ for $P$ and the target phrase $(j, \ell)$ exists, the data structure for $P \cdot (j, \ell - 1)$ can be updated by $\text{Union}(i + \ell - 1, j + \ell - 1)$ operation.

The infinite loops of references can be detected using the find operation in the union-find data structure. When $P \cdot (j, \ell - 1)$ is a valid and the extension of starting position $i + \ell - 1$ next to the PBP
and reference position $j + \ell - 1$ is computed, if $\text{Find}(i + \ell - 1)$ is equal to $\text{Find}(j + \ell - 1)$ if and only if infinite loops of references exist. LF algorithm checks this condition each time. Formally, the following corollary holds.

**Corollary 4.** Let $Q = P \cdot (j, \ell)$ be a valid PBP and $Q' = P \cdot (j, \ell + 1)$ be a PBP for an integer $\ell \in \{0, \ldots, \lcp(i, j) - 1\}$, and $\mathcal{D}_P$ be disjoint sets on $\{1, \ldots, n\}$ such that each set consists of all positions of the same source for a PBP $P$, where $P \cdot (j, 0)$ is $P$ and $i$ is the starting position of the last target phrase (i.e., $(j, \ell)$) in $Q$. (1) If $\text{Find}(i + \ell) \neq \text{Find}(j + \ell)$ holds on $\mathcal{D}_Q$, then $Q'$ is valid. Otherwise $Q'$ is invalid. (2) $\mathcal{D}_Q'$ is equal to the set created by $\text{Union}(i + \ell, j + \ell)$ on $\mathcal{D}_Q$.

Algorithm 2 shows the algorithm for computing $LF(P, j)$ function using Corollary 4, and the algorithm stated previously. Thus, we can compute the length $\ell_{\text{max}}$ of the longest valid target phrase with reference position $j$ and following the PBP $P$ by $O(\ell_{\text{max}})$ union and find operations on the given union-find data structure for $\mathcal{D}_P$.

Note that we need to modify Algorithm 2 for LP($P$) algorithm. This is because LF algorithms in our LP($P$) algorithm need the same union-find data structure determined by the PBP $P$. On the other hand, the given union-find data structure is changed by union operations in Algorithm 2. By modifying Algorithm 2 using an additional union-find data structure, we can compute $LF(P, j)$ without updating the given union-find data structure. Formally, the following lemma holds.

**Lemma 5.** Given the union-found data structure $L$ for $\mathcal{D}_P$, we can compute $LF(P, j)$ in $O(n)$ working space by $O(\ell_{\text{max}})$ Find operations on $L$ and $O(\ell_{\text{max}})$ union and find operations on an additional union-find data structure $L'$ for $O(\ell_{\text{max}})$ disjoint sets. $L'$ is disposed after $LF(P, j)$ is computed.

**Proof.** See Appendix.

### 3.3 Computation of LZRR($T$)

Since $LF(P, j)$ algorithm for each $p \in \{0, \ldots, b - 1\}$ uses the union-find data structure for disjoint sets of the current LZRR phrases (i.e., $P_p$), we update the union-find data structure when the $(p + 1)$-th LZRR phrase is selected. This needs at most $|LP(P_{p+1})|$ Union operations by Corollary 4.

### 4 Theoretical analysis

#### 4.1 The proof of Theorem 1

We show that the working space of LZRR parsing is $O(n)$ space. LZRR parsing needs two data structures: (1) the union-find data structures for LF algorithm and (2) the data structure to compute the sequence $W_p = SA_s[1], \lcp(s_p, SA_s[1]), \ldots, SA_k[k_p], \lcp(i, SA_k[k_p])$ for LP($P_{p-1}$) algorithm, where $k_p$ is $k_{\text{max}}$ in LP($P_{p-1}$) algorithm and $s_p$ is the starting position of $p$-th LZRR phrase.

We can compute $W_p[1..k]$ in $O(k)$ time in an online manner using arrays of SA, ISA, and LCP for two integers $p$ and $k$ (See Appendix).

SA, ISA, and LCP of a given a string $T$ can be constructed in $O(n)$ time and working space [4, 5]. Therefore, the second data structure can be constructed in $O(n)$ time and space, and the LZRR parsing algorithm runs in $O(n)$ working space.

Next, we show that the running time of LZRR parsing is $O(n^2 \alpha_n(n^2))$. Let $G$ be the sequence of operations on disjoint-sets executed by LZRR parsing and $W$ be the sequence of $W_1 \cdots W_b$, where $b$ is the number of phrases in LZRR($T$). Then the running time is the sum of the computation time for executing $G$ and computing $W$, and the prepossessing time of SA, ISA and LCP, which is $O(n)$.
We show that \( W \) can be computed in \( O(n^2) \) time. For an integer \( p \in \{1, \ldots, b\} \), \( W_p \) can be computed in \( O(k_p) = O(n) \) time. This is because \( k_p = |\text{Occ}(T,f_p)| \leq n \) holds since \( T[S\alpha_i[y]..] \) has \( f_p \) as a prefix for all \( y \in \{1, \ldots, k_p\} \), where \( f_p \) is the string represented by the \( p \)-th LZRR phrase. Thus, \( |W| = O(n^2) \) since \( b \leq n \). Hence \( W \) can be computed in \( O(n^2) \) time using the above online algorithm.

We show that \( G \) is performed in \( O(n^2 \alpha_n(n^2)) \) time. \( |G| = O(\sum_{p=1}^{b}(|f_p| \times k_p)) \) holds because \( \text{LP}(P_p) \) performs \( O(k_{p+1} \times |f_{p+1}|) \) union and find operations for \( p \in \{0, \ldots, b - 1\} \). Since \( |f_1| + \cdots + |f_n| = n \) and \( |\text{Occ}(T,f_p)| \leq n \) for all \( p \), \( |G| = O(n^2) \) holds. Therefore, \( G \) is performed in \( O(n^2 \alpha_n(n^2)) \) time by union-find data structures.

As a result, we can compute LZRR \((T)\) in \( O(n^2 \alpha_n(n^2)) \) time and \( O(n) \) working space.

### 4.2 The proof of Theorem 2

We define two BPs \( L^Z(T) \) and LZOR \((T) \) for Theorem 2 and show three formulas: (1) \( |L^Z(T)| = |LZ(T)| \), (2) \( |\text{LZRR}(T)| \leq |\text{LZOR}(T)| \), and (3) \( |\text{LZOR}(T)| = |LZ(T^R)| \). Theorem 2 clearly holds in (1), (2), and (3), i.e., LZRR \((T) \) \( \leq \) LZ \((T^R) \). The detailed proofs are in Appendix.

The proof of \( |L^Z(T)| = |LZ(T)| \). \( L^Z(T) = f_1, \ldots, f_k \) parses greedily \( T \) in the right-to-left order such that each phrase is the longest substring occurring previously (left) in \( T \). A key idea of this proof is that if \( L^Z(T) \) chooses a substring as an \( L^Z \) phrase, then there exists an LZ phrase starting at a position on the \( L^Z \) phrase and including the ending position of the \( L^Z \) phrase. This is because the \( L^Z \) phrase occurs previously in \( T \) and the LZ phrase is the longest substring occurring previously in \( T \). Since the fact holds for every \( L^Z \) phrase, \( |LZ(T)| \leq |L^Z(T)| \) holds. Conversely, if \( L^Z(T) \) chooses a substring as an LZ phrase, then there exists an \( L^Z \) phrase starting at a position on the LZ phrase and including the starting position of the LZ phrase. This is because the LZ phrase occurs previously in \( T \) and the \( L^Z \) phrase is the longest substring occurring previously in \( T \). Since this fact holds for every LZ phrase, \( |L^Z(T)| \leq |LZ(T)| \) holds. Therefore, \( |L^Z(T)| = |LZ(T)| \) holds.

The proof of \( |\text{LZRR}(T)| \leq |\text{LZOR}(T)| \). LZOR \((T) = f_1, \ldots, f_k \) parses \( T \) in the left-to-right order such that each phrase is the longest substring occurring subsequently in \( T \). A key idea of this proof is that if LZOR \((T) \) can choose a substring at a position as an LZOR phrase, then LZRR \((T) \) also can choose the substring as an LZRR phrase. This is because candidate phrases with right reference positions are always valid phrases in LZRR parsing. Since the fact holds for every position on \( T \), \( |\text{LZRR}(T)| \leq |\text{LZOR}(T)| \) holds.

The proof of \( |\text{LZOR}(T)| = |LZ(T^R)| \). Parsing a string in the left-to-right order using the longest substring occurring subsequently in the string is equal to parsing the reversed string in the right-to-left order using the longest substring occurring previously in the reversed string. Thus, \( |\text{LZOR}(T)| = |LZ(T^R)| \) holds.

### 5 Experiments

In this section, we demonstrate the effectiveness of LZRR parsing with benchmark strings. We used two types of strings of pseudo-real and real repetitive collections in the Pizza & Chili corpus downloadable from [https://github.com/TNishimoto/lzrr](https://github.com/TNishimoto/lzrr). We compared our LZRR parsing with LZ77 and lex-parse. We used execution time, memory, and number of phrases as evaluation measures for each method. The C++ programming language was used for implementing all the parsing algorithms. The implementations used in this experiment are available at [https://github.com/TNishimoto/lzrr](https://github.com/TNishimoto/lzrr) and lex-parse were implemented in the standard manner and work in time and space linear to string length using SA, ISA, and LCP arrays. For each method, we computed two sets of phrases for original string \( T \) and reverse string \( T^R \), respectively, and we took the set with the smaller number of phrases. We denote numbers of phrases as \( |LZ77| \), \( |LEX| \), and \( |LZRR| \) for parsing algorithms of LZ77, lex-parse (LEX), and LZRR, respectively. We performed all the experiments on one core of a quad-core Intel(R) Xeon(R) E5-2680 v2 (2.80 GHz) CPU with 256 GB of memory.

### 5.1 Results

Table 1 shows the number of phrases for each method. The number of LZRR phrases was smaller than that of LZ77 phrases for all benchmark strings. Specifically, the number of LZRR phrases was approximately five percent smaller than that of LZ77 for all the strings except for fib41, rs.13, and tm29. The number of LZRR phrases was smaller that of lex-parse phrases for most of the strings.
Table 1: The number of phrases for each method. The smallest number of phrases for each string is depicted in bold.

| String             | String length | [LZ77] | [LEX] | [LZRR] | [LZ77] | [LEX] | [LZRR] |
|--------------------|---------------|--------|-------|--------|--------|-------|--------|
| db4ri              | 207,914,296   | 22     | 4     | 3      | 0.227  |
| rs.13              | 216,747,218   | 52     | 40    | 51     | 0.981  |
| tm29               | 268,435,456   | 76     | 58    | 43     | 0.554  |
| fib41              | 267,914,296   | 22     | 4     | 5      | 0.227  |
| rs.13              | 216,747,218   | 52     | 40    | 51     | 0.981  |
| tm29               | 268,435,456   | 76     | 58    | 43     | 0.554  |
| dblp.xml.0001.1    | 104,557,600   | 52     | 40    | 51     | 0.981  |
| dblp.xml.0001.2    | 104,557,600   | 52     | 40    | 51     | 0.981  |
| sources.001.1      | 104,557,600   | 52     | 40    | 51     | 0.981  |
| dblp.xml.0001.1    | 104,557,600   | 52     | 40    | 51     | 0.981  |
| dblp.xml.0001.2    | 104,557,600   | 52     | 40    | 51     | 0.981  |
| sources.001.2      | 104,557,600   | 52     | 40    | 51     | 0.981  |
| dna.001.1          | 204,704,296   | 22     | 4     | 5      | 0.227  |
| proteins.001.1     | 204,704,296   | 22     | 4     | 5      | 0.227  |
| english.001.2      | 204,704,296   | 22     | 4     | 5      | 0.227  |
| english.001.2      | 204,704,296   | 22     | 4     | 5      | 0.227  |
| eisenstein.de.txt  | 467,626,544   | 130    | 85    | 147    | 11,418 | 11,418| 19,196 |
| eisenstein.en.txt  | 467,626,544   | 130    | 85    | 147    | 11,418 | 11,418| 19,196 |
| worldleaders       | 46,968,181    | 8      | 5     | 16     | 1,148  | 1,148| 1,939  |
| influenza           | 154,808,555   | 42     | 27    | 51     | 3,781  | 3,781| 6,351  |
| kernel              | 257,961,616   | 89     | 58    | 43     | 6,299  | 6,299| 10,602 |
| cere                | 461,286,644   | 131    | 90    | 500    | 11,264 | 11,264| 18,925 |
| coreutils           | 429,265,758   | 125    | 85    | 203    | 10,481 | 10,481| 17,609 |
| Escherichia_Coli    | 429,265,758   | 125    | 85    | 203    | 10,481 | 10,481| 17,609 |

Table 2: The execution time and memory for each method.

| String             | String length | Execution time [sec] | Memory consumption [MB] |
|--------------------|---------------|----------------------|-------------------------|
|                    |               | [LZ77] | [LEX] | [LZRR] | [LZ77] | [LEX] | [LZRR] |
| eisenstein.de.txt  | 467,626,544   | 24     | 16    | 27     | 2,266  | 2,266| 3,808  |
| eisenstein.en.txt  | 467,626,544   | 130    | 85    | 147    | 11,418 | 11,418| 19,196 |
| worldleaders       | 46,968,181    | 8      | 5     | 16     | 1,148  | 1,148| 1,939  |
| influenza           | 154,808,555   | 42     | 27    | 51     | 3,781  | 3,781| 6,351  |
| kernel              | 257,961,616   | 89     | 58    | 43     | 6,299  | 6,299| 10,602 |
| cere                | 461,286,644   | 131    | 90    | 500    | 11,264 | 11,264| 18,925 |
| coreutils           | 429,265,758   | 125    | 85    | 203    | 10,481 | 10,481| 17,609 |

Table 2 shows execution time and memory on limited benchmark strings for each method. The table for all the strings is presented in Appendix. Although our LZRR parsing needs $O(n^2 \alpha(n^2))$ time, the execution time was at most four times slower than that of LZ77 parsing. This is because the number of while-loops in Algorithm 1 is much smaller than $n$ in practice. The memory for LZRR parsing was at most two times larger than that for LZ77 parsing. This is because the proposed algorithm needs the data structure for LF along with SA, ISA, and LCP arrays.

6 Conclusions
We presented a new bidirectional parsing algorithm named Lempel-Zip 77 parsing with right reference (LZRR). The number of LZRR phrases is theoretically guaranteed to be smaller than that of LZ77. Experimental results using benchmark strings showed LZRR parsing works in practice. An interesting line of future work is to devise the LZRR parsing algorithm working in $o(n^2 \alpha(n^2))$ time or a compressed space.

Acknowledgments. We would like to thank Simon J. Puglisi for notifying us some related work [1, 2].

References
[1] Dinklage, P., Fischer, J., Köppl, D., Löbel, M., Sadakane, K.: Compression with the tudocomp framework. In: Proceedings of SEA. pp. 13:1–13:22 (2017)
[2] Gagie, T., Navarro, G., Prezza, N.: On the approximation ratio of lempel-ziv parsing. In: Proceedings of LATIN. pp. 490–503 (2018)
[3] Jez, A.: A really simple approximation of smallest grammar. Theor. Comput. Sci. 616, 141–150 (2016)
[4] Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proceedings of ICALP. pp. 943–955 (2003)

[5] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proceedings of CPM. pp. 181–192 (2001)

[6] Kempa, D., Prezza, N.: At the roots of dictionary compression: string attractors. In: Proceedings of STOC. pp. 827–840 (2018)

[7] Kreft, S., Navarro, G.: LZ77-like compression with fast random access. In: Proceedings of DCC. pp. 239–248 (2010)

[8] Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on information theory 22(1), 75–81 (1976)

[9] Navarro, G., Prezza, N.: On the approximation ratio of greedy parsings. CoRR abs/1803.09517 (2018)

[10] Rytter, W.: Application of lempel-ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)

[11] Storer, J.A., Szymanski, T.G.: Data compression via textural substitution. Journal of the ACM 29(4), 928–951 (1982)

[12] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22(2), 215–225 (1975)
Appendix A: The proof of Lemma 5

Algorithm 3: Modified LF algorithm.

Input: $P$, $L$, $j$, and $W$;
Output: $LF(P,j)$;
Create the union-find data structure for $D' = \emptyset$;
$\ell \leftarrow 1$;
while $T[i + \ell - 1] = T[j + \ell - 1]$ and $source(i + \ell - 1) = source(j + \ell - 1)$ do
  $mUnion(i + \ell - 1, j + \ell - 1)$;
  $\ell \leftarrow \ell + 1$;
Delete the union-find data structure for $D'$;
Initialize $W$;
return $\ell$;

Function $source(x)$
$y \leftarrow Find(x)$ on $D_P$;
if $i \leq y \leq i + \ell - 1$ then
  $z \leftarrow Find(W[y])$ on $D'$;
  return $z$;
else
  return $y$;

Function $mUnion(x, y)$
$x' \leftarrow source(x)$;
$MakeSet'(y)$;
if $W[x'] \neq -1$ then
  $Union(W[x'], W[y])$ on $D'$;
else
  $MakeSet'(x')$;
  $Union(x', y)$ on $D'$;

Function $MakeSet'(x)$
$p \leftarrow MakeSet$ on $D'$;
$W[x] \leftarrow p$;

Figure 1: This figure represents the change of sources when a new target phrase is appended into a PBP. The “t” and “c” in rectangles mean target phrases and character phrases, respectively.

To compute $LF(P,j)$ without changing the union-find data structure $L$ for $D_P$, we create an additional union-find data structure $L'$ and we emulate find operations on $D_{P,(j,\ell)}$ using union-find data structures $L$ and $L'$ for $D_P$. Since union operations are performed on $L'$, $L$ is not changed in LF algorithm.

A key idea is that sources on the last phrase $(j,\ell)$ are only changed by extending $P \cdot (j,\ell)$. See Figure 1. The left figure represents sources of positions on target phrases. The right figure represents the change of sources by appending new target phrase $(j,\ell)$ to the target phrases. The new target phrase changes only sources on the phrase and these sources are determined by the phrase. This suggests that sources not on the target phrase $(j,\ell)$ can be computed using $L$, and the other sources can be computed using $L$ and the additional union-find data structure $L'$ that manages sources of positions on the phrase $(j,\ell)$. In addition, disjoint sets managed by $L'$ can be updated by union operations as Corollary 4.
Formally, let $U(P \cdot \langle j, \ell \rangle)$ be the set of positions on the phrase $\langle j, \ell \rangle$ and sources of those positions (i.e., $U(P \cdot \langle j, \ell \rangle) = \{i, \ldots, i + \ell - 1\}$) and let $D'_{P \cdot \langle j, \ell \rangle}$ be disjoint sets on $U(P \cdot \langle j, \ell \rangle)$ such that each set consists of all positions of the same source, where $i$ is the position following $P$. Let MakeSet$'$($x$) be the operation on disjoint-sets $D$ that adds $\{x\}$ into $D$ if $D$ does not contain $x$. Then the following lemma and corollary hold.

**Lemma 6.** For a position $x \in \{1, 2, \ldots, n\}$, if $source(P, x) \not\in \{i, i + 1, \ldots, i + \ell - 1\}$ holds, then $source(P, x) = source(P \cdot \langle j, \ell \rangle, x)$ holds. Otherwise, $source(P \cdot \langle j, \ell \rangle, x') = source(P \cdot \langle j, \ell \rangle, x)$ holds, where $x' = source(P, x)$.

**Proof.** $T[r]$ is a character phrase on $B_P$ for each position $r \in \{i, i + 1, \ldots, i + \ell - 1\}$. If the source $x'$ of $x$ on $B_P$ for each position $r \in \{i, i + 1, \ldots, i + \ell - 1\}$, then $x$ does not reach any position in $\{i, \ldots, i + \ell - 1\}$. When a phrase is appended into $P$, the source $x'$ is changed if and only if the character phrase on $x'$ is changed. Thus, the source of $x$ is not changed by appending $(j, \ell)$ into $P$, i.e., $source(P, x) = source(P \cdot \langle j, \ell \rangle, x)$.

Otherwise, the source $x'$ is in $\{i, \ldots, i + \ell - 1\}$ on $B_P$ and $x'$ has a source $x''$ on $B_{P \cdot \langle j, \ell \rangle}$ because $T[x']$ is not a character phrase on $B_{P \cdot \langle j, \ell \rangle}$. Since the source of $x'$ is that of $x$ on $B_{P \cdot \langle j, \ell \rangle}$, Lemma 6 holds.

**Corollary 7.** (1) For an integer $x \in \{i, \ldots, i + \ell - 1\}$, there exists a set $X \subseteq D'_{P \cdot \langle j, \ell \rangle}$ that contains two positions $x$ and $source(P \cdot \langle j, \ell \rangle, x)$. (2) $D'_{P \cdot \langle j, \ell \rangle}$ can be created by performing $O(1)$ Union and MakeSet$'$ operations on $D'_{P \cdot \langle j, \ell \rangle}$.

We compute the source of a given position on $\{1, \ldots, n\}$ by $O(1)$ find queries on $D_P$ and $D'_{P \cdot \langle j, \ell \rangle}$ using Lemma 6 and Corollary 7. Note that we need to compute the position on the character phrase in a given set to obtain the source of a given position. For this reason, we use the position on a character phrase as the id of the set that contains the position. We can maintain such id using an additional array of length $m$ with the same time complexity, where $m$ is the cardinality of disjoint-sets.

We also note that we need to convert integers in $D'_{P \cdot \langle j, \ell \rangle}$ to consecutive integers. This is because disjoint sets of $L'$ are on consecutive integers since MakeSet$'$ creates the element $\{m + 1\}$. Thus, we use an array $W$ of size $n$, where $W[x]$ stores the integer in $L'$ that corresponds to $x$ if $x \in U(P \cdot \langle j, \ell \rangle)$; otherwise $W[x] = -1$. This array also enables us to emulate MakeSet$'$ operations. Since the size of $W$ is $n$, we reuse $W$ during the LZRR parsing algorithm, and the algorithm creates the array in advance. It takes $O(n)$ time and space. $W$ can be initialized in $O(m)$ time, where $m$ is the number of positive integers in $W$.

Algorithm 3 shows the modified algorithm for computing LF($P, j$) function using Lemma 6 and Corollary 7. Algorithm 3 computes LF($P, j$) by $O(l_{max})$ union and find operations and does not perform union operations on $D_P$, where $l_{max}$ is the length of the longest valid phrase following $P$ with reference position $j$. As a result, Lemma 5 holds.

Note that Algorithms 2 and 3 can fail if there exists an invalid PBP $P \cdot \langle j, \ell' \rangle$ for an integer $\ell' \in \{1, 2, \ldots, l_{max}\}$. If such an integer exists, then algorithms return $\ell' - 1$ and fail. However, such cases do not occur because we cannot remove infinite loops of references from an invalid PBP by appending phrases into the PBP.

**Appendix B: Computing $SA_k[1..\ell]$ and lcp($k, SA_k[1]$), \ldots, lcp($k, SA_k[\ell]$)**

We show that we can compute $SA_k[1..\ell]$ and lcp($k, SA_k[1]$), \ldots, lcp($k, SA_k[\ell]$) for a given $k$ and $\ell$ in $O(\ell)$ time using $T$ and $SA$, $ISA$, $LCP$ arrays.

We use the known fact that lcp($SA[i], SA[j]$) = min{lcp($[i+1], \ldots, lcp([j])$) holds for two integers $1 \leq i < j \leq n$. When $SA_k[1..\ell']$ stores the permutation of $SA[i', j']$ containing $k$ for some integer $\ell'$, $SA_k[1..\ell' + 1]$ can store $SA[i' - 1]$ or $SA[j' + 1]$ by the above fact, where $i'$ and $j'$ are integers such that $j' - i' + 1 = \ell$. Then $SA_k[1..\ell' + 1]$ is also the permutation of a subarray of $SA$ containing $k$. Thus, we compute $SA_k[1..\ell']$ by using the above observation.

We compute $SA_k[\ell' + 1]$ using $i' , j' , p$ and $q$, where $p = lcp(k, SA[i'])$ and $q = lcp(k, SA[j'])$. Since lcp($k, SA[j' - 1]$) = min{lcp($[i'], [j]$), $SA_k[i' + 1] = \min\{lcp[j'+1], q\}$, $SA_k[\ell' + 1]$ can be computed in constant time. In addition, we can appropriately update the four parameters in constant time for $SA_k[\ell' + 2]$. Therefore, we can compute $SA_k[1..\ell]$ and lcp($k, SA_k[1]$), \ldots, lcp($k, SA_k[\ell]$) in $O(\ell)$ time and constant working space using a simple algorithm.
Appendix C: The proof of the upper bound of LZRR phrases

We show three formulas using injective functions: for two BPs $F = f_1, f_2, \ldots, f_k$ and $F' = f'_1, f'_2, \ldots, f'_k$ of $T$, if there exists an injective function $w$ that maps phrases in $F$ into distinct phrases in $F'$, then $k \leq k'$ holds. In the remaining section, let $s_x$ and $e_x'$ (resp. $s_x'$ and $e_x''$) be starting and ending positions of $x$-th phrase in $F$ (resp. $F'$).

The proof of $|LZ'(T)| = |LZ(T)|$. $LZ'(T) = f_1, f_2, \ldots, f_k$ parses greedily $T$ in the right-to-left order such that each phrase is the longest substring occurring previously (left) in $T$. Formally, let $LPF'$ be the integer array of length $n$ such that $LPF'[i]$ stores the length of the longest substring of $T$ ending at position $i$ and occurring on $T[1..i] = \{1, 2, \ldots, n\}$, i.e., $LPF'[i] = \max(0) \cup \{\ell \mid \ell \in [1..i], \text{Occ}(T[1..i], T[i+1..i+\ell]) > 0\}$. Then $LZ'(T) = f_1, f_2, \ldots, f_k$ is the valid BP of $T$ such that for all $i \in \{1, 2, \ldots, k\}$, the starting position $s_x$ of $f_x$ is $s_x = s_{x+1} - \max\{1, LPF'[s_{x+1} - 1]\}$, where $s_{k+1} = n$. Figure 2 illustrates examples of $LZ(T)$ and $LZ'(T)$.

For $LZ(T) = f_1, f_2, \ldots, f_k$ and $LZ'(T) = f'_1, f'_2, \ldots, f'_k$, we define the function $w(x')$ that returns the integer $x$ such that $f_x$ contains $e_x'$ (i.e., $s_x \leq e_x' \leq e_x$ holds). $w$ is injective if the starting position of each $LZ'$ phrase is not larger than that of the LZ phrase containing the ending position of the $LZ'$ phrase, i.e., $s_{x'} \leq w(x')$ holds for all $x' \in \{1, 2, \ldots, k' - 1\}$. This is because no $LZ$ phrases contain two ending positions in $LZ'$ phrases, i.e., no integer exists $y'$ such that $w(y') = (w(x') + 1)$ holds if $s_{x'} < w(x')$ holds for all $x'$.

We show $s_{x'} \leq w(x')$ using the substring $S$ starting at the starting position of $f_{w(x')}$ and ending at the ending position of $f_{x'}$, i.e., $S = T[s_{w(x')}..e_{w(x')}']$. When $|f_{x'}| \geq |S|$ holds, $s_{x'} \leq w(x')$ holds because $S$ is a suffix of $f_{x'}$ and $S$ is a prefix of $f_{w(x')}$. Thus, we show $|f_{x'}| \geq |S|$ always holds. If $S$ occurs in $T$ twice, then $|f_{x'}| \geq |S|$ because $LZ'$ chooses the longest substring ending at position $e_x'$ and occurring on $T[1..e_x' - 1]$. Otherwise, $|S| = 1$ and $S = f_{x'} = w(x')$ hold since $S$ is a new character, i.e., $\text{Occ}(T[1..e_x' - 1], T[e_x']) = 0$. Therefore, $s_{x'} \leq w(x')$ holds for all $x' \in \{1, 2, \ldots, k' - 1\}$.

Similarly, $|LZ'(T)| \geq |LZ(T)|$ holds by constructing the injective function that returns the integer $x'$ such that $f'_{x'}$ contains $s_x$ for a given $x$.

The proof of $|LZRR(T)| \leq |LZOR(T)|$. $LZOR(T) = f_1, f_2, \ldots, f_k$ parses $T$ in the left-to-right order such that each phrase is the longest substring occurring subsequently in $T$. Formally, let $LNF$ be the integer array of length $n$ such that $LNF[i]$ stores the length of the longest substring of $T$ starting at position $i$ and occurring on $T[i+1..n]$ for all $i \in \{1, 2, \ldots, n\}$, i.e., $LNF[i] = \max(0) \cup \{\ell \mid \ell \in [1..n-i+1], \text{Occ}(T[i+1..n], T[i+i+\ell-1]) > 0\}$. Then $LZOR(T) = f_1, f_2, \ldots, f_k$ is the BP of $T$ such that for all $x \in \{1, 2, \ldots, k\}$, the starting position $s_x$ of $f_x$ is $s_x = s_{x-1} + \max\{1, LNF[s_{x-1}]\}$ and $s_1 = 1$. Figure 2 illustrates an example of $LZOR(T)$.

For $LZRR(T) = f_1, f_2, \ldots, f_k$ and $LZOR(T) = f'_1, f'_2, \ldots, f'_k$, let $w(x)$ be the function that returns the integer $x'$ such that $f'_{x'}$ contains $s_x$. Then $w$ is injective if $w(x) < w(x+1)$ holds for all $x \in \{1, 2, \ldots, k-1\}$. We use the following lemma.

Lemma 8. Let $P = f_1, \ldots, f_k$ be a valid BP of $T$. Then $P \cdot (j, \ell)$ is also valid for any right target phrase $(j, \ell)$, i.e., $T[i..i+\ell-1] = T[j..j+\ell-1]$ and $j > i$ hold, where $i = [f_1 \cdots f_k] + 1$.

Proof. $T[i..n]$ are represented character phrases on $B_P$ since $T[i..n]$ has not been parsed. This means that $\text{source}(P, i+\ell-1) \neq \text{source}(P, j+\ell-1)$ for any $\ell' \in [1, 2, \ldots, \ell]$. Therefore $P \cdot (j, \ell)$ is valid by Corollary 4.

$w(x) < w(x+1)$ holds if $\max\{1, LNF[s_x]\} \leq |f_x|$ holds for all $x \in \{1, 2, \ldots, k\}$. Recall that $LP$ function returns the valid longest bidirectional phrase. The $LNF$ array and Lemma 8 suggest that the length of
the phrase of $\text{LZRR}(T)$ starting at position $i$ is at least $\max\{1, \text{LNF}[i]\}$. Thus $w(x) < w(x + 1)$ holds for all $x$, $w$ is injective, and hence $|\text{LZRR}(T)| \leq |\text{LZOR}(T)|$ holds.

The proof of $|\text{LZOR}(T)| = |\text{LZ}(T_R)|$. $|\text{LZOR}(T)| = |\text{LZ}(T_R)|$ holds clearly because $\text{LPF}'[x] = \text{LNF}_{T_R}[n - x + 1]$ holds for all $x \in \{1, 2, \ldots, n\}$, where $\text{LNF}_{T_R}$ is the LNF array of $T_R$.

Appendix D: The proof of Lemma 3

Proof. Recall that $\text{lcp}(i, \text{SA}_i[1]) \geq \cdots \geq \text{lcp}(i, \text{SA}_i[n])$ holds. On the other hand, $\text{LF}(P, j) \leq \text{lcp}(i, j)$ holds for all $j \in \{1, 2, \ldots, n\}$ because $\text{LF}(P, j)$ represents the length of the common prefix of $T[i..]$ and $T[j..]$. Therefore, $\ell_{\max} = \ell_n = k_{\max}$ holds, which means at least one position $j'$ exists such that $\text{LF}(P, j') = \ell_{\max}$ in $\text{SA}[..k_{\max}]$.

Appendix E: Experiments

Table 3: The full version of Table 2.

| String      | String length | Execution time [sec] | Memory consumption [MB] |
|-------------|---------------|----------------------|-------------------------|
| fib41       | 267,914,296   | 99                    | 99                      |
| rs.11       | 216,479,218   | 79                    | 79                      |
| tm29        | 268,435,456   | 108                   | 108                     |
| dbip.xml.0001.1 | 104,857,600 | 30                    | 30                      | 2,561 |
| dbip.xml.0001.2 | 104,857,600 | 30                    | 30                      | 2,561 |
| dbip.xml.0001.1 | 104,857,600 | 30                    | 30                      | 2,561 |
| dbip.xml.0001.2 | 104,857,600 | 30                    | 30                      | 2,561 |
| sources.001.1 | 104,857,600 | 30                    | 30                      | 2,561 |
| dna.001.1   | 104,857,600   | 30                    | 30                      | 2,561 |
| proteins.001.1 | 104,857,600 | 30                    | 30                      | 2,561 |
| english.001.2 | 104,857,600 | 30                    | 30                      | 2,561 |
| enstein.de.txt | 92,758,441  | 24                    | 24                      | 2,266 |
| enstein.en.txt | 467,626,544 | 130                   | 130                     | 13,416 |
| coreutils   | 205,281,778   | 56                    | 56                      | 5,013 |
| Escherichia_Coli | 112,689,515 | 32                    | 32                      | 2,752 |
| para        | 429,265,758   | 125                   | 125                     | 10,481 |
| influenza   | 154,808,555   | 42                    | 42                      | 3,781 |
| kernel      | 257,961,016   | 71                    | 71                      | 6,299 |
| cere        | 401,280,644   | 131                   | 131                     | 11,267 |
| corcutils   | 205,231,778   | 56                    | 56                      | 5,013 |
| Escherichia_Coli | 112,689,515 | 32                    | 32                      | 2,752 |
| para        | 429,265,758   | 125                   | 125                     | 10,481 |