UNIVERSITÀ
DEGLI STUDI
DI UDINE

Università degli studi di Udine

Computing LZ77 in Run-Compressed Space

Original

Availability:
This version is available http://hdl.handle.net/11390/1098488 since 2021-03-24T13:40:58Z

Publisher:

Published
DOI:10.1109/DCC.2016.30

Terms of use:
The institutional repository of the University of Udine (http://air.uniud.it) is provided by ARIC services. The aim is to enable open access to all the world.

Publisher copyright

(Article begins on next page)

01 September 2024

Computing LZ77 in Run-Compressed Space

Alberto Policriti[†*] and Nicola Prezza[†]

[†]University of Udine
Via delle scienze, 206
33100, Udine, Italy
alberto.policriti@uniud.it
prezza.nicola@spes.uniud.it

[*]Institute of Applied Genomics
Via J.Linussio, 51
33100, Udine, Italy

Abstract

In this paper, we show that the LZ77 factorization of a text $T \in \Sigma^n$ can be computed in $\mathcal{O}(\mathcal{R} \log n)$ bits of working space and $\mathcal{O}(n \log \mathcal{R})$ time, $\mathcal{R}$ being the number of runs in the Burrows-Wheeler transform of T (reversed). For (extremely) repetitive inputs, the working space can be as low as $\mathcal{O}(\log n)$ bits: *exponentially* smaller than the text itself. Hence, our result finds important applications in the construction of repetition-aware self-indexes and in the compression of repetitive text collections within small working space.

Introduction

Being able to estimate and exploit the self-repetitiveness of a text $T \in \Sigma^n$ is a task that stands at the basis of many efficient compression algorithms. This issue is particularly relevant in situations where the text to be processed is extremely large and repetitive (e.g. consider all versions of the articles belonging to the Wikipedia corpus or a large set of genomes belonging to the same species): in such cases, it is not always feasible to load the text into main memory in order to process it, even if the size of the final compressed representation could easily fit in RAM.

While fixed-order statistical methods are able to exploit only short text regularities [1], techniques such as Lempel-Ziv parsing (LZ77) [2], grammar compression [3], and run-length encoding of the Burrows-Wheeler transform [4, 5] have been shown superior in the task of compressing highly repetitive texts. Some recent works showed, moreover, that such efficient representations can be augmented without asymptotically increasing their space usage in order to support also fast search functionalities [6–8] (repetition-aware self-indexes). One of the most remarkable properties of such indexes is the possibility of representing (extremely) repetitive texts in *exponentially* less space than that of the text itself.

Among the above mentioned repetition-aware compression techniques, LZ77 has been shown to be more efficient than both grammar-compression [9] and run-length encoding of the Burrows-Wheeler transform (RLBWT) [6]. For this reason, much research is focusing into methods to efficiently build, access, and index LZ77-compressed text [8, 10]. A major concern while computing LZ77 and building LZ77-based self-indexes is to use limited working space. This is particular concerning in situations where the input text is highly repetitive: in these domains, algorithms working in space $\Theta(n \log n)$ [11], $\mathcal{O}(n \log |\Sigma|)$ [12, 13], or even $\mathcal{O}(nH_k)$ [14, 15] bits are of little

use as they could be exponentially more memory-demanding than the final compressed representation. Very recent results suggested that it is possible to achieve these goals in repetition-aware working space. Let z be the number of phrases of the LZ77 parse. Gagie [16] proposed a randomized algorithm to compute in $\mathcal{O}(n^{1+\epsilon})$ time and $\mathcal{O}(z/\epsilon)$ words of space an approximation of the parsing consisting of $\mathcal{O}(z/\epsilon)$ phrases, where $0 < \epsilon \leq 1$. Nishimoto et al. in [17] show how to build the LZ77 parsing in $\mathcal{O}(z \log n \log^* n)$ words of space.

In this work, we focus on the measure of repetitiveness $\mathcal{R}$: the number of equal-letter runs in the BWT of the (reversed) text. Several works [4–6] studied the empirical behavior of $\mathcal{R}$ on highly repetitive text collections, suggesting that on such instances $\mathcal{R}$ grows at the same rate as z. Let $\Sigma = \{s_1, \ldots, s_\sigma\}$ be the alphabet. Both z and $\mathcal{R}$ are at least σ and can be $\Theta(\sigma)$, e.g. in the text $(s_1 s_2 \ldots s_\sigma)^e$, $e > 0$. However, the rate $\mathcal{R}/z$ can be $\Theta(\log_\sigma n)$: this happens, for example, for de Bruijn sequences (of order $k > 1$). In this paper, we show how to build the LZ77 parsing of a text T in space bounded by the number $\mathcal{R}$ of runs in the BWT of T reversed. The main obstacle in computing the LZ77 parsing with a RLBWT index within *repetition-aware space* is the suffix array (SA) sampling: by sampling the SA at regular text positions, this structure requires $\mathcal{O}((n/k) \log n)$ bits of working space and supports *locate* queries in time proportional to k (for any $0 < k \leq n$). In this work we prove that—in order to compute the LZ77 parsing—it is sufficient to store at most two samples per BWT run, therefore reducing the sampling size to $\mathcal{O}(\mathcal{R} \log n)$ bits. Our algorithm reads the text *only once* from left to right in $\mathcal{O}(\log \mathcal{R})$ time per character (which makes it useful also in the streaming model). After reading the text, $\mathcal{O}(n \log \mathcal{R})$ additional time is required in order to output the LZ77 phrases in text-order (the parsing itself is not stored in main memory). The total space usage is $\mathcal{O}(\mathcal{R} \log n)$ bits.

A consequence of our result is that a class of repetition-aware self-indexes—described in [6]—combining LZ77 and RLBWT can be built in asymptotically optimal $\mathcal{O}(z + \mathcal{R})$ words of working space. The only other known repetition-aware index that can be built in asymptotically optimal working space is based on grammar compression and is described in [18].

Notation

For space constraints we assume the reader to be familiar with the notions related to *BWT*-based self indexing: backward search, LF mapping, suffix array sampling.

Let our input text be of the form $T = \#T'\$ \in \Sigma^n$, with $T' \in (\Sigma \setminus \{\$, \#\})^{n-2}$, \$ LZ77-terminator, and #—lexicographically smaller than all elements in Σ—BWT-terminator[1].

The *LZ77 parsing* (or *factorization*) of a text T is the stream of z *phrases* (or *factors*)

$$\langle \pi_1, \lambda_1, c_1 \rangle \ldots \langle \pi_i, \lambda_i, c_i \rangle \ldots \langle \pi_z, \lambda_z, c_z \rangle,$$

where $\pi_i \in \{0, \ldots, n-1\} \cup \{\perp\}$ and $\perp$ stands for "undefined", $\lambda_i \in \{0, \ldots, n-2\}$, $c_i \in \Sigma$, and:

[1]We put # in first position since we will build the BWT of the *reverse* of T. Adding # to our input increases only by one the number of output LZ77 factors.

1. $T = \omega_1 c_1 \dots \omega_z c_z$, with $\omega_i = \epsilon$ if $\lambda_i = 0$ and $\omega_i = T[\pi_i, \dots, \pi_i + \lambda_i - 1]$ otherwise.
2. For any $i = 1, \dots, z$, ω_i is the *longest* prefix of $\omega_i c_i \dots \omega_z c_z$ that occurs at least twice in $\omega_1 c_1 \dots \omega_i$

The notation $\overleftarrow{S}$ indicates the reverse of the string $S \in \Sigma^*$. All BWT intervals are inclusive, and we denote them as $[l, r]$ (left-right positions on the BWT). An (equal-letter) a-run in a string S is a maximal single-character a substring of S.

A substring V of a string $S \in \Sigma^*$ is *right-maximal* if there exist two distinct characters $a \neq b,\ a, b \in \Sigma$ such that both Va and Vb are substrings of S.

Algorithm

We now describe our algorithm, deferring a detailed description of the employed data structures to the next section. The main data structure we use is a dynamic RLBWT of the text $\overleftarrow{T}$. The algorithm works in two phases.

In the first phase, we read T from left to right, building a RLBWT representation of $\overleftarrow{T}$. This step employs a well-known online BWT construction algorithm which requires a dynamic string data structure D to represent the BWT. The algorithm performs a total amount of $|T|$ *rank* and *insert* operations on D (see, e.g., [15] for a detailed description of the procedure). In our case, D will be designed to be also *run-length compressed* (see next section for all technicalities).

In the second phase, the algorithm scans T left to right once more, this time using the BWT just built—i.e. by repeatedly using the LF mapping on the entire BWT of $\overleftarrow{T}$ starting from $T[0]$— and outputs the LZ77 factors.

While reading $T[j]$ for $j > 0$ in the second phase, we must determine whether $T[i, \dots, j]$, with i first position of the current LZ-phrase, occurs in $T[0, \dots, j-1]$. If this is not the case, then we output the LZ triple $\langle \pi, j-i, T[j] \rangle$, where π corresponds to the source of the current LZ-phrase (and, hence, $T[\pi, \dots, \pi+j-i-1] = T[i, \dots, j-1]$ and $\pi = \perp$ in case $i = j$). The computation must be performed on an index for the *entire* text.

In the following we show how to implement our algorithm in $\mathcal{O}(\mathcal{R} \log n)$ bits of working space, by maintaining σ dynamic sets equipped with a total of $\mathcal{O}(\mathcal{R})$ SA-samples.

Dynamic Suffix Array Sampling

From now on *BWT* stands for the Burrows-Wheeler transform of $\overleftarrow{T}$. Note that, even though we say that we *sample the suffix array*, we actually sample text positions associated with BWT positions, i.e. we sample T-positions on the L-column instead of T-positions on the F-column of the BWT matrix. Moreover, since we enumerate positions in T-order (not $\overleftarrow{T}$-order), k-th BWT-position will correspond to sample $(n - SA[k]) \bmod n$, where $SA[k]$ is the k-th entry in the (standard) suffix array of $\overleftarrow{T}$.

Let j be a T-position and k its corresponding BWT-position: $T[j] = BWT[k]$. We store SA-samples as pairs $\langle j, k \rangle$ and each pair is of one of three types: *singleton*,

denoted as $\langle j,k\rangle^\circ$, *open*, denoted as ${}^{\lceil}\langle j,k\rangle$, and *close*, denoted as $\langle j,k\rangle^{\rceil}$. If the pair type is not relevant for the discussion, we simply write $\langle j,k\rangle$.

Let $\Sigma = \{s_1,\ldots,s_\sigma\}$ be the alphabet. Samples are stored in σ red-black trees $\mathcal{B}_{s_1},\ldots,\mathcal{B}_{s_\sigma}$ and are ordered by BWT coordinate (i.e. the second component of the pairs). While reading $a = T[j] = BWT[k]$ we first locate the (inclusive) bounds $l \le k \le r$ of its associated BWT a-run, then we update the trees according to the following rules:

(A) If for all $\langle j',k'\rangle \in \mathcal{B}_a$, $k' \notin [l,r]$, then we insert the singleton $\langle j,k\rangle^\circ$ in $\mathcal{B}_a$.

(B) If there exists $\langle j',k'\rangle^\circ \in \mathcal{B}_a$ such that $k' \in [l,r]$, then we remove it and:

  (a) If $k < k'$, then we insert in $\mathcal{B}_a$ the pairs ${}^{\lceil}\langle j,k\rangle$ and $\langle j',k'\rangle^{\rceil}$,

  (b) If $k' < k$, then we insert in $\mathcal{B}_a$ the pairs ${}^{\lceil}\langle j',k'\rangle$ and $\langle j,k\rangle^{\rceil}$.

(C) If there exist ${}^{\lceil}\langle j',k'\rangle, \langle j'',k''\rangle^{\rceil} \in \mathcal{B}_a$ such that $k',k'' \in [l,r]$:

  (a) If $k < k' < k''$, then we remove ${}^{\lceil}\langle j',k'\rangle$ from $\mathcal{B}_a$ and insert ${}^{\lceil}\langle j,k\rangle$ in $\mathcal{B}_a$,

  (b) If $k' < k'' < k$, then we remove $\langle j'',k''\rangle^{\rceil}$ from $\mathcal{B}_a$ and insert $\langle j,k\rangle^{\rceil}$ in $\mathcal{B}_a$,

  (c) Otherwise ($k' < k < k''$), we leave the trees unchanged.

We say that a BWT a-run $BWT[l,\ldots,r]$ *contains a pair* or, equivalently, *contains a SA-sample*, if there exists some $\langle j,k\rangle \in \mathcal{B}_a$ such that $k \in [l,r]$. It is easy to see that the following invariants hold for the above three rules: (i) each BWT run contains either no pairs, a singleton pair, or two pairs—one open and one close; (ii) If a BWT run contains an open ${}^{\lceil}\langle j',k'\rangle$ and a close $\langle j'',k''\rangle^{\rceil}$ pair, then $k' < k''$; (iii) once we add a SA-sample inside a BWT run, that run will always contain at least one SA-sample.

We say that BWT-position k is *marked by SA-sample* $\langle j,k\rangle$, when $a = T[j] = BWT[k]$ and $\langle j,k\rangle \in \mathcal{B}_a$.

Let $BWT[k_\#] = \#$. By saying that T-positions $0,\ldots,j$ have been *processed*, we mean that—starting with all trees empty—we have applied the update rules to the SA-samples $\langle 0,k_\#\rangle$, $\langle 1,BWT.LF(k_\#)\rangle$, $\langle 2,BWT.LF^2(k_\#)\rangle,\ldots,\langle j,BWT.LF^j(k_\#)\rangle$, where $BWT.LF^i(k_\#)$ denotes i applications of the LF map starting from BWT-position $k_\#$. We now prove that, after processing $0,\ldots,j$, we can locate at least one occurrence of any string that occurs in $T[0,\ldots,j]$. This property will allow us to locate LZ phrase boundaries and previous occurrences of LZ phrases.

Lemma 1. *If $0,\ldots,j$ have been processed and $[l,r]$ is the BWT interval associated with $\overleftarrow{V} \in \Sigma^m$, with V* right-maximal *in T, then*

$$\exists \langle j',k'\rangle \in \mathcal{B}_a \text{ such that } k' \in [l,r] \text{ if and only if } Va \text{ occurs in } T[0,\ldots,j].$$

Proof. ($\Rightarrow$) If $\langle j', k' \rangle \in \mathcal{B}_a$ with $k' \in [l, r]$ exists, then clearly $T[j' - m, \ldots, j'] = Va$. Moreover, since we processed T-positions $0, \ldots, j$ only, it must be the case that $j' \leq j$ and hence Va occurs in $T[0, \ldots, j]$.

($\Leftarrow$) Let $T[t, \ldots, t + m] = Va$, with $t \leq j - m$. Consider the BWT a-run corresponding to $T[t + m] = a$. One of the following cases can hold true:

(1) The BWT a-run is entirely included in $BWT[l, \ldots, r]$ *and* is neither a prefix nor a suffix of $BWT[l, \ldots, r]$, that is $BWT[l, \ldots, r] = Xca^e dY$, for some $X, Y \in \Sigma^*,\ c, d \neq a, e > 0$. Then, it follows from invariant (iii) and rule (A) that since we have visited T-position $t + m$, the a-run must contain at least one SA-sample. This is the pair $\langle j', k' \rangle$ we are looking for.

(2) The BWT a-run spans either position l or position r. Since V is right-maximal in T, then $BWT[l, \ldots, r]$ contains also a character $b \neq a$. We therefore have that either (i) $BWT[l, \ldots, r] = a^e XbY$, or (ii) $BWT[l, \ldots, r] = YbXa^e$, where $X, Y \in \Sigma^*,\ e > 0$. The two cases are symmetric hence we discuss only (i).

Consider all T-prefixes $T[0, \ldots, j'']$ such that $j'' \leq j$, Va is a suffix of $T[0, \ldots, j'']$, and the lexicographic rank of $\overleftarrow{T[0, \ldots, j'' - 1]}$ among all $\overleftarrow{T}$-suffixes is $k'' \in [l, l + e - 1]$ (i.e. the suffix lies in $BWT[l, \ldots, l + e - 1] = a^e$). There exists at least one such T-prefix: $T[0, \ldots, t + m]$. Then, it is easy to see that the rank k' of the lexicographically largest $\overleftarrow{T}$-suffix with the above properties is such that $\langle j', k' \rangle \in \mathcal{B}_a$ for some $j' \leq j$. This is implied by the three update rules described above. The BWT position k corresponding to T-position $t + m$ lies in the BWT interval $[l, l + e - 1]$, therefore either (i) k is the rightmost position visited in its run (and it is marked with a SA-sample), or (ii) the rightmost visited position $k' > k$ in $[l, l + e - 1]$ is marked with a SA-sample (note that *lexicographically largest* translates to *rightmost* on BWT intervals). □

We can drop the right-maximality requirement from Lemma 1.

Corollary 1. *Once processed T-positions $0, \ldots, j-1$ (none if $j = 0$), after processing also $j, \ldots, j + m - 1$, $m > 0$, if a string $W \in \Sigma^m$ occurs in $T[0, \ldots, j + m - 1]$, then we can locate one of its occurrences.*

Proof. We prove the property by induction on $|W| = m > 0$. Let $W = Va,\ V \in \Sigma^{m-1},\ a \in \Sigma$. If $m = 1$, then $V = \epsilon$ (empty string). Since T contains at least two distinct characters (a and #), V is right-maximal. Therefore we can apply Lemma 1 to find an occurrence of $W = a$.

If $m > 1$, then $|V| > 0$ and two cases can occur. If V is right-maximal, then we can again apply Lemma 1 to find an occurrence of $W = Va$ in $T[0, \ldots, j + m - 1]$ (remember that Va occurs in $T[0, \ldots, j + m - 1]$). If, instead, V is not right-maximal, then it is always followed by a in T. By inductive hypothesis we can locate an occurrence π of V in $T[0, \ldots, j + m - 2]$. But then, since all occurrences of V in T are followed by a, π is also an occurrence of $W = Va$ in $T[0, \ldots, j + m - 1]$. □

Pseudocode

Our complete procedure is reported as Algorithm 1. In line 1 we build the RLBWT of $\overleftarrow{T}$ using the online algorithm mentioned at the beginning of this section and

employing a dynamic run-length encoded string data structure to represent the BWT. This is the only step requiring access to the input text, which is read only once from left to right. Since the dynamic string we use is run-length compressed, this step requires $\mathcal{O}(\mathcal{R} \log n)$ bits of working space.

From lines 2 to 9 we initialize all variables. In order: the text length n, the current position j in T, the position k in $RLBWT$ corresponding to position j in T (at the beginning, $T[0] = RLBWT[k_{\#}] = \#$), the current LZ77 phrase prefix length λ (last character $T[j]$ excluded), the T-position $\pi < j$ at which the current phrase prefix $T[j-\lambda, \ldots, j-1]$ occurs ($\pi = \perp$ if $\lambda = 0$), the red-black trees $\mathcal{B}_{s_1}, \ldots, \mathcal{B}_{s_\sigma}$ used to store SA-samples, the current character $c = T[j] = RLBWT[k]$, and the interval $[l, r]$ corresponding to the current reversed LZ phrase prefix $\overleftarrow{T[j-\lambda, \ldots, j-1]}$ in $RLBWT$ (when $\lambda = 0$, $[l, r]$ is the full interval $[0, n-1]$).

The *while* loop on line 10 scans T positions from the first to last. First of all, we have to discover if the current character $T[j] = c$ ends a LZ phrase. In line 11 we count the number u of runs that intersect interval $[l, r]$ on $RLBWT$. If $u = 1$, then the current phrase prefix $T[j-\lambda, \ldots, j-1]$ is always followed by c in T (i.e. it is not right-maximal), and consequently $T[j]$ cannot be the last character of the current LZ phrase. Otherwise, by Lemma 1 $T[j-\lambda, \ldots, j]$ occurs in $T[0, \ldots, j-1]$ if and only if there exists a SA-sample $\langle j', k' \rangle \in \mathcal{B}_c$ such that $l \leq k' \leq r$. The existence of such pair can be verified with a binary search on the red-black tree $\mathcal{B}_c$. In line 12 we perform these two tests. If at least one of these two conditions holds, then $T[j-\lambda, \ldots, j]$ occurs in $T[0, \ldots, j-1]$ and therefore it is not a LZ phrase. If this is the case, we now have to find $\pi < j - \lambda$ such that $T[\pi, \ldots, \pi+\lambda] = T[j-\lambda, \ldots, j]$ (i.e. a previous occurrence of the current LZ phrase prefix). The implementation of this task follows the inductive proof of Corollary 1. If $u = 1$ (current phrase prefix is not right-maximal) then π is already the value we need. Otherwise (Lines 13-14) we find a SA-sample $\langle j', k' \rangle \in \mathcal{B}_c$ such that $k' \in [l, r]$ (such pair must exist since $u > 1$ and the condition in Line 12 succeeded). Procedure $\mathcal{B}_c.locate(l, r)$ returns such j' (to make the procedure deterministic, one could return the value j' associated with the smallest BWT position $k' \in [l, r]$). Then, we assign to π the value $j' - \lambda$ (Line 14). We can now increment the current LZ phrase prefix length (Line 15) and update the BWT interval $[l, r]$ so that it corresponds to the string $\overleftarrow{T[j-\lambda+1, \ldots, j]}$ (LF mapping in Line 16).

If both the conditions at line 12 fail, then the string $T[j-\lambda, \ldots, j]$ does not occur in $T[0, \ldots, j-1]$ and therefore is a LZ phrase. By the inductive hypothesis of Corollary 1, $\pi < j-\lambda$ is either $\perp$—if $\lambda = 0$—or such that $T[\pi, \ldots, \pi+\lambda-1] = T[j-\lambda, \ldots, j-1]$ otherwise. At line 18 we can therefore output the LZ factor. We now have to open (and start searching in RLBWT) a new LZ phrase: at lines 19-21 we reset the current phrase prefix length, set π to $\perp$, and reset the interval associated to the current (reversed) phrase prefix to the full interval.

All we are left to do now is to process position j (i.e. apply the update rules to the SA-sample $\langle j, k \rangle$) and proceed to the next text position. At line 22 we locate the (inclusive) borders $[l_{run}, r_{run}]$ of the BWT run containing position k (i.e. $k \in [l_{run}, r_{run}]$). This information is used at line 23 to apply the update rules on $\mathcal{B}_c$

and on the SA-sample $\langle j, k\rangle$. Finally, we increment the current T-position j (line 24), compute the corresponding position k on RLBWT (line 25), and read the next T-character c on the RLBWT.

Algorithm 1: LZ77_in_RLE_space(T)

input : A text $T \in \Sigma^n$ beginning with # and ending with \$
output: LZ77 factors of T in text order.

```
1  RLBWT ← build_rev_RLBWT(T);  /* Build online the RLBWT of the reverse of T */
2  n ← |T|;                                                    /* T length */
3  j ← 0 ;        /* Last position (on T) of current LZ phrase prefix */
4  k ← k_# ;                                  /* Position of # in RLBWT */
5  λ ← 0;                          /* Length of current LZ phrase prefix */
6  π ← ⊥;          /* Previous occurrence of current LZ phrase prefix */
7  B_{s_1}, ..., B_{s_σ} ← ∅;       /* Initialize red-black trees of SA-samples */
8  c ← RLBWT[k];                                  /* Current T character */
9  [l, r] ← [0, n − 1]; /* Range of current LZ phrase prefix in RLBWT */
10 while j < n do
11     u ← RLBWT.number_of_runs(l, r);        /* Runs intersecting [l, r] */
12     if u = 1 or B_c.exists_sample(l, r) then
13         if u > 1 then
14             π ← B_c.locate(l, r) − λ;       /* Occurrence of phrase prefix */
15         λ ← λ + 1;             /* Increase length of current LZ phrase */
16         [l, r] ← RLBWT.LF([l, r], c);             /* Backward search step */
17     else
18         Output ⟨π, λ, c⟩;                          /* Output LZ77 factor */
19         λ ← 0;                              /* Reset phrase prefix length */
20         π ← ⊥;                         /* Reset phrase prefix occurrence */
21         [l, r] ← [0, n − 1]; /* Reset range of current LZ phrase prefix */
22     [l_run, r_run] ← RLBWT.locate_run(k) ;      /* run of BWT position k */
23     B_c.update_tree(⟨j, k⟩, [l_run, r_run]);              /* Apply update rules */
24     j ← j + 1;                                  /* Increment T position */
25     k ← RLBWT.LF(k);         /* RLBWT position corresponding to j */
26     c ← RLBWT[k];                             /* Read next T character */
```

Structures

Below we illustrate how to efficiently implement the dynamic RLBWT data structure used in Line 1 of Algorithm 1. We adopt the general approach of [4], that is run-length encoding of the FM index. We store one character per run in a string $H \in \Sigma^{\mathcal{R}}$,

we mark the beginning of the runs with a 1 in a bit-vector $V_{all}[0, \ldots, n-1]$, and for every $c \in \Sigma$ we store all c-runs lengths consecutively in a bit-vector V_c as follows: every m-length c-run is represented in V_c as 10^{m-1}. For example, letting $BWT = bc\#bbbbccccbaaaaaaaaaaa$, we have $H = bc\#bcba$, $V_{all} = 11110001000110000000000$, $V_a = 10000000000$, $V_b = 110001$, and $V_c = 11000$ ($V_\#$ is always 1). Then, *rank/access* on the BWT are reduced to *rank/select/access* on H, V_{all}, and V_c. The structure takes $\mathcal{O}(\mathcal{R} \log n)$ bits of space if all bit-vectors are gap-encoded and supports the insertion of character c in the BWT, by (possibly) one character insertion in H followed by a constant number of *rank*, *select*, *insertions* and *deletions of 0-bits* in V_{all} and V_c. We leave details to the reader.

All structures are implemented as dynamic. For H we can use the result in [19], guaranteeing $\mathcal{O}(\mathcal{R} \log n)$ bits of space (in [19] there is an extra $\mathcal{O}(\sigma \log \mathcal{R})$ spatial term which however amounts to $\mathcal{O}(\mathcal{R} \log n)$ bits since $\sigma \leq \mathcal{R} \leq n$) and $\mathcal{O}(\log \mathcal{R})$-time *rank*, *select*, *access*, and *insert*. We can reduce dynamic gap-encoded bit-vectors to the so-called *Searchable Partial Sums with Indels* (SPSI) problem. The SPSI asks for a data structure PS to maintain a sequence $s_1, \ldots, s_m$ of non-negative k-bits integers (in our case, $k \in \Theta(\log n)$, n being the text length), supporting the following operations:

- $PS.sum(i) = \sum_{j=1}^{i} s_j$;
- $PS.search(x)$ is the smallest i such that $\sum_{j=1}^{i} s_j \geq x$;
- $PS.update(i, \delta)$: update s_i to $s_i + \delta$. δ can be negative as long as $s_i + \delta \geq 0$;
- $PS.insert(i)$: insert 0 between s_{i-1} and s_i (if $i = 0$, insert in first position).

Below (section *SPSI implementation*) we briefly outline how to implement PS in $\mathcal{O}(m \cdot k)$ bits of space with $\mathcal{O}(\log m)$ time-cost for each of the above operations.

Hence, a length-n bit-vector $B = 10^{s_1-1}10^{s_2-1} \ldots 10^{s_m-1}$ ($s_i > 0$) can be encoded in $\mathcal{O}(m \log n)$ bits of space with a partial sum PS on the sequence $s_1, \ldots, s_m$. We need to show how to answer the following queries on B: $B[i]$ (*access*), $B.rank(i) = \sum_{j=0}^{i} B[j]$, $B.select(i)$ (the position j such that $B[j] = 1$ and $B.rank(j) = i$), $B.insert(i, b)$ (insert bit $b \in \{0, 1\}$ between positions $i-1$ and i), and $B.delete_0(i)$, where $B[i] = 0$ (delete $B[i]$).

It is easy to see that *rank/access* and *select* operations on B reduce to *search* and *sum* operations on PS, respectively. $B.delete_0(i)$ requires just a search and an update on PS. To support *insert* on B, we can operate as follows: $B.insert(i, 0)$, $i > 0$, is implemented with $PS.update(PS.search(i), 1)$. $B.insert(0, 1)$ is implemented with $PS.insert(0)$ followed by $PS.update(0, 1)$. $B.insert(i, 1)$, $i > 0$, "splits" an integer into two integers: let $j = PS.search(i)$ and $\delta = PS.sum(j) - i$. We first decrease s_j with $PS.update(j, -\delta)$. Then, we insert a new integer $\delta + 1$ with $PS.insert(j + 1)$ and $PS.update(j + 1, \delta + 1)$.

SPSI implementation

In our case, the bit-length of the integers in each of our PS-structures is $k \in \Theta(\log n)$. We can use $\mathcal{O}(m \cdot k) = \mathcal{O}(m \log n)$ bits of space by employing red-black trees (RBT).

We store $s_1, \ldots, s_m$ in the leaves of a RBT and we store in each internal node of the tree the number of nodes and partial sum of its subtrees. *Sum* and *search* queries can then be easily implemented with a traversal of the tree from the root to the target leaf. *Update* queries require finding the integer (leaf) of interest and then updating $\mathcal{O}(\log m)$ partial sums while climbing the tree from the leaf to the root. Finally, *insert* queries require finding an integer (leaf) s_i immediately preceding or following the insert position, substituting it with an internal node with two children leaves s_i and 0 (the order depending on the insert position—before or after s_i), incrementing by one $\mathcal{O}(\log m)$ subtree-size counters while climbing the tree up to the root, and applying the RBT update rules. This last step requires the modification of $\mathcal{O}(1)$ counters (subtree-size/partial sum) if RBT rotations are involved. All operations take $\mathcal{O}(\log m)$ time.

Analysis

It is easy to see that *rank, access*, and *insert* operations on RLBWT take $\mathcal{O}(\log \mathcal{R})$ time each. Operations $\mathcal{B}_c.exists\_sample(l, r)$ (line 12) and $\mathcal{B}_c.locate(l, r)$ (Line 14) require just a binary search on the red-black tree of interest and can also be implemented in $\mathcal{O}(\log \mathcal{R})$ time. $RLBWT.number\_of\_runs(l, r)$ is the number of bits set in $V_{all}[l, \ldots, r]$, plus 1 if $V_{all}[l] = 0$: this operation requires therefore $\mathcal{O}(1)$ *rank/access* operations on V_{all} ($\mathcal{O}(\log \mathcal{R})$ time). Similarly, $RLBWT.locate\_run(k)$ requires finding the two bits set preceding and following position k in V_{all} ($\mathcal{O}(\log \mathcal{R})$ time with a constant number of *rank* and *select* operations). We obtain:

Theorem 1. *Algorithm 1 computes the LZ77 factorization of a text $T \in \Sigma^n$ in $\mathcal{O}(\mathcal{R} \log n)$ bits of working space and $\mathcal{O}(n \log \mathcal{R})$ time, $\mathcal{R}$ being the number of runs in the Burrows-Wheeler transform of $\overleftarrow{T}$.*

As a direct consequence of Theorem 1, we obtain an asymptotically optimal-space construction algorithm for a class of repetition-aware indexes [6] combining a RLBWT with the LZ77 factorization. The main idea behind this class of indexes is to use a RLBWT structure on $\overleftarrow{T}$ to compute the lexicographic order of the reversed LZ77 T-factors and of the T-suffixes starting at LZ77 phrase boundaries. This, combined with two geometric range data structures, permits to efficiently count and locate pattern occurrences in T within $\mathcal{O}(\mathcal{R} + z)$ words of space (see [6] for full details). The construction of such indexes requires building the RLBWT of $\overleftarrow{T}$, computing the LZ77 factorization of T, and building additional structures of $\mathcal{O}(z)$ words of space. We observe that with our algorithm all these steps can be carried out in $\mathcal{O}(\mathcal{R} + z)$ words of working space, which is (asymptotically) the same space of the index described in [6].

References

[1] Travis Gagie, "Large alphabets and incompressibility," *Information Processing Letters*, vol. 99, no. 6, pp. 246–251, 2006.

[2] Jacob Ziv and Abraham Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[3] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat, "The smallest grammar problem," *Information Theory, IEEE Transactions on*, vol. 51, no. 7, pp. 2554–2576, 2005.

[4] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro, "Run-length compressed indexes are superior for highly repetitive sequence collections," in *String Processing and Information Retrieval*. Springer, 2009, pp. 164–175.

[5] Jouni Sirén et al., *Compressed full-text indexes for highly repetitive collections*, Ph.D. thesis, 2012.

[6] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot, "Composite repetition-aware data structures," in *Proc. CPM*, 2015, pp. 26–39.

[7] Francisco Claude and Gonzalo Navarro, "Self-indexed grammar-based compression," *Fundamenta Informaticae*, vol. 111, no. 3, pp. 313–337, 2011.

[8] Sebastian Kreft and Gonzalo Navarro, "Self-indexing based on LZ77," in *Combinatorial Pattern Matching*. Springer, 2011, pp. 41–54.

[9] Wojciech Rytter, "Application of lempel–ziv factorization to the approximation of grammar-based compression," *Theoretical Computer Science*, vol. 302, no. 1, pp. 211–222, 2003.

[10] Djamal Belazzougui, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei, "Queries on lz-bounded encodings," *arXiv preprint arXiv:1412.0967*, 2014.

[11] Maxime Crochemore and Lucian Ilie, "Computing longest previous factor in linear time and applications," *Information Processing Letters*, vol. 106, no. 2, pp. 75–80, 2008.

[12] Enno Ohlebusch and Simon Gog, "Lempel-Ziv factorization revisited," in *Combinatorial Pattern Matching*. Springer, 2011, pp. 15–26.

[13] Djamal Belazzougui and Simon J Puglisi, "Range Predecessor and Lempel-Ziv Parsing," *arXiv preprint arXiv:1507.07080*, 2015.

[14] Sebastian Kreft and Gonzalo Navarro, "Self-index based on LZ77 (Ph.D. thesis)," *arXiv preprint arXiv:1112.4578*, 2011.

[15] Alberto Policriti and Nicola Prezza, "Fast online lempel-ziv factorization in compressed space," in *String Processing and Information Retrieval*, vol. 9309 of *Lecture Notes in Computer Science*, pp. 13–20. Springer International Publishing, 2015.

[16] Travis Gagie, "Approximating lz77 in small space," *arXiv preprint arXiv:1503.02416*, 2015.

[17] Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, et al., "Dynamic index, LZ factorization, and lce queries in compressed space," *arXiv preprint arXiv:1504.06954*, 2015.

[18] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto, "Online self-indexed grammar compression," in *proc. SPIRE*, vol. 9309 of *Lecture Notes in Computer Science*, pp. 258–269. Springer International Publishing, 2015.

[19] Gonzalo Navarro and Yakov Nekrich, "Optimal dynamic sequence representations," *SIAM Journal on Computing*, vol. 43, no. 5, pp. 1781–1806, 2014.