LZ77 Computation Based on the Run-Length Encoded BWT

Alberto Policriti[1,2] and Nicola Prezza[1]

1   Department of Informatics, Mathematics, and Physics, University of Udine,
    Italy.
2   Applied Genomics Institute, Udine.

Abstract

Computing the LZ77 factorization is a fundamental task in text compression and indexing, being the size z of this compressed representation closely related to the self-repetitiveness of the text. A long-standing problem is to compute LZ77 using small working space. Considering that $\mathcal{O}(z)$ words of space can be significantly (up to *exponentially*) smaller than the size n of the input text, even succinct and entropy-compressed solutions are often unduly memory demanding.

In this work we focus on an important measure of text repetitiveness: the number r of equal-letter runs in the Burrows-Wheeler transform of the reversed input text. As z, the measure r is closely related to the number of repetitions in the text and can be exponentially smaller than n.

We describe two algorithms computing LZ77 in $\mathcal{O}(r \log n)$ bits of working space and $\mathcal{O}(n \log r)$ time. Roughly speaking, our algorithms store a constant number of memory words per BWT run to keep track of first-last run-positions and a suitable indexing mechanism to sample the *runs* of the BWT (instead of its positions).

Important consequences of our results include (i) the possibility to convert from RLBWT- to LZ77-based compressed formats *without* first decompressing the text, and (ii) the existence of asymptotically-optimal construction algorithms for repetition-aware self-indexes based on these compression techniques.

We finally describe an implementation of our solutions and present extensive experiments on highly repetitive datasets. Our algorithms use a working space as small as 1% of the dataset size and are two to three orders of magnitude more space-efficient (albeit slower) than existing solutions based, respectively, on entropy compression and suffix arrays.

Keywords and phrases run-length encoded BWT, Lempel-Ziv factorization, repetitive text collections, repetition-aware data structures.

1   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 individuals of 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 [15], techniques such as Lempel-Ziv parsing (LZ77) [36], grammar compression [7], and run-length encoding of the Burrows-Wheeler transform [32, 31] 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 [3, 8, 22] (repetition-aware self-indexes). One of the most remarkable properties of such indexes is the possibility of representing (extremely) repetitive texts in (up to) *exponentially* less space than that of the text itself.

Among the above mentioned repetition-aware compression techniques, LZ77 has been shown to be superior to both grammar-compression [30] and run-length encoding of the Burrows-Wheeler transform (RLBWT) [3]. For this reason, much research is focusing into methods to efficiently build, access, and index LZ77-compressed text [4, 22]. 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 $\Theta(n \log n)$ [9], $\mathcal{O}(n \log |\Sigma|)$ [26, 5], or even $\mathcal{O}(nH_k)$ [22, 29] bits are of little use as they could be much 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. Fischer et al. in [14] proposed a randomized algorithm to compute in $\mathcal{O}(\epsilon^{-1} n \log n)$ time and $\mathcal{O}(z)$ words of space an approximation of the parsing consisting of at most $(1 + \epsilon)z$ phrases, where $0 < \epsilon \leq 1$. Nishimoto et al. in [25] 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 r of repetitiveness: the number of equal-letter runs in the BWT of the (reversed) text. Several works [3, 31, 32] studied the empirical behavior of r on highly repetitive text collections, suggesting that on such instances r grows at the same rate as z. Let $\Sigma = \{s_1, \ldots, s_\sigma\}$ be the alphabet. Both z and 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, infinite families of strings for which $r/z \in \Theta(\log_\sigma n)$ exist: this happens, for example, in de Bruijn sequences of order $k > 1$. To see this, consider the BWT row-partition induced by length-$(k-1)$ contexts. Each $x \in \Sigma^{k-1}$ appears exactly σ times in the de Bruijn sequence and all such occurrences are preceded by different characters. It follows that each of the above BWT partitions contains at least $\sigma - 1$ runs, so the BWT has at least $(\sigma - 1)\sigma^{k-1} \in \mathcal{O}(\sigma^k) = \mathcal{O}(n)$ runs. The number of LZ77 phrases of any text is, on the other hand, always $\mathcal{O}(n/\log_\sigma n)$ [36]. The opposite relation $z/r \in \Theta(\log_\sigma n)$ also holds true for certain families of strings. This is the case—for example—of Fibonacci words. Such words are defined recursively as follows: $f_1 = a$, $f_2 = b$, $f_n = f_{n-1} f_{n-2}$. Fibonacci words are a particular case of *standard words*; such words produce a total clustering of the alphabet letters in the BWT [23] (i.e. two runs). On the other hand, the LZ77 factorization of f_n corresponds to the factorization of f_n into *singular words* $\hat{f}_i$, where each $\hat{f}_i$ is obtained by complementing the first letter in the left rotation of the Fibonacci word f_i (see [13] for more details). Since $|f_i|$ is exponential in i, it follows that the Lempel-Ziv factorization of f_n has $\Theta(\log |f_n|)$ factors. We emphasize the fact that the algorithms presented in this work use a space proportional to the number of runs in the BWT of the *reversed* text. Experimentally it has been observed [3] that the number of runs in $BWT(T)$ and $BWT(\overleftarrow{T})$ are two measures of repetitiveness that behave very similarly. This should be expected since, if T is very repetitive, then so is $\overleftarrow{T}$. However, we are not aware of theoretical results relating in a more precise way the two measures.

The main obstacle in building LZ77 within $\mathcal{O}(r \log n)$ bits of space with a run-length encoded FM-index is the suffix array (SA) sampling: by sampling the SA every $0 < k \leq n$ text positions, this structure takes $\mathcal{O}((n/k) \log n)$ bits of space and supports *locate* queries in time proportional to k. The main contributions of this work are two algorithms that compute LZ77 by combining a (dynamic) run-length BWT with a repetition-aware sparse suffix array sampling. The first algorithm stores only two samples per BWT equal-letter run, while the

second stores at most one sample per LZ77 factor. Both algorithms run in $\mathcal{O}(n \log r)$ time and require $\mathcal{O}(r \log n)$ bits of working space.

As a by-product of our results we obtain a $\mathcal{O}(r \log n)$-space algorithm to convert from RLBWT- to LZ77-based compressed formats. This is one of the first works showing how to *convert* a compressed format into another without first decompressing the text; see [1, 2, 30] (grammar compression to/from Lempel-Ziv), and [34] (run-length encoding of the text to LZ78) for similar results. Another important application of our results is related to text indexing. In particular, we obtain that indexes based on combinations of LZ77 and RLBWT compressors—see, e.g. [3]—can be built in asymptotically optimal $\mathcal{O}(z+r)$ words of working space. To the best of our knowledge, the only other repetition-aware index that can be built in asymptotically optimal working space is based on grammar compression and is described in [33].

The paper is organized as follows. We first describe—in Section 3—a dynamic run-length encoded string data structure. This structure is used in our algorithms to build online and in small space the RLBWT of the reversed input text. In Sections 4 and 5 we describe our two algorithms to compute LZ77 in repetition-aware working space.

We conclude by presenting a C++ implementation of our algorithms and extensive results on highly repetitive datasets. Our implementation is available as part of the DYNAMIC library [10], featuring several dynamic compressed data structures. In some real-case scenarios, our algorithms are two and three orders of magnitude more space-efficient than existing solutions based, respectively, on entropy compression and suffix arrays. This space efficiency is, however, paid in terms of running times, which in some cases are up to two orders of magnitude higher than those of suffix array-based algorithms.

2   Preliminaries

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. We put # in first position since we will build the BWT of the *reverse* of T. We assume, for simplicity, that we are working on an integer alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ (in this respect, we reserve codes 0 and 1 for the two terminators).

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. $T = \omega_1 c_1 \ldots \omega_z c_z$, with $\omega_i = \epsilon$ if $\lambda_i = 0$ and $\omega_i = T[\pi_i, \ldots, \pi_i + \lambda_i - 1]$ otherwise, with $\pi_i < |\omega_1 c_1 \ldots \omega_{i-1} c_{i-1}|$
2. For any $i = 1, \ldots, z$, the string ω_i is the *longest* string that occurs at least twice in $\omega_1 c_1 \ldots \omega_i$

The notation $\overleftarrow{S}$ indicates the reverse of the string $S \in \Sigma^*$.

An (equal-letter) *run* in a string S is a maximal substring a^k, with $k > 0$ and $a \in \Sigma$.

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.

The *Burrows-Wheeler* transform $BWT(S)$ of a (#-terminated) string S is the S-permutation obtained by sorting all circular permutations of S in a *conceptual* matrix of size $n \times n$ and by taking the last column of this matrix [6]. With *F-* and *L-positions* we denote positions on the

first an last column of the BWT, respectively. The *LF mapping* is a function associating to each L-position i its corresponding F-position i' (i.e. i and i' correspond to the same text's position). We denote this function as $BWT.LF(i)$ (BWT being a structure representing the Burrows-Wheeler transform of some string and supporting LF-function computation). Let V be a substring of S. Note that V is a prefix of some interval $[l, r]$ of rows in the matrix representation of $BWT(S)$; we call $[l, r]$ the $BWT(S)$ *interval* of V (or simply *BWT interval* when S is clear from the context). The LF mapping naturally extends to BWT intervals. Letting $[l, r]$ be the BWT interval of some string V and $c \in \Sigma$, $BWT.LF([l, r], c)$ returns the BWT interval $[l', r']$ of string cV. All BWT intervals are inclusive, and we denote them as $[l, r]$ (left-right positions on the BWT).

We indicate by $r(S)$ (or simply by r, when clear from the context) the number of (equal-letter) runs of $BWT(S)$. Throughout the paper, we will work with $BWT(\overleftarrow{S})$. As a consequence, the quantity r will always denote the number of (equal-letter) runs of $BWT(\overleftarrow{S})$. A run-length encoded representation of $BWT(S)$—to be denoted as $RLBWT(S)$—is any representation of $BWT(S)$ storing it as a sequence of runs and taking therefore space proportional to r words. See [32] for an example of such representation. We remind that the BWT can be turned into a *self-index* by encoding it with a structure supporting `rank` queries and by augmenting it with a sampling of the *suffix array* (see, e.g., [11, 12]). If a RLBWT is used, the resulting index takes space proportional to r plus the size of the suffix array sampling [3, 31, 32].

We recall that $BWT(\overleftarrow{S})$ can be built online with an algorithm that reads S-characters left-to-right and inserts them in a dynamic string data structure (see, e.g., [24, 28] for a detailed description of this algorithm). Let $a \in \Sigma$. Briefly, the algorithm is based on the idea of backward-searching the extended reversed text $\overleftarrow{Sa}$ in the BWT index for $\overleftarrow{S}$. This operation leads to an empty interval $[l, l)$ (since Sa does not appear in S) such that l is the lexicographic position of $\overleftarrow{Sa}$ among all $\overleftarrow{S}$'s suffixes. At this point, it is sufficient to insert $\#$ at position l in $BWT(\overleftarrow{S})$ and replace the old $\#$ with a to obtain $BWT(\overleftarrow{Sa})$.

3   Dynamic RLBWT Data Structure

In this section we describe a run-length encoded string data structure supporting `access` and `rank` operations. In the next sections we will use this structure to encode the BWT of the reversed input.

We adopt the general approach of [32], that is run-length encoding of the FM index. We store one character per run in a string $H \in \Sigma^r$, we mark the beginning of the runs with a 1 in a bit-vector $G_{all}[0, \ldots, n-1]$, and for every $c \in \Sigma$ we store all c-runs lengths consecutively in a bit-vector G_c as follows: every m-length c-run is represented in G_c as 10^{m-1}. For example, letting $BWT = bc\#bbbbccccbaaaaaaaaaaa$, we have $H = bc\#bcba$, $G_{all} = 111100010001100000000000$, $G_a = 10000000000$, $G_b = 110001$, and $G_c = 11000$ ($G_\#$ is always 1). Then, `rank`/`access` on the BWT are reduced to `rank/select/access` on H, G_{all}, and G_c. Briefly: to answer $rank_c(i)$ we count the number of bits set in G_{all} before position i and use this value to access the position j in H corresponding to the run containing position i. Then, with a $rank_c(j)$ query on H we retrieve the number k of c-runs before position i in the BWT. Finally, we call $select_1(k+1)$ on G_c to retrieve the number of c's contained in all c-runs appearing before position i in the BWT. Special care has to be taken in the case i falls *inside* a c-run. To answer $select_c(i)$, we proceed as follows. Suppose, for simplicity, that the i-th c is the first of its c-run (the general case is slightly more complicated and we do not discuss it here). We count the number j of bits set before position i in G_c.

We then call $select_c(j)$ on H to find the rank k (among all runs) of the c-run containing the i-th c. Finally, we call $G_{all}.select_1(k)$ to find the text position corresponding to the i-th c.

The structure takes $\mathcal{O}(r)$ words 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`, `insert` and `delete` (of 0-bits) operations in G_{all} and G_c.

All structures are implemented dynamically. For H we can use the result in [24], guaranteeing $\mathcal{O}(r \log n)$ bits of space. Note that in [24] there is an extra $\mathcal{O}(\sigma \log r)$ spatial term amounting, in our case, to $\mathcal{O}(r \log n)$ bits, since $\sigma \leq r \leq n$. This structure supports $\mathcal{O}(\log r)$-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, δ): 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 3.1) 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`), $B.\text{rank}(i) = \sum_{j=0}^{i} B[j]$, $B.\text{select}(i)$ (the position j such that $B[j] = 1$ and $B.\text{rank}(j) = i$), $B.\text{insert}(i, b)$ (insert bit $b \in \{0, 1\}$ between positions $i - 1$ and i), and $B.\text{delete}_0(i)$, 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. $B.\text{delete}_0(i)$ requires just a search and an update on PS. To support `insert` on B, we can operate as follows: $B.\text{insert}(i, 0)$, $i > 0$, is implemented with `PS.update(PS.search`$(i), 1)$. $B.\text{insert}(0, 1)$ is implemented with `PS.insert`(0) followed by `PS.update`$(0, 1)$. $B.\text{insert}(i, 1)$, $i > 0$, "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`$(j + 1, \delta + 1)$. We obtain:

▶ **Lemma 1.** *Let $S \in \Sigma^n$ and let r be the number of runs in S. The structure above described takes $\mathcal{O}(r \log n)$ bits of space and supports `rank`, `access`, and `insert` operations on S in $\mathcal{O}(\log r)$ time.*

Combining the BWT construction algorithm sketched in the Preliminaries section with the above data structure, we obtain:

▶ **Theorem 2.** *Let r be the number of runs in $BWT(\overleftarrow{S})$. We can build online $RLBWT(\overleftarrow{S})$ by reading T left-to-right in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r \log n)$ bits of space.*

3.1 The Searchable Partial Sums with Indels Problem

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 a red-black tree (RBT) in which we store integers $s_1, \ldots, s_m$ in the leaves. Internal nodes of the tree are used instead to store the number of nodes and partial sum of its subtrees. `sum` and `search` queries can then be 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 $\mathcal{O}(\log m)$ counters (subtree-size/partial sum) if RBT rotations are involved. All operations take $\mathcal{O}(\log m)$ time.

4 First Algorithm: SA Sampling Based on BWT Runs

In this section we describe our first algorithm. The main data structures we use are a dynamic RLBWT of the text $\overleftarrow{T}$ and σ sets storing the suffix array sampling. The algorithm works in two phases.

In the first phase, we read T from left to right, building $RLBWT(\overleftarrow{T})$ (see Theorem 2). This step employs the online BWT construction algorithm briefly illustrated in Section 2, 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. In our case, D will be designed to be also run-length compressed: we represent it with the data structure described in the previous section.

In the second phase, the algorithm scans T left to right once more, this time using the RLBWT 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 $T[i, \ldots, j]$, with i first position of the current LZ-phrase, occurs in $T[0, \ldots, j-1]$. 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, $T[\pi, \ldots, \pi + j - i - 1] = T[i, \ldots, j-1]$ and $\pi = \perp$ in case $i = j$). Note that the computation is performed on an index of the entire text (not just of $T[0, \ldots, j]$), thus we need to take special care to ensure that the occurrences of $T[i, \ldots, j]$ we find are indeed *previous* occurrences. Informally, we need an index of the entire text for the following reason. Our strategy will consist in maintaining this invariant: we keep track, for each BWT run, of the two most external suffix array samples (i.e. text positions) encountered while scanning the text left-to-right. Using an index for $T[0, \ldots, j]$ only, we do not know whether an equal-letter run a^k will later be split in two runs $a^{k'} c a^{k''}$ (with $k' + k'' = k$ and $a \neq c$). In such a case, we would have to sample the last and first a's of the two new runs $a^{k'}$ and $a^{k''}$, respectively, in order to preserve the validity of our invariant. Sampling (i.e. mapping an L-position on the text) is an expensive task as it requires navigating the BWT until a sample is found ($\mathcal{O}(n)$ backwards steps), so this strategy is not feasible. Notice that keeping an index for the entire text solves this problem as we already have access to all runs and therefore we know which L-positions, among the ones we have already visited, are the most external in their run.

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

4.1 Strategy and Correctness

From now on BWT stands for $BWT(\overleftarrow{T})$. As said above, our strategy will consist in keeping track, for each BWT run, of the two most external suffix array samples while scanning the text left-to-right. In this respect, when saying that we *sample the suffix array* we actually mean that we associate to some L-positions their corresponding text position (the sparse

suffix array is, instead, a sampling of F-positions). Moreover, since we enumerate positions in T-order (not $\overleftarrow{T}$-order), k-th L-position will correspond to sample $(n - SA[k])$ mod 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 L-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 $^{[}\langle j, k \rangle$, and *close*, denoted as $\langle j, k \rangle^{]}$. 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 $^{[}\langle j, k \rangle$ and $\langle j', k' \rangle^{]}$,
    b. If $k' < k$, then we insert in $\mathcal{B}_a$ the pairs $^{[}\langle j', k' \rangle$ and $\langle j, k \rangle^{]}$.

(C) If there exist $^{[}\langle j', k' \rangle, \langle j'', k'' \rangle^{]} \in \mathcal{B}_a$ such that $k', k'' \in [l, r]$:
    a. If $k < k' < k''$, then we remove $^{[}\langle j', k' \rangle$ from $\mathcal{B}_a$ and insert $^{[}\langle j, k \rangle$ in $\mathcal{B}_a$,
    b. If $k' < k'' < k$, then we remove $\langle j'', k'' \rangle^{]}$ from $\mathcal{B}_a$ and insert $\langle j, k \rangle^{]}$ 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 $^{[}\langle j', k' \rangle$ and a close $\langle j'', k'' \rangle^{]}$ 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 L-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 L-position $k_\#$. We now prove that, after processing $0, \ldots, j$, we can quickly locate at least one occurrence of any string that occurs in $T[0, \ldots, j]$. Intuitively, this property will allow us to locate LZ phrase boundaries and previous occurrences of LZ phrases.

▶ **Lemma 3.** *Let $a \in \Sigma$. If $0, \ldots, j$ have been processed and $[l, r]$ is the BWT interval associated with a string $\overleftarrow{V} \in \Sigma^m$, 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 $j' \le j$ and hence Va occurs in $T[0, \ldots, j]$.

($\Leftarrow$) Let $T[t, \ldots, t + m] = Va$, with $t \le j - m$. Consider the BWT a-run containing $T[t + m] = a$. One of the following cases holds 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 $X, Y \in \Sigma^*$, $c, d \ne a, e > 0$. 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) $BWT[l, \ldots, r] = a^e X b Y$, or (ii) $BWT[l, \ldots, r] = Y b X a^e$, where $X, Y \in \Sigma^*$, $e > 0$. The two cases are symmetric hence we discuss only (i). Consider all T-prefixes $T[0, \ldots, j'']$ satisfying the following properties:

1. $j'' \leq j$
2. Va is a suffix of $T[0, \ldots, j'']$, and
3. the lexicographic rank of $\overleftarrow{T}[0, \ldots, j'' - 1]$ among all $\overleftarrow{T}$-suffixes is $k'' \in [l, l + e - 1]$

Property 3 means that the $\overleftarrow{T}$-suffix $\overleftarrow{T}[0, \ldots, j'' - 1]$ is a prefix of the k''-th row of the $BWT(\overleftarrow{T})$ matrix. Note that the requirement $k'' \in [l, l + e - 1]$ implies that character $T[j'']$ appears inside the length-e a-run prefixing $BWT[l, \ldots, r] = a^e X b Y$. There exists at least one T-prefix satisfying the above properties: $T[0, \ldots, t + m]$ (by definition). Then, the rank k' of the lexicographically largest $\overleftarrow{T}$-suffix satisfying the above properties is such that $\langle j', k' \rangle \in \mathcal{B}_a$ for some $j' \leq j$. In other words: there exists a sampled BWT position inside $BWT[l, \ldots, l + e - 1]$; this position is the rightmost we have visited in its run $BWT[l, \ldots, l + e - 1]$ (note that *lexicographically largest* translates to *rightmost* on the BWT). 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. ◄

The intuition behind our algorithm is to search the LZ phrase prefix ending at the end of the current text prefix. If the phrase prefix occurs before, then we proceed to the next prefix. Otherwise, we output a new phrase. Note that prefixes of LZ phrases are not necessarily right-maximal. We therefore need a way to apply Lemma 3 also to non-right-maximal strings. We start by showing how to quickly detect right-maximality of a string.

▶ **Lemma 4.** *Let $[l, r]$ be the BWT range of a string $\overleftarrow{V} \in \Sigma^*$. In $\mathcal{O}(\log r)$ time we can check whether V is right maximal or not.*

Proof. It is easy to see that V is right-maximal if and only if $G_{all}[l + 1, \ldots, r]$ contains at least one bit set. This property can be checked with two `rank` operations on G_{all}. The claimed complexity follows from Lemma 1. ◄

We now show how we can drop the right-maximality requirement from Lemma 3.

▶ **Lemma 5.** *If $0, \ldots, j - 1$ have been processed (none if $j = 0$) and, for any $m > 0$, a string $W \in \Sigma^m$ occurs in $T[0, \ldots, j + m - 1]$ at positions $i_1, \ldots, i_t$, then we can locate one of such occurrences in $\mathcal{O}(m \log r)$ time.*

Proof. We prove the property by induction on $|W| = m > 0$. The idea is to process also positions $j, \ldots, j + m - 1$ while locating W prefixes. Let $W = Va$, $V \in \Sigma^{m-1}$, $a \in \Sigma$.

If $m = 1$, then $V = \epsilon$ (empty string). We process position j. Since T contains at least two distinct characters (a and $\#$), V is right-maximal. Therefore we can apply Lemma 3 to find an occurrence of $W = a$ in $T[0, \ldots, j]$.

If $m > 1$, then $|V| > 0$. First of all, we process also position $j + m - 1$. Two cases can occur. (i) V is not right-maximal. Then, V is always followed by a in T (since Va occurs in T by hypothesis). By inductive hypothesis we can locate (*before* processing position $j + m - 1$) 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 $T[0, \ldots, j + m - 1]$.

(ii) V is right-maximal. Note that by inductive hypothesis we already processed positions $j, \ldots, j + m - 2$ (and located an occurrence of $V[0, \ldots, |V| - 1]$). We apply Lemma 3 to find an occurrence of $W = Va$ in $T[0, \ldots, j + m - 1]$.

Note that we perform overall m steps. In each step, we execute a backward search query to extend the BWT interval of current W's prefix, check for its right-maximality, and query the red-black trees storing suffix array samples. Our claimed complexity follows from Lemmas 1 and 4.   ◀

Lemma 5 directly gives us an efficient algorithm to locate phrase boundaries and previous occurrences of phrases (and, therefore, compute the LZ77 factorization of T using a RLBWT data structure). Figures 1, 2, and 3 depict the three cases of the strategy (see next section for a more detailed description). In Figure 1 the phrase prefix is right-maximal but the letter that follows is not sampled on the BWT range (we output an LZ factor); in Figure 2 the phrase prefix is right-maximal and the letter that follows is sampled on the BWT range (we extend the current LZ factor); in Figure 3 the phrase prefix is not right-maximal (we extend the current LZ factor).

Figure 1 Case 1: we are trying to extend the phrase prefix 'G' with a 'G'. The range of 'G' spans more than 1 run ('G' is right-maximal) and there are no sampled 'G' in the range. It follows that 'GG' does not appear before in the text. Note that in this and in the following pictures, the text is already LZ77-factored (vertical bars) for clarity.

4.2   Pseudocode

Our complete procedure is reported as Algorithm 1. The pseudocode implements an iterative version of Lemma 5. 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

Figure 2 Case 2: we are trying to extend the phrase prefix 'A' with a 'G'. The range of 'A' spans more than 1 run ('A' is right-maximal) and there is a sampled 'G' in the range. It follows that 'AG' appears before in the text (at position $sample - length = 3 - 1 = 2$).
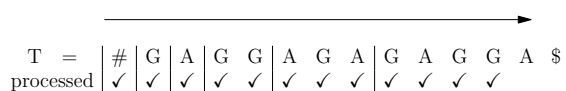
input text, which is read only once from left to right. Since the dynamic string we use is run-length compressed, this step requires $\mathcal{O}(r \log n)$ bits of working space.

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 ($\pi = \bot$ 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 at Line 10 scans T positions from the first to last. Note that we do not actually access the text T itself; rather, we extract T's characters from $RLBWT$. First of all, we have to discover if the current character $T[j] = c$ ends an 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 3, $T[j - \lambda, \ldots, j]$ 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 an 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 Lemma 5. If $u = 1$ (current phrase prefix is not right-maximal) then π is already the value we need. Otherwise (Lines

$$BWT(\overleftarrow{T}) \qquad \text{processed} \quad \text{sample}$$

	processed	sample
#$AGGAGAGAGGAG	✓	1
AG#$AGGAGAGAGG	✓	
AGAGAGGAG#$AGG	✓	
AGAGGAG#$AGGAG	✓	
AGGAG#$AGGAGAG	✓	6
AGGAGAGAGGAG#$		
G#$AGGAGAGAGGA	✓	2
GAG#$AGGAGAGAG	✓	4
GAGAGAGGAG#$AG	✓	11
GAGAGGAG#$AGGA	✓	9
GAGGAG#$AGGAGA	✓	
GGAG#$AGGAGAGA	✓	5
GGAGAGAGGAG#$A		
$AGGAGAGAGGAG#	✓	0

BWT range of 'GGAG' { GGAG#$AGGAGAGA, GGAGAGAGGAG#$A

$$T = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline \# & G & A & G & G & A & G & A & G & A & G & G & A & \$ \\ \hline \end{array}$$
processed: ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

Figure 3 Case 3: we are trying to extend the phrase prefix 'GAGG' with an 'A'. The range of 'GGAG' contains only one run ('GAGG' is not right-maximal). Then, all 'GAGG' are followed by 'A' in the text. Since we previously located 'GAGG' (inductive hypothesis) at position 1, it follows that also 'GAGGA' appears at position 1. Note that not all processed positions are marked with a SA sample (only the most external ones in each run).

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 an LZ phrase. By the inductive hypothesis of Lemma 5, $\pi < j - \lambda$ is either $\bot$—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 $\bot$, 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: rle_lz77_1(T)

---

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

1 $RLBWT \leftarrow build\_rev\_RLBWT(T)$;   /* Build online the RLBWT of $\overleftarrow{T}$ */
2 $n \leftarrow |T|$;   /* T length */
3 $j \leftarrow 0$;   /* Last position (on T) of current LZ phrase prefix */
4 $k \leftarrow k_\#$;   /* Position of # in RLBWT */
5 $\lambda \leftarrow 0$;   /* Length of current LZ phrase prefix */
6 $\pi \leftarrow \perp$;   /* Previous occurrence of current LZ phrase prefix */
7 $\mathcal{B}_{s_1}, \dots, \mathcal{B}_{s_\sigma} \leftarrow \emptyset$;   /* Initialize red-black trees of SA-samples */
8 $c \leftarrow RLBWT[k]$;   /* Current T character */
9 $[l, r] \leftarrow [0, n-1]$;   /* Range of current LZ phrase prefix in RLBWT */

10 **while** $j < n$ **do**

11 | $u \leftarrow RLBWT.number\_of\_runs(l, r)$;   /* Runs intersecting $[l, r]$ */

12 | **if** $u = 1$ **or** $\mathcal{B}_c.exists\_sample(l, r)$ **then**

13 | | **if** $u > 1$ **then**
14 | | | $\pi \leftarrow \mathcal{B}_c.locate(l, r) - \lambda$;   /* Occurrence of phrase prefix */

15 | | $\lambda \leftarrow \lambda + 1$;   /* Increase length of current LZ phrase */
16 | | $[l, r] \leftarrow RLBWT.LF([l, r], c)$;   /* Backward search step */
17 | **else**
18 | | **Output** $\langle \pi, \lambda, c \rangle$;   /* Output LZ77 factor */

19 | | $\lambda \leftarrow 0$;   /* Reset phrase prefix length */
20 | | $\pi \leftarrow \perp$;   /* Reset phrase prefix occurrence */
21 | | $[l, r] \leftarrow [0, n-1]$;   /* Reset range of current LZ phrase prefix */

22 | $[l_{run}, r_{run}] \leftarrow RLBWT.locate\_run(k)$;   /* run of BWT position k */
23 | $\mathcal{B}_c.update\_tree(\langle j, k \rangle, [l_{run}, r_{run}])$;   /* Apply update rules */

24 | $j \leftarrow j + 1$;   /* Increment T position */
25 | $k \leftarrow RLBWT.LF(k)$;   /* RLBWT position corresponding to j */
26 | $c \leftarrow RLBWT[k]$;   /* Read next T character */

4.3   Analysis

rank, access, and insert operations on RLBWT take $\mathcal{O}(\log r)$ time each. Operations $\mathcal{B}_c.exists\_sample(l, r)$ (Line 12) and $\mathcal{B}_c.locate(l, r)$ (Line 14) require a binary search on the red-black tree and can also be implemented in $\mathcal{O}(\log r)$ time. $RLBWT.number\_of\_runs(l, r)$ is the number of bits set in $G_{all}[l, \dots, r]$, plus 1 if $G_{all}[l] = 0$: this operation requires therefore $\mathcal{O}(1)$ rank/access operations on G_{all} ($\mathcal{O}(\log r)$ time). Similarly, $RLBWT.locate\_run(k)$ requires finding the two bits set preceding and following position k in G_{all} ($\mathcal{O}(\log r)$ time with a constant number of rank and select operations). Correctness of our algorithm follows easily from Lemma 5. We obtain:

▶ **Theorem 6.** *Algorithm 1 computes the LZ77 factorization of a text $T \in \Sigma^n$ in $\mathcal{O}(r \log n)$ bits of working space and $\mathcal{O}(n \log r)$ time, r being the number of runs in the Burrows-Wheeler*

transform of $\overleftarrow{T}$.

Note that in Algorithm 1 we can remove the step at Line 1 and take as input a RLBWT encoding of T, i.e. a series of pairs $\langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$, where λ_i is the length of the i-th c_i-run in $BWT(T)$. We can then easily turn—in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r \log n)$ bits of space—this representation into a run-length encoded string data structure with support for `access` and `rank`[1] and continue with the execution of Algorithm 1. We obtain the following result:

▶ **Theorem 7.** *We can convert the run-length BWT encoding* $\langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$ *of a text* $T \in \Sigma^n$ *into the LZ77 factorization of* $\overleftarrow{T}$ *in* $\mathcal{O}(r \log n)$ *bits of working space and* $\mathcal{O}(n \log r)$ *time, r being the number of runs in the Burrows-Wheeler transform of T.*

5 Second Algorithm: SA Sampling Based on LZ77 Factors

The algorithm presented in the previous section employs a RLBWT and additional data structures storing $2r$ suffix array samples. Even with a careful implementation of these components, the overall space is therefore lower bounded by roughly $3r$ words. The question we raise in this section is the following: can we reduce the impact of the constant involved in this lower bound? We propose a solution based on the following observation. As previous works [3, 32] suggested, in practice the size z of the LZ77 parsing is often (much) smaller than the number r of runs in the Burrows-Wheeler transform. From the practical point of view, it could be therefore more advantageous to employ a suffix array sampling based on LZ77 phrases rather than on BWT runs. The solution presented in this section employs a RLBWT and sparse suffix array sampled at the end of LZ phrases. The overall working space is lower-bounded by roughly $r + 2z$ words. In the case z is larger than r, we moreover show how we can compute z with a RLBWT data structure so that we can choose the most space-efficient strategy between the ones presented in this section and in the previous one.

We first give an overview of the algorithm, which is described more in detail in the next subsection. The algorithm works in three steps. In the first step, we build online $RLBWT(\overleftarrow{T})$ by reading T-characters from left to right and by inserting them in a run-length compressed dynamic string data structure (with the same algorithm used in the previous section). At the same time, we search in the RLBWT the current (reversed) LZ77 phrase prefix using backward search. While doing this, we mark BWT positions corresponding to sources of (reversed) LZ phrases with the corresponding phrase rank: while searching the j-th LZ phrase, as soon as the BWT interval for $\overleftarrow{Wc}$, $W \in \Sigma^\lambda$, $c \in \Sigma, \lambda > 0$ becomes empty, we mark one of the F-positions in the BWT interval for $\overleftarrow{W}$ with the integer j, being careful of choosing a position corresponding to a *previous* occurrence of W in the text (not the current one). Note that a F-position can be assigned more than one integer, so we need to maintain *sets* of integers on a *subset* of F-positions. This problem can be solved efficiently with a dynamic sparse bitvector marking with a bit set F-positions with at least one integer, with a dynamic succinct bitvector storing sets multiplicities in unary (i.e. a size-k set, $k > 0$, corresponds to the sequence of bits 10^{k-1} in this bitvector), and with a dynamic sequence of integers (for this last component we can use the SPSI described in section 3.1).

In the second step, we scan T from left-to-right by using the $RLBWT$ just built (i.e. by applying iteratively the LF function starting from F-position 0) and we use the integers stored in the previous step to locate the sources of LZ phrases. We store such sources in a

---

[1] e.g. by inserting the characters $c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_r^{\lambda_r}$ in the dynamic string structure described in the previous section

vector $SOURCES[0, \ldots, z - 1]$ initialized with $\perp$ (null) values: while reading text position i, if the position is associated with a set $\{j_1, ..., j_t\}$ of integers, we assign the value i to $SOURCES[j_1], \ldots, SOURCES[j_t]$. Note that i is the *last* position of the source, so in the next algorithm step we will need to subtract λ from it before outputting the LZ77 factor.

In the third and last step we delete all structures except $SOURCES$ and re-build $RLBWT$ by reading T left-to-right. As done in step 1, while building $RLBWT$ we search the current (reversed) LZ phrase. In this way, each time the BWT interval for $\overleftarrow{W}c$, $W \in \Sigma^\lambda$, $c \in \Sigma, \lambda \geq 0$ becomes empty, we output the LZ77 factor $\langle (SOURCES[j] - \lambda) + 1, \lambda, c \rangle$ (or $\langle \perp, 0, c \rangle$ if $\lambda = 0$), $j = 0, \ldots, z - 1$ being the rank of the current LZ phrase. Note that $SOURCES[j]$ is $\perp$ if and only if the j-th phrase is a single character.

5.1 Pseudocode

Algorithm 2 describes steps 1 (Lines 1-22) and 2 (Lines 23-29) sketched above. In Lines 1-5 we initialize the number z of LZ phrases (0 at the beginning: we will count them online), the RLBWT (as an empty run-length encoded string data structure), the BWT interval $[l, r]$ of the current LZ phrase prefix, the current LZ phrase prefix length λ, and the position k of the BWT terminator character $\#$ on the L column of the BWT. At the beginning, $k = \perp$ (undefined) as the BWT is empty.

We are going to read $T[i]$ for $i = 0, \ldots, n - 1$ and build $RLBWT(\overleftarrow{T})$ while computing phrase boundaries. At Lines 7-8 we perform a backward search step to extend the BWT interval $[l, r]$ with the current text character. Two cases can occur.

If the interval $[l', r']$ corresponding to $\overleftarrow{W}c$, $W \in \Sigma^\lambda$, $c = T[i] \in \Sigma, \lambda \geq 0$ is empty (Line 9), then Wc is an LZ77 phrase and—in the case $\lambda > 0$—we need to assign the current phrase rank to one of the sources of $\overleftarrow{W}$ on the RLBWT. If $\lambda = 0$, then the phrase is a single character and it has no source. Otherwise (Lines 11-14), we need to pick a position inside $[l, r]$ different than the *current* occurrence of W. Since the last character we inserted in the RLBWT is $W[|W| - 1]$, the current occurrence of W appears at the k-th BWT row, k being the position of $\#$ in the L-column. At Lines 12 and 14 we therefore insert the integer z, z being the current LZ phrase rank, in the set associated with either F-position l or r, depending on which one is different than k. In pseudocode 2 we denote the integer set associated with F-position k with $RLBWT.set\_at(k)$. Note that it must be the case that $r > l$ since W occurs at least twice in $T[0, \ldots, i - 1]$. We finally update $RLBWT(\overleftarrow{T[0, \ldots, i - 1]})$ to $RLBWT(\overleftarrow{T[0, \ldots, i]})$ with an extension step (Line 15) and, at Lines 16-18 we reset the BWT interval to the full interval, reset the phrase length λ to 0, and increase the number z of LZ phrases seen until now.

In the second case, the interval $[l', r']$ corresponding to $\overleftarrow{W}c$, $W \in \Sigma^\lambda$, $c = T[i] \in \Sigma, \lambda \geq 0$ is not empty (Line 19). Then, we simply increase the length of the current phrase prefix (Line 15) and extend the RLBWT with $T[i]$ (Line 21). The extension step at Line 21 returns the new position k of the $\#$ character on the L column of the BWT. Note that the (reverse of the) current occurrence of Wc falls inside $[l', r']$, so we need to update this interval by extending its right boundary by 1 (Line 22).

We can now scan the RLBWT and assign a source to each phrase. At Line 23 we initialize the $SOURCES$ vector with $\perp$ values. From here, k represents the F-position on the BWT corresponding to text position i (starting from $i = 0$). Since we start from $T[0] = \#$ and $\#$ appears at position 0 on the F-column, at Line 24 we initialize k to 0. For each $i = 0, \ldots, n-1$, we then check if F-position k is associated with a nonempty set $RLBWT.set\_at(k)$ of integers. If this is the case, for each such integer $j \in RLBWT.set\_at(k)$ at Line 27 we assign the

source i to the j-th LZ phrase. Synchronization between indexes i and k is guaranteed by the execution of the LF step at Line 28.

The complete procedure to compute the parse is reported as Algorithm 3. We do not discuss it in detail as it basically repeats the online construction of the RLBWT described above while computing LZ phrase boundaries. While doing this, at Line 10 we access the $SOURCES$ vector computed with procedure $find\_sources(T)$ and output LZ77 phrases in text order, being careful to subtract the phrase length from the content of $SOURCES$ since this vector contains the *last* position of each phrase source. To simplify the description, at this Line we use the convention that $(SOURCES[j] - \lambda) + 1 = \bot$ if $SOURCES[j] = \bot$.

5.2 Analysis

Building the RLBWT in the first and third steps and performing the n backward search steps takes overall $\mathcal{O}(n \log r)$ time (for the same reasons discussed in Section 4). We update the sets of integers once per phrase; we remind that such sets are encoded with a dynamic gap-encoded bitvector, a dynamic succinct bitvector, and a dynamic string. The total number of integers is z, so each update operation on the sets takes $\mathcal{O}(\log z)$ time with the structures described in Section 3 and the red-black tree implementing the dynamic string. Since $z \in \mathcal{O}(n/\log_\sigma n)$ [36], updating and querying the sets takes therefore $\mathcal{O}(z \log z) \subseteq \mathcal{O}(n \log \sigma) \subseteq \mathcal{O}(n \log r)$ time.

As discussed in Section 4, the RLBWT takes $\mathcal{O}(r \log n)$ bits of space. Each integer stored in the sets takes $\mathcal{O}(\log n)$ bits, so the algorithm uses overall $\mathcal{O}((r + z) \log n)$ bits of working space. In the case z is asymptotically larger than r, this strategy is less space-efficient than Algorithm 1. We can however choose—within $\mathcal{O}(r \log n)$ bits of working space—the most space-efficient strategy:

▶ **Lemma 8.** *The number z of LZ77 phrases of a text T can be computed with an online algorithm running in $\mathcal{O}(n \log r)$ time and using $\mathcal{O}(r \log n)$ bits of working space, r being the number of equal-letter runs in $BWT(\overleftarrow{T})$.*

Proof. Algorithm 3 without the instructions at Lines 5 and 10 solves exactly this problem: we just need to return the value z at the end of its execution. ◀

We can use Lemma 8 and compute z in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r \log n)$ bits of working space before computing the actual parse. If $z \leq r$, we execute Algorithm 3, otherwise Algorithm 1. Overall, this combined strategy runs therefore in $\mathcal{O}(n \log r)$ time and uses $\mathcal{O}(r \log n)$ bits of working space. We obtain:

▶ **Theorem 9.** *Our combined strategy computes the LZ77 factorization of a text $T \in \Sigma^n$ in $\mathcal{O}(r \log n)$ bits of working space and $\mathcal{O}(n \log r)$ time, r being the number of runs in the Burrows-Wheeler transform of $\overleftarrow{T}$.*

As a direct consequence of Theorems 6 and 9, we obtain asymptotically optimal-space construction algorithms for indexes based on LZ77 and RLBWT compressors such as the ones described in [3]. The main idea behind these 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 geometric range data structures, permits to efficiently count and locate pattern occurrences in T within $\mathcal{O}(r + z)$ words of space (see [3] 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 algorithms all these steps can be carried out in $\mathcal{O}(r + z)$ words of working space, which is asymptotically the same space of the resulting

---

Algorithm 2: find_sources(T)

input : A text $T \in \Sigma^n$ beginning with # and ending with $
output: A vector $SOURCES[0, \ldots, z-1]$, z being the size of the LZ77 parse, such that $SOURCES[j]$ is the last position of j-th phrase's source.

1 $z \leftarrow 0$;   /* Initialize size of the parse */
2 $RLBWT \leftarrow \epsilon$;   /* Initialize RLBWT to empty string */
3 $[l, r] \leftarrow [0, 0]$;   /* Initialize range on RLBWT */

4 $\lambda \leftarrow 0$;   /* Length of current LZ phrase */
5 $k \leftarrow \perp$;   /* Position of # in RLBWT (here $\perp$ because $RLBWT = \epsilon$) */

6 for $i = 0 \ldots |T| - 1$ **do**
7 | $c \leftarrow T[i]$;   /* read current text character */
8 | $[l', r'] \leftarrow RLBWT.LF([l, r], c)$;   /* backward search step */

9 | **if** $l' > r'$ **then**
10 | | **if** $\lambda > 0$ **then**
11 | | | **if** $k = l$ **then**
12 | | | | $RLBWT.set\_at(r).insert(z)$;
13 | | | **else**
14 | | | | $RLBWT.set\_at(l).insert(z)$;

15 | | $RLBWT.extend(c)$;   /* insert character c in the BWT */
16 | | $[l, r] \leftarrow [0, i]$;   /* reset $[l, r]$ to full interval */
17 | | $\lambda \leftarrow 0$;   /* reset phrase length */
18 | | $z \leftarrow z + 1$;   /* increase number of phrases */
19 | **else**
20 | | $\lambda \leftarrow \lambda + 1$;   /* increase current phrase length */
21 | | $k \leftarrow RLBWT.extend(c)$;   /* extend with c. Return position of # */
22 | | $[l, r] \leftarrow [l', r' + 1]$;   /* new suffix falls inside $[l', r']$: increment r' */

23 $SOURCES[0, \ldots, z-1] \leftarrow \langle \perp, \ldots, \perp \rangle$;   /* initialize SOURCES */
24 $k \leftarrow 0$;   /* position of # on F column */

25 for $i = 0 \ldots |T| - 1$ **do**
26 | **for each** $j \in RLBWT.set\_at(k)$ **do**
27 | | $SOURCES[j] \leftarrow i$;   /* assign source to the j-th phrase */

28 | $k \leftarrow RLBWT.LF(k)$;   /* LF step: navigate T forward */

29 return $SOURCES$;

---

index. To the best of our knowledge, 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 [33].

Algorithm 3: rle_lz77_2(T)

---

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

  1   $z \leftarrow 0$;   */* Initialize size of the parse */*
  2   $RLBWT \leftarrow \epsilon$;   */* Initialize RLBWT to empty string */*
  3   $[l, r] \leftarrow [0, 0]$;   */* Initialize range on RLBWT */*
  4   $\lambda \leftarrow 0$;   */* Length of current LZ phrase */*

  5   $SOURCES \leftarrow find\_sources(T)$;   */* locate phrase sources */*

  6   **for** $i = 0 \dots |T| - 1$ **do**
  7       $c \leftarrow T[i]$;   */* read current text character */*
  8       $[l', r'] \leftarrow RLBWT.LF([l, r], c)$;   */* backward search step */*
  9   **if** $l' > r'$ **then**
 10   **Output** $\langle (SOURCES[z] - \lambda) + 1, \lambda, c \rangle$;   */* Output LZ77 factor */*
 11           $RLBWT.extend(c)$;   */* insert character c in the BWT */*
 12           $[l, r] \leftarrow [0, i]$;   */* reset interval */*
 13           $\lambda \leftarrow 0$;   */* reset phrase length */*
 14           $z \leftarrow z + 1$;   */* increase number of phrases */*
 15   **else**
 16           $\lambda \leftarrow \lambda + 1$;   */* increase current phrase length */*
 17           $RLBWT.extend(c)$;   */* extend with c */*
 18           $[l, r] \leftarrow [l', r' + 1]$;   */* new suffix falls inside $[l', r']$: increment r' */*

---

6   Implementation and Experimental Results

We implemented the structures described in Section 3 and the two previously presented algorithms in the DYNAMIC [10] C++ library. The SPSI structure has been implemented using B-trees to improve cache efficiency (w.r.t. red-black trees). In order to reduce space usage while still guaranteeing very fast operations on integers stored on leaves, integers are packed contiguously in a word array and the same bit-size is used (i.e. the bit-size of the largest integer), for each leaf. Dynamic succinct bitvectors are implemented using an SPSI where all stored integers are either 0 or 1 (in this case we accelerate operations by using built-in bitwise operations such as `popcount`, masks and shifts), while dynamic strings are implemented with a Huffman-shaped wavelet tree built upon dynamic succinct bitvectors. Gap-encoded bitvectors and dynamic run-length encoded strings are implemented using SPSI structures and dynamic strings as described in Section 3. Instead of using red-black trees, the dynamic SA sampling of Algorithm 1 is stored using σ dynamic sparse vectors of integers, each implemented with a gap-encoded bitvector and a sequence of integers (we used an SPSI structure for this component). Sets of integers on RLBWT positions used in Algorithm 2 are implemented—as mentioned at the beginning of Section 5.2—with a gap-encoded bitvector, a succinct bitvector, and a sequence of integers (an SPSI).

File	Size (MB)	7-Zip-compressed size (MB)	Compression rate (%)
cere	440.0	8.10	1.84
para	410.0	9.80	2.39
influenzae	148.0	2.50	1.69
escherichia	108.0	7.10	6.57
sdsl	1024.0	0.60	0.06
samtools	1024.0	1.20	0.12
boost	1024.0	0.20	0.02
bwa	419.0	0.38	0.09
Einstein	1024.0	1.60	0.16
earth	1024.0	1.70	0.17
Bush	1024.0	1.90	0.19
wikipedia	1024.0	2.40	0.23

■ **Table 1** Size before and after 7-Zip-compression of the files. Last column is the rate (in percentage) between columns 3 and 2. Note that software repositories are extremely repetitive: in particular, the `boost` C++ library is compressed by over 5000 times with 7-Zip.

6.1   Experimental setup

We created two scripts that generate repetitive datasets by downloading all versions of Wikipedia web pages [35] and all revisions of GitHub repositories [16]. In addition, we downloaded four repetitive DNA datasets from the `pizza&chili` repetitive corpus [27]. To limit resources (computation time and RAM space), we truncated all files to 1GB (when bigger). The datasets are:

- `DNA` (from `pizza&chili` repetitive corpus):
  - `cere`: 37 sequences of *Saccharomyces Cerevisiae*
  - `para`: 36 sequences of *Saccharomyces Paradoxus*
  - `influenzae`: 78041 sequences of *Haemophilus Influenzae*
  - `escherichia`: 23 sequences of *Escherichia Coli*
- `Git repositories`. Concatenation of source files from the last revisions of:
  - `sdsl`. https://github.com/simongog/sdsl-lite
  - `samtools`. https://github.com/samtools/samtools
  - `boost`. https://github.com/boostorg/boost
  - `bwa`. https://github.com/lh3/bwa
- `Wikipedia`. Concatenation of all versions of:
  - `Einstein`. en.wikipedia.org/wiki/Albert_Einstein
  - `earth`. en.wikipedia.org/wiki/Earth
  - `Bush`. en.wikipedia.org/wiki/George_W._Bush
  - `wikipedia`. en.wikipedia.org/wiki/Wikipedia

Table 1 reports the sizes of the above files before and after compression with 7-Zip, followed by compression rate (size-after/size-before).

We tested implementations of six LZ77 factorization algorithms on the datasets (in order of decreasing space usage):

1. `ISA6r` [21, 18]. $\mathcal{O}(n \log n)$ space (in practice: $6n$ Bytes)
2. `KKP1s` [20, 18]. $\mathcal{O}(n \log n)$ space (in practice: $5n$ Bytes)

3. `LZscan` [19, 18]. $\mathcal{O}(n \log \sigma)$ space (in practice: $n + \mathcal{O}(n/d)$ Bytes, $d > 0$)
4. `h0-lz77` [29, 10]. $nH_0 + o(n \log \sigma)$ space
5. `rle-lz77-1` [10]. Our first algorithm. $\mathcal{O}(r \log n)$ space
6. `rle-lz77-2` [10]. Our second algorithm. $\mathcal{O}(r \log n)$ space

As far as `LZscan` is concerned, we chose d in such a way that the term $\mathcal{O}(n/d)$ was always around 50% text's size (the tool requires n/d to be an integer number of MB). We included `ISA6r` as it is specialized for repetitive inputs. We moreover note that `KKP1s` is a semi-external algorithm (i.e., streams the suffix array from disk using a small buffer), while all other algorithms use only internal memory. We ran all experiments on an `intel core i7` machine with 12 GB of RAM running Linux Ubuntu 15.10.

6.2 Results

Figure 4 reports the results of the experiments, with solid and dotted horizontal lines marking the sizes of the plain input files and the 7-Zip-compressed files, respectively. For a more precise comparison, in Table 2 we report detailed working space and running times values of all tools on three representative datasets: `sdsl`, `cere`, and `einstein`. As expected, the linear-space `LZscan` algorithm and the zero-order compressed-space `h0-lz77` algorithm always use space close to the plain file size, with `h0-lz77` always slightly below and `LZscan` slightly above the solid lines. `ISA6r` and `KKP1s` exhibit very similar performances: these algorithms are the fastest but use one order of magnitude more space than the plain file size. In all cases but `para` and `sdsl`, the algorithm `ISA6r` was slightly faster than `KKP1s`. This behavior is probably due to the fact that `ISA6r` is specialized for highly repetitive inputs, on which it should be faster than `KKP1s`. `rle-lz77-2` always dominates `rle-lz77-1`, suggesting that the SA sampling based on LZ77 factors in practice is much more effective than the one based on BWT runs. Our two algorithms use approximately one order of magnitude more space than the 7-Zip-compressed file size, and in almost all cases from 2 to 3 orders of magnitude less space than all other methods. The only exceptions occur in correspondence of the DNA datasets, which are much less repetitive than the others. In such cases our algorithms use a working space comparable to (in one case higher than) the uncompressed file size. Table 2 shows that `rle-lz77-2`'s working space is approximately 60% of `rle-lz77-1`'s working space on very compressible datasets (`sdsl` and `einstein`). On the less repetitive dataset `cere`, this fraction drops to 31%. It is worth to note that, on very compressible datasets, `rle-lz77-2` uses a working space close to only 1% of the dataset size. This space efficiency is not paid in terms of running times: `rle-lz77-2` requires approximately 75% of `rle-lz77-2`'s running time to terminate. As expected, the algorithms making use of complex dynamic data structures (`h0-lz77`, `rle-lz77-1`, and `rle-lz77-2`) are much slower than the others (from 1 to 3 orders of magnitude).

7   Conclusions and Future Work

In this work we proved that LZ-based text compression and indexing can be carried out in a working space proportional to the size of the run-length compressed Burrows-Wheeler transform of the text. We believe our results are both of theoretical as well as of practical interest. We achieve the first algorithms that compute the exact LZ77 parse in a space that can turn out (up to) exponentially smaller than that of the input text, if this is highly compressible. Moreover, we showed that also in practice our techniques use only one order of magnitude more space than the 7-Zip-compressed file size; on repetitive inputs, this space
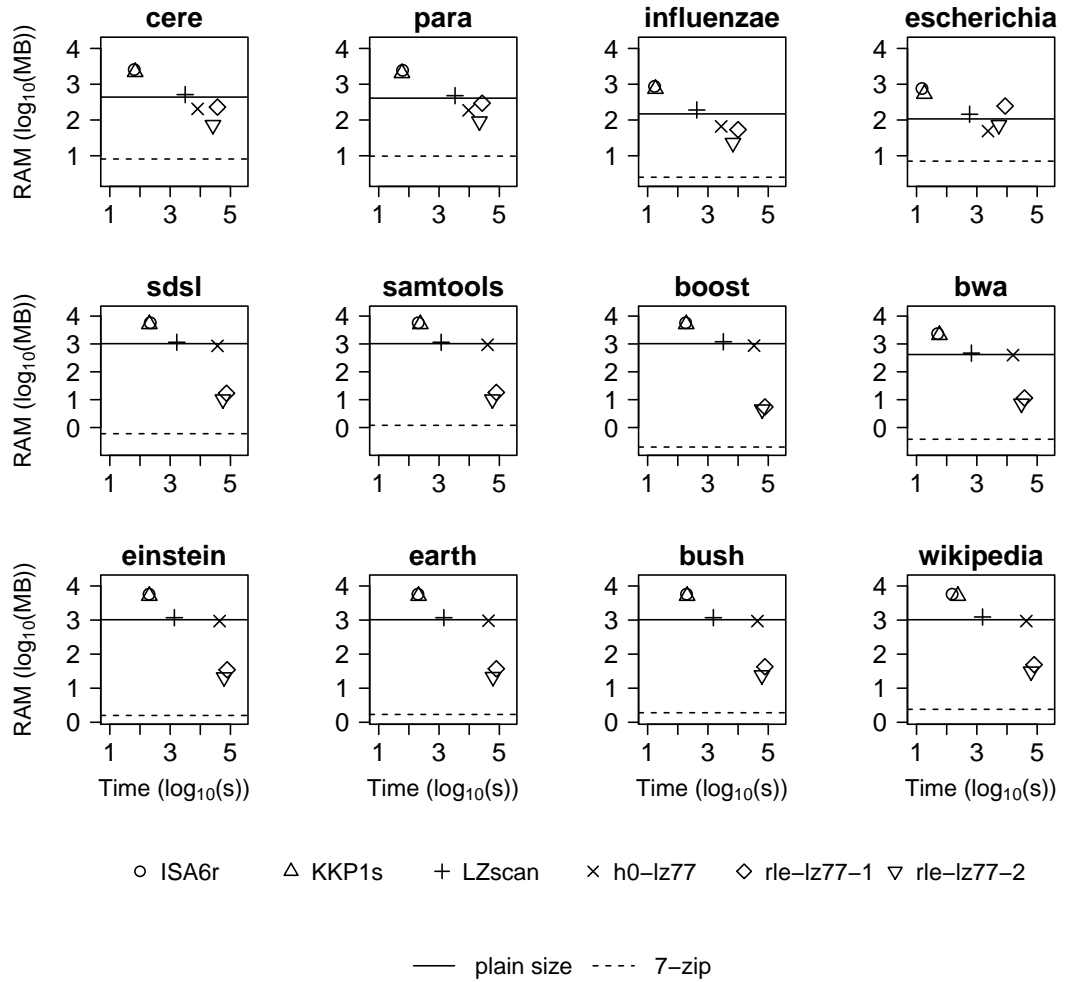
Figure 4 Running times and RAM working space (logarithmic scales) of the six tested tools on the twelve datasets. Solid and dotted horizontal lines mark the sizes of the plain and 7-Zip-compressed files, respectively.

tool	data	dataset size (MB)	WS (MB)	WS (%)	time (s)
`ISA6r`	`sdsl`	1024	5770.58	563.53	218
`KKP1s`	`sdsl`	1024	5121.76	500.17	205
`LZscan`	`sdsl`	1024	1153.45	112.64	1648
`h0-lz77`	`sdsl`	1024	855.97	83.59	36805
`rle-lz77-1`	`sdsl`	1024	16.85	1.65	74398
`rle-lz77-2`	`sdsl`	1024	10.26	1.00	55635
`ISA6r`	`cere`	440	2594.15	589.58	65
`KKP1s`	`cere`	440	2201.33	500.30	69
`LZscan`	`cere`	440	516.65	117.42	3146
`h0-lz77`	`cere`	440	203.52	46.25	8056
`rle-lz77-1`	`cere`	440	227.03	51.60	37007
`rle-lz77-2`	`cere`	440	73.15	16.63	26071
`ISA6r`	`einstein`	1024	5782.76	564.72	199
`KKP1s`	`einstein`	1024	5121.76	500.17	206
`LZscan`	`einstein`	1024	1182.20	115.45	1391
`h0-lz77`	`einstein`	1024	924.92	90.32	43495
`rle-lz77-1`	`einstein`	1024	34.45	3.36	77048
`rle-lz77-2`	`einstein`	1024	20.66	2.02	61267

Table 2 Detailed working space (WS: both absolute and relative w.r.t. original dataset size) and running times of all tools on three representative datasets.

is several orders of magnitude smaller than the size of the text and of data structures—e.g. suffix arrays—employed in other algorithms described in literature. The only practical weak point of our strategies is the use of complex dynamic data structures such as dynamic run-length encoded strings. Despite proved to have update-times nearly-optimal in theory, these structures are very slow in practice. One solution could be to use parallelization to speed up operations on these components. Alternatively, one could use faster algorithms to build the RLBWT (e.g. the incremental algorithm of [31]).

We note that the suffix-array sampling based on BWT runs allows to find *at least one* previous occurrence of the factors, but not all occurrences of an arbitrary string in the text. It follows that this technique cannot be directly used in a full-text index. We leave open the question whether this sampling can be used in a compressed index to support locate (e.g. by augmenting it with additional structures); such a result would lead to the first compressed text index taking $\mathcal{O}(r)$ words of space.

Our results find important applications in the space-efficient construction of compressed text indexes, potentially requiring exponentially less space than standardly employed algorithms. Another promising line of research explored in our work is that of converting between compressed representations of a text. The topic of *compressed computation* is becoming of increasing importance in many fields—first of all bioinformatics and web databases—as it is not always feasible to decompress data before manipulating it. We showed how our results can be used to compute LZ77 of the reversed text from a RLBWT-based compressed text representation in a space proportional to its input size. Other published works in this direction include [30] (LZ77 to grammar compression), [1, 2] (grammar compression to LZ78), and [34] (LZ78 to run-length encoding of T and back). We leave to future works

the problem of computing the LZ77 factorization of T (instead of $\overleftarrow{T}$) from the RLBWT of T and the opposite direction (LZ77 to RLBWT). The opposite direction can be easily obtained by extracting text from LZ77 and building online a RLBWT structure. The problem with this approach is that text extraction from LZ77 compressed text representations is a computationally expensive task as it requires to follow chains of character's copies. In the worst case, the height h of the LZ77 parse can be $\mathcal{O}(n)$, which leads to a quadratic-time solution. We leave open the problem whether this can be done more efficiently—e.g. in $\mathcal{O}(n \log n)$ time.

The implementation of the two algorithms described in this paper led to the creation of a C++ library, DYNAMIC [10], collecting several compressed dynamic data structures. To date, several excellent libraries such as sdsl [17] offer efficient implementations of *static* compressed data structures; however, few code can be found on the dynamic side. Our library has been written in modern C++11 standard and features dynamic partial sums, succinct and gap-encoded bitvectors, Huffman and run-length compressed strings and FM indexes. DYNAMIC has been heavily profiled in order to get the best space/time trade-offs and we believe it will be useful also in other works making use of dynamic compressed data structures.

To conclude, we provide two scripts [16, 35] that can be used to generate very repetitive datasets by downloading all versions of a Wikipedia web page and a GitHub repository. The scripts are easy to use and can generate heavy datasets (up to several GB) compressible by thousands of times with techniques such as the Lempel-Ziv factorization.

References

1   Hideo Bannai, Paweł Gawrychowski, Shunsuke Inenaga, and Masayuki Takeda. Converting SLP to LZ78 in almost Linear Time. In *Combinatorial Pattern Matching*, pages 38–49. Springer, 2013.

2   Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 factorization of grammar compressed text. In *String Processing and Information Retrieval*, pages 86–98. Springer, 2012.

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

4   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.

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

6   M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

7   Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *Information Theory, IEEE Transactions on*, 51(7):2554–2576, 2005.

8   Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

9   Maxime Crochemore and Lucian Ilie. Computing Longest Previous Factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

10   DYNAMIC: dynamic succinct/compressed data structures library. `https://github.com/nicolaprezza/DYNAMIC`. Accessed: 2016-05-20.

11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

12 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *String Processing and Information Retrieval*, pages 150–160. Springer, 2004.

13 Gabriele Fici. Factorizations of the fibonacci infinite word. *Journal of Integer Sequences*, 18(2):3, 2015.

14 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Algorithms-ESA 2015*, pages 533–544. Springer, 2015.

15 Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

16 get-git-revisions: Get all revisions of a git repository. `https://github.com/nicolaprezza/get-git-revisions`. Accessed: 2016-05-20.

17 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

18 LZ77 factorization algorithms. `https://www.cs.helsinki.fi/group/pads/lz77.html`. Accessed: 2016-05-20.

19 Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms*, pages 139–150. Springer, 2013.

20 Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*, pages 189–200. Springer, 2013.

21 Dominik Kempa and Simon J Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the Meeting on Algorithm Engineering & Experriments*, pages 103–112. Society for Industrial and Applied Mathematics, 2013.

22 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

23 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows–wheeler transform and sturmian words. *Information Processing Letters*, 86(5):241–246, 2003.

24 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

25 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic Index and LZ Factorization in Compressed Space. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2016*, pages 158–170, Czech Technical University in Prague, Czech Republic, 2016.

26 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching*, pages 15–26. Springer, 2011.

27 pizza&chili repetitive corpus. `http://pizzachili.dcc.uchile.cl/repcorpus/real/`. Accessed: 2016-05-20.

28 Alberto Policriti, Nicola Gigante, and Nicola Prezza. Average linear time and compressed space construction of the Burrows-Wheeler transform. In *Language and Automata Theory and Applications*, volume 8977 of *Lecture Notes in Computer Science*, pages 587–598. Springer International Publishing, 2015.

29 Alberto Policriti and Nicola Prezza. Fast Online Lempel-Ziv Factorization in Compressed Space. In *String Processing and Information Retrieval*, volume 9309 of *Lecture Notes in Computer Science*, pages 13–20. Springer International Publishing, 2015. `doi:10.1007/978-3-319-23826-5_2`.

30 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.

31 Jouni Sirén et al. *Compressed full-text indexes for highly repetitive collections*. PhD thesis, Helsingin yliopisto, 2012.

32 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*, pages 164–175. Springer, 2009.

33 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online self-indexed grammar compression. In *proc. SPIRE*, volume 9309 of *Lecture Notes in Computer Science*, pages 258–269. Springer International Publishing, 2015.

34 Yuya Tamakoshi, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masanori Takeda. From run length encoding to LZ78 and back again. In *Data Compression Conference (DCC), 2013*, pages 143–152. IEEE, 2013.

35 wiki-get: Download all versions of a Wikipedia page. `https://github.com/nicolaprezza/wiki_get`. Accessed: 2016-05-20.

36 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.