LZ77 Computation Based on the Run-Length Encoded BWT

From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back

Alberto Policriti[1,2] and Nicola Prezza[3⋆]

[1] University of Udine, Department of Informatics, Mathematics, and Physics, Italy
[2] Institute of Applied Genomics, Udine, Italy
[3] Technical University of Denmark, DTU Compute

Abstract. The Lempel-Ziv factorization (LZ77) and the Run-Length encoded Burrows-Wheeler Transform (RLBWT) are two important tools in text compression and indexing, being their sizes z and r closely related to the amount of text self-repetitiveness. In this paper we consider the problem of converting the two representations into each other within a working space proportional to the input and the output. Let n be the text length. We show that $RLBWT$ can be converted to $LZ77$ in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space. Conversely, we provide an algorithm to convert $LZ77$ to $RLBWT$ in $\mathcal{O}\big(n(\log r + \log z)\big)$ time and $\mathcal{O}(r + z)$ words of working space. Note that r and z can be *constant* if the text is highly repetitive, and our algorithms can operate with (up to) *exponentially* less space than naive solutions based on full decompression.

1 Introduction

The field of *compressed computation*—i.e. computation on compressed representations of the data without first fully decompressing it—is lately receiving much attention due to the ever-growing rate at which data is accumulating in archives such as the web or genomic databases. Being able to operate directly on the compressed data can make an enormous difference, considering that repetitive collections, such as sets of same-species genomes or software repositories, can be compressed at rates that often exceed 1000x. In such cases, this set of techniques makes it possible to perform most of the computation directly in primary memory and enables the possibility of manipulating huge datasets even on resource-limited machines.

Central in the field of compressed computation are *compressed data structures* such as compressed full-text indexes, geometry (e.g. 2D range search), trees, graphs. The compression of these structures (in particular those designed for unstructured data) is based on an array of techniques which include entropy compression, Lempel-Ziv parsings [1, 2] (LZ77/LZ78), grammar compression [3], and the Burrows-Wheeler transform [4] (BWT). Grammar compression, Run-Length encoding of the BWT [5, 6] (RLBWT), and LZ77 have been shown superior in the task of compressing highly-repetitive data and, as a consequence, much research is lately focusing on these three techniques.

In this paper we address a central point in compressed computation: can we convert between different compressed representations of a text while using an amount of working space proportional to the input/output? Being able to perform such task would, for

⋆ Part of this work was done while the author was a PhD student at the University of Udine, Italy. Work supported by the Danish Research Council (DFF-4005-00267)

instance, open the possibility of converting between compressed data structures (e.g. self-indexes) based on different compressors, all within compressed working space.

It is not the fist time that this problem has been addressed. In [7] the author shows how to convert the LZ77 encoding of a text into a grammar-based encoding, while in [8,9] the opposite direction (though pointing to LZ78 instead of LZ77) is considered. In [10] the authors consider the conversions between LZ78 and run-length encoding of the text. Note that LZ77 and run-length encoding of the BWT are much more powerful than LZ78 and run-length encoding of the text, respectively, so methods addressing conversion between LZ77 and RLBWT would be of much higher interest. In this work we show how to efficiently solve this problem in space proportional to the sizes of these two compressed representations. See the Definitions section for a formal definition of $RLBWT(T)$ and $LZ77(T)$ as a list of r pairs and z triples, respectively. Let $RLBWT(T) \rightarrow LZ77(T)$ denote the computation of the list $LZ77(T)$ using as input the list $RLBWT(T)$ (analogously for the opposite direction). The following results are illustrated below:

(1) We can compute $RLBWT(T) \rightarrow LZ77(T)$ in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space
(2) We can compute $LZ77(T) \rightarrow RLBWT(T)$ in $\mathcal{O}\big(n(\log r + \log z)\big)$ time and $\mathcal{O}(r + z)$ words of working space

Result (1) is based on our own recent work [11] and requires space proportional to the input *only* as output is streamed to disk. Result (2) requires space proportional to the input *plus* the output, since data structures based on both compressors are used in main memory. In order to achieve result (2), we show how we can (locally) decompress $LZ77(T)$ while incrementally building a run-length BWT data structure of the reversed text. Extracting text from LZ77 is a computationally expensive task, as it requires a time proportional to the parse height h per extracted character [12] (with h as large as n, in the worst case). The key ingredient of our solution is to use the run-length BWT data structure itself to efficiently extract text from $LZ77(T)$.

2 Basics

Since we work with both LZ77 [1] and the Burrows-Wheeler transform [4] (see below for definitions), we assume that our text T contains both *LZ* and *BWT terminator* characters. More specifically, let T be of the form $T = \#T'\$ \in \Sigma^n$, with $T' \in (\Sigma \setminus \{\$, \#\})^{n-2}$, where \$ is the LZ77-terminator, and #—lexicographically smaller than all elements in Σ—is the BWT-terminator. Note that adding the two terminator characters to our text increases only by two the number of LZ77 factors and by at most four the number of BWT runs.

The *Burrows-Wheeler Transform* [4] $BWT(T)$ is a permutation of T defined as follows. Sort all cyclic permutations of T in a *conceptual* matrix $M \in \Sigma^{n \times n}$. $BWT(T)$ is the last column of M. With F and L we will denote the first and last column of M, respectively, and we will say *F-positions* and *L-positions* to refer to positions on these two columns. On compressible texts, $BWT(T)$ exhibits some remarkable properties that

permit to boost compression. In particular, it can be shown [6] that repetitions in T generate equal-letter runs in $BWT(T)$. We can efficiently represent this transform as the list of pairs

$$RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r_T}$$

where $\lambda_i > 0$ is the length of the *maximal* i-th c_i-run, $c_i \in \Sigma$. Equivalently, $RLBWT(T)$ is the *shortest* list of pairs $\langle \lambda_i, c_i \rangle_{i=1,\ldots,r_T}$ satisfying $BWT(T) = c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_{r_T}^{\lambda_{r_T}}$. Let $\overleftarrow{T}$ be the reverse of T. To simplify notation we define $r = \max\{r_T, r_{\overleftarrow{T}}\}$ (in practical cases $r_T \approx r_{\overleftarrow{T}}$ holds [13], and this definition simplifies notation).

With $RLBWT^+(T)$ we denote a run-length encoded BWT *data structure* on the text T, taking $\mathcal{O}(r)$ words of space and supporting `insert`, `rank`, `select`, and `access` operation on the BWT. Using these operations, functions LF and FL (mapping L-positions to F-positions and *vice versa*) and function `extend` (turning $RLBWT^+(T)$ into $RLBWT^+(aT)$ for some $a \in \Sigma$) can be supported in $\mathcal{O}(\log r)$ time. We leave to the next sections details concerning the particular implementation of this data structure.

We recall that $BWT(\overleftarrow{T})$ can be built online with an algorithm that reads T-characters left-to-right and inserts them in a dynamic string data structure [14, 15]. Briefly, letting $a \in \Sigma$, the algorithm is based on the idea of backward-searching the extended reversed text $\overleftarrow{Ta}$ in the BWT index for $\overleftarrow{T}$. This operation leads to the F-position l where $\overleftarrow{Ta}$ should appear among all sorted $\overleftarrow{T}$'s suffixes. At this point, it is sufficient to insert # at position l in $BWT(\overleftarrow{T})$ and replace the old # with a to obtain $BWT(\overleftarrow{Ta})$.

The *LZ77 parsing* [1] (or *factorization*) of a text T is the sequence of *phrases* (or *factors*)

$$LZ77(T) = \langle \pi_i, \lambda_i, c_i \rangle_{i=1,\ldots,z}$$

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.
2. For any $i = 1, \ldots, z$, the string ω_i is the *longest* occurring at least twice in $\omega_1 c_1 \ldots \omega_i$.

3 From RLBWT to LZ77

Our algorithm to compute $RLBWT(T) \rightarrow LZ77(T)$ is based on the result [11]: an algorithm to compute—in $\mathcal{O}(r)$ words of working space and $\mathcal{O}(n \log r)$ time—$LZ77(T)$ using T as input. The data structure at the core of this result is a dynamic run-length compressed string:

Theorem 1. *[11, 16] Let $S \in \Sigma^n$ and let $\bar{r}$ be the number of equal-letter runs in S. There exists a data structure taking $\mathcal{O}(\bar{r})$ words of space and supporting* `rank`*,* `select`*,* `access`*, and* `insert` *operations on S in $\mathcal{O}(\log \bar{r})$ time.*

The algorithm works in two steps, during the first of which builds $RLBWT^+(\overleftarrow{T})$ by inserting left-to-right T-characters in a *dynamic* $RLBWT$ represented with the data structure of Theorem 1—using the procedure sketched in the previous section. In the

second step, the procedure scans T once more left-to-right while searching (reversed) LZ77 phrases in $RLBWT^+(\overleftarrow{T})$. At the same time, a dynamic suffix array sampling is created by storing, for each BWT equal-letter run, the two most external (i.e. leftmost and rightmost in the run) text positions seen up to the current position; the key property proved in [11] is that this sparse suffix array sampling is sufficient to locate LZ77 phrase boundaries and sources. LZ77 phrases are outputted in text order, therefore they can be directly streamed to output. The total size of the suffix array sampling never exceeds $2r$. From Theorem 1, all operations (*insert*, *LF-mapping*, *access*) are supported in $\mathcal{O}(\log r)$ time and the structure takes $\mathcal{O}(r)$ words of space. The claimed space/time bounds of the algorithm easily follow.

Note that, using the algorithm described in [11], we can only perform the conversion $RLBWT^+(\overleftarrow{T}) \rightarrow LZ77(T)$. Our full procedure to achieve conversion $RLBWT(T) \rightarrow LZ77(T)$ consists of the following three steps:

1. convert $RLBWT(T)$ to $RLBWT^+(T)$, i.e. we add support for `rank`/`select`/`access` queries on $RLBWT(T)$;
2. compute $RLBWT^+(\overleftarrow{T})$ using $RLBWT^+(T)$;
3. run the algorithm described in [11] and compute $LZ77(T)$ using $RLBWT^+(\overleftarrow{T})$.

Let $RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$ (see the previous section). Step 1 can be performed by just inserting characters $c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_r^{\lambda_r}$ (in this order) in the dynamic run-length encoded string data structure of Theorem 1. Step 2 is performed by extracting characters $T[0], T[1], \ldots, T[n-1]$ from $RLBWT^+(T)$ and inserting them (in this order) in a dynamic $RLBWT$ data structure with the BWT construction algorithm sketched in the Section (2). Since this algorithm builds the $RLBWT$ of the *reversed* text, the final result is $RLBWT^+(\overleftarrow{T})$. We can state our first result:

Theorem 2. *Conversion $RLBWT(T) \rightarrow LZ77(T)$ can be performed in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space.*

Proof. We use the dynamic RLBWT structure of Theorem 1 to implement components $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. Step 1 requires n `insert` operations in $RLBWT^+(T)$, and terminates therefore in $\mathcal{O}(n \log r)$ time. Since the string we are building contains r_T runs, this step uses $\mathcal{O}(r)$ words of working space. Step 2 calls n `extend` and `FL` queries on dynamic RLBWTs. `extend` requires a constant number of `rank` and `insert` operations [15]. FL function requires just an `access` and a `rank` on the F column and a `select` on the L column. From Theorem 1, all these operations are supported in $\mathcal{O}(\log r)$ time, so also step 2 terminates in $\mathcal{O}(n \log r)$ time. Recall that r is defined to be the maximum between the number of runs in $BWT(T)$ and $BWT(\overleftarrow{T})$. Since in this step we are building $RLBWT^+(\overleftarrow{T})$ using $RLBWT^+(T)$, the overall space is bounded by $\mathcal{O}(r)$ words. Finally, step 3 terminates in $\mathcal{O}(n \log r)$ time while using $\mathcal{O}(r)$ words of space [11]. The claimed bounds for our algorithm to compute $RLBWT(T) \rightarrow LZ77(T)$ follow.

4 From LZ77 to RLBWT

Our strategy to convert $LZ77(T)$ to $RLBWT(T)$ consists of the following steps:

1. extract $T[0], T[1], \ldots, T[n-1]$ from $LZ77(T)$ and build $RLBWT^+(\overleftarrow{T})$;
2. convert $RLBWT^+(\overleftarrow{T})$ to $RLBWT^+(T)$;
3. extract equal-letter runs from $RLBWT^+(T)$ and stream $RLBWT(T)$ to the output.

Step 2 is analogous to step 2 discussed in the previous section. Step 3 requires reading characters $RLBWT^+(T)[0]$, ..., $RLBWT^+(T)[n-1]$ (`access` queries on $RLBWT^+(T)$) and keeping in memory a character storing last run's head and a counter keeping track of last run's length. Whenever we open a new run, we stream last run's head and length to the output.

The problematic step is the first. As mentioned in the introduction, extracting a character from $LZ77(T)$ requires to follow a chain of character copies. In the worst case, the length h of this chain—also called the parse height (see [12] for a formal definition)—can be as large as n. Our observation is that, since we are building $RLBWT^+(\overleftarrow{T})$, we can use this component to efficiently extract text from $LZ77(T)$: while decoding factor $\langle \pi_v, \lambda_v, c_v \rangle$, we convert π_v to a position on the RLBWT and extract λ_v characters from it. The main challenge in efficiently achieving this goal is to convert text positions to RLBWT positions (taking into account that the RLBWT is dynamic and therefore changes in size and content).

4.1 Dynamic functions

Considering that $RLBWT^+(\overleftarrow{T})$ is built incrementally, we need a data structure to encode a function $\mathcal{Z} : \{\pi_1, ..., \pi_z\} \to \{0, ..., n-1\}$ mapping those text positions that are the source of some LZ77 phrase to their corresponding $RLBWT$ positions. Moreover, the data structure must be *dynamic*, that is it must support the following three operations (see below the list for a description of how these operations will be used):

- `map`: $\mathcal{Z}(i)$. Compute the image of i
- `expand`: $\mathcal{Z}.expand(j)$. Set $\mathcal{Z}(i)$ to $\mathcal{Z}(i) + 1$ for every i such that $\mathcal{Z}(i) \geq j$
- `assign`: $\mathcal{Z}(i) \leftarrow j$. Call $\mathcal{Z}.expand(j)$ and set $\mathcal{Z}(i)$ to j

To keep the notation simple and light, we use the same symbol $\mathcal{Z}$ for the function as well as for the data structure representing it. We say that $\mathcal{Z}(i)$ is *defined* if, for some j, we executed an `assign` operation $\mathcal{Z}(i) \leftarrow j$ at some previous stage of the computation. For technical reasons that will be clear later, we restrict our attention to the case where we execute `assign` operations $\mathcal{Z}(i) \leftarrow j$ for increasing values of i, i.e. if $\mathcal{Z}(i_1) \leftarrow j_1, \ldots, \mathcal{Z}(i_q) \leftarrow j_q$ is the sequence (in temporal order) of the calls to `assign` on $\mathcal{Z}$, then $i_1 < \cdots < i_q$. This case will be sufficient in our case and, in particular, $i_1, \ldots, i_q$ will be the sorted non-null phrases sources $\pi_1, \ldots, \pi_z$. Finally, we assume that $\mathcal{Z}(i)$ is always called when $\mathcal{Z}(i)$ has already been defined—again, this will be the case in our algorithm.

Intuitively, $\mathcal{Z}.expand(j)$ will be used when we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and j is not associated with any phrase source (i.e. $i \neq \pi_v$ for all $v = 1, \ldots, z$). When we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and $i = \pi_v$ for some $v = 1, \ldots, z$ (possibly more than one), $\mathcal{Z}(i) \leftarrow j$ will be used.

The existence and associated query-costs of the data structure $\mathcal{Z}$ are proved in the following lemma.

Lemma 1. *Letting z be the number of phrases in the LZ77 parsing of T, there exists a data structure taking $\mathcal{O}(z)$ words of space and supporting* `map`*,* `expand`*, and* `assign` *operations on $\mathcal{Z} : \{\pi_1, ..., \pi_z\} \to \{0, ..., n-1\}$ in $\mathcal{O}(\log z)$ time*

Proof. First of all notice that, since $LZ77(T)$ is our input, we know beforehand the domain $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c \rangle \in LZ77(T) \ \wedge \pi \neq \perp\}$ of $\mathcal{Z}$. We can therefore map the domain to rank space and restrict our attention to functions $\mathcal{Z}' : \{0, ..., d-1\} \to \{0, ..., n-1\}$, with $d = |\mathcal{D}| \leq z$. To compute $\mathcal{Z}(i)$ we map $0 \leq i < n$ to a rank $0 \leq i' < d$ by binary-searching a precomputed array containing the sorted values of $\mathcal{D}$ and return $\mathcal{Z}'(i')$. Similarly, $\mathcal{Z}(i) \leftarrow j$ is implemented by executing $\mathcal{Z}'(i') \leftarrow j$ (with i' defined as above), and $\mathcal{Z}.expand(j)$ simply as $\mathcal{Z}'.expand(j)$.

We use a dynamic gap-encoded bitvector C marking (by setting a bit) those positions j such that $j = \mathcal{Z}(i)$ for some i. A dynamic gap-encoded bitvector with b bits set can easily be implemented using a red-black tree such that it takes $\mathcal{O}(b)$ words of space and supports `insert`, `rank`, `select`, and `access` operations in $\mathcal{O}(\log b)$ time; see [11] for such a reduction. Upon initialization of $\mathcal{Z}$, C is empty. Let k be the number of bits set in C at some step of the computation. We can furthermore restrict our attention to *surjective* functions $\mathcal{Z}'' : \{0, ..., d-1\} \to \{0, ..., k-1\}$ as follows. $\mathcal{Z}'(i')$ (`map`) returns $C.select_1(\mathcal{Z}''(i'))$. The `assign` operation $\mathcal{Z}'(i') \leftarrow j$ requires the `insert` operation $C.insert(1, j)$ followed by the execution of $\mathcal{Z}''(i') \leftarrow C.rank_1(j)$. Operation $\mathcal{Z}'.expand(j)$ is implemented with $C.insert(0, j)$.

To conclude, since we restrict our attention to the case where—when calling $\mathcal{Z}(i) \leftarrow j$—argument i is greater than all i' such that $\mathcal{Z}(i')$ is defined, we will execute `assign` operations $\mathcal{Z}''(i') \leftarrow j''$ for increasing values of $i' = 0, 1, \ldots, d-1$. In particular, at each `assign` $\mathcal{Z}''(i') \leftarrow j''$, $i' = k$ will be the current domain size. We therefore focus on a new operation, `append`, denoted as $\mathcal{Z}''.append(j'')$ and whose effect is $Z''(k) \leftarrow j''$. We are left with the problem of finding a data structure for a *dynamic permutation* $\mathcal{Z}'' : \{0, ..., k-1\} \to \{0, ..., k-1\}$ with support for `map` and `append` operations. Note that both domain and codomain size (k) are incremented by one after every `append` operation.

Example 1. Let $k = 5$ and $\mathcal{Z}''$ be the permutation $\langle 3, 1, 0, 4, 2 \rangle$. After $\mathcal{Z}''.append(2)$, k increases to 6 and $\mathcal{Z}''$ turns into the permutation $\langle 4, 1, 0, 5, 3, 2 \rangle$. Note that $\mathcal{Z}''.append(j'')$ has the following effect on the permutation: all numbers larger than or equal to j'' are incremented by one, and j'' is appended at the end of the permutation.

To implement the dynamic permutation $\mathcal{Z}''$, we use a red-black tree $\mathcal{T}$. We associate to each internal tree node x a counter storing the number of leaves contained in the subtree rooted in x. Let m be the size of the tree. The tree supports two operations:

- $\mathcal{T}.insert(j)$. Insert a new leaf at position j, i.e. the new leaf will be the j-th leaf to be visited in the in-order traversal of the tree. This operation can be implemented using subtree-size counters to guide the insertion. After the leaf has been inserted, we

need to re-balance the tree (if necessary) and update at most $\mathcal{O}(\log m)$ subtree-size counters. The procedure returns (a pointer to) the tree leaf x just inserted. Overall, $\mathcal{T}.insert(j)$ takes $\mathcal{O}(\log m)$ time
- $\mathcal{T}.locate(x)$. Take as input a leaf in the red-black tree and return the (0-based) rank of the leaf among all leaves in the in-order traversal of the tree. $\mathcal{T}.locate(x)$ requires climbing the tree from x to the root and use subtree-size counters to retrieve the desired value, and therefore runs in $\mathcal{O}(\log m)$ time.

At this point, the dynamic permutation $\mathcal{Z}''$ is implemented using the tree described above and a vector N of red-black tree leaves supporting `append` operations (i.e. insert at the end of the vector). N can be implemented with a simple vector of words with initial capacity 1. Every time we need to add an element beyond the capacity of N, we re-allocate $2|N|$ words for the array. N supports therefore constant-time access and amortized constant-time append operations. Starting with empty $\mathcal{T}$ and N, we implement operations on $\mathcal{Z}''$ as follows:

- $\mathcal{Z}''.map(i)$ returns $\mathcal{T}.locate(N[i])$
- $\mathcal{Z}''.append(j)$ is implemented by calling $N.append(\mathcal{T}.insert(j))$

Taking into account all components used to implement our original dynamic function $\mathcal{Z}$, we get the bounds of our lemma.

The algorithm The steps of our algorithm to compute $RLBWT^+(\overleftarrow{T})$ from $LZ77(T)$ are the following:

1. sort $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c\rangle \in LZ77(T) \ \wedge \pi \neq \bot\}$;
2. process $\langle \pi_v, \lambda_v, c_v\rangle_{v=1,\ldots,z}$ from the first to last triple as follows. When processing $\langle \pi_v, \lambda_v, c_v\rangle$:
   (a) use our dynamic function $\mathcal{Z}$ to convert text position π_v to RLBWT position $j' = \mathcal{Z}(\pi_v)$
   (b) extract λ_v characters from RLBWT starting from position j' by using the LF function; at the same time, extend RLBWT with the extracted characters.
   (c) when inserting a character at position j of the RLBWT, if j corresponds to some text position $i \in \mathcal{D}$, then update $\mathcal{Z}$ accordingly by setting $\mathcal{Z}(i) \leftarrow j$. If, instead, j does not correspond to any text position in $\mathcal{D}$, execute $\mathcal{Z}.expand(j)$.

Our algorithm is outlined below as Algorithm 1. Follows a detailed description of the pseudocode and a result stating its complexity.

In Lines 1-5 we initialize all structures and variables. In order: we compute and sort set $\mathcal{D}$ of phrase sources, we initialize current text position i (i is the position of the character to be read), we initialize an empty RLBWT data structure (we will build $RLBWT^+(\overleftarrow{T})$ online), and we create an empty dynamic function data structure $\mathcal{Z}$. In Line 6 we enter the main loop iterating over LZ77 factors. If the current phrase's source is not empty (i.e. if the phrase copies a previous portion of the text), we need to extract

λ_v characters from the RLBWT. First, in Line 8 we retrieve the RLBWT position j' corresponding to text position π_v with a `map` query on $\mathcal{Z}$. Note that, if $\pi_v \neq \perp$, then $i > \pi_v$ and therefore $\mathcal{Z}(\pi_v)$ is defined (see next). We are ready to extract characters from RLBWT. For λ_v times, we repeat the following procedure (Lines 10-19). We read the l-th character from the source of the v-th phrase (Line 10) and insert it in the RLBWT (Line 11). Importantly, the `extend` operation at Line 11 returns the RLBWT position j at which the new character is inserted; RLBWT position j correspond to text position i. We now have to check if i is the source of some LZ77 phrase. If this is the case (Line 12), then we link text position i to RLBWT position j by calling a `assign` query on $\mathcal{Z}$ (Line 13). If, on the other hand, i is not the source of any phrase, then we call a `expand` query on $\mathcal{Z}$ on the codomain element j. Note that, after the `extend` query at Line 11, RLBWT positions after the j-th are shifted by one. If j' is one of such positions, then we increment it (Line 17). Finally, we increment text position i (Line 19). At this point, we finished copying characters from the v-th phrase's source (or we did not do anything if the v-th phrase consists of only one character). We therefore extend the RLBWT with the v-th trailing character (Line 20), and (as done before) associate text position i to RLBWT position j if i is the source of some phrase (Lines 21-24). We conclude the main loop by incrementing the current position i on the text (Line 25). Once all characters have been extracted from LZ77, RLBWT is a run-length BWT structure on $\overleftarrow{T}$. At Line 26 we convert it to $RLBWT^+(T)$ (see previous section) and return it as a series of pairs $\langle \lambda_v, c_v \rangle_{v=1,\ldots,r}$.

Theorem 3. *Algorithm 1 converts $LZ77(T) \rightarrow RLBWT(T)$ in $\mathcal{O}(n(\log r + \log z))$ time and $\mathcal{O}(r+z)$ words of working space*

Proof. Sorting set $\mathcal{D}$ takes $\mathcal{O}(z \log z) \subseteq \mathcal{O}(n \log z)$ time. Overall, we perform $\mathcal{O}(z)$ `map`/`assign` and n `expand` queries on $\mathcal{Z}$. All these operations take globally $\mathcal{O}(n \log z)$ time. We use the structure of Theorem 1 to implement $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. We perform n `access`, `extend`, and `LF` queries on $RLBWT^+(\overleftarrow{T})$. This takes overall $\mathcal{O}(n \log r)$ time. Finally, inverting $RLBWT^+(\overleftarrow{T})$ at Line 26 takes $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of space (see previous section). We keep in memory the following structures: $\mathcal{D}$, $\mathcal{Z}$, $RLBWT^+(\overleftarrow{T})$, and $RLBWT^+(T)$. The bounds of our theorem easily follow.

5 Conclusions

In this paper we presented space-efficient algorithms converting between two compressed file representations—the run-length Burrows-Wheeler transform (RLBWT) and the Lempel-Ziv 77 parsing (LZ77)—using a working space proportional to the input and the output. Both representations can be significantly (up to exponentially) smaller than the text; our solutions are therefore particularly useful in those cases in which the text does not fit in main memory but its compressed representation does. Another application of the results discussed in this paper is the optimal-space construction of compressed self-indexes based on these compression techniques (e.g. [13]) taking as input the RLBWT/LZ77 *compressed* file.

Algorithm 1: lz77_to_rlbwt($\langle \pi_v, \lambda_v, c_v \rangle_{v=1,\ldots,z}$)

input : LZ77 factorization $LZ77(T) = \langle \pi_v, \lambda_v, c_v \rangle_{v=1,\ldots,z}$ of a text T
output: RLBWT representation $\langle \lambda_v, c_v \rangle_{v=1,\ldots,r}$ of T

```
1  D ← {π | ⟨π, λ, c⟩ ∈ LZ77(T) ∧ π ≠ ⊥};                    /* Phrase sources */
2  sort(D);                                                  /* Sort phrase sources */
3  i ← 0;                                                    /* Current position on T */
4  RLBWT ← ε;                                                /* Init empty RLBWT of reversed text */
5  Z ← ∅;                                                    /* Init empty dynamic function structure */
6  for v = 1, ..., z do
7      if π_v ≠ ⊥ then
8          j' ← Z(π_v);                                      /* Map text position to RLBWT position */
9          for l = 1, ..., λ_v do
10             c ← RLBWT[j'];                                /* read char from source */
11             j ← RLBWT.extend(c);                          /* left-extend reverse text's RLBWT */
12             if i ∈ D then
13                 Z(i) ← j;                                 /* j is the image of i */
14             else
15                 Z.expand(j);                              /* j does not have counter-image */
16             if j ≤ j' then
17                 j' ← j' + 1;                              /* new char falls before j' */
18             j' ← RLBWT.LF(j');
19             i ← i + 1;                                    /* Advance text position */
20     j ← RLBWT.extend(c_v);                                /* Extend with trailing character */
21     if i ∈ D then
22         Z(i) ← j;
23     else
24         Z.expand(j);
25     i ← i + 1;                                            /* Advance text position */
26 return reverse(RLBWT);                                    /* Build and return RLBWT(T) */
```

We point out two possible developments of our ideas. First of all, our algorithms rely heavily on dynamic data structures. On the experimental side, it has been recently shown [17] that algorithms based on compressed dynamic strings can be hundreds of times slower than others not making use of dynamism (despite offering very similar theoretical guarantees). This is due to factors ranging from cache misses to memory fragmentation; dynamic structures inherently incur into these problems as they need to perform a large number of memory allocations and de-allocations. A possible strategy for overcoming these difficulties is to build the RLBWT by merging two static RLBWTs while using a working space proportional to the output size. A second improvement over our results concerns theoretical running times. We note that our algorithms perform a number of steps proportional to the size n of the text. Considering that the compressed

file could be *exponentially* smaller than the text, it is natural to ask whether it is possible to perform the same tasks in a time proportional to $r + z$. This seems to be a much more difficult goal due to the intrinsic differences among the two compressors—one is based on suffix sorting, while the other on replacement of repetitions with pointers.

References

1. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on information theory **23**(3) (1977) 337–343
2. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on **24**(5) (1978) 530–536
3. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. Information Theory, IEEE Transactions on **51**(7) (2005) 2554–2576
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. (1994)
5. Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-length compressed indexes are superior for highly repetitive sequence collections. In: String Processing and Information Retrieval, Springer (2009) 164–175
6. Sirén, J., et al.: Compressed full-text indexes for highly repetitive collections. PhD thesis, Helsingin yliopisto (2012)
7. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science **302**(1) (2003) 211–222
8. Bannai, H., Inenaga, S., Takeda, M.: Efficient LZ78 factorization of grammar compressed text. In: String Processing and Information Retrieval, Springer (2012) 86–98
9. Bannai, H., Gawrychowski, P., Inenaga, S., Takeda, M.: Converting SLP to LZ78 in almost Linear Time. In: Combinatorial Pattern Matching, Springer (2013) 38–49
10. Tamakoshi, Y., Tomohiro, I., Inenaga, S., Bannai, H., Takeda, M.: From run length encoding to LZ78 and back again. In: Data Compression Conference (DCC), 2013, IEEE (2013) 143–152
11. Policriti, A., Prezza, N.: Computing LZ77 in Run-Compressed Space. In: Data Compression Conference (DCC), 2016, IEEE (2016)
12. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science **483** (2013) 115–133
13. Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Proc. CPM. (2015) 26–39
14. Hon, W.K., Lam, T.W., Sadakane, K., Sung, W.K., Yiu, S.M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica **48**(1) (2007) 23–36
15. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms (TALG) **3**(2) (2007) 21
16. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3) (2010) 281–308
17. Prezza, N.: A Framework of Dynamic Data Structures for String Processing. arXiv preprint arXiv:1701.07238 (2017)