Optimal Construction of Compressed Indexes for Highly Repetitive Texts*

Dominik Kempa†

Abstract

We propose algorithms that, given the input string of length n over integer alphabet of size σ, construct the Burrows–Wheeler transform (BWT), the permuted longest-common-prefix (PLCP) array, and the LZ77 parsing in $\mathcal{O}(n/\log_\sigma n + r \operatorname{polylog} n)$ time and working space, where r is the number of runs in the BWT of the input. These are the essential components of many compressed indexes such as compressed suffix tree, FM-index, and grammar and LZ77-based indexes, but also find numerous applications in sequence analysis and data compression. The value of r is a common measure of repetitiveness that is significantly smaller than n if the string is highly repetitive. Since just accessing every symbol of the string requires $\Omega(n/\log_\sigma n)$ time, the presented algorithms are time and space optimal for inputs satisfying the assumption $n/r \in \Omega(\operatorname{polylog} n)$ on the repetitiveness. For such inputs our result improves upon the currently fastest general algorithms of Belazzougui (STOC 2014) and Munro et al. (SODA 2017) which run in $\mathcal{O}(n)$ time and use $\mathcal{O}(n/\log_\sigma n)$ working space. We also show how to use our techniques to obtain optimal solutions on highly repetitive data for other fundamental string processing problems such as: Lyndon factorization, construction of run-length compressed suffix arrays, and some classical "textbook" problems such as computing the longest substring occurring at least some fixed number of times.

1 Introduction

The problem of text indexing is to preprocess the input text T so that given any query pattern P, we can quickly (typically $\mathcal{O}(|P| + \mathrm{occ})$, where $|P|$ is the length of P and occ is the number of occurrences of P in T) find all occurrences of P in T. The two classical data structures for this problem are the suffix tree [56] and the suffix array [45]. The suffix tree is a trie containing all suffixes of T with each unary path compressed into a single edge labeled by the text substring. The suffix array is a list of suffixes of T in lexicographic order where each suffix is encoded using its starting position. Both data structures take $\Theta(n)$ words of space. In addition to indexing, these data structures underpin dozens of applications in bioinformatics, data compression, and information retrieval. Suffix arrays, in particular, have become central to modern genomics, where they are used for genome assembly and short read alignment, data-intensive tasks at the forefront of modern medical and evolutionary biology [42]. This can be attributed mostly to their space-efficiency and simplicity.

In modern applications, however, which require indexing datasets of size close to the size of available RAM, even the suffix arrays can be prohibitively large, particularly in applications where the text consists of symbols from some alphabet Σ of small size $\sigma = |\Sigma|$ (e.g., in bioinformatics $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and so $\sigma = 4$). For such collections, the classical indexes are $\Theta(\log_\sigma n)$ times larger than the text which requires only $\Theta(n \log \sigma)$ bits, i.e., $\Theta(n/\log_\sigma n)$ words.

The invention of FM-index [13, 14] and the compressed suffix array (CSA) [21, 22] at the turn of the millennium addressed this issue and revolutionized the field of string algorithms for nearly two decades. These data structures require only $\mathcal{O}(n/\log_\sigma n)$ words of space and provide random access to the suffix array in $\mathcal{O}(\log^\epsilon n)$ time. Dozens of papers followed the two seminal papers, proposing various improvements, generalizations, and practical implementations (see [49, 47, 7] for excellent surveys). These indexes are now widespread, both in theory where they provide off-the-shelf small space indexing structures and in practice, particularly bioinformatics, where they are the central component of many read-aligners [40, 41].

The other approach to indexing, recently gaining popularity due to the quick increase in the amount of highly repetitive data, such as software repositories or genomic databases is designing indexes specialized for repetitive strings. The first such index [34] was based on the Lempel–Ziv (LZ77) parsing [57], the popular

*Research partially supported by the Centre for Discrete Mathematics and its Applications (DIMAP) and by EPSRC award EP/N011163/1.

†Helsinki Institute for Information Technology (HIIT), Department of Computer Science, University of Helsinki, and Department of Computer Science and Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick.

dictionary compression algorithms (used, e.g., in gzip and 7-zip compressors). Many improvements to the basic scheme were proposed since then [17, 8, 6, 16, 3, 2, 1], and now the performance of LZ-based indexes is often on par with the FM-index or CSA [12]. Independently to the development of LZ-based indexes, it was observed that the Burrows–Wheeler transform (BWT) [9], which underlies the FM-index, produces long runs of characters when applied to highly repetitive data [44, 55]. Gagie et al. [19] recently proposed a run length compressed suffix array (RLCSA) that provides fast access to suffix array and pattern matching queries in $\mathcal{O}(r \operatorname{polylog} n)$ or even $\mathcal{O}(r)$ space, where r is the number of runs in the BWT of the text. The value of r is, next to z (the size of LZ77 parsing), a common measure of repetitiveness [36].

Given the small space usage of the compressed indexes, their space-efficient construction emerged as one of the major open problems. A gradual improvement [39, 23, 24] in the construction of compressed suffix array culminated with the work of Belazzougui [5] who described the (randomized) $\mathcal{O}(n)$ time construction working in optimal space of $\mathcal{O}(n/\log_\sigma n)$. An alternative (and deterministic) construction was proposed by Munro et al. [46]. These algorithms achieve the optimal construction space but their running time is up to $\Theta(\log n)$ times larger than the lower bound of $\Omega(n/\log_\sigma n)$ time (required to read the input/write the output).

Our contribution. We propose algorithms that, given the input string of length n over integer alphabet of size σ, construct the Burrows–Wheeler transform (BWT), the permuted longest-common-prefix (PLCP) array, and the LZ77 parsing in $\mathcal{O}(n/\log_\sigma n + r \operatorname{polylog} n)$ time and working space, where r is the number of runs in the BWT of the input.

These are the essential components of nearly every compressed text index developed in the last two decades: all variants of FM-index rely on BWT [13, 19], compressed suffix arrays/trees rely on Ψ [21, 54] (which is dual to the BWT [46, 24]) and the PLCP array, and LZ77-based and grammar-based indexes rely on the LZ77 parsing [34, 53]. Apart from text indexing, these data structures have also numerous applications in sequence analysis and data compression [42, 48, 52].

Since just accessing every symbol of the string requires $\Omega(n/\log_\sigma n)$ time, the presented algorithms are time and space optimal for inputs satisfying the assumption $n/r \in \Omega(\operatorname{polylog} n)$ on the repetitiveness. Our results have particularly important implications for bioinformatics, where most of the data is highly-repetitive [44, 42, 43] and over small (DNA) alphabet. For such inputs, our result improves upon the currently fastest general algorithms of Belazzougui [5] and Munro et al. [46] which run in $\mathcal{O}(n)$ time and use $\mathcal{O}(n/\log_\sigma n)$ working space.

We also show how to use our techniques to obtain an $\mathcal{O}(n/\log_\sigma n + r \operatorname{polylog} n)$ time and space algorithms for other fundamental string processing problems such as: Lyndon factorization [10], construction of run-length compressed suffix arrays [19], and some classical "textbook" problems such as computing the longest substring occurring at least some fixed number of times.

On the way to the above results, we show how to generalize the RLCSA of Gagie et al. [19] to achieve a trade-off between index size and query time. In particular, we obtain a $\mathcal{O}(r \operatorname{polylog} n)$-space data structure that can answer suffix array queries in $\mathcal{O}(\log n/\log\log n)$ time which improves on the $\mathcal{O}(\log n)$ query time of [19].

2 Preliminaries

We assume a word-RAM model with a word of $w = \Theta(\log n)$ bits and with all usual arithmetic and logic operations taking constant time. Unless explicitly specified otherwise, all space complexities are given in words. All our algorithms are deterministic.

Throughout we consider a string $T[1..n]$ of symbols from an alphabet $\Sigma = [1..\sigma]$ of size $\sigma \leq n$. We assume $T[n] = \$$ with a numerical value of \$ equal to 0. For $j \in [1..n]$, we write $T[j..n]$ to denote the suffix j of T. We define the *rotation* of T as a string $T[j..n]T[1..j-1]$ for any position $j \in [1..n]$.

The *suffix array* [45, 20] of T is an array $\mathrm{SA}[1..n]$ which contains a permutation of the integers $[1..n]$ such that $T[\mathrm{SA}[1]..n] \prec T[\mathrm{SA}[2]..n] \prec \cdots \prec T[\mathrm{SA}[n]..n]$, where $\prec$ denotes the lexicographical order. The inverse suffix array ISA is the inverse permutation of SA, i.e., $\mathrm{ISA}[j] = i$ iff $\mathrm{SA}[i] = j$. The array $\Phi[1..n]$ (see [32]) is defined by $\Phi[\mathrm{SA}[i]] = \mathrm{SA}[i-1]$ for $i \in [2..n]$, and $\Phi[\mathrm{SA}[1]] = \mathrm{SA}[n]$, that is, the suffix $\Phi[j]$ is the immediate lexicographical predecessor of suffix j.

Let $\mathrm{lcp}(j_1, j_2)$ denote the length of the longest-common-prefix (LCP) of suffix j_1 and suffix j_2. The *longest-common-prefix array* [45, 35], $\mathrm{LCP}[1..n]$, is defined as $\mathrm{LCP}[i] = \mathrm{lcp}(\mathrm{SA}[i], \mathrm{SA}[i-1])$ for $i \in [2..n]$ and $\mathrm{LCP}[1] = 0$. The *permuted LCP array* [32] $\mathrm{PLCP}[1..n]$ is the LCP array permuted from the lexicographical order into the text order, i.e., $\mathrm{PLCP}[\mathrm{SA}[i]] = \mathrm{LCP}[i]$ for $i \in [1..n]$. Then $\mathrm{PLCP}[j] = \mathrm{lcp}(j, \Phi[j])$ for all $j \in [1..n]$.

The *succinct PLCP array* [54, 32] $\mathrm{PLCP}_{\mathrm{succ}}[1..2n]$ represents the PLCP array using $2n$ bits. Specifically, $\mathrm{PLCP}_{\mathrm{succ}}[j'] = 1$ if $j' = 2j + \mathrm{PLCP}[j]$ for some $j \in [1..n]$, and $\mathrm{PLCP}_{\mathrm{succ}}[j'] = 0$ otherwise. Any lcp value can be recovered by the equation $\mathrm{PLCP}[j] = \mathrm{select}_{\mathrm{PLCP}_{\mathrm{succ}}}(1, j) - 2j$, where $\mathrm{select}_S(c, j)$ returns the location of the j^{th} c in S.

The *Burrows–Wheeler transform* [9] $\mathrm{BWT}[1..n]$ of

T is defined by $\mathrm{BWT}[i] = T[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] > 1$ and $\mathrm{BWT}[i] = T[n]$ otherwise. Let $\mathcal{M}$ denote the $n \times n$ matrix, whose rows are lexicographically sorted rotations of T. We denote the rows by $\mathcal{M}[i]$, $i \in [1..n]$. Note that BWT is the last column of $\mathcal{M}$.

The *LF-mapping* [13] is defined by the equation $\mathrm{LF}[\mathrm{ISA}[j]] = \mathrm{ISA}[j-1]$, $j \in [2..n]$, and $\mathrm{LF}[\mathrm{ISA}[1]] = \mathrm{ISA}[n]$. By Ψ we denote the inverse of LF. The significance of LF (and the principle underlying FM-index [13]) lies in the fact that, for $i \in [1..,n]$, $\mathrm{LF}[i] = C[\mathrm{BWT}[i]] + \mathrm{rank}_{\mathrm{BWT}}(\mathrm{BWT}[i], i)$, where $C[c]$ is the number of symbols in T that are smaller than c, and $\mathrm{rank}_S(c,i)$ is the number of occurrences of c in $S[1..i]$. From the formula for LF we obtain the following fact.

LEMMA 2.1. *Let* $\mathrm{BWT}[b..e]$ *be a run of the same symbol and let* $i, i' \in [b,e]$. *Then,* $\mathrm{LF}[i] = \mathrm{LF}[i'] + (i - i')$.

If i is the rank (i.e., the number of smaller suffixes) of P among suffixes of T, then $C[c] + \mathrm{rank}_{\mathrm{BWT}}(c,i)$ is the rank of cP. This is called *backward search* [13].

We say that an lcp value $\mathrm{LCP}[i] = \mathrm{PLCP}[\mathrm{SA}[i]]$ is *reducible* if $\mathrm{BWT}[i] = \mathrm{BWT}[i-1]$ and *irreducible* otherwise. The significance of reducibility is summarized in the following two lemmas.

LEMMA 2.2. ([32]) *If* $\mathrm{PLCP}[j]$ *is reducible, then* $\mathrm{PLCP}[j] = \mathrm{PLCP}[j-1] - 1$ *and* $\Phi[j] = \Phi[j-1] + 1$.

LEMMA 2.3. ([32, 29]) *The sum of all irreducible lcp values is* $\leq n \log n$.

It can be shown [44] that repetitions in T generate equal-letter runs in BWT. By r we denote the number of runs in BWT. We can efficiently represent this transform as the list of pairs $\mathrm{RLBWT} = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$, where $\lambda_i > 0$ is the starting position of the i-th run and $c_i \in \Sigma$. Note that r is also the number of irreducible lcp values.

3 Augmenting RLBWT

In this section we present extensions of run-length compressed BWT needed by our algorithms. Each extension expands its functionality while maintaining small space usage and low construction time/space.

3.1 Rank and select support. One of the basic operations we will need are rank and select queries on BWT. We will now show that a run-length compressed BWT can be quickly augmented with a data structure capable of answering these queries in BWT-runs space.

THEOREM 3.1. *Given RLBWT of size* r *for text* $T[1..n]$ *we can add* $\mathcal{O}(r)$ *space so that, given* $i \in [0..n]$ *and* $c \in [1..\sigma]$, *values* $\mathrm{rank}_{\mathrm{BWT}}(c,i)$ *and* $\mathrm{select}_{\mathrm{BWT}}(c,i)$ *can be computed in* $\mathcal{O}(\log r)$ *time. The data structure can be constructed in* $\mathcal{O}(r \log r)$ *time using* $\mathcal{O}(r)$ *space.*

Proof. We augment each BWT-run with its length and sort the runs using the symbol as the primary key, and the start of the run as the secondary key. This allows us to compute, for every run $[b..e]$, the value $\mathrm{rank}_{\mathrm{BWT}}(c,b)$ where $c = \mathrm{BWT}[b]$. Using this list, both queries can be answered in $\mathcal{O}(\log r)$ time using binary search. □

3.2 LF/Ψ and backward search support. We now show that with the help of the above rank/select data structures we can support more complicated navigational queries, namely, given any $i \in [1..n]$ such that $\mathrm{SA}[i] = j$ we can compute $\mathrm{ISA}[j-1]$ (i.e., $\mathrm{LF}[i]$) and $\mathrm{ISA}[j+1]$ (i.e., $\Psi[i]$). Note that none of the queries will require the knowledge of j. As a simple corollary, we obtain efficient support for backward search on RLBWT.

THEOREM 3.2. *Given RLBWT of size* r *for text* $T[1..n]$ *we can add* $\mathcal{O}(r)$ *space so that, given* $i \in [1..n]$, *values* $\mathrm{LF}[i]$ *and* $\Psi[i]$ *can be computed in* $\mathcal{O}(\log r)$ *time. The data structure can be constructed in* $\mathcal{O}(r \log r)$ *time using* $\mathcal{O}(r)$ *working space.*

Proof. Similarly as in Theorem 3.1 we prepare a (sorted) list containing, for each symbol c occurring in T, the total frequency of symbols smaller than c.

To answer $\mathrm{LF}[i]$ we first compute $\mathrm{BWT}[i]$ (by searching the list of runs), then $C[\mathrm{BWT}[i]]$ (by searching the above frequency table), and finally apply Theorem 3.1. To compute $\Psi[i]$ we first determine (using the frequency table) the symbol c following $\mathrm{BWT}[i]$ in text and the number k such that this c is the k-th occurrence of c in the first column of $\mathcal{M}$. It then remains to find the k-th occurrence of c in the BWT using Theorem 3.1. □

COROLLARY 3.1. *Given RLBWT of size* r *for text* $T[1..n]$ *we can add* $\mathcal{O}(r)$ *space so that, given a rank* $i \in [0..n]$ *of a string* P *among the suffixes of* T, *for any* $c \in [1..\sigma]$ *we can compute in* $\mathcal{O}(\log r)$ *time the rank of* cP. *The data structure can be constructed in* $\mathcal{O}(r \log r)$ *time using* $\mathcal{O}(r)$ *working space.*

3.3 Suffix-rank support. In this section we describe an extension of RLBWT that will allow us to efficiently merge two RLBWTs during the BWT construction algorithm. We start by defining a generalization of BWT-runs and stating their basic properties.

Let $\mathrm{lcs}(x,y)$ denote the length of the longest common suffix of strings x and y. We define the $\mathrm{LCS}[1..n]$ array [30] as $\mathrm{LCS}[i] = \mathrm{lcs}(\mathcal{M}[i], \mathcal{M}[i-1])$ for $i \in [2..n]$ and $\mathrm{LCS}[1] = 0$ (recall that $\mathcal{M}$ is a matrix containing sorted rotations of T). Let $\tau \geq 1$ be an integer. We say that a range $[b..e]$ of BWT is a *τ-run* if $\mathrm{LCS}[b] < \tau$, $\mathrm{LCS}[e+1] < \tau$, and for any $i \in [b+1..e]$, $\mathrm{LCS}[i] \geq \tau$. By this definition, a BWT run is a 1-run. For $j \geq 0$

let $Q_j = \{i \in [1..n] \mid \mathrm{LCS}[i] = j\}$ and $R_\tau = \bigcup_{j=0}^{\tau-1} Q_j$. Then, R_τ is exactly the set of starting positions of τ-runs.

LEMMA 3.1. ([30]) *For any $i \in [2..n]$,*

$$\mathrm{LCS}[i] = \begin{cases} 0 & \text{if } \mathrm{BWT}[i] \neq \mathrm{BWT}[i-1] \\ \mathrm{LCS}[\mathrm{LF}[i]] + 1 & \text{otherwise} \end{cases}$$

Since Ψ is the inverse of LF we obtain that for any $j \geq 1$, $Q_j = \{\Psi[i] \mid i \in Q_{j-1} \text{ and } \Psi[i] \notin Q_0\}$. Thus, the set R_τ can be efficiently computed by iterating each of the starting positions of BWT-runs $\tau - 1$ times using Ψ and taking a union of all visited positions. From the above we see that $|Q_{j+1}| \leq |Q_j|$, which implies that the number of τ-runs satisfies $|R_\tau| \leq |Q_0|\tau = r\tau$.

THEOREM 3.3. *Let $S[1..m]$, $S'[1..m']$ be strings with r and r' (respectively) runs in the* BWT. *Given RLBWTs of S and S' it is possible, for any integer $\tau \geq 1$, to build a data structure of size $\mathcal{O}(\frac{m}{\tau} + r + r')$ that can, given a rank $i \in [0..m]$ of some suffix $S[j..m]$ among suffixes of S, compute the rank of $S[j..m]$ among suffixes of S' in $\mathcal{O}(\tau(\log \frac{m}{\tau} + \log r + \log r'))$ time. The data structure can be constructed in $\mathcal{O}(\tau^2(r+r')\log(r\tau + r'\tau) + \frac{m}{\tau}(\log(r\tau) + \log(r'\tau) + \log \frac{m}{\tau}))$ time and $\mathcal{O}(\tau^2(r+r') + \frac{m}{\tau})$ space.*

Proof. We start by augmenting both RLBWTs with Ψ and LF support (Theorem 3.2) and RLBWT of S' with the backward search support (Corollary 3.1). This requires $\mathcal{O}(r \log r + r' \log r')$ time and $\mathcal{O}(r + r')$ space.

We then compute a (sorted) set of starting positions of τ-runs for both RLBWTs. For S this requires answering $r\tau$ Ψ-queries which takes $\mathcal{O}(r\tau \log r)$ time in total, and then sorting the resulting set of positions in $\mathcal{O}((r\tau)\log(r\tau))$ time. Analogous processing for S' takes $\mathcal{O}((r'\tau)\log(r'\tau))$ time. The starting positions of all τ-runs require $\mathcal{O}((r+r')\tau)$ space in total.

Next, for any τ-run $[b..e]$ we compute and store the associated τ symbols. We also store the value $\mathrm{LF}^\tau[b]$, but only for τ-runs of S. Due to simple generalization of Lemma 2.1, this will allow us to compute the value $\mathrm{LF}^\tau[i]$ for *any* i. In total this requires answering $\tau^2(r+r')$ LF-queries and hence takes $\mathcal{O}(\tau^2(r+r')\log r)$ time. The space needed to store all symbols is $\mathcal{O}(\tau^2(r+r'))$.

We then lexicographically sort all length-τ strings associated with τ-runs (henceforth called *τ-substrings*) and assign to each run the rank of the associated substring in the sorted order. Importantly, τ-substrings of S and S' are sorted together. These ranks will be used as order-preserving names for τ-substrings. We use an LSD string sort with a stable comparison-based sort for each position hence the sorting takes $\mathcal{O}\left(\tau^2(r+r')\log(r\tau + r'\tau)\right)$ time. The working space does not exceed $\mathcal{O}(\tau(r+r'))$. After the names are computed, we discard the substrings.

We now observe that order-preserving names for τ-substrings allow us to perform backward search τ symbols at a time. We build a rank-support data structure analogous to the one from Theorem 3.1 for names of τ-substrings of S'. We also add support for computing the total number of occurrences of names smaller than a given name. This takes $\mathcal{O}(r'\tau \log(r'\tau))$ time and $\mathcal{O}(r'\tau)$ space. Then, given a rank i of suffix $S[j..m]$ among suffixes of S', we can compute the rank of suffix $S[j-\tau..m]$ among suffixes of S' in $\mathcal{O}(\log(r'\tau))$ time by backward search on S' using i as a position, and the name of τ-substring preceding $S[j..m]$ as a symbol.

We now use the above multi-symbol backward search to compute the rank of every suffix of the form $S[m - k\tau..m]$ among suffixes of S'. We start from the shortest suffix and increase the length by τ in every step. During the computation we also maintain the rank of the current suffix of S among suffixes of S. This allows us to efficiently compute the name of the preceding τ-substring. The rank can be updated using values LF^τ stored with each τ-run of S. Thus, for each of the m/τ suffixes of S we obtain a pair of integers $(i_S, i_{S'})$, denoting its rank among the suffixes of S and S'. We store these pairs as a list sorted by i_S. Computing the list takes $\mathcal{O}\left(\frac{m}{\tau}(\log(r\tau) + \log(r'\tau)) + \frac{m}{\tau}\log\frac{m}{\tau}\right)$ time. After the list is computed we discard all data structures associated with τ-runs.

Using the above list of ranks we can answer the query from the claim as follows. Starting with i, we compute a sequence of τ positions in the BWT of S by iterating Ψ on i. For each position we can check in $\mathcal{O}(\log \frac{m}{\tau})$ time whether that position is in the list of ranks. Since we evenly sampled text positions, one of these positions has to correspond to the suffix of S for which we computed the rank in the previous step. Suppose we found such position after $\Delta \leq \tau$ steps, i.e., we now have a pair $(i_S, i_{S'})$ such that $i_{S'}$ is the rank of $S[j+\Delta..m]$ among suffixes of S'. We then perform Δ steps of the standard backward search starting from rank $i_{S'}$ in the BWT of S' using symbols $S[j+\Delta-1], \ldots, S[j]$. This takes $\mathcal{O}\left(\Delta(\log r + \log r')\right) = \mathcal{O}\left(\tau(\log r + \log r')\right)$ time. □

4 Constructing BWT

In this section we show that given the packed encoding of text $T[1..n]$ over alphabet $\Sigma = [1..\sigma]$ of size $\sigma \leq n$ (i.e., using $\mathcal{O}(n/\log_\sigma n)$ words of space), we can compute the packed encoding of BWT of T in $\mathcal{O}(n/\log_\sigma n + r\log^7 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^5 n)$ space, where r is the number of runs in the BWT of T.

4.1 Algorithm overview.

The basic scheme of our algorithm follows the algorithm of Hon et al. [24]. Assume for simplicity that $w/\log\sigma = 2^k$ for some

integer k. The algorithm works in $k+1$ rounds, where $k = \log\log_\sigma n$. In the i-th round, $i \in [0..k]$, we interpret T as a string over superalphabet $\Sigma_i = [1..\sigma_i]$ of size $\sigma_i = \sigma^{2^i}$, i.e., we group symbols of T into supersymbols consisting of 2^i original symbols. We denote this string as T_i. The rounds are executed in decreasing order of i. The input to the i-th round, $i \in [0..k-1]$, is the run-length compressed BWT of T_{i+1}, and the output is the run-length compressed BWT of T_i. We denote the size of RLBWT of T_i by r_i. The final output is the run-length compressed BWT of $T_0 = T$, which we then convert into packed encoding taking $\mathcal{O}(n/\log_\sigma n)$ words.

For the k-th round, we observe that $|\Sigma_k| = \Theta(n)$ and $|T_k| = \Theta(n/\log_\sigma n)$ hence to compute BWT of T_k it suffices to first run any of the linear-time algorithms for constructing the suffix array [33, 51, 38, 37] for T_k and then naively compute the RLBWT from the suffix array. This takes $\mathcal{O}(n/\log_\sigma n)$ time and space.

Let $S = T_i$ for some $i \in [0..k-1]$ and suppose we are given the RLBWT of T_{i+1}. Let S_o be the string of length $|S|/2$ created by grouping together symbols $S[2j-1]S[2j]$ for all j, and let S_e be the analogously constructed string for pairs $S[2j]S[2j+1]$. Clearly we have $S_o = T_{i+1}$ (recall that we start indexing from 1). Furthermore, it is easy to see that the BWT of S can be obtained by interleaving BWTs of S_o and S_e, and discarding (more significant) half of the bits in the encoding of each symbol.

The construction of RLBWT for S consists of two steps: (1) first we compute the RLBWT of S_e from RLBWT of S_o, and then (2) merge RLBWTs of S_o and S_e into RLBWT of S.

4.2 Computing BWT of S_e. In this section we assume that $S = T_i$ for some $i \in [0..k-1]$ and that we are given the RLBWT of $S_o = T_{i+1}$ of size $r_o = r_{i+1}$. Denote the size of RLBWT of S_e by r_e. We will show that RLBWT of S_e can be computed in $\mathcal{O}(r_e + r_o \log r_o)$ time using $\mathcal{O}(r_o + r_e)$ working space.

Recall that both S_o and S_e are over alphabet Σ_{i+1}. Each of the symbols in that alphabet can be interpreted as a concatenation of two symbols in the alphabet Σ_i. Let c be the symbol of either S_o or S_e and assume that $c = S[j]S[j+1]$ for some $j \in [1..|S|-1]$. By *major subsymbol* of c we denote a symbol (equal to $S[j]$) from Σ_i encoded by the more significant half of bits encoding c, and by *minor subsymbol* we denote symbol encoded by remaining bits (equal to $S[j+1]$).

We first observe that by enumerating all runs of the RLBWT of S_o in increasing order of their minor subsymbols (and in case of ties, in the increasing order of run beginnings), we obtain (on the remaining bits) the minor subsymbols of the BWT of S_e in the correct order. Such enumeration could easily be done in $\mathcal{O}(r_o \log r_o)$ time and $\mathcal{O}(r_o)$ working space. To obtain the missing (major) part of the encoding of symbols in the BWT of S_e, it suffices to perform the LF-step for each of the runs in the BWT of S_o in the sorted order above (i.e., by minor subsymbol), and look up the minor subsymbols in the resulting range of BWT of S_o.

The problem with the above approach is the running time. While it indeed produces correct RLBWT of S_e, having to scan all runs in the range of BWT of S_o obtained by performing the LF-step on each of the runs of S_o could be prohibitively high. To address this we first construct a run-length compressed sequence of minor subsymbols extracted from BWT of S_o and use it to extract minor subsymbols of BWT of S_o in total time proportional to the number of runs in the BWT of S_e.

LEMMA 4.1. *Given RLBWT of size r_o for $S_o = T_{i+1}$ we can compute the RLBWT of S_e in $\mathcal{O}(r_e + r_o \log r_o)$ time and $\mathcal{O}(r_o + r_e)$ working space, where r_e is the size of RLBWT of S_e.*

Proof. The whole process requires scanning the BWT of S_o to create a run-length compressed encoding of minor subsymbols, adding the LF support to (the original) RLBWT of S_o, sorting the runs in RLBWT of S_o by the minor subsymbol, and executing r_o LF-queries on the BWT of S_o, which altogether takes $\mathcal{O}(r_o \log r_o)$. All other operations take time proportional to $\mathcal{O}(r_o + r_e)$. The space never exceeds $\mathcal{O}(r_o + r_e)$. □

4.3 Merging BWTs of S_e and S_o. As in the previous section, we assume $S = T_i$ for some $i \in [0..k-1]$ and that we are given the RLBWT of $S_o = T_{i+1}$ of size $r_o = r_{i+1}$ and RLBWT of S_e of size r_e. We will show how to use these to efficiently compute the RLBWT of S in $\mathcal{O}(|S|/\log|S| + (r_o + r_e)\,\mathrm{polylog}\,|S|)$ time and space.

We start by observing that to obtain BWT of S it suffices to merge the BWT of S_e and BWT of S_o and discard all major subsymbols in the resulting sequence. The algorithm of Hon et al. [24] achieves this by performing the backward search. This requires $\Omega(|S|)$ time and hence is too expensive in our case.

Instead, we employ the following observation. Suppose we have already computed the first t runs of the BWT of S and let the next unmerged character in the BWT of S_o be a part of a run of symbol c_o. Let c_e be the analogous symbol from the BWT of S_e. Further, let c'_e (resp. c'_o) be the minor subsymbol of c_e (resp. c_o). If $c'_o = c'_e$ then either all symbols in the current run in the BWT of S_o (restricted to minor subsymbols) or all symbols in the current run in the (also restricted) BWT of S_e will belong to the next run in the BWT of S. Assuming we can determine the order between any

two arbitrary suffixes of S_o and S_e given their ranks in the respective BWTs, we could consider both cases and in each perform a binary search to find the exact length of $(t+1)$-th run in the BWT of S. We first locate the end of the run of c'_o (resp. c'_e) in the BWT of S_o (resp. S_e) restricted to minor subsymbols; this can be done after preprocessing input BWTs without increasing the time/space of the merging. We then find the largest suffix of S_e (resp. S_o) not greater than the suffix at the end of the run in the BWT of S_o. Importantly, the time to compute the next run in the BWT of S does not depend on the number of times the suffixes in that run alternate between S_o and S_e. The case $c'_e \neq c'_o$ is handled similarly, except we do not need to locate the end of each run. The key property of this algorithm is that the number of pattern searches is $\mathcal{O}(r_i \log |S|)$.

Thus, the merging problem can be reduced to the problem of efficient comparison of suffixes of S_e and S_o. To achieve that we augment both RLBWTs of S_e and S_e with the suffix-rank support data structure from Section 3.3. This will allow us to determine, given a rank of any suffix of S_o, the number of smaller suffixes of S_e and vice-versa, thus eliminating even the need for binary search. Our aim is to achieve $\mathcal{O}(|S|/\log|S|)$ space and construction time assuming small r values, thus we apply Theorem 3.3 with $\tau = \log^2|S|$.

LEMMA 4.2. *Given RLBWT of size r_e for S_e and RLBWT of size r_o for $S_o = T_{i+1}$ we can compute the RLBWT of $S = T_i$ in $\mathcal{O}((r_o+r_e)\log^5|S| + |S|/\log|S| + r_i \log^3|S|)$ time and $\mathcal{O}(|S|/\log^2|S| + (r_o+r_e)\log^4|S| + r_i)$ working space.*

Proof. Constructing the suffix-rank support for S_o and S_e with $\tau = \log^2|S|$ takes $\mathcal{O}((r_o+r_e)\log^5|S| + |S|/\log|S|)$ time and $\mathcal{O}((r_o+r_e)\log^4|S| + |S|/\log^2|S|)$ working space. The resulting data structures occupy $\mathcal{O}(|S|/\log^2|S| + r_e + r_o)$ space and answer suffix-rank queries in $\mathcal{O}(\log^3|S|)$ time. To compute the RLBWT of S we perform $2r_i$ suffix-rank queries for a total of $\mathcal{O}(r_i \log^3|S|)$ time. □

4.4 Putting things together. To bound the size of RLBWTs in intermediate rounds, consider the i-th round where for $d = 2^i$ we group each d symbols of T to obtain the string $S = T_i$ of length $|T|/d$ and let r_i be the number of runs in the BWT of S. Recall now the construction of generalized BWT-runs from Section 3.3 and observe that the symbols of T comprising each supersymbol $S[j]$ are the same as the substring corresponding to d-run containing suffix $T[jd+1..n]$ in the BWT of T. It is easy to see that the corresponding suffixes of T are in the same lexicographic order as the suffixes of S. Thus, r_i is bounded by the number of d-runs in the BWT of T, which by Section 3.3 is bounded by rd. Hence, the size of the output RLBWT of the i-th round does not exceed $r2^i = \mathcal{O}(r \log n)$. The analogous analysis shows that the size of RLBWT of S_e has the same upper bound $r2^{i+1}$ as $S_o = T_{i+1}$.

THEOREM 4.1. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \leq n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words, the* BWT *of T can be computed in $\mathcal{O}(n/\log_\sigma n + r\log^7 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^5 n)$ working space, where r is the number of runs in the* BWT *of T.*

Proof. The k-th round of the algorithm takes $\mathcal{O}(n/\log_\sigma n)$ time working space and produces a BWT taking $\mathcal{O}(n/\log_\sigma n)$ words of space. Consider the i-th round of the algorithm for $i < k$ and let $S = T_i$, and r_e and r_o denote the sizes of RLBWT of S_e and S_o respectively. By the above discussion, we have $r_o, r_e = \mathcal{O}(r \log n)$. Thus, by Lemma 4.1 and Lemma 4.2 the i-th round takes $\mathcal{O}(n_i/\log n_i + r\log^6 n_i) = \mathcal{O}(n/(2^i \log n) + r\log^6 n)$ time and the working space does not exceed $\mathcal{O}(n/\log^2 n + r\log^5 n)$ words, where $n_i = |T_i| = n/2^i$, and we used the fact that for $i < k$, $\log n_i = \Theta(\log n)$. Hence over all rounds we spend $\mathcal{O}(n/\log_\sigma n + r\log^7 n)$ time and never use more than $\mathcal{O}(n/\log_\sigma n + r\log^5 n)$ space. Finally, it is easy to convert RLBWT into the packed encoding in $\mathcal{O}(n/\log_\sigma n + r\log n)$ time. □

Thus, we obtained a time- and space-optimal construction algorithm for BWT under the assumption $n/r = \Omega(\text{polylog}\, n)$ on the repetitiveness of the input.

5 Constructing PLCP

In this section we show that given the run-length compressed representation of BWT of T, it is possible to compute the $\text{PLCP}_{\text{succ}}$ bitvector in $\mathcal{O}(n/\log n + r\log^{11} n)$ time and $\mathcal{O}(n/\log n + r\log^{10} n)$ working space..

The key observation used to construct the PLCP values is that it suffices to only compute the irreducible LCP values. Then, by Lemma 2.2, all other values can be quickly deduced. This significantly simplifies the problem because it is known (Lemma 2.3) that the sum of irreducible LCP values is bounded by $\mathcal{O}(n \log n)$.

The main idea of the construction is to compute (as in Theorem 3.3) names of τ-runs for $\tau = \log^5 n$. This will allow us to compare τ symbols at a time and thus quickly compute a lower bound for large irreducible LCP values. Before we can use this, we need to augment the BWT with the support for SA/ISA queries.

5.1 Computing SA/ISA **support.** Suppose that we are given a run-length compressed BWT of $T[1..n]$ taking $\mathcal{O}(r)$ space. Let $\tau \geq 1$ be an integer. Assume for simplicity that n is a multiple of τ. We start by

computing the sorted list of starting positions of all τ-runs similarly, as in Theorem 3.3. This requires augmenting the RLBWT with the LF/Ψ support first and in total takes $\mathcal{O}(\tau r \log(\tau r))$ time and $\mathcal{O}(\tau r)$ working space. We then compute and store, for the first position of each τ-run $[b..e]$, the value of $\mathrm{LF}^\tau[b]$. This will allow us to efficiently compute $\mathrm{LF}^\tau[i]$ for any $i \in [1..n]$.

We then locate the occurrence i_0 of the symbol \$ in BWT and perform n/τ iterations of LF^τ on i_0. By definition of LF, the position i visited after j iterations of LF^τ is equal to $\mathrm{ISA}[n - j\tau]$, i.e., $\mathrm{SA}[i] = n - j\tau$. For any such i we save the pair $(i, n - j\tau)$ into a list. When we finish the traversal we sort the list by the first component (assume this list is called L_{SA}). We then create the copy of the list (call it L_{ISA}) and sort it by the second component. Creating the lists takes $\mathcal{O}\left((n/\tau)(\log(r\tau) + \log(n/\tau))\right)$ time and they occupy $\mathcal{O}(n/\tau)$ space. After the lists are computed we discard LF^τ samples associated with all runs. Having these lists allows us to efficiently query SA/ISA as follows.

To compute $\mathrm{ISA}[j]$ we find in $\mathcal{O}(1)$ time (since we can store L_{ISA} in an array) the pair (p, j') in L_{ISA} such that $j' = \lceil j/\tau \rceil \tau$. We then perform $j' - j < \tau$ steps of LF on position p. The total query time is thus $\mathcal{O}(\tau \log r)$.

To compute $\mathrm{SA}[i]$ we perform τ steps of LF (each taking $\mathcal{O}(\log r)$ time) on position i. Due to the way we sampled SA/ISA values, one of the visited positions has to be the first component in the L_{SA} list. For each position, we can check this in $\mathcal{O}(\log(n/\tau))$ time. Suppose we found a pair after $\Delta < \tau$ steps, i.e., a pair $(\mathrm{LF}^\Delta[i], p)$ is in L_{SA}. This implies $\mathrm{SA}[\mathrm{LF}^\Delta[i]] = p$, i.e., $\mathrm{SA}[i] = p + \Delta$. The query time is $\mathcal{O}\left(\tau(\log r + \log(n/\tau))\right)$.

THEOREM 5.1. *Given RLBWT of size r for text $T[1..n]$, we can, for any integer $\tau \geq 1$, build a data structure taking $\mathcal{O}(r + n/\tau)$ space that, for any $i \in [1..n]$, can answer* $\mathrm{SA}[i]$ *query in $\mathcal{O}(\tau(\log r + \log(n/\tau)))$ time and* $\mathrm{ISA}[i]$ *query in $\mathcal{O}(\tau \log r)$ time. The construction takes $\mathcal{O}\left((n/\tau)(\log(r\tau) + \log(n/\tau)) + \tau^2 r \log(r\tau)\right)$ time and $\mathcal{O}(n/\tau + r\tau)$ working space.*

5.2 Computing irreducible LCP values. We start by augmenting the RLBWT with the SA/ISA support as explained in the previous section using $\tau_1 = \log^2 n$. The resulting data structure answers SA/ISA queries in $\mathcal{O}(\log^3 n)$ time. We then compute τ_2-runs and their names using the technique introduced in Theorem 3.3 for $\tau_2 = \log^5 n$.

Given any $j_1, j_2 \in [1..n]$ we can check whether it holds $T[j_1..j_1 + \tau_2 - 1] = T[j_2..j_2 + \tau_2 - 1]$ using the above names as follows. Compute $i_1 = \mathrm{ISA}[j_1 + \tau_2]$ and $i_2 = \mathrm{ISA}[j_2 + \tau_2]$ using the ISA support. Then compare the names of τ_2-substrings preceding these two suffixes. Thus, comparing two arbitrary substrings of T of length τ_2, given their text positions, takes $\mathcal{O}(\log^3 n)$ time.

The above toolbox allows computing all irreducible LCP values as follows. For any $i \in [1..n]$ such that $\mathrm{LCP}[i]$ is irreducible (such i can be recognized by checking if $\mathrm{BWT}[i-1]$ belongs to a BWT-run different than $\mathrm{BWT}[i]$) we compute $j_1 = \mathrm{SA}[i-1]$ and $j_2 = \mathrm{SA}[i]$. We then have $\mathrm{LCP}[i] = \mathrm{lcp}(T[j_1..n], T[j_2..n])$. We start by computing the lower-bound for $\mathrm{LCP}[i]$ using the names of τ_2-substrings. Since the sum of irreducible LCP values is bounded by $\mathcal{O}(n \log n)$, over all irreducible LCP values this will take $\mathcal{O}(r \log^3 n + \log^3 n \cdot (n \log n)/\tau_2) = \mathcal{O}(r \log^3 n + n/\log n)$ time. Finishing the computation of each LCP value requires at most τ_2 symbol comparisons. This can be done by following Ψ for both pointers as long as the preceding symbols (found in the BWT) are equal. Over all irreducible LCP values, finishing the computation takes $\mathcal{O}(r\tau_2 \log n) = \mathcal{O}(r \log^6 n)$ time.

THEOREM 5.2. *Given RLBWT of size r for $T[1..n]$, the* $\mathrm{PLCP}_{\mathrm{succ}}$ *bitvector (or the list storing irreducible LCP values in text order) can be computed in $\mathcal{O}(n/\log n + r \log^{11} n)$ time and $\mathcal{O}(n/\log n + r \log^{10} n)$ working space.*

Proof. Adding the SA/ISA support using $\tau_1 = \log^2 n$ takes $\mathcal{O}\left(n/\log n + r \log^5 n\right)$ time and $\mathcal{O}(n/\log^2 n + r \log^2 n)$ working space (Theorem 5.1). The resulting structure needs $\mathcal{O}(r + n/\log^2 n)$ space and answers SA/ISA queries in $\mathcal{O}(\log^3 n)$ time.

Computing the names takes $\mathcal{O}(\tau_2^2 r \log(\tau_2 r)) = \mathcal{O}(r \log^{11} n)$ time and $\mathcal{O}(\tau_2^2 r) = \mathcal{O}(r \log^{10} n)$ working space (see the proof of Theorem 3.3). The names need $\mathcal{O}(\tau_2 r) = \mathcal{O}(r \log^5 n)$ space. Then, by the above discussion, computing all irreducible LCP values takes $\mathcal{O}(n/\log n + r \log^6 n)$ time. □

By combining with Theorem 4.1 we obtain the following result.

THEOREM 5.3. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \leq n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words, the* $\mathrm{PLCP}_{\mathrm{succ}}$ *bitvector (or the list storing irreducible LCP values in text order) can be computed in $\mathcal{O}(n/\log_\sigma n + r \log^{11} n)$ time and $\mathcal{O}(n/\log_\sigma n + r \log^{10} n)$ working space, where r is the number of runs in the* BWT *of T.*

6 Construction of RLCSA

In this section, we show how to use the techniques presented in this paper to quickly build the run-length compressed suffix array (RLCSA) recently proposed by Gagie et al. [19]. They observed that if BWT of T has r runs then the arrays SA/ISA and LCP have a bidirectional parse of size $\mathcal{O}(r)$ after being differentially encoded. They use a locally-consistent parsing [4, 26] to grammar-compress these arrays and describe the

necessary augmentations to achieve fast decoding of the original values. This allowed them to obtain a $\mathcal{O}(r \operatorname{polylog} n)$-space structure that can answer SA/ISA and LCP queries in $\mathcal{O}(\log n)$ time.

The structure described below is slightly different than the original index proposed by Gagie et al. [19]. Rather than compressing the differentially-encoded suffix array, we directly exploit the structure of the array. It can be thought of as a multi-ary block tree [6] modified to work with arrays indexed in "lex-order" instead of the original "text-order". Our data structure matches the space and query time of [19], but we additionally show how to achieve a trade-off between space and query time. In particular, we achieve $\mathcal{O}(\log n/\log\log n)$ query time in $\mathcal{O}(r \operatorname{polylog} n)$ space.

Data structure. Suppose we are given RLBWT of size r for text $T[1..n]$. The data structure is parametrized by an integer parameter $\tau > 1$. For simplicity, we assume that r divides n and that n/r is a power of τ. The data structure is organized into $\log_\tau(n/r)$ *levels*. The main idea is, for every level, to store 2τ pointers for each BWT-run boundary. The purpose of pointers is to reduce the SA query near the associated run boundary into SA query at a position that is closer (by at least a factor of τ) to some (usually different) run boundary. Level controls the allowed proximity of the query. At the last level, the SA value at each run boundary is stored explicitly.

More precisely, for $1 \le k \le \log_\tau(n/r)$, let $b_k = n/(r\tau^k)$ and let $\mathrm{BWT}[b..e]$ be one of the runs in the BWT. Consider 2τ non-overlapping consecutive blocks of size b_k evenly spread around position b, i.e., $\mathrm{BWT}[b + ib_k..b + (i+1)b_k - 1]$, $i = -\tau, \ldots, \tau - 1$. For each block $\mathrm{BWT}[s..t]$ we store the smallest d (called *LF-distance*) such that there exists at least one $i \in [s..t]$ such that $\mathrm{LF}^d[i]$ is the beginning of the run in the BWT of T (note that it is possible that $d = 0$). With each block we also store the value $\mathrm{LF}^d[s]$ (called *LF-shortcut*), both as an absolute value in $[1..n]$ and as a pointer to the BWT-run that contains it. Due to the simple generalization of Lemma 2.1, this allows us to compute $\mathrm{LF}^d[i]$ for *any* $i \in [s..t]$. At each level, we store 2τ integers for each of r BWT runs thus in total we store $\mathcal{O}(r\tau \log_\tau(n/r))$ words.

To access $\mathrm{SA}[i]$ we proceed as follows. Assume first that i is not more than n/r positions from the closest run boundary. We first find the BWT run that contains i. We then follow the LF-shortcuts starting at level 1 down to the last level. After every step, the distance to the closest run boundary is reduced by a factor τ. Thus, after $\log_\tau(n/r)$ steps the current position is equal to boundary b of some run $\mathrm{BWT}[b..e]$. Let d_{sum} denote the total lengths of LF-distances of the used shortcuts. Since $\mathrm{SA}[b]$ is stored we can now answer the query as $\mathrm{SA}[i] = \mathrm{SA}[b] + d_{\mathrm{sum}}$. To handle positions further than n/r from the nearest run boundary, we add a lookup table $LT[1..r]$ such that $LT[i]$ stores the LF-shortcut and LF-distance for block $\mathrm{BWT}[(i-1)(n/r)+1..i(n/r)]$. The query time is $\mathcal{O}(\log_\tau(n/r))$, since blocks in the same level have the same length and hence at each level we spend $\mathcal{O}(1)$ time to find the pointer to the next level. Note that the lookup table eliminates the initial search of run containing i.

The above data structure can be generalized to extract segments of $\mathrm{SA}[p..p + \ell - 1]$, for any p and ℓ, faster than ℓ single SA-accesses, that would cost $\mathcal{O}(\ell \log_\tau(n/r))$. The main modification is that at level k we instead consider $4\tau - 1$ blocks of size b_k, evenly spread around position b, each overlapping the next by exactly $b_k/2$ symbols, i.e., $\mathrm{BWT}[b + ib_k/2..b + (i+2)b_k/2 - 1]$, $i = -2\tau, \ldots, 2(\tau - 1)$. This guarantees that any segment-access to SA of length at most $b_k/2$ at level k can be transformed into the segment-access at level $k + 1$. We also truncate the data structure at level k where k is the smallest integer with $b_k < \log_\tau(n/r)$. At that level we store a segment of $2\log_\tau(n/r)$ SA values around each BWT run. These values take $\mathcal{O}(r \log_\tau(n/r))$ space, and hence the two modifications do not increase the space needed by the data structure. This way we can extract $\mathrm{SA}[p..p + \alpha - 1]$, where $\alpha = \log_\tau(n/r)$ in $\mathcal{O}(\alpha)$ time, and consequently a segment $\mathrm{SA}[p..p + \ell - 1]$ in $\mathcal{O}((\ell/\alpha + 1)\alpha) = \mathcal{O}(\ell + \log_\tau(n/r))$ time.

THEOREM 6.1. *Assume that* BWT *of* $T[1..n]$ *consist of* r *runs. For any integer* $\tau > 1$*, there exists a data structure of size* $\mathcal{O}(r\tau \log_\tau(n/r))$ *that, for any* $p \in [1..n]$ *and* $\ell \ge 1$*, can compute* $\mathrm{SA}[p..p + \ell - 1]$ *in* $\mathcal{O}(\ell + \log_\tau(n/r))$ *time.*

For $\tau = 2$ the above data structure matches the space and query time of [19]. For $\tau = \log^\epsilon n$, where $\epsilon > 0$ is an arbitrary constant it achieves $\mathcal{O}(r \log^\epsilon n \log(n/r))$ space and $\mathcal{O}(\log n/\log\log n)$ query time. Finally, for $\tau = (n/r)^\epsilon$ it achieves $\mathcal{O}(r^{1-\epsilon} n^\epsilon)$ space and $\mathcal{O}(1)$ time query. In particular, if $r = o(n)$ the data structure takes $o(n)$ space and is able to access (any segment of) SA in optimal time.

Construction algorithm. Assume we are given the run-length compressed BWT of $T[1..n]$ of size r. Consider any block $\mathrm{BWT}[s..t]$. Let d be the corresponding LF-distance and let $\mathrm{LF}^d[i] = b$ for some $i \in [s..t]$ be the beginning of a BWT-run $[b..e]$. We observe that this implies $\mathrm{LCP}[b]$ is irreducible and $\mathrm{LCP}[b] \ge d$.

We start by augmenting the RLBWT with the SA/ISA support from Section 5.1 using $\tau_1 = \log^2 n$. This, by Theorem 5.1, takes $\mathcal{O}\left(n/\log n + r\log^5 n\right)$

time and $\mathcal{O}(n/\log^2 n + r\log^2 n)$ working space. The resulting structure needs $\mathcal{O}(r + n/\log^2 n)$ space and allows answering SA/ISA queries in $\mathcal{O}(\log^3 n)$ time.

Consider now the sorted sequence Q containing every position j in T such that PLCP$[j]$ is irreducible. Such list can be obtained by computing value SA$[b]$ for every BWT run $[b..e]$ and sorting the resulting values. Computing the list Q takes $\mathcal{O}(r\log^3 n)$ time and $\mathcal{O}(r)$ working space. The list itself is stored in plain form using $\mathcal{O}(r)$ space. Next, for any irreducible value PLCP$[j]$ we compute, for any $t = 1, \ldots, \lfloor \ell'/\tau_2 \rfloor$ a pair containing ISA$[j + t\tau_2]$ (as key) and $t\tau_2$ (as value), where $\tau_2 = \log^4 n$, and ℓ' is the distance between j and its successor in Q. Since the sum of ℓ' values is $\mathcal{O}(n)$, computing all pairs takes $\mathcal{O}(\log^3 n \cdot (r + n/\tau_2)) = \mathcal{O}(n/\log n + r\log^3 n)$ time and $\mathcal{O}(n/\tau_2) = \mathcal{O}(n/\log^4 n)$ working space. The resulting pairs need $\mathcal{O}(n/\log^4 n)$ space.

We then sort all the computed pairs by the keys and build a static RMQ data structure over the associated values. This can be done in $\mathcal{O}(n/\tau_2)) = \mathcal{O}(n/\log^4 n)$ time and space so that an RMQ query takes $\mathcal{O}(\log n)$ time (using static balanced BST).

Having the above samples augmented with the RMQ allows us to compute LF-shortcuts as follows. Let BWT$[s..t]$ be one of the blocks. We perform τ_2 LF-steps on position s. In step Δ we first check in $\mathcal{O}(\log r)$ time whether the block $[\mathrm{LF}^{\Delta}[s]..\mathrm{LF}^{\Delta}[s]+(t-s)]$ contains a boundary of a BWT-run. If yes, then we found the LF-distance and terminate the procedure. Otherwise, in $\mathcal{O}(\log n)$ we compute the minimal value $d_{\min}$ and its position for the block $[\mathrm{LF}^{\Delta}[s]..\mathrm{LF}^{\Delta}[s]+(t-s)]$ using the RMQ structure (if the block is empty we skip this step). We call $d_{\min} + \Delta$ the *candidate value*. From the way we computed the pairs, the minimum candidate value is equal to the LF-distance of BWT$[s..t]$. It is easy to extend this procedure to also return the LF-shortcut.

Thus, the LF-shortcut for any block can be computed in $\mathcal{O}(\tau_2 \log n) = \mathcal{O}(\log^5 n)$ time. Over all blocks (and including the shortcuts for the lookup table $LT[1..r]$) this takes $\mathcal{O}(r\tau\log_\tau(n/r)\log^5 n) = \mathcal{O}(r\tau\log^6 n)$ time. Finally, computing segments of SA values at the last level (after truncating the tree) takes $\mathcal{O}(r\log_\tau(n/r)\log^3 n)$ time.

THEOREM 6.2. *Given RLBWT of size r for text $T[1..n]$ we can build the data structure from Theorem 6.1 in $\mathcal{O}(n/\log n + r\tau\log^6 n)$ time and $\mathcal{O}(n/\log^2 n + r(\tau\log_\tau(n/r) + \log^2 n))$ working space.*

By combining with Theorem 4.1 we obtain the following theorem.

THEOREM 6.3. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \le n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words we can build the data structure from Theorem 6.1 in $\mathcal{O}(n/\log_\sigma n + r(\tau\log^6 n + \log^7 n))$ time and $\mathcal{O}(n/\log_\sigma n + r(\tau\log_\tau(n/r) + \log^5 n))$ working space, where r is the number of runs in the* BWT *of T.*

7 Construction of LZ77 parsing

In this section, we show how to use the techniques introduced in previous sections to obtain a fast and space-efficient LZ77 factorization algorithm for highly repetitive strings.

7.1 Definitions. The LZ77 factorization [57] uses the notion of the *longest previous factor* (LPF). The LPF at position i (denoted LPF$[i]$) in T is a pair (p_i, ℓ_i) such that, $p_i < i$, $T[p_i..p_i + \ell_i - 1] = T[i..i + \ell_i - 1]$ and $\ell_i > 0$ is maximized. In other words, $T[i..i + \ell_i - 1]$ is the longest prefix of $T[i..n]$ which also occurs at some position $p_i < i$ in T. If $T[i]$ is the leftmost occurrence of a symbol in T then such a pair does not exist. In this case we define $p_i = T[i]$ and $\ell_i = 0$. Note that there may be more than one potential p_i, and we do not care which one is used.

The LZ77 factorization (or LZ77 parsing) of a string T is then just a greedy, left-to-right parsing of T into longest previous factors. More precisely, if the j^{th} LZ factor (or *phrase*) in the parsing is to start at position i, then we output (p_i, ℓ_i) (to represent the j^{th} phrase), and then the $(j+1)^{\text{th}}$ phrase starts at position $i + \ell_i$, unless $\ell_i = 0$, in which case the next phrase starts at position $i + 1$. For the example string $T = zzzzzipzip$, the LZ77 factorization produces:

$$(z, 0), (1, 4), (i, 0), (p, 0), (5, 3).$$

We denote the number of phrases in the LZ77 parsing of T by z. The following theorem shows that LZ77 parsing can be encoded in $\mathcal{O}(n\log\sigma)$ bits.

THEOREM 7.1. (E.G. [27]) *The number of phrases z in the LZ77 parsing of a text of n symbols over an alphabet of size σ is $\mathcal{O}(n/\log_\sigma n)$.*

The LPF pairs can be computed using *next and previous smaller values* (NSV/PSV) defined as

$$\mathrm{NSV}_{\mathrm{lex}}[i] = \min\{j \in [i+1..n] \mid \mathrm{SA}[j] < \mathrm{SA}[i]\}$$
$$\mathrm{PSV}_{\mathrm{lex}}[i] = \max\{j \in [1..i-1] \mid \mathrm{SA}[j] < \mathrm{SA}[i]\}.$$

If the set on the right hand side is empty, we set the value to 0. We further define

$$\mathrm{NSV}_{\mathrm{text}}[i] = \mathrm{SA}[\mathrm{NSV}_{\mathrm{lex}}[\mathrm{ISA}[i]]]$$
$$\mathrm{PSV}_{\mathrm{text}}[i] = \mathrm{SA}[\mathrm{PSV}_{\mathrm{lex}}[\mathrm{ISA}[i]]].$$

If $\mathrm{NSV}_{\mathrm{lex}}[\mathrm{ISA}[i]] = 0$ ($\mathrm{PSV}_{\mathrm{lex}}[\mathrm{ISA}[i]] = 0$) we set $\mathrm{NSV}_{\mathrm{text}}[i] = 0$ ($\mathrm{PSV}_{\mathrm{text}}[i] = 0$).

The usefulness of the NSV/PSV values is summarized by the following lemma.

LEMMA 7.1. ([11]) *For $i \in [1..n]$, let $i_{nsv} = \mathrm{NSV}_{\mathrm{text}}[i]$, $i_{psv} = \mathrm{PSV}_{\mathrm{text}}[i]$, $\ell_{nsv} = \mathrm{lcp}(i, i_{nsv})$ and $\ell_{psv} = \mathrm{lcp}(i, i_{psv})$. Then*

$$\mathrm{LPF}[i] = \begin{cases} (i_{nsv}, \ell_{nsv}) & \text{if } \ell_{nsv} > \ell_{psv} \\ (i_{psv}, \ell_{psv}) & \text{if } \ell_{psv} = \max(\ell_{nsv}, \ell_{psv}) > 0 \\ (T[i], 0) & \text{if } \ell_{nsv} = \ell_{psv} = 0. \end{cases}$$

7.2 Algorithm overview. The general approach of our algorithm follows the lazy LZ77 factorization algorithms of [31]. Namely, we opt out from computing all LPF values and instead compute $\mathrm{LPF}[j]$ only when there is an LZ factor starting at position j.

Suppose we have already computed the parsing of $T[1..j-1]$. To compute the factor starting at position j we first query $i = \mathrm{ISA}[j]$. We then compute (using a small-space data structure introduced next) values $i_{\mathrm{nsv}} = \mathrm{NSV}_{\mathrm{lex}}[i]$ and $i_{\mathrm{psv}} = \mathrm{PSV}_{\mathrm{lex}}[i]$. By Lemma 7.1 it then suffices to compute the lcp of $T[j..n]$ and each of the two suffixes starting at positions $\mathrm{SA}[i_{\mathrm{psv}}]$ and $\mathrm{SA}[i_{\mathrm{nsv}}]$.

It is easy to see that the total length of computed lcps will be $\mathcal{O}(n)$, since after each step we increase j by the longest of the two lcps. To perform the lcp computation efficiently we will employ the technique from Section 5 which allows comparing multiple symbols at a time. This will allow us to spend $\mathcal{O}(z \operatorname{polylog} n + n/\log n)$ time in the lcp computation. The problem is thus reduced to being able to quickly answer $\mathrm{NSV}_{\mathrm{lex}}/\mathrm{PSV}_{\mathrm{lex}}$ queries.

7.3 Computing NSV/PSV support for SA. Assume that we are given RLBWT of size $\mathcal{O}(r)$ for text $T[1..n]$. We will show how to quickly build a small-space data structure that, given any $i \in [1..n]$ can compute $\mathrm{NSV}_{\mathrm{lex}}[i]$ or $\mathrm{PSV}_{\mathrm{lex}}[i]$ in $\mathcal{O}(\operatorname{polylog} n)$ time.

We split $\mathrm{BWT}[1..n]$ into blocks of size $\tau = \Theta(\operatorname{polylog} n)$ and for each $j \in [1..n/\tau]$ we compute the minimum value in $\mathrm{SA}[(j-1)\tau+1..j\tau]$ together with its position. We then build a balanced binary tree over the array of minimas and augment each internal node with the minimum value in its subtree. This allows, for any $j \in [1..n/\tau]$, and any value x, to find the maximal (resp. minimal) $j' < j$ (resp. $j' > j$) such that $\mathrm{SA}[(j'-1)\tau+1..j'\tau]$ contains a value smaller than x. At query time we first scan the SA positions preceding or following the query position $i \in [1..n]$ inside the block containing i. If there is no value smaller than $\mathrm{SA}[i]$, we use the RMQ to find the closest block with a value smaller than $\mathrm{SA}[i]$. To finish the query it then suffices to scan the SA values inside that block. It takes $\mathcal{O}(\log^3 n)$ time to compute SA value (Theorem 5.1), hence answering a single $\mathrm{NSV}_{\mathrm{lex}}/\mathrm{PSV}_{\mathrm{lex}}$ query will take $\mathcal{O}(\tau \log^3 n)$.

To compute the minimum for each of the size-τ blocks of SA we observe that, up to a shift by a constant, there is only $r\tau$ different blocks. More specifically, consider a block $\mathrm{SA}[(j-1)\tau+1..j\tau]$. Let k be the smallest integer such that for some $t \in [(j-1)\tau+1..j\tau]$, $\mathrm{LF}^k[t]$ is the beginning of a run in BWT. It is easy to see that, due to Lemma 2.1, $\mathrm{SA}[(j-1)\tau+1..j\tau] = k + \mathrm{SA}[\mathrm{LF}^k[j\tau]-\tau+1..\mathrm{LF}^k[j\tau]]$, in particular, the equality holds for the minimum element. Thus, it suffices to precompute the minimum value and its position for each of the $r\tau$ size-τ blocks intersecting a boundary of a BWT-run. This takes $\mathcal{O}(r\tau \log^3 n)$ time and $\mathcal{O}(r\tau)$ working space. The resulting values need $\mathcal{O}(r\tau)$ space.

It thus remains to compute the "LF-distance" for each of the n/τ blocks of SA, i.e., the smallest k such that for at least one position t inside the block, $\mathrm{LF}^k[t]$ is the beginning of a BWT-run. To achieve this we utilize the technique used in Section 6. There we presented a data structure of size $\mathcal{O}(r + n/\log^2 n + n/\tau_2)$ that can be built in $\mathcal{O}(n/\log n + r\log^5 n + (n\log^3 n)/\tau_2)$ time and $\mathcal{O}(n/\log^2 n + r\log^2 n + n/\tau_2)$ working space, and is able to compute the LF-shortcut for any block $[s..t]$ in SA in $\mathcal{O}(\tau_2 \log n)$ time.

THEOREM 7.2. *Given RLBWT of size r for text $T[1..n]$, we can build a data structure of size $\mathcal{O}(r + n/\log^2 n)$ that can answer $\mathrm{PSV}_{\mathrm{lex}}/\mathrm{NSV}_{\mathrm{lex}}$ queries in $\mathcal{O}(\log^9 n)$ time. The data structure can be built in $\mathcal{O}(n/\log n + r\log^9 n)$ time and $\mathcal{O}(n/\log^2 n + r\log^6 n)$ working space.*

Proof. We start by augmenting the RLBWT with SA/ISA support. This takes (Theorem 5.1) $\mathcal{O}(n/\log n + r\log^5 n)$ time and $\mathcal{O}(n/\log^2 n + r\log^2 n)$ working space. The resulting data structure takes $\mathcal{O}(r + n/\log^2 n)$ space and answers SA/ISA queries in $\mathcal{O}(\log^3 n)$ time.

To achieve the $\mathcal{O}(n/\log n)$ term in the construction time for the structure from Section 6 we set $\tau_2 = \log^4 n$. Then, computing the LF-shortcut for any block in SA takes $\mathcal{O}(\log^5 n)$ time. Since we have n/τ blocks to query, we set $\tau = \log^6 n$ to obtain $\mathcal{O}(n/\log n)$ total query time. Answering a single $\mathrm{NSV}_{\mathrm{lex}}/\mathrm{PSV}_{\mathrm{lex}}$ query then takes $\mathcal{O}(\tau \log^3 n) = \mathcal{O}(\log^9 n)$.

The RMQ data structure built on top of the minimas of the blocks of SA takes $\mathcal{O}(n/\tau) = \mathcal{O}(n/\log^6 n)$ space, hence the space of the final data structure is dominated by SA/ISA support taking $\mathcal{O}(r + n/\log^2 n)$ words.

The construction time is split between precomputing the minimas in each of the $r\tau$ blocks crossing boundaries of BWT-runs in $\mathcal{O}(r\tau \log^3 n) = \mathcal{O}(r \log^9 n)$ time, and other steps introducing term $\mathcal{O}(n/\log n)$.

The working space is maximized when building the SA/ISA support and during the precomputation of minimas in each of the $r\tau$ blocks, for a total of $\mathcal{O}(n/\log^2 n + r\log^6 n)$. □

7.4 Algorithm summary

THEOREM 7.3. *Given RLBWT of size r of $T[1..n]$, the LZ77 factorization of T can be computed in $\mathcal{O}(n/\log n + r\log^9 n + z\log^9 n)$ time and $\mathcal{O}(n/\log^2 n + z + r\log^8 n) = \mathcal{O}(n/\log_\sigma n + r\log^8 n)$ working space, where z is the size of the LZ77 parsing of T.*

Proof. We start by augmenting the RLBWT with the SA/ISA support from Section 5.1 using $\tau_1 = \log^2 n$. This, by Theorem 5.1, takes $\mathcal{O}\left(n/\log n + r\log^5 n\right)$ time and $\mathcal{O}(n/\log^2 n + r\log^2 n)$ working space. The resulting structure needs $\mathcal{O}(r + n/\log^2 n)$ space and answers SA/ISA queries in $\mathcal{O}(\log^3 n)$ time.

Next, we initialize the data structure supporting the $\mathrm{PSV}_{\mathrm{lex}}/\mathrm{NSV}_{\mathrm{lex}}$ queries from Section 7.3. By Theorem 7.2 the resulting data structure needs $\mathcal{O}(r + n/\log^2 n)$ space and answers queries in $\mathcal{O}(\log^9 n)$ time. The data structure can be built in $\mathcal{O}(n/\log n + r\log^9 n)$ time and $\mathcal{O}(n/\log^2 n + r\log^6 n)$ working space. Over the course of the whole algorithm, we ask $\mathcal{O}(z)$ queries hence in total we spend $\mathcal{O}(z\log^9 n)$ time.

Lastly, we compute τ_3-runs and their names using the technique introduced in Section 5.2 for $\tau_3 = \log^4 n$. This takes $\mathcal{O}(\tau_3^2 r\log(\tau_3 r)) = \mathcal{O}(r\log^9 n)$ time and $\mathcal{O}(\tau_3^2 r) = \mathcal{O}(r\log^8 n)$ working space (see the proof of Theorem 5.2). The names need $\mathcal{O}(\tau_3 r) = \mathcal{O}(r\log^4 n)$ space. The names allow, given any $j_1, j_2 \in [1..n]$, to compute $\ell = \mathrm{lcp}(j_1, j_2)$ in $\mathcal{O}\left(\log^3 n(1 + \ell/\tau_3) + \tau_3\log n\right) = \mathcal{O}\left(\log^5 n + \ell/\log n\right)$ time. Thus, over the course of the whole algorithm we will spend $\mathcal{O}(z\log^5 n + n/\log n)$ time computing lcp values. □

By combining with Theorem 5.2 we obtain the following result.

THEOREM 7.4. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \le n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words, we can compute the LZ77 factorization of T in $\mathcal{O}(n/\log_\sigma n + r\log^9 n + z\log^9 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^8 n)$ working space, where r is the number of runs in the* BWT *of T and z is the size of the LZ77 parsing of T.*

Since $z = \mathcal{O}(r\log n)$ [18], the above algorithm achieves $\mathcal{O}(n/\log_\sigma n)$ runtime and working space when $n/r \in \Omega(\mathrm{polylog}\, n)$.

8 Construction of Lyndon factorization

In this section, we show another application of our techniques. Namely, we show that we can obtain a fast and space-efficient construction of Lyndon factorization for highly repetitive strings.

8.1 Definitions. A string S is called a *Lyndon word* if S is lexicographically smaller than all its non-empty proper suffixes. The *Lyndon factorization* (also called *Standard factorization*) of a string T is its unique (see [10]) factorization $T = f_1^{e_1}\cdots f_m^{e_m}$ such that each f_i is a Lyndon word, $e_i \ge 1$, and $f_i \succ f_{i+1}$ for all $1 \le i < m$. We call each f_i a *Lyndon factor* of T, and each $F_i = f_i^{e_i}$ a *Lyndon run* of T. The size of the Lyndon factorization is m, the number of distinct Lyndon factors, or equivalently, the number of Lyndon runs.

Each Lyndon run can be encoded as a triple of integers storing the boundaries of some occurrence of f_i in T and the exponent e_i. Since, for any string, it holds $m < 2z$ [28] and $z = \mathcal{O}(n/\log_\sigma n)$ [27], where z is the number of phrases in the LZ77 parsing, it follows that Lyndon factorization can be stored in $\mathcal{O}(n\log\sigma)$ bits.

8.2 Algorithm overview. Our algorithm utilizes many of the algorithms from the long line of research on algorithms operating on compressed representations such as grammars or LZ77 parsing:

- Furuya et al. [15] have shown that given an SLP (i.e., a grammar in Chomsky normal form generating a single string) of size g generating string T of length n, the Lyndon factorization of T can be computed in $\mathcal{O}(P(g,n) + Q(g,n)\, g\log\log n)$ time and $\mathcal{O}(g\log n + S(g,n))$ space, where $P(g,n)$, $S(g,n)$, $Q(g,n)$ are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) queries on SLPs. The LCE query, given two positions i and j in the string T, returns $\mathrm{lcp}(i,j)$, i.e., the length of the longest common prefix of suffixes $T[i..n]$ and $T[j..n]$.
- On the other hand, Nishimoto et al. [50, Thm 3] have shown how, given an SLP of size g generating string T of length n, to construct an LCE data structure in $\mathcal{O}(g\log\log g\log n\log^* n) = \mathcal{O}(g\log^3 n)$ time and $\mathcal{O}(g\log^* n + z\log n\log^* n) = \mathcal{O}(g\log^2 n)$ space, where z is the size of LZ77 parsing of T. The resulting data structure answers a query $\mathrm{LCE}(i,j)$ in $\mathcal{O}(\log n + \log\ell\log^* n) = \mathcal{O}(\log^2 n)$ time, where $\ell = \mathrm{lcp}(i,j)$. Thus, they achieve $P(g,n) = \mathcal{O}(g\log^3 n)$, $S(g,n) = \mathcal{O}(g\log^2 n)$, and $Q(g,n) = \mathcal{O}(\log^2 n)$. More recently, I [25, Thm 2] improved (using different techniques) this to $P(g,n) = \mathcal{O}(g\log(n/g))$, $S(g,n) = \mathcal{O}(g + z\log(n/z))$, and $Q(g,n) = \mathcal{O}(\log n)$.
- Finally, Rytter [53, Thm 2] have shown how, given the LZ77 parsing of string T of length n, to convert it into an SLP of size $g = \mathcal{O}(z\log n)$ in $\mathcal{O}(z\log n)$ time and $\mathcal{O}(z\log n)$ working space.

The above pipeline leads to a fast and space-efficient algorithm for Lyndon factorization, assuming

the compressed representation (such as SLP or LZ77) of text is given *a priori*. It still, however, needs $\Omega(n)$ time if we take into account the time to compute LZ77 or a small grammar using the previously fastest known algorithms. Section 7 completes this line of research by providing fast and space-efficient construction of the initial component (LZ77 parsing).

THEOREM 8.1. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \leq n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words of space, we can compute the Lyndon factorization of T in $\mathcal{O}(n/\log_\sigma n + r\log^9 n + z\log^9 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^8 n + z\log^2 n)$ working space.*

Proof. We start by computing the LZ77 parsing using Theorem 7.4. This takes $\mathcal{O}(n/\log_\sigma n + r\log^9 n + z\log^9 n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^8 n)$ space. The resulting parsing, by Theorem 7.1, takes $\mathcal{O}(n/\log_\sigma n)$ space.

We then use the Rytter's [53] conversion from LZ77 to SLP of size $g = \mathcal{O}(z\log n)$ that takes $\mathcal{O}(z\log n)$ time and $\mathcal{O}(z\log n)$ working space. The resulting SLP is then turned into an LCE data structure of I [25]; this takes $\mathcal{O}(g\log(n/g)) = \mathcal{O}(z\log^2 n)$ time and $\mathcal{O}(g + z\log(n/z)) = \mathcal{O}(z\log n)$ working space. The resulting LCE data structure takes $\mathcal{O}(z\log n)$ space. Finally, we plug this data structure into the algorithm of Furuya [15] which gives us the Lyndon factorization in $\mathcal{O}(z\log^3 n)$ time and $\mathcal{O}(z\log^2 n)$ working space. Thus, the whole pipeline is dominated (in time and space) by the construction of LZ77 parsing. □

Similarly as in Section 7, since $z = \mathcal{O}(r\log n)$ [18], the above algorithm achieves $\mathcal{O}(n/\log_\sigma n)$ runtime and working space when $n/r \in \Omega(\text{polylog}\, n)$.

9 Solutions to textbook problems

Lastly, we show how to utilize the techniques presented in this paper to efficiently solve some "textbook" string problems on highly repetitive inputs. Their solution usually consists of computing SA or LCP and performing some simple scan/traversal (e.g., computing the longest repeating substring amounts to finding the maximal value in the LCP array and hence by Theorem 5.3 it can be solved efficiently for highly repetitive input), but in some cases requires explicitly applying some of the observations from previous sections. Next, we show two examples of such problems.

9.1 Number of distinct substrings.

The number d of distinct substrings of a string T of length n is given by the formula

$$d = \frac{n(n+1)}{2} - \sum_{i=1}^{n} \text{LCP}[i].$$

Suppose we are given a (sorted) list $(i_1, \ell_1), \ldots, (i_r, \ell_r)$ of irreducible lcp values (i.e., $\text{PLCP}[i_k] = \ell_k$) of string T. Since all other lcp values can be derived from this list using Lemma 2.2, we can rewrite the above formula (letting $i_{r+1} = n+1$) as:

$$d = \frac{n(n+1)}{2} - \sum_{k=1}^{r} f(\ell_k, i_{k+1} - i_k),$$

where

$$f(v,d) = \begin{cases} \frac{v(v+1)}{2} & \text{if } v < d \\ d(v-d) + \frac{d(d+1)}{2} & \text{otherwise} \end{cases}$$

Thus, by Theorem 5.3 we immediately obtain the following result.

THEOREM 9.1. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \leq n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words, we can compute the number d of distinct substrings of T in $\mathcal{O}(n/\log_\sigma n + r\log^{11} n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^{10} n)$ space, where r is the number of runs in the* BWT *of T.*

9.2 Longest substring occurring k times.

Suppose that we want to find the length ℓ of the longest substring of T that occurs in T at least $2 \leq k = \mathcal{O}(1)$ times. This amounts to computing

$$\ell = \max_{i=1}^{n-k+2} \min_{j=0}^{k-2} \text{LCP}[i+j].$$

For $k = 2$ the above formula can be evaluated by only looking at irreducible lcp values, i.e., using the definition from the previous section, $\ell = \max_{i=1}^{r} \ell_i$. For $k > 2$, this does not work, since we have to inspect blocks of LCP values of size $k-1$ in "lex-order". We instead utilize observations from previous sections. More precisely, recall from Section 7 that for any τ, up to a shift by a constant, there is only $r\tau$ different blocks of size τ in SA, i.e., for any block block $\text{SA}[i..i+\tau-1]$ there exists k such that $\text{SA}[i..i+\tau-1] = k + \text{SA}[j..j+\tau-1]$ and $\text{BWT}[j..j+\tau-1]$ contains a BWT-run boundary.

We now observe that an analogous property holds for the LCP array: for any block $\text{LCP}[i..i+\tau-1]$ there exists k (the same as above) such that $\text{LCP}[i..i+\tau-1] = \text{LCP}[j..j+\tau-1] - k$ and $\text{BWT}[j..j+\tau-1]$ contains boundary of some BWT-run. This implies that we only need to precompute and store the minimum value inside blocks of LCP of length $k-1$ that are not further than τ positions from the closest BWT-run boundary. All other blocks of LCP can be handled using the above observation and the structure from Section 6 for computing the LF-shortcut for any block of BWT. More precisely, after a suitable overlap (by at least k) of blocks of size $\tau = \Omega(\text{polylog}\, n)$, we can get the answer for all such blocks in $\mathcal{O}(n/\text{polylog}\, n + r\,\text{polylog}\, n)$ time.

THEOREM 9.2. *Given string $T[1..n]$ over alphabet $[1..\sigma]$ of size $\sigma \leq n$ encoded in $\mathcal{O}(n/\log_\sigma n)$ words, we can find the length of the longest substring occurring $\geq k = \mathcal{O}(1)$ times in T in $\mathcal{O}(n/\log_\sigma n + r \operatorname{polylog} n)$ time and space.*

10 Concluding remarks

An important avenue for future work is to reduce the exponent in the $\mathcal{O}(r \operatorname{polylog} n)$-term of our bounds and to determine whether the presented algorithms can be efficiently implemented in practice. Another interesting problem is to settle whether the $\mathcal{O}(\log n/\log\log n)$ bound obtained in Section 6 is optimal within $\mathcal{O}(r \operatorname{polylog} n)$ space.

Acknowledgments

We would like to thank Tomasz Kociumaka for helpful comments and Isamu Furuya, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda for sharing an early version of their paper [15].

References

[1] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. *ACM J. Exp. Algor.*, 15, 2010.

[2] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Inf. Comput.*, 209(7):1070–1102, 2011.

[3] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101, 2012.

[4] T. Batu, F. Ergün, and S. C. Sahinalp. Oblivious string embeddings and edit distance approximations. In *Proc. 17th Annual Symposium on Discrete Algorithms (SODA 2006)*, pages 792–801, 2006.

[5] D. Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. 46th Annual ACM Symposium on Theory of Computing (STOC 2014)*, pages 148–193, 2014.

[6] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. O. Pereira, S. J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. Data Compression Conference (DCC 2015)*, pages 83–92, 2015.

[7] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4):23:1–23:19, 2014.

[8] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.*, 713:66–77, 2018.

[9] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[10] K.-T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Ann. Math.*, 68:81–95, 1958.

[11] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.

[12] H. Ferrada, D. Kempa, and S. J. Puglisi. Hybrid indexing revisited. In *Proc. 20th Meeting on Algorithm Engineering and Experiments (ALENEX 2018)*, pages 1–8, 2018.

[13] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398, 2000.

[14] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[15] I. Furuya, Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Lyndon factorization of grammar compressed texts revisited. In *Proc. 29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, pages 24:1–24:10, 2018.

[16] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, pages 240–251, 2012.

[17] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Theoretical Informatics Symposium (LATIN 2014)*, pages 731–742, 2014.

[18] T. Gagie, G. Navarro, and N. Prezza. On the approximation ratio of Lempel-Ziv parsing. In *Proc. 13th Latin American Theoretical Informatics Symposium (LATIN 2018)*, pages 490–503, 2018.

[19] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th Annual Symposium on Discrete Algorithms (SODA 2018)*, pages 1459–1477, 2018.

[20] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.

[21] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406, 2000.

[22] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[23] W. Hon, T. W. Lam, K. Sadakane, and W. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation (ISAAC 2003)*, pages 240–249, 2003.

[24] W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of*

Computer Science (FOCS 2003), pages 251–260, 2003.

[25] T. I. Longest common extensions with recompression. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pages 18:1–18:15, 2017.

[26] A. Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015.

[27] J. Kärkkäinen. *Repetition-based Text Indexes.* PhD thesis, University of Helsinki, 1999.

[28] J. Kärkkäinen, D. Kempa, Y. Nakashima, S. J. Puglisi, and A. M. Shur. On the size of Lempel-Ziv and Lyndon factorizations. In *Proc. 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, pages 45:1–45:13, 2017.

[29] J. Kärkkäinen, D. Kempa, and M. Piatkowski. Tighter bounds for the sum of irreducible LCP values. *Theor. Comput. Sci.*, 656:265–278, 2016.

[30] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Slashing the time for BWT inversion. In *Proc. Data Compression Conference (DCC 2012)*, pages 99–108, 2012.

[31] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lazy Lempel-Ziv factorization algorithms. *ACM J. Exp. Algor.*, 21(1):2.4:1–2.4:19, 2016.

[32] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, pages 181–192, 2009.

[33] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, pages 943–955, 2003.

[34] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP 1996)*, pages 141–155, 1996.

[35] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, pages 181–192, 2001.

[36] D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on Theory of Computing (STOC 2018)*, pages 827–840, 2018.

[37] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, pages 186–199, 2003.

[38] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discr. Alg.*, 3(2-4):143–156, 2005.

[39] T. W. Lam, K. Sadakane, W. Sung, and S. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th International Computing and Combinatorics Conference (COCOON 2002)*, pages 401–410, 2002.

[40] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Gen. Biol.*, 10(3), 2009.

[41] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[42] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

[43] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of individual genomes. In *Proc. 13th Conference on Research in Computational Molecular Biology (RECOMB 2009)*, pages 121–137, 2009.

[44] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

[45] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[46] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual Symposium on Discrete Algorithms (SODA 2017)*, pages 408–424, 2017.

[47] G. Navarro. Wavelet trees for all. *J. Discr. Alg.*, 25:2–20, 2014.

[48] G. Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

[49] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[50] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, pages 72:1–72:15, 2016.

[51] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.*, 60(10):1471–1484, 2011.

[52] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

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

[54] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232, 2002.

[55] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, pages 164–175, 2008.

[56] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT/FOCS 1973)*, pages 1–11, 1973.

[57] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.