Refining the $r$-index\(^\star\)

Hideo Bannai\(^{a,b,1}\), Travis Gagie\(^{c,d,2}\), Tomohiro I\(^e,3,\ast\)

\(^a\)Department of Informatics, Kyushu University, Japan
\(^b\)RIKEN Center for Advanced Intelligence Project, Japan
\(^c\)School of Computer Science and Telecommunications, Diego Portales University, Chile
\(^d\)Center for Biotechnology and Bioengineering, Chile
\(^e\)Department of Artificial Intelligence, Kyushu Institute of Technology, Japan

Abstract

Gagie, Navarro and Prezza’s $r$-index (SODA, 2018) promises to speed up DNA alignment and variation calling by allowing us to index entire genomic databases, provided certain obstacles can be overcome. In this paper we first strengthen and simplify Policriti and Prezza’s Toehold Lemma (DCC ’16; Algorithmica, 2017), which inspired the $r$-index and plays an important role in its implementation. We then show how to update the $r$-index efficiently after adding a new genome to the database, which is likely to be vital in practice. As a by-product of this result, we obtain an online version of Policriti and Prezza’s algorithm for constructing the LZ77 parse from a run-length compressed Burrows-Wheeler Transform. Our experiments demonstrate the practicality of all three of these results. Finally, we show how to augment the $r$-index such that, given a new genome and fast random access to the database, we can quickly compute the matching statistics and maximal exact matches of the new genome with respect to the database.

Keywords: Burrow-Wheeler Transform, FM-index, $r$-index, dynamic indexing, LZ77 parsing, matching statistics

\(^\star\)A preliminary version [1] of this paper was presented at CPM 2018 under the title “Online LZ77 parsing and matching statistics with RLBWTs”.

\(^\ast\)Corresponding author.

Email addresses: bannai@inf.kyushu-u.ac.jp (Hideo Bannai),
travis.gagie@gmail.com (Travis Gagie), tomohiro@ai.kyutech.ac.jp (Tomohiro I)

\(^1\)Partially funded by JSPS KAKENHI Grant Number JP16H02783.
\(^2\)Partially funded by Fondecyt grant 1171058.
\(^3\)Partially funded by JSPS KAKENHI Grant Number JP16K16009.

Preprint submitted to Theoretical Computer Science February 15, 2019
1. Introduction

Since the turn of the millennium, advances in DNA sequencing technologies have taken us from sequencing a full human genome for the first time to storing databases of hundreds of thousands of genomes. These advances have far outpaced Moore’s Law and now processing and storing genomic data are becoming a bottleneck. After running a DNA sample through a sequencing machine to obtain tens or hundreds of millions of overlapping substrings of the genome, called reads, the next step is usually to determine how the newly sequenced genome differs from a reference genome. This process is known as variation calling and consists of aligning each read to the most similar section of the reference, building a consensus sequence from the aligned reads, comparing that to the reference sequence, and then encoding the differences in variation-calling format (VCF) [2]. Because humans are genetically almost identical, variation calling is drastically easier than assembling a genome without a reference, which is known as de novo assembly. De novo assembly is often likened to building a huge jigsaw puzzle without the box, while variation calling is like building one while looking at the box from a slightly different puzzle. Of course, both processes are complicated by sequencing errors, uneven coverage of the genome by the reads, repetitions in the genome, etc.

The matching in variation calling is usually done with Bowtie [3], BWA [4], or other software based on the FM-index [5], the success of which has turned it into a cornerstone of bioinformatics and compact data structures. Although the FM-index is well-suited to indexing a single reference genome, however, the standard implementation does not scale well to genomic databases. Such scalability is desirable because if we include more genomes in our index, then more reads will match exactly some section of one or more of those genomes, reducing the need for more difficult approximate matching [6] [7]. Aligning reads against whole genomic databases is called pan-genomic alignment [8] and should help genomic processing and storage catch up with sequencing. Unfortunately, although several authors have proposed other kinds of indexes (see, e.g., [9] [10] and references therein), they lack the complete functionality of the FM-index and have not achieved the same popularity. In particular, they often limit the maximum length of a pattern, which will become problematic as reads get longer and more accurate (so matches get longer).
Figure 1: (a) De novo assembly of the reads ATTTTC, GATTA and TACAT into GATTACAT, indicating the third T in ATTTTC is an error. (b) Variation calling of the reads ACAT and GATA against the reference GATTACAT, indicating they come from a genome with the second T missing. (c) The read ATAC does not match exactly against the reference GATTACAT but does against the second genome GATACAT we assembled, so if we add that genome to the index then we can avoid using approximate pattern matching to align that read.

gives a very small example of de novo assembly, variation calling, and the advantage of pan-genomic alignment.

To understand why the standard implementation of the FM-index does not scale well, it helps to examine its two main components: first, a rank data structure over the Burrows-Wheeler Transform (BWT) of the reference, with which we compute the interval of the suffix array (SA) containing the starting positions of the given pattern, which tells us how often the pattern occurs; and second, an SA sample, with which we can recover the contents of that interval, which tells us where the pattern occurs. Although the run-length compressed BWT (RLBWT) of the database stays small as we add more genomes [11], the regular SA sample either expands or slows down, such that the product of its size and query time grows linearly with the database. For example, if our current database is $\text{GATTACAT}$$^1\text{GATACAT}$$^2\text{GATTAGATA}$$^3$ and we append $\text{GATAGATTA}$$^4$, then the BWT changes from $\text{TTATTTTCCGGGGAAA}$$^1\text{A}$$^3\text{AAATATAA}$ to $\text{TTAATTTTTCGCGGGGAAA}$$^1\text{A}$$^3\text{AAATATAAA}$, with only about 14% more runs, while an SA sample with the same query time grows by
about 37%. This divergence becomes more pronounced when the genomes are longer, more similar and more numerous.

Policriti and Prezza [12] showed how we can store SA entries only at the beginning and end of each run in the BWT and still quickly return the location of one occurrence of the given pattern, and used this to obtain an efficient algorithm for turning the RLBWT into the LZ77 parse. We refer to their result about finding one occurrence as the Toehold Lemma, since Gagie, Navarro and Prezza [13] recently built on it to obtain a scalable version of the FM-index, called the $r$-index, which promises to make pan-genomic alignment practical and useful. Before that promise can be fulfilled, however, several obstacles must still be overcome: first, we need efficient algorithms to build RLBWTs and SA samples of genomic databases, which are the main components of $r$-indexes; second, we need an efficient way to update the $r$-index when we add a new genome to the database, because rebuilding it regularly will be prohibitively slow regardless of the algorithms we use; and third, as reads become longer and more likely to contain combinations of variation that we have seen before individually but not all together, we will need support for finding maximal exact matches between the read and the database. Boucher et al. [14] and Kuhnle et al. [15] have since made substantial progress on the first point, and in this paper we address the second one and give a theoretical solution to the third. As a by-product of making the $r$-index dynamic, we obtain an online algorithm for computing the LZ77 parse in space bounded in terms of the number of runs in the BWT.

In Section 2 we review some previous results that we will use throughout this paper, and strengthen Policriti and Prezza’s Toehold Lemma to require SA entries only at the beginnings of the runs in the BWT — which significantly improves the practical performance of the $r$-index [15] — and simplify its proof. In Section 3 we show how to update the $r$-index efficiently when adding a new genome to the database, and in Section 4 we show how that can be applied to compute the LZ77 parse online from a growing $r$-index. In this paper we concern ourselves only with adding a new genome, not with supporting insertions at a specified point (between two given genomes currently adjacent in the current database); however, we note that this seems possible by combining our approach with Mantaci et al.’s extended BWT [16]. Finally, in Section 5 we show how to further augment the $r$-index such that, given a new genome and fast random access to the database (which can easily be added to VCF files), we can quickly compute the matching statistics and maximal exact matches of the new genome with respect to the database.
Matching statistics are a popular tool in bioinformatics and so calculating them is of independent interest, but in this case we are motivated by rare-disease detection and variation calling with maximal exact matches. We note that in the conference version of this paper the additional space was $O(r \sigma)$ words, where $\sigma$ was the size of the alphabet, but we have reduced this to $O(r)$. In the future we plan to implement our last result and use it for rare disease detection and to build a version of BWA-MEM \cite{17} that works with entire genomic databases.

2. Preliminaries

For the sake of brevity, we assume the reader is familiar with LZ77 (we consider the original version, with which phrases end with mismatched characters), matching statistics, the BWT and RLBWT and how to search with them, etc. In this section we first review how to update a standard BWT or RLBWT when a character is prepended to the text. We also describe our simplification of Policriti and Prezza’s augmented RLBWT.

2.1. Updating an RLBWT

We consider constructing RLBWT while reading $T$ from right to left because updating RLBWTs with prepending a character is easier than appending a character. Suppose we have an RLBWT for $T[i + 1..n]$, where $\$$(T)$ does not occur in $T$, and know the position $k$ of $\$$(T)$ in the current BWT. To obtain an RLBWT for $T[i..n]$, we compute rank$_{T[i]}(k)$ and use it to compute the position $k'$ to which $\$$(T)$ will move. We replace $\$$(T)$ by $T[i]$ in the RLBWT, which may require merging that copy of $T[i]$ with the preceding run, the succeeding run, or both. We then insert $\$$(T)$ at BWT$[k']$, which may require splitting a run. Updating the RLBWT for the reversed string $T^R$ of $T$ is symmetric when we append a character to $T$. Ohno et al. \cite{18} gave a practical implementation that works in $O(r)$ space and supports updates and backward searches in $O(\log r)$ time per character in the pattern.

Lemma 1 (see, e.g., \cite{18}). We can build an RLBWT for $T^R$ incrementally, starting with the empty string and iteratively prepending $T[1], \ldots, T[n]$ — so that after $i$ steps we have an RLBWT for $(T[1..i])^R$ — using a total of $O(n \log r)$ time. Backward searches always take $O(\log r)$ time per character in the pattern.
2.2. Refining the Toehold Lemma

Policriti and Prezza augmented the RLBWT to store the SA entries $SA[i]$ and $SA[j]$ that are the positions in the text of the first and last characters in each run $BWT[i..j]$. They showed how, with this extra information, a backward search for a pattern can be made to return the location of one of its occurrence (assuming it occurs at all).

We can simplify and strengthen Policriti and Prezza’s result slightly, storing only the position of the first character of each run and finding the starting position of the lexicographically first suffix starting with a given pattern. When we start a backward search for a pattern $P[1..m]$, the initial interval is all of $BWT[1..n]$ and we know $SA[1]$ since $BWT[1]$ must be the first character in a run. Now suppose we have processed $P[i..m]$, the current interval is $BWT[j..k]$ and we know $SA[j]$. If $BWT[j] = P[i - 1]$ then the interval for $P[i-1..m]$ starts with $BWT[LF(j)]$, where $LF(j) = SA^{-1}[SA[j] - 1]$ is the last-to-first mapping and can be computed as usual with a rank query, and so we know $SA[LF(j)] = SA[j] - 1$. Otherwise, the interval for $P[i-1..m]$ starts with $BWT[LF(j')]$, where $j'$ is the position of the first occurrence of $P[i-1]$ in $BWT[j..k]$; since $BWT[j']$ is the first character in a run, $j'$ is easy to compute and we have $SA[j']$ stored and can thus compute $SA[LF(j')] = SA[j'] - 1$.

**Lemma 2.** We can augment an RLBWT with $O(r)$ words, where $r$ is the number of runs in the BWT, such that after each step in a backward search for a pattern, we can return the starting position of the lexicographically first text suffix prefixed by the suffix of the pattern we have processed so far.

Generalizing a little bit the above trick, we get the following argument, which will be used to support online update of augmented RLBWTs.

**Lemma 3.** Suppose we have the augmented RLBWT for $T$, which allows us to access the SA entry for the first character of every run. If we know $j = SA[k + 1]$ for some position $k$, we can compute, for any character $c$, the text position $j'$ such that $T[j'..]$ is the lexicographically smallest suffix that is larger than $cT[SA[k]..]$ (if such $T[j'..]$ exists).

**Proof.** Let $i = SA[k]$ We consider two cases depending on whether $BWT[k + 1..]$ contains $c$ or not.

- If $BWT[k + 1..]$ contains $c$: Let $p$ be the smallest position such that $BWT[p] = c$ in $BWT[k + 1..]$. Then it holds that $cT[SA[p]..] = T[j'..]$, 

namely, \( j' = \text{SA}[p] - 1 \). If \( p = k + 1 \), we have \( j = \text{SA}[p] \) by the assumption. Otherwise, \( p \) must be the first position of a \( c \)'s run, and thus, we have \( \text{SA}[p] \) stored.

- If \( \text{BWT}[k + 1..] \) does not contain \( c \): Let \( c' \) be the lexicographically smallest character that appears in \( T \) and is larger than \( c \). If such \( c' \) does not exist, it means \( cT[\text{SA}[k]..] \) is larger than the lexicographically largest suffix of \( T \), and thus, \( T[j'..] \) does not exist. If \( c' \) exists, then it holds that \( c'T[\text{SA}[p]..] = T[j'..] \) and \( j' = \text{SA}[p] - 1 \), where \( p \) is the smallest position such that \( \text{BWT}[p] = c' \). Apparently \( p \) corresponds to the first position of a run, and thus, we have \( \text{SA}[p] \) stored.

Finally we remark that if \( k \) is the last position of \( \text{BWT} \) (namely \( k + 1 \) is out of bounds), we can obtain \( j' \) without \( j \), proceeding as in the second case.

3. Dynamizing the \( r \)-index

In order to locate all the occurrences of pattern \( P \), we have to retrieve \( \text{SA}[i..j] \) (all of which may not be stored explicitly), where \([i..j]\) is the interval for \( P \). If we can efficiently compute \( \text{SA}[k + 1] \) from a given value \( \text{SA}[k] \) for any \( 1 \leq k < n \), then \( \text{SA}[i..j] \) can be retrieved incrementally from \( \text{SA}[i] \), which we get during a backward search by Lemma 2. Gagie, Navarro and Prezza [13] showed how to solve this subproblem. Let \( B \) be the set of pair \( (\text{SA}[k'], \text{SA}[k' + 1]) \) of text positions such that \( k' \) and \( k' + 1 \) are on a run's boundary, i.e., \( k' \) is the last position of some run of \( \text{BWT} \) and \( k' + 1 \) is the first position of the next run. Consider a predecessor data structure to support the following query: for any text position \( p \) of \( T \), \( \text{pred}_B(p) \) returns \((x, y) \in B \) such that \( x \) is the largest possible with \( x \leq p \). Then, the next lemma holds.

Lemma 4 ([13]). For any \( 1 \leq k < n \), \( \text{SA}[k + 1] = y + \text{SA}[k] - x \) holds, where \( \text{pred}_B(\text{SA}[k]) = (x, y) \).

Proof. By the definition of \( x \), for any \( 0 \leq d < \text{SA}[k] - x \), \( \text{BWT}[\text{LF}^d(k)] \) does not correspond to the end of a run while \( \text{BWT}[\text{LF}^d(\text{SA}[k] - x)(k)] \) does. This means that the suffixes \( T[\text{SA}[k]..] \) and \( T[\text{SA}[k + 1]..] \) are both preceded by the same string of length \( \text{SA}[k] - x \), and implies that for any \( 0 \leq d' \leq \text{SA}[k] - x \) the suffixes \( T[x + d'..] \) and \( T[y + d'..] \) are lexicographically adjacent. By setting \( d' = \text{SA}[k] - x \), we see that the lexicographically next suffix of \( T[\text{SA}[k]..] \) is \( T[y + \text{SA}[k] - x..] = T[\text{SA}[k + 1]..] \), from which the statement immediately follows. \( \square \)
In this paper, we show that the $r$-index can be constructed in an online manner while reading text from right to left (or symmetrically appending characters to $T$ but constructing the RLBWT for $T^R$). Let $r$ be the number of runs in the BWT string for the current text $T$. Our online $r$-index maintains:

- a data structure to compute LF in $O(\log r)$ time,
- a data structure to compute $\text{pred}_B$ and insertion/deletion of new element to/from $B$ in $O(\log r)$ time (using a standard balanced search tree), and
- a data structure to get, for each run of BWT, the SA entry for the “last” character of the run.

Note that by combining the last two data structures we can retrieve the SA entry for the first character of a run, and thus, we essentially have an access to the SA entries for the first and last character of every run.

Let $k$ be the position of $\$$ in the current BWT. Since $k - 1$ and respectively $k + 1$ are corresponding to last and first positions of runs (unless they are out of bounds of BWT), we have $\text{SA}[k - 1]$ and $\text{SA}[k + 1]$. When we prepend $c$ to $T$, we first replace $\$$ with $c$, which might cause a merging of runs with the preceding run, the succeeding run, or both. As we have $\text{SA}[k - 1]$, $\text{SA}[k]$ and $\text{SA}[k + 1]$, we can properly update the data structures. Next we update LF and insert $\$$ into the new position $k' = \text{LF}(k)$. If $k'$ is on a run’s boundary, we need to update the data structures storing SA entries. In particular, when the insertion causes splitting a run, we need to know the $\text{SA}[k' - 1]$ and $\text{SA}[k' + 1]$, which might not be stored explicitly. Notice that $T[\text{SA}[k' + 1]..]$ is the lexicographically smallest suffix that is larger than new $cT$. Since we have SA entry for the first character of a run and $\text{SA}[k + 1]$, we can use Lemma 3 to compute $\text{SA}[k' + 1]$. In a symmetric way, $\text{SA}[k' - 1]$ can be also obtained.

The information is enough to deal with the changes of SA entries to be stored along with the insertion of $\$$ at $k'$.

### 3.1. Experimental results

We implemented in C++ our online construction of $r$-index (the source code is available at [19]) and compared its performance with offline $r$-indexes. The implementation of offline $r$-index is taken from [20], which has three options for construction:

- sesais: Use a space efficient suffix sorting of [21].
• **divsuf**: Use a suffix sorting of [22], which is faster than sesais but uses more space.

• **bigbwt**: Use a prefix-free parsing technique, which is shown to be useful to reduce the working space and accelerate BWT construction [14, 15].

Our online r-index is implemented based on the online RLBWT proposed in [18], which runs fast but uses $2r \log r$ bits to support rank queries (which is slightly costly compared to existing and offline variants). All the experiments were conducted on a 6core Xeon E5-1650V3 (3.5GHz) machine using a single core with 32GiB memory running Linux CentOS7.

We tested on the datasets used in [13] (and available in [23]).

• **DNA**: A pseudo-real DNA sequence consisting of 629145 copies of a DNA sequence of length 1000 where each character was mutated with probability $10^{-3}$.

• **boost**: concatenated versions of the GitHubs boost library.

• **einstein**: concatenated versions of Wikipedia’s article for Einstein.

• **worldLeaders**: a collection of all pdf files of CIA World Leaders from January 2003 to December 2009 from repcorpus.

We also tested on real genomic datasets obtained by concatenating up to 50 versions of chromosome 19. Let chr19\_x denote the dataset containing x versions. Following the setting of [15], we removed all characters besides A, C, G, T and N from the sequences in advance and delimited each sequence in chr19\_x by a line break.

Table 1 shows the statistics of the datasets. Note that r is the number of runs in the BWT for a “reversed” input string, which is produced by our online variant. Since offline variants create BWTs for input strings read in a standard order, the values of r are slightly different.

Table 2 shows the comparison in construction time and working space for DNA, boost, einstein and worldLeaders, where bigbwt halted with errors due to some unknown reason for non-genomic datasets. It shows that online runs in reasonable time while working in compressed space.

Figure 2 shows how the construction time and working space increase when the collection of chr19 sequences grows. At the point of chr19\_50, sesais used up 32GiB memory, and we expect that divsuf will do sooner
Table 1: Statistics of datasets, where $\sigma$ is the alphabet size, $n$ is the text length, $r$ is the number of runs.

| dataset         | $\sigma$ | $n$         | $r$         | $n/r$  |
|-----------------|----------|-------------|-------------|--------|
| DNA             | 10       | 629,140,006 | 1,288,876   | 488    |
| boost           | 96       | 629,145,600 | 60,281      | 10,437 |
| einstein        | 194      | 629,145,600 | 964,973     | 652    |
| worldleaders    | 89       | 46,968,181  | 583,395     | 81     |
| chr19_1         | 6        | 59,128,984  | 30,660,114  | 2      |
| chr19_10        | 6        | 591,254,545 | 32,225,116  | 18     |
| chr19_30        | 6        | 1,773,750,965 | 33,617,233 | 53     |
| chr19_50        | 6        | 3,015,374,692 | 34,688,812 | 87     |

Figure 2: Increase of build time and space of $r$-indexes.

or later. On the other hand, online and bigbwt show a potential to handle more sequences. The throughput of online is about 0.8 MiB / sec. Since $r$ grows very slowly as sequences increase (see Table 1), we expect that the performance of online (both in terms of throughput and working space) is kept even when more sequences are added. Hence we conclude that online and bigbwt complement each other, i.e., bigbwt can construct the $r$-index in a batch very efficiently, and after that, online can handle incrementally added sequences.

Before finalizing the construction, our online $r$-index is always ready for answering count/locate queries as well as updating. We tested the performance of count/locate comparing with the finalized $r$-index (i.e., offline). For each dataset, we fed 1000 randomly chosen substrings of length 8 as patterns to count/locate. In locating, both programs just list the occurrences (positions)
Table 2: Comparison of online and offline $r$-indexes in construction time and working space.

| dataset          | construction time (sec) and working space (MiB) |
|------------------|-------------------------------------------------|
|                  | online  sesais  divsuf  bigbwt                  |
| DNA              | 284.22  274.77  120.47  100.51                   |
|                  | 41.03    2483    4202    1025                     |
| boost            | 213.00  311.64  106.86  —                         |
|                  | 4.07     2464    4203    —                         |
| einstein         | 268.72  369.37  111.10  —                         |
|                  | 31.45    2466    4204    —                         |
| world_leaders    | 20.21    15.15   4.41     —                        |
|                  | 20.43    253     316     —                         |

in a vector as they find. Table 3 shows the results. Firstly, online takes about four times larger space than offline. Besides the overhead needed to prepare for online updates, this could be attributed to the $2r \log r$ bits used in our base implementation of RLBWT. The results for locate show a tendency that online are about 10% slower than offline. On the other hand in count operations, online sometimes outperformed offline. This probably reflects the difference in the implementation of backward steps; each backward step of online takes $O(\log r)$ time (regardless of the alphabet size) while that of offline takes $O(\log(n/r) + H_0)$ time, where $H_0$ is the zero-order entropy of the run heads.

4. Online LZ77 Parsing

Our first idea is to extend the technique from Subsection 2.1 for updating an unaugmented RLBWT, to apply to an augmented RLBWT. Then we can build an augmented RLBWT for $T^R$ incrementally, starting with the empty string and iteratively prepending $T[1], \ldots, T[n]$. Our second idea is to mix prepending characters to a suffix of $T^R$ with backward searching for a prefix of that suffix, which is equivalent to appending characters to a prefix of $T$ while searching for a suffix of that prefix. In contrast to the $r$-index, we only need to report one occurrence for an LZ77 phrase, and thus, the data structure can be simplified (specifically, $\text{pred}_B$ is not needed).
Table 3: Comparison in count/locate operations.

| dataset            | data structure size (MiB) | count time (µs / pattern) | locate time (µs / occ) |
|--------------------|---------------------------|---------------------------|------------------------|
|                    | online offline             | online offline             | online offline         |
| DNA                | 41.03                     | 13.13                     | 4.54                   |
|                    |                           |                           |                        |
| boost              | 4.07                      | 0.76                      | 5.05                   |
|                    |                           |                           |                        |
| einstein           | 31.45                     | 10.29                     | 7.35                   |
|                    |                           |                           |                        |
| world_leaders      | 20.43                     | 5.37                      | 6.90                   |
|                    |                           |                           |                        |
| chr19_1            | 931                       | 233                       | 12.16                  |
| chr19_10           | 1037                      | 298                       | 10.28                  |
| chr19_30           | 1117                      | 330                       | 10.35                  |
| chr19_50           | 1162                      | 355                       | 10.26                  |

4.1. Updating an augmented RLBWT

Recall that the augmented RLBWT of Subsection 2.2 has the SA entry for the first character of every run. Suppose we have an augmented RLBWT for $T[i + 1..n]$, where $\$ does not occur in $T$, and know the position $k$ of $\$ in the current BWT. Since $k + 1$ corresponds to the first position of a run (unless it is out of bounds of BWT), we have SA[$k + 1$]. To obtain the augmented RLBWT for $T[i..n]$, we move $\$ to the new position $k'$ as always for updating BWTs (see Subsection 2.1). When we insert $\$ to $k'$, it may require splitting a run, which requires us to know the SA[$k' + 1$] for the new created run. Since we have SA[$k + 1$] and SA entry for the first character of a run, we can compute SA[$k' + 1$] by Lemma 3. Updating the augmented RLBWT for $T^R$ is symmetric when we append a character to $T$. We can extend Ohno et al.’s implementation to support updates to the augmented RLBWT for $T^R$ in $O(n \log r)$ time and backward searches still in $O(\log r)$ time per character in the pattern.

Lemma 5. We can build an augmented RLBWT for $T^R$ incrementally, starting with the empty string and iteratively prepending $T[1], \ldots, T[n]$ — so that after $i$ steps we have an RLBWT for $(T[1..i])^R$ — using a total of $O(n \log r)$ time. Backward searches always take $O(\log r)$ time per character in the pattern.
4.2. Computing the parse

Suppose we currently have an augmented RLBWT for \((T[1..j])^R\) and the following information:

- the phrase containing \(T[j+1]\) in the LZ77 parse of \(T\) starts at \(T[i]\);
- the non-empty interval \(I\) for \((T[i..j])^R\) in the BWT for \((T[1..j-1])^R\);
- the position in \((T[1..j-1])^R\) of the first character in \(I\);
- the interval \(I'\) for \((T[i..j+1])^R\) in the BWT for \((T[1..j])^R\);
- the position in \((T[1..j])^R\) of the first character in \(I'\), if \(I'\) is non-empty.

If \(I'\) is empty, then the phrase containing \(T[j+1]\) is \(T[i..j+1]\) with \(T[j+1]\) being the mismatch character, and we can compute the position of an occurrence of \(T[i..j]\) in \(T[1..j-1]\) from the position of the first character in \(I\). We then prepend \(T[j+1]\) to \((T[1..j])^R\), update the augmented RLBWT, and start a new backward search for \(T[j+1]\).

If \(I'\) is non-empty, then we know the phrase containing \(T[j+2]\) starts at \(T[i]\), so we prepend \(T[j+1]\) to \((T[1..j])^R\), update the augmented RLBWT (while keeping track of the endpoints of \(I'\)), and perform a backward step for \(T[j+2]\) to obtain the interval \(I''\) for \((T[i..j+2])^R\) in the BWT for \((T[1..j+1])^R\). If \(I''\) is non-empty, the augmented RLBWT returns the position in \((T[1..j+1])^R\) of the first character in \(I''\).

Continuing like this, we can simultaneously incrementally build the augmented RLBWT for \(T^R\) while parsing \(T\). Each step takes \(O(\log r)\) time and we use constant workspace on top of the augmented RLBWT, which always contains at most \(r\) runs, so we use \(O(r)\) space. This gives us the following result:

**Theorem 6.** We can compute the LZ77 parse for \(T[1..n]\) online using \(O(n \log r)\) time and \(O(r)\) space, where \(r\) is the number of runs in the BWT for \(T^R\).

4.3. Experimental results

We implemented in C++ the online LZ77 parsing algorithm of Theorem 6 (the source code is available at [19]). We evaluate the performance of our method by comparing with the state-of-the-art implementations for LZ77 parsing that potentially can work in the peak RAM usage smaller than
\(n \log \sigma + n \log n\) bits. A brief explanation and setting of each method we tested is the following:

- **LZscan** [24, 25]. It runs in \(O(n d \log(n/d))\) time and \((n/d) \log n\) bits in addition to the input string, where \(d\) is a parameter that can be used to control time-space tradeoffs. We set \(d\) so that \((n/d) \log n\) is roughly half of the input size.

- **h0-lz77** [26, 27]. Online LZ77 parsing based on BWT running in \(O(n \log n)\) time and \(n H_0 + o(n \log \sigma) + O(\sigma \log n)\) bits of space. The current implementation runs in \(O(n \log n \log \sigma)\) time.

- **rle-lz77-1** [28, 27]. Offline LZ77 parsing algorithm based on RLBWT with two sampled suffix array entries for each run. In theory it runs in \(O(n \log r)\) time and \(2r \log n + r \log \sigma + o(r \log \sigma) + O(r \log(n/r) + \sigma \log n)\) bits of working space. The current implementation runs in \(O(n \log r \log \sigma)\) time.

- **rle-lz77-2** [28, 27]. Offline LZ77 parsing algorithm based on RLBWT that theoretically runs in \(O(n \log r)\) time and \(z \log n + lg z + r \log \sigma + o(r \log \sigma) + O(r \log(n/r) + \sigma \log n)\) bits of working space. The current implementation runs in \(O(n \log r \log \sigma)\) time.

- **rle-lz77-o** [Theorem 6]. To accomplish the parsing done in a reasonable time, our online RLBWT implementation is based on [18], which runs faster (actually in \(O(n \log r)\) time) than [27] but needs \(2r \log r\) extra bits. Online LZ77 parsing can be done in \(O(n \log r)\) time and \(2r \log r + r \log n + O(r \log(n/r) + \sigma \log n)\) bits of working space.

For the above methods other than rle-lz77-2, the output space is not counted in the working space since they compute LZ77 phrases sequentially. On the other hand, rle-lz77-2 counts \(z \log n\) bits of working space to store the starting positions of the phrases as they are not computed sequentially.

We tested on highly repetitive datasets in repcorpus\(^4\) well-known corpus in this field, and some larger datasets created from git repositories. For the latter, we use the script [29] to create 1024MiB texts (obtained by concatenating

\(^4\)See [http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf](http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf) for statistics of the datasets.
source files from the latest revisions of a given repository, and truncated to be 1024MiB) from the repositories for boost\(^5\) samtools\(^6\) and sdsllite\(^7\) (all accessed at 2017-03-27). The programs were compiled using g++6.3.0 with -Ofast -march=native option. The experiments were conducted on a 6core Xeon E5-1650V3 (3.5GHz) machine using a single core with 32GiB memory running Linux CentOS7.

In Table 4, we compare our method \texttt{rle-lz77-o} with \texttt{rle-lz77-2}, which is the most relevant to our method as well as the most space efficient one. The result shows that our method significantly improves the running time while keeping the increase of the space within 4 times. It can be observed that the working space of \texttt{rle-lz77-o} gets worse as the input is less compressible in terms of RLBWT (especially for Escherichia_Coli).

Figure 3 compares all the tested methods for some selected datasets. It shows that \texttt{rle-lz77-o} exhibits an interesting time-space tradeoff: running in just a few times slower than LZscan while working in compressed space. After the conference version of this paper was published, an extended experiment was conducted in [18] where two versions are added for testing the performance of their RLBWT construction mixed with \texttt{rle-lz77-1} and \texttt{rle-lz77-2}. The results show that \texttt{rle-lz77-o} is still outstanding almost dominating those two versions.

5. Matching Statistics

The matching statistics of \(S[1..m]\) with respect to \(T\) tell us, for each suffix \(S[i..m]\) of \(S\), what is the length \(\ell_i\) of the longest substring \(S[i..i+\ell_i-1]\) that occurs in \(T\) and the position \(p_i\) of one of its occurrences there. We can compute \(\ell_i\) and \(p_i\) using an RLBWT for \(T^R\) with SA entries stored at the beginnings of runs, by performing a backward search for each \((S[i..m])^R\) — i.e., performing a backward step for \(S[i]\), then another for \(S[i+1]\), etc. — until the interval in the BWT becomes empty, and then undoing the last backward step. However, to compute all the matching statistics this way takes time proportional to the sum of all the \(\ell\) values — which can be quadratic in \(m\) — times the time for a backward step.

\(^5\)https://github.com/boostorg/boost
\(^6\)https://github.com/samtools/samtools
\(^7\)https://github.com/simongog/sdsl-lite
Suppose we use Policriti and Prezza’s augmented RLBWT for $T$ (which stores the positions in $T$ of both the first and last character of each run) to perform a backward search for $S$ — i.e., performing a backward step for $S[m]$, then another for $S[m-1]$, etc. — until the interval in the BWT becomes empty, and then undo the last backward step. This gives us the last few $\ell$ and $p$ values in the matching statistics for $S$, and the interval $\text{BWT}[i..j]$ for some suffix $S[k..m]$ of $S$ such that $S[k-1..m]$ does not occur in $T$ (meaning $S[k-1]$ does not occur in $\text{BWT}[i..j]$). Consider the suffixes of $T$ starting with the occurrences of $S[k-1]$ preceding $\text{BWT}[i]$ and following $\text{BWT}[j]$ in the BWT, which are the last and first characters in runs, respectively. By the definition of the BWT, one of these two suffixes has the longest common prefix (LCP) with $S[k-1..m]$ — and, equivalently, with $S[k-1]T[p_k..n]$ — of all the suffixes of $T$. Therefore, if we know which of those two suffixes has the longer common prefix with $S[k-1]T[p_k..n]$, we can deduce $p_{k-1}$.
Our first idea is to further augment Policriti and Prezza’s RLBWT such that, for any position \( i \) in the BWT and any character \( c \), we can tell whether \( cT[SA[i]] \) has a longer common prefix with the suffix of \( T \) starting with the occurrence of \( c \) preceding \( \text{BWT}[i] \), or with the one starting with the occurrence of \( c \) following \( \text{BWT}[i] \). Although it sounds at first as if this should use \( \Omega(n) \) space, in fact it takes constant space per run in the BWT. With this information, we can compute the \( p \) values for the matching statistics, using a right-to-left pass over \( S \).

Once we have the \( p \) values, we use a left-to-right pass over \( S \) to compute the \( \ell \) values. Notice that it would again take time at least proportional to the sum of the \( \ell \) values, to start at each \( T[p] \) and extract characters until finding a mismatch. Since \( \ell_{i+1} \) cannot be less than \( \ell_i - 1 \), however, if we have a compact data structure that supports \( O(\log \log n) \)-time random access to \( T \) — such as a Tabix index \([30]\), the RLZ parse implemented with a y-fast trie or a bitvector \([31, 32]\), or a variant of that approach adapted for VCF — then we can compute all the \( \ell \) values in \( O(m \log \log n) \) total time using small space. Since the size of the RLZ parse is generally comparable to that of the RLBWT when there is a natural reference sequence (which is the case when dealing with databases of genomes from the same species) and most genomic databases are stored in VCF anyway, using random access to \( T \) seems unlikely to be an obstacle in practice.

5.1. Further augmentation

For each consecutive pair of runs \( \text{BWT}[g..h] \) and \( \text{BWT}[j..k] \) of a character \( c \), we add to Policriti and Prezza’s augmented RLBWT the threshold position \( i \) between the end \( h \) of the first run and the start \( j \) of the second run such that, for \( h < i' \leq i \), each string \( T[SA[i']..n] \) has a longer common prefix with \( T[SA[h]..n] \) than with \( T[SA[j]..n] \) and, for \( i < i' < j \), each string \( T[SA[i']..n] \) has a longer common prefix with \( T[SA[j]..n] \) than with \( T[SA[h]..n] \). By the definition of the BWT and the SA, the lengths of the longest common prefixes with \( T[SA[h]..n] \) are non-increasing as we go from \( T[SA[h+1]..n] \) to \( T[SA[j]..n] \), and the lengths of the longest common prefixes with \( T[SA[j]..n] \) are non-decreasing; therefore there is at most one such threshold \( i \). This adds a total of \( O(r) \) space (where \( r \) is now the number of runs in BWT of \( T \), not \( T^R \)).

Lemma 7. We can augment an RLBWT for \( T \) with \( O(r) \) words, where \( r \) is the number of runs in the BWT for \( T \), such that for any position \( i \) in the
BWT and any character c, in O(1) time we can tell whether cT[SA[i]] has a longer common prefix with the suffix of T starting with the occurrence of c preceding BWT[i], or with the one starting with the occurrence of c following BWT[i].

Notice that, when we add a new genome to the database, we need to recompute the positions of the thresholds only when characters are inserted in the BWT exactly at those positions or at the beginnings and endings of runs. We are currently working with this property to make this version of the r-index dynamic as well, by augmenting the r-index to support limited LCP queries [33, Section 3.2].

5.2. Algorithm

As we have said, our algorithm consists of first computing all the p values in the matching statistics using a right-to-left pass over S, then computing all the ℓ values using a left-to-right pass. We first choose q to be the position of the first or last character in any run and set t to be its position in T. We then walk backward in S and T until we find a mismatch S[i] ≠ BWT[q], at which point we reset q to be the position of either the copy of S[i] preceding BWT[q] or of the one following it, depending on whether BWT[q] is before or after the threshold position for S[i] in the gap between the preceding and following runs of S[i]. The time is dominated by backward-stepping, which can be made O(log log n) with Policriti and Prezza’s RLBWT, so we use a total of O(m log log n) time. Algorithm 1 shows pseudocode.

Once we have the p values, we make a left-to-right pass over S to compute the ℓ values. We start with S[1] and T[p1] and walk forward, comparing S to T character by character, until we find a mismatch S[1+ℓ1−1] ≠ T[p1+ℓ1−1], and set ℓ1 appropriately. We know ℓ2 ≥ ℓ1 − 1, so S[2..2+ℓ2−2] = T[p2..p2 + ℓ2 − 2] and we can jump directly to comparing S[2+ℓ1−1..2+ℓ2−1] = T[p2+ℓ1−1..m] character by character until we find a mismatch, S[2+ℓ2−1] ≠ T[p2+ℓ2−1], and set ℓ2 appropriately. Continuing like this with O(log log n)-time random access to T, we compute all the ℓ values in O(m log log n) time. Algorithm 2 shows pseudocode. This gives us our second main result:

**Theorem 8.** We can augment an RLBWT for T with O(r) words, where r is the number of runs in the BWT for T, such that later, given S[1..m] and O(log log n)-time random access to T, we can compute the matching statistics for S with respect to T in O(m log log n) time.
Algorithm 1 Computing \( p \) values for the matching statistics of \( S \) with respect to \( T \), using an augmented RLBWT for \( T \). For simplicity we ignore special cases, such as when some character in \( S \) does not occur in \( T \).

\[
\text{procedure computePs}(S) \\
q \leftarrow \text{position of the first or last character in any run} \\
t \leftarrow \text{position of BWT}[q] \text{ in } T \\
\text{for } i \leftarrow m \ldots 1 \text{ do} \\
\quad \text{if } \text{BWT}[q] \neq S[i] \text{ then} \\
\quad \quad \text{if } \text{BWT}[q] \text{ is before the threshold between the preceding and following runs of } S[i] \text{ then} \\
\quad \quad \quad q \leftarrow \text{position of the preceding occurrence of } S[i] \text{ in the BWT} \\
\quad \quad \text{else} \\
\quad \quad \quad q \leftarrow \text{position of the following occurrence of } S[i] \text{ in the BWT} \\
\quad t \leftarrow \text{position of BWT}[q] \text{ in } T \\
p_i \leftarrow t \\
q \leftarrow \text{LF}(q) \\
t \leftarrow t - 1 \\
\]

We note as an aside that, in practice, we need not really store information at both ends of runs of the BWT. If we store information only at the beginning of each run but adapt the data structures of the \( r \)-index to support \( \phi \) queries \([34]\) instead of \( \phi^{-1} \) queries and during a backward search always keep track of the last entry in the current SA interval, then if we need the SA entry for the end of a run we can compute it from the SA entry at the next character (the beginning of a run) and a \( \phi \) query. We must modify the Toehold Lemma again slightly: suppose we have processed \( P[i..m] \), the current interval is \( \text{BWT}[j..k] \) and we know \( \text{SA}[k] \); if \( \text{BWT}[k] \neq P[i - 1] \) then we find the last occurrence \( \text{BWT}[k'] \) of \( P[i - 1] \) in \( \text{BWT}[j..k] \), which is the end of interval; we have \( \text{SA}[k'+1] \) stored, since \( \text{BWT}[k'+1] \) is the beginning of a run, and we can compute \( \text{SA}[k'] \) with a \( \phi \) query. The details of \( \phi \) queries are beyond the scope of this paper, so we refer the reader to the papers on the \( r \)-index that we have cited.

5.3. Application: Rare-disease detection

Each substring \( S[i..i + \ell_i - 1] \) is necessarily a right-maximal substring of \( S \) that has a match in \( T \), but not necessarily a left-maximal one. We can easily post-process the matching statistics of \( S \) in \( O(m) \) time to find the maximal
Algorithm 2 Computing $\ell$ values for the matching statistics of $S$ with respect to $T$, using the $p$ values and random access to $T$. Again, for simplicity we ignore special cases, such as when some character in $S$ does not occur in $T$.

\begin{algorithm}
\begin{procedure}[h]
\textbf{procedure} $\text{computeLs}(S, p_1, \ldots, p_m)$
\begin{algorithmic}
\State $\ell_0 \leftarrow 1$
\For{$i \leftarrow 1 \ldots m$}
\State $\ell_i \leftarrow \ell_{i-1} + 1$
\While{$S[i + \ell_i] = T[p_i + \ell_i]$}
\State $\ell_i \leftarrow \ell_i + 1$
\EndWhile
\EndFor
\end{algorithmic}
\end{procedure}
\end{algorithm}

substrings with matches in $T$: if $\ell_i = \ell_{i+1} + 1$, then we discard $\ell_{i+1}$ and $p_{i+1}$. Similarly, in $O(m)$ time we can find all the minimal substrings of $S$ that have no matches in $T$: for each maximal matching substring of $S$, extending it either one character to the right or one character to the left yields a minimal non-matching substring; assuming each character in $S$ occurs in $T$, this yields all the minimal non-matching substrings of $S$.

Finding all the non-matching substrings of a string relative to a large database of strings has applications to bioinformatics, specifically, in rare-disease discovery. For example, we might want to preprocess a large database of human genomes such that when a patient arrives with an unknown disease we suspect to be genetic, we can quickly find all the minimal substrings of his or her genome that do not occur in the database.

5.4. Application: Extending BWA-MEM to work with genomic databases

BWA-MEM [17] is part of the popular BWA aligner but, unlike standard BWA, it does not try to match entire reads. Instead, it looks for maximal exact matches (MEMs) between reads and reference, and uses those as anchors for the alignment. This approach makes BWA-MEM better suited to handling chimeric reads resulting from structural variation in genomes (i.e., cases in which parts of the genome are arranged differently in different individuals, so the first part of a read matches to one part of the reference but the rest matches somewhere else), as well as longer but more error-prone reads.

We believe that BWA-MEM can benefit even more than Bowtie or BWA from using an entire genomic database as a reference instead of a single genome. Suppose that we store at the beginning and end of each run in the BWT the position in the standard reference that character aligns to. Then, when processing a read with several variations that we have seen before
individually but never all together (which is more likely with longer, third-
generation reads), we can still see quickly if the MEMs all align consistently
to the same region of the standard reference. In contrast, BWA-MEM with a
single reference cannot find matches that span variation sites.

We are currently working to extend the r-index to report the positions
where MEMs align but, even just considering a static version, this could require
significant modification of the construction algorithms, which themselves
are still in development [14 15]. We are optimistic, however: our current
constructions are based on prefix-free parsing, which generates a dictionary
and a parse, and it seems that we can augment the parse slightly (specifically,
with a range-minimum data structure over its LCP array) such that, given
two positions in the SA, we can quickly compute the length of the their LCP.
Our plan is to complete the implementation and demonstration of the r-index
without support for maximal exact matching, and then collaborate with
bioinformaticians to determine what is the best way to add that functionality.

References

[1] H. Bannai, T. Gagie, T. I, Online LZ77 parsing and matching statistics
with RLBWTs, in: Proceedings of the 29th Annual Symposium on
Combinatorial Pattern Matching (CPM), 2018.

[2] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A.
DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al.,
The variant call format and VCFtools, Bioinformatics 27 (15) (2011)
2156–2158.

[3] B. Langmead, S. L. Salzberg, Fast gapped-read alignment with Bowtie
2, Nature methods 9 (4) (2012) 357.

[4] H. Li, R. Durbin, Fast and accurate long-read alignment with burrows–
wheeler transform, Bioinformatics 26 (5) (2010) 589–595.

[5] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM
(JACM) 52 (4) (2005) 552–581.
[6] A. Backurs, P. Indyk, Edit distance cannot be computed in strongly subquadratic time (unless SETH is false), in: Proceedings of the fortieth-seventh annual ACM Symposium on Theory of Computing (STOC), ACM, 2015, pp. 51–58.

[7] V. Cohen-Addad, L. Feuilloley, T. A. Starikovskaya, Lower bounds for text indexing with mismatches and differences, in: Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2019, pp. 1146–1164.

[8] C. P.-G. Consortium, Computational pan-genomics: status, promises and challenges, Briefings in Bioinformatics (2016) bbw089.

[9] T. Gagie, S. J. Puglisi, Searching and indexing genomic databases via kernelization, Frontiers in bioengineering and biotechnology 3 (2015) 12.

[10] D. Valenzuela, T. Norri, N. Välimäki, E. Pitkänen, V. Mäkinen, Towards pan-genome read alignment to improve variation calling, BMC genomics 19 (2) (2018) 87.

[11] T. Gagie, G. Navarro, N. Prezza, On the approximation ratio of Lempel-Ziv parsing, in: Latin American Symposium on Theoretical Informatics, Springer, 2018, pp. 490–503.

[12] A. Policriti, N. Prezza, From LZ77 to the run-length encoded Burrows-Wheeler transform, and back, in: Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM), 2017, pp. 17:1–17:10.

[13] T. Gagie, G. Navarro, N. Prezza, Optimal-time text indexing in BWTruns bounded space, in: Proceedings of the 19th Symposium on Discrete Algorithms (SODA), 2018, pp. 1459–1477.

[14] C. Boucher, T. Gagie, A. Kuhnle, G. Manzini, Prefix-free parsing for building big BWTs, in: Proceedings of the 18th International Workshop on Algorithms in Bioinformatics (WABI), 2018, pp. 2:1–2:16.

[15] A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, G. Manzini, Efficient construction of a complete index for pan-genomics read alignment, in: Proceedings of the 23rd International Conference on Computational Molecular Biology (RECOMB), 2019, to appear.
[16] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows–Wheeler transform, Theoretical Computer Science 387 (3) (2007) 298–312.

[17] H. Li, Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, arXiv preprint arXiv:1303.3997.

[18] T. Ohno, K. Sakai, Y. Takabatake, T. I, H. Sakamoto, A faster implementation of online RLBWT and its application to LZ77 parsing, J. Discrete Algorithms 52 (2018) 18–28.

[19] Online RLBWT. https://github.com/itomomoti/OnlineRlbwt

[20] Taher Mun’s Fork of r-index: https://github.com/alshai/r-index

[21] Sdsl-lite: Succinct Data Structure Library. https://github.com/simongog/sdsl-lite

[22] Yuta Mori. divsufsort: A lightweight suffix array construction algorithm. https://github.com/y-256/libdivsufsort

[23] Nicola Prezza. r-index: the run-length BWT index. https://github.com/nicolaprezza/r-index

[24] J. Kärkkäinen, D. Kempa, S. J. Puglisi, Lightweight Lempel-Ziv parsing, in: Proceedings of the 13th Symposium on Experimental Algorithms (SEA), 2013, pp. 139–150.

[25] LZscan. https://www.cs.helsinki.fi/group/pads/

[26] A. Policriti, N. Prezza, Fast online Lempel-Ziv factorization in compressed space, in: Proceedings of the 22nd Symposium on String Processing and Information Retrieval (SPIRE), 2015, pp. 13–20.

[27] DYNAMIC: dynamic succinct/compressed data structures library. https://github.com/xxsds/DYNAMIC

[28] A. Policriti, N. Prezza, LZ77 computation based on the run-length encoded BWT, Algorithmica.

[29] Get-git-revisions: Get all revisions of a git repository. https://github.com/nicolaprezza/get-git-revisions
[30] H. Li, Tabix: fast retrieval of sequence features from generic TAB-delimited files, Bioinformatics 27 (5) (2011) 718–719.

[31] S. Kuruppu, S. J. Puglisi, J. Zobel, Relative lempel-ziv compression of genomes for large-scale storage and retrieval, in: Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE), 2010, pp. 201–206.

[32] A. J. Cox, A. Farruggia, T. Gagie, S. J. Puglisi, J. Sirén, RLZAP: relative Lempel-Ziv with adaptive pointers, in: Proceedings of the 23rd Symposium on String Processing and Information Retrieval (SPIRE), 2016, pp. 1–14.

[33] T. Gagie, G. Navarro, N. Prezza, Fully-functional suffix trees and optimal text searching in bwt-runs bounded space, CoRR abs/1809.02792.

[34] J. Kärkkäinen, G. Manzini, S. J. Puglisi, Permutated longest-common-prefix array, in: Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM), Springer, 2009, pp. 181–192.
Table 4: Comparison of LZ77 parsing time and working space (WS) between rle-lz77-o (shortened as -o) and rle-lz77-2 (shortened as -2), where |T| is the input size (considering each character takes one byte), z is the number of LZ77 phrases for T and r is the number of runs in RLBWT for T.

| dataset        | |T| (MiB) | z  | r   | time (sec) | WS (MiB) |
|----------------|----------------|--------|-----|-----|------------|---------|
|                |                |        |     |     | -o         | -2      |
|                |                |        |     |     | o-2        |         |
| fib41          | 255.503        | 40     | 42  | 131 | 1334       | 0.065  0.071 |
| rs.13          | 206.706        | 39     | 76  | 111 | 1402       | 0.065  0.072 |
| tm29           | 256.000        | 54     | 82  | 104 | 1889       | 0.065  0.072 |
| dblp.xml.00001.1 | 100.000     | 48,882 |     | 172,195 | 94 | 4754       | 2.694  2.258 |
| dblp.xml.00001.2 | 100.000     | 48,865 |     | 175,278 | 94 | 4786       | 2.744  2.273 |
| dblp.xml.0001.1  | 100.000      | 58,180 |     | 240,376 | 97 | 4823       | 3.791  2.714 |
| dblp.xml.0001.2  | 100.000      | 58,171 |     | 269,690 | 97 | 4804       | 4.253  2.860 |
| dna.001.1       | 100.000       | 198,362 |     | 1,717,162 | 114 | 3951     | 27.537 9.672 |
| english.001.2   | 100.000       | 216,828 |     | 1,436,696 | 112 | 4884     | 23.115 10.177 |
| proteins.001.1  | 100.000       | 221,819 |     | 1,278,264 | 111 | 4288     | 20.481 9.246 |
| sources.001.1   | 100.000       | 178,138 |     | 1,211,104 | 105 | 4886     | 19.524 9.007 |
| cere            | 439.917       | 1,394,808 | 11,575,582 | 737 | 17883 | 199.436 73.154 |
| coreutils       | 195.772       | 1,286,069 | 4,732,794 | 252 | 9996 | 78.414 51.822 |
| einstein.de.txt | 88.461        | 28,226 | 99,833 | 82 | 4098 | 1.606 1.618 |
| einstein.en.txt | 445.963       | 75,778 | 286,697 | 437 | 21198 | 4.675 3.773 |
| Escherichia_Coli | 107.469      | 1,752,701 | 15,045,277 | 233 | 4674 | 255.363 72.527 |
| influenza       | 147.637       | 557,348 | 3,018,824 | 168 | 5909 | 49.319 23.078 |
| kernel          | 246.011       | 705,790 | 2,780,095 | 291 | 12053 | 46.036 28.426 |
| para            | 409.380       | 1,879,634 | 15,635,177 | 734 | 17411 | 272.722 91.515 |
| world_leaders   | 44.792        | 155,936 | 583,396 | 43 | 2002 | 9.092 5.932 |
| boost           | 1024.000      | 20,630 | 63,710 | 925 | 46760 | 1.094 1.344 |
| samtools        | 1024.000      | 158,886 | 562,326 | 1020 | 48967 | 9.445 7.190 |
| sdsl            | 2014.000      | 210,501 | 758,657 | 1010 | 47964 | 12.677 9.138 |