Online LZ77 Parsing and Matching Statistics with RLBWTs

Hideo Bannai
Department of Informatics, Kyushu University, Japan
RIKEN Center for Advanced Intelligence Project, Japan
bannai@inf.kyushu-u.ac.jp
https://orcid.org/0000-0002-6856-5185

Travis Gagie[1]
Diego Portales University and CeBiB, Chile
travis.gagie@gmail.com

Tomohiro I[2]
Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp
https://orcid.org/0000-0001-9106-6192

Abstract

Lempel-Ziv 1977 (LZ77) parsing, matching statistics and the Burrows-Wheeler Transform (BWT) are all fundamental elements of stringology. In a series of recent papers, Policriti and Prezza (DCC 2016 and *Algorithmica*, CPM 2017) showed how we can use an augmented run-length compressed BWT (RLBWT) of the reverse T^R of a text T, to compute offline the LZ77 parse of T in $O(n \log r)$ time and $O(r)$ space, where n is the length of T and r is the number of runs in the BWT of T^R. In this paper we first extend a well-known technique for updating an unaugmented RLBWT when a character is prepended to a text, to work with Policriti and Prezza's augmented RLBWT. This immediately implies that we can build *online* the LZ77 parse of T while still using $O(n \log r)$ time and $O(r)$ space; it also seems likely to be of independent interest. Our experiments, using an extension of Ohno, Takabatake, I and Sakamoto's (IWOCA 2017) implementation of updating, show our approach is both time- and space-efficient for repetitive strings. We then show how to augment the RLBWT further – albeit making it static again and increasing its space by a factor proportional to the size of the alphabet – such that later, given another string S and $O(\log \log n)$-time random access to T, we can compute the matching statistics of S with respect to T in $O(|S| \log \log n)$ time.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Lempel-Ziv 1977, Matching Statistics, Run-Length Compressed Burrows-Wheeler Transform

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.7

Acknowledgements This collaboration started at Shonan seminar 126, "Computation over Compressed Structured Data" and the results from the first part of the paper were proven there.

[1] Supported by FONDECYT Grant Number 1171058.
[2] Supported by JSPS KAKENHI Grant Number JP16K16009.

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 7; pp. 7:1–7:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Indexes based on the Burrows-Wheeler Transform [4, 8] (BWT) have been in popular use for over a decade, particularly in bioinformatics, for dealing with datasets that are moderately large and compressible. Indexes based on the run-length compressed BWT [17] (RLBWT) can achieve drastically better compression on massive and highly repetitive datasets, such as databases of human genomes, but until recently their apparently inability to support fast locating queries in small space stood in the way of their widespread use. There have still been promising results published about RLBWTs, however, such as Policriti and Prezza's [23, 24, 25] recent demonstrations that we can use an augmented RLBWT of the reverse of a text to compute offline the LZ77 parse of the text quickly in space proportional to the number of runs in the BWT of the reversed text; and Ohno, Takabatake, I and Sakamoto's [20] recent practical implementation of an RLBWT that supports efficient prepending of characters. Notably, Policriti and Prezza's work led directly to Gagie, Navarro and Prezza's [10] very recent development of an RLBWT that supports fast locating while still taking space proportional to the number of runs in the BWT.

In this paper we strengthen Policriti and Prezza's result by showing how we can compute *online* the LZ77 parse of T, while still using $O(n \log r)$ time and $O(r)$ space. To do this, we first extend a well-known technique for updating an unaugmented RLBWT when a character is prepended to a text, to work with Policriti and Prezza's augmented RLBWT. This result seems likely to be of independent interest since, as we will show in the full version of this paper, it can be applied to Gagie, Navarro and Prezza's RLBWT as well. Assuming that index will be used to store massive genomic databases to which new genomes will sometimes be added, the cost of even occasional rebuilding could be prohibitive. We have implemented our method of updating Policriti and Prezza's augmented RLBWT on top of Ohno et al.'s implementation of updating an RLBWT, and used it to compute the LZ77 parse online for various datasets. Our experiments show our approach is both time- and space-efficient for repetitive strings.

In the second part of this paper we show how to augment the RLBWT further – albeit making it static again and increasing its space by a factor proportional to the size of the alphabet – such that later, given another string S and $O(\log \log n)$-time random access to T, we can compute the matching statistics of S with respect to T in $O(|S| \log \log n)$ time. We note that there are practical compressed data structures supporting $O(\log \log n)$-time random access to T in theory, that also usually perform well in practice. Matching statistics are a popular tool in bioinformatics and so calculating them is of independent interest [16, 19], but in this case we are motivated by a particular application to rare-disease detection, which involves finding the minimal substrings in a genome that do not occur in a genomic database. Our result is similar to Belazzougui and Cunial's [2, 1] with two notable differences: first, they use succinct space (i.e., $O(n \log \sigma)$ bits, where σ is the alphabet size), whereas we use compressed space, bounded in terms of r; second, they do not consider pointers to longest matches in T as part of the matching statistics of S with respect to T, which we do.

Indexes based on the BWT, not run-length compressed, have been augmented to support functionalities far beyond standard pattern matching, but their "killer app" was DNA assembly. That may not be the case for RLBWTs: high-throughput sequencing is resulting in massive genomic databases and – although assembling genomes using entire databases as references may be interesting for, e.g., pan-genomics [26] – there are many other things we might want to do with those databases and some of them require rich query functionalities. We note that another primary task in pan-genomics, parsing a given genome into the minimum number of phrases that can each be found in a database or, similarly, computing the genomes RLZ [13] of the genome with respect to the database, is also straightforward with an RLBWT. If we want the phrases to be aligned, we can use a combination of the RLBWT and PBWT [6].

There are already compact data structures with rich functionalities, such as straight-line programs [14], CDAWGS [3] and compressed suffix trees [18, 9], but none of them has caught on among practitioners the way BWTs did. At the same time as we try to improve the theoretical and practical time- and space-bounds of those data structures, therefore, we should try to extend the functionalities of the RLBWT (while keeping it practical). Even apart from the specific results we present in this paper, we hope it provides momentum for that effort.

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 mismatch characters), matching statistics, the BWT and RLBWT and how to search with them, etc. In this section we first briefly describe our simplification of Policriti and Prezza's augmented RLBWT (without going into their algorithm for LZ77 parsing) and then we review how to update a standard BWT or RLBWT when a character is prepended to the text.

2.1 Policriti and Prezza's augmented RLBWT

In addition to all the data structures associated with the unaugmented RLBWT for a text, Policriti and Prezza's augmented RLBWT stores the suffix-array entries $\mathrm{SA}[i]$ and $\mathrm{SA}[j]$ that are the positions in the text of the first and last characters in each run $\mathrm{BWT}[i..j]$. 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 $\mathrm{BWT}[1..n]$ and we know $\mathrm{SA}[1]$ since $\mathrm{BWT}[1]$ must be the first character in a run. Now suppose we have processed $P[i..m]$, the current interval is $\mathrm{BWT}[j..k]$ and we know $\mathrm{SA}[j]$. If $\mathrm{BWT}[j] = P[i-1]$ then the interval for $P[i-1..m]$ starts with $\mathrm{BWT}[\mathrm{LF}(j)]$, where $\mathrm{LF}(j) = \mathrm{SA}^{-1}[\mathrm{SA}[j]-1]$ can be found as usual, and so we know $\mathrm{SA}[\mathrm{LF}(j)] = \mathrm{SA}[j]-1$. Otherwise, the interval for $P[i-1..m]$ starts with $\mathrm{BWT}[\mathrm{LF}(j')]$, where j' is the position of the first occurrence of $P[i-1]$ in $\mathrm{BWT}[j..k]$; since $\mathrm{BWT}[j']$ is the first character in a run, j' is easy to compute and we have $\mathrm{SA}[j']$ stored and can thus compute $\mathrm{SA}[\mathrm{LF}(j')]$.

▶ **Lemma 1.** *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 suffix prefixed by the suffix of the pattern we have processed so far.*

2.2 Updating an RLBWT

Suppose we have an RLBWT for $\$T[i+1..n]$, where \$ does not occur in T, and know the position d of \$ in the current BWT. To obtain an RLBWT for $\$T[i..n]$, we compute $\mathrm{rank}_{T[i]}(d)$ and use it to compute $\mathrm{LF}(p)$, where p is the position (which we need not compute) of the occurrence before \$ of $T[i]$ in the current BWT. We replace \$ 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 \$ after BWT[LF(p)], which may require splitting a run. Updating the RLBWT for T^R is symmetric when we append a character to T. Ohno et al. gave a practical implementation that supports updates in $O(r)$ time and backward searches in $O(\log r)$ time per character in the pattern.

▶ **Lemma 2** (see [20]). *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.*

3 Online LZ77 Parsing

Our first idea is to extend the technique from Subsection 2.2 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.

3.1 Updating an augmented RLBWT

Suppose we have an augmented RLBWT for $\$T[i+1..n]$, where \$ does not occur in T, and know the position d of \$ in the current BWT. To obtain an augmented RLBWT for $\$T[i..n]$, we compute $\mathrm{rank}_{T[i]}(d)$ and use it to compute LF(p), where p is the position (which we need not compute) of the occurrence before \$ of $T[i]$ in the current BWT. At this point we perform some calculations that were not necessary in Subsection 2.2 since, if we will split a run when we reinsert \$, we should know the position in T of the character after where we will reinsert it.

If there is a copy of $T[i]$ after \$ in the current BWT, we find the first such copy BWT[q], which must be the first character of a run so we have SA[q] stored. If there is no such copy, then we find the first copy BWT[q] of the smallest character lexicographically larger than $T[i]$, which again must be the first character of a run so we have SA[q] stored. If there is no such character, then BWT[LF(p)] is the last character in the current BWT, in which case we can proceed as in Subsection 2.2.

We replace \$ 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 \$ after BWT[LF(p)], which may require splitting a run. If so, the position in T of the character now after \$ is SA[q] -1. 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 $O(r)$ time and backward searches still in $O(\log r)$ time per character in the pattern.

▶ **Lemma 3.** *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.*

3.2 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 our first main result:

▶ **Theorem 4.** *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.*

3.3 Experimental results

We implemented in C++ the online LZ77 parsing algorithm of Theorem 4 (the source code is available at [21]). We evaluate the performance of our method comparing with the state-of-the-art implementations for LZ77 parsing that potentially can work in the peak RAM usage smaller than $n \lg \sigma + n \lg n$ bits. A brief explanation and setting of each method we tested is the following:

- `LZscan` [12, 15]. It runs in $O(nd \log(n/d))$ time and $(n/d) \lg n$ 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 $(n/d) \lg n$ is roughly half of the input size.
- `h0-lz77` [22, 7]. Online LZ77 parsing based on BWT running in $O(n \log n)$ time and $nH_0 + o(n \log \sigma) + O(\sigma \log n)$ bits of space. The current implementation runs in $O(n \log n \log \sigma)$ time.
- `rle-lz77-1` [25, 7]. 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 $2r \lg n + r \lg \sigma + o(r \lg \sigma) + O(r \lg(n/r) + \sigma \lg n)$ bits of working space. The current implementation runs in $O(n \log r \log \sigma)$ time.
- `rle-lz77-2` [25, 7]. Offline LZ77 parsing algorithm based on RLBWT that theoretically runs in $O(n \log r)$ time and $z(\lg n + \lg z) + r \lg \sigma + o(r \lg \sigma) + O(r \lg(n/r) + \sigma \lg n)$ bits of working space. The current implementation runs in $O(n \log r \log \sigma)$ time.
- `rle-lz77-o` [Theorem 4]. To make the parsing done in a reasonable time, our online RLBWT implementation is based on [20], which runs faster (actually in $O(n \log r)$ time) than [7] but needs $2r \lg r$ extra bits. Online LZ77 parsing can be done in $O(n \log r)$ time and $2r \lg r + r \lg n + O(r \lg(n/r) + \sigma \lg n)$ 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 $z \lg n$ 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[3], well-known corpus in this field, and some larger datasets created from git repositories. For the latter, we use the script [11] to create 1024MiB texts (obtained by concatenating source files from the latest revisions of a given repository, and truncated to be 1024MiB) from the repositories for boost[4], samtools[5] and sdsl-lite[6] (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 1, we compare our method `rle-lz77-o` with `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 `rle-lz77-o` gets worse as the input is less compressible in terms of RLBWT (especially for Escherichia_Coli).

Figure 1 compares all the tested methods for some selected datasets. It shows that `rle-lz77-o` exhibits an interesting time-space tradeoff: running in just a few times slower than `LZscan` while working in compressed space.

4 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 ℓ_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 ℓ_i and p_i using Policriti and Prezza's augmented RLBWT for T^R 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 ℓ values – which can be quadratic in m – times the time for a backward step.

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 ℓ and p values in the matching statistics for S, and the interval 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 BWT$[i..j]$). Consider the suffixes of T starting with the occurrences of $S[k - 1]$ preceding BWT$[i]$ and following 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 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}.

[3] See `http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf` for statistics of the datasets.
[4] `https://github.com/boostorg/boost`
[5] `https://github.com/samtools/samtools`
[6] `https://github.com/simongog/sdsl-lite`

Table 1 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^R.

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.001.2	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	1024.000	210,501	758,657	1010	47964	12.677	9.138

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 $cT[\mathrm{SA}[i]]$ has a longer common prefix with the suffix of T starting with the occurrence of c preceding $\mathrm{BWT}[i]$, or with the one starting with the occurrence of c following $\mathrm{BWT}[i]$. Although it sounds at first like this should use $\Omega(n)$ space, in fact it takes $O(\sigma)$ space per run in the BWT, where σ is the alphabet size. 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 ℓ values. Notice that it would again take time at least proportional to the sum of the ℓ values, to start at each $T[p_i]$ and extract characters until finding a mismatch. Since ℓ_{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 the RLZ parse implemented with a y-fast trie or a bitvector [13, 5] – then we can compute all the ℓ 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, using random access to T seems unlikely to be an obstacle in practice.

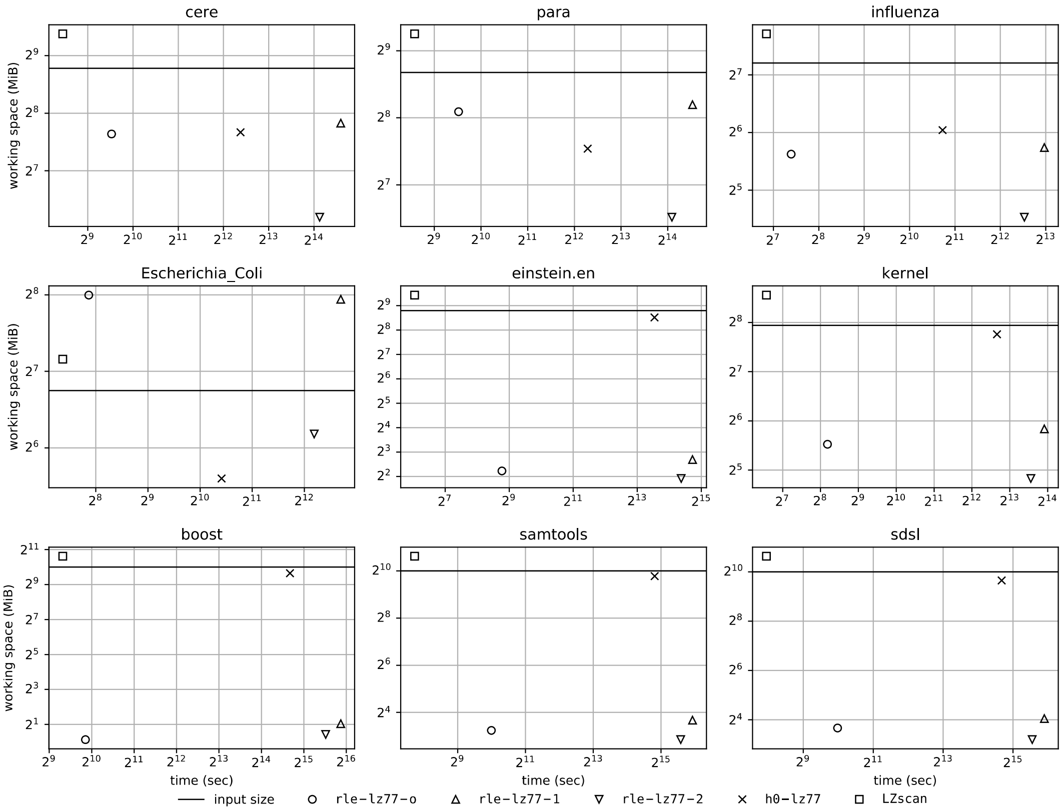

Figure 1 Comparison of LZ77 parsing time and working space.

4.1 Further augmentation

For each run BWT$[i..k]$ and each character c except the one in that run, we add to Policriti and Prezza's augmented RLBWT the threshold position j between i and k such that, for $i \leq i' < j$, each string $cT[\mathrm{SA}[i']..n]$ has a longer common prefix with the suffix of T starting at the copy of c preceding BWT$[i]$ but, for $j \leq k' \leq k$, each string $cT[\mathrm{SA}[k']..n]$ has a longer common prefix with the suffix of T starting at the copy of c following BWT$[k]$. By the definition of the BWT, the length of the longest common prefix with the suffix of T starting at the copy of c preceding BWT$[i]$, is non-increasing as we go from BWT$[i]$ to BWT$[k]$, and the length of the longest common prefix with the suffix of T starting at the copy of c following BWT$[k]$ is non-decreasing; therefore there is at most one such threshold j. Dealing with special cases, such as when even $cT[\mathrm{SA}[i]..n]$ has a longer common prefix with the suffix of T starting at the copy of c following BWT$[k]$, takes a constant number of extra bits, so in total we use $O(r\sigma)$ space for this additional augmentation, where r is now the number of runs in the BWT for T (not T^R).

▶ **Lemma 5.** *We can augment an RLBWT for T with $O(r\sigma)$ words, where r is the number of runs in the BWT for T and σ is the size of the alphabet, such that for any position i in the BWT and any character c, in $O(1)$ time we can tell whether $cT[\mathrm{SA}[i]]$ 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$[i]$.

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.

```
procedure COMPUTEPS($S$)
    $q \leftarrow$ position of the first or last character in any run
    $t \leftarrow$ position of BWT$[q]$ in $T$
    for $i \leftarrow m \ldots 1$ do
        if BWT$[q] \neq S[i]$ then
            if BWT$[q]$ is before the threshold for its run then
                $q \leftarrow$ position of the preceding occurrence of $S[i]$ in the BWT
            else
                $q \leftarrow$ position of the following occurrence of $S[i]$ in the BWT
            $t \leftarrow$ position of BWT$[q]$ in $T$
        $p_i \leftarrow t$
        $q \leftarrow \mathrm{LF}(q)$
        $t \leftarrow t - 1$
```

4.2 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 $S[i] \neq \mathrm{BWT}[q]$, at which point we reset q to be the position of either the copy of $S[i]$ preceding $\mathrm{BWT}[q]$ or of the one following it, depending on whether $\mathrm{BWT}[q]$ is before or after the threshold position for $S[i]$ in its run. 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 $T[p_1]$ and walk forward, comparing S to T character by character, until we find a mismatch $S[1 + \ell_1 - 1] \neq T[p_1 + \ell_1 - 1]$, and set ℓ_1 appropriately. We know $\ell_2 \geq \ell_1 - 1$, so $S[2..2 + \ell_1 - 2] = T[p_2..p_2 + \ell_1 - 2]$ and we can jump directly to comparing $S[2 + \ell_1 - 1..m]$ to $T[p_2 + \ell_1 - 1..m]$ character by character until we find a mismatch, $S[2 + \ell_2 - 1] \neq T[p_2 + \ell_2 - 1]$, and set ℓ_2 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 6.** *We can augment an RLBWT for T with $O(r\sigma)$ words, where r is the number of runs in the BWT for T and σ is the size of the alphabet, 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.*

4.3 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 substrings with matches in T: if $\ell_i = \ell_{i+1} + 1$, then we discard ℓ_{i+1} and p_{i+1}. Similarly, in $O(m)$ time we can find all the minimal substrings

Algorithm 2 Computing ℓ 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.

```
procedure COMPUTELS($S, p_1, \ldots, p_m$)
    $\ell_0 \leftarrow 1$
    for $i \leftarrow 1 \ldots m$ do
        $\ell_i \leftarrow \ell_{i-1} - 1$
        while $S[i + \ell_i] = T[p_i + \ell_i]$ do
            $\ell_i \leftarrow \ell_i + 1$
```

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 Recent and Future Work

We noticed recently (after the submission deadline) that we can easily remove the σ from the space bound in Theorem 6: between two consecutive runs of a character, we need store only a single position where suffixes switch from having a longer common prefix with the suffix following the last character in the earlier run, to having a longer common prefix with the suffix following the first character in the later run. Given a position in the BWT and a character c, we find the preceding and following runs of c's and simply check on which side of the threshold between them the given position lies.

We think we have also found an efficient way to build the augmented RLBWT from Theorem 6, by first storing the length of the longest common prefix (LCP) of the suffixes following the characters on either side of each run boundary in the BWT. To find these LCP values, we start at the position in the BWT of the first character of the text, find the positions in the text of its neighbours in the BWT, and use random access to walk backwards in the text, performing character-by-character comparisons until we know the LCP values. We then move to the position in the BWT of the second character in the text. This time, however, we know the LCP values are not less than the previous LCP values minus 1 each, and we can use random access to skip pairs of characters we already know match, avoiding unnecessary comparisons. Repeating this for each character of the text, from left to right, we calculate all the LCP values in $O(n \log\log n)$ total time, but we store only those at the ends of runs in the BWT. We can then support access to the array of all LCP values at the same time we support access to the suffix array. For each pair of consecutive runs of a character, we scan the LCP array entries between then to find the position of the threshold between them. This takes a total of $O(\sigma n \log\log n)$ time.

We will update Section 4 appropriately in the full version of this paper. Now that we have a reasonable way of constructing the data structure from Theorem 6, we also plan to implement and test it, working toward a prototype for our target application of rare-disease detection.

References

1 Djamal Belazzougui and Fabio Cunial. Fast matching statistics in small space. In *Proceedings of the 17th Symposium on Experimental Algorithms (SEA)*, pages 179–190, 2014.

2 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *Proceedings of the 21st Symposium on String Processing and Information Retrieval (SPIRE)*, pages 179–190, 2014.

3 Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:13, 2017.

4 Michael Burrows and David J. Wheeler. A block sorting lossless compression algorithm. Technical Report 124, DEC, 1994.

5 Anthony J. Cox, Andrea Farruggia, Travis Gagie, Simon J. Puglisi, and Jouni Sirén. RLZAP: relative Lempel-Ziv with adaptive pointers. In *Proceedings of the 23rd Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–14, 2016.

6 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.

7 Dynamic: dynamic succinct/compressed data structures library. URL: `https://github.com/xxsds/DYNAMIC`.

8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

9 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. Technical Report 1705.10382, arXiv.org, 2017.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the 19th Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.

11 get-git-revisions: Get all revisions of a git repository. URL: `https://github.com/nicolaprezza/get-git-revisions`.

12 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proceedings of the 13th Symposium on Experimental Algorithms (SEA)*, pages 139–150, 2013.

13 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceeings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

14 Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

15 Lzscan. URL: `https://www.cs.helsinki.fi/group/pads/`.

16 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

17 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

18 Gonzalo Navarro and Alberto Ordóñez Pereira. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 21(1):1.8:1–1.8:38, 2016.

19 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

20 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler transform. In *Proceedings of the 28th International Workshop on Combinatorial Algorithms (IWOCA)*, 2017. To appear.

21 Online rlbwt. URL: `https://github.com/itomomoti/OnlineRlbwt`.

22 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In *Proceedings of the 22nd Symposium on String Processing and Information Retrieval (SPIRE)*, pages 13–20, 2015.

23 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In *Proceedings of the Data Compression Conference (DCC)*, pages 23–32, 2016.

24 Alberto Policriti and Nicola Prezza. From LZ77 to the run-length encoded Burrows-Wheeler transform, and back. In *Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 17:1–17:10, 2017.

25 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, Jul 2017.

26 Daniel Valenzuela and Veli Mäkinen. CHIC: a short read aligner for pan-genomic references. *bioRxiv*, 2017.