Merging RLBWTs Adaptively

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

We show how to merge two run-length compressed Burrows-Wheeler Transforms (RLBWTs) into a run-length compressed extended Burrows-Wheeler Transform (eBWT) in $O(r)$ space and $O((r + L)\log(m + n))$ time, where m and n are the lengths of the uncompressed strings, r is the number of runs in the final eBWT and L is the sum of its irreducible LCP values.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Burrows-Wheeler Transform, run-length compression, RLBWT, construction, merging

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.16

Funding Funded by NSERC Discovery Grant RGPIN-07185-2020.

1 Introduction

The Burrows-Wheeler Transform (BWT) [8] is an important tool in modern genomics. As DNA sequencing technologies have advanced and researchers have started working with pangenomic references, they have begun using run-length compressed BWTs (RLBWTs). There are good algorithms for building RLBWTs for massive pangenomic references in which all the genomes are similar [4, 14] but the case when the reference contains many genomes from each of many species is still challenging. One possible solution is to build a separate RLBWT for each species and then merge them into a single RLBWT or run-length compressed extended BWT (eBWT) [19].

Merging BWTs is an established research topic [11, 12, 24] and merging RLBWTs [23, 17] is a natural extension of that. As far as we know, however, until recently all algorithms for merging RLBWTs were based on dynamic RLBWTs [22, 2] and used time at least roughly linear in the size of the uncompressed inputs, even if they used compressed space. Díaz-Domínguez et al. [9] gave an algorithm for merging BWTs that can be extended to RLBWTs and works faster when the inputs are individually repetitive but not similar to each other. Because it is based on prefix-free parsing, however, it does not have good worst-case bounds. In this paper we show how to merge two RLBWTs into a run-length compressed eBWT in $O(r)$ space and $O((r + L)\log(m + n))$ time, where m and n are the lengths of the uncompressed strings, r is the number of runs in the final eBWT and L is the sum of the longest common prefix (LCP) values at the beginnings of those runs (known as its irreducible LCP values). It is known that $L \in O((m + n)\log\delta)$ [15], where $\delta \leq r$ is a powerful measure of the eBWT's compressibility [16].

We describe here only how to merge the RLBWTs of two strings. Our algorithm can easily be extended to merge many RLBWTs or run-length compressed eBWTs, but we leave that to the full version of the paper. In Section 2 we review some preliminary concepts. As warm-ups, in Section 3 we present simple algorithms for merging positional BWTs (PBWTs) and RLBWTs. In Section 4 we present our main algorithm and in Section 5 we present a first analysis that yields a time bound of $O((r\log r + L)\log(m + n))$. In Section 6 we give the proof of a technical lemma we use, whose details we refer to in Section 7 when we optimize our algorithm slightly to reduce the $r\log r$ in our time bound to r, so the whole bound becomes $O((r + L)\log(m + n))$.

2 Preliminaries

For the sake of brevity, we assume readers are familiar with run-length compression, the Burrows-Wheeler Transform (BWT), run-length compressed BWTs (RLBWTs), positional BWTs (PBWTs) [10], suffix arrays (SAs), the Ψ function and longest common prefix (LCP) arrays; see Mäkinen et al.'s [18] and Navarro's [20] texts for an introduction. The extended BWT (eBWT) [19] of two strings $S[1..m]$ and $T[1..n]$ is an interleaving $\mathrm{BWT}_{S,T}[1..m+n]$ of the characters $\mathrm{BWT}_S[1..m]$ and $\mathrm{BWT}_T[1..n]$ of S and T. Since BWT_S and BWT_T are subsequences of $\mathrm{BWT}_{S,T}$, if it has r runs then they each have at most r runs. For convenience, we write BWT_S, BWT_T and $\mathrm{BWT}_{S,T}$ to denote both the BWTs and the RLBWTs, stating when they are run-length compressed.

For $1 \leq h \leq m+n$, $\mathrm{BWT}_{S,T}[h]$ has the lexicographically hth smallest context. The contexts of $\mathrm{BWT}_S[i]$ and $\mathrm{BWT}_T[j]$ are

$$\begin{aligned} \mathrm{context}_S(i) &= S[\mathrm{SA}_S[i]..m] \circ S[1..\mathrm{SA}_S[i]-1] \\ \mathrm{context}_T(j) &= T[\mathrm{SA}_T[j]..n] \circ T[1..\mathrm{SA}_T[j]-1] \end{aligned}$$

(unless $\mathrm{SA}_S[i] = 1$, in which case $\mathrm{context}_S(i)$ is just $S[1..m]$, or $\mathrm{SA}_T[j] = 1$, in which case $\mathrm{context}_T(j)$ is just $T[1..n]$), where $\circ$ denotes concatenation. If we assume S and T are each terminated by an end-of-string symbol that occurs nowhere else in S and T then two characters in the same BWT cannot have the same context, and a character in BWT_S and a character in BWT_T have the same context if and only if $S = T$ and the characters are in the same positions (in which case we can break ties arbitrarily without affecting $\mathrm{BWT}_{S,T}$).

We can store a move structure [21] (see also [6]) for the Ψ function Ψ_S of S in $O(r)$ space such that, given BWT_S and position i in it, we can extract $\mathrm{BWT}_S[i]$'s context

$$S[\mathrm{SA}_S[i]..n] \circ S[1..\mathrm{SA}_S[i]-1] = \mathrm{BWT}_S[\Psi_S(i)]\,\mathrm{BWT}_S[\Psi_S^2(i)]\,\mathrm{BWT}_S[\Psi_S^3(i)] \cdots \mathrm{BWT}_S[\Psi_S^n(i)]$$

character by character by iteratively evaluating Ψ_S in $O(\log r)$ time plus constant time per iteration or, equivalently, per character extracted. Brown [5] and Bertram et al. [3] (see also [7]) showed how we can build such a move structure in $O(r \log r)$ time. An analogous result holds for T.

▶ **Lemma 1.** *Given the RLBWTs* BWT_S *and* BWT_T*, in* $O(r \log r)$ *time we can build* $O(r)$*-space data structures for iteratively evaluating the* Ψ *functions for* S *and* T *in* $O(\log r)$ *time plus constant time per iteration.*

It follows that we can compare $\mathrm{context}_S(i)$ and $\mathrm{context}_T(j)$ in $O(\log r)$ time plus time proportional to the length of their longest common prefix. We give a proof of Lemma 1 in Section 6 and then explain in Section 7 how we can pay the $O(\log r)$ overhead only three times during a binary search to find, for example, the smallest value j with $\mathrm{context}_T(j)$ lexicographically larger than $\mathrm{context}_S(i)$, denoted $\mathrm{context}_T(j) \succ \mathrm{context}_S(i)$, rather than paying it at every step of the search.

3 Warm-ups

Suppose we have the PBWTs for two sets of haplotypes from the same species, with the same number of columns (but not necessarily the same number of rows) and with the columns representing the same variation sites, and we want the PBWT for all the haplotypes. For example, consider the PBWTs shown in Figure 1, with the bottom PBWT being the merge of the top two (and the example from [13] with indexing from 1 for consistency with the rest

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1 (1)	1 (5)	0 (1)	1 (1)	1 (9)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	1 (1)	1 (8)	1 (2)	1 (5)
1 (2)	1 (6)	0 (2)	1 (2)	0 (1)	1 (5)	0 (10)	0 (10)	1 (10)	1 (9)	0 (5)	1 (5)	0 (2)	1 (10)	1 (6)
1 (3)	1 (7)	0 (3)	1 (3)	1 (2)	1 (6)	0 (9)	0 (9)	0 (9)	0 (5)	0 (6)	1 (6)	0 (10)	0 (5)	1 (7)
1 (4)	1 (8)	0 (4)	1 (4)	1 (3)	1 (7)	0 (2)	1 (2)	0 (5)	0 (6)	0 (7)	1 (7)	1 (1)	0 (6)	1 (3)
0 (5)	1 (9)	0 (5)	1 (5)	1 (4)	1 (8)	0 (3)	1 (3)	0 (6)	0 (7)	0 (8)	0 (8)	0 (5)	0 (7)	1 (4)
0 (6)	1 (10)	0 (6)	1 (6)	0 (5)	0 (10)	0 (4)	1 (4)	0 (7)	0 (8)	0 (2)	0 (2)	0 (6)	0 (3)	1 (9)
0 (7)	0 (1)	0 (7)	1 (7)	0 (6)	0 (9)	0 (5)	0 (5)	0 (8)	0 (2)	0 (3)	1 (3)	0 (7)	0 (4)	1 (8)
0 (8)	0 (2)	0 (8)	1 (8)	0 (7)	0 (2)	0 (6)	0 (6)	0 (2)	0 (3)	0 (4)	1 (4)	0 (3)	0 (9)	1 (1)
0 (9)	0 (3)	0 (9)	0 (9)	0 (8)	0 (3)	0 (7)	0 (7)	0 (3)	0 (4)	0 (10)	0 (10)	0 (4)	0 (8)	1 (2)
0 (10)	0 (4)	0 (10)	1 (10)	0 (10)	0 (4)	0 (8)	0 (8)	0 (4)	0 (10)	1 (9)	1 (9)	0 (9)	0 (1)	1 (10)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0 (11)	1 (11)	0 (11)	1 (11)	1 (12)	0 (15)	0 (15)	0 (15)	1 (15)	0 (17)	0 (17)	1 (17)	1 (20)	1 (11)	1 (19)
0 (12)	1 (12)	0 (12)	0 (12)	1 (13)	0 (16)	0 (16)	0 (16)	1 (16)	0 (12)	1 (12)	1 (19)	1 (15)	0 (19)	0 (12)
0 (13)	1 (13)	0 (13)	0 (13)	1 (14)	1 (18)	0 (11)	0 (11)	1 (11)	0 (19)	0 (19)	1 (18)	1 (16)	0 (12)	1 (13)
0 (14)	1 (14)	0 (14)	0 (14)	0 (15)	0 (11)	0 (17)	0 (17)	0 (17)	0 (18)	0 (18)	0 (20)	0 (11)	0 (13)	1 (14)
0 (15)	1 (15)	0 (15)	0 (15)	0 (16)	0 (17)	0 (12)	0 (12)	0 (12)	0 (20)	0 (20)	0 (15)	1 (17)	0 (14)	1 (20)
0 (16)	1 (16)	0 (16)	0 (16)	0 (18)	0 (12)	0 (13)	0 (13)	1 (13)	0 (15)	0 (15)	0 (16)	0 (19)	0 (20)	1 (15)
0 (17)	1 (17)	0 (17)	1 (17)	1 (19)	0 (13)	0 (14)	0 (14)	1 (14)	0 (16)	0 (16)	0 (11)	1 (18)	0 (15)	1 (16)
1 (18)	1 (19)	1 (19)	0 (18)	1 (20)	0 (14)	0 (19)	0 (19)	0 (19)	0 (11)	0 (11)	1 (12)	0 (12)	0 (16)	1 (17)
0 (19)	1 (20)	1 (20)	0 (19)	0 (11)	0 (19)	1 (20)	0 (18)	0 (18)	0 (13)	1 (13)	1 (13)	0 (13)	0 (17)	1 (18)
0 (20)	1 (18)	0 (18)	0 (20)	0 (17)	0 (20)	0 (18)	0 (20)	0 (20)	0 (14)	1 (14)	1 (14)	0 (14)	0 (18)	1 (11)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1 (1)	1 (5)	0 (1)	1 (1)	1 (9)	0 (15)	0 (15)	0 (15)	1 (15)	0 (1)	0 (1)	1 (1)	1 (8)	1 (2)	1 (19)
1 (2)	1 (6)	0 (2)	1 (2)	1 (12)	0 (16)	0 (16)	0 (16)	1 (16)	0 (17)	0 (17)	1 (17)	1 (20)	1 (10)	1 (5)
1 (3)	1 (7)	0 (3)	1 (3)	1 (13)	1 (18)	0 (1)	0 (1)	0 (1)	1 (9)	1 (12)	1 (19)	0 (2)	1 (11)	1 (6)
1 (4)	1 (8)	0 (4)	1 (4)	1 (14)	0 (1)	0 (10)	0 (10)	1 (10)	0 (12)	0 (19)	1 (18)	1 (15)	0 (19)	1 (7)
0 (5)	1 (9)	0 (5)	1 (5)	0 (15)	1 (5)	0 (11)	0 (11)	1 (11)	0 (19)	0 (18)	1 (5)	1 (16)	0 (5)	1 (3)
0 (6)	1 (10)	0 (6)	1 (6)	0 (16)	1 (6)	0 (17)	0 (17)	0 (17)	0 (18)	0 (5)	1 (6)	0 (10)	0 (6)	1 (4)
0 (7)	1 (11)	0 (7)	1 (7)	0 (18)	1 (7)	0 (9)	0 (9)	0 (9)	0 (5)	0 (6)	1 (7)	0 (11)	0 (7)	0 (12)
0 (8)	1 (12)	0 (8)	1 (8)	1 (19)	1 (8)	0 (12)	0 (12)	0 (12)	0 (6)	0 (7)	0 (8)	1 (1)	0 (3)	1 (13)
0 (9)	1 (13)	0 (9)	0 (9)	1 (20)	0 (10)	0 (13)	0 (13)	1 (13)	0 (7)	0 (8)	0 (20)	1 (17)	0 (4)	1 (14)
0 (10)	1 (14)	0 (10)	1 (10)	0 (1)	0 (11)	0 (14)	0 (14)	1 (14)	0 (8)	0 (20)	0 (2)	0 (19)	0 (12)	1 (9)
0 (11)	1 (15)	0 (11)	1 (11)	1 (2)	0 (17)	0 (19)	0 (19)	0 (19)	0 (20)	0 (2)	1 (3)	1 (18)	0 (13)	1 (8)
0 (12)	1 (16)	0 (12)	0 (12)	1 (3)	0 (9)	1 (20)	1 (2)	0 (18)	0 (2)	0 (3)	1 (4)	0 (5)	0 (14)	1 (20)
0 (13)	1 (17)	0 (13)	0 (13)	1 (4)	0 (12)	0 (2)	1 (3)	0 (5)	0 (3)	0 (4)	0 (15)	0 (6)	0 (9)	1 (15)
0 (14)	1 (19)	0 (14)	0 (14)	0 (5)	0 (13)	0 (3)	1 (4)	0 (6)	0 (4)	0 (15)	0 (16)	0 (7)	0 (8)	1 (16)
0 (15)	1 (20)	0 (15)	0 (15)	0 (6)	0 (14)	0 (4)	0 (18)	0 (7)	0 (15)	0 (16)	0 (10)	0 (3)	0 (20)	1 (1)
0 (16)	0 (1)	0 (16)	0 (16)	0 (7)	0 (19)	0 (18)	0 (5)	0 (8)	0 (16)	0 (10)	0 (11)	0 (4)	0 (15)	1 (17)
0 (17)	0 (2)	0 (17)	1 (17)	0 (8)	0 (20)	0 (5)	0 (6)	0 (20)	0 (10)	0 (11)	1 (12)	0 (12)	0 (16)	1 (18)
1 (18)	0 (3)	1 (19)	0 (18)	0 (10)	0 (2)	0 (6)	0 (7)	0 (2)	0 (11)	1 (13)	1 (13)	0 (13)	0 (1)	1 (2)
0 (19)	0 (4)	1 (20)	0 (19)	0 (11)	0 (3)	0 (7)	0 (8)	0 (3)	0 (13)	1 (14)	1 (14)	0 (14)	0 (17)	1 (10)
0 (20)	1 (18)	0 (18)	0 (20)	0 (17)	0 (4)	0 (8)	0 (20)	0 (4)	0 (14)	1 (9)	1 (9)	0 (9)	0 (18)	1 (11)

Figure 1 Two PBWTs for 10 haplotypes each **(above)**, and the PBWT for all 20 haplotypes **(below)**. The entries of the prefix arrays are shown in parentheses. Cells are red if their information is for one of the first 10 haplotypes and blue if it is for one of the second 10.

of this paper). The entries of the prefix arrays (showing what haplotype each PBWT entry corresponds to) are shown in parentheses. Cells are red if their information is for one of the first 10 haplotypes and blue if it is for one of the second 10. This makes it easy to see that, by the definition of the PBWT, each column in the merged PBWT is an interleaving of the corresponding columns in the two input PBWTs.

The first column from the left of the merged PBWT is the concatenation of the the first columns of the two input PBWTs. Suppose we have already merged the $(j-1)$st columns of the input PBWTs into the $(j-1)$st column of the merged PBWT, and now we want to merge their jth columns into its jth column. In particular, suppose we know how the $(j-1)$st column of the merged PBWT is divided into maximal blocks of consecutive bits that all come from the same input PBWT. In our example, the blocks in the 9th column have lengths $2, 2, 2, 1, 5, 4, 1, 3$ and alternate between blue and red with the first block being blue.

To compute the jth column of the merged PBWT after computing the $(j-1)$st column, we first consider all the 0s in the $(j-1)$st column and list the next bits in their haplotypes (in the 0s' order in the $(j-1)$st column), then consider all the 1s in the $(j-1)$st column

and list the next bits in their haplotypes (in the 1s' order in the $(j-1)$st column); the jth column of the merged PBWT is the concatenation of the two lists. By induction, the bits in the jth column are in the co-lexicographic order of the preceding prefixes in their haplotypes.

Because the jth columns of the two input PBWTs already contain all those next bits and in the correct order, we can use the input PBWTs instead of the explicit haplotypes. To do this, we start with an empty draft of the jth column of the merged PBWT and scan the $(j-1)$st column of the merged PBWT twice. During the first scan, whenever we see a 0 we copy a bit (the next bit in that 0's haplotype) from the jth column of the input PBWT that is the source of that 0. During the second scan, whenever we see a 1 we copy a bit (the next bit in that 1's haplotype) from the jth column of the input PBWT that is the source of that 1. However, our algorithm may work more efficiently considering blocks rather than literally bit by bit.

Suppose the $(j-1)$st column of the merged PBWT consists of b blocks. For k from 1 to b, if there are c copies of 0 in the kth block in the $(j-1)$st column then we append to our draft of the jth column of the merged PBWT the next c bits of the jth column of the input PBWT that is the source of the bits in the kth block. After that, for k from 1 to b, if there are c copies of 1 in the kth block of the $(j-1)$st column then we append to our draft of the jth column of the merged PBWT the next c bits of the jth column of the input PBWT that is the source of the bits in the kth block.

In our example, there are no copies of 0s in the first block of the 9th column, which is blue, so we append no bits from the 10th column of the blue PBWT; there is 1 copy of 0 in the second block of the 9th column, which is red, so we append 1 bit (0) from the 10th column of the red PBWT (from haplotype (1)); then 1 bit (0) from the blue PBWT (from haplotype (17)); then 1 bit (1) from the red PBWT (from haplotype (9)); then 3 bits (000) from the blue PBWT (from haplotypes (12), (19) and (18)); then 4 bits (0000) from the red PBWT (from haplotypes (5), (6), (7) and (8)); then 1 bit (0) from the blue PBWT (from haplotype (20)); then 3 bits (000) from the red PBWT (from haplotypes (2), (3) and (4)).

There are 2 copies of 1 in the first block of the 9th column, which is blue, so we append 2 bits (00) from the 10th column of the blue PBWT (from haplotypes (15) and (16)); then 1 bit (0) from the from the red PBWT (from haplotype (10)); then 1 bit (0) from the blue PBWT (from haplotype (11)); then no bits from the red PBWT; then 2 bits (00) from the blue PBWT (from haplotypes (13) and (14)). When we are finished, we know the 10th column of the merged PBWT is 00100000000000000000 and we know its blocks have lengths 1, 1, 3, 4, 1, 3, 2, 1, 3; its prefix array is (1) (17) (9) (12) (19) (18) (5) (6) (7) (8) (20) (2) (3) (4) (15) (16) (10) (11) (13) (14).

If a block in the $(j-1)$st column of the merged PBWT overlaps t runs of bits in that column, then computing the numbers of 0s and 1s in that block takes $O(t)$ time. We perform one append operation for the 0s in that block and another for the 1s. If the input PBWTs are run-length compressed and we want only the jth column and its block structure – which is all we need to continue merging the two input PBWTs – but not its prefix array, then each append operation takes time proportional to the number of runs in the substring of bits we append. It follows that we can merge two PBWTs in time $O(r+B)$, where r and B are the total numbers of runs and blocks in all the columns of the merged PBWT, respectively.

▶ **Theorem 2.** *We can merge two PBWTs with run-length compressed columns in $O(r+B)$ time, where r and B are the total numbers of runs and blocks in all the columns of the merged PBWT, respectively.*

Theorem 2 may be of independent interest but we have proven it mainly as a warm-up. As a second warm-up, we now give a simple algorithm for merging RLBWTs that uses $O(r)$ space and $O\left(r \log r + (b \log r + B + \sigma) \log(m+n)\right)$ time, where r is the number of runs in

0	$ATGTAATC$ATGTATC$TATCTAAT	C	1	CAT$GATACAT$GATTAGATA$GATT	A
	$ATGTATC$TATCTAATC$ATGTAAT	C		CAT$GATTAGATA$GATTACAT$GAT	A
1	$GATACAT$GATTAGATA$GATTACA	T	1	CTAATC$ATGTAATC$ATGTATC$TA	T
	$GATTACAT$GATACAT$GATTAGAT	A	0	GATA$GATTACAT$GATACAT$GATT	A
	$GATTAGATA$GATTACAT$GATACA	T		GATACAT$GATTAGATA$GATTACAT	$
1	$TATCTAATC$ATGTAATC$ATGTAT	C		GATTACAT$GATACAT$GATTAGATA	$
0	A$GATTACAT$GATACAT$GATTAGA	T		GATTAGATA$GATTACAT$GATACAT	$
1	AATC$ATGTAATC$ATGTATC$TATC	T	1	GTAATC$ATGTATC$TATCTAATC$A	T
	AATC$ATGTATC$TATCTAATC$ATG	T		GTATC$TATCTAATC$ATGTAATC$A	T
1	ACAT$GATACAT$GATTAGATA$GAT	T	0	T$GATACAT$GATTAGATA$GATTAC	A
	ACAT$GATTAGATA$GATTACAT$GA	T		T$GATTAGATA$GATTACAT$GATAC	A
	AGATA$GATTACAT$GATACAT$GAT	T		TA$GATTACAT$GATACAT$GATTAG	A
	AT$GATACAT$GATTAGATA$GATTA	C	2	TAATC$ATGTAATC$ATGTATC$TAT	C
	AT$GATTAGATA$GATTACAT$GATA	C		TAATC$ATGTATC$TATCTAATC$AT	G
	ATA$GATTACAT$GATACAT$GATTA	G	2	TACAT$GATACAT$GATTAGATA$GA	T
	ATACAT$GATTAGATA$GATTACAT$	G		TACAT$GATTAGATA$GATTACAT$G	A
2	ATC$ATGTAATC$ATGTATC$TATCT	A		TAGATA$GATTACAT$GATACAT$GA	T
	ATC$ATGTATC$TATCTAATC$ATGT	A	2	TATC$TATCTAATC$ATGTAATC$AT	G
	ATC$TATCTAATC$ATGTAATC$ATG	T		TATCTAATC$ATGTAATC$ATGTATC	$
	ATCTAATC$ATGTAATC$ATGTATC$	T		TC$ATGTAATC$ATGTATC$TATCTA	A
	ATGTAATC$ATGTATC$TATCTAATC	$		TC$ATGTATC$TATCTAATC$ATGTA	A
	ATGTATC$TATCTAATC$ATGTAATC	$		TC$TATCTAATC$ATGTAATC$ATGT	A
2	ATTACAT$GATACAT$GATTAGATA$	G		TCTAATC$ATGTAATC$ATGTATC$T	A
	ATTAGATA$GATTACAT$GATACAT$	G		TGTAATC$ATGTATC$TATCTAATC$	A
0	C$ATGTAATC$ATGTATC$TATCTAA	T		TGTATC$TATCTAATC$ATGTAATC$	A
	C$ATGTATC$TATCTAATC$ATGTAA	T	1	TTACAT$GATACAT$GATTAGATA$G	A
	C$TATCTAATC$ATGTAATC$ATGTA	T		TTAGATA$GATTACAT$GATACAT$G	A

Figure 2 The lexicographically sorted cyclic shifts of the concatenation of the three $-terminated strings GATTACAT, GATACAT and GATTAGATA **(in red)**, and of the concatenation of those three strings' $-terminated reverse complements **(in blue)**, with the LCP values shown at block boundaries **(in black)**. The eBWT of the two concatenations is the last column of the matrix, slightly set from the rest. The red subsequence of the eBWT is the BWT of the first concatenation and the blue subsequence is the BWT of the second.

the resulting eBWT, b is the number of maximal blocks of characters in that eBWT that all come from the same input RLBWT, B is the sum of the LCP values at the beginnings of those blocks, σ is the size of the alphabet, and m and n are the lengths of the uncompressed input BWTs.

Suppose we are to merge the RLBWTs $\mathrm{BWT}_S[1..m]$ and $\mathrm{BWT}_T[1..n]$ of strings $S[1..m]$ and $T[1..n]$ into the run-length compressed eBWT $\mathrm{BWT}_{S,T}[1..m+n]$ of S and T together, and we have already built the data structures for Ψ for S and T described in Lemma 1 so that we can compare characters' contexts. For example, if S is the concatenation of the $-terminated strings GATTACAT, GATACAT and GATTAGATA and T is the concatenation of those three strings' $-terminated reverse complements, then

$$\begin{aligned}
\mathrm{BWT}_S &= \mathrm{TAT}^5\mathrm{C}^2\mathrm{G}^4\mathrm{A}^3\$^3\mathrm{A}^3\mathrm{TATA}^2 \\
\mathrm{BWT}_T &= \mathrm{C}^3\mathrm{T}^2\mathrm{A}^2\mathrm{T}^2\$^2\mathrm{T}^6\mathrm{CG}^2\$\mathrm{A}^6 \\
\mathrm{BWT}_{S,T} &= \mathrm{C}^2\mathrm{TATCT}^6\mathrm{C}^2\mathrm{G}^2\mathrm{A}^2\mathrm{T}^2\$^2\mathrm{G}^2\mathrm{T}^3\mathrm{A}^2\mathrm{TA}\$^3\mathrm{T}^2\mathrm{A}^3\mathrm{CGTATG}\$\mathrm{A}^8\,,
\end{aligned}$$

with exponents indicating run lengths, and BWT_S and BWT_T are interleaved in $\mathrm{BWT}_{S,T}$ as shown in Figure 2. In this toy example, $r = 27$, $b = 18$, $B = 18$, $\sigma = 5$ and $m + n = 54$.

We first set i and j to 1, then compare $\mathrm{context}_S(i)$ and $\mathrm{context}_T(j)$ to check whether the first block of characters in $\mathrm{BWT}_{S,T}$ is from BWT_S or BWT_T, and set a Boolean flag to true in the former case and false in the latter. As long as $i \leq m$ and $j \leq n$, if the

Algorithm 1 Pseudocode for merging $\mathrm{BWT}_S[1..m]$ and $\mathrm{BWT}_T[1..n]$ into $\mathrm{BWT}_{S,T}[1..m+n]$.

```
1: i ← 1
2: j ← 1
3: if context_S(i) ≺ context_T(j) then
4:     flag ← true
5: else
6:     flag ← false
7: end if
8: while i ≤ m and j ≤ n do
9:     if flag then
10:        doubling search for the largest value i' with context_S(i') ≺ context_T(j)
11:        output BWT_S[i..i']
12:        i ← i' + 1
13:        flag ← false
14:    else
15:        doubling search for the largest value j' with context_T(j') ≺ context_S(i)
16:        output BWT_T[j..j']
17:        j ← j' + 1
18:        flag ← true
19:    end if
20: end while
21: if flag then
22:    output BWT_S[i..m]
23: else
24:    output BWT_T[j..n]
25: end if
```

flag is true then we use a doubling search in $\mathrm{BWT}_S[i..m]$ to find the largest value i' with $\mathrm{context}_S(i') \prec \mathrm{context}_T(j)$, output $\mathrm{BWT}_S[i..i']$, reset i to $i'+1$, and reset the flag to false. If the flag is false then we use a doubling search in $\mathrm{BWT}_T[j..n]$ to find the largest value j' with $\mathrm{context}_T(j') \prec \mathrm{context}_S(i)$, output $\mathrm{BWT}_T[j..j']$, reset j to $j'+1$, and reset the flag to true. Eventually, when $i = m+1$ or $j = n+1$, we output either $\mathrm{BWT}_T[j..n]$ or $\mathrm{BWT}_S[i..m]$, respectively. The pseudocode for our algorithm is shown in Algorithm 1.

By induction, during each pass through the while loop, we output the next block of $\mathrm{BWT}_{S,T}$, from BWT_S if the flag is true and from BWT_T if the flag is false. If the flag is true then all of the $O(\log m)$ comparisons we perform during the current pass are against $\mathrm{context}_T(j)$, which is in the block from BWT_T we will output during the next pass, so the number of characters we extract for each of those comparisons is at most 1 plus the maximum of the LCP values at the boundaries of that block from BWT_T. If the flag is false then all of the $O(\log n)$ comparisons we perform during the current pass are against $\mathrm{context}_S(i)$, which is in the block from BWT_S we will output during the next pass, so the number of characters we extract for each of those comparisons is at most 1 plus the maximum of the LCP values at the boundaries of the block from BWT_S.

It follows that we can charge all the comparisons to blocks of $\mathrm{BWT}_{S,T}$ such that each block is charged $O(\log(m+n))$ comparisons and for each comparison charged to a block, the number of characters we extract is at most 1 plus the maximum of the LCP values at the boundaries of that block. Since at most σ of those LCP values are 0, this means

we use $O\left(r \log r + (b \log r + B + \sigma) \log(m+n)\right)$ total time, including the $O(r \log r)$ time to build the data structures for comparing contexts and the $O(\log r)$ overhead for each of the $O(b \log(m+n))$ comparisons. Our algorithm is adaptive in the sense that it takes advantage of cases when b and B are small, which is likely when S and T are individually repetitive but dissimilar.

▶ **Theorem 3.** *Given the RLBWTs* $\mathrm{BWT}_S[1..m]$ *and* $\mathrm{BWT}_T[1..n]$*, we can merge them into the run-length compressed eBWT* $\mathrm{BWT}_{S,T}[1..m+n]$ *using* $O(r)$ *space and* $O\left(r \log r + (b \log r + B + \sigma) \log(m+n)\right)$ *time, where* r *is the number of runs in the resulting eBWT,* b *is the number of maximal blocks of characters in that eBWT that all come from the same input RLBWT,* B *is the sum of the LCP values at the beginnings of those blocks,* σ *is the size of the alphabet, and* m *and* n *are the lengths of the uncompressed input BWTs.*

4 Main algorithm

Again, suppose we are to merge the RLBWTs $\mathrm{BWT}_S[1..m]$ and $\mathrm{BWT}_T[1..n]$ of strings $S[1..m]$ and $T[1..n]$ into the run-length compressed eBWT $\mathrm{BWT}_{S,T}[1..m+n]$ of S and T together, and we have already built the data structures for Ψ for S and T described in Lemma 1 so that we can compare characters' contexts. The pseudocode for our algorithm is shown in Algorithm 2.

Assume we have already merged $\mathrm{BWT}_S[1..i-1]$ and $\mathrm{BWT}_T[1..j-1]$ into $\mathrm{BWT}_{S,T}[1..i+j-2]$ correctly. When we first reach the while loop (lines 3–43 in Algorithm 2) $i = j = 1$ so this assumption is trivially true. If $i > m$ then $\mathrm{BWT}_{S,T}[i+j-1..m+n] = \mathrm{BWT}_T[j..n]$ (line 45) and if $j > n$ then $\mathrm{BWT}_{S,T}[i+j-1..m+n] = \mathrm{BWT}_S[i..m]$ (line 47), so assume $i \leq m$ and $j \leq n$ and consider only the while loop. Let k and ℓ be the lengths of the leading runs of $\mathrm{BWT}_S[i..m]$ and $\mathrm{BWT}_T[j..n]$ respectively (lines 4 and 5).

First suppose $\mathrm{BWT}_S[i] = \mathrm{BWT}_T[j]$ (lines 7–21), meaning the leading runs $\mathrm{BWT}_S[i..i+k-1]$ and $\mathrm{BWT}_T[j..j+\ell-1]$ of $\mathrm{BWT}_S[i..m]$ and $\mathrm{BWT}_T[j..n]$ consist of copies of the same character. If there exists a character $\mathrm{BWT}_S[i+k]$ after the run $\mathrm{BWT}_S[i..i+k-1]$ and $\mathrm{context}_S(i+k) \prec \mathrm{context}_T(j+\ell-1)$ (lines 8–11) then $\mathrm{BWT}_S[i+k]$ precedes some non-empty suffix $\mathrm{BWT}_T[j'..j+\ell-1]$ of the run $\mathrm{BWT}_T[j..j+\ell-1]$ in $\mathrm{BWT}_{S,T}$, with j' being the smallest value with $\mathrm{context}_T(j') \succ \mathrm{context}_S(i+k)$. We use binary search to find j' in the range $[j, j+\ell)$ (line 8), output

$$\mathrm{BWT}_S[i..i+k-1] \circ \mathrm{BWT}_T[j..j'-1] = (\mathrm{BWT}_S[i])^{k+j'-j}$$

(line 9) and reset i to $i+k$ (line 10) and j to j' (line 11). Since the characters we output are all equal, we need not consider their order to merge the RLBWTs. (We note, however, that we would need to consider the characters' order if we were also merging the RLBWTs' SAs.) The case when there exists a character $\mathrm{BWT}_T[j+\ell]$ after the run $\mathrm{BWT}_T[j..j+\ell-1]$ and $\mathrm{context}_T(j+\ell) \prec \mathrm{context}_S(i+k-1)$ (lines 13–16) is symmetric.

By the definition of the eBWT, it is impossible for $\mathrm{BWT}_S[i+k]$ to precede a non-empty suffix of the run $\mathrm{BWT}_T[j..j+\ell-1]$ in $\mathrm{BWT}_{S,T}$ and for $\mathrm{BWT}_T[j+\ell]$ to precede a non-empty suffix of the run $\mathrm{BWT}_S[i..i+k-1]$ in $\mathrm{BWT}_{S,T}$ in $\mathrm{BWT}_{S,T}$, since then $\mathrm{BWT}_S[i+k]$ would have to both precede and follow $\mathrm{BWT}_T[j+\ell]$. Therefore, the only remaining case with $\mathrm{BWT}_S[i] = \mathrm{BWT}_T[j]$ is when either $i+k = m+1$ or $\mathrm{BWT}_S[i+k]$ follows the entire run $\mathrm{BWT}_T[j..j+\ell-1]$ in $\mathrm{BWT}_{S,T}$ and either $j+\ell = n+1$ or $\mathrm{BWT}_T[j+\ell]$ follows the entire run $\mathrm{BWT}_S[i..i+k-1]$ in $\mathrm{BWT}_{S,T}$. In this case, we output

$$\mathrm{BWT}_S[i..i+k-1] \circ \mathrm{BWT}_T[j..j+\ell-1] = (\mathrm{BWT}_S[i])^{k+\ell}$$

(line 18) and reset i to $i+k$ (line 19) and j to $j+\ell$ (line 20).

Algorithm 2 Pseudocode for merging $\mathrm{BWT}_S[1..m]$ and $\mathrm{BWT}_T[1..n]$ into $\mathrm{BWT}_{S,T}[1..m+n]$.

```
1: i ← 1
2: j ← 1
3: while i ≤ m and j ≤ n do
4:     k ← length of leading run of BWT_S[i..m]
5:     ℓ ← length of leading run of BWT_T[j..n]
6:     if BWT_S[i] = BWT_T[j] then
7:         if i + k ≤ m and context_S(i + k) ≺ context_T(j + ℓ − 1) then
8:             binary search for the smallest value j′ with context_T(j′) ≻ context_S(i + k)
9:             output BWT_S[i..i + k − 1] ∘ BWT_T[j..j′ − 1]
10:            i ← i + k
11:            j ← j′
12:        else if j + ℓ ≤ n and context_T(j + ℓ) ≺ context_S(i + k − 1) then
13:            binary search for the smallest value i′ with context_S(i′) ≻ context_T(j + ℓ)
14:            output BWT_S[i..i′ − 1] ∘ BWT_T[j..j + ℓ − 1]
15:            i ← i′
16:            j ← j + ℓ
17:        else
18:            output BWT_S[i..i + k − 1] ∘ BWT_T[j..j + ℓ − 1]
19:            i ← i + k
20:            j ← j + ℓ
21:        end if
22:    else
23:        if context_S(i) ≺ context_T(j) then
24:            if context_S(i + k − 1) ≺ context_T(j) then
25:                output BWT_S[i..i + k − 1]
26:                i ← i + k
27:            else
28:                binary search for the smallest value i′ with context_S(i′) ≻ context_T(j)
29:                output BWT_S[i..i′ − 1]
30:                i ← i′
31:            end if
32:        else
33:            if context_T(j + ℓ − 1) ≺ context_S(i) then
34:                output BWT_T[j..j + ℓ − 1]
35:                j ← j + ℓ
36:            else
37:                binary search for the smallest value j′ with context_T(j′) ≻ context_S(i)
38:                output BWT_T[j..j′ − 1]
39:                j ← j′
40:            end if
41:        end if
42:    end if
43: end while
44: if i ≤ m then
45:    output BWT_S[i..m]
46: else if j ≤ n then
47:    output BWT_T[j..n]
48: end if
```

Now suppose $\mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$ (lines 23–42). Because $S[\mathrm{SA}_S[i]-1] = \mathrm{BWT}_S[i]$ (or $S[m] = \mathrm{BWT}_S[i]$ if $\mathrm{SA}_S[i] = 1$ so $S[1..\mathrm{SA}_S[i]-1]$ is empty) and $T[\mathrm{SA}_T[j]-1] = \mathrm{BWT}_T[j]$ (or $T[n] = \mathrm{BWT}_T[j]$ if $\mathrm{SA}_T[j] = 1$ so $T[1..\mathrm{SA}_T[j]-1]$ is empty), $\mathrm{context}_S(i)$ and $\mathrm{context}_T(j)$ differ on their last characters. Therefore, we consider only two cases: when $\mathrm{context}_S(i) \prec \mathrm{context}_T(j)$ (lines 24–31) and when $\mathrm{context}_S(i) \succ \mathrm{context}_T(j)$ (lines 33–40).

Suppose $\mathrm{context}_S(i) \prec \mathrm{context}_T(j)$, meaning $\mathrm{BWT}_S[i]$ precedes $\mathrm{BWT}_T[j]$ in $\mathrm{BWT}_{S,T}$. If $\mathrm{context}_S(i+k-1) \prec \mathrm{context}_T(j)$ then the entire run $\mathrm{BWT}_S[i..i+k-1]$ precedes $\mathrm{BWT}_T[j]$ in $\mathrm{BWT}_{S,T}$, so we output that run (line 25) and reset i to $i+k$ (line 26). Otherwise, some non-empty prefix $\mathrm{BWT}_S[i..i'-1]$ of the run $\mathrm{BWT}_S[i..i+k-1]$ precedes $\mathrm{BWT}_T[j]$ in $\mathrm{BWT}_{S,T}$ and some non-empty suffix $\mathrm{BWT}_S[i'..i+k-1]$ follows it, with i' being the smallest value with $\mathrm{context}_S(i') \succ \mathrm{context}_T(j)$. We use binary search (line 28) to find i' in the range $(i, i+k-1]$, comparing the context of a character in $\mathrm{BWT}_S[i+1..i+k-1]$ to $\mathrm{context}_T(j)$ at each step. We output $\mathrm{BWT}_S[i..i'-1]$ (line 29) and reset i to i' (line 30). The case when $\mathrm{context}_S(i) \succ \mathrm{context}_T(j)$ is symmetric.

We exit the while loop only after having consumed all of at least one of BWT_S and BWT_T. If we have not consumed all of the other, we output the rest of it (lines 44 to 48).

5 First analysis

Building the structures in Lemma 1 takes $O(r \log r) \subseteq O(r \log(m+n))$ time, where r is the number of runs in $\mathrm{BWT}_{S,T}$. Since BWT_S and BWT_T both consist of at most r runs, outputting $\mathrm{BWT}_S[i..m]$ (line 45) or $\mathrm{BWT}_T[j..n]$ (line 47) in run-length compressed form takes $O(r)$ time. Therefore, we can focus on the complexity of the while loop.

▶ **Lemma 4.** *During each pass through the while loop, we output an entire run in* $\mathrm{BWT}_{S,T}$.

Proof. Whenever we reach line 9 we have $\mathrm{context}_S(i+k) \prec \mathrm{context}_T(j')$ so, after we output

$$\mathrm{BWT}_S[i..i+k-1] \circ \mathrm{BWT}_T[j..j'-1] = (\mathrm{BWT}_S[i])^{k+j'-j},$$

the next character we output is $\mathrm{BWT}_S[i+k] \neq \mathrm{BWT}_S[i]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. Symmetrically, after we reach line 14 and output

$$\mathrm{BWT}_S[i..i'-1] \circ \mathrm{BWT}_T[j..j+\ell-1] = (\mathrm{BWT}_S[i])^{i'-i+\ell},$$

the next character we output is $\mathrm{BWT}_T[j+\ell] \neq \mathrm{BWT}_T[j]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. After we reach line 18 and output

$$\mathrm{BWT}_S[i..i+k-1] \circ \mathrm{BWT}_T[j..j+\ell-1] = (\mathrm{BWT}_S[i])^{k+\ell},$$

the next character we output (if there is one) is either $\mathrm{BWT}_S[i+k] \neq \mathrm{BWT}_S[i]$ or $\mathrm{BWT}_T[j+\ell] \neq \mathrm{BWT}_T[j]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. After we reach line 25 and output

$$\mathrm{BWT}_S[i..i+k-1] = (\mathrm{BWT}_S[i])^{k},$$

the next character we output is either $\mathrm{BWT}_S[i+k] \neq \mathrm{BWT}_S[i]$ or $\mathrm{BWT}_T[j] \neq \mathrm{BWT}_S[i]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. Symmetrically, after we reach line 34 and output

$$\mathrm{BWT}_T[j..j+\ell-1] = (\mathrm{BWT}_T[j])^{\ell},$$

the next character we output is either $\mathrm{BWT}_T[i+\ell] \neq \mathrm{BWT}_T[j]$ or $\mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. After we reach line 29 and output

$$\mathrm{BWT}_S[i..i'-1] = (\mathrm{BWT}_S[i])^{i'-i},$$

the next character we output is $\mathrm{BWT}_T[j] \neq \mathrm{BWT}_S[i]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. Symmetrically, after we reach line 38 and output

$$\mathrm{BWT}_T[j..j'-1] = (\mathrm{BWT}_T[j])^{j'-j},$$

the next character we output is $\mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$ and it starts a new run in $\mathrm{BWT}_{S,T}$. ◀

It follows from Lemma 4 that we make $O(r)$ passes through the while loop. Since BWT_S and BWT_T and $\mathrm{BWT}_{S,T}$ are run-length compressed, the only operations that can take more than constant time during a single pass through the while loop are comparisons between characters' contexts, of which we make $O(\log(m+n))$ during each pass, so $O(r\log(m+n))$ in total.

▶ **Lemma 5.** *Whenever we compare two characters' contexts*

- *those two characters are not equal, so they are in different runs in* $\mathrm{BWT}_{S,T}$*;*
- *at least one of those two characters is in the run in* $\mathrm{BWT}_{S,T}$ *output during the current pass through the while loop or the run in* $\mathrm{BWT}_{S,T}$ *output during the next pass.*

Proof. When we check whether $\mathrm{context}_S(i+k) \prec \mathrm{context}_T(j+\ell-1)$ in line 7,

$$\mathrm{BWT}_S[i+k] \neq \mathrm{BWT}_S[i] = \mathrm{BWT}_T[j] = \mathrm{BWT}_T[j+\ell-1]$$

and we output either $\mathrm{BWT}_S[i+k]$ or $\mathrm{BWT}_T[j+\ell]$ during the next pass. The comparison in line 12 is symmetric. When we perform the binary search in line 8, in each step we check whether $\mathrm{context}_T(j') \succ \mathrm{context}_S(i+k)$ for some $j' < j+\ell$ with

$$\mathrm{BWT}_T[j'] = \mathrm{BWT}_T[j] = \mathrm{BWT}_S[i] \neq \mathrm{BWT}_S[i+k]$$

and we output $\mathrm{BWT}_S[i+k]$ during the next pass. The binary search in line 13 is symmetric. When we check whether $\mathrm{context}_S(i) \prec \mathrm{context}_T(j)$ in line 23, $\mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$ and we output either $\mathrm{BWT}_S[i]$ or $\mathrm{BWT}_T[j]$ during the current pass. When we check whether $\mathrm{context}_S(i+k-1) \prec \mathrm{context}_T(j)$ in line 24,

$$\mathrm{BWT}_S[i+k-1] = \mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$$

and we output either $\mathrm{BWT}_S[i+k-1]$ during the current pass or $\mathrm{BWT}_T[j]$ during the next pass. The comparison in line 33 is symmetric. When we perform the binary search in line 28, in each step we check whether $\mathrm{context}_S(i') \succ \mathrm{context}_T(j)$ for some $i' < i+k$ with

$$\mathrm{BWT}_S[i'] = \mathrm{BWT}_S[i] \neq \mathrm{BWT}_T[j]$$

and we output $\mathrm{BWT}_T[j]$ during the next pass. The binary search in line 37 is symmetric. ◀

It follows from Lemma 5 that we can charge all the context comparisons to runs in $\mathrm{BWT}_{S,T}$ such that

- each run in $\mathrm{BWT}_{S,T}$ has $O(\log(m+n))$ context comparisons charged to it;
- if a comparison between two characters' contexts is charged to a run in $\mathrm{BWT}_{S,T}$ then one of the characters is in that run and the other is not.

The number of characters we extract to compare the context of a character in a run $\mathrm{BWT}_{S,T}[a..b]$ to the context of a character in another run is at most

$$1 + \max\left(\mathrm{LCP}_{S,T}[a], \mathrm{LCP}_{S,T}[b+1]\right)$$

(unless $b = m + n$, in which case the number is just $\mathrm{LCP}_{S,T}[a]$), where $\mathrm{LCP}_{S,T}[1] = 0$ and, for $1 < h \leq m + n$, $\mathrm{LCP}_{S,T}[h]$ is the length of the longest common prefix of the contexts of the hth and $(h-1)$st characters in $\mathrm{BWT}_{S,T}$. This takes

$$O\left(\log r + \max\left(\mathrm{LCP}_{S,T}[a], \mathrm{LCP}_{S,T}[b+1]\right)\right)$$

time. Therefore, the time for the context comparisons charged to $\mathrm{BWT}_{S,T}[a..b]$ is

$$O\left(\left(\log r + \max\left(\mathrm{LCP}_{S,T}[a], \mathrm{LCP}_{S,T}[b+1]\right)\right)\log(m+n)\right).$$

Summing over the r runs in $\mathrm{BWT}_{S,T}$, the total time for context comparisons is

$$O((r \log r + L)\log(m+n)),$$

where L is the sum of its irreducible LCP values. Combined with our previous observations, this gives us a preliminary result:

▶ **Theorem 6.** *Given the RLBWTs* $\mathrm{BWT}_S[1..m]$ *and* $\mathrm{BWT}_T[1..n]$*, we can merge them into the run-length compressed eBWT* $\mathrm{BWT}_{S,T}[1..m+n]$ *using* $O(r)$ *space and* $O((r \log r + L)\log(m+n))$ *time, where* m *and* n *are the lengths of the uncompressed strings,* r *is the number of runs in* $\mathrm{BWT}_{S,T}$ *and* L *is the sum of its irreducible LCP values.*

6 Proof of Lemma 1

The $\log r$ in Theorem 6's time bound comes from paying the $O(\log r)$ overhead in Lemma 1 at every step of each binary search. In Section 7 we will explain how to avoid that and pay the overhead only three times during a binary search, but first it helps to examine the proof of Lemma 1.

If an RLBWT for a string of length n has r runs then its Ψ function is a permutation on $\{1, \ldots, n\}$ with

$$|\{1\} \cup \{i \ : \ 1 \leq i \leq n, \Psi(i) \neq \Psi(i-1) + 1\}| = r\,.$$

Given the RLBWT, in $O(r \log r)$ total time we can first build the set

$$\{(1, \Psi(1))\} \cup \{(i, \Psi(i)) \ : \ 1 \leq i \leq n, \Psi(i) \neq \Psi(i-1) + 1\}\,;$$

then apply Lemma 7 below to obtain a slightly larger superset P' with useful properties; and finally build a move structure with which, given i and i's predecessor in P', in constant time we can find $\Psi(i)$ and $\Psi(i)$'s predecessor in P'. Given only i, we can find i's predecessor in P' in $O(\log r)$, then use the move structure to iterate ψ and extract the context of the ith character in the RLBWT character by character in constant time per character; Lemma 1 follows.

We refer readers to Nishimoto and Tabei's [21], Brown et al.'s [6] and Bertram et al.'s [3] papers and Brown's [5] thesis for more details on building move structures. For the sake of completeness, we include a proof of Lemma 7 in the appendix, mostly following Brown et al.'s and Bertram et al.'s presentation but with a potential-function argument by Alhadi [1].

▶ **Lemma 7.** *Let π be a permutation on $\{1, \ldots, n\}$ and*

$$P = \{1\} \cup \{i \ : \ 1 < i \leq n, \pi(i) \neq \pi(i-1) + 1\}\,.$$

Given the set $\{(i, \pi(i)) \ : \ i \in P\}$ and an integer $d \geq 2$, we can build a set $\{(i, \pi(i)) \ : \ i \in P'\}$, where

- $P \subseteq P' \subseteq \{1, \ldots, n\}$,
- *if a and b are consecutive elements in $\{\pi(i) \ : \ i \in P'\} \cup \{n+1\}$ then $|[a, b) \cap P'| < 2d$,*
- $|P'| \leq \frac{d|P|}{d-1}$.

This takes $O(|P| \log |P|)$ time in general but $O\left(\frac{|P| \log |P|}{d}\right)$ time when n is polynomial in $|P|$.

7 Optimization

The $O(\log r)$ overhead in Lemma 1 comes from a predecessor query on a set of size $O(r)$, as described in Section 6. For our algorithm, that is multiplied by the $O(r)$ passes we make through the while loop and by the $O(\log(m+n))$ context comparisons we may make during each pass if we reach line 8, 13, 28 or 37 and perform a binary search in an RLBWT. If we do not perform such a binary search during a pass then we make only a constant number of context comparisons in that pass and they contribute only $O(\log r) \subseteq O(\log(m+n))$ overhead, so we need not worry about this case.

To see how to speed up the binary searches in the RLBWTs, consider how we search for the smallest value j' with $\text{context}_T(j') \succ \text{context}_S(i+k)$ in line 8, for example. We repeatedly choose candidate values j' in $[j, j+\ell)$ and, for each one, compare $\text{context}_T(j')$ with $\text{context}_S(i+k)$. If we choose the candidate values j' naïvely, then each context comparison starts with a predecessor query on P'_T, where P'_T is the set we build with Lemma 7 as part of building a move structure for Ψ on T. If we choose $j' \in P'_T$ or already knowing the predecessor of j' in P'_T, however, then we do not need such a predecessor query and we do not pay the $O(\log n)$ overhead for that step of the binary search.

In an optimized binary search for j', we first pay the $O(\log r)$ overhead once to be able to extract $\text{context}_S(i+k)$. We then pay $O(\log r)$ overheads twice more to find the successor of j and the predecessor of $j+\ell-1$ in P'_T. We then use binary search to find the smallest value $p \in [j, j+\ell) \cap P'_T$ with $\text{context}_T(p) \succ \text{context}_S(i+k)$, if there is one. Since $p \in [j, j+\ell) \cap P'_T \subseteq P'_T$, we do not pay the $O(\log n)$ overhead for these steps. Whether p exists or not, we can now focus our search on interval of $[j, j+\ell)$ in which all the elements have the same predecessor in P'_T, so we do not pay the $O(\log n)$ overhead for these steps either. Binary searches on BWT_S can be sped up symmetrically.

Lemma 5 still holds and we can charge all the context comparisons as before. Now, however, the time for the context comparisons charged to a run $\text{BWT}_{S,T}[a..b]$ in $\text{BWT}_{S,T}$ is only

$$O\left(\log r + \left(1 + \max\left(\text{LCP}_{S,T}[a], \text{LCP}_{S,T}[b+1]\right)\right) \log(m+n)\right)\,.$$

Summing over the r runs in $\text{BWT}_{S,T}$, the total time for context comparisons is

$$O\left(r \log r + (r+L) \log(m+n)\right) = O((r+L) \log(m+n))\,.$$

This gives us our main result:

▶ **Theorem 8.** *Given the RLBWTs* $\mathrm{BWT}_S[1..m]$ *and* $\mathrm{BWT}_T[1..n]$*, we can merge them into the run-length compressed eBWT* $\mathrm{BWT}_{S,T}[1..m+n]$ *using* $O(r)$ *space and* $O((r+L)\log(m+n))$ *time, where* m *and* n *are the lengths of the uncompressed strings,* r *is the number of runs in* $\mathrm{BWT}_{S,T}$ *and* L *is the sum of its irreducible LCP values.*

With care, it is possible to combine Theorems 3 and 8 and obtain the advantages of each – using doubling search among run boundaries and achieving adaptivity with respect to b, B and L simultaneously (better than simply dovetailing) – but we leave that as an exercise for the reader until we publish the full version of this paper.

References

1 Anas Alhadi. Personal communication, 2026.

2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812, 2020. doi:10.1016/J.TCS.2019.08.005.

3 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: Optimizing the r-index. In *Proceedings of the 22nd Symposium on Experimental Algorithms (SEA)*, 2024. doi:10.4230/LIPIcs.SEA.2024.1.

4 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14, 2019. doi:10.1186/S13015-019-0148-5.

5 Nathaniel K. Brown. BWT-runs compressed data structures for pan-genomics text indexing. Mcs thesis, Dalhousie University, 2023.

6 Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *Proceedings of the 20th Symposium on Experimental Algorithms (SEA)*, 2022. doi:10.4230/LIPIcs.SEA.2022.16.

7 Nathaniel K. Brown and Ben Langmead. Bounding the average move structure query for faster and smaller RLBWT permutations. In *Proceedings of the 24th Symposium on Experimental Algorithms (SEA)*, 2026.

8 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

9 Diego Díaz-Domínguez, Travis Gagie, Veronica Guerrini, Ben Langmead, Zsuzsanna Lipták, Giovanni Manzini, Francesco Masillo, and Vikram Shivakumar. Prefix-free parsing for merging big BWTs. In *Proceedings of the 32nd Symposium on String Processing and Information Retrieval (SPIRE)*, 2025.

10 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30, 2014. doi:10.1093/BIOINFORMATICS/BTU014.

11 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63, 2012. doi:10.1007/S00453-011-9535-0.

12 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30, 2014. doi:10.1093/BIOINFORMATICS/BTU584.

13 Uwaise Ibna Islam, Davide Cozzi, Travis Gagie, Rahul Varki, Vincenza Colonna, Erik Garrison, Paola Bonizzoni, and Christina Boucher. Scalable PBWT queries with minimum-length SMEM constraints. *bioRxiv*, 2025.

14 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the 30th Symposium on Discrete Algorithms (SODA)*, 2019.

15 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Communications of the ACM*, 65, 2022. doi:10.1145/3531445.

16 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69, 2022.

17 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40, 2024.

18 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Bioinformatics in the Era of High-Throughput Sequencing*. Cambridge University Press, 2nd edition, 2023.

19 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387, 2007. doi:10.1016/J.TCS.2007.07.014.

20 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

21 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2021. doi:10.4230/LIPIcs.ICALP.2021.101.

22 Tatsuya Ohno, Kensuke Sakai, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online RLBWT and its application to LZ77 parsing. *Journal of Discrete Algorithms*, 52, 2018.

23 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *Proceedings of the 31st Data Compression Conference (DCC)*, 2021.

24 Jouni Sirén. Burrows-Wheeler transform for terabases. In *Proceedings of the 26th Data Compression Conference (DCC)*, 2016.

A Proof of Lemma 7

▶ **Lemma 7.** *Let π be a permutation on $\{1,\ldots,n\}$ and*

$$P = \{1\} \cup \{i \ : \ 1 < i \leq n, \pi(i) \neq \pi(i-1)+1\}.$$

Given the set $\{(i,\pi(i)) \ : \ i \in P\}$ and an integer $d \geq 2$, we can build a set $\{(i,\pi(i)) \ : \ i \in P'\}$, where

- $P \subseteq P' \subseteq \{1,\ldots,n\}$,
- *if a and b are consecutive elements in $\{\pi(i) \ : \ i \in P'\} \cup \{n+1\}$ then $|[a,b) \cap P'| < 2d$,*
- $|P'| \leq \frac{d|P|}{d-1}$.

This takes $O(|P| \log |P|)$ time in general but $O\left(\frac{|P| \log |P|}{d}\right)$ time when n is polynomial in $|P|$.

Proof. Let $Q = \{\pi(i) \ : \ i \in P\} \cup \{n+1\}$. We build AVL trees T_P and T_Q storing the elements of P and Q, respectively. For each $i \in P$ we store $\pi(i)$ as satellite data with i in T_P, and we store i as satellite data with $\pi(i)$ in T_Q. For each pair of consecutive elements a and b in Q, if $|[a,b) \cap P| \geq 2d$ then we store the pair (a,b) in a list L. The bottleneck is sorting P and Q, which takes $O(|P| \log |P|)$ time in general but $O(|P|)$ time when n is polynomial in $|P|$.

We work in rounds and maintain the relationships between T_P, T_Q and L. In each round we check if L is empty and, if so, we return each element i in T_P together with its satellite data $\pi(i)$ and then stop; if not, we remove 1 element from L and then insert 1 element into T_P, 1 element into T_Q and up to 2 elements into L.

Let P_j and Q_j be the contents of T_P and T_Q after j rounds, so $P_0 = P$ and $Q_0 = Q$. Suppose we remove (s,e) from L during the $(j+1)$st round, meaning $[s,e) \cap P_j \geq 2d$. We use T_P to find the $(d+1)$st smallest element p in $[s,e) \cap P_j$. We use T_Q to find $\pi^{-1}(p)$ by finding the predecessor q of p in Q_j and adding $p-q$ to $\pi^{-1}(q)$, which is stored as satellite data with q. We insert $\pi^{-1}(p)$ into T_P with p as satellite data, and insert p into T_Q with $\pi^{-1}(p)$ as satellite data. Therefore $Q_{j+1} = \{\pi(i) \ : \ i \in P_{j+1}\} \cup \{n+1\}$.

To see why $\pi^{-1}(p) = \pi^{-1}(q) + p - q$ and suppose $\pi(x) = y$. If $x \in P_j$ then $y \in Q_j$, so if $y \notin Q_j$ then $x \notin P_j$. Therefore, if $y \notin Q_j$ then $\pi(x) = \pi(x-1) + 1$ and so $y - 1 = \pi(x-1)$; applying π^{-1} to both sides we have $\pi^{-1}(y-1) = x - 1$ and so $\pi^{-1}(y) = \pi^{-1}(y-1) + 1$. Since q is p's predecessor in Q_j, it follows that $\pi^{-1}(p) = \pi^{-1}(q) + p - q$.

We use T_P and T_Q to check first whether $|[p, e) \cap P_{j+1}| \geq 2d$; if so, we add (p, e) to L. We then use T_P and T_Q to find the predecessor s' of $\pi^{-1}(p)$ in Q_{j+1} and next element e' after s' in Q_{j+1}, and to check whether $|[s', e') \cap P_{j+1}| = 2d$; if so, we add (s', e') to L. Notice that if $|[s', e') \cap P_{j+1}| > 2d$ then $|[s', e') \cap P_j| \geq 2d$ so (s', e') was already in L. This completes the $(j+1)$st round, which takes $O(\log |P_j|)$ total time.

To bound the number of rounds we use, let

$$f(k) = \sum \left\{ \max\left(|[a, b) \cap P_k| - d, 0 \right) \ : \ a \text{ and } b \text{ are consecutive elements in } Q_k \right\}$$

and consider how we obtain $f(j+1)$ from $f(j)$. We subtract the term $\max\left(|[s, e) \cap P_j| - d, 0 \right)$; add the terms $\max\left(|[s, p) \cap P_j| - d, 0 \right)$ and $\max\left(|[p, e) \cap P_j| - d, 0 \right)$; and finally increment the term $\max\left(|[s', e') \cap P_j| - d, 0 \right)$, to make everything with respect to P_{j+1} instead of P_j. Since

$$2d \leq |[s, e) \cap P_j| = |[s, p) \cap P_j| + |[p, e) \cap P_j| = d + |[p, e) \cap P_j| \,,$$

we have

$$\begin{aligned}
\max\left(|[s, e) \cap P_j| - d, 0 \right) &= |[p, e) \cap P_j| \,, \\
\max\left(|[s, p) \cap P_j| - d, 0 \right) &= 0 \,, \\
\max\left(|[p, e) \cap P_j| - d, 0 \right) &= |[p, e) \cap P_j| - d \,.
\end{aligned}$$

It follows that $f(j+1) \leq f(j) - (d-1)$.

Since $f(0) \leq |P|$ and we stop if $f(j+1) = 0$ and $|P_{j+1}| \leq |P_j| + 1$, we use at most $\frac{|P|}{d-1}$ rounds and return P' with

$$|P'| \leq |P| + \frac{|P|}{d-1} = \frac{d|P|}{d-1} \,.$$

Since the $(j+1)$st round takes $O(\log |P_j|) = O(\log |P|)$ time for all j, over all the rounds we use $O\left(\frac{|P| \log |P|}{d} \right)$ total time. ◀