An Efficient Data Structure and Algorithm for Long-Match Query in Run-Length Compressed BWT

Ahsan Sanaullah
Department of Computer Science, University of Central Florida, Orlando, FL, USA

Degui Zhi
McWilliams School of Biomedical Informatics, University of Texas Health Science Center at Houston, TX, USA

Shaojie Zhang
Department of Computer Science, University of Central Florida, Orlando, FL, USA

Abstract

String matching problems in bioinformatics are typically for finding exact substring matches between a query and a reference text. Previous formulations often focus on maximum exact matches (MEMs). However, multiple occurrences of substrings of the query in the text that are long enough but not maximal may not be captured by MEMs. Such long matches can be informative, especially when the text is a collection of similar sequences such as genomes. In this paper, we describe a new type of match between a pattern and a text that aren't necessarily maximal in the query, but still contain useful matching information: locally maximal exact matches (LEMs). There are usually a large amount of LEMs, so we only consider those above some length threshold $\mathcal{L}$. These are referred to as long LEMs. The purpose of long LEMs is to capture substring matches between a query and a text that are not necessarily maximal in the pattern but still long enough to be important. Therefore efficient long LEMs finding algorithms are desired for these datasets. However, these datasets are too large to query on traditional string indexes. Fortunately, these datasets are very repetitive. Recently, compressed string indexes that take advantage of the redundancy in the data but retain efficient querying capability have been proposed as a solution. We therefore give an efficient algorithm for computing all the long LEMs of a query and a text in a BWT runs compressed string index. We describe an $O(m + occ)$ expected time algorithm that relies on an $O(r)$ words space string index for outputting all long LEMs of a pattern with respect to a text given the matching statistics of the pattern with respect to the text. Here m is the length of the query, occ is the number of long LEMs outputted, and r is the number of runs in the BWT of the text. The $O(r)$ space string index we describe relies on an adaptation of the move data structure by Nishimoto and Tabei. We are able to support $LCP[i]$ queries in constant time given $SA[i]$. In other words, we answer $PLCP[i]$ queries in constant time. These $PLCP$ queries enable the efficient long LEM query. Long LEMs may provide useful similarity information between a pattern and a text that MEMs may ignore. This information is particularly useful in pangenome and biobank scale haplotype panel contexts.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Data compression

Keywords and phrases BWT, LEM, Long LEM, MEM, Run Length Compressed BWT, Move Data Structure, Pangenome

Digital Object Identifier 10.4230/LIPIcs.WABI.2025.17

Related Version *Preprint*: https://arxiv.org/abs/2505.15698 [41]

Funding This work was supported by the National Institutes of Health grant R01 HG010086.

1 Introduction

Bioinformatics sequence data is often large and very repetitive. Furthermore, efficient matching queries on the data are frequently needed for many biological analyses. Therefore, bioinformatics problems have incentivized and profited from the development of efficient string indexes. The Burrows-Wheeler transform (BWT) has thus been used in bioinformatics algorithms. The BWT is a permutation of a text that has found wide use in string indexing and data compression [11]. Position i in the BWT of the text is essentially the character preceding the i-th lexicographically smallest suffix of the text. Due to this lexicographic sorting, adjacent characters in the BWT correspond to the characters preceding highly locally similar suffixes of the text. Therefore, the BWT of highly repetitive texts tends to have large runs of one character, with an overall small number of runs. The BWT of highly repetitive texts therefore compresses well. In fact, the number of runs in BWT, r, is sometimes used as a measure of the repetitiveness of a string [35]. Finally, given only the BWT of a text, the text can be reconstructed in linear time [11] and the BWT of a text can be constructed in linear time by construction of the suffix array [22]. The BWT ordering also allows efficient string indexes. In other words, given a pattern, find all occurrences of the pattern within the text. String indexes have been shown that output all occurrences of a pattern (a locate query) in space linear to the product of the length of the text and the size of the alphabet and time linear to the sum of the length of the pattern and the number of occurrences [18].

Compressed string indexes have also been shown [3, 18, 32]. These indexes output all occurrences of a pattern in space sublinear to the size of the text. Although the time complexity of locating these occurrences is not linear in the length of the pattern and the number of occurrences, they are typically independent of the length of the text barring logarithmic factors and close to linear in the length of the pattern and number of occurrences. In particular for highly repetitive texts, the space of the index can be much smaller than the space of the text. Notably, recent compressed string indexes have achieved space linear to the number of runs in the BWT (r) [19, 36]. The r-index by Gagie et al. was the first compressed string index offering close to linear time locate queries in $O(r)$ space [19]. Nishimoto and Tabei recently improved on this result with their OptBWTR, which achieves linear time locate queries for texts with alphabets of size polylogarithmic in the length of the text. OptBWTR relies on the move data structure, which was introduced in the same paper [36].

Compressed string indexes have been fruitfully applied to the growing collection of bioinformatics data. Over the past two decades, large collections of genomics data have grown increasingly larger in size. For example, the UK Biobank has whole genome sequencing data of roughly one million haplotypes [29], and the All of Us program has released whole genome sequencing data of half a million haplotypes [7]. Furthermore, recent arguments have been made that a human reference pangenome should be used instead of a singular human reference genome to avoid reference bias in downstream analyses [33, 43, 46]. The Human Pangenome Reference Consortium has released a draft human pangenome reference of more than two hundred high quality phased diploid assemblies and is planning to release over three hundred and fifty in the final release [12, 30]. The UK Biobank whole genome sequencing data has 1.5 billion variants, the All of Us whole genome sequencing data has 1 billion variants, and the typical diploid assembly in the draft human pangenome has 6 billion bases. Therefore, these datasets have 1,500 trillion, 250 trillion, and 1.2 trillion characters each respectively. However, while very large, these datasets are very repetitive. Furthermore, queries on these datasets are frequently needed for biological applications including read mapping [24, 27], read alignment [28], read classification for metagenomes [1, 15, 45] or pangenomes [10]. Many general purpose compressed string indexes have also been implemented for exact pattern matching and matching substrings of the pattern [16, 26, 38, 49]. These indexes may compute the maximal exact matches (MEMs) of the pattern with respect to the text. MEMs are matches between the pattern and the text that cannot be extended in the pattern.

While MEMs typically refer to matches that are maximal in the pattern, matches that are simultaneously maximal in the pattern and the text may sometimes be desired. Notably, in two data structures related to the BWT, algorithms for outputting matches that are simultaneously maximal in the pattern and the text have already been developed. These data structures are the positional Burrows-Wheeler transform (PBWT) and the graph Burrows-Wheeler transform (GBWT) [17, 44]. In the PBWT and GBWT, matches that cannot be extended in the pattern are referred to as set maximal matches, and matches that cannot be simultaneously extended in the pattern are referred to as locally maximal matches. Locally maximal matches that are longer than some length threshold $\mathcal{L}$ are referred to as $\mathcal{L}$-long matches, or long matches for short. Algorithms for outputting set maximal matches and long matches have been published in the PBWT in uncompressed [17, 34, 40] and compressed space [8, 13, 42, 48]. Algorithms for outputting these matches have also been published for the GBWT in compressed space [39].

In this paper, we use these concepts in the traditional pattern and text context, and name matches that cannot be simultaneously extended in the pattern and the text locally maximal exact matches (LEMs). LEMs that are longer than $\mathcal{L}$ are long LEMs. The distinction between matches that do not extend in the pattern and matches that do not extend simultaneously in the pattern and the text has been made before. Notably, in ropebwt3 MEMs refers to LEMs of our paper and super maximal exact matches (SMEMs) refers to MEMs of our paper [26]. The term SMEM has been used in place of MEM in a few papers to avoid the confusion in terminology [8, 13, 15, 26], however MEM is still the most common term by far for matches that cannot be extended in the pattern. The authors are not aware of any published algorithms for the computation of LEMs or long LEMs.

In this work, we describe an algorithm for outputting all long LEMs of a pattern with respect to a text in $O(m+occ)$ expected time given the matching statistics of the pattern with respect to the text, where m is the length of the pattern and occ is the number of long LEMs it has with respect to the text. In order to do so, we modify the OptBWTR data structure of Nishimoto and Tabei to also compute $LCP[i]$ given $SA[i]$ (i.e. compute $PLCP$). We name this modified OptBWTR, *OptBWTRL* (i.e. OptBWTR for long LEMs or OptBWTR with LCP). OptBWTRL maintains the $O(r)$ words space complexity of OptBWTR and computes $\phi[i]$ and $PLCP[i]$ in constant time. The long LEM finding algorithm also requires as input an OptBWTRL of the text. We also discuss possible future work related to this paper, including avenues for improving the results, utilization of constant time $PLCP$ computation to speed up matching statistics computation, and biological applications of long LEMs. Long LEMs may have many biological applications, from identity by descent segment detection and local ancestry inferences, to seeds or anchors for approximate matching algorithms for genome to genome alignment, genome to pangenome, read to genome or other alignments. In this paper, our main contributions are the following:

- **OptBWTRL:** OptBWTRL is an $O(r)$ words space data structure that maintains the capabilities of OptBWTR and adds the ability to compute $\phi, PLCP$, and long LEMs efficiently. r is the number of runs in the BWT of the text.
  - **PLCP:** OptBWTRL enables constant time $PLCP[i]$ computation in $O(r)$ space. Note that $PLCP[SA[j]] = LCP[j]$, therefore $PLCP$ computation in constant time allows $LCP[j]$ computation in constant time given $SA[j]$.

  - **Long LEM Query:** We describe an $O(m + occ)$ expected time long LEM query for pattern P and text T given the matching statistics of P with respect to T. The underlying index (OptBWTRL) uses $O(r)$ space. m is the length of P and occ is the number of long LEMs P has with respect to T. A deterministic time bound for a similar algorithm we show is $O\left(m + occ\sqrt{\frac{\log occ}{\log\log occ}}\right)$.
- **Long LEM Query with random access to the text:** Given $O(t_{RA})$ time random access to the text and a BWT related index, algorithms for computing matching statistics efficiently are known. Therefore, our long LEM query algorithm results in the following.
  - **In Uncompressed Space:** An algorithm for long LEM query in $O(m+occ)$ expected time in uncompressed string indexes such as the FM Index (Corollary 3.2, variant with $O(n\sigma)$ space, where n is the length of the text and σ is the size of the alphabet) [18].
  - **In Compressed Space:** An algorithm for long LEM query in $O(m \log \frac{n}{\delta} + occ)$ expected time in $O(r + \delta \log \frac{n}{\delta})$ space given a block tree [4, 23] (with random access to the text in $O(\log \frac{n}{\delta})$ time in $O(n \log \frac{n}{\delta})$ space) and an OptBWTRL of the text.

2 Background

In this section, we review definitions used throughout the rest of the paper. We begin with strings, then in Section 2.1, we review BWT related concepts. In Section 2.2, we give a short overview of the results of Nishimoto and Tabei in [36]. Matching statistics are reviewed in Section 2.3. Finally we review maximal exact matches (MEMs) and define locally maximal exact matches (LEMs) in Section 2.4.

Let $\Sigma = \{1, 2, 3, \ldots, \sigma\}$ be an ordered alphabet of size σ. The size (number of characters it contains) of a string T is represented by $|T|$. T refers to a text of length n ($|T| = n$) where the last character is \$. The character \$ is lexicographically smaller than all other characters in T and occurs only in the last position of T. The i-th character of T is $T[i]$, $i \in [1, n]$. $T[i, j]$ refers to the substring of T that starts at position i and ends at position j, inclusive ($T[i, j] = T[i]T[i+1]T[i+1] \ldots T[j]$). Prefix i of T is the string $T[1, i]$, suffix i of T is $T[i, n]$. The longest common prefix of two strings T and T' is referred to by $lcp(T, T')$. $|lcp(T, T')|$ is the largest value i s.t. $i \leq \min(|T|, |T'|)$ and $T[1, i] = T'[1, i]$ (then, $lcp(T, T') = T[1, i] = T'[1, i]$). A string T' being lexicographically smaller than T is represented by $T' \prec T$. If $T' = T$, $T' \not\prec T$ and $T \not\prec T'$. If $T' \neq T$, $T' \prec T$ iff $T' = lcp(T, T')$ or $T'[|lcp(T, T')| + 1] < T[|lcp(T, T')| + 1]$.

2.1 Burrows-Wheeler Transform

The Suffix Array (SA) of a text T is an array of length $n = |T|$ where the i-th position stores the index of the i-th lexicographically smallest suffix of T. Therefore, $T[SA[1], n] \prec T[SA[2], n] \prec T[SA[3], n] \prec \cdots \prec T[SA[n], n]$. The Burrows-Wheeler Transform (BWT) of a text T is a string of length n where the i-th character in the string is the $SA[i] - 1$-th character of T (the n-th character if $SA[i] = 1$). The LF array is an array of length n that stores the position of the previous suffix in the suffix array, $LF[i] = j$ s.t. $SA[j] = SA[i] - 1$ for all $SA[j] \in [1, n-1]$, $LF[i] = j$ s.t. $SA[j] = n$ for $SA[i] = 1$. The ϕ array stores at position i, the suffix above suffix i in the suffix array, i.e. if $SA[k] = i$, $\phi[i] = SA[k-1]$ ($\phi[i] = SA[n]$ if $i = SA[1]$). The ϕ^{-1} array stores at position i, the suffix below suffix i in the suffix array, i.e. if $SA[k] = i$, $\phi^{-1}[i] = SA[k+1]$ ($\phi^{-1}[i] = SA[1]$ if $i = SA[n]$). Therefore, $\phi[\phi^{-1}[i]] = i$ and $\phi^{-1}[\phi[i]] = i$. The LCP array is an array of length n where $LCP[i]$ stores the length of the longest common prefix of suffix $SA[i]$ and $SA[i-1]$. $LCP[1] = 0$ and for ordered by position in SA

i	SA	LCP	LF	F cyclic shifts of text L
1	20	0	2	$mississismississippi
2	19	0	11	i$mississismississipp
3	16	1	13	ippi$mississismississ
4	5	1	14	isismississippi$miss
5	7	2	15	ismississippi$missis
6	13	2	16	issippi$missisismiss
7	2	4	9	issisismississippi$m
8	10	5	10	ississippi$missisism
9	1	0	1	missisismississippi$
10	9	6	17	mississippi$missisis
11	18	0	12	pi$missisismississip
12	17	1	3	ppi$missisismississi
13	15	0	18	sippi$missisismissis
14	4	2	19	sisismississippi$mis
15	6	3	4	sismississippi$missi
16	12	3	20	sissippi$missisismis
17	8	1	5	smississippi$missisi
18	14	1	6	ssippi$missisismissi
19	3	3	7	ssisismississippi$mi
20	11	4	8	ssissippi$missisismi

ordered by position in text

Text:	m	i	s	s	i	s	i	s	m	i	s	s	i	s	s	i	p	p	i	$
ISA:	9	7	19	14	4	15	5	17	10	8	20	16	6	18	13	3	12	11	2	1
PLCP:	0	4	3	2	1	3	2	1	6	5	4	3	2	1	0	1	1	0	0	0
ϕ:	10	13	14	15	16	4	5	12	1	2	3	6	7	8	17	19	18	9	20	11
ϕ^{-1}:	9	10	11	6	7	12	13	14	18	1	20	8	2	3	4	5	15	17	16	19

RLBWT: (i,1),(p,2),(s,3),(m,7),($,9),(s,10),(p,11)(i,12),(s,13),(i,15),(s,16),(i,17)

Figure 1 BWT and related structures for $T = missisismississippi\$$. SA, LCP, LF, F, and L are ordered by position in SA while $ISA, PLCP, \phi$, and ϕ^{-1} are ordered by position in the text.

$i \in [2, n]$, $LCP[i] = |lcp(T[SA[i], n], T[SA[i-1], n])|$. The $PLCP$ (permuted LCP) array is an array of length n where the LCP array is stored by suffix index. Therefore, if $SA[i] = j$, $PLCP[j] = LCP[i] = |lcp(T[j, n], T[\phi[j], n])|$. Finally, the inverse suffix array, ISA, is an array of length n that stores at position i the position of suffix i in the suffix array, if $ISA[i] = j$, $SA[j] = i$. $ISA[SA[i]] = i$ and $SA[ISA[i]] = i$. The SA, LF, ϕ, ϕ^{-1}, and ISA arrays are permutations of the integers in $[1, n]$. SA and ISA are inverses of each other and ϕ and ϕ^{-1} are inverses of each other.

The run-length Burrows-Wheeler Transform (RLBWT) is the run-length encoding of the BWT of a text. Call L the BWT of text T. Then, L is partitioned into r nonempty substrings $L_1, L_2, \ldots, L_r$. L_i is a substring of L corresponding to the i-th run of L. A run is a maximal repetition of the same character in L. Therefore, $L_i[1] = L_i[2] = \cdots = L_i[|L_i|]$ for all $i \in [1, r]$ and $L_i[1] \neq L_{i+1}[1]$ for all $i \in [1, r-1]$. l_i is the starting position of the run L_i in L. The RLBWT is represented as r pairs $(L_i[1], l_i)$ for $i \in [1, r]$. All of these structures can be seen in Figure 1 for a text $T = missisismississippi\$$.

2.2 Move Data Structure

The move data structure is a data structure for representing a permutation of a contiguous range of integers efficiently. It was introduced by Nishimoto and Tabei [36]. In the original introduction, the structure was described for a permutation of $[1, n]$. This of course may be extended to any bijective function from a contiguous range of integers to another contiguous range of integers. The move data structure takes space proportional to the number of intervals conserved in the function. An interval is conserved in a bijective function from a contiguous range of integers to another contiguous range of integers if for any i, j in the interval, $f(i) - f(j) = i - j$ (therefore, $f(i) = f(j) + i - j$ and $f(i) - i = f(j) - j$). The move data structure computes the represented function in constant time. The important arrays, LF and ϕ^{-1}, are permutations of $[1, n]$ with $O(r)$ conserved intervals, where r is the number of runs in the BWT. Therefore, Nishimoto and Tabei define the OptBWTR data structures using move data structures. OptBWTR supports efficient count and locate queries in BWT-runs compressed space. Below, we more formally review some of the results from their paper [36].

2.2.1 Disjoint Interval Sequence

$I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ is a sequence of k pairs of integers. Let $p_{k+1} = n + 1$. Then i is a *disjoint interval sequence* iff there exists a permutation π of $[1, k]$ s.t. (i) $p_1 = 1 < p_2 < \cdots < p_k \leq n$, (ii) $q_{\pi[1]} = 1$, and (iii) $q_{\pi[i]} = q_{\pi[i-1]} + (p_{\pi[i-1]+1} - p_{\pi[i-1]})$. $[p_i, p_{i+1} - 1]$ is referred to as the i-th input interval, and $[q_i, q_i + (p_{i+1} - p_i) - 1]$ as the i-th output interval. The input intervals don't overlap, and their union is $[1, n]$. The output intervals don't overlap and their union is $[1, n]$.

A *move query* on a disjoint interval sequence I takes as input (i, x), where i is an index in $[1, n]$ and x is the index of the input interval sequence that contains it, $i \in [1, n]$ and $p_x \leq i < p_{x+1}$ and $x \in [1, k]$. The move query outputs (i', x') where $i' = q_x + (i - p_x)$ and $p_{x'} \leq i' < p_{x'+1}$, i.e. i' is the mapping of position i from the input to output intervals by I and x' is the index of the input interval that contains i'. f, a permutation of $[1, n]$ with k conserved intervals, can be represented by a disjoint interval sequence where the input intervals are the conserved intervals and the output intervals are the mapping of the input intervals by f. Then, a move query of (i, x) returning (i', x') computes f by $f(i) = i'$.

Nishimoto and Tabei show that move queries on a disjoint interval sequence of k input intervals (and therefore k output intervals) can be computed in constant time and $O(k)$ space with the move data structure. The move data structure is built by splitting the k input intervals of I into at most $2k$ intervals. This results in a disjoint interval sequence of at most $2k$ input intervals (and an equivalent number of output intervals) that represents the same permutation as the original disjoint interval sequence. The split interval sequence of i that the move data structure is built on is referred to as a balanced interval sequence. The notation for a balanced interval sequence of I is $B(I)$, and the notation for a move data structure of I is $F(I)$. In this paper, we occasionally use input interval of $F(I)$ as shorthand for input interval of $B(I)$ (for example, i-th input interval of a move data structure refers to the i-th input interval of the balanced interval sequence it was built on). Brown et al. extend the balanced interval sequence result of Nishimoto and Tabei to splitting I's k intervals into at most $k + \frac{k}{d-1}$ intervals, resulting in a move data structure with $O(d)$ time move query computation for any $d \geq 2$ [9].

2.2.2 OptBWTR

The arrays LF and ϕ^{-1} are permutations of $[1, n]$ with $O(r)$ conserved intervals. For LF, a conserved interval is within a run in the BWT. For ϕ^{-1}, a conserved interval is a range of suffixes of T that don't occur at the bottom of a run in the BWT (except the first position of the interval may be at the bottom of a run). Therefore, Nishimoto and Tabei define the OptBWTR data structure as the combination of the move data structures of the LF and ϕ^{-1} functions along with a rank-select data structure on an $O(r)$ length string L_{first}. OptBWTR supports $O(m \log\log_w \sigma)$ time count queries and $O(m \log\log_w \sigma + occ)$ time locate queries in $O(r)$ words of space, where r is the number of runs in the BWT of the text, m is the length of the pattern, occ is the number of occurrences of the pattern in the text, w is the word size, σ is the size of the alphabet (Theorem 9 of [36]). The input intervals of $B(I_{LF})$, the disjoint interval sequence of the move data structure of LF, are contained within a run in the BWT. Call the i-th input interval of $B(I_{LF})$ $[p_i, p_{i+1} - 1]$. Then, $L_{first} = L[p_1]L[p_2]L[p_3] \ldots L[p_k]$, where $k \leq 2r$ is the number of input intervals of $B(I_{LF})$, L is the BWT of T, and $\forall i \in [1, k], j \in [p_i, p_{i+1} - 1] L[j] = L[p_i]$. Call $B(I_{SA})$ the disjoint interval sequence of the move data structure of ϕ^{-1}, and $[p_i^-, p_{i+1}^-]$ its i-th input interval. OptBWTR is composed of:

- move data structures for LF and ϕ^{-1} ($F(I_{LF})$ and $F(I_{SA})$ respectively),
- a rank-select data structure on L_{first} ($R(L_{first})$),
- samples of the SA at the beginning of input intervals of the LF move data structure (SA^+, where $SA^+[i] = SA[p_i]$), and
- the index of the input interval of the ϕ^{-1} move data structure that contains each SA sample in SA^+ (SA^+_{index}, where $SA^+_{index} = y \iff SA^+[i] \in [p^-_y, p^-_{y+1}]$).

2.3 Matching Statistics

The matching statistics of a pattern P with respect to a text T represents information on the local similarity of the pattern to the text. The matching statistics of P with respect to T, ${}_PMS_T$, is an array of length $|P| = m$ that stores at position i three values: ${}_PMS_T[i].len$, ${}_PMS_T[i].suff$, and ${}_PMS_T[i].row$. ${}_PMS_T[i].len$ is the length of the longest substring of P starting at i that occurs in T. ${}_PMS_T[i].suff$ is a suffix of T that has a longest common prefix with $P[i, m]$ of length ${}_PMS_T[i].len$ (or equivalently, ${}_PMS_T[i].suff$ is the starting position of an occurrence of $P[i, i + {}_PMS_T[i].len - 1]$ in T). ${}_PMS_T[i].row$ is the index in the SA of T that has value ${}_PMS_T.suff$. Formally for all $i \in [1, m]$,

- ${}_PMS_T[i].len = \max_{j\in[1,|T|]} |lcp(P[i, m], T[j, |T|])|$,
- $|lcp(T[{}_PMS_T[i].suff, n], P[i, m])| = {}_PMS_T[i].len$, and
- $SA[{}_PMS_T[i].row] = {}_PMS_T[i].suff$.

When P and T are clear from the context, we omit them from ${}_PMS_T$ and refer to the matching statistics of P with respect to T as MS.

2.4 Maximal and Locally Maximal Exact Matches

For a pattern P and a text T ($|P| = m, |T| = n$), a maximal exact match (MEM), $P[i, j] = T[i', j']$, is a match between P and T that cannot be extended left or right in the pattern. Formally, ($i = 1$ or $P[i - 1, j]$ doesn't occur in T) and ($j = m$ or $P[i, j + 1]$ doesn't occur in T). A MEM can be fully specified by the triple (i, i', k) where $k = j - i + 1$ is the length of the match and i and i' are the starting positions of the match in the pattern and the text respectively.

For a pattern P and a text T, a locally maximal exact match (LEM), $P[i, j] = T[i', j']$, is a match between P and T that cannot be simultaneously extended in the pattern and the text. The match cannot be simultaneously extended left in the pattern and the text. Likewise, it cannot be simultaneously extended right in the pattern and the text. Formally, ($i = 1$ or $i' = 1$ or $P[i-1, j] \neq T[i'-1, j']$) and ($j = m$ or $j' = n$ or $P[i, j+1] \neq T[i', j'+1]$). A LEM can also be fully specified by the triple (i, i', k) where k is the length of the LEM and $k = j - i + 1$. For some length threshold $\mathcal{L}$, a long LEM is a LEM with length at least $\mathcal{L}$. See Figure 2 for a depiction of MEMs and LEMs in a text representing a pangenome.

3 Methods

Here we describe the main results of our paper. In Section 3.1, we prove move data structures can compute ϕ and $PLCP$ in constant time. Then we describe OptBWTRL, our modification of OptBWTR that utilizes these move data structures. In Section 3.2, we describe multiple algorithms for long LEM query provided an OptBWTRL of the text and matching statistics of the pattern with respect to the text.

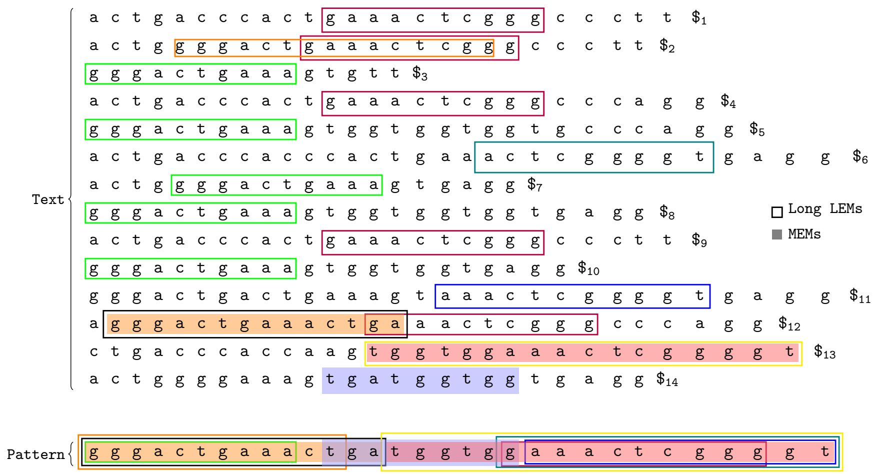

Figure 2 MEMs and LEMs of a pattern (haplotype) vs a text (pangenome). Haplotype i is the sequence of characters between $\$_{i-1}$ and $\$_i$. The text is the concatenation of the haplotypes $T =$ "$actgacccactgaaactcgggccctt\$_1actggggactgaaactcgggccctt\$_2\ldots$". MEMs and long LEMs (length threshold for long LEMs: $\mathcal{L} = 10$) of the pattern (a haplotype not contained in the pangenome) with respect to the text (the pangenome) are highlighted. MEMs are shaded in while LEMs are boxed in. In this example, MEMs are only able to detect relationships among the haplotypes most closely related to the pattern haplotype. Haplotypes similar to the pattern but not maximally similar at any location remain undetected. Notably, haplotype 2 is very similar to the pattern but doesn't contain any MEMs with it. The number of undetected similar haplotypes in biobank scale haplotype panels may be an order of magnitude larger.

3.1 Computing LCP with Move Data Structures

We define p_j^+ to be the j-th smallest suffix that occurs at the top of a run in the BWT. Therefore let (i) $p_1^+ < p_2^+ < \cdots < p_r^+ < p_{r+1}^+ = n+1$ and (ii) $\{p_1^+, p_2^+, \ldots, p_r^+, p_{r+1}^+\} = \{SA[l_1], SA[l_2], \ldots, SA[l_r], n+1\}$. Lemma 1 and its proof are phrased very similarly to Lemma 4 in [36] to demonstrate its derivativeness and the similarity of the properties.

▶ **Lemma 1.** *(i) Let x be the integer satisfying $p_x^+ \le i < p_{x+1}^+$ for some integer $i \in [1, n]$. Then $LCP[ISA[i]] = LCP[ISA[p_x^+]] - (i - p_x^+)$.*

Proof. Lemma 1(i) clearly holds for $i = p_x^+$. We show that Lemma 4(i) holds for $i \neq p_x^+$ (i.e., $i > p_x^+$). Let s_t be the position in SA with sa-value $p_x^+ + t$ for an integer $t \in [1, y]$ (i.e., $SA[s_t] = p_x^+ + t$) where $y = i - p_x^+$. Two adjacent positions $s_t - 1$ and s_t are contained in an interval $[l_v, l_v + |L_v| - 1]$ on LCP which corresponds to the v-th run L_v of L. This is because s_t is not the starting position of a run, i.e., $(SA[s_t] = p_x^+ + t) \notin \{p_1^+, p_2^+, \ldots, p_r^+\}$. The LF function maps s_t to s_{t-1}, where s_0 is the position with sa-value p_x^+. LF also maps $s_t - 1$ to $s_{t-1} - 1$ by Lemma 3(i) of [36]. $LCP[s_{t-1}] = LCP[s_t] + 1$ due to s_t and $s_t - 1$ being in the same interval on L, L_v. These relationships produce y equalities $LCP[s_0] = LCP[s_1] + 1, LCP[s_1] = LCP[s_2] + 1, \ldots, LCP[s_{y-1}] = LCP[s_y] + 1$. The equalities lead to $LCP[s_0] = LCP[s_y] + y$, and therefore $LCP[s_y] = LCP[s_0] - y$. Which represents $LCP[ISA[i]] = LCP[ISA[p_x^+] - (i - p_x^+)$ by $ISA[i] = s_y, ISA[p_x^+] = s_0$, and $y = (i - p_x^+)$. ◀

▶ **Lemma 2.** *(i) Let x be the integer satisfying $p_x^+ \le i < p_{x+1}^+$ for some integer $i \in [1, n]$. Then $PLCP[i] = PLCP[p_x^+] - (i - p_x^+)$.*

Proof. By Lemma 1 and $PLCP[j] = LCP[ISA[j]]$ for all $j \in [1, n]$ [21]. ◀

3.1.1 Move Data Structure for ϕ

p^+_j remains as defined in the previous section. Let δ^+ be a permutation of $[1, r]$ satisfying $\phi(p^+_{\delta^+[1]}) < \phi(p^+_{\delta^+[2]}) < \cdots < \phi(p^+_{\delta^+[r]})$. ϕ has the following properties on RLBWT.

▶ **Lemma 3.** *The following three statements hold: (i) Let x be the integer satisfying $p^+_x \leq i < p^+_{x+1}$ for some integer $i \in [1, n]$. Then $\phi(i) = \phi(p^+_x) + (i - p^+_x)$; (ii) $\phi(p^+_{\delta^+[1]}) = 1$ and $\phi(p^+_{\delta^+[i]}) = \phi(p^+_{\delta^+[i-1]}) + d$ where $d = p^+_{\delta^+[i-1]+1} - p^+_{\delta^+[i-1]}$; (iii) $p^+_1 = 1$.*

Proof. See Appendix A. ◀

We can compute ϕ by using a move data structure. A sequence I_ϕ consists of r pairs $(p^+_1, \phi(p^+_1)), (p^+_2, \phi(p^+_2)), \ldots, (p^+_r, \phi(p^+_r))$. I_ϕ satisfies the three conditions of a disjoint interval sequence by Lemma 3, and ϕ is equal to the bijective function represented by I_ϕ.

▶ **Lemma 4.** *(i) I_ϕ is a disjoint interval sequence. (ii) ϕ is equal to the bijective function represented by I_ϕ.*

Proof. (i) I_ϕ has the following three properties: (a) $p^+_1 = 1 < p^+_2 < \cdots < p^+_r \leq n$ holds by Lemma 3(iii) and the definition of the sequence $p^+_1, p^+_2, \ldots, p^+_{r+1}$, (b) $\phi(p^+_{\delta^+[1]}) = 1$ by Lemma 3(ii), and (c) $\phi(p^+_{\delta^+[i]}) = \phi(p^+_{\delta^+[i-1]}) + (p^+_{\delta^+[i-1]+1} - p^+_{\delta^+[i-1]})$. Therefore I_ϕ satisfies the three conditions of the disjoint interval sequence.

(ii) Let f_ϕ be the bijective function represented by I_ϕ. Then $f_\phi(i) = \phi(p^+_x) + (i - p^+_x)$ where x is the integer such that $p^+_x \leq i < p^+_{x+1}$ holds. On the other hand, $\phi(i) = \phi(p^+_x) + (i - p^+_x)$ holds by Lemma 3(i). Therefore $f_\phi(i) = \phi(i)$ and f_ϕ and ϕ are the same function. ◀

Let $F(I_\phi)$ be the move data structure built on the balanced interval sequence $B(I_\phi)$ for I_ϕ. By Lemma 6 of [36], $F(I_\phi)$ requires $O(r)$ words of space. By the results of Section 3.2 of [36], evaluation of a move query using a move data structure for a balanced disjoint interval sequence takes constant time. Finally, $\phi(i) = i'$ holds for a move query $Move(B(I_\phi), i, x) = (i', x')$ by Lemma 4. Therefore we have proved (i) of the following lemma.

▶ **Lemma 5.** *(i) There exists a move data structure $F(I_\phi)$ that computes $\phi(i)$ in $O(r)$ space and constant time given x, the index of the input interval of I_ϕ that contains i. (ii) This move data structure can be modified to also compute $PLCP[i]$ in $O(r)$ space and constant time given i and x. Call the modified move data structure $F(I_{\phi,PLCP})$.*

Proof. Say that $B(I_\phi)$ has k^+ input intervals and the i-th input interval is $[p^+_i, p^+_{i+1}]$. Then we modify the move data structure $F(I_\phi)$ by adding an array LCP^+ of size k^+. $LCP^+[i]$ stores the value $LCP[ISA[p^+_x]]$ for each $x \in [1, k^+]$. ($PLCP[p^+_x] = LCP[ISA[p^+_x]]$.) $PLCP[i] = PLCP[p^+_x] - (i - p^+_x)$ by Lemma 1. Therefore $PLCP[i]$ is computed in constant time by evaluating $LCP^+[x] - (i - p^+_x)$. Call this modified move data structure $F(I_{\phi,PLCP})$. ◀

A similar function that we may need to compute is $LCP[i+1]$ given $SA[i]$, i.e. given $SA[i] = y$, compute $|lcp(T[y, n], T[\phi^{-1}(y), n])| = PLCP[\phi^{-1}(y)] = LCP[i+1]$. Nishimoto and Tabei described $F(I_{SA})$, a move data structure computing ϕ^{-1}. $F(I_{SA})$ can be modified to compute $PLCP[\phi^{-1}(y)]$ in constant time as well in a similar fashion to the modification of the $F(I_\phi)$ data structure. Call the disjoint interval sequence $F(I_{SA})$ is built on $B(I_{SA})$. Call the i-th input interval of $B(I_{SA})$ $[p^-_i, p^-_{i+1} - 1]$, where $B(I_{SA})$ has k^- input intervals and $p^-_{k^-+1} = n + 1$. Note that by the construction of Nishimoto and Tabei, every suffix at the bottom of a BWT run is the start of an input interval, $\{SA[l_2 - 1], SA[l_3 - 1], SA[l_4 - 1], \ldots, SA[l_r - 1], SA[n]\} \subseteq \{p^-_1, p^-_2, \ldots, p^-_k\}$. Then for any $i, j \in [p^-_x, p^-_{x+1} - 1], \phi^{-1}(i) -$

$\phi^{-1}(j) = i - j$. Therefore $\phi^{-1}(i) = \phi^{-1}(j) + (i - j)$ (see Lemma 4 in [36]). Below, we prove (ii) that for any $i, j \in [p_x^-, p_{x+1}^- - 1]$, $PLCP[\phi^{-1}(i)] - PLCP[\phi^{-1}(j)] = j - i$, therefore $PLCP[\phi^{-1}(i)] = PLCP[\phi^{-1}(j)] + j - i$.

▶ **Lemma 6.** *Let x be the integer satisfying $p_x^- \le i < p_{x+1}^-$ for some $i \in [1, n]$. Then (i) $PLCP[\phi^{-1}(i)] = PLCP[\phi^{-1}(p_x^-)] - (i - p_x^-)$. Therefore, (ii) for any $i, j \in [p_x^-, p_{x+1}^- - 1]$, $PLCP[\phi^{-1}(i)] = PLCP[\phi^{-1}(p_x^-)] - (i - p_x^-)$, $PLCP[\phi^{-1}(j)] = PLCP[\phi^{-1}(p_x^-)] - (j - p_x^-)$, and $PLCP[\phi^{-1}(i)] - PLCP[\phi^{-1}(j)] = j - i$.*

Proof. Lemma 6(i) clearly holds for $i = p_x^-$. We show that Lemma 6(i) holds for $p_x^- < i < p_{x+1}^-$ (i.e. $i \neq p_x^-$). Let s_t be the position in the SA with sa-value $p_x^- + t$ for an integer $t \in [1, y]$ where $y = i - p_x^-$. Two adjacent positions s_t and $s_t + 1$ are contained in an interval $[l_v, l_{v+1} - 1]$ corresponding to the v-th run in the BWT (L_v). This is because s_t is not the ending position of a run, $SA[s_t] \notin \{p_1^-, p_2^-, \ldots, p_{k^-}^-\}$. The LF function maps s_t to s_{t-1}, where s_0 is the position in the SA with value p_x^-. LF also maps $s_t + 1$ to $s_{t-1} + 1$ by Lemma 3(i) of [36]. $PLCP[\phi^{-1}(p_x^- + t - 1)] = LCP[s_{t-1} + 1] = LCP[s_t + 1] + 1 = PLCP[\phi^{-1}(p_x^- + t)] + 1$ since s_t and $s_t + 1$ are in the same interval in the BWT, L_v. These relationships produce y equalities $PLCP[\phi^{-1}(p_x^-)] = PLCP[\phi^{-1}(p_x^- + 1)] + 1$,$PLCP[\phi^{-1}(p_x^- + 1)] = PLCP[\phi^{-1}(p_x^- + 2)] + 1$,…,$PLCP[\phi^{-1}(p_x^- + y - 1)] = PLCP[\phi^{-1}(p_x^- + y)] + 1$. This leads to $PLCP[\phi^{-1}(p_x^-)] = PLCP[\phi^{-1}(p_x^- + y)] + y$. Which leads to $PLCP[\phi^{-1}(i)] = PLCP[\phi^{-1}(p_x^-)] - (i - p_x^-)$ by $y = i - p_x^-$ and $p_x^- + y = i$. ◀

Therefore, the move data structure that computes $\phi^{-1}(i)$, $F(I_{SA})$, can be modified to compute $PLCP[\phi^{-1}(i)]$ as well.

▶ **Lemma 7.** *$F(I_{SA})$ can be modified to compute $PLCP[\phi^{-1}(i)]$ as well as $\phi^{-1}(i)$ in constant time and $O(r)$ space given x, the index of the input interval of $B(I_{SA})$ that contains i. Call the modified move data structure $F(I_{\phi^{-1},PLCP})$.*

Proof. We modify the $F(I_{SA})$ move data structure by LCP^-, an array of size k^- where the x-th element stores the value $PLCP[\phi^{-1}(p_x^-)]$. Then, $PLCP[\phi^{-1}(i)]$ can be computed in constant time by evaluating $LCP^-[x] - (i - p_x^-)$ by $LCP^-[x] = PLCP[p_x^-]$ and Lemma 6(i). We call this modified move data structure $F(I_{\phi^{-1},PLCP})$. ◀

3.1.2 OptBWTRL

We slightly modify OptBWTR by adding a move data structure that computes ϕ and $PLCP$ and arrays that allow jumping to the closest input intervals corresponding to adjacent runs in the BWT in constant time. We call it *OptBWTRL*, L for LCP and $\mathcal{L}$ long LEMs. In addition to the structures of OptBWTR, OptBWTRL contains $F(I_{\phi,PLCP})$, ND, PD, SA^-, SA^+_ϕ, SA^-_{index}, and SA^-_ϕ. Furthermore, the $F(I_{SA})$ move data structure of OptBWTR is replaced by the $F(I_{\phi^{-1},PLCP})$ move data structure described in Lemma 7. Recall that $B(I_{LF})$ is the disjoint interval sequence the move data structure $F(I_{LF})$ is built on. Let $B(I_{LF})$ contain k input intervals where the i-th input interval is $[p_i, p_{i+1} - 1]$, and $p_{k+1} = n + 1$. Further recall that every input interval is contained in a run in the BWT, i.e. for all $i \in [1, k]$, $\forall j, j' \in [p_i, p_{i+1} - 1], L[j] = L[j']$. Then, ND and PD are arrays of length k where ND contains the index of the next input interval with a different character in the BWT and PD contains the index of the previous input interval with a different character in the BWT. Formally, for all $i \in [1, k]$, $ND[i] = \min\{j > i | L[p_i] \neq L[p_j]\}$, and $PD[i] = \max\{j < i | L[p_i] \neq L[p_j]\}$. If no such j exists, $ND[i] = k + 1$ and $PD[i] = -1$. ND and PD can be constructed in $O(k)$ (and therefore, $O(r)$) time given L_{first}. SA^- are samples of the SA at the ends of input intervals of $B(I_{LF})$. SA^+_ϕ are the indices of the input intervals of the top of $B(I_{LF})$ input interval suffix array samples in $F(I_{\phi,PLCP})$. SA^-_{index} and SA^-_ϕ are the indices of the input intervals of the bottom of $B(I_{LF})$ input interval suffix array samples in $F(I_{\phi^{-1},PLCP})$ and $F(I_{\phi,PLCP})$ respectively. Below, let $[p^+_i, p^+_{i+1}-1]$ and $[p^-_i, p^-_{i+1}-1]$ be the i-th input intervals of $F(I_{\phi,PLCP})$ and $F(I_{\phi^{-1},PLCP})$ respectively. Then, OptBWTRL differs from OptBWTR in the following ways.

- Replaced $F(I_{SA})$ with $F(I_{\phi^{-1},PLCP})$ from Lemma 7.
- Added $F(I_{\phi,PLCP})$ from Lemma 5.
- Added ND and PD. $ND[i] = \min_{j>i}\{L_{first}[j] \neq L_{first}[i] \text{ or } j = n+1\}$. $PD[i] = \max_{j>i}\{L_{first}[j] \neq L_{first}[i] \text{ or } j = -1\}$.
- Added SA^-. $SA^-[i] = SA[l_{i+1}-1]$.
- Added SA^+_ϕ. $SA^+_\phi[i] = j$ s.t. $p^+_j \leq SA^+[i] < p^+_{j+1}$.
- Added SA^-_{index}. $SA^-_{index}[i] = j$ s.t. $p^-_j \leq SA^-[i] < p^-_{j+1}$.
- Added SA^-_ϕ. $SA^-_\phi[i] = j$ s.t. $p^+_j \leq SA^-[i] < p^+_{j+1}$.

3.2 Computing Long LEMs

Here, we describe an algorithm for outputting all the long LEMs of a pattern P with respect to a text T in $O(m+occ)$ expected time using an index of size $O(r)$ words given the matching statistics of P with respect to T and an OptBWTRL of T. m is the length of P and occ is the number of long LEMs P has with T. Furthermore, the matching statistics are slightly augmented to contain the input intervals it's corresponding data are contained in. In particular, the input interval of $F(I_{LF})$ that $MS.row$ is contained in is stored as $MS.i$, the input interval of $F(I_{\phi,PLCP})$ that $MS.suff$ is contained in is stored as $MS.w$, and the input interval of $F(I_{\phi^{-1},PLCP})$ that $MS.suff$ is contained in is stored as $MS.x$. Note that the long LEM query algorithm we present here does not necessarily result in an $O(m+occ)$ expected time algorithm for outputting all long LEMs of P with respect to T given a OptBWTRL of T because an algorithm for computing the matching statistics of P with respect to T in $O(m)$ time and $O(r)$ space is not known.

We define the *balanced sa_{lcp}-interval* of a string P as a 13-tuple $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$ where $[b, e]$ is the sa-interval of P, $d \in [b, e]$, i, j, and k are the indexes of the input intervals of $B(I_{LF})$ that contain b, d, and e respectively, v and w are the indexes of the input intervals of $B(I_{\phi,PLCP})$ containing $SA[b]$ and $SA[d]$ respectively, and x and y are the indexes of the input intervals of $B(I_{\phi^{-1},PLCP})$ of $SA[d]$ and $SA[e]$ respectively. The balanced sa_{lcp}-interval keeps track of three positions in the sa-interval: the top (b), bottom (e), and the middle (d). d is any position in the interval, it may be equivalent to the top or the bottom. Each position also maintains its corresponding suffix array value and index of the input interval of the position in $F(I_{LF})$ (i, j, and k for top, middle, and bottom respectively). Finally, the top maintains the index of the input interval of its sa-value in $F(I_{\phi,PLCP})$ (v), the bottom maintains the index of the input interval of its sa-value in $F(I_{\phi^{-1},PLCP})$ (y), and the middle maintains the index of the input interval of its sa-value in both $F(I_{\phi,PLCP})$ and $F(I_{\phi^{-1},PLCP})$ (w and x respectively). The balanced sa_{lcp}-interval of a string P with no occurrences in T is undefined.

The high level idea of the long LEM finding algorithm is to compute the balanced sa_{lcp}-interval of adjacent substrings of length $\mathcal{L}$ of the pattern while outputting long LEMs along the way. I.E. given the balanced sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$, compute the sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ and output all long LEMs of the form $P[f+1, g] = T[f', g']$. We call this problem *long sa_{lcp}-interval advancement*. Given an algorithm for long sa_{lcp}-interval advancement in $O(t_{\mathcal{L}})$ time, a straightforward long LEM computation algorithm is iterating from $f = m \to 1$, repeatedly advancing the $\mathrm{sa_{lcp}}$-interval and outputting all long LEMs in $O(mt_{\mathcal{L}})$ time. In Section 3.2.1, we outline an algorithm for balanced $\mathrm{sa_{lcp}}$-interval extension and in Section 3.2.2, we outline an algorithm for long $\mathrm{sa_{lcp}}$-interval advancement. These algorithms result in an $O(m + occ)$ expected time algorithm for long LEM computation.

3.2.1 Balanced $\mathrm{sa_{lcp}}$-interval Extension

Here, we provide algorithms for obtaining the balanced $\mathrm{sa_{lcp}}$-interval of cP given the balanced $\mathrm{sa_{lcp}}$-interval of P and an OptBWTRL of T. The first algorithm runs in $O(\log\log_w \sigma)$ time by making use of the rank-select structure on L_{first}. The second runs in time linear to the number of runs in the balanced $\mathrm{sa_{lcp}}$-interval of P, r_P, by iterating through them. Call the balanced $\mathrm{sa_{lcp}}$-interval of P $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$ and the balanced $\mathrm{sa_{lcp}}$-interval of cP $(b', d', e', SA[b'], SA[d'], SA[e'], i', j', k', v', w', x', y')$. Recall that p_j, p_j^+, and p_j^- are the starting indexes of the j-th input intervals of $F(I_{LF}), F(I_{\phi,PLCP})$, and $F(I_{\phi^{-1},PLCP})$ respectively.

We first discuss the computation of the top values, $b', SA[b'], i'$, and v'. If $L[b] = c$, then $b' = LF[b]$ and i' can be computed with $F(I_{LF})$ in constant time using (b, i). $SA[b'] = SA[b] - 1$, and $v' = v$ if $SA[b] \neq p_v^+$, otherwise $v' = v - 1$. If $L[b] \neq c$, $b' = LF[\hat{b}]$, where $\hat{b}$ is the first location in $[b, e]$ such that $L[\hat{b}] = c$. If $\hat{\imath}$ is the index of first input interval $i \le \hat{\imath} \le k$ such that $L_{first}[\hat{\imath}] = c$, then $\hat{b} = p_{\hat{\imath}}$, where p_a is the starting position of the a-th input interval of $F(I_{LF})$. $\hat{\imath}$ can be computed in $O(\log\log_w \sigma)$ time using $R(L_{first})$ or $O(r_P)$ time by iterating through the runs of balanced $\mathrm{sa_{lcp}}$-interval of P using the ND array. Then, i' and $b' = LF[\hat{b}]$ can be computed with $F(I_{LF})$ in constant time using $(\hat{b}, \hat{\imath})$. $SA[b'] = SA^+[\hat{\imath}] - 1$, and $v' = SA_\phi^+[\hat{\imath}] - 1$.

The bottom values $e', SA[e'], k'$, and y' can be computed in a similar fashion. If $L[e] = c$, then $e' = LF[e]$ and k' can be computed with $F(I_{LF})$ in constant time using (e, k). $SA[e'] = SA[e] - 1$, and $y' = y$ if $SA[e] \neq p_k^-$, otherwise $y' = y - 1$. If $L[e] \neq c$, then $e' = LF[\hat{e}]$, where $\hat{e}$ is the last location in $[b, e]$ such that $L[\hat{e}] = c$. If $\hat{k}$ is the index of the last input input interval $i \le \hat{k} \le k$ such that $L_{first}[\hat{k}] = c$, then $\hat{e} = p_{\hat{k}+1} - 1$. $\hat{k}$ can be computed in $O(\log\log_w \sigma)$ time using $R(L_{first})$ or $O(r_p)$ time by iterating through the runs of the balanced $\mathrm{sa_{lcp}}$-interval of P using the PD array. Then, k' and $e' = LF[\hat{e}]$ can be computed with $F(I_{LF})$ in constant time using $(\hat{e}, \hat{k})$. Finally, $SA[e'] = SA^-[\hat{k}] - 1$, and $y' = SA_{index}^+[\hat{k}] - 1$.

Lastly, the middle values $d', SA[d'], j', w'$ and x' need to be computed. Pseudocode for middle value computation is provided as Algorithm 4 in Appendix B. If $L[d] = c$, then $d' = LF[d]$ and j' can be computed in constant time with $F(I_{LF})$ using (d, j). $SA[d'] = SA[d-1]$. $w' = w$ if $SA[d] \neq p_w^+$, otherwise $w' = w - 1$. Finally, $x' = x$ if $SA[d] \neq p_x^-$, otherwise $x' = x - 1$. If $L[d] \neq c$ and cP occurs in T, then there is a preceding or succeeding input interval of $B(I_{LF})$ that intersects with the balanced $\mathrm{sa_{lcp}}$-interval of P and has value c in the BWT. Suppose there is a preceding interval, $\hat{\jmath}$. Then the middle values can be updated similar to the bottom values. Let $\hat{d} = p_{\hat{\jmath}+1} - 1$, then j' and $d' = LF[\hat{d}]$ are computed in constant time with $F(I_{LF})$, $SA[d'] = SA^-[\hat{\jmath}] - 1$, $x' = SA_{index}^-[\hat{\jmath}] - 1$, and $w' = SA_\phi^-[\hat{\jmath}]$ if $SA[\hat{d}] \neq p_{SA_\phi^-[\hat{\jmath}]}^+$, otherwise $w' = SA_\phi^-[\hat{\jmath}] - 1$. If there is no preceding interval, then set $\hat{\jmath}$ to the index of the succeeding interval. Then the middle values can be updated similar to the top values. Let $\hat{d} = p_{\hat{\jmath}}$, then j' and $d' = LF[\hat{d}]$ are computed in constant time with $F(I_{LF})$, $SA[d'] = SA^+[\hat{\jmath}]$, $w' = SA_\phi^+[\hat{\jmath}] - 1$, and $x' = SA_{index}^+[\hat{\jmath}]$ if $SA[\hat{d}] \neq p_{SA_{index}^+[\hat{\jmath}]}^-$, otherwise $x' = SA_{index}^+[\hat{\jmath}] - 1$. The index, $\hat{\jmath}$, of the preceding or succeeding interval in the $\mathrm{sa_{lcp}}$-interval of P with value c in the BWT can be found in $O(\log\log_w \sigma)$ time with $R(L_{first})$ or $O(r_P)$

time by iterating through the runs in the BWT with PD and ND. Therefore, the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of cP can be computed in $O(\log\log_w \sigma)$ time or $O(r_P)$ time given the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of P. See Algorithm 3 in Appendix B for the $O(r_P)$ time algorithm pseudocode.

3.2.2 Long $\mathrm{sa}_{\mathrm{lcp}}$-interval Advancement

Let $occ_{start,f+1}$ be the number of long LEMs of the form $P[f+1,g] = T[f',g']$ and $occ_{end,f+\mathcal{L}-1}$. be the number of long LEMs of the form $P[h, f+\mathcal{L}-1] = T[f'',g'']$. Here, we describe an algorithm that computes the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f, f+\mathcal{L}-1]$ and a dynamic dictionary of the suffixes of T present in the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f, f+\mathcal{L}-1]$. This algorithm also outputs all $occ_{start,f+1}$ long LEMs of the form $P[f+1,g] = T[f',g']$. The algorithm runs in $O(occ_{start,f+1} + occ_{end,f+\mathcal{L}-1})$ expected time and requires as input the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f+1, f+\mathcal{L}]$, an OptBWTRL of T, and a dynamic dictionary of the suffixes of T present in the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f+1, f+\mathcal{L}]$.

We begin with the description of the dynamic dictionary, $dict_{occ}$. There are numerous dynamic dictionary data structures that support expected constant time insertion, deletion, and queries [5, 6, 14, 37]. Therefore, we maintain a dynamic dictionary of the suffixes in the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval. More precisely, if the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f+1, f+\mathcal{L}]$ is $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$, then the dynamic dictionary provided as input to the long $\mathrm{sa}_{\mathrm{lcp}}$-interval advancement algorithm has $e-b+1$ elements. $\forall a \in [b,e], SA[a]-(f+1)$ is contained in the dictionary and has the value $(f+1)+|lcp(T[SA[a],n], P[f+1,m])|-1 = f+|lcp(T[SA[a],n], P[f+1,m])|$ associated with it. I.E. the value associated with each suffix $SA[a]$ of the text contained in the dictionary is the ending position (in the pattern) of the longest match between suffix $SA[a]$ of the text and suffix $f+1$ of the pattern. It is not possible that multiple suffixes of T share the same key in $dict_{occ}$. Each suffix of T can occur only once in the dictionary because each suffix of T can occur only once in any balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval. Each suffix of T can occur only once in any balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval since each suffix of T occurs exactly once in SA.

Here, we describe the procedure for outputting all $occ_{start,f+1}$ long LEMs of the form $P[f+1,g] = T[f',g']$ in $O(occ_{start,f+1})$ expected time (we call this $outputMatches$). The high level idea is to iterate through the input intervals of $B(I_{LF})$, skipping intervals corresponding to a run of $P[f]$ in constant time per run using ND. We outline two functions: $outputMatchesDown(s,\iota,z)$ and $outputMatchesUp(s,\iota,z)$. For both functions, s represents a suffix of T and ι is the index of the input interval that contains it in $F(I_{\phi^{-1},PLCP})$ and $F(I_{\phi,PLCP})$ in $outputMatchesDown$ and $outputMatchesUp$ respectively. z represents the number of matches to output (directly above s in SA for $outputMatchesUp$ and directly below s in SA for $outputMatchesDown$) including s. $outputMatchesUp(s,\iota,1)$ outputs a match $P[f+1,g] = T[s, s+g-(f+1)]$, where $g = dict_{occ}[s-(f+1)]$, and removes the key-value pair $(s-(f+1), g)$ from $dict_{occ}$. $outputMatchesUp(s,\iota,z)$ for $z > 1$ similarly outputs a match $P[f+1,g] = T[s, s+g-(f+1)]$ where $g = dict_{occ}[s-(f+1)]$, then removes the key-value pair $(s-(f+1), g)$ from $dict_{occ}$. Then, it recurses on $outputMatchesUp(s',\iota',z-1)$, where ι' and $s' = \phi(s)$ are computed in constant time using $F(I_{\phi,PLCP})$. $outputMatchesDown(s,\iota,z)$ operates in the same way as $outputMatchesDown$ except it computes ϕ^{-1} instead of ϕ (using $F(I_{\phi^{-1},PLCP})$ instead of $F(I_{\phi,PLCP})$). It is simple to see that $outputMatchesUp(s,\iota,z)$ and $outputMatchesDown(s,\iota,z)$ operate in $O(z)$ expected time and output z matches each. Now we utilize $outputMatchesUp$ and $outputMatchesDown$ to output the $occ_{start,f+1}$ long LEMs of the form $P[f+1,g] = T[f',g']$. If the $\mathrm{sa}_{\mathrm{lcp}}$-interval of $P[f+1, f+\mathcal{L}]$ is fully contained in one input interval of $F(I_{LF})$, then $i = k$. If $L_{first}[i] = P[f]$, then there are no matches to output, otherwise, every suffix in the balanced $\mathrm{sa}_{\mathrm{lcp}}$-interval needs to be outputted and we do so by calling $outputMatchesUp(SA[d], w, d-b+1)$ and $outputMatchesDown(\phi^{-1}(SA[d]), x', e-b)$, where x' and $\phi^{-1}(SA[d])$ are computed with $(SA[d], x)$ and $F(I_{\phi^{-1},PLCP})$. In the case where the balanced sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$ is not fully contained in one input interval ($i \neq k$), we do the following. For the first input interval, i, if $L_{first}[i] \neq P[f]$, then the $p_{i+1} - b$ long LEMs starting at $SA[p_{i+1}-1], SA[p_{i+1}-2], \ldots, SA[b]$ in the text are outputted in $O(p_{i+1}-b)$ expected time by calling $outputMatchesUp(SA^{-}[i], SA^{-}_{\phi}[i], p_{i+1} - b)$. For any middle input interval o, $i < o < j$, if $L_{first}[o] = P[f]$, then this run in the BWT is skipped, $o = ND[o]$. Otherwise, if $L_{first}[o] \neq P[f]$, then the long LEMs starting at $SA[p_o], SA[p_o+1], \ldots, SA[p_{o+1}-1]$ are outputted by calling $outputMatchesDown(SA^{+}[o], SA^{+}_{index}[o], p_{o+1} - p_o)$. For the last input interval, k, if $L_{first}[k] \neq P[f]$, then the $e-p_k+1$ long LEMs starting at $SA[p_k], SA[p_k+1], \ldots, SA[e]$ are outputting by calling $outputMatchesDown(SA^{+}[k], SA^{+}_{index}[k], e - p_k + 1)$. Overall, outputting the $occ_{start,f+1}$ long LEMs of the form $P[f+1, g] = T[f', g']$ takes $O(occ_{start,f+1} + r_{P[f+1,f+\mathcal{L}]})$ expected time. Furthermore, for every run of character $P[f]$ intersecting the sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$ except the first one, there is a run of characters $\neq P[f]$. Therefore $r_{P[f+1,f+\mathcal{L}]} = O(occ_{start,f+1})$ Therefore outputting the $occ_{start,f+1}$ long LEMs of the form $P[f+1, g] = T[f', g']$ takes $O(occ_{start,f+1})$ expected time. See Algorithms 5–7 in Appendix B for *outputMatches* and related pseudocodes.

Finally, we must compute the balanced sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$. First suppose that the balanced sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$ is nonempty. Then, we use the algorithm described in Section 3.2.1 to obtain the sa_{lcp}-interval of $P[f, f+\mathcal{L}]$ in $O(r_{P[f+1,f+\mathcal{L}]})$ time. Now, let the sa_{lcp}-interval of $P[f, f+\mathcal{L}]$ be $(\hat{b}, \hat{d}, \hat{e}, SA[\hat{b}], SA[\hat{d}], SA[\hat{e}], \hat{\imath}, \hat{\jmath}, \hat{k}, \hat{v}, \hat{w}, \hat{x}, \hat{y})$ and the sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ be $(b', d', e', SA[b'], SA[d'], SA[e'], i', j', k', v', w', x', y')$. These sa_{lcp}-intervals differ only by those suffixes of the text whose lcp with $P[f, m]$ has length exactly $\mathcal{L}$. There are exactly $occ_{end,f+\mathcal{L}-1}$ such suffixes. Furthermore, $PLCP[SA[b']] < \mathcal{L}$ and $PLCP[\phi^{-1}(SA[e'])] < \mathcal{L}$. Finally, $\forall b' < a \leq \hat{b}$, $LCP[a] = PLCP[SA[a]] \geq \mathcal{L}$, and $\forall \hat{e} \leq a < e'$, $LCP[a+1] = PLCP[SA[a+1]] = PLCP[\phi^{-1}(a)] \geq \mathcal{L}$. Therefore, we initialize $b' = \hat{b}$, $SA[b'] = SA[\hat{b}]$, $i' = \hat{\imath}$, and $v' = \hat{v}$. Then, while $LCP[b'] = PLCP[SA[b']] \geq \mathcal{L}$, we (i) set $i' = i' - 1$ if $b' = p_{i'}$, (ii) set $b' = b' - 1$, (iii) update $SA[b']$ and v' by $F(I_{\phi,PLCP})$, and (iv) insert the key $SA[b']-f$ into $dict_{occ}$ with value $f+\mathcal{L}-1$. When $LCP[b'] = PLCP[SA[b']] < \mathcal{L}$, the final value b' has been computed. Similarly for e', we initialize $e' = \hat{e}, SA[e'] = SA[\hat{e}], k' = \hat{k}$, and $y' = \hat{y}$. Then, while $LCP[e'+1] = PLCP[SA[e'+1]] = PLCP[\phi(SA[e'])] \geq \mathcal{L}$, we (i) set $k' = k' - 1$ if $e' = p_{k'+1} - 1$, (ii) set $e' = e' - 1$, (iii) update $SA[e']$ and y' by $F(I_{\phi^{-1},PLCP})$, and (iv) insert the key $SA[e'] - f$ into $dict_{occ}$ with value $f + \mathcal{L} - 1$. When $LCP[e'+1] = PLCP[SA[e'+1]] = PLCP[\phi^{-1}(e')] < \mathcal{L}$, the final value e' has been computed. This takes constant time per suffix added to the interval, therefore $O(occ_{end,f+\mathcal{L}-1})$ time.

If the balanced sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$ is empty, the balanced sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ is only nonempty if $MS[f].len = \mathcal{L}$. If it is, we initialize the balanced sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ to $(\hat{b} = MS[f].row, \hat{d} = MS[f].row, \hat{e} = MS[f].row, SA[\hat{b}] = MS[f].suff, SA[\hat{d}] = MS[f].suff, SA[\hat{e}] = MS[f].suff, \hat{\imath} = MS[f].i, \hat{\jmath} = MS[f].i, \hat{k} = MS[f].i, \hat{v} = MS[f].w, \hat{w} = MS[f].w, \hat{x} = MS[f].x, \hat{y} = MS[f].x)$ and insert the key $MS.suff - f$ into $dict_{occ}$ with value $f + \mathcal{L} - 1$. Then, the interval is expanded to the sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ in $O(occ_{end,f+\mathcal{L}-1})$ time as in the other case.

In the case where the balanced sa_{lcp}-interval of $P[f+1, f+\mathcal{L}]$ is empty, long sa_{lcp}-interval advancement is performed in $O(occ_{end,f+\mathcal{L}-1})$ expected time . If it is not empty, the algorithm we have described first performs sa_{lcp}-interval extension, obtaining the sa_{lcp}-interval of $P[f, f+\mathcal{L}]$ in $O(r_{P[f+1,f+\mathcal{L}]})$ time and then takes $O(occ_{end,f+\mathcal{L}-1})$ expected time to compute the sa_{lcp}-interval of $P[f, f+\mathcal{L}-1]$ from the sa_{lcp}-interval of $P[f, f+\mathcal{L}]$. Finally,

$r_{P[f+1,f+\mathcal{L}]} = O(occ_{end,f+\mathcal{L}-1})$. Therefore, the algorithm described here performs the long $\mathrm{sa}_{\mathrm{lcp}}$-interval advancement in $O(occ_{start,f+1} + occ_{end,f+\mathcal{L}-1})$ expected time. See Algorithm 2 in the Appendix for the pseudocode of this algorithm.

3.3 Time Complexity

If the above algorithm is iterated from $f = m \rightarrow 0$, all long MEMs of the pattern with respect to the text are outputted. The time complexity of the algorithm is the sum of the time complexity of the m long $\mathrm{sa}_{\mathrm{lcp}}$-interval advancements. Note that the sum of $occ_{start,f+1}$ for $f = m \rightarrow 0$ is occ and the sum of the $occ_{end,f+\mathcal{L}-1}$ for $f = m \rightarrow 0$ is also occ. Therefore, the time complexity of the algorithm overall is $O(m + occ)$ expected time. See Algorithm 1 in Appendix B for pseudocode. The algorithm takes $O(r)$ space for the OptBWTRL and $O(occ)$ space for maintaining the dynamic dictionary [5]. Also note that if a deterministic time bound is desired, this algorithm runs in $O\left(m + occ\sqrt{\frac{\log occ}{\log\log occ}}\right)$ time with the same space by replacing the dictionary with a deterministic dictionary implemented by exponential search trees [2, 47]. Recall these complexities are when given the modified matching statistics. A linear time algorithm for computing matching statistics in $O(r)$ space is not known. However, note that since the values of matching statistics are only needed for positions i where $MS[i].len = \mathcal{L}$, a straightforward algorithm for long LEM query follows from our algorithm in $O(m\mathcal{L}\log\log_w \sigma + occ)$ expected time when matching statistics are not given as input. This algorithm is obtained by computing the $\mathrm{sa}_{\mathrm{lcp}}$-interval of each $P[i, i + \mathcal{L} - 1]$ independently in $O(\mathcal{L}\log\log_w \sigma)$ time using the standard count algorithm described by Nishimoto and Tabei [36] followed by performing the long LEM query described here. The long LEM query algorithm described here results in an $O(m + occ)$ expected time long LEM query algorithm in uncompressed string indexes since algorithms for $O(m)$ time matching statistics computation are known in uncompressed space.

4 Discussion

In this paper, we have described OptBWTRL, a modification of OptBWTR by Nishimoto and Tabei [36]. OptBWTRL adds the ability to compute $PLCP$ and ϕ in constant time with additional move data structures. It also retains a space complexity of $O(r)$ words. We also define locally maximal exact matches (LEMs), a match that cannot be simultaneously extended in the pattern and the text instead of one that is only unable to be extended in the pattern (MEMs). Finally, we describe an algorithm for outputting all LEMs with length at least $\mathcal{L}$ in $O(m + occ)$ expected time given an OptBWTRL of the text and the matching statistics of the pattern with respect to the text. Note that this doesn't result in a linear time algorithm for computing long LEMs in $O(m+occ)$ expected time in $O(r)$ space because an algorithm for computing matching statistics of a pattern with respect to a text in linear time in $O(r)$ space is not known. A deterministic bound for our long LEM query algorithm is $O\left(m + occ\sqrt{\frac{\log occ}{\log\log occ}}\right)$. Finally, our long LEM query admits a direct computation of long LEMs in $O(m\mathcal{L} + occ)$ expected time without being provided matching statistics as input. This algorithm may be faster than computation of matching statistics followed by $O(m+occ)$ long LEM query in some cases, especially when $\mathcal{L}$ is small.

It is likely that the move data structures $F(I_{\phi,PLCP})$ and $F(I_{\phi^{-1},PLCP})$ can be merged into one data structure that still takes $O(r)$ space and computes $\phi, \phi^{-1}, PLCP[i]$, and $PLCP[\phi^{-1}(i)]$ in constant time in one data structure. This would greatly reduce the number of samples needed per input interval $F(I_{LF})$. It would also allow bidirectional movement in the SA with one input interval index. This is left as future work. Possible other future work includes a practical implementation of the structures and algorithms described here, possibly as a modification of MOVI or b-move [16, 49]. Thirdly, the ability to compute $PLCP$ in constant time may speed up matching statistics computation in compressed space. The intuition is that when $MS[i].len \leq MS[i+1].len$, then when $MS[i].len$ is large, the sa-interval of $P[i, i + MS[i].len - 1]$ is small and is faster to compute with $PLCP$ and ϕ and ϕ^{-1} than with reverse LF. When $MS[i].len$ is small, computing the sa-interval is faster with reverse LF. In Heng Li's forward-backward algorithm [25], the new sa-interval is always computed by reverse LF. Computing the sa-interval with $PLCP$ and reverse LF simultaneously is likely to be faster in practice than reverse LF alone while retaining the same worst case time complexity. The authors are currently exploring this idea. Furthermore, variable length threshold long LEMs may be useful. I.E. output long LEMs that are $x\%$ of the length of the MEMs in the same area. The authors believe a linear time algorithm for this or a similar problem given matching statistics exist. Finally, applications utilizing the long LEMs of a pattern with respect to the text is a possible fruitful direction for future work. Particularly in biological applications.

Long LEMs may have many biological applications. In general, in any application where long MEMs are used, long LEMs may also be used. Note that MEMs are a subset of LEMs and long MEMs are a subset of long LEMs. For example, in biobank scale haplotype datasets, long matches (long LEMs) in the PBWT have revealed genealogical relationships that set maximal matches (MEMs) are not able to uncover. As the compressive power of compressed string indexes increases and the number of variants in biobank scale whole genome sequencing data increases, storing unaligned genomes becomes more viable. In that case, algorithms for outputting long LEMs are needed to replace the long match algorithms in the PBWT. These matches have many applications from identity by descent segment detection, haplotype phasing, haplotype imputation, inferring genealogical relationships, and ancestry inference. Utilizing unaligned matches from a large collection of haplotype sequences instead of aligned matches from haplotypes aligned to a linear reference genome may also reduce reference bias. Finally, novel applications for long LEMs may exist, long LEMs may be used as seeds for seed and extend algorithms. They may be used as anchors for approximate matching matching algorithms [20] possibly for long read alignment to either a reference pangenome or a linear reference genome [31]. Lastly, genome to genome or genome to pangenome long LEMs detection may find similar sequences in the genomes on different genomic regions. MEMs detection may miss these similar sequences on different genomic regions because these matches will typically be overshadowed by larger encompassing matches that occur in roughly the same region in the pattern and the text. The long LEMs may therefore reveal old structural variants that MEMs and general alignment algorithms are both unable to reveal. MEMs don't reveal these variants due to looking for only the largest matches in a region on the pattern while alignment algorithms don't due to better alignments existing in closeby genomic regions or alignment algorithms being too computationally expensive to run on very large datasets.

Overall, we have provided a linear time algorithm for outputting all long LEMs of a pattern with respect to a text in BWT runs compressed space given the matching statistics of the pattern with respect to the text. We have also applied the move data structure of Nishimoto and Tabei to computation of $PLCP$ in constant time. Therefore, we can compute $LCP[i]$ given $SA[i]$ in constant time. We apply these results to modify the OptBWTR, creating OptBWTRL. OptBWTRL is an $O(r)$ space data structure that computes ϕ and $PLCP$ in constant time and long LEMs in linear time given matching statistics. These algorithms result in a linear time long LEM query algorithm in uncompressed string indexes.

References

1 Omar Y Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. SPUMONI 2: improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24(1):122, 2023. doi:10.1186/s13059-023-02958-1.

2 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13–es, June 2007. doi:10.1145/1236457.1236460.

3 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching*, pages 26–39, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-19929-0_3.

4 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez, Simon J. Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

5 Michael A. Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, pages 1284–1297, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519935.3519969.

6 Michael A. Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. *Modern Hashing Made Simple*, pages 363–373. Society for Industrial and Applied Mathematics, 2024. doi:10.1137/1.9781611977936.33.

7 Alexander G. Bick, Ginger A. Metcalf, Kelsey R. Mayo, Lee Lichtenstein, Shimon Rura, Robert J. Carroll, Anjene Musick, Jodell E. Linder, I. King Jordan, Shashwat Deepali Nagar, Shivam Sharma, Robert Meller, Melissa Basford, Eric Boerwinkle, Mine S. Cicek, Kimberly F. Doheny, Evan E. Eichler, Stacey Gabriel, Richard A. Gibbs, David Glazer, Paul A. Harris, Gail P. Jarvik, Anthony Philippakis, Heidi L. Rehm, Dan M. Roden, Stephen N. Thibodeau, Scott Topper, Ashley L. Blegen, Samantha J. Wirkus, Victoria A. Wagner, Jeffrey G. Meyer, Donna M. Muzny, Eric Venner, Michelle Z. Mawhinney, Sean M. L. Griffith, Elvin Hsu, Hua Ling, Marcia K. Adams, Kimberly Walker, Jianhong Hu, Harsha Doddapaneni, Christie L. Kovar, Mullai Murugan, Shannon Dugan, Ziad Khan, Niall J. Lennon, Christina Austin-Tse, Eric Banks, Michael Gatzen, Namrata Gupta, Emma Henricks, Katie Larsson, Sheli McDonough, Steven M. Harrison, Christopher Kachulis, Matthew S. Lebo, Cynthia L. Neben, Marcie Steeves, Alicia Y. Zhou, Joshua D. Smith, Christian D. Frazar, Colleen P. Davis, Karynne E. Patterson, Marsha M. Wheeler, Sean McGee, Christina M. Lockwood, Brian H. Shirts, Colin C. Pritchard, Mitzi L. Murray, Valeria Vasta, Dru Leistritz, Matthew A. Richardson, Jillian G. Buchan, Aparna Radhakrishnan, Niklas Krumm, Brenna W. Ehmen, Sophie Schwartz, M. Morgan T. Aster, Kristian Cibulskis, Andrea Haessly, Rebecca Asch, Aurora Cremer, Kylee Degatano, Akum Shergill, Laura D. Gauthier, Samuel K. Lee, Aaron Hatcher, George B. Grant, Genevieve R. Brandt, Miguel Covarrubias, Ashley Able, Ashley E. Green, Jennifer Zhang, Henry R. Condon, Yuanyuan Wang, Moira K. Dillon, C. H. Albach, Wail Baalawi, Seung Hoan Choi, Xin Wang, Elisabeth A. Rosenthal, Andrea H. Ramirez, Sokny Lim, Siddhartha Nambiar, Bradley Ozenberger, Anastasia L. Wise, Chris Lunt, Geoffrey S. Ginsburg, Joshua C. Denny, The All of Us Research Program Genomics Investigators, Manuscript Writing Group, All of Us Research Program Genomics Principal Investigators, Mayo Biobank, Genome Center: Baylor-Hopkins Clinical Genome Center, Color Genome Center: Broad, Mass General Brigham Laboratory for Molecular Medicine, Genome Center: University of Washington, Data, Research Center, All of Us Research Demonstration Project Teams, and NIH All of Us Research Program Staff. Genomic data in the All of Us Research Program. *Nature*, 627(8003):340–346, March 2024. doi:10.1038/s41586-023-06957-x.

8 Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. Data structures for SMEM-finding in the PBWT. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 89–101, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43980-3_8.

9 Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT Tricks. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2022.16.

10 Nathaniel K. Brown, Vikram S. Shivakumar, and Ben Langmead. Improved pangenomic classification accuracy with chain statistics. In Sriram Sankararaman, editor, *Research in Computational Molecular Biology*, pages 190–208, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-90252-9_12.

11 Michael Burrows. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

12 Human Pangenome Reference Consortium. HPRC data release 2. [online]. URL: https://humanpangenome.org/hprc-data-release-2/.

13 Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ-PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data. *Bioinformatics*, 39(9):btad552, September 2023. doi:10.1093/bioinformatics/btad552.

14 Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pǎtraşcu. De dictionariis dynamicis pauco spatio utentibus. In José R. Correa, Alejandro Hevia, and Marcos Kiwi, editors, *LATIN 2006: Theoretical Informatics*, pages 349–361, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11682462_34.

15 Lore Depuydt, Omar Y. Ahmed, Jan Fostier, Ben Langmead, and Travis Gagie. Run-length compressed metagenomic read classification with SMEM-finding and tagging. *bioRxiv*, 2025. doi:10.1101/2025.02.25.640119.

16 Lore Depuydt, Luca Renders, Simon Van de Vyver, Lennart Veys, Travis Gagie, and Jan Fostier. b-move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2024.10.

17 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, January 2014. doi:10.1093/bioinformatics/btu014.

18 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.

19 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1), January 2020. doi:10.1145/3375890.

20 Chirag Jain, Daniel Gibney, and Sharma V. Thankachan. Algorithms for colinear chaining with overlaps and gap costs. *Journal of Computational Biology*, 29(11):1237–1251, 2022. doi:10.1089/cmb.2022.0266.

21 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02441-2_17.

22 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005. Combinatorial Pattern Matching (CPM) Special Issue. doi:10.1016/j.jda.2004.08.002.

23 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In Yoshiharu Kohayakawa and Flávio Keidi Miyazawa, editors, *LATIN 2020: Theoretical Informatics*, pages 207–219, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61792-9_17.

24 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.

25 Heng Li. Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, May 2012. doi:10.1093/bioinformatics/bts280.

26 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, November 2024. doi:10.1093/bioinformatics/btae717.

27 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, May 2009. doi:10.1093/bioinformatics/btp324.

28 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, January 2010. doi:10.1093/bioinformatics/btp698.

29 Shuwei Li, Keren J Carss, Bjarni V Halldorsson, Adrian Cortes, and UK Biobank Whole-Genome Sequencing Consortium. Whole-genome sequencing of half-a-million UK Biobank participants. *medRxiv*, 2023. doi:10.1101/2023.12.06.23299426.

30 Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023. doi:10.1038/s41586-023-05896-x.

31 Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, September 2012. doi:10.1093/bioinformatics/bts414.

32 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. doi:10.1089/cmb.2009.0169.

33 Karen H Miga and Ting Wang. The need for a human pangenome reference sequence. *Annual Review of Genomics and Human Genetics*, 22(1):81–102, 2021. doi:10.1146/annurev-genom-120120-081921.

34 Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, July 2019. doi:10.1093/bioinformatics/btz347.

35 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2), March 2021. doi:10.1145/3434399.

36 Takaaki Nishimoto and Yasuo Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101:1–101:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.101.

37 Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming*, pages 357–368, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/3-540-45061-0_30.

38 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022. PMID: 35041495. doi:10.1089/cmb.2021.0290.

39 Ahsan Sanaullah, Seba Villalobos, Degui Zhi, and Shaojie Zhang. Haplotype matching with GBWT for pangenome graphs. *bioRxiv*, 2025. doi:10.1101/2025.02.03.634410.

40 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-PBWT: dynamic positional burrows–wheeler transform. *Bioinformatics*, 37(16):2390–2397, February 2021. doi:10.1093/bioinformatics/btab117.

41 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. An efficient data structure and algorithm for long-match query in run-length compressed BWT, 2025. doi:10.48550/arXiv.2505.15698.

42 Pramesh Shakya, Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Dynamic μ-PBWT: Dynamic run-length compressed pbwt for biobank scale data. In Sriram Sankararaman, editor, *Research in Computational Molecular Biology*, pages 209–226, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-90252-9_13.

43 Vipin Singh, Shweta Pandey, and Anshu Bhardwaj. From the reference human genome to human pangenome: Premise, promise and challenge. *Frontiers in Genetics*, 13:1042550, 2022. doi:10.3389/fgene.2022.1042550.

44 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, July 2019. doi:10.1093/bioinformatics/btz575.

45 Li Song and Ben Langmead. Centrifuger: lossless compression of microbial genomes for efficient and accurate metagenomic sequence classification. *Genome Biology*, 25(1):106, 2024. doi:10.1007/978-1-0716-3989-4_22.

46 Dylan J. Taylor, Jordan M. Eizenga, Qiuhui Li, Arun Das, Katharine M. Jenike, Eimear E. Kenny, Karen H. Miga, Jean Monlong, Rajiv C. McCoy, Benedict Paten, and Michael C. Schatz. Beyond the Human Genome Project: The age of complete human genome sequences and pangenome references. *Annual Review of Genomics and Human Genetics*, 25(Volume 25, 2024):77–104, 2024. doi:10.1146/annurev-genom-021623-081639.

47 Mikkel Thorup. Mihai pătraşcu: obituary and open problems. *SIGACT News*, 44(1):110–114, March 2013. doi:10.1145/2447712.2447737.

48 Victor Wang, Ardalan Naseri, Shaojie Zhang, and Degui Zhi. Syllable-PBWT for space-efficient haplotype long-match query. *Bioinformatics*, 39(1):btac734, November 2022. doi:10.1093/bioinformatics/btac734.

49 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27(12), 2024. doi:10.1016/j.isci.2024.111464.

A Proofs

▶ **Lemma 3.** *The following three statements hold: (i) Let x be the integer satisfying $p^+_x \leq i < p^+_{x+1}$ for some integer $i \in [1, n]$. Then $\phi(i) = \phi(p^+_x) + (i - p^+_x)$; (ii) $\phi(p^+_{\delta^+[1]}) = 1$ and $\phi(p^+_{\delta^+[i]}) = \phi(p^+_{\delta^+[i-1]}) + d$ where $d = p^+_{\delta^+[i-1]+1} - p^+_{\delta^+[i-1]}$; (iii) $p^+_1 = 1$.*

Proof. (i) Lemma 3(i) clearly holds for $i = p^+_x$. We show that Lemma 3(i) holds for $i \neq p^+_x$ (i.e., $i > p^+_x$). Let s_t be the position in SA with sa-value $p^+_x + t$ for an integer $t \in [1, y]$ (i.e., $SA[s_t] = p^+_x + t$), where $y = i - p^+_x$. Two adjacent positions s_t and $s_t - 1$ are contained in an interval $[l_v, l_v + |L_v| - 1]$ on SA (i.e., $s_t, s_t - 1 \in [l_v, l_v + |L_v| - 1]$), which corresponds to the v-th run L_v of L. This is because s_t is not the starting position of a run, i.e. $(SA[s_t] = p^+_x + t) \notin \{p^+_1, p^+_2, \ldots, p^+_r\}$. The LF function maps s_t to s_{t-1}, where s_0 is the position with sa-value v_x. LF also maps $s_t - 1$ to $s_{t-1} - 1$, by Lemma 3(i) of [36]. The two mapping relationships established by LF produce y equalities $\phi(SA[s_1]) = \phi(SA[s_0]) + 1, \phi(SA[s_2]) = \phi(SA[s_1]) + 1, \ldots, \phi(SA[s_y]) = \phi(SA[s_{y-1}]) + 1$. The equalities lead to $\phi(SA[s_y]) = \phi(SA[s_0]) + y$, which represents $\phi(i) = \phi(p^+_x) + (i - p^+_x)$ by $SA[s_y] = i, SA[s_0] = p^+_x$, and $y = i - p^+_x$.

(ii) Let p be the integer satisfying $L_p = \$$. Then there exists an integer q such that p^+_q is the sa-value at position $l_p + 1$ ($l_p + 1 = l_{p+1}$) if $p \neq r$; otherwise if $p = r$, p^+_q is the sa-value at position 1 and $q = 1$. $\phi(p^+_q) = 1$, because $SA[l_p] = 1$ always holds. Hence $\phi(p^+_{\delta^+[1]}) = 1$ holds by $\delta^+[1] = q$.

Next, $\phi(p^+_{\delta+[i]}) = \phi(p^+_{\delta+[i-1]}) + d$ holds for any $i \in [2, r]$ because (a) ϕ maps the interval $[p^+_{\delta+[i]}, p^+_{\delta+[i]} + d - 1]$ into the interval $[\phi(p^+_{\delta+[i]}), \phi(p^+_{\delta+[i]}) + d - 1]$ by Lemma 3(i) for any $i \in [1, r]$, (b) ϕ is a bijection from $[1, n]$ to $[1, n]$, and (c) $\phi(p^+_{\delta+[1]}) < \phi(p^+_{\delta+[2]}) < \cdots < \phi(p^+_{\delta+[r]})$ holds.

(iii) Recall that p is an integer satisfying $L_p = \$$. Then there exists an integer q' such that $p^+_{q'}$ is the sa-value at position l_p. Finally, recall $SA[l_p] = 1$. Hence, $v_1 = v_{q'} = 1$ holds. ◀

B Algorithm Pseudocodes

Pseudocode for the long LEM query algorithm is provided in Algorithm 1. The long LEM query algorithm is separated into subroutines to aid understanding. The long LEM query algorithm is provided as input the length threshold $\mathcal{L}$, the query/pattern P, and an OptBWTRL of the text T. Subroutines have access to their calling function's variables implicitly. Structures of the OptBWTRL of T are referenced directly, i.e., $ND[i]$ instead of OptBWTRL$_T$.$ND[i]$. Finally, operations on move data structures use the notation of Nishimoto and Tabei [36].

Algorithm 1 LongLEMQuery: Long LEM Query.

```
/* Input: ℒ: length threshold, P: pattern, and OptBWTRL of T    */
/* Outputs all long LEMs of P to T with length at least ℒ       */
(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) = ∅
for f = |P| → 1 do
  | (b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) = LongAdvance(P[f], MS[f]);
OutputMatches(#)  // # is any character that doesn't occur in the text
```

Algorithm 2 LongAdvance: Long $\mathrm{sa}_{\mathrm{lcp}}$-interval Advancement.

/* Input: c: character to advance by, ms: one element of the matching statistics of P */

/* Subroutine of Algorithm 1, implicitly has access to $\mathcal{L}$ and $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$ */

OutputMatches(c)

if $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) \neq \emptyset$ **then**

  $(\hat{b}, \hat{d}, \hat{e}, SA[\hat{b}], SA[\hat{d}], SA[\hat{e}], \hat{\imath}, \hat{\jmath}, \hat{k}, \hat{v}, \hat{w}, \hat{x}, \hat{y}) =$ BalancedExtend$(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$

if $ms.len = \mathcal{L}$ **then**

  // Note that $ms.len = \mathcal{L}$ iff $(\hat{b}, \hat{d}, \hat{e}, SA[\hat{b}], SA[\hat{d}], SA[\hat{e}], \hat{\imath}, \hat{\jmath}, \hat{k}, \hat{v}, \hat{w}, \hat{x}, \hat{y}) = \emptyset$

  $\hat{b} = \hat{d} = \hat{e} = ms.row$

  $SA[\hat{b}] = SA[\hat{d}] = SA[\hat{e}] = ms.suff$

  $\hat{\imath} = \hat{\jmath} = \hat{k} = ms.i$

  $\hat{v} = \hat{w} = ms.w$

  $\hat{x} = \hat{x} = ms.x$

if $(\hat{b}, \hat{d}, \hat{e}, SA[\hat{b}], SA[\hat{d}], SA[\hat{e}], \hat{\imath}, \hat{\jmath}, \hat{k}, \hat{v}, \hat{w}, \hat{x}, \hat{y}) \neq \emptyset$ **then**

  $(s', v', lcp) = Move(B(I_{\phi,PLCP}), SA[\hat{b}], \hat{v})$

  while $lcp \geq \mathcal{L}$ **do** // Expand block boundary upwards

    if $\hat{b} = F(I_{LF}).D_{pair}[\hat{v}].first$ **then** $\hat{v} = \hat{v} - 1$

    $\hat{b} = \hat{b} - 1$

    $SA[\hat{b}] = s'$

    $\hat{v} = v'$

    $dict_{occ}[s' - f] = f + \mathcal{L} - 1$

    $(s', v', lcp) = Move(B(I_{\phi,PLCP}), SA[\hat{b}], \hat{v})$

  $(s', y', lcp) = Move(B(I_{\phi^{-1},PLCP}), SA[\hat{e}], \hat{y})$

  while $lcp \geq \mathcal{L}$ **do** // Expand block boundary downwards

    $\hat{e} = \hat{e} + 1$

    if $\hat{e} = B(I_{LF}).D_{pair}[\hat{y} + 1].first$ **then** $\hat{y} = \hat{y} + 1$

    $SA[\hat{e}] = s'$

    $\hat{y} = y'$

    $dict_{occ}[s' - f] = f + \mathcal{L} - 1$

    $(s', y', lcp) = Move(B(I_{\phi^{-1},PLCP}), SA[\hat{e}], \hat{y})$

return $(\hat{b}, \hat{d}, \hat{e}, SA[\hat{b}], SA[\hat{d}], SA[\hat{e}], \hat{\imath}, \hat{\jmath}, \hat{k}, \hat{v}, \hat{w}, \hat{x}, \hat{y})$

Algorithm 3 BalancedExtend: Balanced $\mathtt{sa_{lcp}}$-interval extension.

/* Input: c: character to extend by,
$(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y)$: balanced $\mathtt{sa_{lcp}}$-interval of P',
and OptBWTRL of T */
/* Output: $(b', d', e', SA[b'], SA[d'], SA[e'], i', j', k', v', w', x', y')$: balanced
$\mathtt{sa_{lcp}}$-interval of cP' */
if $(b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) = \emptyset$ **then** return
$\hat{b} = b,\ SA[\hat{b}] = SA[b],\ \hat{\imath} = i,\ \hat{v} = v$
while $\hat{\imath} \leq k$ **do** // compute $\hat{b}$, $SA[\hat{b}]$, $\hat{\imath}$, and $\hat{v}$
  if $L_{first}[\hat{b}] = c$ **then** break
  $\hat{\imath} = ND[\hat{\imath}]$
  $\hat{b} = F(I_{LF}).D_{pair}[\hat{\imath}].first$
  $SA[\hat{b}] = SA^{+}[\hat{\imath}]$
  $\hat{v} = SA^{+}_{\phi}[\hat{\imath}]$
if $\hat{\imath} > k$ **then** return $\emptyset$ // cP doesn't exist in T
$(b', i') = Move(B(I_{LF}), \hat{b}, \hat{\imath})$
$SA[b'] = SA[\hat{b}] - 1$
if $SA[\hat{b}] = F(I_{\phi,PLCP}).D_{pair}[\hat{v}].first$ **then** $v' = \hat{v} - 1$
else $v' = \hat{v}$
$\hat{e} = e,\ SA[\hat{e}] = SA[e],\ \hat{k} = k,\ \hat{y} = y$
while $\hat{k} \geq i$ **do** // compute $\hat{e}$, $SA[\hat{e}]$, $\hat{k}$, and $\hat{y}$
  if $L_{first}[\hat{e}] = c$ **then** break
  $\hat{k} = PD[\hat{k}$
  $\hat{e} = F(I_{LF}).D_{pair}[\hat{k} + 1].first - 1$
  $SA[\hat{e}] = SA^{-}[\hat{k}]$
  $\hat{y} = SA^{-}_{index}[\hat{k}]$
$(e', k') = Move(B(I_{LF}), \hat{e}, \hat{k})$
$SA[e'] = SA[\hat{e}] - 1$
if $SA[\hat{e}] = F(I_{\phi^{-1},PLCP}).D_{pair}[\hat{y}].first$ **then** $y' = \hat{y} - 1$
else $y' = \hat{y}$
$(d', SA[d'], j', w', x') = ComputeMiddle()$
return $(b', d', e', SA[b'], SA[d'], SA[e'], i', j', k', v', w', x'y')$

Algorithm 4 ComputeMiddle.

```
// Subroutine of Algorithm 3, has access to everything in Algorithm 3
```
$\hat{d} = d$, $SA[\hat{d}] = SA[d]$, $\hat{j} = j$, $\hat{w} = w$, $\hat{x} = x$

/* compute $\hat{d}$, $SA[\hat{d}]$, $\hat{j}$, $\hat{w}$, and $\hat{x}$ */

while $\hat{j} \geq i$ **do** // try traversing upwards

  if $L_{first}[[\hat{j}]] = c$ **then** break

  $\hat{j} = PD[\hat{j}]$

  $\hat{d} = F(I_{LF}).D_{pair}[\hat{j} + 1].first - 1$

  $SA[\hat{d}] = SA^{-}[\hat{j}]$

  $\hat{w} = SA^{-}_{\phi}[\hat{j}]$

  $\hat{x} = SA^{-}_{index}[\hat{j}]$

if $\hat{j} < i$ **then**

  $\hat{d} = d$, $SA[\hat{d}] = SA[d]$, $\hat{j} = j$, $\hat{w} = w$, $\hat{x} = x$

  while $\hat{j} \leq k$ **do** // traverse downwards

    if $L_{first}[\hat{j}]$ **then** break

    $\hat{j} = ND[\hat{j}]$

    $\hat{d} = F(I_{LF}).D_{pair}[\hat{j}].first$

    $SA[\hat{d}] = SA^{+}[\hat{j}]$

    $\hat{w} = SA^{+}_{\phi}[\hat{j}]$

    $\hat{x} = SA^{+}_{index}[\hat{j}]$

$(d', j') = Move(B(I_{LF}), \hat{d}, \hat{j})$

$SA[d'] = SA[\hat{d}] - 1$

if $SA[\hat{d}] = F(I_{\phi,PLCP}).D_{pair}[\hat{w}].first$ **then** $w' = \hat{w} - 1$

else $w' = \hat{w}$

if $SA[\hat{d}] = F(I_{\phi^{-1},PLCP}).D_{pair}[\hat{x}].first$ **then** $x' = \hat{x} - 1$

else $x' = \hat{x}$

return $(d', SA[d'], j', w', x')$

Algorithm 5 OutputMatches.

```
/* Input: c                                                              */
/* Subroutine of Algorithm 2, implicitly has access to
   (b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) and OptBWTRL of T  */
if (b, d, e, SA[b], SA[d], SA[e], i, j, k, v, w, x, y) = ∅ then return
if i = k then
    if L_first[i] ≠ c then
        OutputMatchesUp(SA[d], w, d − b + 1)
        (s', x') = Move(B(I_{φ^{-1},PLCP}), SA[d], x)
        OutputMatchesDown(s', x', e − d)
else
    if L_first[i] ≠ c then
        OutputMatchesUp(SA^-[i], SA^-_φ[i], F(I_LF).D_pair[i + 1].first − b)
    if L_first[k] ≠ c then
        OutputMatchesDown(SA^+[k], SA^+_index[k], e − F(I_LF).D_pair[k].first + 1)
    i' = i + 1
    while i' < k do
        if L_first[k] ≠ c then
            OutputMatchesDown(SA^+[i'], SA^+_index[i'], F(I_LF).D_pair[i' + 1].first −
              F(I_LF).D_pair[i'].first)
            i' = i' + 1
        else i' = ND[i']
```

Algorithm 6 OutputMatchesDown.

```
/* Input: s: suffix of T, ι: interval of s in F(I_{φ−1],PLCP}), z:
   number of matches to output                                           */
/* Subroutine of Algorithm 5, implicitly has access to f, dict_occ, and
   OptBWTRL of T                                                         */
for a = 1 → z do
    g = dict_occ[s − f − 1]
    dict_occ.remove(s − f − 1)
    output match T[s, s + g − f − 1] = P[f + 1, g]              // Long LEM
    (s, ι) = Move(B(I_{φ^{-1},PLCP}), s, ι)
```

Algorithm 7 OutputMatchesUp.

```
/* Input: s: suffix of T, ι: interval of s in F(I_{φ,PLCP}), z:
   number of matches to output                                           */
/* Subroutine of Algorithm 5, implicitly has access to f, dict_occ, and
   OptBWTRL of T                                                         */
for a = 1 → z do
    g = dict_occ[s − f − 1]
    dict_occ.remove(s − f − 1)
    output match T[s, s + g − f − 1] = P[f + 1, g]              // Long LEM
    (s, ι) = Move(B(I_{φ,PLCP}), s, ι)
```