Optimal-Time Mapping in Run-Length Compressed PBWT

Paola Bonizzoni
Department of Computer Science, University of Milano-Bicocca, Italy

Davide Cozzi
Department of Computer Science, University of Milano-Bicocca, Italy

Younan Gao
Department of Computer Science, University of Milano-Bicocca, Italy

Abstract

The Positional Burrows–Wheeler Transform (PBWT) is a data structure designed for efficiently representing and querying large collections of sequences, such as haplotype panels in genomics. Forward and backward stepping operations – analogues to LF- and FL-mapping in the traditional BWT – are fundamental to the PBWT, underpinning many algorithms based on the PBWT for haplotype matching and related analyses. Although the run-length encoded variant of the PBWT (also known as the μ-PBWT) achieves $O(\tilde{r})$-word space usage, where $\tilde{r}$ is the total number of runs, no data structure supporting both forward and backward stepping in constant time within this space bound was previously known. In this paper, we consider the multi-allelic PBWT that is extended from its original binary form to a general ordered alphabet $\{0, \ldots, \sigma - 1\}$. We first establish bounds on the size $\tilde{r}$ and then introduce a new $O(\tilde{r})$-word data structure built over a list of haplotypes $\{S_1, \ldots, S_{\mathtt{h}}\}$, each of length $\mathtt{w}$, that supports constant-time forward and backward stepping.

We further revisit two key applications – haplotype retrieval and prefix search – leveraging our efficient forward stepping technique. Specifically, we design an $O(\tilde{r})$-word space data structure that supports haplotype retrieval in $O(\log \log_w h + \mathtt{w})$ time. For prefix search, we present an $O(\mathtt{h} + \tilde{r})$-word data structure that answers queries in $O(m' \log \log_w \sigma + \mathtt{occ})$ time, where m' denotes the length of the longest common prefix returned and $\mathtt{occ}$ denotes the number of haplotypes prefixed the longest prefix.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases PBWT, LF-Mapping, prefix searches, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.22

Funding All authors have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539, as well as from grant MIUR 2022YRB97K (PINC, *Pangenome Informatics: From Theory to Applications*), funded by the European Union under the NextGenerationEU programme, Mission 4.

Acknowledgements The authors would like to thank Travis Gagie for bringing prefix searches to their attention.

1 Introduction

Background and motivation. The *Positional Burrows–Wheeler Transform (PBWT)* [5] is a data structure designed for efficiently representing and querying large collections of sequences, such as haplotype panels in genomics. Originally proposed by Durbin [5], the PBWT stores a set of $\mathtt{h}$ haplotypes across $\mathtt{w}$ variant sites in a $\mathtt{h} \times \mathtt{w}$ binary matrix, where the rows at each column j are arranged in co-lexicographic order according to the prefixes of the haplotypes up to column $j - 1$. This ordering facilitates efficient querying between a given haplotype and the panel, enabling the identification of set-maximal exact matches (SMEMs).

© Paola Bonizzoni, Davide Cozzi, and Younan Gao;
licensed under Creative Commons License CC-BY 4.0
37th Annual Symposium on Combinatorial Pattern Matching (CPM 2026).
Editors: Philip Bille and Nicola Prezza; Article No. 22; pp. 22:1–22:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Two fundamental operations are defined on the PBWT: *forward stepping* and *backward stepping*. Forward stepping, denoted by $\texttt{fore}[i][j]$, maps the position of a haplotype in the permutation of the PBWT at site j to its position in the permutation at site $j+1$, while backward stepping, denoted by $\texttt{back}[i][j]$, performs the inverse mapping, tracing a haplotype's position at site $j+1$ back to its position at site j. These operations play an essential role in many PBWT-based algorithms for haplotype matching and analysis.

Although the PBWT enables efficient computations, its memory usage grows rapidly with large haplotype datasets, posing a challenge for population-scale cohorts like the UK Biobank [8]. To address this limitation, the *μ-PBWT* [4], also discussed in [2], was introduced as a compressed variant of the PBWT that leverages *run-length encoding (RLE)* to reduce space usage. A *run* in a sequence is defined as a maximal contiguous block of identical symbols, and the μ-PBWT′s storage requirement is $O(\tilde{r})$ words, where $\tilde{r}$ denotes the total number of runs in the corresponding PBWT across all $\texttt{w}$ sites. This run-length compression reduces the μ-PBWT's memory usage to only a fraction of that of the original PBWT, making it feasible to index and query massive haplotype panels such as those in the UK Biobank [8].

Beyond space efficiency, the μ-PBWT's structure has proven particularly useful for downstream applications such as computing matching statistics [4, 3], Minimal Positional Substring Covers (MPSC) [3], and SMEMs [4, 3]. Each of these applications relies on repeated forward and backward stepping operations. However, within the $O(\tilde{r})$-space bound of the μ-PBWT, no data structure supporting both operations in constant time was known prior to our work. Achieving faster stepping operations would directly improve the performance of all these applications and further enhance the efficiency of run-length compressed PBWT indexing for large-scale genomic analyzes.

Related work. Durbin [5] noted that, due to the co-lexicographic ordering of the PBWT, the forward (resp, backward) stepping operation is the natural analogue of the LF- (resp, FL-) mapping in the classical Burrows–Wheeler Transform (BWT). Gagie et al. [7, Lemma 2.1] presented a data structure requiring $O(r)$ words of space, built over a text of length n, that supports LF- and FL-mappings on the BWT in $O(\log\log_w(n/r))$ time, where r is the number of runs in the BWT of the text and w is the number of bits in a machine word. By applying this data structure to each column of the PBWT, we can achieve, for every site j, an $O(r_j)$-word data structure that supports both forward and backward stepping queries in $O(\log\log_w(\texttt{h}/r_j))$ time, where r_j denotes the number of runs at site j.

Nishimoto and Tabei [12] recently proposed the *move structure* to accelerate LF-mapping operations on the BWT. Their approach first divides the r runs in the BWT into at most $2r$ *sub-runs*, and then constructs an $O(r)$-word data structure over these sub-runs. Given a position i in the BWT and the index of the sub-run containing i, the structure can compute $\mathrm{LF}(i)$ and determine the index of the sub-run containing $\mathrm{LF}(i)$ in constant time.

Typically, in applications of the PBWT, forward and backward stepping are performed iteratively. For example, to traverse a haplotype in the PBWT from site j to site j', the forward stepping procedure is invoked sequentially $j' - j$ times. However, efficient (e.g., constant-time) forward and backward stepping in the run-length encoded PBWT cannot be achieved simply by applying the move structure independently at each site. Indeed, the move structure only returns the index of the sub-run at site j that contains $\texttt{fore}[i][j]$, whereas continuing the forward step from site $j + 1$ to site $j + 2$ requires the index of the sub-run at site $j + 1$ that contains $\texttt{fore}[i][j]$. See Figure 1 for an example. Hence, new methods are needed to accelerate $\texttt{fore}$ and $\texttt{back}$ queries while maintaining the run-length encoded PBWT.

	1	2	3	4	5
S_1	1	0	1	1	0
S_2	0	0	1	1	0
S_3	0	1	1	0	1
S_4	1	0	1	1	1
S_5	1	0	0	0	1

(a) The Input haplotypes.

	1	2	3	4	5
1	1	0	1	0	1
2	0	1	1	1	1
3	0	0	1	1	0
4	1	0	0	1	0
5	1	0	1	0	1

(b) The PBWT with runs.

	1	2	3	4	5
1	1	2	2	5	5
2	2	3	1	2	3
3	3	1	4	1	2
4	4	4	5	4	1
5	5	5	3	3	4

(c) PA.

Figure 1 Example of PBWT and PA built for bi-allelic haplotypes $\{S_1, S_2, S_3, S_4, S_5\}$. The operation fore[5][4] returns 2, and the position 2 is in the first run of the fifth column of the PBWT.

Two key applications of the back and fore queries on the PBWT are *haplotype retrieval* and *prefix search* [6]. In haplotype retrieval, the goal is to extract any haplotype from the PBWT. In prefix search, the objective is to find the longest common prefix $P[1..m']$ between any haplotype and a query pattern $P[1..m]$, and to list the indices of all haplotypes prefixed by $P[1..m']$. Using the data structure from [7, Lemma 2.1], built over each column of the PBWT, haplotype retrieval can be performed in $O(\mathtt{w} \log\log_w h)$ time by invoking forward stepping iteratively w times, with an overall space usage of $O(\tilde{r})$ words. Gagie et al. [6] present a textbook solution based on the PBWT that uses $O(\mathtt{h} \cdot \mathtt{w})$ words of space and supports each prefix query in $O(m \log \mathtt{h} + \mathtt{occ})$ time, where occ denotes the number of reported indices. They also provide various time-space tradeoffs in the same work. Nishimoto and Tabei [12] propose a data structure for prefix search that combines a BWT built over all haplotypes, a *compact trie* [10], the move data structure, and at most $2\mathtt{h} - 1$ marked positions. The structure supports computing the longest prefix $P[1..m']$ in $O(m')$ time, using $O(r^* + \mathtt{h})$ words of space, where r^* denotes the number of runs in the BWT of the concatenated sequence.

Our results. We consider the multi-allelic PBWT as described in [11]. We propose a "move data structure"−style solution specifically tailored for the PBWT. Our approach introduces a simple yet effective algorithm that partitions the $\tilde{r}$ runs of the PBWT into at most $2\tilde{r}$ sub-runs across all w columns of the PBWT (Section 4). We then design an $O(\tilde{r})$-word data structure over the sub-runs (Theorem 13) that supports $O(1)$-time computation of both forward and backward stepping operations, allowing to be invoked iteratively (Section 5).

As a first application, we present a PBWT-based solution for prefix searches. Specifically, assuming all haplotypes have the same length, we design an $O(\tilde{r} + \mathtt{h})$-word data structure that finds the longest common prefix $P[1..m']$ between any haplotype and a query pattern $P[1..m]$ in $O(m' \log\log_w \sigma + \mathtt{occ})$ time. When the haplotypes are sorted lexicographically, we can further reduce the space complexity from $O(\tilde{r} + \mathtt{h})$ to $O(\tilde{r})$. Note that the problem setting considered by Nishimoto and Tabei [12] allows haplotypes of varying lengths. We also show that our PBWT-based solution can be adapted to this more general case. While our query time is not as efficient as that of Nishimoto and Tabei [12], our approach has the advantage of not requiring a compact trie. As a second application, we design an $O(\tilde{r})$-word data structure to represent the haplotypes, supporting retrieval of any haplotype S_i ($1 \le i \le \mathtt{h}$) in $O(\log\log_w \mathtt{h} + \mathtt{w})$ time (Theorem 17).

Before introducing our "move"-like $O(\tilde{r})$-word data structure, we complement our results with lower and upper bounds on the measure $\tilde{r}$. We establish a connection between $\tilde{r}$ and the number $\mathtt{h}''$ of adjacent haplotype pairs (S_i, S_{i+1}) such that $S_i \neq S_{i+1}$. Specifically, we show that $\tilde{r} \geq \mathtt{h}'' + 1$ and $\tilde{r} \leq \mathtt{w}(\mathtt{h}'' + 1)$.

Paper organization. In Section 2, we introduce the notation and preliminary results used throughout the paper. In Section 3, we establish lower and upper bounds on the size $\tilde{r}$. Section 4 defines the *three-overlap* constraint over two lists of intervals and presents an algorithm that divides intervals in one list into sub-intervals satisfying this constraint. Applying this algorithm, we describe in Section 5 how to divide PBWT runs into sub-runs for `back` and `fore` queries. In Section 6, we design data structures over these sub-runs and develop the corresponding algorithms for `back` and `fore` queries. Section 7 discusses two applications of `back` and `fore` queries: haplotype retrieval and prefix searches. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Preliminaries

All results in this paper are presented under the word RAM (random-access machine) model. We evaluate the space cost of data structures in words. Each word is of w bits.

Notations. We denote by $[i, j]$ the interval of integers $i, i+1, \ldots, j$, and define $[i, j] = \emptyset$ if $j < i$. Given an interval $\mathsf{I} = [i, j]$, we denote its left and right endpoints by $\mathsf{I}.b$ and $\mathsf{I}.e$, respectively, so that $\mathsf{I}.b = i$ and $\mathsf{I}.e = j$. For any matrix A, we denote by $\mathtt{col}_j(A)$ its j-th column and by $\mathtt{col}_j(A)[i]$ the entry $A[i][j]$. Let M be the matrix that stores $S_1, S_2, \ldots, S_{\mathtt{h}}$ in rows $1, 2, \ldots, \mathtt{h}$. We note that the input haplotypes might not be pairwise distinct.

Consider an ordered alphabet $\{0, \ldots, \sigma-1\}$ with $0 < 1 < \cdots < \sigma-1$. A *string* α over this alphabet is a finite sequence of symbols from $\{0, \ldots, \sigma-1\}$, that is, $\alpha = \alpha[1]\alpha[2]\ldots\alpha[|\alpha|]$, where $|\alpha|$ denotes the length of α, and $\alpha[i] \in \{0, \ldots, \sigma-1\}$ for all $1 \leq i \leq |\alpha|$. The empty string is denoted by ε. For indices $1 \leq i \leq j \leq |\alpha|$, we denote by $\alpha[i..j]$ the *substring* of α spanning positions i through j (and define $\alpha[i..j] = \varepsilon$ if $i > j$). The string $\alpha[1..i]$ is referred to as the i-th *prefix* of α, and $\alpha[i..|\alpha|]$ as the i-th *suffix*, for $1 \leq i \leq |\alpha|$. A *proper prefix* (respectively, *proper suffix*) of α is a prefix (respectively, suffix) β such that $\beta \neq \alpha$.

Given two strings α and β over the alphabet set $\{0, \ldots, \sigma-1\}$, we say that α is *lexicographically smaller* than β, denoted by $\alpha \prec \beta$, if and only if one of the following holds: i) there exists an index k such that $\alpha[i] = \beta[i]$ for all $i < k$, and $\alpha[k] < \beta[k]$; ii) or α is a proper prefix of β. We say that α is *co-lexicographically smaller* than β, denoted by $\alpha \prec_{colex} \beta$, if and only if one of the following holds: i) There exists an index k such that $\alpha[|\alpha| - i + 1] = \beta[|\beta| - i + 1]$ for all $1 \leq i < k$, and $\alpha[|\alpha| - k + 1] < \beta[|\beta| - k + 1]$; ii) or α is a proper suffix of β.

Predecessor queries. Given a sorted list S of integers, a predecessor query takes an integer x as input and returns $\max\{y \in S \mid y \leq x\}$ and its rightmost position in S.

▶ **Lemma 1** ([1, Theorem A.1]). *Given an increasingly sorted list of n' integers, drawn from the universe $\{0, \ldots, \sigma-1\}$, there is a data structure that occupies $O(n' \log \sigma)$ bits of space and answers a predecessor query in $O(\log \log_w \sigma)$ time.*

Rank and select queries. Given a sequence $A[1..n']$ over an alphabet $\{0,\dots,\sigma-1\}$, the operation $\texttt{rank}_c(A,j)$ returns the number of occurrences of c in $A[1..j]$, for $c \in \{0,\dots,\sigma-1\}$. The operation $\texttt{select}_c(A,j)$ returns the position of the j-th occurrence of c in A, for $1 \le j \le \texttt{rank}_c(A,n')$, and returns $n'+1$ if $j > \texttt{rank}_c(A,n')$.

▶ **Lemma 2** ([1]). *There exists a data structure of $O(n')$ words built on $A[1..n']$ that supports* **rank** *queries in $O(\log\log_w \sigma)$ time and* **select** *queries in $O(1)$ time.*

Positional Burrows–Wheeler Transform (PBWT). Closely related to the PBWT is the *Prefix Array (PA)*, a matrix that records, for each column, the permutation of haplotype indices induced by the PBWT. Formally, the Prefix Array PA built for the matrix M is an $\texttt{h} \times \texttt{w}$ matrix, in which $\texttt{col}_1(\texttt{PA})$ is simply the list $1, 2, \dots, \texttt{h}$, and $\texttt{col}_j(\texttt{PA})$, for $j > 1$, stores the permutation of the set $\{1,\dots,\texttt{h}\}$ induced by the co-lexicographic ordering of prefixes of $\{S_1,\dots,S_\texttt{h}\}$ up to column $j-1$, that is, $\texttt{col}_j(\texttt{PA})[i] = k$ if and only if $S_k[1..j-1]$ is ranked i in the co-lexicographic order of $S_1[1..j-1],\dots,S_\texttt{h}[1..j-1]$.

Let PBWT be the matrix representing the positional BWT of M. Then PBWT is also an $\texttt{h} \times \texttt{w}$ matrix in which $\texttt{col}_j(\texttt{PBWT})[i] = \texttt{col}_j(M)[\texttt{col}_j(\texttt{PA})[i]]$ for all $i \in [1..\texttt{h}]$ and $j \in [1..\texttt{w}]$. We refer to a maximal substring of identical characters in $\texttt{col}_j(\texttt{PBWT})$ as a *run*. Throughout the paper, let r_j denote the number of runs for $\texttt{col}_j(\texttt{PBWT})$. We define $\tilde{r}$ as $\sum_{1\le j\le \texttt{w}} r_j$.

Forward and backward stepping on PBWT. We define $\texttt{fore}[i][j]$, for $i \in [1..\texttt{h}]$ and $j \in [1..\texttt{w})$, as the (row) index of $\texttt{col}_j(\texttt{PA})[i]$ in $\texttt{col}_{j+1}(\texttt{PA})$, and $\texttt{back}[i][j]$, for $i \in [1..\texttt{h}]$ and $j \in (1..\texttt{w}]$, as the (row) index of $\texttt{col}_j(\texttt{PA})[i]$ in $\texttt{col}_{j-1}(\texttt{PA})$.

Previously, the operation $\texttt{fore}[i][j]$ could be implemented as follows. Let C_c denote the number of occurrences of symbols $c' < c$ in column j of the PBWT, that is, in $\texttt{col}_j(\texttt{PBWT})$. Then, $\texttt{fore}[i][j] = C_c + \texttt{rank}_c(\texttt{col}_j(\texttt{PBWT}), i)$, where $c = \texttt{col}_j(\texttt{PBWT})[i]$. The backward stepping operation can be implemented in a symmetric manner.

Proposition 3 states the key properties of fore.

▶ **Proposition 3.** *(a) If $\texttt{col}_j(\texttt{PBWT})[i'] = \texttt{col}_j(\texttt{PBWT})[i'']$ and $i' < i''$, then $\texttt{fore}[i'][j] < \texttt{fore}[i''][j]$, and (b) if $\texttt{col}_j(\texttt{PBWT})[i'] = \texttt{col}_j(\texttt{PBWT})[i'+1]$, then $\texttt{fore}[i'][j]+1 = \texttt{fore}[i'+1][j]$.*

Proof. When constructing the j-th column of PBWT, note that the prefixes of the haplotypes up to column $j-1$ are stably sorted. Therefore, statements (a) and (b) follow directly from this stable ordering. ◀

3 A Lower Bound and an Upper Bound on $\tilde{r}$

In this section, we establish lower and upper bounds on $\tilde{r}$ in terms of $\texttt{h}''$, where $\texttt{h}''$ denotes the number of pairs (S_i, S_{i+1}) such that $1 \le i < \texttt{h}$ and $S_i \ne S_{i+1}$.

▶ **Lemma 4.** *It holds that $\tilde{r} \ge \texttt{h}'' + 1$.*

Proof. Consider any pair (S_i, S_{i+1}) such that $S_i \ne S_{i+1}$. Let j'' denote the length of the longest common prefix between S_i and S_{i+1}. Observe that the indices i and $i+1$ remain consecutive in $\texttt{col}_j(\texttt{PA})$ for all $1 \le j \le j''+1$. Let τ be the row index such that $\texttt{col}_{j''+1}(\texttt{PA})[\tau] = i$. By the observation above, we have $\texttt{col}_{j''+1}(\texttt{PA})[\tau+1] = i+1$. Since $\texttt{col}_{j''+1}(\texttt{PBWT})[\tau] \ne \texttt{col}_{j''+1}(\texttt{PBWT})[\tau+1]$ by the definition of j'', there must be a run boundary between rows τ and $\tau+1$ in the $(j''+1)$-st column of PBWT. Therefore, each pair (S_i, S_{i+1}) with $S_i \ne S_{i+1}$ corresponds to a run boundary in PBWT. The mapping from such a pair to a run boundary at $(\tau, j''+1)$ is injective, since each haplotype index i appears at a unique row τ in PA of any column. This establishes that $\tilde{r} \ge \texttt{h}'' + 1$. ◀

▶ **Corollary 5.** *$\tilde{r}$ is at least the number of distinct haplotypes in $\{S_1, S_2, \ldots, S_{\mathtt{h}}\}$.*

Proof. The statement holds because the number of distinct haplotypes in $\{S_1, S_2, \ldots, S_{\mathtt{h}}\}$ is at most $\mathtt{h}'' + 1$. ◀

We call an interval $[b, e]$ *canonical* with respect to column j of the PBWT, for $1 \leq j \leq \mathtt{w}$, if $[b, e]$ is the maximal interval such that $S_{\mathtt{col}_j(\mathtt{PA})[b]} = S_{\mathtt{col}_j(\mathtt{PA})[b+1]} = \cdots = S_{\mathtt{col}_j(\mathtt{PA})[e]}$. Let ℓ_j denote the number of canonical intervals with respect to column j for $1 \leq j \leq \mathtt{w}$. Lemma 6 describes the relationship between ℓ_j and r_j.

▶ **Lemma 6.** *It holds that $r_j \leq \ell_j$.*

Proof. Consider column j for any $1 \leq j \leq \mathtt{w}$. Let $[b, e]$ denote any canonical interval with respect to this column. Observe that $\mathtt{col}_j(\mathtt{PBWT})[b] = \mathtt{col}_j(\mathtt{PBWT})[b+1] = \cdots = \mathtt{col}_j(\mathtt{PBWT})[e]$. Hence, $[b, e]$ corresponds to a contiguous block of identical symbols in $\mathtt{col}_j(\mathtt{PBWT})$, although the block might not be maximal. This implies that $r_j \leq \ell_j$. ◀

▶ **Lemma 7.** *It holds that $\ell_j \leq h'' + 1$.*

Proof. The proof proceeds by induction on j. Let H be the array consisting of $\{\mathtt{h}\} \cup \{\, i \mid S_i \neq S_{i+1} \text{ and } 1 \leq i < \mathtt{h} \,\}$ in increasing order. It follows that $|H| = 1 + \mathtt{h}''$.

For the base case $j = 1$, observe that each interval $[H[t-1]+1, H[t]]$ for $1 \leq t \leq |H|$ (under the convention $H[0] = 0$) forms a distinct canonical interval with respect to the first column. Since $\bigcup_t [H[t-1]+1, H[t]] = [1, \mathtt{h}]$, we have $\ell_1 = |H|$.

Assume inductively that $\ell_{j-1} \leq \mathtt{h}'' + 1$. Let $[b_{j-1}, e_{j-1}]$ denote any canonical interval with respect to column $j-1$, so we have $\mathtt{col}_{j-1}(\mathtt{PBWT})[b_{j-1}] = \cdots = \mathtt{col}_{j-1}(\mathtt{PBWT})[e_{j-1}]$. By Proposition 3(b), the integers $\mathtt{fore}[b_{j-1}][j-1], \mathtt{fore}[b_{j-1}+1][j-1], \ldots, \mathtt{fore}[e_{j-1}][j-1]$ are consecutive, forming an interval $[\mathtt{fore}[b_{j-1}][j-1], \mathtt{fore}[e_{j-1}][j-1]]$. Furthermore, by the definitions of $\mathtt{fore}$ and $[b_{j-1}, e_{j-1}]$, the haplotypes $S_{\mathtt{col}_j(\mathtt{PA})[t]}$ for all $t \in [\mathtt{fore}[b_{j-1}][j-1], \mathtt{fore}[e_{j-1}][j-1]]$ are identical. Thus, this interval is contained within some canonical interval with respect to column j. This containment may be strict whenever multiple canonical intervals with respect to column $j-1$, corresponding to identical haplotype sequences, are mapped to adjacent positions in column j, thereby merging into a single canonical interval with respect to column j. Therefore, each canonical interval with respect to column $j-1$ is mapped by $\mathtt{fore}$ to a subinterval of a canonical interval with respect to column j. Since the union of these mapped intervals covers all rows $[1, \mathtt{h}]$, the number of canonical intervals in column j must be less than or equal to the number in column $j-1$ ($\ell_j \leq \ell_{j-1}$). By the induction hypothesis, $\ell_j \leq \ell_{j-1} \leq \mathtt{h}'' + 1$, completing the proof. ◀

▶ **Theorem 8.** *It holds that $\tilde{r} \leq w \cdot (h'' + 1)$.*

Proof. By Lemmas 6 and 7, we have $r_j \leq \mathtt{h}'' + 1$ for all j. Therefore, $\tilde{r} = \sum_{1 \leq j \leq \mathtt{w}} r_j \leq \sum_{1 \leq j \leq \mathtt{w}} (\mathtt{h}'' + 1) = \mathtt{w}(\mathtt{h}'' + 1)$. ◀

4 The Three-Overlap Constraint and a Normalization Algorithm

In this section, we define the three-overlap constraint and present a new algorithm for partitioning runs based on it.

▶ **Definition 9** (The Three-Overlap Constraint). *Let $I_p = \{[p_1, p'_1], [p_2, p'_2], \ldots, [p_x, p'_x]\}$ be a collection of x pairwise disjoint intervals that partition the range $[1, n]$, where $p_1 = 1$, $p'_x = n$, and $p_{i+1} = p'_i + 1$ for $1 \leq i < x$. Similarly, let $I_q = \{[q_1, q'_1], [q_2, q'_2], \ldots, [q_y, q'_y]\}$ be*

a collection of y pairwise disjoint intervals that also partition $[1, n]$, where $q_1 = 1$, $q'_y = n$, and $q_{j+1} = q'_j + 1$ for $1 \leq j < y$. We say that I_p satisfies the three-overlap constraint with respect to I_q *if every interval in I_p overlaps with at most three intervals in I_q.*

Note that the three-overlap constraint differs from the *balancing* property introduced in the move data structure [12] in two key aspects. Under the balancing property,

- I_p and I_q are in bijective correspondence – that is, there exists a bijection $f(\cdot)$ (with inverse $f^{-1}(\cdot)$) such that for each interval $\mathsf{I} \in I_p$ (resp., $\mathsf{I} \in I_q$), the interval $[f(\mathsf{I}.b), f(\mathsf{I}.e)]$ belongs to I_q (resp., $[f^{-1}(\mathsf{I}.b), f^{-1}(\mathsf{I}.e)] \in I_p$), and
- each interval in I_p contains at most three left endpoints of intervals in I_q; consequently, a single interval in I_p may overlap with up to four distinct intervals from I_q.

▶ **Lemma 10.** *Let I_p and I_q denote two lists of intervals into which $[1, n]$ is partitioned. There exists an $O(|I_p| + |I_q|)$-time algorithm that partitions all intervals in I_p into at most $(|I_p| + \left\lfloor \frac{|I_q|}{2} \right\rfloor)$ sub-intervals, satisfying the three-overlap constraint, with respect to I_q.*

Proof. Let $x = |I_p|$ and $I_p = [p_1, p'_1], [p_2, p'_2], \ldots, [p_x, p'_x]$ be x pairwise-disjoint intervals into which $[1, n]$ is partitioned, where $p_1 = 1, p'_x = n$, and $p'_i + 1 = p_{i+1}$ for $1 \leq i < x$. Let $y = |I_q|$. Without loss of generality, assume that $y \geq 4$; otherwise, I_p immediately satisfies the three-overlap constraint. Let $I_q = [q_1, q'_1], [q_2, q'_2], \ldots, [q_y, q'_y]$ be y pairwise-disjoint intervals into which $[1, n]$ is partitioned, where $q_1 = 1, q'_y = n$, and $q'_j + 1 = q_{j+1}$ for $1 \leq j < y$.

The algorithm `normalization`(I_p, I_q) is described as follows. Create a variable k and initiate k to 1. Iterate and process each interval $[p_i, p'_i]$ in I_p from left to right as follows: If $[p_i, p'_i]$ overlaps with at most three intervals of I_q, then skip $[p_i, p'_i]$ and move on to the next interval; otherwise, divide $[p_i, p'_i]$ into two sub-intervals, $[p_i, d_k]$ and $[d_k + 1, p'_i]$, where $1 \leq d_k \leq n$ is the largest integer such that $[p_i, d_k]$ overlaps three intervals of I_q. Then, substitute $[p_i, p'_i]$ for $[p_i, d_k]$ and $[d_k + 1, p'_i]$, increment k by one, and move on to the next interval, that is, $[d_k + 1, p'_i]$.

Let $\hat{I}_p$ denote the list of intervals outputted by the above algorithm. Figure 2 illustrates an example of the algorithm. Clearly, we have $|\hat{I}_p| = x + k$ and it follows that every interval in $\hat{I}_p$ overlaps at most three intervals in I_q. It remains to prove that $k \leq \left\lfloor \frac{y}{2} \right\rfloor$. To this end, we define $\Lambda(\hat{I}_p[i])$ for any $1 \leq i \leq x + k$ to be the set of intervals in I_q, overlapping $\hat{I}_p[i]$.

▷ Claim 11. For any $1 \leq t_1 < t_2 < k$, let I_1 (resp. I_2) denote the interval in $\hat{I}_p$, of which the right endpoint is d_{t_1} (resp. d_{t_2}). We have $\Lambda(\mathsf{I}_1) \cap \Lambda(\mathsf{I}_2) = \emptyset$.

Proof. Observe that we have $d_\tau \in \{q'_1, \ldots, q'_y\}$ for any $1 \leq \tau < k$, and thus the integers d_τ and $d_\tau + 1$ are always in different intervals in I_q. Moreover, $d_{t_1} + 1 \notin \tilde{\mathsf{I}}$ for any $\tilde{\mathsf{I}} \in \Lambda(\mathsf{I}_1)$, in view of the algorithm.

Assume that there exists an interval $\mathsf{I} \in \Lambda(\mathsf{I}_1) \cap \Lambda(\mathsf{I}_2)$. Let z denote the left endpoint of the interval I_2. Then, we have $d_{t_1} \in \mathsf{I}$ and $z \in \mathsf{I}$. Since $d_{t_1} < d_{t_1} + 1 \leq z$, we have $d_{t_1} + 1 \in \mathsf{I} \in \Lambda(\mathsf{I}_1)$ as well, a contradiction. Hence, the assumption is false. ◁

In view of the algorithm, the interval in $\hat{I}_p$ ending at d_τ for any $1 \leq \tau < k$ overlaps exactly three intervals in I_q. In view of Claim 11, we have $k - 1 \leq \left\lfloor \frac{y}{3} \right\rfloor$; therefore, $x + k \leq x + 1 + \left\lfloor \frac{y}{3} \right\rfloor \leq x + \left\lfloor \frac{y}{2} \right\rfloor$, for $y \geq 4$.

Clearly, the algorithm runs in $O(|\hat{I}_p| + |I_p| + |I_q|) \subseteq O(|I_p| + |I_q|)$ time. ◀

Figure 2 Example of the normalization algorithm. Given $I_p = \{[1,1],[2,11],[12,16]\}$ and $I_q = \{[1,2],[3,3],[4,5],[6,7],[8,9],[10,10],[11,13],[14,14],[15,16]\}$, the output is $\hat{I}_p = \{[1,1],[2,5],[6,10],[11,11],[12,16]\}$, where each interval of $\hat{I}_p$ overlaps at most 3 intervals of I_q.

5 Constructing Sub-Runs and Bounding Their Number

In this section, we introduce *sub-runs* for forward and backward stepping (i.e., `fore` and `back`) in the PBWT.

While runs are maximal-length substrings consisting of the same character, we define sub-runs as substrings of the same character without the maximal-length restriction.

We define a *run interval* with respect to column j ($1 \leq j \leq \mathtt{w}$) as the maximal interval $[b, e]$ such that all symbols in positions $b, b+1, \ldots, e$ of $\mathtt{col}_j(\mathtt{PBWT})$ are identical. Let $\mathtt{intervals}_j$ denote the list of the r_j run intervals in column j, sorted in increasing order of their starting positions. Similarly, a *sub-run interval* is any contiguous subrange contained within a run interval. We abuse notation slightly and use the terms run intervals (resp. sub-run intervals) and runs (resp. sub-runs) interchangeably.

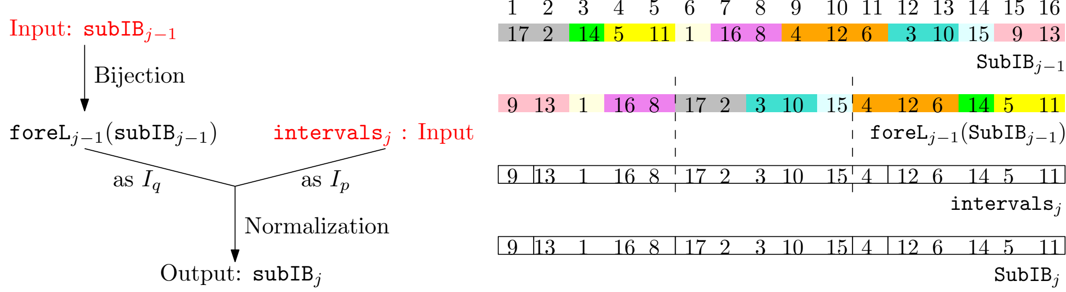
In the following, we define two bijection functions `foreL` and `backL` that will be used, respectively, in the construction of two lists $\mathtt{SubIB}_j$ and $\mathtt{SubIF}_j$ of sub-runs for each column j. The lists $\mathtt{SubIB}_j$ and $\mathtt{SubIF}_j$ are used later to implement `back` queries and `fore` queries, respectively. More precisely, $\mathtt{foreL}_j(L)$ returns the list $\{[\mathtt{fore}[b_\tau][j], \mathtt{fore}[e_\tau][j]] \mid 1 \leq \tau \leq |L|\}$, sorted in increasing order by the left endpoint of each interval, for any set of intervals $L = \{[b_\tau, e_\tau] \mid 1 \leq \tau \leq |L|, 1 \leq b_\tau \leq e_\tau \leq \mathtt{h}\}$ and any $1 \leq j < \mathtt{w}$.

By Proposition 3(b), it follows that $\mathtt{fore}[i+1][j] - \mathtt{fore}[i][j] = 1$ if $\mathtt{col}_j(\mathtt{PBWT})[i] = \mathtt{col}_j(\mathtt{PBWT})[i+1]$. Hence, $\mathtt{foreL}_j(L)$ is well-defined (i.e., $\mathtt{fore}[e_\tau][j] \geq \mathtt{fore}[b_\tau][j]$ for every $1 \leq \tau \leq |L|$) if each interval in L is a sub-interval of some interval in $\mathtt{intervals}_j$.

Symmetrically, we define $\mathtt{backL}_j(L)$ to return the list $\{[\mathtt{back}[b_\tau][j], \mathtt{back}[e_\tau][j]] \mid 1 \leq \tau \leq |L|\}$, sorted in increasing order by the left endpoint of each interval.

Constructing sub-runs for `back` queries. The construction of $\mathtt{SubIB}_j$ proceeds by induction. In the base case, we set $\mathtt{SubIB}_1 = \mathtt{intervals}_1$. For each j from 2 to $\mathtt{w}$, we construct $\mathtt{SubIB}_j$ as follows: we apply the algorithm $\mathtt{normalization}(I_p, I_q)$ (see Lemma 10) with $I_p := \mathtt{intervals}_j$ and $I_q := \mathtt{foreL}_{j-1}(\mathtt{SubIB}_{j-1})$. Recall that the function $\mathtt{foreL}_{j-1}$ maps sub-intervals in column $j-1$ to sub-intervals in column j. The list of sub-intervals output by this algorithm is assigned to $\mathtt{SubIB}_j$. See Figure 3 for an illustration of the algorithm and an example.

In view of the algorithm in Lemma 10, each interval in $\mathtt{SubIB}_j$ for $1 \leq j \leq \mathtt{w}$ is a sub-interval of some interval in $\mathtt{intervals}_j$ and overlaps at most three intervals in the list $\mathtt{foreL}_{j-1}(\mathtt{SubIB}_{j-1})$. Recall that every interval in $\mathtt{intervals}_j$ corresponds to a run in $\mathtt{col}_j(\mathtt{PBWT})$. Hence, every interval in $\mathtt{SubIB}_j$ corresponds to a sub-run in $\mathtt{col}_j(\mathtt{PBWT})$.



(a) The scheme of building SubIB$_j$. **(b)** An Example of building SubIB$_j$.

Figure 3 Illustration of the algorithm scheme for building sub-runs in SubIB$_j$ and an example. In this example, SubIB$_{j-1} = \{ [1,2], [3,3], [4,5], [6,6], [7,8], [9,11], [12,13], [14,14], [15,16] \}$ and intervals$_j = \{ [1,1], [2,11], [12,16] \}$. The bijective function foreL$_{j-1}$ maps the list SubIB$_{j-1}$ to the list $\{ [1,2], [3,3], [4,5], [6,7], [8,9], [10,10], [11,13], [14,14], [15,16] \}$. Intervals highlighted in the same color contain the same haplotype indices and indicate corresponding pairs under this bijection. After applying the normalization algorithm to intervals$_j$ and foreL$_{j-1}$(SubIB$_{j-1}$), the intervals in intervals$_j$ are partitioned into SubIB$_j = \{ [1,1], [2,5], [6,10], [11,11], [12,16] \}$. Each interval in SubIB$_j$ overlaps with at most three intervals in foreL$_{j-1}$(SubIB$_{j-1}$).

Constructing sub-runs for fore queries. The construction of SubIF$_j$ is also performed by induction. In the base case, we set SubIF$_{\mathtt{w}}$ = intervals$_{\mathtt{w}}$. Then, for j from $\mathtt{w} - 1$ down to 1, we construct SubIF$_j$ as follows: we apply the algorithm normalization(I_p, I_q) (see Lemma 10), where I_p = foreL$_j$(intervals$_j$) and I_q = SubIF$_{j+1}$. Let SubIF$'_{j+1}$ be the list of intervals output by the algorithm; we then assign SubIF$_j$ = backL$_{j+1}$(SubIF$'_{j+1}$). Figure 6 (in Appendix A) illustrates the construction scheme along with an example.

By the normalization algorithm shown in Lemma 10, each interval in SubIF$'_{j+1}$ for $1 \le j \le \mathtt{w}$ is a sub-interval of some interval in foreL$_j$(intervals$_j$) and overlaps at most three intervals in SubIF$_{j+1}$. Recall that every interval in foreL$_j$(intervals$_j$) corresponds to a run in col$_j$(PBWT). Likewise, every interval in SubIF$'_{j+1}$ corresponds to a sub-run in col$_j$(PBWT). Since SubIF$_j$ = backL$_{j+1}$(SubIF$'_{j+1}$), every interval in SubIF$_j$ is a sub-run in col$_j$(PBWT).

▶ **Lemma 12.** *We have $\sum_{1 \le j \le w} |SubIB_j| < 2\tilde{r}$ and $\sum_{1 \le j \le w} |SubIF_j| < 2\tilde{r}$.*

Proof. Observe that |foreL$_j$(SubIB$_j$)| = |SubIB$_j$|. Recall that |intervals$_j$| $= r_j$ is always true. In the base case, we have |SubIB$_1$| $= r_1$. For $1 < j \le \mathtt{w}$, we have the recursion $|\mathtt{SubIB}_j| \le |\mathtt{intervals}_j| + \lfloor \frac{|\mathtt{SubIB}_{j-1}|}{2} \rfloor \le r_j + \frac{|\mathtt{SubIB}_{j-1}|}{2}$, in view of Lemma 10 and the observation above. By solving the recursion, it follows that $|\mathtt{SubIB}_j| = \frac{r_1}{2^{j-1}} + \frac{r_2}{2^{j-2}} + \cdots + \frac{r_j}{2^{j-j}} = \sum_{1 \le \tau \le j} \frac{r_\tau}{2^{j-\tau}}$. Therefore, we have $\sum_{1 \le j \le \mathtt{w}} |\mathtt{SubIB}_j| \le \sum_{1 \le j \le \mathtt{w}} \sum_{1 \le \tau \le j} \frac{r_\tau}{2^{j-\tau}} < 2(r_1 + r_2 + \cdots + r_{\mathtt{w}}) = 2\tilde{r}$. The bound on the total number of |SubIF$_j$| for all j can be computed similarly. This concludes the proof. ◀

6 The Data Structure for Constant-Time fore/back Queries

In this section, we design the data structure constructed over these sub-runs to support fore/back queries and access to entries in the matrix PBWT.

▶ **Theorem 13.** *There exists a data structure of $O(\tilde{r})$ words, constructed over SubIB$_j$ and SubIF$_j$ for $1 \le j \le w$, that supports each of the following operations in constant time – without accessing the original matrix M, its PBWT, or its prefix arrays PA:*

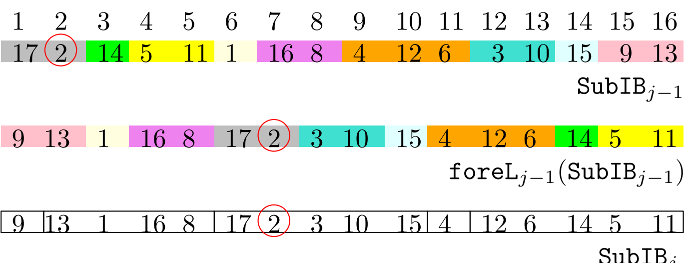


Figure 4 Example illustrating the data structure and algorithm for back queries. The third interval $[6, 10]$ in $\texttt{SubIB}_j$ overlaps three intervals in $\texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$: $[6, 7]$, $[8, 9]$, and $[10, 10]$. These correspond to the first $[1, 2]$, seventh $[12, 13]$, and eighth $[14, 14]$ intervals in $\texttt{SubIB}_{j-1}$, respectively. Thus, the data structureB^3_j stores the quadruples $\{(6, 7, 1, 1), (8, 9, 12, 7), (10, 10, 14, 8)\}$ in the form $(\tilde{s}, \tilde{t}, s, \lambda)$. For a query $\texttt{back}[7][j]$ with index 3, the algorithm finds $(\tilde{s} : 6, \tilde{t} : 7, s : 1, \lambda : 1)$ in B^3_j (since $7 \in [6, 7]$) and returns $7 - \tilde{s} + s = 2$ and $\lambda = 1$.

- *Given an index $i' \in [1..h]$ and the index of the interval in $\texttt{SubIB}_{j'}$ that contains i', one can find $\texttt{back}[i'][j']$, determine the index of the interval in $\texttt{SubIB}_{j'-1}$ containing $\texttt{back}[i'][j']$, and retrieve $\texttt{col}_{j'}(PBWT)[i']$.*
- *Given an index $i' \in [1..h]$ and the index of the interval in $\texttt{SubIF}_{j'}$ that contains i', one can find $\texttt{fore}[i'][j']$, determine the index of the interval in $\texttt{SubIF}_{j'+1}$ containing $\texttt{fore}[i'][j']$, and retrieve $\texttt{col}_{j'}(PBWT)[i']$.*

Proof. We first describe the data structure for back queries. For each column j of the PBWT, we construct the list $\texttt{SubIB}_j$ as described in Section 5 (see Constructing sub-runs for back queries). Observe that each interval in $\texttt{SubIB}_j$ overlaps at most three intervals in $\texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$. Moreover, each interval $\tilde{\mathsf{I}} \in \texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$ corresponds to the interval $[\texttt{back}[\tilde{\mathsf{I}}.b][j], \texttt{back}[\tilde{\mathsf{I}}.e][j]]$ belonging to $\texttt{SubIB}_{j-1}$. For every $1 < j \le \texttt{w}$ and each interval $\texttt{SubIB}_j[\tau]$ with $1 \le \tau \le |\texttt{SubIB}_j|$, we store a list B^τ_j of quadruples $(\tilde{s}, \tilde{t}, s, \lambda)$, where:

- $[\tilde{s}, \tilde{t}]$ is a distinct interval in $\texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$ that overlaps $\texttt{SubIB}_j[\tau]$;
- $s = \texttt{back}[\tilde{s}][j]$; and
- λ is the index of the interval in $\texttt{SubIB}_{j-1}$ whose left endpoint is s.

Since each $\texttt{SubIB}_j[\tau]$ overlaps at most three intervals in $\texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$, the list B^τ_j contains at most three tuples. By Lemma 12, $\sum_j |\texttt{SubIB}_j| < 2\tilde{r}$, so the total space usage is $O(\tilde{r})$ words.

Let x, given in a query, denote the index of the sub-run in $\texttt{SubIB}_{j'}$ containing i'. To answer a query $\texttt{back}[i'][j']$, we search in the list $B^x_{j'}$ for the quadruple $(\tilde{s}, \tilde{t}, s, \lambda)$ satisfying $i' \in [\tilde{s}, \tilde{t}]$. Then $\texttt{back}[i'][j'] = i' - \tilde{s} + s$, and λ is the index of the interval in $\texttt{SubIB}_{j'-1}$ containing $\texttt{back}[i'][j']$. We return $(i' - \tilde{s} + s, \lambda)$ as the result. Since $|B^x_{j'}| \le 3$, each query takes $O(1)$ time. See Figure 4 for an illustration of the data structure and algorithm.

Accessing $\texttt{col}_{j'}(\texttt{PBWT})[i']$ from SubIB lists. We store, for each $1 \le j \le \texttt{w}$, an array $\texttt{ValBack}_j$ of length $|\texttt{SubIB}_j|$, where each entry $\texttt{ValBack}_j[\tau]$ is set to $\texttt{col}_j(\texttt{PBWT})[\texttt{SubIB}_j[\tau].b]$. This requires $O(r_j)$ words per column, and $O(\tilde{r})$ words overall. Given the index x of the sub-run containing i' in $\texttt{SubIB}_{j'}$, we have $\texttt{col}_{j'}(\texttt{PBWT})[i'] = \texttt{ValBack}_{j'}[x]$, which can be retrieved in $O(1)$ time.

Data structure for fore queries. We construct the lists $\texttt{SubIF}_j$ for each column j as described in Section 5 (see Constructing sub-runs for fore queries). Each interval $\mathsf{I} \in \texttt{SubIF}_j$ corresponds to $[\texttt{fore}[\mathsf{I}.b][j], \texttt{fore}[\mathsf{I}.e][j]] \in \texttt{foreL}_j(\texttt{SubIF}_j)$, and each interval in $\texttt{foreL}_j(\texttt{SubIF}_j)$ overlaps at most three intervals in $\texttt{SubIF}_{j+1}$. For every $1 \le j < \texttt{w}$ and each interval $\texttt{SubIF}_j[\tau]$, we store a list F^τ_j of quintuplets $(s', \tilde{s}, s, t, \lambda)$, where:

- $s' = \texttt{SubIF}_j[\tau].b$, $\tilde{s} = \texttt{fore}[s'][j]$,
- $[s,t]$ is an interval in $\texttt{SubIF}_{j+1}$ that overlaps $[\texttt{fore}[\texttt{SubIF}_j[\tau].b][j], \texttt{fore}[\texttt{SubIF}_j[\tau].e][j]]$,
- λ is the index of $[s,t]$ in $\texttt{SubIF}_{j+1}$.

Since each such interval, i.e., $[\texttt{fore}[\texttt{SubIF}_j[\tau].b][j], \texttt{fore}[\texttt{SubIF}_j[\tau].e][j]]$, overlaps at most three intervals in $\texttt{SubIF}_{j+1}$, each F^{τ}_j has at most three tuples. By Lemma 12, the total space is again $O(\tilde{r})$ words.

Let x denote the index of the sub-run containing i' in $\texttt{SubIF}_{j'}$. To answer $\texttt{fore}[i'][j']$, we search $F^x_{j'}$ for the quintuple $(s', \tilde{s}, s, t, \lambda)$ satisfying $i' - s' + \tilde{s} \in [s,t]$. Then $\texttt{fore}[i'][j'] = i' - s' + \tilde{s}$, and λ is the index of the interval in $\texttt{SubIF}_{j'+1}$ containing $\texttt{fore}[i'][j']$. We return $(i' - s' + \tilde{s}, \lambda)$ as the result. Since $|F^x_{j'}| \leq 3$, each query takes $O(1)$ time. An illustration of the data structure and the algorithm is depicted in Figure 7 (in Appendix B).

Accessing $\texttt{col}_j(\texttt{PBWT})[i]$ from $\texttt{SubIF}$ lists. Analogously, for each $1 \leq j \leq \texttt{w}$, we store an array $\texttt{ValFore}_j$ of length $|\texttt{SubIF}_j|$, where $\texttt{ValFore}_j[\tau] = \texttt{col}_j(\texttt{PBWT})[\texttt{SubIF}_j[\tau].b]$. This requires $O(r_j)$ words per column, and $O(\tilde{r})$ words in total. Given the index x of the sub-run containing i in $\texttt{SubIF}_j$, we have $\texttt{col}_j(\texttt{PBWT})[i] = \texttt{ValFore}_j[x]$, retrievable in $O(1)$ time. ◀

7 Applications

In this section, we present applications that rely on the iterative use of $\texttt{fore}/\texttt{back}$ queries. Section 7.1 introduces a PBWT-based solution to prefix searches, while our method to haplotype retrieval is given in Section 7.2.

Henceforth, set $\rho_j := |\texttt{SubIF}_j|$ for every $1 \leq j \leq \texttt{w}$. According to Theorem 13, given i, j and x, where $1 \leq j \leq \texttt{w}$, $1 \leq x \leq \rho_j$, and $\texttt{SubIF}_j[x].b \leq i \leq \texttt{SubIF}_j[x].e$, we define $\texttt{forePair}_j(i,x)$ that returns (i', x') such that $i' = \texttt{fore}[i][j]$ and $\texttt{SubIF}_{j+1}[x'].b \leq i' \leq \texttt{SubIF}_{j+1}[x'].e$. In other words, $\texttt{forePair}_j(i,x)$ returns $\texttt{fore}[i][j]$ and the index of the sub-interval in $\texttt{SubIF}_{j+1}$ that contains $\texttt{fore}[i][j]$.

7.1 A PBWT-Based Solution to Prefix Search

We first provide a solution under the assumption that all haplotypes $\{S_1, \ldots, S_\texttt{h}\}$ are of the same length $\texttt{w}$ and then generalize it to the case where haplotypes are of arbitrary length.

Similarly as *sa-interval* in suffix arrays [9], given a query pattern $P[1..m]$, we define $\texttt{pa-interval}_j$, with $2 \leq j \leq m$, as the maximal continuous range of indices in $\texttt{col}_j(\texttt{PA})$ such that $P[1..j-1]$ is a prefix of S_i for each $i \in \texttt{pa-interval}_j$. In particular, $\texttt{pa-interval}_1$ is defined as $[1..\texttt{h}]$, corresponding to the empty string $P[1..0]$.

▶ **Theorem 14.** *There exists an $O(\tilde{r})$-word data structure constructed over an arbitrarily ordered list $\{S_1, \ldots, S_\texttt{h}\}$ of haplotypes of length $\texttt{w}$ over the alphabet $\{0, \ldots, \sigma-1\}$ such that, given a pattern $P[1..m]$, one can compute in $O(m' \log\log_w \sigma)$ time:*

1. *the longest common prefix $P[1..m']$ between $P[1..m]$ and any haplotype,*
2. *the number $\texttt{occ}$ of haplotypes prefixed by $P[1..m']$, and*
3. *an index i such that S_i is prefixed by $P[1..m']$.*

If $\{S_1, \ldots, S_\texttt{h}\}$ are lexicographically sorted, then the data structure finds, in $O(m' \log\log_w \sigma)$ time, the interval $[\gamma..\gamma']$ such that S_i is prefixed by $P[1..m']$ for every $i \in [\gamma..\gamma']$ and $\gamma' = \gamma + \texttt{occ} - 1$.

In the remainder of this section, we prove Theorem 14.

The data structures. The construction of the data structure proceeds as follows. We first build the PBWT matrix PBWT and the prefix array matrix PA for the h haplotypes, each consisting of w columns and h rows. Next, we compute the sub-runs $\texttt{SubIF}_j$ for all $1 \leq j \leq \texttt{w}$, as described in Section 5. Over these sub-runs, we then construct the $O(\tilde{r})$-space data structure from Theorem 13 to support fore queries efficiently.

We also build the following auxiliary arrays for $1 \leq j \leq \texttt{w}$. Recall that $\rho_j = |\texttt{SubIF}_j|$.

- $\texttt{sPA}_j[1..\rho_j]$: each entry $\texttt{sPA}_j[\tau]$ ($\tau \in [1..\rho_j]$) stores $\texttt{col}_j(\texttt{PA})[\texttt{SubIF}_j[\tau].b]$;
- $\texttt{sVal}_j[1..\rho_j]$: each entry $\texttt{sVal}_j[\tau]$ stores $\texttt{col}_j(\texttt{PBWT})[\texttt{SubIF}_j[\tau].b]$;
- $\texttt{sCount}_j[1..\rho_j]$: each entry $\texttt{sCount}_j[\tau]$ stores the number of occurrences of $\texttt{sVal}_j[\tau]$ in $\texttt{col}_j(\texttt{PBWT})[1..\texttt{SubIF}_j[\tau].b]$, that is, $\texttt{rank}_{\texttt{sVal}_j[\tau]}(\texttt{col}_j(\texttt{PBWT}), \texttt{SubIF}_j[\tau].b)$.

We then build the data structures for rank and select queries over $\texttt{sVal}_j$ as in Lemma 2. After building these arrays and supporting structures, we discard PBWT and PA.

The structure of Theorem 13 occupies $O(\tilde{r})$ words of space. By Lemma 12, we have $\sum_j \rho_j = O(\sum_j r_j) = O(\tilde{r})$. Hence, storing all the arrays and auxiliary rank/select structures requires $O(\sum_j \rho_j) = O(\tilde{r})$ words in total. Consequently, the overall space complexity of our data structure is $O(\tilde{r})$ words.

The query algorithm. The algorithm $\texttt{Partial Prefix Search}(P[1..m])$ iterates through columns from left to right. At each column j, it updates a state quintuple $(b, e, x, x', index)$ defined as follows:

- $b, e \in \{1, \ldots, \texttt{h}\}$ represent the interval boundaries $\texttt{pa-interval}_j.b$ and $\texttt{pa-interval}_j.e$, respectively,
- $x, x' \in \{1, \ldots, |\texttt{SubIF}_j|\}$ are indices such that $b \in \texttt{SubIF}_j[x]$ and $e \in \texttt{SubIF}_j[x']$, respectively, and
- $index \in \{1, \ldots, \texttt{h}\}$ stores the value $\texttt{col}_j(\texttt{PA})[b]$.

Initially, we set the quintuple as $[b, e] = [1, h] = \texttt{pa-interval}_1$, $x = 1$, $x' = |\texttt{SubIF}_1|$, and $index = \texttt{sPA}_1[1]$. In the j-th iteration ($1 \leq j \leq m$), the following steps are performed:

- We first determine the positions $\tilde{b}$ and $\tilde{e}$, where $b \leq \tilde{b} \leq \tilde{e} \leq e$ denote the indices of the first and last occurrences of $P[j]$ in $\texttt{col}_j(\texttt{PBWT})$ within the range $[b, e]$. Formally, $\tilde{b} = \min\{\ell \mid b \leq \ell \leq e,\ \texttt{col}_j(\texttt{PBWT})[\ell] = P[j]\}$ and $\tilde{e} = \max\{\ell \mid b \leq \ell \leq e,\ \texttt{col}_j(\texttt{PBWT})[\ell] = P[j]\}$. If neither $\tilde{b}$ nor $\tilde{e}$ exists, the interval $\texttt{pa-interval}_{j+1}$ corresponding to the prefix $P[1..j]$ is empty. Otherwise, we locate the indices $\tilde{x}$ and $\tilde{x}'$ such that $\tilde{b} \in \texttt{SubIF}_j[\tilde{x}]$ and $\tilde{e} \in \texttt{SubIF}_j[\tilde{x}']$, respectively. In the pseudocode, we show that all of $\tilde{b}$, $\tilde{e}$, $\tilde{x}$, and $\tilde{x}'$ can be obtained using rank and select queries over the array $\texttt{sVal}_j$. In addition, if $\tilde{b} \neq b$, we update the variable $index := \texttt{sPA}_j[\tilde{x}]$. We will later prove that the variable $index$ always stores $\texttt{col}_j(\texttt{PA})[b]$.
- Second, if $j < m$ and $\texttt{pa-interval}_{j+1} \neq \emptyset$, we apply the queries $\texttt{forePair}_j(\tilde{b}, \tilde{x})$ and $\texttt{forePair}_j(\tilde{e}, \tilde{x}')$ according to Theorem 13, and update $(b, x) := \texttt{forePair}_j(\tilde{b}, \tilde{x})$ and $(e, x') := \texttt{forePair}_j(\tilde{e}, \tilde{x}')$, respectively. Consequently, for $j < m$, it follows that $b = \texttt{pa-interval}_{j+1}.b$ and $e = \texttt{pa-interval}_{j+1}.e$. Thus, $\texttt{pa-interval}_{j+1}$, corresponding to the prefix $P[1..j]$, is obtained and stored as $[b, e]$, with the values x and x' updated accordingly. If $j = m$, we instead set $(b, x) := (\tilde{b}, \tilde{x})$ and $(e, x') := (\tilde{e}, \tilde{x}')$.
- Third, we increment j by 1 and proceed to the next iteration.

We stop proceeding to the next iteration if $\texttt{pa-interval}_{j+1} = \emptyset$ or $j = m + 1$. The following observation is crucial: in either case, the longest common prefix shared by $P[1..m]$ and any haplotype is $P[1..j-1]$.

If $\texttt{pa-interval}_{j+1} = \emptyset$, then $[b, e]$ is $\texttt{pa-interval}_j$, corresponding to the prefix $P[1..j-1]$. If $j = 1$, then the longest common prefix shared by $P[1..m]$ and any haplotype is an empty string. Otherwise, the longest common prefix is $P[1..j-1]$ by the observation mentioned earlier, and the number of haplotypes prefixed by $P[1..j-1]$ is $e - b + 1$ by the definition of $\texttt{pa-interval}_j$; therefore, we return $P[1..j-1], e-b+1$, and *index* as the answer.

If instead $j = m + 1$, it follows that $P[1..m]$ is the longest common prefix and that b and e store, respectively, the indices of the first and last occurrences of $P[m]$ in $\texttt{col}_m(\texttt{PBWT})[\hat{b}, \hat{e}]$, where $[\hat{b}, \hat{e}] = \texttt{pa-interval}_m$ (note that $\texttt{pa-interval}_m$ corresponds to the prefix $P[1..m-1]$). In this case, we compute the numbers of occurrences of $P[m]$ in $\texttt{col}_m(\texttt{PBWT})[1..b]$ and $\texttt{col}_m(\texttt{PBWT})[1..e]$, which are $\texttt{sCount}_m[x] + b - \texttt{SubIF}_m[x].b$ and $\texttt{sCount}_j[x'] + e - \texttt{SubIF}_j[x'].b$, respectively, and store them in variables $count_1$ and $count_2$. The number of haplotypes prefixed by $P[1..m]$ is $count_2 - count_1 + 1$. In the end, we return $P[1..m], count_2 - count_1 + 1, index$ as the answer; recall that the variable *index* stores $\texttt{col}_m(\texttt{PA})[b]$. The pseudocode of the algorithm can be found in Appendix C.

We have shown that the procedure $\texttt{Partial Prefix Search}(P[1..m])$ identifies the longest prefix $P[1..m']$ shared between $P[1..m]$ and any haplotype, and counts the number of haplotypes prefixed by $P[1..m']$. It remains to show that the variable *index* returned by the procedure indeed stores the index of one of these haplotypes. To this end, it suffices to prove that the invariant $index = \texttt{col}_j(\texttt{PA})[b]$ holds in each iteration j.

▶ **Lemma 15.** *It follows that* $index = \texttt{col}_j(\texttt{PA})[b]$ *in each iteration* j *with* $1 \leq j \leq m$.

Proof. We give a proof by induction. When $j = 1$, the procedure sets $b := 1$ and $index := \texttt{sPA}_1[1]$. Since $\texttt{col}_j(\texttt{PA})[b] = \texttt{col}_j(\texttt{PA})[1] = \texttt{sPA}_1[1]$, we have $index = \texttt{col}_j(\texttt{PA})[b]$; the base case for $j = 1$ holds trivially.

Assume by induction that at the beginning of the j-th iteration, we have $index = \texttt{col}_j(\texttt{PA})[b]$. Consider the j-th iteration. Note that in the beginning of this iteration, we have $\texttt{sVal}_j[x] = \texttt{col}_j(\texttt{PBWT})[b]$, as $b \in \texttt{SubIF}_j[x]$.

If $\texttt{sVal}_j[x] = P[j]$, then $\texttt{col}_j(\texttt{PBWT})[b] = P[j]$; in this case, the variable *index* remains in this iteration, and the variable b is set to $\texttt{fore}[b][j]$. Note that by the definition of $\texttt{fore}$ queries, we have $\texttt{col}_j(\texttt{PA})[b] = \texttt{col}_{j+1}(\texttt{PA})[\texttt{fore}[b][j]]$. By the inductive assumption, we have $index = \texttt{col}_j(\texttt{PA})[b]$, thereby $index = \texttt{col}_{j+1}(\texttt{PA})[\texttt{fore}[b][j]]$; therefore, at the beginning of the next iteration, the statement still holds.

Otherwise, we have $\texttt{col}_j(\texttt{PBWT})[b] \neq P[j]$; in this case, the procedure finds the index $\tilde{b}$ of the first occurrence of $P[j]$ in $\texttt{col}_j(\texttt{PBWT})$ within the range $[b..e]$ and the index $\tilde{x}$ that satisfies $\tilde{b} = \texttt{SubIF}_j[\tilde{x}].b$, and sets $index := \texttt{sPA}_j[\tilde{x}]$. Since $\texttt{col}_j(\texttt{PA})[\tilde{b}] = \texttt{sPA}_j[\tilde{x}]$, we have $\texttt{col}_j(\texttt{PA})[\tilde{b}] = index$. If $j = m$, then the procedure sets $b := \tilde{b}$; therefore, we have $\texttt{col}_m(\texttt{PA})[b] = \texttt{col}_m(\texttt{PA})[\tilde{b}] = \texttt{sPA}_m[\tilde{x}] = index$. When the while-loop terminates in the m-th iteration, we have $\texttt{col}_m(\texttt{PA})[b] = index$. If $j < m$, then b is set to $\texttt{fore}[\tilde{b}][j]$. By the definition of $\texttt{fore}$ queries, we have $\texttt{col}_j(\texttt{PA})[\tilde{b}] = \texttt{col}_{j+1}(\texttt{PA})[\texttt{fore}[\tilde{b}][j]]$. Recall that $\texttt{col}_j(\texttt{PA})[\tilde{b}] = index$, so it follows that $index = \texttt{col}_{j+1}(\texttt{PA})[\texttt{fore}[\tilde{b}][j]] = \texttt{col}_{j+1}(\texttt{PA})[b]$. Thus, at the beginning of the next iteration, the statement holds, completing the proof. ◀

Moreover, by Proposition 3(a), it follows that the variable *index* returned by the algorithm stores the smallest index ℓ such that S_ℓ is prefixed by $P[1..m']$; that is, $index = \min\{\, \ell \mid 1 \leq \ell \leq \texttt{h}$ and $P[1..m']$ is a prefix of $S_\ell \,\}$.

The running time analysis. The while-loop in the algorithm performs at most m iterations. Recall that m' denotes the length of the longest common prefix to return. So, $\texttt{pa-interval}_{m'+2}$ does not exist, and at most $\min(m'+1, m)$ iterations are executed. In

each iteration j, the queries `rank` and `select` over the array $\texttt{sVal}_j$, as well as the query `forePair` are called at most twice, respectively. By Lemma 2 and Theorem 13, a `rank` query takes $O(\log\log_w \sigma)$ time, a query `select` or $\texttt{forePair}_j$ takes $O(1)$ time. Hence, the overall running time of the algorithm is bounded by $O(m' \log\log_w \sigma)$.

When $\{S_1, S_2, \ldots, S_{\texttt{h}}\}$ are sorted lexicographically. In this case, we first invoke the procedure `Partial Prefix Search`$(P[1..m])$ to obtain $P[1..m']$, the count `occ`, and the index i of a haplotype that is prefixed by $P[1..m']$. We then compute the interval $[\gamma..\gamma']$ as $[i..i+\texttt{occ}-1]$. To verify correctness, observe that if $S_1, S_2, \ldots, S_{\texttt{h}}$ are presorted lexicographically, then the list $\{\gamma, \gamma+1, \ldots, \gamma'-1, \gamma'\}$ forms a sub-list of $\texttt{col}_j(\texttt{PA})$ for every $1 \leq j \leq \min(m'+1, m)$. Recall that $i = \min\{\, \ell \mid 1 \leq \ell \leq \texttt{h}$ and $P[1..m']$ is a prefix of $S_\ell \,\}$. Hence, it follows that $i = \gamma$ and $\gamma' = i + \texttt{occ} - 1$.

This completes the proof of Theorem 14. In Corollary 16, we extend this result to enumerate all haplotype indices prefixed by $P[1..m']$.

▶ **Corollary 16.** *There is an $O(\tilde{r} + \texttt{h})$-word data structure built over an arbitrary ordered list $\{S_1, \ldots, S_h\}$ of haplotypes such that, given a pattern $P[1..m]$, the indices of the haplotypes prefixed by $P[1..m']$ can be enumerated in $O(m' \log\log_w \sigma + \texttt{occ})$ time, where $P[1..m']$ is the longest common prefix between $P[1..m]$ and any haplotype and `occ` denotes the number of haplotypes prefixed by $P[1..m']$.*

Proof. We first sort all haplotypes in lexicographic order, build the data structure of Theorem 14 over the sorted list, and store the permutation $\pi^{-1}(i)$ for $1 \leq i \leq \texttt{h}$ such that $S_{\pi(1)} \prec S_{\pi(2)} \prec \cdots \prec S_{\pi(\texttt{h})}$. The resulting data structure uses $O(\tilde{r} + \texttt{h})$ words of space.

Given a query pattern $P[1..m]$, Theorem 14 allows us to find the longest common prefix $P[1..m']$ in $O(m' \log\log_w \sigma)$ time, as well as the interval $[\gamma, \gamma']$ such that $S_{\pi(i)}$ for any $\pi(i) \in [\gamma, \gamma']$ is prefixed by $P[1..m']$. The number `occ` of occurrences is then $\gamma' - \gamma + 1$, and the original indices, before sorting, of these haplotypes prefixed by $P[1..m']$ are $\{\pi^{-1}(\tau) \mid \tau \in [\gamma, \gamma']\}$. Using the permutation π^{-1} together with the interval $[\gamma, \gamma']$, the list of indices can be reported in $O(\texttt{occ})$ time. ◀

Handling haplotypes of arbitrary lengths. We consider a general setting where the haplotypes $\{S_1, \ldots, S_{\texttt{h}}\}$ have arbitrary length. We append a special symbol $\texttt{\#} \notin \{0, \ldots, \sigma-1\}$ – assumed to be the smallest – to the end of each haplotype. Next, we construct the PBWT representation `PBWT` and the prefix arrays `PA` for the new haplotypes, which consist of $\texttt{h} + \sum_{i \in [1..\texttt{h}]} |S_i|$ symbols in total.

The construction of `PBWT` proceeds column by column from left to right, as in the case of haplotypes of equal length. In this setting, however, both `PBWT` and the `PA` are no longer matrices; instead, each consists of `w` columns of possibly different lengths, denoted by $\{\texttt{PBWT}_1, \texttt{PBWT}_2, \ldots, \texttt{PBWT}_{\texttt{w}}\}$ and $\{\texttt{PA}_1, \texttt{PA}_2, \ldots, \texttt{PA}_{\texttt{w}}\}$, respectively. Here, `w` denotes the maximum extended haplotype length among $\{S_1, \ldots, S_{\texttt{h}}\}$, that is, $\texttt{w} = \max\{|S_i| + 1 \mid 1 \leq i \leq \texttt{h}\}$. When constructing the entries of $\texttt{PBWT}_j$ and $\texttt{PA}_j$, we exclude any haplotype S_i such that $j > |S_i| + 1$. Figure 5 shows an example of `PBWT` and `PA` built for haplotypes of arbitrary length.

The construction procedure described above ensures that each column $\texttt{PBWT}_j$ stores a permutation of all entries $S_i[j]$ for every $1 \leq i \leq \texttt{h}$ and $j \leq |S_i| + 1$. Note that $S_i[|S_i| + 1] = \texttt{\#}$. As a result, the symbol `#` appears in `PBWT` exactly `h` times across all columns. Let $\tilde{r}$ denote the total number of runs in `PBWT`, and let $\tilde{r}'$ denote the number of runs excluding those composed entirely of `#`. Since `#` occurs `h` times in `PBWT`, it follows that $\tilde{r} \leq \tilde{r}' + \texttt{h}$.

	1	2	3	4	5	6	7	8	9
S_1	a	b	a	a	b	a	#		
S_2	a	b	a	a	b	b	#		
S_3	a	b	a	c	#				
S_4	b	c	b	c	a	b	a	#	
S_5	b	c	b	c	a	b	b	#	
S_6	b	c	b	c	b	b	a	#	
S_7	b	c	b	c	b	b	b	a	#

(a) Extended haplotypes.

	1	2	3	4	5	6	7	8	9
1	a	b	a	a	b	b	#	#	#
2	a	b	a	a	b	b	a	#	
3	a	b	a	c	#	a	b	#	
4	b	c	b	c	a	b	#	a	
5	b	c	b	c	a	b	a		
6	b	c	b	c	b	b	b		
7	b	c	b	c	b				

(b) PBWT.

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	4	1	4	7
2	2	2	2	2	2	5	4	6	
3	3	3	3	3	3	1	5	5	
4	4	4	4	4	4	2	2	7	
5	5	5	5	5	5	6	6		
6	6	6	6	6	6	7	7		
7	7	7	7	7	7				

(c) PA.

Figure 5 Example of PBWT and PA built for haplotypes of arbitrary length.

Upon the runs of PBWT for the new haplotypes trailing with #, we build the same data structures as we have seen before for constructing sub-runs $\texttt{SubIF}_j$'s, for fore queries, and for prefix searches (Corollary 16). The space cost is bounded by $O(\tilde{r}+\mathtt{h})$ words, and a prefix search query can be answered in $O(m' \log\log_w(\sigma+1)+\mathtt{occ})$ time, since the new alphabet set is $\{0,\ldots,\sigma-1\}\cup\{\#\}$. Note that a prefix search query never calls $\mathtt{fore}[i][j]$ for any i and j such that $\mathtt{PBWT}_j[i]=\#$.

7.2 Haplotype Retrieval within $O(\tilde{r})$ Space

▶ **Theorem 17.** *The list of haplotypes $\{S_1,\ldots,S_\mathtt{h}\}$, each of length $\mathtt{w}$, can be represented in $O(\tilde{r})$ words of space, allowing S_i with any $1\le i\le \mathtt{h}$ to be retrieved in $O(\mathtt{w}+\log\log_w \mathtt{h})$ time.*

Proof. We construct the data structure, denoted by DS, as described in Theorem 13 for supporting fore queries. Let p_τ, for $1\le\tau\le|\texttt{SubIF}_1|$, be the starting positions of the τ-th sub-run in $\texttt{col}_1(\texttt{PBWT})$. Clearly, these positions $p_1,p_2,\ldots$ are sorted in increasing order. The list consists of exactly r_1 positions. We build the data structure from Lemma 1 over the list $p_1,p_2,\ldots$ to support predecessor queries. Since these positions are drawn from the universe $\{1,\ldots,\mathtt{h}\}$, each predecessor query can be answered in $O(\log\log_w\mathtt{h})$ time. The data structure requires $O(r_1+\tilde{r})=O(\tilde{r})$ words of space.

Given a query position i, we apply Lemma 1 to find the index x_1 of the sub-run that contains i; that is, x_1 is the position of the predecessor of i in the list $p_1,p_2,\ldots$. Using x_1, we can retrieve $\texttt{col}_1(\texttt{PBWT})[i]$ and $\texttt{forePair}_1(i,x_1)$ in constant time via DS. The former gives the entry $S_i[1]$, while the latter provides both the position i_2 – where i is stored in $\texttt{col}_2(\texttt{PA})$ – and the index x_2 of the sub-run in $\texttt{col}_2(\texttt{PBWT})$ that contains i_2. With x_2 and i_2, we can obtain the entry $S_i[2]$ and continue in the same manner for the subsequent columns. The algorithm terminates after all $\mathtt{w}$ entries of S_i have been retrieved.

The predecessor query is invoked once, costing $O(\log\log_w\mathtt{h})$ time by Lemma 1. The forePair query is invoked $\mathtt{w}$ times, and exactly $\mathtt{w}$ entries of the matrix PBWT are accessed. Thus, the total query time is bounded by $O(\log\log_w\mathtt{h}+\mathtt{w})$. ◀

8 Conclusions

In this work, we presented new $O(\tilde{r})$-space data structures and algorithms that support efficient forward and backward stepping operations on the PBWT. Our data structure enables constant-time computation of both fore and back operations. We also established lower and upper bounds on $\tilde{r}$. To demonstrate the utility of the optimal-time mapping, we revisited two applications: prefix search and haplotype retrieval. For the first application, we proposed an

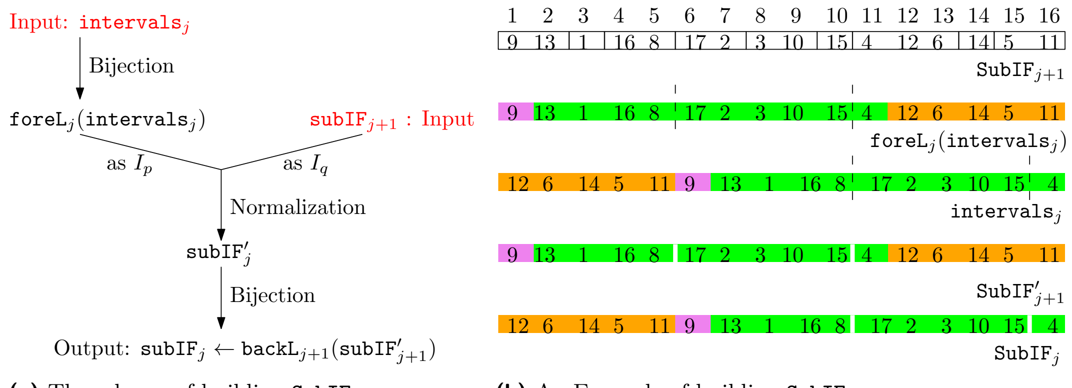
$O(\tilde{r}+\mathtt{h})$–word data structure that answers each query in $O(m' \log\log_w \sigma)$ time. When the haplotypes are provided in lexicographic order, the space requirement reduces to $O(\tilde{r})$ words. Moreover, our PBWT-based approach to prefix search naturally extends to haplotypes of arbitrary length. For the second application, we designed an $O(\tilde{r})$-word data structure that supports haplotype retrieval in $O(\mathtt{w} + \log\log_w \mathtt{h})$ time. While this work primarily focuses on the theoretical approach for achieving constant-time mapping in the run-length encoded PBWT, it also opens new directions for practical implementations and other applications, such as accelerating the computation of SMEMs [4] or MPSC in haplotype threading [13, 3]. We leave these aspects for future work.

References

1. Djamal Belazzougui and Gonzalo Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Transactions on Algorithms*, 11(4), 2015. doi:10.1145/2629339.
2. 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 - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, Lecture Notes in Computer Science, pages 89–101. Springer, 2023. doi:10.1007/978-3-031-43980-3_8.
3. Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, and Yuri Pirola. Solving the Minimal Positional Substring Cover Problem in Sublinear Space. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, volume 296 of *LIPIcs*, pages 12:1–12:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2024.12.
4. 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. *Bioinform.*, 39(9), 2023. doi:10.1093/BIOINFORMATICS/BTAD552.
5. Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinform.*, 30(9):1266–1272, 2014. doi:10.1093/BIOINFORMATICS/BTU014.
6. Travis Gagie, Giovanni Manzini, and Marinella Sciortino. Teaching the Burrows-Wheeler Transform via the Positional Burrows-Wheeler Transform. *CoRR*, abs/2208.09840, 2022. doi:10.48550/arXiv.2208.09840.
7. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.
8. Bjarni V Halldorsson, Hannes P Eggertsson, Kristjan HS Moore, Hannes Hauswedell, Ogmundur Eiriksson, Magnus O Ulfarsson, Gunnar Palsson, Marteinn T Hardarson, Asmundur Oddsson, Brynjar O Jensson, et al. The sequences of 150,119 genomes in the UK Biobank. *Nature*, pages 1–9, 2022. doi:10.1038/s41586-022-04965-x.
9. Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
10. Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. doi:10.1145/321479.321481.
11. Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Multi-allelic positional Burrows-Wheeler transform. *BMC Bioinform.*, 20-S(11):279:1–279:8, 2019. doi:10.1186/S12859-019-2821-6.
12. 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, Glasgow, Scotland (Virtual Conference), July 12-16, 2021*, LIPIcs, pages 101:1–101:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.101.

13 Ahsan Sanaullah, Degui Zhi, and Shaoije Zhang. Haplotype Threading Using the Positional Burrows-Wheeler Transform. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics, WABI 2022, Potsdam, Germany, September 5-7, 2022*, LIPIcs, pages 4:1–4:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.WABI.2022.4.

A The Figure Omitted from Section 5



(a) The scheme of building SubIF$_j$. **(b)** An Example of building SubIF$_j$.

Figure 6 Illustration of the algorithm scheme for building sub-runs in SubIF$_j$ and an example. In this example, SubIF$_{j+1}$ = { [1, 2], [3, 3], [4, 5], [6, 7], [8, 9], [10, 10], [11, 13], [14, 14], [15, 16] } and intervals$_j$ = { [1, 5], [6, 6], [7, 16] }. The bijective function foreL$_j$ maps the list intervals$_j$ to the list { [1, 1], [2, 11], [12, 16] }. Intervals highlighted in the same color contain the same haplotype indices and indicate corresponding pairs under this bijection. After applying the normalization algorithm to foreL$_j$(intervals$_j$) and SubIF$_{j+1}$, the intervals in foreL$_j$(intervals$_j$) are partitioned into SubIF$'_j$ = { [1, 1], [2, 5], [6, 10], [11, 11], [12, 16] }. Each interval in SubIF$'_{j+1}$ overlaps with at most three intervals in SubIF$_{j+1}$. Finally, by setting SubIF$_j$ = backL$_{j+1}$(SubIF$'_{j+1}$), the intervals intervals$_j$ are partitioned into SubIF$_j$ = {[1, 5], [6, 6], [7, 10], [11, 15], [16, 16]}.

B The Figure Omitted from Section 6

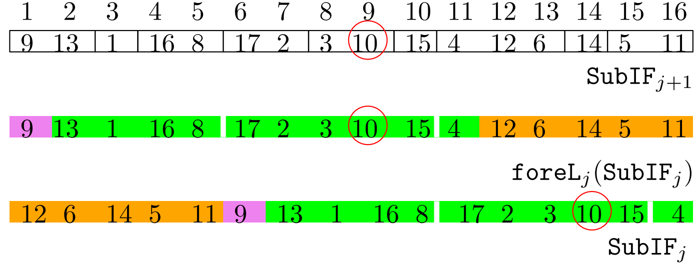


Figure 7 Example illustrating the data structure and algorithm for fore queries. The fourth interval [11, 15] in SubIF$_j$ corresponds to the third interval [6, 10] in foreL$_j$(SubIF$_j$), which overlaps three intervals in SubIF$_{j+1}$: the fourth [6, 7], the fifth [8, 9], and the sixth [10, 10]. Accordingly, the data structure F_j^4 built for the interval [11, 15] stores the quintuples $\{(11, 6, 6, 7, 4), (11, 6, 8, 9, 5), (11, 6, 10, 10, 6)\}$ in the form $(s', \tilde{s}, s, t, \lambda)$. Given a query fore[14][j] with index 4, the algorithm finds the tuple $(s' = 11, \tilde{s} = 6, s = 8, t = 9, \lambda = 5)$ in F_j^4 (since $14 - s' + \tilde{s} = 14 - 11 + 6 = 9 \in [8, 9]$) and returns $14 - s' + \tilde{s} = 9$ and $\lambda = 5$ as the answer.

C The Pseudocode for Prefix Searches

```
Partial Prefix Search(P[1..m])
01.     (b, e, x, x', index) ← (1, h, 1, |SubIF_1|, sPA_1[1]);
02.     j ← 1;
03.     while j ≤ m do
04.         c ← P[j];
05.         (b̃, x̃) ← (b, x);
06.         if sVal_j[x] ≠ c then
07.             count ← rank_c(sVal_j, x);
08.             x̃ ← select_c(sVal_j, count + 1);
09.             if x̃ = |SubIF_j| + 1 then
10.                 break;
11.             b̃ ← SubIF_j[x̃].b;
12.             index ← sPA_j[x̃];
13.         (ẽ, x̃') ← (e, x');
14.         if sVal_j[x'] ≠ c then
15.             count ← rank_c(sVal_j, x');
16.             x̃' ← select_c(sVal_j, count);
17.             ẽ ← SubIF_j[x'].e;
18.         if j < m then
19.             (b, x) ← forePair_j(b̃, x̃);
20.             (e, x') ← forePair_j(ẽ, x̃');
21.         else
22.             (b, x) ← (b̃, x̃);
23.             (e, x') ← (ẽ, x̃');
24.         j ← j + 1;
25.     if j = 1 then
26.         return the longest common prefix is empty;
27.     else if 1 < j ≤ m
28.         return P[1..j − 1], e − b + 1, index;
29.     else
30.         count_1 ← sCount_m[x] + b − SubIF_m[x].b;
31.         count_2 ← sCount_m[x'] + e − SubIF_m[x'].b;
32.         count ← count_2 − count_1 + 1;
33.         return P[1..m], count, index;
```