Minimum Segmentation for Pan-genomic Founder Reconstruction in Linear Time

Tuukka Norri
Department of Computer Science, University of Helsinki, Helsinki, Finland
tuukka.norri@helsinki.fi
https://orcid.org/0000-0002-8276-0585

Bastien Cazaux
Department of Computer Science, University of Helsinki, Helsinki, Finland
bastien.cazaux@helsinki.fi
https://orcid.org/0000-0002-1761-4354

Dmitry Kosolobov
Department of Computer Science, University of Helsinki, Helsinki, Finland
dkosolobov@mail.ru
https://orcid.org/0000-0002-2909-2952

Veli Mäkinen
Department of Computer Science, University of Helsinki, Helsinki, Finland
veli.makinen@helsinki.fi
https://orcid.org/0000-0003-4454-1493

Abstract

Given a threshold L and a set $\mathcal{R} = \{R_1, \ldots, R_m\}$ of m strings (haplotype sequences), each having length n, the minimum segmentation problem for founder reconstruction is to partition $[1, n]$ into set P of disjoint segments such that each segment $[a, b] \in P$ has length at least L and the number $d(a, b) = |\{R_i[a, b] \colon 1 \leq i \leq m\}|$ of distinct substrings at segment $[a, b]$ is minimized over $[a, b] \in P$. The distinct substrings in the segments represent founder blocks that can be concatenated to form $\max\{d(a, b) \colon [a, b] \in P\}$ founder sequences representing the original $\mathcal{R}$ such that crossovers happen only at segment boundaries. We give an optimal $O(mn)$ time algorithm to solve the problem, improving over earlier $O(mn^2)$. This improvement enables to exploit the algorithm on a pan-genomic setting of input strings being aligned haplotype sequences of complete human chromosomes, with a goal of finding a representative set of references that can be indexed for read alignment and variant calling. We implemented the new algorithm and give some experimental evidence on the practicality of the approach on this pan-genomic setting.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Applied computing → Bioinformatics

Keywords and phrases Pan-genome indexing, founder reconstruction, dynamic programming, positional Burrows–Wheeler transform, range minimum query

Digital Object Identifier 10.4230/LIPIcs.WABI.2018.15

Funding This work is partially supported by the Academy of Finland (grant 309048).

1 Introduction

A key problem in *pan-genomics* is to develop a sufficiently small, efficiently queriable, but still descriptive representation of the variation common to the subject under study [1]. For example, when studying human population, one would like to take all publicly available

18th International Workshop on Algorithms in Bioinformatics (WABI 2018).
Editors: Laxmi Parida and Esko Ukkonen; Article No. 15; pp. 15:1–15:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

variation datasets (e.g. [19, 4, 20]) into account. Many approaches encode the variation as a graph [16, 9, 18, 2, 10, 8] and then one can encode the different haplotypes as paths in this graph [13, 17]. An alternative was proposed in [22], based on a compressed indexing scheme for a multiple alignment of all the haplotypes [11, 14, 23, 5, 7]. In either approach, scalability is hampered by the encoding of all the haplotypes.

We suggest to look for a smaller set of representative haplotype sequences to make the above pan-genomic representations scalable.

Finding such set of representative haplotype sequences that retain the original contiguities as well as possible, is known as the *founder sequence reconstruction* problem [21]. In this problem, one seeks a set of d founders such that the original m haplotypes can be mapped with minimum amount of *crossovers* to the founders. Here a crossover means a position where one needs to jump from one founder to another to continue matching the content of the haplotype in question. Unfortunately, this problem in NP-hard even to approximate within a constant factor [15].

For founder reconstruction to be scalable to the pan-genomic setting, one would need an algorithm to be nearly linear to the input size. With this is mind, we study a relaxation of founder reconstruction that is known to be polynomial time solvable: Namely, when limiting all the crossovers to happen at the same locations, one obtains a *minimum segmentation* problem specific to founder reconstruction [21]. A dynamic programming algorithm given in [21] has complexity $O(n^2m)$, where m is the number of haplotypes and n is the length of each of them.

In this paper, we improve the running time of solving the minimum segmentation problem of founder reconstruction to the optimal $O(mn)$ (linear in the input size).

We also implement the new algorithm, as well as a further heuristic that aims to minimize crossovers over the segment boundaries (yielded by the optimal solution to the minimum segmentation problem). In our experiments, we show that the approach is practical on human genome scale setting. Namely, we apply the implementation on a multiple alignment representing 5009 haplotypes of human chromosome 6, and the result is 130 founder sequences with the average distance of two crossovers being 9358 bases. Preserving such long contiguities in just 2.5% of the original input space is promising for the accuracy and scalability of the short read alignment and variant calling motivating our study.

The main technique behind the improvement is the use of *positional Burrows–Wheeler transform* (pBWT) [3], and more specifically its extension to larger alphabets [12]. While the original dynamic programming solution uses $O(nm)$ time to look for the best preceding segment boundary for each column of the input, we observe that at most m values in pBWT determine segment boundaries where the number of distinct founder substrings change. Minimums on the already computed dynamic programming values between each such interesting consecutive segment boundaries give the requested result. However, it turns out that we can maintain the minimums directly in pBWT internal structures (with some modifications) and have to store only the last L computed dynamic programming values, thus spending only $O(m + L)$ additional space, where L is the input threshold on the length of each segment. The segmentation is then reconstructed by standard backtracking approach in $O(n)$ time using an array of length n.

2 Notation and Problem Statement

For a string $s = c_1c_2 \cdots c_n$, denote by $|s|$ its length n. We write $s[i]$ for the letter c_i of s and $s[i, j]$ for the *substring* $c_ic_{i+1} \cdots c_j$. An analogous notation is used for arrays. For any numbers i and j, the set of integers $\{x \in \mathbb{Z} \colon i \leq x \leq j\}$ (possibly empty) is denoted by $[i, j]$.

The input for our problem is the set $\mathcal{R} = \{R_1, \ldots, R_m\}$ of strings of length n, called *recombinants*. A set $\mathcal{F} = \{F_1, \ldots, F_d\}$ of strings of length n is called a *founder set* of $\mathcal{R}$ if for each string $R_i \in \mathcal{R}$, there exists a partition P_i of the segment $[1, n]$ into disjoint subsegments such that, for each $[a, b] \in P_i$, the string $R_i[a, b]$ is equal to $F_j[a, b]$ for some $j \in [1, d]$. The partition P_i together with the mapping of the segments $[a, b] \in P_i$ to substrings $F_j[a, b]$ is called a *parse of* R_i *in terms of* $\mathcal{F}$, and a set of parses for all $R_i \in \mathcal{R}$ is called a *parse of* $\mathcal{R}$ *in terms of* $\mathcal{F}$. The integers a and $b + 1$, for $[a, b] \in P_i$, are called *crossover points*; thus, in particular, 1 and $n + 1$ are always crossover points.

It follows from the definition that, in practice, it makes sense to consider founder sets only for pre-aligned recombinants. Throughout the paper we implicitly assume that this is the case, although all our algorithms, clearly, work in the unaligned setting too but the produce results may hardly make any sense.

We consider the problem of finding a "good" founder set $\mathcal{F}$ and a "good" corresponding parse of $\mathcal{R}$ according to a reasonable measure of goodness. Ukkonen [21] pointed out that such measures may contradict each other: for instance, a minimum founder set obviously has size $d = \max_{j \in [1,n]} |\{R_1[j], \ldots, R_m[j]\}|$, but parses corresponding to such set may have unnaturally many crossover points; conversely, $\mathcal{R}$ is a founder set of itself and the only crossover points of its trivial parse are 1 and $n + 1$, but the size m of this founder set is in most cases unacceptably large. Following Ukkonen's approach, we consider compromise parameterized solutions. The *minimum founder set problem* is, given a bound L and a set of recombinants $\mathcal{R}$, to find a smallest founder set $\mathcal{F}$ of $\mathcal{R}$ such that there exists a parse of $\mathcal{R}$ in terms of $\mathcal{F}$ in which the distance between any two crossover points is at least L (the crossover points may belong to parses of different recombinants, i.e., for $[a, b] \in P_i$ and $[a', b'] \in P_j$, where P_i and P_j are parses of R_i and R_j, we have either $a = a'$ or $|a - a'| \geq L$).

It is convenient to reformulate the problem in terms of segmentations of $\mathcal{R}$. A *segment* of $\mathcal{R} = \{R_1, \ldots, R_m\}$ is a set $\mathcal{R}[j, k] = \{R_i[j, k] \colon R_i \in \mathcal{R}\}$. A *segmentation* of $\mathcal{R}$ is a collection S of disjoint segments that covers the whole $\mathcal{R}$, i.e., for any distinct $\mathcal{R}[j, k]$ and $\mathcal{R}[j', k']$ from S, $[j, k]$ and $[j', k']$ do not intersect and, for each $x \in [1, n]$, there is $\mathcal{R}[j, k]$ from S such that $x \in [j, k]$. The *minimum segmentation problem* [21] is, given a bound L and a set of recombinants $\mathcal{R}$, to find a segmentation S of $\mathcal{R}$ such that $\max\{|\mathcal{R}[j, k]| \colon \mathcal{R}[j, k] \in S\}$ is minimized and the length of each segment from S is at least L; in other words, the problem is to compute

$$\min_{S \in S_L} \max\{|\mathcal{R}[j, k]| \colon \mathcal{R}[j, k] \in S\}, \tag{1}$$

where S_L is the set of all segmentations in which all segments have length at least L.

The minimum founder set problem and the minimum segmentation problem are, in a sense, equivalent: any segmentation S with segments of length at least L induces in an obvious way a founder set of size $\max\{|\mathcal{R}[j, k]| \colon \mathcal{R}[j, k] \in S\}$ and a parse in which all crossover points are located at segment boundaries (and, hence, at distance at least L from each other); conversely, if $\mathcal{F}$ is a founder set of $\mathcal{R}$ and $\{j_1, \ldots, j_p\}$ is the sorted set of all crossover points in a parse of $\mathcal{R}$ such that $j_q - j_{q-1} \geq L$ for $q \in [2, p]$, then $S = \{\mathcal{R}[j_{q-1}, j_q - 1] \colon q \in [2, p]\}$ is a segmentation of $\mathcal{R}$ with segments of length at least L and $\max\{|\mathcal{R}[j, k]| \colon \mathcal{R}[j, k] \in S\} \leq |\mathcal{F}|$.

Our main result is an algorithm that solves the minimum segmentation problem in the optimal $O(mn)$ time. The solution normally does not uniquely define a founder set of $\mathcal{R}$: for instance, if the built segmentation of $\mathcal{R} = \{baaaa, baaab, babab\}$ is $S = \{\mathcal{R}[1, 1], \mathcal{R}[2, 3], \mathcal{R}[4, 5]\}$, then the possible founder sets induced by S are $\mathcal{F}_1 = \{baaab, babaa\}$ and $\mathcal{F}_2 = \{baaaa, babab\}$. In other words, to construct a founder set, one concatenates fragments of recombinants corresponding to the found segments in a certain order. We return to this ordering problem in Sect. 4 and now focus on the details of the segmentation problem.

Hereafter, we assume that the input alphabet Σ is the set $[0, |\Sigma|-1]$ of size $O(m)$, which is a natural assumption considering that the typical alphabet size is 4 in our problem. It is sometimes convenient to view the set $\mathcal{R} = \{R_1, \ldots, R_m\}$ as a matrix with m rows and n columns. We say that an algorithm processing the recombinants $\mathcal{R}$ is streaming if it reads the input from left to right "columnwise", for each k from 1 to n, and outputs an answer for each set of recombinants $\{R_1[1,k], \ldots, R_m[1,k]\}$ immediately after reading the "column" $\{R_1[k], \ldots, R_m[k]\}$. The main result of the paper is the following theorem.

▶ **Theorem 1.** *Given a bound L and recombinants $\mathcal{R} = \{R_1, \ldots, R_m\}$, each having length n, there is an algorithm that computes* (1) *in a streaming fashion in the optimal $O(mn)$ time and $O(m+L)$ space. Using an additional array of length n, one can also find in $O(n)$ time a segmentation on which* (1) *is attained, thus solving the minimum segmentation problem.*

3 Minimum Segmentation Problem

Given a bound L and a set of recombinants $\mathcal{R} = \{R_1, \ldots, R_m\}$ each of which has length n, Ukkonen [21] proposed a dynamic programming algorithm that solves the minimum segmentation problem in $O(mn^2)$ time based on the following recurrence relation:

$$M(k) = \begin{cases} +\infty & \text{if } k < L, \\ |\mathcal{R}[1,k]| & \text{if } L \le k < 2L, \\ \min\limits_{0 \le j \le k-L} \max\{M(j), |\mathcal{R}[j+1,k]|\} & \text{if } k \ge 2L. \end{cases} \tag{2}$$

It is obvious that $M(n)$ is equal to the solution (1); the segmentation itself can be reconstructed by "backtracking" in a standard way (see [21]). We build on the same approach.

For a given $k \in [1,n]$, denote by $j_{k,1}, \ldots, j_{k,r_k}$ the sequence of all positions $j \in [1, k-L]$ in which the value of $|\mathcal{R}[j,k]|$ changes, i.e., $1 \le j_{k,1} < \cdots < j_{k,r_k} \le k-L$ and $|\mathcal{R}[j_{k,h}, k]| \neq |\mathcal{R}[j_{k,h}+1, k]|$ for $h \in [1, r_k]$. We complement this sequence with $j_{k,0} = 0$ and $j_{k,r_k+1} = k-L+1$, so that $j_{k,0}, \ldots, j_{k,r_k+1}$ can be interpreted as a splitting of the range $[0, k-L]$ into segments in which the value $|\mathcal{R}[j+1,k]|$ stays the same: namely, for $h \in [0, r_k]$, one has $|\mathcal{R}[j+1,k]| = |\mathcal{R}[j_{k,h+1}, k]|$ provided $j_{k,h} \le j < j_{k,h+1}$. Hence, $\min\limits_{j_{k,h} \le j < j_{k,h+1}} \max\{M(j), |\mathcal{R}[j+1,k]|\} = \max\{|\mathcal{R}[j_{k,h+1}, k]|, \min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)\}$ and, therefore, (2) can be rewritten as follows:

$$M(k) = \begin{cases} +\infty & \text{if } k < L, \\ |\mathcal{R}[1,k]| & \text{if } L \le k < 2L, \\ \min\limits_{0 \le h \le r_k} \max\{|\mathcal{R}[j_{k,h+1}, k]|, \min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)\} & \text{if } k \ge 2L. \end{cases} \tag{3}$$

Our crucial observation is that, for $k \in [1,n]$ and $j \in [1,k]$, one has $|\mathcal{R}[j+1,k]| \le |\mathcal{R}[j,k]| \le m$. Therefore, $m \ge |\mathcal{R}[j_{k,1}, k]| > \cdots > |\mathcal{R}[j_{k,r_k+1}, k]| \ge 1$ and $r_k < m$. Hence, $M(k)$ can be computed in $O(m)$ time using (3), provided one has the following components:

(i) the numbers $|\mathcal{R}[j_{k,h+1}, k]|$, for $h \in [0, r_k]$;

(ii) the values $\min\{M(j) \colon j_{k,h} \le j < j_{k,h+1}\}$, for $h \in [0, r_k]$.

In the remaining part of the section, we describe a streaming algorithm that reads the strings $\{R_1, \ldots, R_m\}$ "columnwise" from left to right and computes the components (i) and (ii) immediately after reading each "column" $\{R_1[k], \ldots, R_m[k]\}$, for $k \in [1,n]$, and all in $O(mn)$ total time and $O(m+L)$ space.

To reconstruct a segmentation corresponding to the found solution $M(n)$, we build along with the values $M(k)$ an array of size n whose kth element, for each $k \in [1,n]$, stores 0 if $M(k) = |\mathcal{R}[1,k]|$, and stores a number $j \in [1, k-L]$ such that $M(k) = \max\{M(j), |\mathcal{R}[j+1,k]|\}$

$R_1 = tttccat$
$R_2 = accatta$
$R_3 = actacct$
$R_4 = actccat$
$R_5 = cttacct$
$R_6 = atcacat$

i	1	2	3	4	5	6
$a_k[i]$	2	6	4	1	3	5
$d_k[i]$	8	8	5	3	7	3

i	1	2	3	4	5	6	7
$\lvert\mathcal{R}[i,k]\rvert$	6	6	4	4	3	3	2

$a_k[1] = 2$
$a_k[2] = 6$
$a_k[3] = 4$
$a_k[4] = 1$
$a_k[5] = 3$
$a_k[6] = 5$

a	c	c	a	t	t	a
a	t	c	a			
a	c					
t	t	t	c	c	a	t
a	c	t	a	c	c	
c	t					

Figure 1 The pBWT for a set of recombinants $\mathcal{R} = \{R_1, \ldots, R_6\}$ with $k = 7$ and the corresponding trie containing the reversed strings $R_1[1,k], \ldots, R_6[1,k]$ in lexicographic order.

otherwise; then, the segmentation can be reconstructed from the array in an obvious way in $O(n)$ time. In order to maintain the array, our algorithm computes, for each $k \in [1,n]$, along with the values $\min\{M(j) \colon j_{k,h} \le j < j_{k,h+1}\}$, for $h \in [0, r_k]$, positions j on which these minima are attained (see below). Further details are straightforward and, thence, omitted.

3.1 Positional Burrows–Wheeler Transform

Let us fix $k \in [1,n]$. Throughout this subsection, the string $R_i[k]R_i[k-1]\cdots R_i[1]$, which is the reversal of $R_i[1,k]$, is denoted by $R'_{i,k}$, for $i \in [1,m]$. Given a set of recombinants $\mathcal{R} = \{R_1, \ldots, R_m\}$ each of which has length n, a *positional Burrows–Wheeler transform (pBWT)*, as defined by Durbin [3], is a pair of integer arrays $a_k[1,m]$ and $d_k[1,m]$ such that:

1. $a_k[1,m]$ is a permutation of $[1,m]$ such that $R'_{a_k[1],k} \le \cdots \le R'_{a_k[m],k}$ lexicographically;
2. $d_k[i]$, for $i \in [1,m]$, is an integer such that $R_{a_k[i]}[d_k[i],k]$ is the longest common suffix of $R_{a_k[i]}[1,k]$ and $R_{a_k[i-1]}[1,k]$, and $d_k[i] = k+1$ if either this suffix is empty or $i = 1$.

▶ **Example 2.** Consider the following example, where $m = 6$, $k = 7$, and $\Sigma = \{a, c, t\}$. It is easy to see that the pBWT implicitly encodes the trie depicted in the right part of Figure 1, and such interpretation drives the intuition behind this structure: The trie represents the *reversed* sequences $R_1[1,k], \ldots, R_6[1,k]$ (i.e., read from right to left) in lexicographic order. Leaves (values a_k) store the corresponding input indices. The branches correspond to values d_k (the distance from the root subtracted from $k+1$). Our main algorithm in this paper makes implicitly a sweep-line over the trie stopping at the branching positions.

Durbin [3] showed that a_k and d_k can be computed from a_{k-1} and d_{k-1} in $O(m)$ time on the binary alphabet. Mäkinen and Norri [12] further generalized the construction for integer alphabets of size $O(m)$, as in our case. For the sake of completeness, we describe in this subsection the solution from [12] (see Figure 2a), which serves then as a basis for our main algorithm. We also present a modification of this solution (see Figure 2b), which, albeit seems to be slightly inferior in theory (we could prove only $O(m \log |\Sigma|)$ time upper bound), showed better performance in practice and thus, as we believe, is interesting by itself.

▶ **Lemma 3.** *The arrays $a_k[1,m]$ and $d_k[1,m]$ can be computed from $a_{k-1}[1,m]$ and $d_{k-1}[1,m]$ in $O(m)$ time, assuming the input alphabet is $[0, |\Sigma|-1]$ with $|\Sigma| = O(m)$.*

Proof. Given a_{k-1} and d_{k-1}, we are to show that the algorithm from Figure 2a correctly computes a_k and d_k. Since, for any $i, j \in [1,m]$, we have $R'_{i,k} \le R'_{j,k}$ iff either $R_i[k] < R_j[k]$, or $R_i[k] = R_j[k]$ and $R'_{i,k-1} \le R'_{j,k-1}$ lexicographically, it is easy to see that the array a_k can be deduced from a_{k-1} by radix sorting the sequence of pairs $\{(R_{a_{k-1}[i]}[k], R'_{a_{k-1}[i],k-1})\}_{i=1}^{m}$. Further, since, by definition of a_{k-1}, the second components of the pairs are already in a sorted order, it remains to sort the first components by the counting sort. Accordingly, in

zero initialize $C[0, |\Sigma|]$ and $P[0, |\Sigma| - 1]$
for $i \leftarrow 1$ **to** m **do**
  $C[R_i[k] + 1] \leftarrow C[R_i[k] + 1] + 1$;
for $i \leftarrow 1$ **to** $|\Sigma| - 1$ **do** $C[i] \leftarrow C[i] + C[i - 1]$;
for $i \leftarrow 1$ **to** m **do**
  $b \leftarrow R_{a_{k-1}[i]}[k]$;
  $C[b] \leftarrow C[b] + 1$;
  $a_k[C[b]] \leftarrow a_{k-1}[i]$;
  if $P[b] = 0$ **then** $d_k[C[b]] \leftarrow k + 1$;
  else $d_k[C[b]] \leftarrow \max\{d_{k-1}[\ell] \colon P[b] < \ell \leq i\}$;
  $P[b] \leftarrow i$;

(a) The basic pBWT algorithm computing a_k and d_k from a_{k-1} and d_{k-1}.

zero initialize $C[0, |\Sigma|]$ and $P[0, |\Sigma| - 1]$
for $i \leftarrow 1$ **to** m **do**
  $C[R_i[k] + 1] \leftarrow C[R_i[k] + 1] + 1$;
for $i \leftarrow 1$ **to** $|\Sigma| - 1$ **do** $C[i] \leftarrow C[i] + C[i - 1]$;
for $i \leftarrow 1$ **to** m **do**
  $b \leftarrow R_{a_{k-1}[i]}[k]$;
  $C[b] \leftarrow C[b] + 1$;
  $a_k[C[b]] \leftarrow a_{k-1}[i]$;
  $a_{k-1}[i] \leftarrow i + 1$; ▷ erase $a_{k-1}[i]$
  if $P[b] = 0$ **then** $d_k[C[b]] \leftarrow k + 1$;
  else $d_k[C[b]] \leftarrow \mathsf{maxd}(P[b] + 1, i)$;
  $P[b] \leftarrow i$;
function $\mathsf{maxd}(j,\ i)$
  if $j \neq i$ **then**
    $d_{k-1}[j] \leftarrow \max\{d_{k-1}[j], \mathsf{maxd}(a_{k-1}[j], i)\}$;
    $a_{k-1}[j] \leftarrow i + 1$;
  return $d_{k-1}[j]$;

(b) The algorithm with simple RMQ; a_{k-1} and d_{k-1} are used as auxiliary arrays (and corrupted).

Figure 2 The computation of a_k and d_k from a_{k-1} and d_{k-1} in the pBWT.

Figure 2a, the first loop counts occurrences of letters in the sequence $\{R_i[k]\}_{i=1}^m$ using an auxiliary array $C[0, |\Sigma|]$; as is standard in the counting sort, the second loop modifies the array C so that, for each letter $b \in [0, |\Sigma|-1]$, $C[b] + 1$ is the first index of the "bucket" that will contain all $a_{k-1}[i]$ such that $R_{a_{k-1}[i]}[k] = b$; finally, the third loop fills the buckets incrementing the indices $C[b] \leftarrow C[b] + 1$, for $b = R_{a_{k-1}[i]}[k]$, and performing the assignments $a_k[C[b]] \leftarrow a_{k-1}[i]$, for $i = 1, \ldots, m$. Thus, the array a_k is computed correctly. All is done in $O(m + |\Sigma|)$ time, which is $O(m)$ since the input alphabet is $[0, |\Sigma|-1]$ and $|\Sigma| = O(m)$.

The last three lines of the algorithm are responsible for computing d_k. Denote the length of the longest common prefix of any strings s_1 and s_2 by $\mathtt{LCP}(s_1, s_2)$. The computation of d_k relies on the following well-known fact: given a sequence of strings $s_1, \ldots, s_r$ such that $s_1 \leq \cdots \leq s_r$ lexicographically, one has $\mathtt{LCP}(s_1, s_r) = \min\{\mathtt{LCP}(s_{i-1}, s_i) \colon 1 < i \leq r\}$. Suppose that the last loop of the algorithm, which iterates through all i from 1 to m, assigns $a_k[i'] \leftarrow a_{k-1}[i]$, for a given $i \in [1, m]$ and some $i' = C[b]$. Let j be the maximum integer such that $j < i$ and $R_{a_{k-1}[j]}[k] = R_{a_{k-1}[i]}[k]$ (if any). The definition of a_k implies that $a_k[i'-1] = a_{k-1}[j]$ if such j exists. Hence, $\mathtt{LCP}(R'_{a_k[i'-1],k}, R'_{a_k[i'],k}) = 1 + \min\{\mathtt{LCP}(R'_{a_{k-1}[\ell-1],k-1}, R'_{a_{k-1}[\ell],k-1}) \colon j < \ell \leq i\}$ if such number j does exist, and $\mathtt{LCP}(R'_{a_k[i'-1],k}, R'_{a_k[i'],k}) = 0$ otherwise. Therefore, since $d_k[i']$ equals $k+1-\mathtt{LCP}(R'_{a_k[i'],k}, R'_{a_k[i'-1],k})$, we have either $d_k[i'] = \max\{d_{k-1}[\ell] \colon j < \ell \leq i\}$ or $d_k[i'] = k+1$ according to whether the required j exists. To find j, we simply maintain an auxiliary array $P[0, |\Sigma|-1]$ such that on the ith loop iteration, for any letter $b \in [0, |\Sigma|-1]$, $P[b]$ stores the position of the last seen b in the sequence $R_{a_{k-1}[1]}[k], R_{a_{k-1}[2]}[k], \ldots, R_{a_{k-1}[i-1]}[k]$, or $P[b] = 0$ if b occurs for the first time. Thus, d_k is computed correctly.

In order to calculate the maximums $\max\{d_{k-1}[\ell] \colon P[b] \leq \ell \leq i\}$ in $O(1)$ time, we build a range maximum query (RMQ) data structure on the array $d_{k-1}[1, m]$ in $O(m)$ time (e.g., see [6]). Therefore, the running time of the algorithm from Figure 2a is $O(m)$. ◀

In practice the bottleneck of the algorithm is the RMQ data structure, which, although answers queries in $O(1)$ time, has a sensible constant under the big-O in the construction time. We could naively compute the maximums by scanning the ranges $d_{k-1}[P[b]+1, i]$ from

left to right but such algorithm works in quadratic time since same ranges of d_{k-1} might be processed many times in the worst case. Our key idea is to store the work done by a simple scanning algorithm to reuse it in future queries. We store this information right in the arrays a_{k-1} and d_{k-1} rewriting them; in particular, since a_{k-1} is accessed sequentially from left to right in the last loop, the range $a_{k-1}[1, i]$ is free to use after the ith iteration.

More precisely, after the ith iteration of the last loop, the subarrays $a_{k-1}[1, i]$ and $d_{k-1}[1, i]$ are modified so that the following invariant holds: for any $j \in [1, i]$, $j < a_{k-1}[j] \leq i + 1$ and $d_{k-1}[j] = \max\{d'_{k-1}[\ell] \colon j \leq \ell < a_{k-1}[j]\}$, where d'_{k-1} denotes the original array d_{k-1} before modifications; note that the invariant holds if one simply puts $a_{k-1}[j] = j + 1$ without altering $d_{k-1}[j]$. Then, to compute $\max\{d'_{k-1}[\ell] \colon j \leq \ell \leq i\}$, we do not have to scan all elements but can "jump" through the chain $j, a_{k-1}[j], a_{k-1}[a_{k-1}[j]], \ldots, i$ and use maximums precomputed in $d_{k-1}[j], d_{k-1}[a_{k-1}[j]], d_{k-1}[a_{k-1}[a_{k-1}[j]]], \ldots, d_{k-1}[i]$; after this, we redirect the "jump pointers" in a_{k-1} to $i + 1$ and update the maximums in d_{k-1} accordingly. This idea is implemented in Figure 2b. Notice the new line $a_{k-1}[i] \leftarrow i + 1$ in the main loop (it is commented), which erases $a_{k-1}[i]$ and makes it a part of the "jump table". The correctness of the algorithm is clear. But it is not immediate even that the algorithm works in $O(m \log m)$ time. The next lemma states that the bound is actually even better, $O(m \log |\Sigma|)$. The proof of the lemma is given in Appendix.

▶ **Lemma 4.** *The algorithm from Figure 2b computes the arrays $a_k[1, m]$ and $d_k[1, m]$ from $a_{k-1}[1, m]$ and $d_{k-1}[1, m]$ in $O(m \log |\Sigma|)$ time, assuming the input alphabet is $[0, |\Sigma|-1]$ with $|\Sigma| = O(m)$.*

3.2 Modification of the pBWT

We are to modify the basic pBWT construction algorithm in order to compute the sequence $j_{k,1}, \ldots, j_{k,r_k}$ of all positions $j \in [1, k - L]$ in which $|\mathcal{R}[j, k]| \neq |\mathcal{R}[j + 1, k]|$, and to calculate the numbers $|\mathcal{R}[j_{k,h+1}, k]|$ and $\min\{M(j) \colon j_{k,h} \leq j < j_{k,h+1}\}$, for $h \in [0, r_k]$ (assuming $j_{k,0} = 0$ and $j_{k,r_k+1} = k - L + 1$); see the beginning of the section. As it follows from (3), these numbers are sufficient to calculate $M(k)$, as defined in (2) and (3), in $O(m)$ time. The following lemma reveals relations between the sequence $j_{k,1}, \ldots, j_{k,r_k}$ and the array d_k.

▶ **Lemma 5.** *Consider recombinants $\mathcal{R} = \{R_1, \ldots, R_m\}$, each having length n. For $k \in [1, n]$ and $j \in [1, k - 1]$, one has $|\mathcal{R}[j, k]| \neq |\mathcal{R}[j + 1, k]|$ iff $j = d_k[i] - 1$ for some $i \in [1, m]$.*

Proof. Suppose that $|\mathcal{R}[j, k]| \neq |\mathcal{R}[j + 1, k]|$. It is easy to see that $|\mathcal{R}[j, k]| > |\mathcal{R}[j + 1, k]|$, which implies that there are two indices h and h' such that $R_h[j + 1, k] = R_{h'}[j + 1, k]$ and $R_h[j] \neq R_{h'}[j]$. Denote by $a_k^{-1}[h]$ the number x such that $a_k[x] = h$. Without loss of generality, assume that $a_k^{-1}[h] < a_k^{-1}[h']$. Then, there exists $i \in [a_k^{-1}[h] + 1, a_k^{-1}[h']]$ such that $R_{a_k[i-1]}[j + 1, k] = R_{a_k[i]}[j + 1, k]$ and $R_{a_k[i-1]}[j] \neq R_{a_k[i]}[j]$. Hence, $d_k[i] = j + 1$.

Suppose now that $j \in [1, k - 1]$ and $j = d_k[i] - 1$, for some $i \in [1, m]$. Since $j < k$ and $d_k[1] = k + 1$, we have $i > 1$. Then, by definition of d_k, $R_{a_k[i-1]}[j + 1, k] = R_{a_k[i]}[j + 1, k]$ and $R_{a_k[i-1]}[j] \neq R_{a_k[i]}[j]$, i.e., $R_{a_k[i]}[j + 1, k]$ can be "extended" to the left in two different ways, thus producing two distinct strings in the set $\mathcal{R}[j, k]$. Therefore, $|\mathcal{R}[j, k]| > |\mathcal{R}[j + 1, k]|$. ◀

Denote by r the number of distinct integers in the array d_k. Clearly, r may vary from 1 to m. For integer ℓ, define $M'(\ell) = M(\ell)$ if $1 \leq \ell \leq k - L$, and $M'(\ell) = +\infty$ otherwise (M' is introduced for purely technical reasons). Our modified algorithm does not store d_k but stores the following four arrays (but we still often refer to d_k for the sake of analysis):

- $s_k[1, r]$ contains all distinct elements from $d_k[1, m]$ in the increasing sorted order;
- $e_k[1, m]$: for $j \in [1, r]$, $e_k[j]$ is equal to the unique index such that $s_k[e_k[j]] = d_k[j]$;

- $t_k[1,r]$: for $j \in [1,r]$, $t_k[j]$ is equal to the number of times $s_k[j]$ occurs in $d_k[1,m]$;
- $u_k[1,r]$: for $j \in [1,r]$, $u_k[j] = \min\{M'(\ell)\colon s_k[j-1]-1 \le \ell < s_k[j]-1\}$, assuming $s_k[0] = 1$.

▶ **Example 6.** In Example 2, where $m = 6$, $k = 7$, and $\Sigma = \{a, c, t\}$, we have $r = 4$, $s_k = [3, 5, 7, 8]$, $t_k = [2, 1, 1, 2]$, $e_k = [4, 4, 2, 1, 3, 1]$. It is easy to see that the array s_k marks positions of the branching nodes in the trie from Figure 1 in the increasing order (in the special case $s_k[1] = 1$, $s_k[1]$ does not mark any such node). The arrays s_k and e_k together emulate d_k. The array t_k will be used below to calculate some numbers $|\mathcal{R}[j,k]|$ required to compute $M(k)$. Further, suppose that $L = 3$, so that $k - L = 4$. Then, $u_k[1] = M(1)$, $u_k[2] = \min\{M(2), M(3)\}$, $u_k[3] = \min\{M(4), M'(5)\} = M(4)$ since $M'(5) = +\infty$, and $u_k[4] = M'(6) = +\infty$. The use of u_k is discussed in the sequel.

For convenience, let us recall Equation (3) defined in the beginning of this section:

$$M(k) = \begin{cases} +\infty & \text{if } k < L, \\ |\mathcal{R}[1,k]| & \text{if } L \le k < 2L, \\ \min\limits_{0 \le h \le r_k} \max\{|\mathcal{R}[j_{k,h+1}, k]|, \min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)\} & \text{if } k \ge 2L, \end{cases} \quad \text{(3 revisited)}$$

where $j_{k,0} = 0$, $j_{k,r_k+1} = k-L+1$, and $j_{k,1}, \ldots, j_{k,r_k}$ is the increasing sequence of all positions $j \in [1, k-L]$ in which $|\mathcal{R}[j,k]| \ne |\mathcal{R}[j+1,k]|$. In order to compute $M(k)$, one has to find the minima $\min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)$ and calculate $|\mathcal{R}[j_{k,h+1}, k]|$. As it follows from Lemma 5 and the definition of s_k, all positions $j \in [1, k-1]$ in which $|\mathcal{R}[j,k]| \ne |\mathcal{R}[j+1,k]|$ are represented by the numbers $s_k[i] - 1$ such that $1 < s_k[i] \le k$ (in the increasing order); hence, the sequence $j_{k,1}, \ldots, j_{k,r_k}$ corresponds to either $s_k[1] - 1, \ldots, s_k[r_k] - 1$ or $s_k[2] - 1, \ldots, s_k[r_k + 1] - 1$, depending on whether $s_k[1] \ne 1$. Then, the minima $\min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)$ are stored in the corresponding elements of u_k (assuming $s_k[0] = 1$): $u_k[i] = \min\{M'(\ell)\colon s_k[i-1]-1 \le \ell < s_k[i]-1\} = \min\{M(\ell)\colon s_k[i-1]-1 \le \ell < \min\{s_k[i]-1, k-L+1\}\} = \min\limits_{j_{k,h} \le j < j_{k,h+1}} M(j)$, provided $s_k[i-1] - 1 = j_{k,h}$. It is clear that $u_k[i] \ne +\infty$ only if the segment $[s_k[i-1] - 1, s_k[i] - 2]$ intersects the range $[1, k-L]$ and, thus, corresponds to a segment $[j_{k,h}, j_{k,h+1} - 1]$, for $h \in [0, r_k]$. Therefore, since $M'(\ell) = +\infty$ for $\ell < 1$ and $\ell > k - L$ and, thus, such values $M'(\ell)$ do not affect, in a sense, the minima stored in u_k, one can rewrite (3) as follows:

$$M(k) = \begin{cases} +\infty & \text{if } k < L, \\ |\mathcal{R}[1,k]| & \text{if } L \le k < 2L, \\ \min\limits_{1 \le j \le |u_k|} \max\{|\mathcal{R}[s_k[j] - 1, k]|, u_k[j]\} & \text{if } k \ge 2L. \end{cases} \quad (4)$$

It remains to compute the numbers $|\mathcal{R}[s_k[j] - 1, k]|$, for $j \in [1, |s_k|]$.

▶ **Lemma 7.** *Consider a set of recombinants $\mathcal{R} = \{R_1, \ldots, R_m\}$, each of which has length n. For $k \in [1, n]$ and $j \in [1, |s_k|]$, one has $|\mathcal{R}[s_k[j] - 1, k]| = t_k[j] + t_k[j+1] + \cdots + t_k[|t_k|]$.*

Proof. Denote $\ell = k - s_k[j] + 1$, so that $\mathcal{R}[s_k[j]-1, k] = \mathcal{R}[k-\ell, k]$. Suppose that $\ell = 0$. Note that $R_{a_k[1]}[k] \le \cdots \le R_{a_k[m]}[k]$. Since $d_k[i] = k + 1$ iff either $i = 1$ or $R_{a_k[i-1]}[k] \ne R_{a_k[i]}[k]$, it is easy to see that $|\mathcal{R}[k,k]|$, the number of distinct letters $R_i[k]$, is equal to the number of time $k + 1 = s_k[|s_k|]$ occurs in d_k, i.e., $t_k[|t_k|]$.

Suppose that $\ell > 0$. It suffices to show that $|\mathcal{R}[k-\ell, k]| - |\mathcal{R}[k-\ell+1, k]| = t_k[j]$. For $i \in [1, m]$, denote by R'_i the string $R_i[k]R_i[k-1]\cdots R_i[k-\ell]$. Fix $w \in \mathcal{R}[k-\ell+1, k]$. Since $R'_{a_k[1]} \le \cdots \le R'_{a_k[m]}$ lexicographically, there are numbers h and h' such that $R_{a_k[i]}[k-\ell+1, k] = w$ iff $i \in [h, h']$. Further, we have $R_{a_k[h]}[k-\ell] \le R_{a_k[h+1]}[k-\ell] \le \cdots \le R_{a_k[h']}[k-\ell]$.

Algorithm 1 Computing e_k, s_k, t_k, u_k, a_k from e_{k-1}, s_{k-1}, t_{k-1}, u_{k-1}, a_{k-1}.

1: copy s_{k-1} into s_k and add the element $k+1$ to the end of s_k, thus incrementing $|s_k|$;
2: copy u_{k-1} into u_k and add the element $M'(k-1)$ to the end of u_k, thus incrementing $|u_k|$;
3: zero initialize $C[0,|\Sigma|]$, $P[0,|\Sigma|-1]$, and $t_k[1,|s_k|]$;
4: **for** $i \leftarrow 1$ **to** m **do** $C[R_i[k]+1] \leftarrow C[R_i[k]+1]+1$;
5: **for** $i \leftarrow 1$ **to** $|\Sigma|-1$ **do** $C[i] \leftarrow C[i]+C[i-1]$;
6: **for** $i \leftarrow 1$ **to** m **do**
7: $\quad b \leftarrow R_{a_{k-1}[i]}[k]$;
8: $\quad C[b] \leftarrow C[b]+1$;
9: $\quad a_k[C[b]] \leftarrow a_{k-1}[i]$;
10: $\quad$ **if** $P[b]=0$ **then** $e_k[C[b]] \leftarrow |s_k|$;
11: $\quad$ **else** $e_k[C[b]] \leftarrow \max\{e_{k-1}[\ell] \colon P[b] < \ell \le i\}$
12: $\quad P[b] \leftarrow i$;
13: **for** $i \leftarrow 1$ **to** m **do** $t_k[e_k[i]] \leftarrow t_k[e_k[i]]+1$;
14: $j \leftarrow 1$;
15: add a new "dummy" element $+\infty$ to the end of u_k and u_{k-1};
16: **for** $i \leftarrow 1$ **to** $|s_k|$ **do**
17: $\quad u_k[j] \leftarrow \min\{u_k[j], u_{k-1}[i]\}$;
18: $\quad$ **if** $t_k[i] \neq 0$ **then**
19: $\quad\quad tmp[i] \leftarrow j$;
20: $\quad\quad s_k[j] \leftarrow s_k[i]$, $t_k[j] \leftarrow t_k[i]$, $u_k[j+1] \leftarrow u_{k-1}[i+1]$;
21: $\quad\quad$ **if** $s_k[j]-1 > k-L$ **and** $(j=1$ **or** $s_k[j-1]-1 \le k-L)$ **then** $u_k[j] \leftarrow \min\{u_k[j], M(k-L)\}$;
22: $\quad\quad j \leftarrow j+1$;
23: shrink s_k, t_k, and u_k to $j-1$ elements, so that $|s_k| = |t_k| = |u_k| = j-1$;
24: **for** $i \leftarrow 1$ **to** m **do** $e_k[i] \leftarrow tmp[e_k[i]]$;

Thus, by definition of d_k, for $i \in [h+1, h']$, we have $R_{a_k[i-1]}[k-\ell] \neq R_{a_k[i]}[k-\ell]$ iff $d_k[i] = k-\ell+1 = s_k[j]$. Note that $d_k[h] > s_k[j]$. Therefore, the number of strings $R_i[k-\ell,k]$ from $\mathcal{R}[k-\ell,k]$ having suffix w is equal to one plus the number of integers $s_k[j]$ in the range $d_k[h,h']$, which implies $|\mathcal{R}[k-\ell,k]| - |\mathcal{R}[k-\ell+1,k]| = t_k[j]$. ◀

By (4) and Lemma 7, one can calculate $M(k)$ in $O(m)$ time using the arrays t_k and u_k. It remains to describe how we maintain a_k, e_k, s_k, t_k, u_k.

▶ **Lemma 8.** *The arrays a_k, e_k, s_k, t_k, u_k can be computed from $a_{k-1}, e_{k-1}, s_{k-1}, t_{k-1}, u_{k-1}$ and from the numbers $M(k-L)$ and $M(k-1)$ in $O(m)$ time, assuming the input alphabet is $[0, |\Sigma|-1]$ with $|\Sigma| = O(m)$.*

Proof. Let us analyze Algorithm 1 that computes a_k, e_k, s_k, t_k, u_k. By definition, $d_{k-1}[i] = s_{k-1}[e_{k-1}[i]]$ for $i \in [1,m]$. The first line of the algorithm initializes s_k so that $d_{k-1}[i] = s_k[e_{k-1}[i]]$, for $i \in [1,m]$, and $s_k[|s_k|] = k+1$. Since after this initialization s_k, obviously, is in the sorted order, one has, for $i,j \in [1,m]$, $e_{k-1}[i] \le e_{k-1}[j]$ iff $d_{k-1}[i] \le d_{k-1}[j]$ and, therefore, for $\ell \in [i,j]$, one has $d_{k-1}[\ell] = \max\{d_{k-1}[\ell'] \colon i \le \ell' \le j\}$ iff $e_{k-1}[\ell] = \max\{e_{k-1}[\ell'] \colon i \le \ell' \le j\}$. Based on this observation, we fill e_k in lines 3–12 so that $d_k[i] = s_k[e_k[i]]$, for $i \in [1,m]$, using exactly the same algorithm as in Figure 2a, where d_k is computed, but instead of the assignment $d_k[C[b]] \leftarrow k+1$, we have $e_k[C[b]] \leftarrow |s_k|$ since $s_k[|s_k|] = k+1$. Here we also compute a_k in the same way as in Figure 2a.

The loop in line 13 fills t_k so that, for $i \in [1,|s_k|]$, $t_k[i]$ is the number of occurrences of the integer i in e_k (t_k was zero initialized in line 3). Since, for $i \in [1,m]$, we have $d_k[i] = s_k[e_k[i]]$ at this point, $t_k[i]$ is also the number of occurrences of the integer $s_k[i]$ in $d_k[1,m]$.

By definition, s_k must contain only elements from d_k, but this is not necessarily the case in line 14. In order to fix s_k and t_k, we simply have to remove all elements $s_k[i]$ for which $t_k[i] = 0$, moving all remaining elements of s_k and non-zero elements of t_k to the left accordingly. Suppose that, for some h and i, we have $e_k[h] = i$ and the number $s_k[i]$ is moved to $s_k[j]$, for some $j < i$, as we fix s_k. Then, $e_k[h]$ must become j. We utilize an additional temporary array $tmp[1, |s_k|]$ to fix e_k. The loop in lines 16–22 fixes s_k and t_k in an obvious way; once $s_k[i]$ is moved to $s_k[j]$ during this process, we assign $tmp[i] = j$. Then, s_k, t_k, u_k (u_k is discussed below) are resized in line 23, and the loop in line 24 fixes e_k using tmp.

Recall that $[s_k[j-1]-1, s_k[j]-2]$, for $j \in [1, |s_k|]$, is a system of disjoint segments covering $[0, k-1]$ (assuming $s_k[0] = 1$). It is now easy to see that this system is obtained from the system $[s_{k-1}[j-1]-1, s_{k-1}[j]-2]$, with $j \in [1, |s_{k-1}|]$ (assuming $s_{k-1}[0] = 1$), by adding the new segment $[k-1, k-1]$ and joining some segments together. The second line of the algorithm copies u_{k-1} into u_k and adds $M'(k-1)$ to the end of u_k, so that, for $j \in [1, |u_{k-1}|]$, $u_k[j]$ is equal to the minimum of $M'(\ell)$ for all ℓ from the segment $[s_{k-1}[j-1]-1, s_{k-1}[j]-2]$ and $u_k[|u_{k-1}|+1] = M'(k-1)$ is the minimum in the segment $[k-1, k-1]$. (This is not completely correct since M' has changed as k was increased; namely, $M'(k-L)$ was equal to $+\infty$ but now is equal to $M(k-L)$.) As we join segments removing some elements from s_k in the loop 16–22, the array u_k must be fixed accordingly: if $[s_k[j-1]-1, s_k[j]-2]$ is obtained by joining $[s_{k-1}[h-1]-1, s_{k-1}[h]-2]$, for $j' \le h \le j''$, then $u_k[j] = \min\{u_{k-1}[h] \colon j' \le h \le j''\}$. We perform such fixes in line 17, accumulating the latter minimum. We start accumulating a new minimum in line 20, assigning $u_k[j+1] \leftarrow u_{k-1}[i+1]$. If at this point the ready minimum accumulated in $u_k[j]$ corresponds to a segment containing the position $k-L$, we have to fix u_k taking into account the new value $M'(k-L) = M(k-L)$; we do this in line 21. To avoid accessing out of range elements in u_k and u_{k-1} in line 20, we add a "dummy" element in, respectively, u_k and u_{k-1} in line 15. ◀

Besides all the arrays of length m, the algorithm from Lemma 8 also requires access to $M(k-L)$ and, possibly, to $M(k-1)$. During the computation of $M(k)$ for $k \in [1, n]$, we maintain the last L calculated numbers $M(k-1), M(k-2), \ldots, M(k-L)$ in a circular array, so that the overall required space is $O(m+L)$; when k is incremented, the array is modified in $O(1)$ time in an obvious way. Thus, Lemma 8 implies Theorem 1

If, as in our case, one does not need s_k, t_k, u_k for all k, the arrays s_k, t_k, u_k can be modified in-place, i.e., s_k, t_k, u_k can be considered as aliases for s_{k-1}, t_{k-1}, u_{k-1}, and yet the algorithm remains correct. Thus, we really need only 7 arrays in total: a_k, a_{k-1}, e_k, e_{k-1}, s, t, u, where s, t, u serve as s_k, t_k, u_k and the array tmp can be organized in place of a_{k-1} or e_{k-1}. It is easy to maintain along with each value $u_k[j]$ a corresponding position ℓ such that $u_k[j] = M'(\ell)$; these positions can be used then to restore the found segmentation of $\mathcal{R}$ using backtracking (see the beginning of the section). To compute e_k, instead of using an RMQ data structure, one can adapt in an obvious way the algorithm from Figure 2b rewriting the arrays a_{k-1} and e_{k-1} during the computation, which is faster in practice but theoretically takes $O(m \log \sigma)$ time by Lemma 4. We do not discuss further details as they are straightforward.

4 Implementation

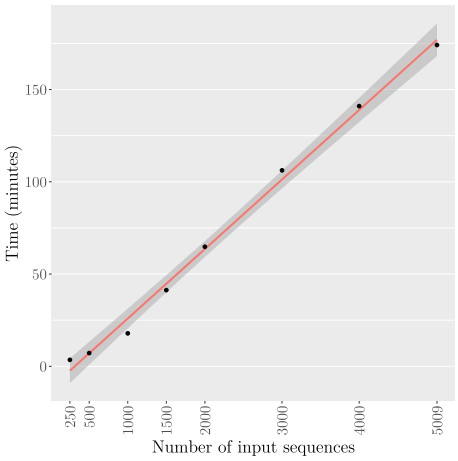
We implemented the segmentation algorithm using Lemma 4 to build the data structures, and thus the overall time complexity of the implemented approach is $O(mn \log \sigma)$. The implementation outputs for each segment the distinct founder sequence fragments, and associates to each fragment the set of haplotypes containing that fragment as a substring

at that location (these are easily deduced given the segmentation and the positional BWT structures). As discussed earlier, to obtain full founder sequences, one needs to concatenate these fragments in some order. If each segment has exactly the same amount of fragments, there is a way of finding an optimal concatenation minimizing the number of crossovers [21]: For any two consecutive segments, form a bipartite graph with fragments as nodes and edges corresponding to plausible concatenations. The cost of an edge is the symmetric difference of the sets associated with the fragments. Minimum cost perfect matching then gives an optimal concatenation order. Each consecutive pair can be solved independently, and founder sequences can be extracted through the paths formed by matches edges. However, not all segments may contain the maximum amount of distinct fragments; we conjecture the ordering problem becomes hard in this case (due to the local matchings being no longer independent). There are many possible heuristic ways to alleviate this issue, among which we opted to preprocess the segments to have the same amount of fragments; we duplicated the fragments greedily according to the sizes of the associated sets. Then we used the matching approach described above. In the upcoming extended version of this paper, we plan to study an alternative approach that provides an approximation guarantee.

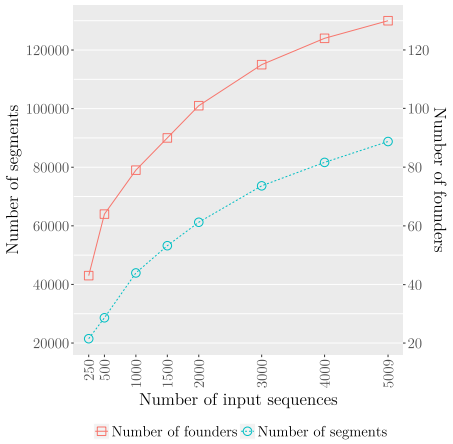
To test the implementation of our algorithm, we generated a multiple alignment of haplotype sequences from chromosome 6 variants from the phase 3 data of the 1000 Genomes Project [19]. This resulted in 5009 haplotype sequences of equal length (including the reference sequence) of approximately 171 million characters. We then discarded columns of identical characters, which reduced each sequence to approximately 5.38 million characters. We used an increasing number of these sequences as an input to our tool to verify its usability. The tests were run on a Ubuntu Linux 16.04 server. The server had 96 Intel Xeon E7-4830 v3 CPUs running at 2.10GHz and 1.4 TB of memory. Results on varying input sizes are shown in Fig. 3a. From this experiment it is conceivable that processing of thousands of complete human genomes takes only few CPU days. Figure 3b plots the number of founders against the number of segments. These look very promising, as using 130 founders instead of 5009 haplotypes as the input to pan-genome indexing of [22] will result into significant saving of resources; this solves the space bottleneck, and the preprocessing of founder reconstruction also saves time in the heavy indexing steps. The average segment length of some 1926 bases (171 million divided by 88778 segments) is also likely to be high enough not to break too badly the contiguity required for successful read alignment. To better see the contiguity of the resulting founder set, we mapped the haplotypes to the founders minimizing the number of jumps (simple greedy algorithm is optimal [15]). The result is shown in Fig. 3c. Since identical columns do not cause any jumps, we can now divide 171 million with 18274 jumps. This gives average distance between two jumps being 9358 bases. The heuristic part of optimizing the concatenation of founder blocks yields almost 5-fold improvement in the contiguity, compared to the worst case of each segment boundary yielding a discontinuity for each input sequence (which should be close to what a random concatenation order would produce).

Our intention was to compare our tool to an implementation of Ukkonen's algorithm [15]. However, initial testing with four input sequences showed that the latter implementation is not practical with a data set of this size.

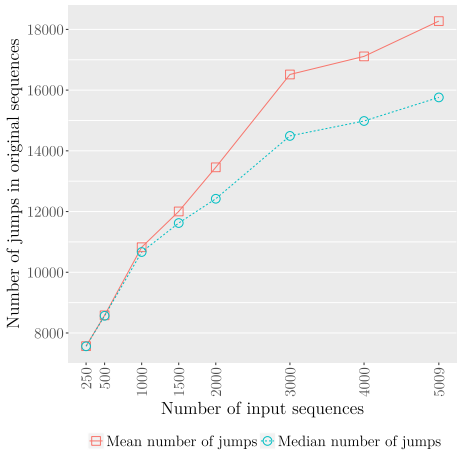
Our implementation is open source and available at the URL `https://github.com/tsnorri/founder-sequences`.



(a) The running time of our implementation plotted against the number of input sequences with $L = 10$. The data points have been fitted with a least-squares linear model, and the grey band shows the 95% confidence interval.



(b) The founder and segment counts as produced by our implementation plotted against the number of input sequences with $L = 10$.



(c) The average and median numbers of jumps, i.e. counts of positions where one needs to change from one founder sequence to another to read an original sequence, plotted against the number of input sequences with $L = 10$.

Figure 3 Evaluation of founder reconstruction on a pan-genomic setting.

5 Discussion

With 5009 haplotypes reducing down to 130 founders with the average distance of two crossovers being 9358 bases, one can expect short read alignment and variant calling to become practical on such pan-genomic setting. We are investigating this on our tool *PanVC* [22], where one can simply replace its input multiple alignment with the one made of the founder sequences. With graph-based approaches, slightly more effort is required: Input variations are encoded with respect to the reference, so one first needs to convert variants into a multiple alignment, apply the founder reconstruction algorithm, and finally convert

the multiple alignment of founder sequences into a directed acyclic graph. PanVC toolbox provides the required conversions. Alternatively, one can construct the pan-genome graph using other methods, and map the founder sequences afterwards to the paths of the graph: If original haplotype sequences are already spelled as paths, each founder sequence is a concatenation of existing subpaths, and can hence be mapped to a continuous path without alignment (possibly requiring adding a few missing edges).

Finally, it will be interesting to see how much the contiguity of the founder sequences can still be improved with different strategies for the concatenation order and with different formulations of the segmentation problem. For the former, we are investigating an approach with approximation guarantee. For the latter, we consider a variant with the number of founder sequenced fixed.

References

1 Computational Pan-Genomics Consortium et al. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, page bbw089, 2016.

2 Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47:682–688, 2015.

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

4 Exome Aggregation Consortium. Analysis of protein-coding genetic variation in 60,706 humans. *Nature*, 536(7616):285–291, 2016.

5 Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J. Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A*, 372, 2014.

6 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM 2006*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

7 Travis Gagie and Simon J. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3(12), 2015.

8 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Michael F Lin, Benedict Paten, and Richard Durbin. Sequence variation aware genome references and read mapping with the variation graph toolkit. *bioRxiv*, 2017. doi:10.1101/234856.

9 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.

10 Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 222–233. Springer, 2016.

11 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.

12 Veli Mäkinen and Tuukka Norri. Applying the positional Burrows–Wheeler transform to all-pairs hamming distance. *Submitted manuscript*, 2018.

13 Tom O Mokveld, Jasper Linthorst, Zaid Al-Ars, and Marcel Reinders. Chop: Haplotype-aware path indexing in population graphs. *bioRxiv*, 2018. doi:10.1101/305268.

14 Gonzalo Navarro. Indexing highly repetitive collections. In *Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, LNCS 7643, pages 274–279, 2012.

15 Pasi Rastas and Esko Ukkonen. Haplotype inference via hierarchical genotype parsing. In *Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings*, pages 85–97, 2007.

16 Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10:R98, 2009.

17 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *arXiv preprint arXiv:1805.03834*, 2018.

18 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

19 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, sep 2015.

20 The UK10K Consortium. The UK10K project identifies rare variants in health and disease. *Nature*, 526(7571):82–90, 2015.

21 Esko Ukkonen. Finding founder sequences from a set of recombinants. In *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, pages 277–286, 2002.

22 Daniel Valenzuela, Tuukka Norri, Välimäki Niko, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC Genomics*, 19(Suppl 2):87, 2018. doi:10.1186/s12864-018-4465-8.

23 Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. Rcsi: Scalable similarity search in thousand(s) of genomes. *PVLDB*, 6(13):1534–1545, 2013.

A Proof of Lemma 4

Proof. Fix $i \in [1, m]$. The ith iteration of the last loop in the algorithm computes the maximum in a range $d'_{k-1}[i', i]$, where d'_{k-1} is the original array d_{k-1} before modifications and $i' = P[b] + 1$ for some b and P. Let $\ell_i = i - i'$. Denote $\tilde{\ell} = \frac{1}{m}\sum_{i=1}^{m} \ell_i$, the "average query length". We are to prove that the running time of the algorithm is $O(m \log \tilde{\ell})$, which implies the result since $m\tilde{\ell} = \sum_{i=1}^{m} \ell_i$ and, obviously, $\sum_{i=1}^{m} \ell_i \le \sigma m$.

We say that a position j is *touched* if the function maxd is called with its first argument equal to j. Clearly, it suffices to prove that the total number of touches is $O(m \log \tilde{\ell})$. While processing the query maxd$(i-\ell_i, i)$, we may have touched many positions. Denote the sequence of all such position, for the given i, by $i_1, \ldots, i_r$; in other words, at the time of the query maxd$(i-\ell_i, i)$, we have $i_1 = i - \ell_i$, $i_j = a_{k-1}[i_{j-1}]$ for $j \in [2, r]$, and $i_r = i$. Obviously, $i_1 < \cdots < i_r$. We say that, for $j \in [1, r-1]$, the touch of i_j in the query maxd$(i-\ell_i, i)$ is *scaling* if there exists an integer r such that $i - i_j > 2^r$ and $i - i_{j+1} \le 2^r$ (see Figure 4). We count separately the total number of scaling and non-scaling touches in all i.

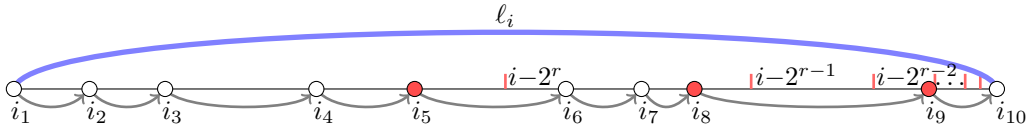


Figure 4 RMQ query on a range $[i - \ell_i, i]$; scaling touches are red.

For position j, denote by $p(j)$ the number of non-scaling touches of j. We are to prove that $P = \sum_{j=1}^{m} p(j) \le 2m \log \tilde{\ell}$. Let $q_h(j)$ denote the value of $a_{k-1}[j] - j$ in the hth non-scaling touch of j, for $h \in [1, p(j)]$. Suppose that this hth touch happens during the processing of a query maxd$(i-\ell_i, i)$. By the definition, $j + q_h(j)$ follows j in the sequence of touched positions. Since the touch of j is non-scaling, we have $i - j > j + q_h(j) > 2^r$, where r is the largest integer such that $i - j > 2^r$, and hence, $q_h(j) < 2^r$. Since maxd$(i - \ell_i, i)$ assigns $a_{k-1}[j] \leftarrow i + 1$, we have $a_{k-1}[j] - j > i - j > 2^r$ after the query. In other words, we had $a_{k-1}[j] - j = q_h(j) < 2^r$ before the query and have $a_{k-1}[j] - j > 2^r$ after. This immediately implies that $q_h(j) \ge 2^{h-1}$, for $h \in [1, p(j)]$, and, therefore, every position can be touched in the non-scaling way at most $O(\log m)$ times, implying $P = O(m \log m)$. But we can deduce a stronger bound. Since the sum of all values $j - a_{k-1}[j]$ for all positions j touched in a query maxd$(i - \ell_i, i)$ is equal to ℓ_i, it is obvious that $\sum_{j=1}^{m} \sum_{h=1}^{p(j)} q_h(j) \le \sum_{i=1}^{m} \ell_i = m\tilde{\ell}$. On the other hand, we have $\sum_{j=1}^{m} \sum_{h=1}^{p(j)} q_h(j) \ge \sum_{j=1}^{m} \sum_{h=1}^{p(j)} 2^{h-1} = \sum_{j=1}^{m} 2^{p(j)} - m$. The well-known property of the convexity of the exponent is that the sum $\sum_{j=1}^{m} 2^{p(j)}$ is minimized whenever all $p(j)$ are equal and maximal, i.e., $\sum_{j=1}^{m} 2^{p(j)} \ge \sum_{j=1}^{m} 2^{P/m}$. Hence, once $P > 2m \log \tilde{\ell}$, we obtain $\sum_{j=1}^{m} \sum_{h=1}^{p(j)} q_h(j) \ge \sum_{j=1}^{m} 2^{P/m} - m > m\tilde{\ell}^2 - m$, which is larger than $m\tilde{\ell}$ for $\tilde{\ell} \ge 2$ (the case $\tilde{\ell} < 2$ is trivial), contradicting $\sum_{j=1}^{m} \sum_{h=1}^{p(j)} q_h(j) \le m\tilde{\ell}$. Thus, $P = \sum_{j=1}^{m} p(j) \le 2m \log \tilde{\ell}$.

It remains to consider scaling touches. The definition implies that each query maxd$(i-\ell_i, i)$ performs at most $\log \ell_i$ scaling touches. Thus, it suffices to upperbound $\sum_{i=1}^{m} \log \ell_i$. Since the function log is concave, the sum $\sum_{i=1}^{m} \log \ell_i$ is maximized whenever all ℓ_i are equal and maximal, i.e., $\sum_{i=1}^{m} \log \ell_i \le \sum_{i=1}^{m} \log(\frac{1}{m}\sum_{j=1}^{m} \ell_j) = m \log \tilde{\ell}$, hence the result follows. ◀