Linear Time Maximum Segmentation Problems in Column Stream Model*

Bastien Cazaux[1][0000−0002−1761−4354], Dmitry Kosolobov[2][0000−0002−2909−2952], Veli Mäkinen[1][0000−0003−4454−1493], and Tuukka Norri[1][0000−0002−8276−0585]

[1] Department of Computer Science, University of Helsinki, Helsinki, Finland
{bastien.cazaux,veli.makinen,tuukka.norri}@helsinki.fi
[2] Ural Federal University, Ekaterinburg, Russia dkosolobov@mail.ru

Abstract. We study a lossy compression scheme linked to the biological problem of founder reconstruction: The goal in founder reconstruction is to replace a set of strings with a smaller set of founders such that the original connections are maintained as well as possible. A general formulation of this problem is NP-hard, but when limiting to reconstructions that form a segmentation of the input strings, polynomial time solutions exist. We proposed in our earlier work (WABI 2018) a linear time solution to a formulation where minimum segment length was bounded, but it was left open if the same running time can be obtained when the targeted compression level (number of founders) is bounded and lossyness is minimized. This optimization is captured by the Maximum Segmentation problem: Given a threshold M and a set $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$ of strings of the same length n, find a minimum cost partition P where for each segment $[i, j] \in P$, the *compression level* $|\{\mathcal{R}_k[i, j] : 1 \leq k \leq m\}|$ is bounded from above by M. We give linear time algorithms to solve the problem for two different (compression quality) measures on P: the average length of the intervals of the partition and the length of the minimal interval of the partition. These algorithms make use of positional Burrows–Wheeler transform and the range maximum queue, an extension of range maximum queries to the case where the input string can be operated as a queue. For the latter, we present a new solution that may be of independent interest. The solutions work in a streaming model where one column of the input strings is introduced at a time.

Keywords: Pan-genome indexing, founder reconstruction, dynamic programming, positional Burrows–Wheeler transform, range maximum queue

1 Introduction

Given a *set of recombinants* $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$, i.e. a set of strings of the same length, a *set of founders* is a set of strings of the same length where all the recombinants can be mapped on the founders for the common positions, i.e. for each position k all the characters at the position k of the recombinants are included in all the characters at the position k of the founders.

* This work was partially supported by the Academy of Finland (grant 309048)

Minimizing the number of *crossovers*, i.e. positions where recombinants need to change between mapped founders, corresponds to the problem *founder sequence reconstruction* [9,8,2]. As this problem is NP-hard [8,2], Ukkonen suggested taking a polynomial variant of this problem through construction of a *segmentation* [9]. Here a segmentation is a decomposition of the set of recombinants into blocks. For a partition P, the corresponding segmentation corresponds to a set of *segments* where for $[i, j] \in P$, the segment of the interval $[i, j]$ is the set of strings $\{\mathcal{R}_k[i, j] : 1 \leq k \leq m\}$.

Ukkonen proposed three different measures for segmentations of recombinants: λ_{min} (the minimum size of the intervals), λ_{ave} (the ratio between the length of a string of $\mathcal{R}$ and the number of segments) and λ_{max} (the maximum of segment sizes) and gave an optimal solution in $O(mn^2)$ time where λ_{min} is bounded by a given user-defined value M and λ_{max} is minimized (*Minimum Segmentation problem*) and an optimal solution in $O(mn)$ time where λ_{ave} is bounded and λ_{max} is minimized [9]. Norri et al. improved the first result to $O(mn)$ time by using the *positional Burrows–Wheeler transform* (pBWT) [3]. This problem corresponds to finding the segmentation that minimizes the maximal segment size where each interval of the corresponding partition have length bigger than a threshold K.

Ukkonen [9] proved that we can find in $O(n(m + M^3))$ time a set of founders which minimizes the number of crossovers in the case where all the segments of the segmentation have the same size M. Norri et al. [7] apply the algorithm of Ukkonen to the case where the segments have different sizes; experimental results show that this approach works well in practice, although optimality is not guaranteed in this case.

Instead of minimizing λ_{max} where λ_{min} is bounded as in the Minimum Segmentation problem, in this paper we study the problem where λ_{max} is bounded and we want to maximize λ_{min} or λ_{ave}. In other words, we take the dual problem of Minimum Segmentation problem where we bound the maximum size of segments, i.e. the number of founders, and we want to optimize the partition corresponding to this segmentation (either maximize the size of the minimal interval of the partition or minimize the number of intervals). This formulation is motivated by the ability to control the size of the *pan-genome index* proposed in [10]: Multiple alignment of thousands of human genomes can be replaced with a multiple alignment of founders. We demonstrated in [7] that reduction from 5009 sequences to 130 founders gives average distance 9358 bases between two crossovers. Such preservation of continuity is sufficient for the approach in [10]. With our new formulation, we can directly control the target number of founders without needing to try out different bounds for the segment length. In more general terms, the approach can be seen as a lossy compression scheme, where the targeted compression level (number of founders) is fixed and the *compression quality* (preservation of continuities) is optimized. Such general formulation might find other applications beyond genome research.

We propose two different algorithms to solve maximization of λ_{min} and λ_{ave}, respectively. In Section 3, we give a greedy algorithm which finds an optimal solution in $O(mn)$ for the first problem and in Section 4, we present a dynamic

programming algorithm using a *Range Maximum Queue* to solve the second problem in linear time. The Range Maximum Queue is an extension of *min-queue* [5]. A min-queue supports minimum value queries on queue in constant time, with the update operations taking also constant time. Our structure supports querying maximum value over a range of items in the queue in constant time, with the update operations taking amortized constant time.

We consider all of the Maximum Segmentation problems in a specific streaming data model. As every input of the Maximum Segmentation problems consists of a bound and a set of strings of the same length, we define the *Column Stream Model* such that we are given the bound and a stream that yields the set of strings of the same length, column-by-column. In essence, at the k^{th} step we have the k^{th} character of each input string. A justification of this model can be found in Appendix.

2 Preliminaries

In this section we present the problems of Maximum Segmentation and some terminologies we will need.

To begin we define some notations for strings and sets. Given a *string* $w = a_1 \dots a_n$, the *length* of w, denoted by $|w|$, is n, the i^{th} element of w, denoted by $w[i]$, is a_i and the *substring* denoted by $w[i, j]$ is $a_i \dots a_j$. We use an analogous notation for the arrays. Given two integers i and j with $i \leq j$, we denote by $[i, j]$ the set of integers between i and j, i.e. $\{k \in \mathbb{Z} : i \leq k \leq j\}$. Given a finite set S, a *partition* $P = \{S_1, \dots, S_k\}$ is a set of subsets of S such that $\cup_{S_i \in P} S_i = S$ and $i \neq j \Rightarrow S_i \cap S_j = \emptyset$. The *cardinality* of S is denoted by $|S|$.

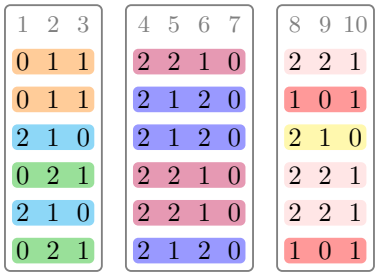
The input of our problems is a set of *recombinants* $\mathcal{R} = \{\mathcal{R}_1, \dots, \mathcal{R}_m\}$ which is a set of m strings of the same length n ($|\mathcal{R}_1| = \dots = |\mathcal{R}_m| = n$). In what follows, we use m as the number of strings of $\mathcal{R}$ and n as the length of each string of $\mathcal{R}$.

For an integer interval $[i, j]$ with $1 \leq i \leq j \leq n$, we denote by $\mathcal{R}[i, j]$ the set of all the substrings $\mathcal{R}_k[i, j]$ with $k \in [1, m]$, i.e. $\mathcal{R}[i, j] = \{\mathcal{R}_k[i, j] : k \in [1, m]\}$. Given a partition P of $[1, n]$, we define the following three measures: $\lambda_{min}(\mathcal{R}, P) = \min_{[i,j] \in P} |[i, j]|$, $\lambda_{ave}(\mathcal{R}, P) = \frac{n}{|P|}$ and $\lambda_{max}(\mathcal{R}, P) = \max_{[i,j] \in P} |\mathcal{R}[i, j]|$. When there is no confusion, we just use the notations λ_{min}, λ_{ave} and λ_{max}.

Example 1. The set $\mathcal{R}$ of 6 recombinants of size 10:

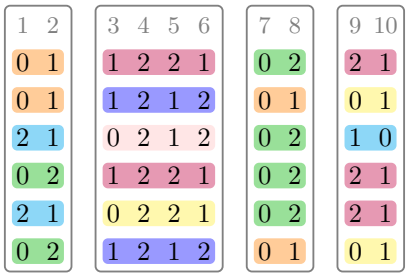
1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	1	0	2	2	1
0	1	1	2	1	2	0	1	0	1
2	1	0	2	1	2	0	2	1	0
0	2	1	2	2	1	0	2	2	1
2	1	0	2	2	1	0	2	2	1
0	2	1	2	1	2	0	1	0	1

By taking the partition $P_1 = \{[1,3],[4,7],[8,10]\}$,



one has $\lambda_{min}(\mathcal{R}, P_1) = \min\{|[1,3]|, |[4,7]|, |[8,10]|\} = \min\{3,4,3\} = 3$, $\lambda_{ave}(\mathcal{R}, P_1) = \frac{10}{3}$ and $\lambda_{max}(\mathcal{R}, P_1) = \max\{|\mathcal{R}[1,3]|, |\mathcal{R}[4,7]|, |\mathcal{R}[8,10]|\} = \max\{3,2,3\} = 3$.

By taking the partition $P_2 = \{[1,2],[3,6],[7,8],[9,10]\}$,



one has $\lambda_{min}(\mathcal{R}, P_2) = \min\{|[1,2]|, |[3,6]|, |[7,8]|, |[9,10]|\} = \min\{2,4,2,2\} = 2$, $\lambda_{ave}(\mathcal{R}, P_2) = \frac{10}{4} = 2.5$ and $\lambda_{max}(\mathcal{R}, P_2) = \max\{|\mathcal{R}[1,2]|, |\mathcal{R}[3,6]|, |\mathcal{R}[7,8]|, |\mathcal{R}[9,10]|\} = \max\{3,4,2,2\} = 4$.

Definition 1. *The problem of* λ_{min}-Maximum Segmentation Partition *(or* λ_{min}-`MSP`*) is, given a bound* M *and a set of recombinants* $\mathcal{R}$*, to find a partition of* $[1,n]$ *which maximizes* $\lambda_{min}(\mathcal{R}, P)$ *subject to* $\lambda_{max}(\mathcal{R}, P) \leq M$*. The problem of* λ_{min}-Maximum Segmentation Length *(or* λ_{min}-`MSL`*) is, given a bound* M *and a set of recombinants* $\mathcal{R}$*, to find the measure* λ_{min} *for an optimal partition* P *of* λ_{min}-`MSP`*.*

We denote analogously by λ_{ave}-*Maximum Segmentation Partition* (or λ_{ave}-`MSP`) and λ_{ave}-*Maximum Segmentation Length* (or λ_{ave}-`MSL`) the similar problems in which $\lambda_{min}(\mathcal{R}, P)$ is substituted with $\lambda_{ave}(\mathcal{R}, P)$.

Hereafter we assume that M is at least $\max\{|\mathcal{R}[k,k]| : k \in [1,n]\}$, otherwise all the Maximum Segmentation problems admit no solution.

Remark 1. The notation of $\lambda_{min}(\mathcal{R}, P)$ and $\lambda_{ave}(\mathcal{R}, P)$ corresponds to the "λ_{min}" and "λ_{ave}" of [9]. The Maximum Segmentation Length problem corresponds to the segmented version of *Maximum fragment length* of [9]. The *Minimum Segmentation problem* of [7] corresponds to finding the measure λ_{max} of a partition P which minimizes λ_{max} where λ_{min} is bounded from below by an integer in input.

In the two following sections we present different algorithms to find in linear time (in the size of the input) an optimal solution of λ_{ave}-MSP and λ_{ave}-MSL (see Section 3) and λ_{min}-MSP and λ_{min}-MSL (see Section 4).

3 λ_{ave}-Maximum Segmentation problems

As $\lambda_{ave} = \frac{n}{|P|}$, maximizing λ_{ave} corresponds to minimizing $|P|$, i.e. the number of intervals of P. The idea of our algorithm solving the λ_{ave}-Maximum Segmentation problems is to use a greedy algorithm from left to right depending on values given by a special data structure, the *positional Burrows–Wheeler Transform*. We begin by explaining which values we want to compute, what is the positional Burrows–Wheeler Transform and how we can use it to build our values and finally we give a proof of the correctness for our greedy algorithm.

3.1 Optimal solution of λ_{ave}-MSP

Lemma 1. *Let i, j, i' and j' be four integers such that $i \leq i' \leq j' \leq j$. We have $|\mathcal{R}[i', j']| \leq |\mathcal{R}[i, j]|$.*

Proof. The property is due to the fact that each string of $\mathcal{R}[i', j']$ is a substring of $\mathcal{R}[i, j]$ on the same interval. □

Hence, for a fixed j, the function $|\mathcal{R}[i, j]|$ is decreasing in i. For a bound M and an integer k, we define c_k as the value such that $|\mathcal{R}[c_k, k]| > M$ and $|\mathcal{R}[c_k + 1, k]| \leq M$; in the case $|\mathcal{R}[1, k]| \leq M$, we take $c_k = 0$.

Remark 2. We can equivalently define c_k as follows:

$$\begin{aligned} c_k &= \min\{j \in [1, k] : |\mathcal{R}[j, k]| \leq M\} - 1 \\ &= \max\{j \in [1, k] : |\mathcal{R}[j, k]| > M\}. \end{aligned}$$

With all the values of c_k for all $k \in [1, n]$, we can build an optimal solution of λ_{ave}-MSP.

Lemma 2. *The solution $P = \{[1, b_p], \ldots, [b_3 + 1, b_2], [b_2 + 1, b_1]\}$ is an optimal solution of the λ_{ave}-Maximum Segmentation Partition problem where $b_1 = n$, $b_{k+1} = c_{b_k}$ for $k \geq 1$, and $c_{b_p} = 0$.*

Proof. The set $P = \{[1, b_p], \ldots, [b_3 + 1, b_2], [b_2 + 1, b_1]\}$ is a partition of $[1, n]$ because $b_1 = n$ and $b_{k+1} = c_{b_k}$. We are to prove that P is an optimal solution. Let $P_{opt} = \{[1, o_{p'}], \ldots, [o_2 + 1, o_1]$ be an optimal partition with $o_1 = n$. We denote $P_1 = P_{opt}$ and $P_k = P_{k-1} \setminus ([o_{k+1} + 1, o_k] \cup [o_k + 1, b_{k-1}]) \bigcup ([o_{k+1} + 1, b_k] \cup [b_k + 1, b_{k-1}])$. We are going to prove by induction on k that each P_k is optimal (for $k \in [1, p']$). The base of induction $k = 1$ holds by definition ($P_1 = P_{opt}$). We assume that P_{k-1} is an optimal solution. As $b_k = c_{b_{k-1}}$, we have $o_k \geq b_k$.

If $b_k \geq o_{k+1}+1$, by Lemma 1, as $o_{k+1}+1 \leq b_k \leq o_k$, we have $|\mathcal{R}[o_{k+1}+1, b_k]| \leq |\mathcal{R}[o_{k+1} + 1, o_k]| \leq M$ and thus P_k is a solution and $|P_k| = |P_{k-1}|$. Hence by induction, P_k is an optimal solution.

If $b_k < o_{k+1} + 1$, $P_k^\star = P_{k-1} \setminus ([o_{k+2} + 1, o_{k+1}] \cup [o_{k+1} + 1, o_k] \cup [o_k + 1, b_{k-1}]) \bigcup ([o_{k+2} + 1, b_k] \cup [b_k + 1, b_{k-1}])$ is a solution and $|P_k^\star| = |P_{k-1}| - 1$ and thus P_{k-1} is not optimal which is impossible by induction.

As $P_{p'}$ is optimal and $P_{p'} = P$, the partition P is an optimal solution. □

3.2 pBWT and linear algorithm for λ_{ave}-MSP

Given a string $T[1, m] = t_1 \dots t_m$, we denote by $\overleftarrow{T}$ the string corresponding to the reverse of T, i.e. $\overleftarrow{T} = t_m \dots t_1$. The *positional Burrows–Wheeler Transform* [3] (or pBWT) of a set of recombinants $\mathcal{R}$ is two sets of n arrays of size m – an array a_k and an array d_k for all $k \in [1, n]$ – where for $k \in [1, n]$, $a_k[1, m]$ is a permutation of $[1, m]$ such that $\overleftarrow{\mathcal{R}_{a_k[1]}[1, k]} \le \dots \le \overleftarrow{\mathcal{R}_{a_k[m]}[1, k]}$ lexicographically and $d_k[i] = 1 + \max\{j \in [1, k] : \mathcal{R}_{a_k[i]}[j] \neq \mathcal{R}_{a_k[i-1]}[j]\}$, for $i \in [2, m]$ and $d_k[1] = k + 1$.

Durbin [3] showed that we can compute recursively a_k and d_k from a_{k-1} and d_{k-1} in $O(m)$ time for a binary alphabet and Mäkinen and Norri [6] further generalized the construction for integer alphabets of size $O(m)$.

Lemma 3 ([6]). *The arrays a_k and d_k can be computed from a_{k-1} and d_{k-1} in $O(m)$ time, assuming the input alphabet is $[0, |\Sigma| - 1]$ with $|\Sigma| = O(m)$.*

Norri et al. [7] use three different arrays s_k, t_k, e_k to store the array d_k in increasing sorted order where s_k contains all distinct elements from d_k in the increasing sorted order (so that the length of s_k might be less than m), e_k is the normalized array d_k where $s_k[e_k[j]] = d_k[j]$ for all $j \in [1, m]$ and t_k is an array of the same length as s_k such that, for any j, $t_k[j]$ indicates the number of times the value $s_k[j]$ occurs in d_k.

Lemma 4 ([7]). *The arrays a_k, s_k, e_k and t_k can be computed from a_{k-1}, s_{k-1}, e_{k-1} and t_{k-1} in $O(m)$ time, assuming the input alphabet is $[0, |\Sigma| - 1]$ with $|\Sigma| = O(m)$.*

With s_k, e_k and t_k we can redefine c_k. As $|\mathcal{R}[s_k[j] - 1, k]| = \sum_{i \in [j, |t_k|]} t_k[i]$, one has

$$c_k = \max\{j \in [1, |s_k|] : \sum_{i \in [j, |t_k|]} t_k[i] > M\}. \tag{1}$$

With this new definition of c_k we obtain the following theorem.

Theorem 1. *Given a bound M and a set of recombinants $\mathcal{R}$, there is an algorithm that computes an optimal solution of the λ_{ave}-Maximum Segmentation Partition problem in a streaming fashion in $O(mn)$ time and $O(m + n)$ space.*

Proof. By Lemma 4 and Equation 1, we can build c_k in $O(m)$ time for each value $k \in [1, n]$. Lemma 2 gives us an optimal solution by using the values c_k. Finally we build and store all the values c_k and make a backtracking from n to 0. □

3.3 Right greedy and linear algorithm for λ_{ave}-MSL

Lemma 2 gives us a greedy solution from right to left, a "Left greedy" version. Here we present a "Right greedy" version working from left to right.

Lemma 5. *The solution* $P = \{[b_1, b_2 - 1], [b_2, b_3 - 1], \ldots, [b_p, n]\}$ *is an optimal solution of the* λ_{ave}-MSP *problem where* $b_1 = 1$ *and* $b_{k+1} = \min\{j \in [b_k, n] : c_j \geq b_k\}$.

Proof. By Lemma 1, we know that for all $k \in [1, n]$ and for all $j \in [k, n]$, $c_k \leq c_j$. Hence, we can adapt the proof of Lemma 2 by extending the optimal solution of the right to prove this result. □

By using the solution of Lemma 5 instead of Lemma 2, we do not need to store all the array of c_k to build the solutions of λ_{ave}-MSL and of λ_{ave}-MSP and we can extend the result of Theorem 1.

Theorem 2. *Given a bound* M *and a set of recombinants* $\mathcal{R}$*, there is an algorithm that computes an optimal solution of the* λ_{ave}*-Maximum Segmentation Length problem in a streaming fashion in* $O(mn)$ *time and* $O(m)$ *space. One can also find in* $O(mn)$ *time and* $O(m + |P|)$ *space the corresponding partition* P*, thus solving the* λ_{ave}*-Maximum Segmentation Partition problem.*

4 λ_{min}-Maximum Segmentation problems

In this section, we give an $O(mn)$ time algorithm building an optimal solution of λ_{min}-MSP and λ_{min}-MSL. We focus on solving the problem λ_{min}-MSL by using dynamic programming algorithm; the corresponding partition (solution of λ_{min}-MSP) can be reconstructed by "backtracking" in a standard way (see [9]).

4.1 Dynamic programming algorithm

Given an integer M and a set of recombinants $\mathcal{R}$, the λ_{min}-Maximum Segmentation Length problem seeks to maximize the smallest cardinality of the intervals of a partition P subject to $\lambda_{max} \leq M$. In other words, this problem is to compute

$$\max_{P \in \mathcal{P}_{M,\mathcal{R}}} \min\{j - i + 1 : [i, j] \in P\} \tag{2}$$

where $\mathcal{P}_{M,\mathcal{R}}$ is the set of all partitions P of $[1, n]$ such that for all $[i, j] \in P$, $|\mathcal{R}[i, j]| \leq M$.

To solve λ_{min}-MSL, we define the following recursion which solves (2):

$$N(k) = \begin{cases} \infty & \text{If } k = 0, \\ \max\limits_{c_k \leq j < k} \min\{N(j), k - j\} & \text{Otherwise.} \end{cases} \tag{3}$$

Algorithm 1 The algorithm $Next(x, k)$.

```
1: z ← k − N(x);
2: w ← Argmax{N(u) : x ≤ u < z};
3: if x < z and N(w) > N(x) then
4:     return Next(x + 1, k);
5: else
6:     return x;
```

Given k between 1 and n, we denote by $\texttt{Previous}(k)$ the set of previous values of k by (3), i.e. $\begin{aligned}\texttt{Previous}(k) &= \text{Argmax}_{c_k \le j < k} \min\{N(j), k-j\} \\ &= \{j \in [c_k, k-1] : N(k) = \min\{N(j), k-j\}\}.\end{aligned}$

We can exhibit two recursive properties, one on c_k and one on $\texttt{Previous}(k)$ (with Algorithm 1).

Lemma 6. *Given $k \in [1, n-1]$, we have*

1. $c_k \le c_{k+1}$,
2. *For all $j \in \texttt{Previous}(k)$, $Next(\max\{j, c_{k+1}\}, k+1) \in \texttt{Previous}(k+1)$.*

Proof. For 1, we straightforwardly obtain $c_k \le c_{k+1}$ due to Lemma 1.

For 2, we begin by proving that for all $j_k \in \texttt{Previous}(k)$, there exists $j_{k+1} \in \texttt{Previous}(k+1)$ with $j_k \le j_{k+1}$. Assume that it is not the case. Let be $j_k \in \texttt{Previous}(k)$ such that $\forall j_{k+1} \in \texttt{Previous}(k+1)$, $j_{k+1} < j_k$. In this case, we have $c_{k+1} < j_k$ and for all $j' \in [c_{k+1}, j_k - 1]$, $N(j') \le N(k) = \min\{N(j_k), k - j_k\}$. If $N(j_k) \le k - j_k$, we have that $N(j_k) < k - j_k + 1$ and thus $N(k+1) \le N(k)$. As $j_k \in [c_{k+1}, k+1]$, j_k is an element of $\texttt{Previous}(k+1)$ which is impossible. Otherwise, we have $N(j_k) > k - j_k$, $N(j_k) \ge k - j_k + 1$ and thus $j_k \in \texttt{Previous}(k+1)$ which is also impossible.

Now, we know that we can search j_{k+1} in $[\max\{j_k, c_{k+1}\}, k+1]$. In Algorithm 1, we decrease the size of the interval by the left until finding an element of $\texttt{Previous}(k+1)$. Indeed for $Next(x, k)$, if $N(w) > N(x)$, there exists $u \in [x, k - N(x) - 1]$ such that $N(u) > N(x)$ and thus $\min\{N(x), k - x\} < \min\{N(u), k - u\}$ and $x \notin \texttt{Previous}(k)$. Otherwise, we know that for all $u \in [x, k - N(x) - 1]$, $\min\{N(x), k - x\} \ge N(u) \ge \min\{N(u), k - u\}$ and for all $u \in [k - N(x), k]$, $\min\{N(x), k - x\} \ge k - u \ge \min\{N(u), k - u\}$. Hence, we have for $x = Next(\max\{j_k, c_{k+1}\}, k+1)$, for all $u \in [\max\{j_k, c_{k+1}\}, k+1]$, $\min\{N(x), k+1-x\} \ge \min\{N(u), k+1-u\}$ and thus $x \in \texttt{Previous}(k+1)$. □

Theorem 3. *Given a bound M and a set of recombinants $\mathcal{R}$, there is an algorithm that computes an optimal solution of the λ_{min}-Maximum Segmentation Length problem in a streaming fashion in $O(nm)$ time and $O(m + \mathcal{C})$ space where $\mathcal{C} = \max\{k - c_k + 1 : k \in [1, n]\}$. Using an additional array of length n, one can also find in $O(n)$ time the corresponding partition, thus solving the λ_{min}-Maximum Segmentation Partition problem.*

Proof. By using a Range Maximum Queue (see Lemma 8) on the table of $N(.)$ initialized in size $\mathcal{C}$, we can build one recursive step of $Next$ (Algorithm 1) in

$O(1)$. By using the pBWT, we can precompute to find $\mathcal{C}$ and build all the c_k in $O(nm)$ time (see Lemma 4 and Equation 1).

By Lemma 6, we can call $O(k)$ times the algorithm $Next$ to build $N(k)$. Hence, we can solve the λ_{min}-Maximum Segmentation Length in the optimal $O(nm)$ time. □

4.2 Range Maximum Queue

Our algorithm for solving λ_{min}-MSL (Theorem 3) requires a Range Maximum Queue data structure. We begin by presenting a semi-dynamic RMQ data structure that can answer RMQ queries on the array Q in constant time and can "extend" Q to the right (see Lemma 7).

Lemma 7. *There exists a data structure that maintains an integer array $Q[1,n]$ and supports the append query, which adds a new element to the end of Q and increments n, in $O(1)$ amortized time and the Range Maximum Query, which, for given $i \in [1,n]$ and $j \in [i,n]$, computes a position $h \in [i,j]$ such that $Q[h] = \max\{Q[\ell]\colon i \le \ell \le j\}$, in $O(1)$ time.*

Proof. Our solution is a straightforward modification of the classical static Range Minimum Queury approach used in, for instance, [4] and [1].

Let n be the current length of Q. Denote $b = \lceil \frac{\log n}{4} \rceil$. We split $Q[1,n]$ into blocks of length b. As is standard, a Range Maximum Query on $Q[i,j]$ is reduced to two queries inside blocks and one "block-aligned" query: provided i and j belong to different blocks (i.e., $\lfloor (i-1)/b \rfloor < \lfloor (j-1)/b \rfloor$), the new three query ranges are $Q[i,i']$, $Q[i'+1,i'']$, $Q[i''+1,j]$ (each might be empty) such that i' and i'' are multiples of b, $i'-i<b$, and $j-i''<b$.

To process the query on $Q[i'+1,i'']$, we maintain, for each $k \in [0,\log n]$, an array $P_k[1,\lfloor \frac{n}{b} \rfloor]$ storing positions of maximums in ranges of 2^k blocks; more precisely, for $h \in [1,\lfloor \frac{n}{b} \rfloor]$, we have $(h-2^k)b < P_k[h] \le hb$ and $Q[P_k[h]] = \max\{Q[\ell]\colon (h-2^k)b < \ell \le hb\}$, assuming $Q[\ell] = +\infty$ for $\ell \le 0$. Then, putting $k = \lfloor \log((i''-i')/b) \rfloor$, the maximum in $Q[i'+1,i'']$ obviously is $\max\{Q[P_k[i''/b]], Q[P_k[i'/b+2^k]]\}$ and we return either $P_k[i''/b]$ or $P_k[i'/b+2^k]$ accordingly. To calculate k in $O(1)$ time, we use either a special processor instruction or a precomputed table $L[1,2^{\lceil \frac{\log n}{2} \rceil}]$ such that $L[x] = \lfloor \log x \rfloor$ for $x \in [1,2^{\lceil \frac{\log n}{2} \rceil}]$ (hence, $\lfloor \log x \rfloor = L[x/2^{\lceil \frac{\log n}{2} \rceil}] + \lceil \frac{\log n}{2} \rceil$ for $x \in [2^{\lceil \frac{\log n}{2} \rceil}+1, n]$). Note that the length of L is $O(\sqrt{n})$.

For "in-block" queries, we maintain an array $C[1,\lceil \frac{n}{b} \rceil]$ succinctly encoding Cartesian trees for all blocks (see below). The *Cartesian tree*[3] for an array $A[h,h']$ is a binary tree with vertices $[h,h']$ whose root is the smallest $r \in [h,h']$ such that $A[r] = \max\{A[\ell]\colon h \le \ell \le h'\}$, and the left (resp., right) child of r (if any) is the root of the Cartesian tree for $A[h,r-1]$ (resp., $A[r+1,h']$). For each $h \in [1,\lceil \frac{n}{b} \rceil - 1]$, we encode the Cartesian tree for the block $Q[(h-1)b+1,hb]$ as a sequence of $2b$ balanced parentheses and store it as a $2b$-bit integer in $C[h]$ (zero/one bits correspond to opening/closing parentheses); $C[\lceil \frac{n}{b} \rceil]$ stores the

[3] The original of cartesian tree is for Range Minimum Query.

Cartesian tree for $Q[(\lceil \frac{n}{b} \rceil - 1)b+1, n]$. It is well known that if a is the lowest common ancestor of two vertices p and q ($p \le q$) in the Cartesian tree for $A[h, h']$, then $A[a] = \max\{A[\ell] \colon p \le \ell \le q\}$. We precalculate a table $T[0, 2^{2b}-1][1, b][1, b]$ such that, given numbers $p, q \in [1, b]$ and a $2b$-bit integer x encoding the Cartesian tree for an integer array $A[1, b]$, $T[x][p][q]$ stores the lowest common ancestor of p and q in the tree (if x does not encode any such tree, the value of $T[x][p][q]$ is undefined). Now, using the table T and the array C, one can straightforwardly answer in-block queries in $O(1)$ time. Note that the size of T is $O(2^{2b}b^2) = O(\sqrt{n}\log^{O(1)} n)$.

It remains to describe how the defined structures are modified. Suppose that a new value is appended to the end of Q and n is incremented. If the new n is a multiple of b, a new element $P_k[\lfloor \frac{n}{b} \rfloor]$ is added to each array P_k: $P_0[\lfloor \frac{n}{b} \rfloor]$ is computed naively in $O(b)$ time and, for $k > 0$, $P_k[\lfloor \frac{n}{b} \rfloor]$ is set to either $\ell = P_{k-1}[\lfloor \frac{n}{b} \rfloor]$ or $\ell' = P_{k-1}[\lfloor \frac{n}{b} \rfloor - 2^{k-1}]$, depending on whether $Q[\ell] < Q[\ell']$. Thus, we spend $\Theta(b + \log n) = \Theta(b)$ time to update all P_k, which is amortized among the previous $b - 1$ appends to Q in which n was not a multiple of b.

To maintain C, we utilize a well-known fact that the Cartesian tree for any array $A[h, h']$ can be constructed in $O(h' - h)$ time online, i.e., we read $A[h, h']$ from left to right and, after processing each prefix $A[h, h'']$, have the Cartesian tree for $A[h, h'']$ (e.g., see [4]). The Cartesian tree for the last block $Q[(\lceil \frac{n}{b} \rceil - 1)b+1, n]$ of Q is maintained using this online algorithm and, thus, when n is incremented, we have a new tree and have to update $C[\lceil \frac{n}{b} \rceil]$. To this end, we construct $b - 1$ tables $T_j[0, 2^{2j}-1][1, 2j]$, for $j \in [1, b-1]$, such that, given $h \in [1, 2j]$ and a $2j$-bit number x that encodes a tree F with j vertices $[1, j]$ in a balanced parentheses form, $T_j[x][h]$ contains a $2(j+1)$-bit integer encoding a tree obtained from F by attaching the new vertex $j + 1$ as a leaf to h if $h \le j$, or by making $j + 1$ the new parent of $h - j$ (so that the old parent of $h - j$, if any, is the parent of $j + 1$) if $h > j$; $T_j[x][h]$ is undefined if x does not encode any tree. Using the tables T_j and the online algorithm, one can maintain C in $O(n)$ total time. Note that the size of all T_j is $O(2^{2b}b^2) = O(\sqrt{n}\log^{O(1)} n)$.

Finally, if the new value of $\lceil \log n \rceil$ differs from the old one, we rebuild all structures in a straightforward way: the tables T, T_j, and L are precomputed in $O(\sqrt{n}\log^{O(1)} n) = o(n)$ time, P_0 is constructed from Q in one pass, each array P_k with $k \in [1, \log n]$ is computed using P_{k-1} in $\Theta(\frac{n}{b}) = \Theta(\frac{n}{\log n})$ time, and the Cartesian trees for all blocks are built in $O(n)$ total time and encoded in the array C. Overall, the rebuilding takes $O(n)$ time. Since this process is initiated only when n is a power of two, the total time is $O(n)$ in the end. □

In our algorithm, all the RMQ queries of the semi-dynamic RMQ data structure are made on a window from left to right. We can see this data structure as a *queue*, i.e. a data structure where we can make insertion at the end and deletion at the beginning in constant time. A *Range Maximum Queue* (or RMQe) is the data structure Q that supports queue operations and range maximum query, which, for given $i \in [1, n]$ and $j \in [i, n]$, computes a position $h \in [i, j]$ such that $Q[h] = \max\{Q[\ell] \colon i \le \ell \le j\}$, in $O(1)$ time. If we know the maximum number of elements of the RMQe data structure, we can improve the space

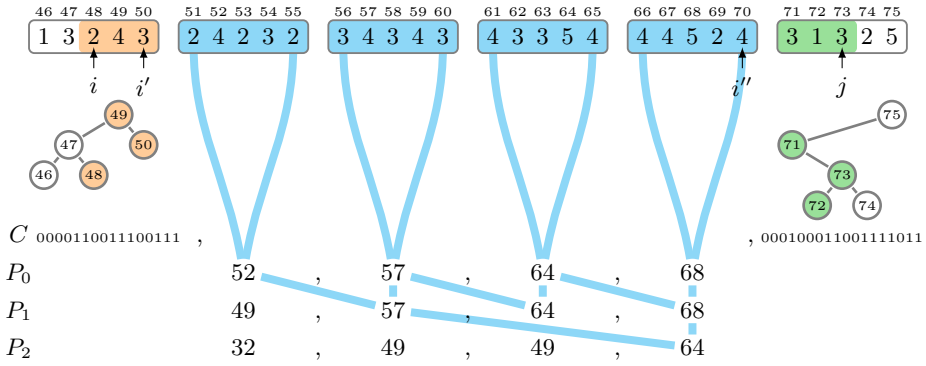


Fig. 1. Example of the construction of the semi-dynamic Range Maximum Query of Lemma 7. For the interval $[48, 73]$ with $b = 5$, we split this interval in three intervals: $[48, 50]$, $[51, 70]$ and $[71, 73]$. We build the positions of the maximum for these three intervals which are $T[0001001111][2][5] = 49$, $P_2[15] = 64$ and $T[0010010111][1][3] = 71$ and we take this one with the maximum value in the array which is 64.

complexity of our algorithm by removing the first elements of the array that will no longer query.

Lemma 8. *Let N be an integer. There exists a queue data structure that maintains an integer array $Q[1, n]$ with $n \leq N$ and supports the Range Maximum query, which adds a new element to the end of Q and increments n, in $O(1)$ amortized time, remove elements to the beginning of Q in $O(1)$ amortized time and the Range Maximum Query, which, for given $i \in [1, n]$ and $j \in [i, n]$, computes a position $h \in [i, j]$ such that $Q[h] = \max\{Q[\ell] : i \leq \ell \leq j\}$, in $O(1)$ time.*

Proof. We use the data structure that we explain in Lemma 7 but we initialize all the arrays in function of N instead of n. Hence we initialize $b = \lceil \frac{\log N}{4} \rceil$, $\log N + 1$ arrays $P_k[1, \lceil \frac{N}{b} \rceil]$ (with $k \in [0, \log N]$), $L[1, 2^{\lceil \frac{\log N}{2} \rceil}]$ and $C[1, \lceil \frac{N}{b} \rceil]$. We add also an integers *begin* initialized to 1 to store the beginning of the arrays P_k and the array C and another integer *start* initiated to 0 to store the shift in the first block (during all our algorithm $start \in [0, b-1]$). We define by $+_A$ the modular addition (plus one) in $[1, A]$, i.e. for all x and y in $[1, A]$, $x +_A y$ is equal to $x + y$ if $x + y \leq A$ and $x + y - A$ otherwise ($x +_A y \in [1, A]$).

To remember, in Lemma 7, a Range Maximum Query on $Q[i, j]$ is reduced to two queries inside blocks $Q[i, i']$ and $Q[i''+1, j]$ and one "block-aligned" query $Q[i'+1, i'']$ with i' and i'' are multiples of b, $i' - i < b$, and $j - i'' < b$. As we shift of *start* elements on the right, we take i' and i'' two multiples of b such that $i + start \leq i' \leq i'' \leq j + start$, $i' - (i + start) < b$, and $(j + start) - i'' < b$. To build $Q[i'+1, i'']$ we put $k = \lfloor \log((i'' - i')/b) \rfloor$ and return $P_k[i''/b +_{\lceil \frac{N}{b} \rceil} begin]$ or $P_k[i'/b + 2^k +_{\lceil \frac{N}{b} \rceil} begin]$. To build the queries inside blocks

$Q[i, i']$ and $Q[i''+1, j]$ we need to compute $T[C[i'/b +_{\lceil \frac{N}{b} \rceil} begin]][i + start][i']$ and $T[C[i''/b +_{\lceil \frac{N}{b} \rceil} begin]][i''][j + start]$.

To remove an element at the beginning of our RMQe we only need to increase *start* by 1: *start* becomes $start + 1$ if $start < b - 1$ and otherwise *start* becomes 0 and we update *begin* to $begin +_{\lceil \frac{N}{b} \rceil} 1$.

To add an element at the end, we use the same online algorithm of Lemma 7 (see [4]) to maintain the P_k arrays and C by updating the elements of index *end* except the fact that if n is a multiple of b, we do not create a new element, we just update *end* to $end +_{\lceil \frac{N}{b} \rceil} 1$.

□

The lemma can be further strenghtened by removing the requirement of knowing the bound N: In that case, one needs to consider the case when $\log n$ changes. Unlike in Lemma 7, series of alternating insertions and deletions can now cause $\lceil \log n \rceil$ to change at each operation, so the amortization argument cannot be used directly. However, this case can be handled by maintaining all structures for two consective $\lceil \log n \rceil$ values: Consider $x = \lceil \log n \rceil$ to change into $x+1$ due to insertion. We build all structures for $x+1$ as in the proof of Lemma 7, but also keep the structures for x. All insertions and deletions are applied on both structures until $\lceil \log n \rceil$ becomes $x - 1$ or $x + 2$. We then build structures for $x - 1$ and keep structures for $x - 1$ and x, or build structures for $x + 2$ and keep structures for $x + 1$ and $x + 2$. Now the $O(n)$ rebuilding cost can be amortized to the $O(n)$ work done before it takes place.

5 Conclusion

In this article, we described linear algorithms for Maximum Segmentation problems (see Table 1). In our Column Stream Model, we assume that we see our data column by column and thus the time complexity is $\Omega(mn)$ and the space complexity is $\Omega(m + \mathcal{X})$ where $\mathcal{X}$ is the size of the output ($\mathcal{X}$ is equal to 1 for λ_{ave}-MSL and λ_{min}-MSL and the cardinality of the optimal partition for λ_{ave}-MSP and λ_{min}-MSP). All of these algorithms can be applied in the random access data model (without data streams): they give exactly the same time complexities.

Problems	Time complexity	Space complexity	Source
λ_{ave}-MSP	$O(mn)$	$O(m + \lvert P \rvert)$ where P is an optimal partition	Theorem 1
λ_{ave}-MSL	$O(mn)$	$O(m)$	Theorem 2
λ_{min}-MSP	$O(mn)$	$O(m + n)$	Theorem 3
λ_{min}-MSL	$O(mn)$	$O(m + \max\{k - c_k : k \in [1, n]\})$	Theorem 3

Table 1. Summary of Maximum Segmentation problems in the column stream model.

As future work, we plan to implement the algorithms and offer them as new features of our founder reconstruction toolbox.[4]

[4] https://github.com/tsnorri/founder-sequences

References

1. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, pages 88–94, 2000. URL: https://doi.org/10.1007/10719839_9, doi:10.1007/10719839_9.
2. Guillaume Blin, Romeo Rizzi, Florian Sikora, and Stéphane Vialette. Minimum mosaic inference of a set of recombinants. *Int. J. Found. Comput. Sci.*, 24(1):51–66, 2013.
3. Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.
4. 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.
5. Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, 1986.
6. Veli Mäkinen and Tuukka Norri. Applying the positional Burrows–Wheeler transform to all-pairs hamming distance. *Submitted manuscript*, 2018.
7. Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Minimum segmentation for pan-genomic founder reconstruction in linear time. In *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland*, pages 15:1–15:15, 2018. doi:10.4230/LIPIcs.WABI.2018.15.
8. 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.
9. 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.
10. 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.

Appendix

About the Column Stream Model

Given an algorithm for a problem with an input $\mathcal{I}$ and an output $\mathcal{O}$, the space complexity of this algorithm corresponds to the space used by $\mathcal{I}$ and by $\mathcal{O}$ and the *auxiliary space* which is the temporary space used by this algorithm. Therefore the space complexity is in $\Omega(|\mathcal{I}| + |\mathcal{O}|)$. In the case of problems of Maximum Segmentation, all algorithms have a space complexity of $\Omega(nm)$ where the input is a set of m strings of size n. As we want to avoid an auxiliary space of $\Theta(nm)$ (this could be too big for a computer), we cannot use the random access model. Indeed the random access model corresponds to open all the file in input in the temporary memory. We suggest a specific streaming data model where the set of strings of the same length is seen column by column: the *Column Stream Model*. In this model, the size of the input is in $\Theta(m)$ which is acceptable.

To prove the realism of this model, we implemented a streaming way to read a file and we tested this implementation with files of different sizes (see Figure 2). The experiments were run on a machine with an Intel Xeon E5-2680 v4 2.4GHz CPU, which has a 35 MB Intel SmartCache. The machine has 256 gigabytes of memory at a speed of 2400MT/s. The code was compiled with `g++` using the `-Ofast` optimization flag.

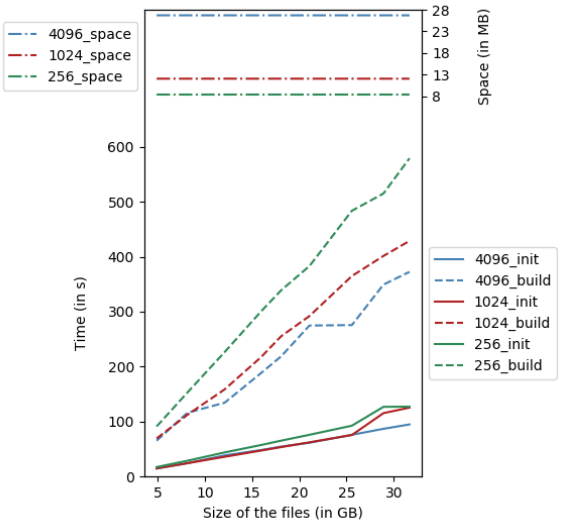


Fig. 2. Time and space complexity to read a set of recombinants depending of the buffer size (256, 1024 and 4096).