Computing Maximal Unique Matches with the r-Index

Sara Giuliani
Department of Computer Science, University of Verona, Italy

Giuseppe Romana
Department of Computer Science, University of Palermo, Italy

Massimiliano Rossi
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Abstract

In recent years, pangenomes received increasing attention from the scientific community for their ability to incorporate population variation information and alleviate reference genome bias. Maximal Exact Matches (MEMs) and Maximal Unique Matches (MUMs) have proven themselves to be useful in multiple bioinformatic contexts, for example short-read alignment and multiple-genome alignment. However, standard techniques using suffix trees and FM-indexes do not scale to a pangenomic level. Recently, Gagie et al. [JACM 20] introduced the r-index that is a Burrows-Wheeler Transform (BWT)-based index able to handle hundreds of human genomes. Later, Rossi et al. [JCB 22] enabled the computation of MEMs using the r-index, and Boucher et al. [DCC 21] showed how to compute them in a streaming fashion.

In this paper, we show how to augment Boucher et al.'s approach to enable the computation of MUMs on the r-index, while preserving the space and time bounds. We add additional $\mathcal{O}(r)$ samples of the longest common prefix (LCP) array, where r is the number of equal-letter runs of the BWT, that permits the computation of the second longest match of the pattern suffix with respect to the input text, which in turn allows the computation of candidate MUMs. We implemented a proof-of-concept of our approach, that we call MUM-PHINDER, and tested on real-world datasets. We compared our approach with competing methods that are able to compute MUMs. We observe that our method is up to 8 times smaller, while up to 19 times slower when the dataset is not highly repetitive, while on highly repetitive data, our method is up to 6.5 times slower and uses up to 25 times less memory.

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

Keywords and phrases Burrows–Wheeler Transform, r-index, maximal unique matches, bioinformatics, pangenomics

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.22

Related Version *Full Version*: `https://arxiv.org/abs/2205.01576`

Supplementary Material *Software*: `https://github.com/saragiuliani/mum-phinder`

Funding *Massimiliano Rossi*: National Science Foundation NSF EAGER (Grant No. 2118251), and National Institutes of Health (NIH) NIAID (Grant No. HG011392).

Acknowledgements We thank Travis Gagie for suggesting this problem as a project for his course CSCI 6905 at Dalhousie University. We also thank the anonymous reviewers for their insightful comments.

1 Introduction

With the advent of third-generation sequencing, the quality of assembled genomes drastically increased. In the last year the Telomere-to-Telomere project released the first complete haploid human genome [19] and the Human Pangenome Reference Consortium (HPRC) plans to release hundreds of high-quality assembled genomes to be used as a pangenome reference. One important step to enable the use of these high-quality assembled genomes is to build a multiple-sequence alignment of the genomes. Tools like MUMmer [13, 18], and Mauve [5] proposed a solution to the original problem of multiple-sequence alignment by using Maximal Unique Matches (MUMs) between two input sequences as prospective anchors for an alignment. MUMs are long stretches of the genomes that are equal in both genomes and occur only once in each of them. To reduce the computational costs of computing the MUMs, progressive approaches have also been developed like progressive Mauve [6] and progressive Cactus [1] that enables the construction of pangenome graphs, among others, that have been used in recent aligners like Giraffe [21]. MUMs have also been proven useful for strain level read quantification [23], and as a computationally efficient genomic distance measure [7].

Recent advances in pangenomics [20, 3] demonstrated that it is possible to index hundreds of Human Genomes and to query such an index to find supersets of MUMs that are maximal exact matches (MEMs), which are substrings of the pattern that occur in the reference and that cannot be extended neither on the left nor on the right. The tool called MONI [20] requires two passes over the query sequence to report the MEMs. Later PHONI [3] showed how to modify the query to compute the MEMs in a streaming fashion, with only one single pass over the query string. Both MONI and PHONI are built on top of an r-index [11] and a straight-line program SLP [9]. Their main objective is to compute the so called matching statistics (see Definition 3) of the pattern with respect to the text, that can be used to compute the MEMs with a linear scan. While, MONI uses the SLP for random access to the text, and needs to store additional information to compute the matching statistics and the MEMs, PHONI uses the SLP to compute efficient longest common extension (LCE) queries which allow to compute the matching statistics and the MEMs with only one scan of the query.

We present MUM-PHINDER, a tool that is able to compute MUMs of a query pattern against an index on a commodity computer. The main observation of our approach is to extend the definition of matching statistics to include, for each suffix of the pattern, the information of the length of the second longest match of the suffix in the text, which allows to decide whether a MEM is also unique. We extended PHONI to keep track at each step of the query, the second longest match of the pattern in the index, and its length. To do this, we add $O(r)$ samples of the longest common prefix (LCP) array to PHONI.

We evaluated our algorithm on real-world datasets, and we tested MUM-PHINDER against MUMmer [18]. We measured time and memory required by both tools for sets of increasing size of haplotypes of human chromosome 19 and SARS-CoV2 genomes and queried using one haplotype of chromosome 19 and one SARS-CoV2 genome not present in the dataset. We report that MUM-PHINDER requires consistently less memory than MUMer for all experiments being up to 25 times smaller. Although MUMer is generally faster than ours (18 times faster for 1 haplotype of chromosome 19, and 6.5 times faster for 12,500 SARS-CoV2 genomes), it cannot process longer sequences due to memory limitations. Additionally, we observe that when increasing the number of sequences in the dataset, the construction time of MUM-PHINDER increases, while the query time decreases. This phenomenon is due to the

increase in the number of matches in the search process, that prevents the use of more computational-demanding operations. Note that, due to the use of the r-index, the efficiency of our method increases when the dataset is highly repetitive as in the case of pangenomes.

2 Preliminaries

Let $\Sigma = \{a_0 < a_1 < \ldots < a_{\sigma-1}\}$ be an *ordered alphabet*, where $<$ represents the lexicographical order. A *string* (or *text*) T is a sequence of characters $T[0]T[1]\cdots T[n-1]$ such that $T[j] \in \Sigma$ for all $j \in [0..n)$. The length of a string is denoted by $|T|$. We refer to the empty string with ε, that is the only substring of length 0.

We denote a factor (or substring) of T as $T[i..j) = T[i]T[i+1]\cdots T[j-1]$ if $i < j$, and $T[i..j) = \varepsilon$ otherwise. We refer to $T[0..j)$ as the $j-1$-th prefix of T and to $T[i..n)$ as the i-th suffix of T.

We assume throughout the paper that the text T is terminated by termination character \$ that does not occur in the original text and it is lexicographically smaller than all the other characters in the alphabet.

Suffix array, inverse suffix array, and longest common prefix array

The *Suffix array* (SA) of a string $T[0..n)$ is an array of length n such that $T[\mathsf{SA}[i]..n) < T[\mathsf{SA}[j]..n)$ for any $0 \le i < j < n$. The *Inverse Suffix array* (ISA) is the inverse of SA, i.e. $\mathsf{ISA}[i] = j$ if and only if $\mathsf{SA}[j] = i$. Let $lcp(u, v)$ be the length of the longest common prefix between two strings u and v, that is $u[0..lcp(u,v)) = v[0..lcp(u,v))$ but $u[lcp(u,v)] \neq v[lcp(u,v)]$ (assuming $lcp(u,v) < \min\{|u|, |v|\}$). The *Longest Common Prefix array* (LCP) of $T[0..n)$ is an array of length n such that $\mathsf{LCP}[0] = 0$ and $\mathsf{LCP}[i] = lcp(T[\mathsf{SA}[i-1]..n), T[\mathsf{SA}[i]..n))$, for any $0 < i < n$.

Burrows-Wheeler Transform, Run-Length Encoding, and r-index

The *Burrows-Wheeler Transform* (BWT) of T is a reversible transformation of the characters of T [4]. That is the concatenation of the characters preceding the suffixes of T listed in lexicographic order, i.e., for all $0 \le i < n$, $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1 \mod n]$. The LF-*mapping* is the function that maps every character in the BWT with its preceding text character, in the BWT, i.e. $\mathsf{LF}(i) = \mathsf{ISA}[\mathsf{SA}[i] - 1 \mod n]$.

The *run-length encoding* of a string T is the representation of maximal equal-letter runs of T as pairs (c, ℓ), where c is the letter of the run and $\ell > 0$ is the length of the run. For example, the run length encoding of $T = AAACAAGGGG$ is $(A,3)(C,1)(A,2)(G,4)$. We refer to the number of runs of the BWT with r.

The BWT tends to create long equal-letter runs on highly repetitive texts such as genomic datasets. The run-length encoding applied to the BWT (in short RLBWT) is the basis of many lossless data compressors and text indexes, such as the FM-index [8] which is the base of widely used bioinformatics tools such as Bowtie [14] and BWA [15]. Although the BWT can be stored and queried in compressed space [17], the number of samples of the SA required by the index grows with the length of the uncompressed text. To overcome this issue Gagie et al. [11] proposed the *r-index* whose number of SA samples grows with the number of runs r of the BWT. The r-index is a text index composed by the run-length encoded BWT and the SA sampled at run boundaries, i.e., in correspondence of the first and last character of a run of the BWT, and it is able to retrieve the missing values of the SA by using a predecessor data structure on the samples of the SA.

Grammar and straight-line program

A *context-free grammar* $\mathcal{G} = \{V, \Sigma, R, S\}$ consists in a set of *variables* V, a set of *terminal symbols* Σ, a set of *rules* R of the type $A \mapsto \alpha$, where $A \in V$ and $\alpha \in \{V \cup \Sigma\}^*$, and the *start variable* $S \in V$. The *language of the grammar* $\mathcal{L}(\mathcal{G}) \subseteq \Sigma^*$ is the set of all words over the alphabet of terminal symbols generated after applying some rules in R starting from S. When $\mathcal{L}(\mathcal{G})$ contains only one string T, that is $\mathcal{G}$ only generates T, then the grammar $\mathcal{G}$ is called *straight-line program* (SLP).

Longest Common Extension, rank, and select queries

Given a text $T[0..n)$, the *longest common extension* (LCE) query between two positions $0 \leq i, j < n$ in T is the length of the longest common prefix of $T[i..n)$ and $T[j..n)$. Thus, if $\ell = \mathsf{LCE}(i, j)$, then $T[i..i+\ell) = T[j..j+\ell)$ and either $T[i+\ell] \neq T[j+\ell]$ or either $i + \ell = n$ or $j + \ell = n$.

Given a character c and an integer i, we define $T.\mathsf{rank}_c(i)$ as the number of occurrences of the character c in the prefix $T[0..i)$, while we define $T.\mathsf{select}_c(i)$ as the position $p \in [0..n)$ of the ith occurrence of c in T if it exists, and $p = n$ otherwise.

3 Computing MUMs using MS

Given a text $T[0..n)$ and a pattern $P[0..m)$, we refer to any factor in P that also occurs in T as a *match*. A match w in P can be defined as a pair (i, ℓ) such that $w = P[i..i+\ell)$. We say that w is *maximal* if the match can not be extended neither on the left nor on the right, i.e. either $i = 0$ or $P[i-1..i+\ell)$ does not occur in T and either $i = m - \ell$ or $P[i..i+\ell+1)$ does not occur in T.

▶ **Definition 1.** *Given a text T and a pattern P, a Maximal Unique Match (*MUM*) is a maximal match that occurs exactly once in T and P.*

▶ **Example 2.** Let T = ACACTCTTAC**ACC**ATATCATCAA\$ be the text and P = AACCTAA the pattern. The factor **AA** is maximal in P and occurs only once in T, while it is repeated in P at positions 0 and 5. The factor **CT** of P starting in position 3 is a maximal match that occurs only once in P, but it is not unique in T. The factor **CC** of P starting in position 2 is unique in both T and P, but both can be extended on the left with an **A**. On the other hand, the factor $P[1..4) = T[10..13)$ =**ACC** is a MUM.

From now on, we refer to the set of all maximal unique matches between T and P as MUMs. In [3] the authors showed how to compute maximal matches (not necessarily unique neither in T nor P) in $\mathcal{O}(r + g)$ space, where r is the number of runs of the BWT of T and g is the size of the SLP representing the text T. This is achieved by computing the *matching statistics*, for which we report the definition given in [3].

▶ **Definition 3** ([3])**.** *The matching statistics* MS *of a pattern $P[0..m)$ with respect to a text $T[0..n)$ is an array of (position, length)-pairs $\mathsf{MS}[0..m)$ such that*

- $P[i..i + \mathsf{MS}[i].\mathsf{len}) = T[\mathsf{MS}[i].\mathsf{pos}..\mathsf{MS}[i].\mathsf{pos} + \mathsf{MS}[i].\mathsf{len})$*;*
- *either $i = m - \mathsf{MS}[i].\mathsf{len}$ or $P[i..i + \mathsf{MS}[i].\mathsf{len} + 1)$ does not occur in T.*

That is, $\mathsf{MS}[i].\mathsf{pos}$ *is the starting position in T of an occurrence of the longest prefix of $P[i..m)$ that occurs in T, and* $\mathsf{MS}[i].\mathsf{len}$ *is its length.*

A known property of the matching statistics is that for all $i > 0$, $\mathsf{MS}[i].\mathsf{len} \geq \mathsf{MS}[i-1].\mathsf{len} - 1$.

Our objective is to show how to further compute MUMs within the same space bound. For our purpose, we extend the definition of MS array with an additional information field to each entry.

▶ **Definition 4.** *Given a text* $T = [0 \dots n)$ *and a pattern* $P = [0 \dots m)$*, we define the extended matching statistics* eMS *as an array of* (pos, len, slen)*-tuples* $\mathsf{eMS}[0 \dots m)$ *such that*

- $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{MS}[i].\mathsf{pos}$ *and* $\mathsf{eMS}[i].\mathsf{len} = \mathsf{MS}[i].\mathsf{len}$*;*
- $\mathsf{eMS}[i].\mathsf{slen}$ *is the largest value* ℓ *for which there exists* $p \neq \mathsf{eMS}[i].\mathsf{pos}$ *such that* $P[i..i+\ell) = T[p..p+\ell)$*.*

In other words, $\mathsf{eMS}[i].\mathsf{slen}$ *is the length of the second longest match of a prefix* $P[i..n)$ *in* T*.*

Note that $\mathsf{eMS}[i].\mathsf{slen} \leq \mathsf{eMS}[i].\mathsf{len}$, for any $i \in [0..m)$.

3.1 Checking Maximality and Uniqueness of matches

We now show how to compute MUMs by using the eMS array. Lemma 5 shows how to verify if a match occurs only once in T.

▶ **Lemma 5.** *Given a text* T*, a pattern* P*, and the* eMS *array computed for* P *with respect to* T*, let* $w = P[i..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$ *be a maximal match between a pattern* $P[0..m)$ *and a text* $T[0..n)\$$*. Then* w *occurs exactly once in* T *if and only if* $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$*.*

Proof. For the *if* direction, we assume by contradiction that w is unique in T and that $\mathsf{eMS}[i].\mathsf{slen} \geq \mathsf{eMS}[i].\mathsf{len}$. By definition, $\mathsf{eMS}[i].\mathsf{slen} \leq \mathsf{eMS}[i].\mathsf{len}$, hence we assume $\mathsf{eMS}[i].\mathsf{slen} = \mathsf{eMS}[i].\mathsf{len}$. By definition of $\mathsf{eMS}[i].\mathsf{slen}$ there exists $p \neq \mathsf{eMS}[i].\mathsf{pos}$ such that $w = P[i..i + \mathsf{eMS}[i].\mathsf{slen}) = T[p..p + \mathsf{eMS}[i].\mathsf{slen}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$, that contradicts the assumption that w occurs only once in the text T. Analogously, assume that $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$ and that there exists a position $j \neq \mathsf{eMS}[i].\mathsf{pos}$ such that $T[j..j+\mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos}+\mathsf{eMS}[i].\mathsf{len})$. However, this is in contradiction with the definition of $\mathsf{eMS}[i].\mathsf{slen}$ and the assumption of $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$, concluding the proof. ◀

We check the maximality of a match in the pattern using an analogous approach as in [20], that we summarize with the following lemma.

▶ **Lemma 6.** *Given a text* T*, a pattern* P*, and the* eMS *array computed for* P *with respect to* T*, let* $w = P[i..i + \mathsf{eMS}[i].\mathsf{len})$ *be a match with a text* T*. Then* w *is a maximal match if and only if either* $i = 0$ *or* $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$*.*

Proof. First we show that if $w = P[i..i + \mathsf{eMS}[i].\mathsf{len})$ is a maximal match then either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$. Let us assume that w is not maximal and either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$, hence either $P[i..i + \mathsf{eMS}[i].\mathsf{len} + 1)$ occurs in T or $P[i-1..i + \mathsf{eMS}[i].\mathsf{len})$ occurs in T. The former case is in contradiction with the definition of eMS, hence $P[i-1..i + \mathsf{eMS}[i].\mathsf{len})$ occurs in T. This implies that $i > 0$ and that $\mathsf{eMS}[i-1].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + 1$ in contradiction with the hypothesis that $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$.

Now we show that if either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$ then w is a maximal match. By definition of $\mathsf{eMS}[i].\mathsf{len}$, we know that either $i + \mathsf{eMS}[i].\mathsf{len} = m$ or $P[i..i + \mathsf{eMS}[i].\mathsf{len} + 1)$ does not occur in $T\$$, that is w cannot be extended on the right in P. If $i = 0$ we can not further extend the match w on the left, hence w is maximal. If $i > 0$, then by definition of matching statistics it holds that $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len} + 1$. Note that if there exists a

character $a \in \Sigma$ such that $P[i-1..i-1+\mathsf{eMS}[i-1].\mathsf{len}) = aw$ and aw occurs in T, then $\mathsf{eMS}[i-1] = \mathsf{eMS}[i]+1$. Hence if $\mathsf{eMS}[i-1] = \mathsf{eMS}[i]+1$ then it is easy to see that w is not maximal because it can be extended on the left. It also follows that if $\mathsf{eMS}[i-1] \leq \mathsf{eMS}[i]$ then w cannot be extended on the left, hence it is maximal and the thesis follows. ◀

Let $\mathcal{L} \subseteq [0..m)$ be the subset of positions in P such that both Lemma 5 and Lemma 6 hold, i.e. $\mathcal{L}$ contains all the positions in P where a maximal match unique in T starts. One can notice that if a match $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is a MUM, then $i \in \mathcal{L}$.

We first show that given $i \in \mathcal{L}$, if a match w_i is not unique in P, then the second occurrence of w_i in P is contained in another maximal match unique in T.

▶ **Lemma 7.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $\mathcal{L}$ be the subset of positions in P such that $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is maximal and occurs only once in T for all $i \in \mathcal{L}$. Then, w_i is not unique in P if and only if there exist $i' \in \mathcal{L} \setminus \{i\}$ and two possibly empty strings u, v such that $w_{i'} = uw_iv$ is a factor of P.*

Proof. Let us assume by contradiction that such i' does not exist, then let $j \notin \mathcal{L}$ be such that $P[j..j+|w_i|) = w_i$. Since $j \notin \mathcal{L}$ then either $P[j..j+|w_i|)$ is not unique in T, or it is not maximal. The former case it contradicts $i \in \mathcal{L}$ because $P[j..j+|w_i|) = w_i$ occurs twice in T. Hence, $P[j..j+|w_i|)$ occurs only once in T and it is not maximal, therefore there exists $k \in \mathcal{L}$ such that $k \leq j$ and $|w_k| > |w_i|$ which contradict the hypothesis. The other direction of the proof is straightforward since by definition of $w_{i'}$, either w_i occurs twice in P or it is not maximal. ◀

The following Lemma shows, for any $i \in \mathcal{L}$, if a match w_i is unique in P by using the eMS array.

▶ **Lemma 8.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $\mathcal{L}$ be the subset of positions in P such that $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is maximal and occurs only once in T, for all $i \in \mathcal{L}$. Then, w_i occurs only once in P if and only if, for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$.*

Proof. We first show that if w_i occurs only once in P then for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$. Since $\mathcal{L}$ contains only positions of maximal matches unique in T, then for all for $i \in \mathcal{L}$ we can map w_i to its occurrence in the text $T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos}+\mathsf{eMS}[i].\mathsf{len})$. Since w_i occurs only once in T, by Lemma 7 we have that $\mathsf{eMS}[i'].\mathsf{pos} = \mathsf{eMS}[i].\mathsf{pos} - |u|$ and $\mathsf{eMS}[i'].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + |u| + |v|$. Hence, $\mathsf{eMS}[i'].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[i'].\mathsf{pos} + \mathsf{eMS}[i'].\mathsf{len}$.

We now show the other direction of the implication. If given a position $i \in \mathcal{L}$ for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$ then w_i occurs only once in P. Assuming by contradiction that there exists a position $i \in \mathcal{L}$ such that for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$ and w_i does not occur only once in P, then by Lemma‘7 there exist $j \in \mathcal{L}$ and two possibly empty strings u, v such that $w_j = uw_iv$ is a factor of P. It is easy to see that $\mathsf{eMS}[j].\mathsf{pos} = \mathsf{eMS}[i].\mathsf{pos} - |u|$ and $\mathsf{eMS}[j].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + |u| + |v|$. Hence, $\mathsf{eMS}[j].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len}$, in contradiction with the hypothesis, concluding the proof. ◀

We can summarize the previous Lemmas in the following Theorem.

▶ **Theorem 9.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, for all $0 \leq i < m$, $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is a* MUM *if and only if $i \in \mathcal{L}$ and Lemma 8 holds.*

▶ **Example 10.** Let T = ACACTCTTAC**ACC**ATATCATCAA\$ be the text and P = AACCTAA the pattern. In the table below we report the values of the eMS of P with respect to T.

i	0	1	2	3	4	5	6
$P[i]$	A	A	C	C	T	A	A
eMS$[i]$.pos	21	10	11	5	6	21	8
eMS$[i]$.len	2	3	2	2	2	2	1
eMS$[i]$.slen	1	2	1	2	2	1	1

It is easy to check that $\mathcal{L} = \{0, 1, 5\}$, where $\mathcal{L}$ contains those indices i which verify both Lemma 5 (eMS$[i]$.slen $<$ eMS$[i]$.len) and Lemma 6 (either $i = 0$ or eMS$[i-1]$.len $\leq$ eMS$[i]$.len). Note that eMS[0].pos = eMS[5].pos and eMS[0].len = eMS[5].len, and by Lemma 8 we know that $P[0..2) (= P[5..7))$ is repeated in P. Since eMS[1].pos $<$ eMS[0].pos = eMS[5].pos, by Theorem 9 the match $P[1..4) = T[10..13)$ = ACC is a MUM.

3.2 Computing the second longest match

Now we show how we can compute eMS extending the algorithm presented in Boucher et al. [3] while preserving the same space-bound.

We can apply verbatim the algorithm of [3] to compute the eMS$[i]$.pos and eMS$[i]$.len while we extend the algorithm to include the computation of eMS$[i]$.slen. The following Lemma shows how to find the second longest match using the LCP array.

▶ **Lemma 11.** *Given a text T, a pattern P, and the* eMS *array of P with respect to T, let $P[i..i + \text{eMS}[i].\text{len}) = T[\text{eMS}[i].\text{pos}..\text{eMS}[i].\text{pos} + \text{eMS}[i].\text{len})$ and $q = \text{ISA}[\text{eMS}[i].\text{pos}]$. Then, for all $0 \leq q < n$, $\text{eMS}[i].\text{slen} = \max\{\text{LCP}[q], \text{LCP}[q+1]\}$, where $\text{LCP}[n] = 0$.*

Proof. Let us consider the set $\mathcal{T} = \{w_0 < w_1 < \ldots < w_n\}$ of the lexicographically sorted suffixes of T. Then, for all $i \in [0..m)$, at least one suffix of T starting with the second longest match $P[i..i + \text{eMS}[i].\text{slen})$ must be adjacent to $w_q = T[\text{eMS}[i].\text{pos}..n)$ in $\mathcal{T}$. Hence, assuming $q \neq 0$ and $q \neq n$, eMS$[i]$.slen is either the LCP value between w_{q-1} and w_q or between w_q and w_{q+1}, that are respectively LCP$[q]$ and LCP$[q+1]$. Note that if $q = 0$ then both LCP[0] and LCP[1] exist, while for the case $q = n$ only LCP$[n]$ is available, that is eMS$[i]$.slen must be LCP$[n]$. ◀

4 Algorithm description

In this section we present the algorithm that we use to compute MUMs that builds on the approach of Boucher et al. [3] for the computation of the MS array. The authors showed how to use the r-index and the SLP of [10, 9] to compute the MS array of a pattern $P[0..m)$ in $\mathcal{O}(m \cdot (t_{\text{LF}} + t_{\text{LCE}} + t_{\text{pred}}))$ time, where t_{LF}, t_{LCE}, and t_{pred} represent the time to perform respectively one LF, one LCE, and one predecessor query. Our algorithm extends Boucher et al.'s method by storing additional $\mathcal{O}(r)$ samples of the LCP array. Given a text $T[0..n)$ and a pattern $P[0..m)$, in the following, we first show how to compute the eMS array of P with respect to T using the r-index, the SLP, and the additional LCP array samples. Then we show how to apply Theorem 9 to compute the MUMs from the eMS array.

4.1 Computing the eMS array

The key point of the algorithm is to extend the last computed match backwards when possible, otherwise we search for the new longest match that can be extended on the left by using the BWT. Let q be the index such that $P[i..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{SA}[q]..\mathsf{SA}[q] + \mathsf{eMS}[i].\mathsf{len})$ is the longest match found at step i:

- if $\mathsf{BWT}[q] = P[i-1]$, then it can be extended on the left, i.e. $P[i-1..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{SA}[q]-1..\mathsf{SA}[q] + \mathsf{eMS}[i].\mathsf{len})$;
- otherwise, we want to find the longest prefix of $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ that is preceded by $P[i-1]$ in the text T. As observed in Bannai et al. [2] it can be either the suffix corresponding to the occurrence of $P[i-1]$ in the BWT immediately preceding or immediately following q, that we refer to as q_p and q_s respectively. Formally, $q_p = \max\{j < q \mid \mathsf{BWT}[j] = P[i-1]\}$ and $q_s = \min\{j > q \mid \mathsf{BWT}[j] = P[i-1]\}$.

The algorithm to compute the pos and len entry of the eMS array is analogous to the procedure detailed in [3]. We use the same data structures as the one defined in [3], that are the run-length encoded BWT and the samples of the SA in correspondence of positions q such that $\mathsf{BWT}[q]$ is either the first or the last symbol of an equal-letter run of the BWT. Note that both q_p and q_s are respectively the last and the first index of their corresponding equal-letter run.

An analogous reasoning can be formulated to compute the second longest match.

▶ **Lemma 12.** *Given a text $T[0..n)$, let* LCP, SA *and* ISA *be respectively the longest common prefix array, suffix array and inverse suffix array of T. Then, for all $0 < q \leq n$, let i, j be two integers such that $q - 1 = \mathsf{LF}[i]$ and $q = \mathsf{LF}[j]$, then if $\mathsf{BWT}[i] \neq \mathsf{BWT}[j]$ then $\mathsf{LCP}[q] = 0$, otherwise $\mathsf{LCP}[q] = \mathsf{LCE}(\mathsf{SA}[i], \mathsf{SA}[j]) + 1$.*

Proof. Let w_q be the q-th suffix in lexicographic order. Note that if $w_q = \$$ then $\mathsf{LCP}[q] = \mathsf{LCP}[q+1] = 0$. For all $1 \leq q < n$, if $w_{q-1} = a u\$$ and $w_q = b v\$$ for some $a < b \in \Sigma$ and some strings u and v, then $\mathsf{LCP}[q] = 0$. On the other hand, if $w_{q-1} = a u\$$ and $w_q = a v\$$, then $\mathsf{LCP}[q] = 1 + lcp(u\$, v\$)$. The thesis follows by observing that the suffixes $u\$$ and $v\$$ respectively correspond to w_i and w_j. ◀

Note that, the second longest match can be retrieved from the LCP values in correspondence of the longest maximal match (Lemma 11). Once we have the maximal match in position q in the BWT, we can compute $\mathsf{LCP}[q]$ and $\mathsf{LCP}[q+1]$ from the LCE queries on $T[\mathsf{SA}[q]..n)$ with $T[\mathsf{SA}[q_p]..n)$ and $T[\mathsf{SA}[q_s]..n)$ (Lemma 12).

Moreover, assuming the index q_p is the greatest index smaller than q such that $\mathsf{BWT}[q_p] = \mathsf{BWT}[q]$, then $\mathsf{LF}(q_p) = \mathsf{LF}(q) - 1$. It follows that if $\mathsf{BWT}[\mathsf{LF}(q_p)] = \mathsf{BWT}[\mathsf{LF}(q) - 1] = \mathsf{BWT}[\mathsf{LF}(q)]$, then $\mathsf{LCP}[\mathsf{LF}(q)])$ is an extension of the LCE query computed between $\mathsf{SA}[q_p]$ and $\mathsf{SA}[q]$ (see Figure 1). Symmetrically, if q_s is the smallest index greater than q such that $\mathsf{BWT}[q_s] = \mathsf{BWT}[q]$, then $\mathsf{LF}(q_s) = \mathsf{LF}(q) + 1$. Thus, at each iteration, we keep track of both LCP values computed to find the second longest match.

With respect to the implementation in [3], we add $\mathcal{O}(r)$ sampled values from the LCP array. Precisely, we store the LCP values between the first and the last two suffixes in correspondence of each equal-letter run (if only one suffix corresponds to a run we simply store 0). As shown later, this allows to overcome the problem of computing the LCE queries in case a position p in T is not stored in the sampled SA, i.e. when $\mathsf{ISA}[p]$ is neither the first nor the last index of its equal-letter run.

For simplicity of exposition we ignore the cases when a select query of a symbol c in the BWT fails. However, whenever it happens, either c does not occur in T or we are attempting to find an occurrence out of the allowed range, that is between 0 and the number

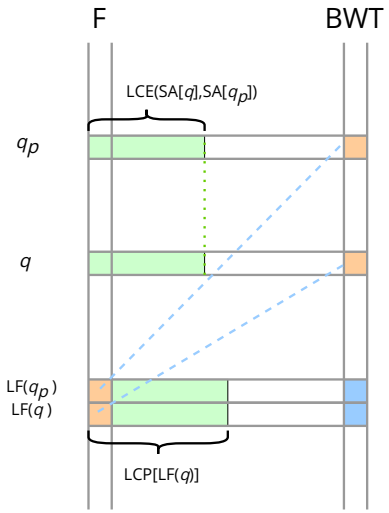


Figure 1 Application of Lemma 12 to compute $\mathsf{LCP}[\mathsf{LF}(q)]$ by extending the result of the last LCE query.

of occurrences of the character c minus 1. For the first case we can simply reset the algorithm starting from the next character of P to process, while the second occurs when we are attempting to compute an LCE query, whose result can be safely set to 0.

Algorithm 1 computes the extended matching statistics eMS of the pattern $P = [0 \dots m)$ with respect to the text $T = [0 \dots n)$ starting from the last element of the pattern (line 2). Moreover, we keep track of the first LCP values with respect to the maximal match of length 1 (line 3).

At each iteration of the loop (line 5), the algorithm tries to extend the match backwards position by position. If the match can be extended (line 7), then we use Algorithm 2 to compute the entry of the eMS. Otherwise, we use Algorithm 3 to compute the next entry of eMS (line 9).

Match case

Suppose $\mathsf{eMS}[i+1 \dots m)$ has already been processed and that $P[i] = T[\mathsf{eMS}[i+1].\mathsf{pos} - 1]$, namely we can further extend the longest match at the previous step by one position to the left. Algorithm 2 handles such scenario.

Let q be such that $\mathsf{SA}[q] = \mathsf{eMS}[i+1].\mathsf{pos} - 1$. Hence, we have that $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{eMS}[i+1].\mathsf{pos} - 1$ and $\mathsf{eMS}[i].\mathsf{len} = \mathsf{eMS}[i+1].\mathsf{len} + 1$ (line 1). At this point, we search for the greatest index q_p among those smaller than q such that $\mathsf{BWT}[q_p] = P[i]$. As discussed before, when $q_p = q - 1$, then $\mathsf{LCP}[\mathsf{LF}(q)] = \mathsf{LCP}[q] + 1 = lcp_p + 1$ (line 3). Otherwise we can compute the LCE query between $\mathsf{SA}[q]$ and $\mathsf{SA}[q_p]$, to which we add 1 for the match with $P[i]$ in correspondence of $\mathsf{BWT}[q]$ and $\mathsf{BWT}[q_p]$ (line 6). Note that $\mathsf{SA}[q] = \mathsf{eMS}[i+1].\mathsf{pos}$, while q_p is the last index of its equal-letter run (and therefore $\mathsf{SA}[q_p]$ is stored).

Analogously we compute lcp_s (lines 7-10) and, by Lemmas 11 and 12, we assign to $\mathsf{eMS}[i].\mathsf{slen}$ the maximum between lcp_p and lcp_s.

Algorithm 1 Computation of eMS.

Input : Pattern $P[0, m)$
Output : Extended matching statistics $\mathsf{eMS}[0..m)$

```
1  q ← BWT.select_{P[m−1]}(1)
2  eMS[m − 1] ← (pos : SA[q] − 1, len : 1, slen : 1)
3  lcp_p ← 0, lcp_s ← 1
4  q ← LF(q)
5  for i ← m − 2 down to 0 do
6      if BWT[q] = P[i] then
7          eMS[i], lcp_p, lcp_s ← MSMatch(P[i], q, eMS[i+1].pos, eMS[i+1].pos, lcp_p, lcp_s)
8      else
9          eMS[i], lcp_p, lcp_s ←
             MSMisMatch(P[i], q, eMS[i + 1].pos, eMS[i + 1].pos, lcp_p, lcp_s)
10     q ← LF(q)
11 return eMS
```

Mismatch case

We use Algorithm 3 when q is such that $\mathsf{BWT}[q] \neq P[i]$. We search for the index q' in SA such that, among the suffixes of T preceded by $P[i]$, at position $\mathsf{SA}[q']$ in T starts the longest match with a prefix of $P[i+1..m)$. Note that $T[\mathsf{SA}[q'] - 1] = P[i]$, and that q' is either q_p or q_s.

Hence, if $q_p = q - 1$, then by Lemma 12 the longest common prefix of $T[\mathsf{SA}[q']..n)$ and $P[i + 1..m)$ has length $lcp'_p = lcp_p$ computed at the previous step (line 5), otherwise we compute and store the LCE between $T[q..n)$ and $T[q_p..n)$ (line 7). A symmetric procedure is used to compute lcp'_s (lines 8-11).

Without loss of generality, we assume that $lcp'_s \geq lcp'_p$, hence $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{SA}[q_s] - 1$. Then $\mathsf{eMS}[i].\mathsf{len} = lcp'_s + 1$ and $lcp_p = lcp'_p + 1$ (line 13). We add 1 to both lcp'_s and lcp'_p because both matches can be extended by one position on the left since $P[i] = \mathsf{BWT}[q_p] = \mathsf{BWT}[q_s]$. In order to compute $\mathsf{eMS}[i].\mathsf{slen}$ we need to compute the value of lcp_s with respect to q_s. To do so, we look for the smallest index q'_s greater than q_s such that $\mathsf{BWT}[q'_s] = P[i]$, and then apply a similar procedure to Algorithm 2 (lines 14-18). In this case, if $\mathsf{BWT}[q_s + 1] = P[i]$, then we can retrieve lcp_s from $\mathsf{LCP}[q_s + 1]$ since q_s is in correspondence of a run boundary. Symmetrically we handle the case $lcp'_p > lcp'_s$ (lines 20-26). Finally, we compute $\mathsf{eMS}[i].\mathsf{slen}$ by picking the maximum between lcp_p and lcp_s.

▶ **Theorem 13.** *Given a text $T[0..n)$, we can build a data structure in $\mathcal{O}(r + g)$ space that allows to compute the set MUMs between any pattern $P[0..m)$ and T in $\mathcal{O}(m \cdot (t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time.*

Proof. Algorithm 1, Algorithm 2 and Algorithm 3 show how to compute the eMS array in m steps by using the data structure used in [3] of size $\mathcal{O}(r + g)$, to which we add $\mathcal{O}(r)$ words from the LCP array, preserving the space bound. Since at each step the dominant cost depends on the LF, LCE, and $\mathsf{rank}/\mathsf{select}$ queries, eMS is computed in $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time. By Lemmas 5 and 6, we can build the set $\mathcal{L}$ in $\mathcal{O}(m)$ steps from the eMS array. Recall that $\mathcal{L}$ contains those indices $i \in [0..m)$ such that $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ is a maximal match that occurs only once in T.

Algorithm 2 MSMatch($P[i], q, \mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{eMS}[i+1].\mathsf{len}, lcp_p, lcp_s$).

```
1  pos ← eMS[i + 1].pos − 1, len ← eMS[i + 1].len + 1
2  c ← BWT.rank_{P[i]}(q)
3  if BWT[q − 1] = P[i] then
       lcp_p ← lcp_p + 1
4  else
5      q_p ← BWT.select_{P[i]}(c)
6      lcp_p ← min(lcp_p, LCE(eMS[i + 1].pos, SA[q_p])) + 1
7  if BWT[q + 1] = P[i] then
       lcp_s ← lcp_s + 1
8  else
9      q_s ← BWT.select_{P[i]}(c + 2)
10     lcp_s ← min(lcp_s, LCE(eMS[i + 1].pos, SA[q_s])) + 1
11 slen ← max(lcp_p, lcp_s)
12 return (pos, len, slen), lcp_p, lcp_s
```

Now we have to search those indices in $\mathcal{L}$ that are also unique in P. A simple algorithm is to build both the LCP and ISA array of P, and then check for each $i \in \mathcal{L}$ if both $\mathsf{LCP}[\mathsf{ISA}[i]]$ and $\mathsf{LCP}[\mathsf{ISA}[i]+1]$ (or only $\mathsf{LCP}[\mathsf{ISA}[i]]$ if $\mathsf{ISA}[i] = m$) are smaller than $\mathsf{eMS}[i].\mathsf{len}$, i.e. the same property that we use to check the uniqueness in T. Both structures can be build in $\mathcal{O}(m)$ time. The overall time is $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}) + m + m)$, which collapses to $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$. ◀

Note that both g and t_{LCE} depends on the grammar scheme chosen. In fact, if exists a data structure of size λ that supports LCE queries on a text T, then we can still compute MUMs in $\mathcal{O}(r + \lambda)$ space and $\mathcal{O}(m \cdot (t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time, with t_{LCE} that depends on the data structure used.

4.2 Computing MUMs from eMS

Here we present a different approach to compute the MUMs from the eMS from the one in Theorem 13, that is of more practical use, and that does not require sorting the suffixes of P. We summarize this approach in Algorithm 4.

Let $\mathcal{L}$ be the set of indexes $i \in [0..m)$ such that $P[i..\mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$ is a maximal and unique match in T. By Lemmas 5 and 6, we can check in constant time if an index i belongs to $\mathcal{L}$. Note that building $\mathcal{L}$ (lines 3-4) can be also executed in streaming while computing the eMS array (for simplicity of exposition of the algorithms we have separated the procedures). Observe that a match $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ such that $i \in \mathcal{L}$ is a MUM if and only if it is not fully contained into another candidate, i.e. it does not exist $j \in \mathcal{L} \setminus \{i\}$ such that (i) $\mathsf{eMS}[j].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and (ii) $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len}$ (Theorem 9). Hence, we sort the elements in $\mathcal{L}$ with respect to the position in T, and starting from $\mathcal{L}[0]$, we compare every entry with the following and if both factors are not contained into the other, we store in the set MUMs the one with the smallest starting position and keep track of the other one, otherwise we simply discard the one that is repeated and continue with the following iteration.

Algorithm 3 MSMismatch$(P[i], q, \mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{eMS}[i+1].\mathsf{len}, lcp_p, lcp_s)$.

$c \leftarrow \mathsf{BWT.rank}_{P[i]}(q)$
$q_p \leftarrow \mathsf{BWT.select}_{P[i]}(c)$
$q_s \leftarrow \mathsf{BWT.select}_{P[i]}(c+1)$
if $q_p = q - 1$ **then**
  $lcp'_p \leftarrow lcp_p$
else
  $lcp'_p \leftarrow \mathsf{min}(\mathsf{eMS}[i+1].\mathsf{len}, \mathsf{LCE}(\mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{SA}[q_p]))$
if $q_s = q + 1$ **then**
  $lcp'_s \leftarrow lcp_s$
else
  $lcp'_s \leftarrow \mathsf{min}(\mathsf{eMS}[i+1].\mathsf{len}, \mathsf{LCE}(\mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{SA}[q_s]))$
if $lcp'_p \leq lcp'_s$ **then**
  $\mathsf{pos} \leftarrow \mathsf{SA}[q_s] - 1, \mathsf{len} \leftarrow lcp'_s + 1, lcp_p \leftarrow lcp'_p + 1$
  $q'_s \leftarrow \mathsf{BWT.select}_{P[i]}(c+2)$
  if $q'_s = q_s + 1$ **then**
    $lcp_s \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCP}[q_s + 1] + 1)$
  else
    $lcp_s \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCE}(\mathsf{SA}[q_s], \mathsf{SA}[q'_s]) + 1)$
  $q \leftarrow q_s$
else
  $\mathsf{pos} \leftarrow \mathsf{SA}[q_p] - 1, \mathsf{len} \leftarrow lcp_p, lcp_s \leftarrow lcp'_s + 1$
  $q'_p \leftarrow \mathsf{BWT.select}_{P[i]}(c-1)$
  if $q'_p = q_p - 1$ **then**
    $lcp_p \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCP}[q_p] + 1)$
  else
    $lcp_p \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCE}(\mathsf{SA}[q_p], \mathsf{SA}[q'_p]) + 1)$
  $q \leftarrow q_p$
$\mathsf{slen} \leftarrow \mathsf{max}(lcp_p, lcp_s)$
return $(\mathsf{pos}, \mathsf{len}, \mathsf{slen}), lcp_p, lcp_s$

To handle the special case when two candidates $i \neq j \in \mathcal{L}$ are such that $T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[j].\mathsf{pos}..\mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len})$, we further keep track whether the current maximal match is unique. This final procedure, excluding the building time for $\mathcal{L}$ that is done in streaming, takes $\mathcal{O}(|\mathcal{L}| \log |\mathcal{L}|)$ time, since the sorting of the indexes in $\mathcal{L}$ dominates the overall cost.

5 Experimental results

We implemented our algorithm for computing MUMs and measured its performances on real biological datasets. We performed the experiments on a desktop computer equipped with 3.4 GHz Intel Core i7-6700 CPU, 8 MiB L3 cache. and 16 GiB of DDR4 main memory. The machine had no other significant CPU tasks running, and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.4.0. All programs were compiled using `gcc` version 8.1.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. We recorded the runtime and memory usage using the wall clock time, CPU time, and maximum resident set size from `/usr/bin/time`.

Algorithm 4 retrieveMUMs(eMS).

Input : Extended Matching Statistics $\mathsf{eMS}[0, m)$
Output : MUMs

```
1  L, MUMs ← ∅
2  for i ← 0 to m − 1 do
3      if (i = 0 or MS[i − 1].len ≤ MS[i].len) and MS[i].len > MS[i].slen then
4          L.add(i)
5  sortByPosition (L)
6  (p, ℓ) ← (eMS[L[0]].pos, eMS[L[0]].len)
7  unique ← true
8  for i ← 1 to |L| − 1 do
9      (p′, ℓ′) ← (eMS[L[i]].pos, eMS[L[i]].len)
10     if p = p′ then
11         if ℓ = ℓ′ then
12             unique ← false
13         else if ℓ < ℓ′ then
14             ℓ ← ℓ′
15             unique ← true
16     else if ℓ < ℓ′ + (p′ − p) then
17         if unique then
18             MUMs.add((p, ℓ))
19         (p, ℓ) ← (p′, ℓ′)
20         unique ← true
21 if unique then
22     MUMs.add((p, ℓ))
23 return MUMs
```

Setup

We compare our method (MUM-PHINDER) with MUMmer [18] (`mummer`). We tested two versions of `mummer`, v3.27 [13] (`mummer3`) and v4.0 [18] (`mummer4`). We executed `mummer` with the `-mum` flag to compute MUMs that are unique in both the text and the pattern, `-l 1` to report all MUMs of length at least 1, and `-n` to match only A,C,G,and T characters. We setup MUM-PHINDER to produce the same output as `mummer`. We did not test against Mauve [6] because the tool does not directly reports MUMs. We also did not consider algorithms that does not produces an index for the text that can be queried with different patterns without reconstructing the index, e.g. the algorithm described in Mäkinen et al. [16, Section 11.1.2]. The experiments that exceeded exceeded 16 GB of memory were omitted from further consideration.

Datasets

We evaluated our method using real-world datasets. We build our index for up to 512 haplotypes of human chromosome 19 from the 1000 Genomes Project [22] and up to 300,000 SARS-CoV2 genomes from EBI's COVID data portal [12]. We provide a complete list of

Table 1 Dataset used in the experiments. For each collection of datasets of the human chromosome 19 (`chr19`) dataset in Table 1a and for the SARSCoV2 (`sars-cov2`) dataset in Table 1b, we report the number of sequences (No. seqs), the length n in Megabytes (MB), and the ratio n/r, where r is the number of runs of the BWT for each number of sequences in a collection.

(a) Collections of chromosome 19.

No. seqs	n (MB)	n/r
1	59	1.92
2	118	3.79
4	236	7.47
8	473	14.78
16	946	29.19
32	1892	57.63
64	3784	113.49
128	7568	222.23
256	15,136	424.93
512	30,272	771.53

(b) Collections of SARS-CoV2 genomes.

No. seqs	n (MB)	n/r
1562	46	459.57
3125	93	515.42
6250	186	576.47
12,500	372	622.92
25,000	744	704.73
50,000	1490	848.29
100,000	2983	1060.07
200,000	5965	1146.24
300,000	8947	1218.82

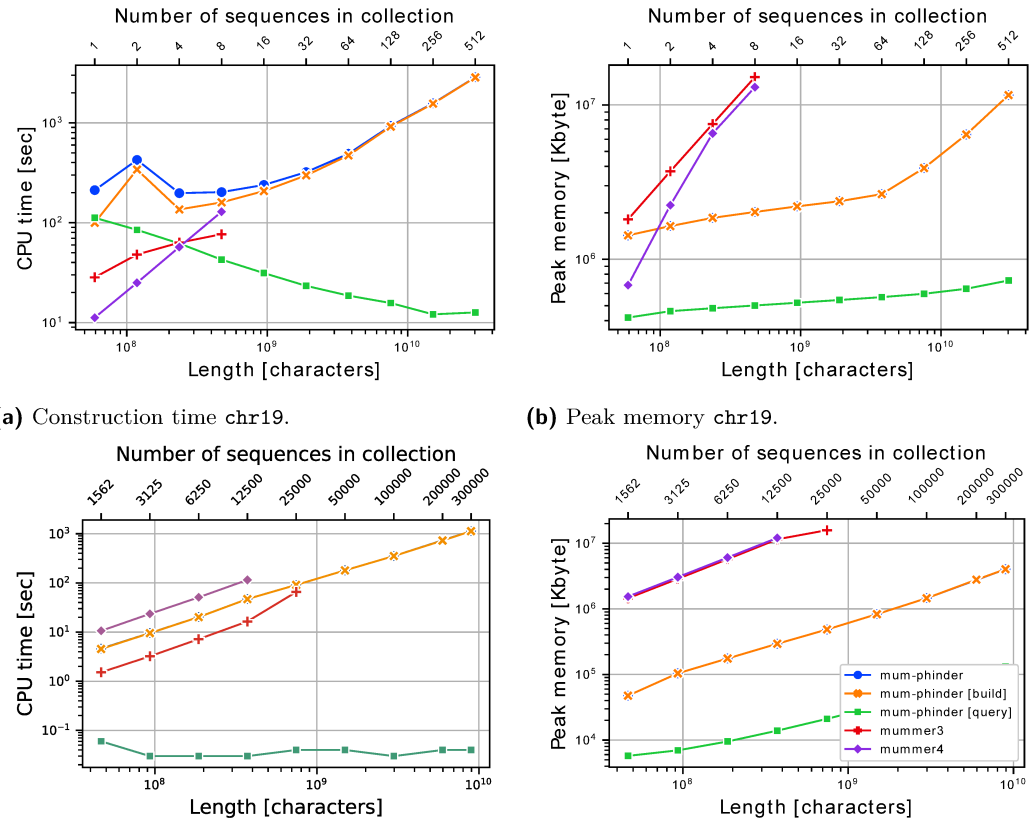
accession numbers in the repository. We divide the sequences into 11 collections of 1, 2, 3, 4, 8, 16, 32, 64, 128, 256, 512 chromosomes 19 (`chr19`) and 9 collections of 1,562, 3,125, 6,250, 1250,00, 25,000, 50,000, 100,000, 200,000, 300,000 genomes of SARS-CoV2 (`sars-cov2`). In both datasets, each collection is a superset of the previous one. In Table 1 we report the length n of each collection and the ratio n/r, where r is the number of runs of the BWT.

Furthermore, for querying the datasets, we used the first haplotype of chromosome 19 of the sample NA21144 from the 1000 Genomes Project, and the genome with accession number MZ477765 from EBI's COVID data portal [12].

Results

In Figure 2 we show the construction and query time and space for MUM-PHINDER and `mummer`. Since `mummer` is not able to decouple the construction of the suffix tree from the query, for our method we report the sum of the running times for construction and query, and the maximum resident set size of the two steps. We observe that on `chr19` `mummer3` is up to 9 times faster than MUM-PHINDER, while using up to 8 times more memory, while `mummer4` is up to 19 times faster than MUM-PHINDER, while using up to 7 times more memory. However both `mummer3` and `mummer4` cannot process more than 8 haplotypes of `chr19` due to memory limitations. MUM-PHINDER was able to build the index and query in 48 minutes for 512 haplotypes of `chr19` while using less than 11.5 GB of RAM. On `sars-cov2`, `mummer3` is up to 6.5 times faster than MUM-PHINDER, while using up to 24 times more memory, while `mummer4` is up to 1.2 times slower than MUM-PHINDER, while using up to 25 times more memory. `mummer3` was not able to process more than 25,000 genomes while `mummer4` were not able to query mote than 12,500 genomes of `sars-cov2` due to memory limitations.

In Figure 2 we also show the construction time and space for MUM-PHINDER. We observe that the construction time grows with the number of sequences in the dataset, however the query time decreases while increasing the number of sequences in the index with a 9x speedup when moving from 1 to 512 haplotypes of `chr19`. A similar phenomenon is observed in [3] and it is attributed to the increase number of match cases (Algorithm 2) while increasing the number of sequences in the index. From our profiling (data not shown) the



(a) Construction time `chr19`.

(b) Peak memory `chr19`.

(c) Construction time `sars-cov2`.

(d) Peak memory `sars-cov2`.

Figure 2 Human chromosome 19 and SARS-CoV2 genomes dataset construction CPU time and peak memory usage. We compare MUM-PHINDER with `mummer3` and `mummer4`. For MUM-PHINDER we report a breakdown of the construction (build) and query time and space. Note that for MUM-PHINDER we consider as time the sum of construction and query time, while for memory we consider the maximum between construction and query memory.

more time-demanding part of the queries are LCE queries, which are not performed in case of matches. This observation also motivates the increase in the control logic of Algorithm 3 to limit the number of LCE queries to the essential ones.

References

1 Joel Armstrong, Glenn Hickey, Mark Diekhans, Ian T. Fiddes, Adam M. Novak, Alden Deran, Qi Fang, Duo Xie, Shaohong Feng, et al. Progressive Cactus is a multiple-genome aligner for the thousand-genome era. *Nature*, 587(7833):246–251, 2020.

2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

3 Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: streamed matching statistics with multi-genome references. In *Proceedings of 2021 Data Compression Conference DCC*, pages 193–202. IEEE, 2021.

4 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL SRC RESEARCH REPORT, 1994.

5 Aaron C. E. Darling, Bob Mau, Frederick R. Blattner, and Nicole T. Perna. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome Res.*, 14(7):1394–1403, 2004.

6 Aaron E. Darling, Bob Mau, and Nicole T. Perna. progressiveMauve: multiple genome alignment with gene gain, loss and rearrangement. *PLoS One*, 5(6):e11147, 2010.

7 Marc Deloger, Meriem El Karoui, and Marie-Agnès Petit. A genomic distance based on MUM indicates discontinuity between most bacterial species and genera. *J. Bacteriol.*, 191(1):91–99, 2009.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *In Proceedings 41st annual Symposium on Foundations of Computer ScienceFOCS*, pages 390–398. IEEE Computer Society, 2000.

9 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, and Yoshimasa Takabatake. Practical Random Access to SLP-Compressed Texts. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval (SPIRE 2020)*, volume 12303 of *LNCS*, pages 221–231. Springer, 2020.

10 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, volume 11811 of *LNCS*, pages 35–44. Springer, 2019. doi:10.1007/978-3-030-32686-9_3.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

12 Peter W. Harrison, Rodrigo Lopez, Nadim Rahman, Stefan Gutnick Allen, Raheela Aslam, Nicola Buso, Carla Cummins, Yasmin Fathy, Eloy Felix, et al. The COVID-19 Data Portal: accelerating SARS-CoV-2 and COVID-19 research through rapid open access data sharing. *Nucleic Acids Research*, 49(W1):W619–W623, 2021.

13 Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biol.*, 5(2):R12, 2004.

14 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

15 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

16 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

17 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

18 Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey Zimin. MUMmer4: A fast and versatile genome alignment system. *PLoS Comput. Biol.*, 14(1):e1005944, 2018.

19 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V. Bzikadze, Alla Mikheenko, Mitchell R. Vollger, Nicolas Altemose, Lev Uralsky, et al. The complete sequence of a human genome. *bioRxiv*, 2021.

20 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *J. Comput. Biol.*, January 2022.

21 Jouni Sirén, Jean Monlong, Xian Chang, Adam M. Novak, Jordan M. Eizenga, Charles Markello, Jonas A. Sibbesen, Glenn Hickey, Pi-Chuan Chang, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

22 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, pages 68–74, 2015.

23 Kaiyuan Zhu, Welles Robinson, Alejandro A. Schäffer, Junyan Xu, Eytan Ruppin, A. Funda Ergun, Yuzhen Ye, and S. Cenk Sahinalp. Strain Level Microbial Detection and Quantification with Applications to Single Cell Metagenomics. *bioRxiv*, page 2020.06.12.149245, 2020.