PHONI: Streamed Matching Statistics with Multi-Genome References*

Christina Boucher*, Travis Gagie†, Tomohiro I‡, Dominik Köppl§, Ben Langmead¶, Giovanni Manzini‖, Gonzalo Navarro**, Alejandro Pacheco** and Massimiliano Rossi*

*U Florida
Gainesville, USA
{cboucher,rossi}
@cise.ufl.edu

†Dalhousie U
Halifax, Canada
travis.gagie
@dal.ca

‡Kyutech
Fukuoka, Japan
tomohiro
@ai.kyutech.ac.jp

§ TMDU
Tokyo, Japan
koeppl.dsc
@tmd.ac.jp
(corresponding)

¶Johns Hopkins U
Baltimore, USA
langmea
@cs.jhu.edu

‖U Piemonte Orientale
Alessandria, Italy
giovanni.manzini
@uniupo.it

**CeBiB, DCC, U Chile
Santiago, Chile
{gnavarro,alpachec}
@dcc.uchile.cl

Abstract

Computing the matching statistics of patterns with respect to a text is a fundamental task in bioinformatics, but a formidable one when the text is a highly compressed genomic database. Bannai et al. gave an efficient solution for this case, which Rossi et al. recently implemented, but it uses two passes over the patterns and buffers a pointer for each character during the first pass. In this paper, we simplify their solution and make it streaming, at the cost of slowing it down slightly. This means that, first, we can compute the matching statistics of several long patterns (such as whole human chromosomes) in parallel while still using a reasonable amount of RAM; second, we can compute matching statistics online with low latency and thus quickly recognize when a pattern becomes incompressible relative to the database. Our code is available at `https://github.com/koeppl/phoni` .

1 Introduction

Computing the matching statistics of patterns with respect to a text is a fundamental and well-studied task in bioinformatics, useful in many applications [1, 2], since they tell us which substrings of those patterns occur in that text. We consider a slightly extended definition, which includes some information on where those substrings occur.

Definition 1. The *matching statistics* MS of a pattern $P[0..m-1]$ with respect to a text $T[0..n-1]$ are an array of (position, length)-pairs $\mathsf{MS}[0..m-1]$ such that

- $P[i..i+\mathsf{MS}[i].\mathsf{len}-1] = T[\mathsf{MS}[i].\mathsf{pos}..\mathsf{MS}[i].\mathsf{pos}+\mathsf{MS}[i].\mathsf{len}-1]$,
- $P[i..i+\mathsf{MS}[i].\mathsf{len}]$ does not occur in T.

That is, $\mathsf{MS}[i].\mathsf{pos}$ is a pointer to the starting position of a copy in T of the longest prefix of $P[i..m-1]$ that occurs in T, and $\mathsf{MS}[i].\mathsf{len}$ is the length of that prefix.

For example, if $T[0..5] = \texttt{CATTAG}$ and $P[0..4] = \texttt{GTTAC}$, then $\mathsf{MS}[0..4]$ can be either $[(5,1),(2,3),(3,2),(4,1),(0,1)]$ or $[(5,1),(2,3),(3,2),(1,1),(0,1)]$. The pair $\mathsf{MS}[1] = (2,3)$ tells us that the prefix $P[1..3] = \texttt{TTA}$ of $P[1..4]$ occurs in T at $T[2..4]$ and $P[1..4]$ does not occur in T.

*MR, TG, BL and CB are funded National Science Foundation NSF IIBR (Grant No. 2029552) and National Institutes of Health (NIH) NIAID (Grant No. HG011392). CB is funded by NSF SCH: INT: (Grant No. 2013998). MR and CB are funded by NSF IIS (Grant No. 1618814) and NIH NIAID (Grant No. R01AI141810). TG is funded by NSERC Discovery Grant RGPIN-07185-2020. DK is funded by JSPS KAKENHI Grant JP18F18120. GN funded by Basal Funds FB0001 and Fondecyt Grant 1-200038, ANID, Chile. AP funded by Basal Funds FB0001 and Doctoral Scholarship grant 21180760, ANID, Chile.

Despite the importance of computing matching statistics, until recently it was not known how to index efficiently a compressed representation of a massive and highly repetitive text, such as a database of genomes of individuals from the same species, so that later we could quickly compute the matching statistics of a given pattern. Bannai et al. [3] augmented Gagie et al.'s [4] r-index to support computation of matching statistics, as we will describe in Section 3, but they did not say how to build their auxiliary data structure. Rossi et al. [5] recently gave a construction, implemented it, and used it to find maximal exact matches (MEMs) between a set of DNA reads and a genomic database. They called their implementation MONI, Finnish for "multi", since their ultimate goal is to align reads to a multi-genome reference.

Bannai et al.'s solution makes two passes over the pattern: during the first, it works from right to left and computes and buffers the pos values; during the second, it works from left to right and uses the pos values and random access to the text to compute the len values. This means the working space grows linearly with the length of the pattern. This is not a serious concern when the patterns are short reads or even long reads, which are generally hundreds or thousands of characters, respectively, but it could limit how many extremely long patterns we can process in parallel.

A human chromosome 1 is a quarter of a billion base pairs, so assuming a pos values takes 8 bytes, we could need 2 GB of RAM to buffer the pos values, on top of the RAM occupied by Bannai et al.'s index. If we have dozens of cores and want to process a chromosome 1 from a different haplotype on each core in parallel, the RAM needed to buffer the pos values could easily dwarf that needed for the index. Bannai et al. themselves proposed computing the matching statistics of whole genomes with respect to genomic databases, in order to identify regions of novel DNA — where the len values are small, meaning the region is incompressible relative to the database — for rare-disease diagnosis. Such whole-genome matching statistics could also be useful in estimating genetic diversity in a population, tracking how pathogens mutate, etc.

Another potential concern with Bannai et al.'s solution is that, if a pattern is being given to us online, character by character, then the maximum latency, from the time we receive a character to the time we return the corresponding pair in the matching statistics, also grows linearly with the length of the pattern. This means we cannot use Bannai et al.'s solution for applications in which we want the matching statistics in real time.

On the other hand, if we can stream the matching statistics in real time then, among other things, we can use the results in applications of DNA sequencing that require rapid computational feedback. For example, when the sequencing process in Oxford Nanopore MinION DNA sequencers starts to degrade in accuracy, their output continues but it no longer contains useful information [6]. With streamed matching statistics, we can stop the sequencing process when the output becomes incompressible relative to the database. Even when the MinION is still producing valid output, it may be sequencing DNA that is irrelevant for our purposes. Real-time computation of matching statistics also allows us to reject such DNA rapidly, and thus target the sequencer to specific genomes or genes of interest [7, 8]. Finally, we may want to stop sequencing once the MinION has found a long enough match to confirm the presence of a pathogen, for example [9]. For all these applications, reducing latency optimizes throughput, and reducing the memory usage allows decisions to be made "close to" the sequencer using embedded or other non-server processors.

In this paper, we simplify Bannai et al.'s solution by using longest-common-extension (LCE) queries to compute the len values at the same time that we compute the pos values. This means we process patterns using a single pass, reading them and writing the matching statistics as streams, without buffering — so our solution can be applied to the tasks described above. (To stream patterns from left to right we should really index the reversed texts, but we ignore that in this version of this paper.) Our experiments show that our implementation, which we call PHONI, needs significantly less time and memory to build than MONI; is significantly smaller once built; and uses less extra RAM to compute the matching statistics of long patterns. For the full version of this paper we will test the maximum latency and the RAM usage when the solutions process queries in parallel.

The rest of this paper is laid out as follows: in Section 2 we review how to compute and use MS, data structures supporting random access to T, and data structures supporting LCE queries; in Section 3 we explain how to simplify Bannai et al.'s algorithm to perform the computation within a single pass by means of LCE queries; and in Section 4 we present our experimental results. Our code is available at `https://github.com/koeppl/phoni` .

Algorithm 1: Lists the starting positions of all the copies of $P[i..j]$ in T for given MS, i and j, by using ϕ, ϕ^{-1} and PLCP.

```
1  if MS[i].len < j − i + 1 then return
2  p ← MS[i].pos
3  output p
4  while PLCP[p] ≥ j − i + 1 do
5  |  p ← φ(p)
6  |  output p
7  p ← φ^{-1}(MS[i].pos)
8  while p ≠ NULL and PLCP[p] ≥ j − i + 1 do
9  |  output p
10 |  p ← φ^{-1}(p)
```

2 MS, Random Access and LCE

2.1 Computing and Using MS

There are practical $O(n \log \sigma)$-bit data structures with which we can compute MS in $O(m \log \sigma)$ time, where σ is the size of the alphabet of T [10]. Once we have MS we can easily compute in $O(m)$ time the maximal exact matches (MEMs) of P with respect to T, for example, which play a key role in short- and long-read alignment [11]. In fact, with additional practical $O(n \log \sigma)$-bit data structures we can quickly list the starting positions of all the copies in T of any substring $P[i..j]$ of P.

In the worst-case we cannot store a text of length n over an alphabet of size σ in fewer than $n \lg \sigma$ bits, and for a single genome using $2n$ bits is reasonable. Due to the high speed and low cost of next-generation sequencing, however, we now have massive genomic databases to index, which are far more compressible than even what their high-order empirical entropies would indicate.

A consensus is emerging that if T is such a text then compressed indexes for it should take space bounded in terms the number r of runs in its Burrows-Wheeler Transform (BWT), where a run is a maximal non-empty unary substring. Within $O(r)$ space, we can efficiently support several powerful queries, but not yet computing MS. Indeed, MS can be easily computed with suffix trees, but suffix-tree functionality has been achieved only with $O(r \log(n/r))$-space data structures [4], which is significantly larger than $O(r)$ both in theory and in practice [12].

Bannai et al. recently presented their two-pass algorithm for quickly computing MS using only an $O(r)$-space data structure during the first pass, from right to left in $O(m \log \log n)$ time, and then random access to T during the second pass, from left to right. We do not know how to support efficient random access to T using only $O(r)$ space, however.

Once we have MS, we can use the r-index [4], which also takes $O(r)$ words of space, to list the starting positions of all the copies in T of any substring $P[i..j]$ of P, in $O(\log \log n)$ time per copy. Specifically, we use Theorem 2 below with Algorithm 1, where

- $\phi(p) = \mathsf{SA}[\mathsf{ISA}[p] - 1]$ (or NULL if $\mathsf{ISA}[p] = 0$),
- $\phi^{-1}(p) = \mathsf{SA}[\mathsf{ISA}[p] + 1]$ (or NULL if $\mathsf{ISA}[p] = n - 1$),
- $\mathsf{PLCP}[p] = \mathsf{LCP}[\mathsf{ISA}[p]]$ (or 0 if $\mathsf{ISA}[p] = 0$),

and SA, ISA, LCP and PLCP are the suffix array, inverse suffix array, longest-common-prefix array and permuted longest-common-prefix array of T, respectively.

Theorem 2 (Gagie, Navarro and Prezza, 2020)**.** We can store T in $O(r)$ space such that, given a text position $p \in [0..n-1]$, we can compute $\phi(p)$, $\phi^{-1}(p)$ and $\mathsf{PLCP}[p]$ in $O(\log \log n)$ time.

2.2 Random Access in Compressed Space

There are many data structures supporting logarithmic-time random access to compressed repetitive texts. For example, one can build a balanced grammar of size $O(z \log(n/z))$ that supports access to any symbol in time $O(\log(n/z))$ [13, 14], where z is the number of phrases in the LZ77 parse of T. In practice, heuristics like RePair [15] yield smaller balanced grammars.

Gagie et al. [16] recently showed how to scale up RePair to handle genomic databases in reasonable time, using as a preprocessing step prefix-free parsing (PFP), a technique Boucher et al. [17, 18] introduced to ease the construction of BWTs of genomic database sequences. PFP parses T by passing a sliding window over it, inserting a phrase break when the Karp-Rabin hash of the contents of the window is 0 modulo a parameter. PFP outputs a *dictionary* of phrases, and a *parse*: a sequence of dictionary symbols that when replaced by the corresponding phrases yields the original text T. PFP produces mostly locally consistent parsings in practice, meaning that long repeated substrings in T tend to be parsed roughly the same way, so the sum of the total lengths of the phrases in the dictionary and the number of phrases in the parse is usually significantly less than n.

To get an SLP for T, Gagie et al. [16] run RePair on the concatenation of the phrases in the dictionary, separated by unique symbols, to obtain an SLP containing a non-terminal for each phrase, whose expansion is that phrase. Then, they run RePair on the parse, treating it as a string of phrase identifiers, to obtain an SLP whose terminals are the phrase identifiers. By replacing the terminals in the latter SLP by the corresponding non-terminals in the former SLP, they obtain an SLP for T. Since the dictionary and the parse are significantly smaller than T, running RePair on them is usually much faster and uses much less memory than running RePair (directly) on T, but the resulting SLP is usually only negligibly larger (and much smaller than Bannai et al.'s [3] data structure, anyway).

The naïve way to augment an SLP to support fast random access is to store with each non-terminal the size of its expansion. Very recently Gagie et al. [19] gave a more space-efficient way to encode SLPs while still supporting fast random access. Rossi et al. [5] use this space-efficient SLP to support random access to T in their implementation of Bannai et al.'s algorithm.

2.3 LCE Queries

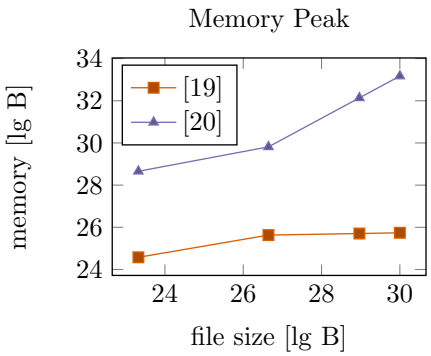

Figure 1: Memory needed for LCE queries on Chromosome 19 samples of various sizes (x-axis).

The longest common extension (LCE) query [21] asks, given two text positions i, j, for the length of the longest common prefix between $T[i..n-1]$ and $T[j..n-1]$. We make use of LCE queries to avoid the second pass in Bannai et al.'s algorithm [3].

Although there are many LCE data structures in the literature, we are not aware of any that in practice can handle genomic databases, which can range from tens of gigabytes to petabytes, while achieving compression comparable to Bannai et al.'s [3] or Gagie et al.'s [19] data structures and supporting reasonably fast queries. For example, Dinklage et al. [20] recently presented an LCE data structure based on sampling. Since its space usage grows linearly with the size of the dataset, this approach is impractical when dealing with genomic databases; see Figure 1 for a comparison of the memory usage between sss_{256} of [20] and the SLP representation of Gagie et al. [19].

We now describe a practical algorithm for LCE queries that uses the same SLP compressed text representation [19] that Rossi et al. [5] used to random-access the input text. Because PFP produces mostly locally consistent parsings in practice, if we are extracting and comparing two suffixes of T that have a long common prefix then usually we will quickly reach a phrase boundary simultaneously in both suffixes. At that point we can start comparing the suffixes phrase by phrase instead of character by character, without expanding the non-terminals representing those phrases. Once we reach phrases that do not match, we can expand their non-terminals and once again compare the suffixes character by character, knowing that we will find characters that do not match within those phrases.

We need not augment the SLP with information about which non-terminals' expansions are phrases: to extract and compare suffixes of T starting at $T[i]$ and $T[j]$, conceptually we descend to the ith and jth leaves of the parse tree for T; starting from those leaves, we then perform synchronized traversals of the parse tree, moving from left to right until we simultaneously arrive at two leaves labelled with different characters; if during the traversals we simultaneously arrive at nodes labelled with the same non-terminal then we need not explore those nodes' subtrees — whether or not that non-terminal's expansion is a phrase — since they are guaranteed to be equal. This technique is described in detail with a similar

Algorithm 2: Computes MS using $O(m)$ rank, select, SA, LF and LCE queries. For simplicity we ignore the cases where q, q' or q'' are undefined.

```
1  q ← BWT.select_{P[m−1]}(1)                          ▷ Assume that P[m − 1] occurs in T
2  MS[m − 1] ← (pos : SA[q] − 1, len: 1)
3  q ← LF(q)                                            ▷ Invariant: T[SA[q]] = P[m − 1]
4  for i = m − 2 down to 0 do
5      if BWT[q] = P[i] then
6          MS[i] ← (pos : MS[i + 1].pos − 1, len : MS[i + 1].len + 1)
7          q ← LF(q)
8      else
9          c ← BWT.rank_{P[i]}(q)
10         q′ ← BWT.select_{P[i]}(c)
11         q″ ← BWT.select_{P[i]}(c + 1)
12         ℓ′ ← min(MS[i + 1].len, LCE(SA[q′], MS[i + 1].pos))
13         ℓ″ ← min(MS[i + 1].len, LCE(SA[q″], MS[i + 1].pos))
14         if ℓ′ ≥ ℓ″ then
15             MS[i] ← (pos : SA[q′] − 1, len : ℓ′ + 1)
16             q ← LF(q′)
17         else
18             MS[i] ← (pos : SA[q″] − 1, len : ℓ″ + 1)
19             q ← LF(q″)
```

parsing in Fischer et al. [22, Sect. 3.3]; see also Nishimoto et al. [23].

3 Simplifying Bannai et al.'s Algorithm

Bannai et al. developed their algorithm starting from Algorithm 2, with which we can compute MS in $m \log^{O(1)} n$ time with $O(n \log \sigma)$-bit data structures. (The precise complexity depends on the auxiliary data structures used.) In this algorithm, $\mathsf{BWT.rank}_x(y)$ returns the number of copies of x in $\mathsf{BWT}[0..y]$, $\mathsf{BWT.select}_x(y)$ returns the position of the yth copy of x in BWT, and $\mathsf{LF}(x)$ returns the position in BWT of the character that precedes $\mathsf{BWT}[x]$ in T. We refer the reader to Navarro's text book [24] for descriptions of $O(n \log \sigma)$-bit data structures with which we can implement rank, select, SA, LF and LCE queries efficiently. For example, we can implement LCE queries using a $(2n + o(n))$-bit position-only range-minimum-query data structure over LCP, and simulating LCP using random access to SA and PLCP.

The crucial observation behind this algorithm is that when we know $\mathsf{MS}[i + 1]$ (both pos and len components):

- if $T[\mathsf{MS}[i + 1].\mathsf{pos} - 1] = P[i]$ then $\mathsf{MS}[i].\mathsf{pos} = \mathsf{MS}[i + 1].\mathsf{pos} - 1$ and $\mathsf{MS}[i].\mathsf{len} = \mathsf{MS}[i + 1].\mathsf{len} + 1$ (Line 5 in Algo. 2);
- otherwise (Line 8 in Algo. 2), a copy of the longest prefix of $P[i..m - 1]$ that occurs in T starts at either $T[p' - 1]$ or $T[p'' - 1]$, where p' and p'' are the starting positions of the lexicographically preceding and succeeding suffixes of $T[\mathsf{MS}[i + 1].\mathsf{pos}..n - 1]$, respectively.

In the latter case, by knowing the lexicographic rank (q in Algo. 2) of $T[\mathsf{MS}[i + 1].\mathsf{pos}..n - 1]$ among the suffixes of T, we can tell whether to set $\mathsf{MS}[i].\mathsf{pos}$ to $p' - 1$ or to $p'' - 1$ by comparing $\mathsf{LCE}(p', \mathsf{MS}[i+1].\mathsf{pos})$ to $\mathsf{LCE}(p'', \mathsf{MS}[i + 1].\mathsf{pos})$ (Line 14 in Algorithm 2).

Bannai et al. observed that most of the operations in this algorithm can be supported quickly using $O(r)$-space data structures. For example, we need access to SA entries only at positions corresponding to the starting or ending positions of runs in BWT, and we can store those entries in $O(r)$ space. The only exceptions are the LCE queries: the standard approach of solving LCE queries using range-minimum-query data structures over LCP cannot be applied since we have no fast $O(r)$-space range-minimum-query data structures and, even if we did, we do not know how to support random access to SA in order to use PLCP to simulate LCP.

To avoid using LCE queries, Bannai et al. observed that during the execution of the algorithm it holds

$$\mathsf{LCE}(p', \mathsf{MS}[i+1].\mathsf{pos}) \;\geq\; \mathsf{LCE}(p'', \mathsf{MS}[i+1].\mathsf{pos}) \tag{1}$$

if and only if there is a copy of $\min(\mathsf{LCP}[q'+1], \cdots, \mathsf{LCP}[q''])$ in $\mathsf{LCP}[q+1..q'']$. Therefore, if we store the position of the first minimum in $\mathsf{LCP}[q'+1..q'']$, for each choice of q' and q'' as the ending position of a run in BWT and the starting position of the next run of the same character, then we can compute the pos components of the MS array: $\mathsf{MS}[m-1].\mathsf{pos}, \mathsf{MS}[m-2].\mathsf{pos}, \ldots, \mathsf{MS}[0].\mathsf{pos}$ in this order. These values can be computed in overall $O(m \log\log n)$ time, since the cost is dominated by rank and select queries.

Without LCE queries it is not possible to compute the len components together with the pos components in an efficient way, so they are computed with a second, left to right, pass over the pattern P. Since $\mathsf{MS}[i+1].\mathsf{len} \geq \mathsf{MS}[i].\mathsf{len} - 1$, once we know $\mathsf{MS}[i].\mathsf{len}$ we can find $\mathsf{MS}[i+1].\mathsf{len}$ without looking at $T[\mathsf{MS}[i+1].\mathsf{pos}..\mathsf{MS}[i+1].\mathsf{pos} + \mathsf{MS}[i].\mathsf{len} - 2]$. The total number of random accesses to T we need forms a telescoping sum, $\mathsf{MS}[0].\mathsf{len} + \sum_{i=1}^{m-1}(\mathsf{MS}[i].\mathsf{len} - \mathsf{MS}[i-1].\mathsf{len}) + O(m)$, which collapses to $O(m)$.

Rossi et al. [5] described an implementation of the resulting two-pass algorithm providing also the missing step of the actual computation of the minima, called *thresholds* in [3], required to indirectly perform the comparison of Equation (1).

4 Experiments

We compared the time and memory needed for building and querying MONI, PHONI, and Belazzougui et al.'s data structure [10] called msfast in the sequel. In particular, we focus on three variants of PHONI and two variants of msfast, which only differ in how the queries are performed:

- $\mathsf{PHONI}_{\text{std}}$ is what we have described in previous sections.
- $\mathsf{PHONI}_{\text{naïve}}$ performs LCE queries via character-by-character extraction, without skipping equal non-terminals as described in Subsection 2.3.
- $\mathsf{PHONI}_{\text{heur}}$ changes the order of the execution of the LCE queries at Lines 12 and 13 of Algo. 2 favoring the LCE query of $q_{\min} \in \{q', q''\}$ with $|q - q_{\min}| = \min(q - q', q'' - q)$, and then omitting the second LCE query whenever the returned LCE length is at least $\mathsf{MS}[i+1].\mathsf{len}$.
- msfast is the execution of Belazzougui et al.'s data structure with default parameters.
- msfast_{+} is the execution of msfast with the parameters `-lazy_wl 1 -double_rank 1 -rank_fail 1` leading to the fastest execution.

All the implementation is done in C++17. We ran our experiments on a machine with an Intel Xeon CPU E5-2620 and 64 GiB RAM running Ubuntu 16.04.7. We executed all programs single threaded. We used the implementation `https://github.com/odenas/indexed_ms` and `https://github.com/maxrossi91/moni` for msfast and MONI.

We built the data structures for datasets consisting of chromosome 19s from human haplotypes, from 16 to 1000, and queried them with prefixes of 10 different chromosome 19s. Table 4 gives some characteristics of the used data and the resulting matching statistics.

Figure 4(a) shows the construction times and 4(b) the query times, with the average time to compute MS for a whole chromosome 19 depending on the size of the dataset on the left, and to compute MS for a prefix of a chromosome 19 on the right. We stopped constructions that took more than 200 minutes. The variations of PHONI and msfast are not shown in Figures 4(a) and 4(c) because their constructions are the same as their respective standard variants. We can clearly observe that $\mathsf{PHONI}_{\text{std}}$ is faster than $\mathsf{PHONI}_{\text{naïve}}$ while being slightly inferior to $\mathsf{PHONI}_{\text{heur}}$. $\mathsf{PHONI}_{\text{std}}$ is faster to build than MONI because it does not need the thresholds, but it is slower at answering queries for relatively small $|T|$ (although it speeds up as the dataset grows, because the fraction of the time we use LF mappings increases).

Figures 4(c) and (d) show the peak RAM (the maximum resident set size) used to build the data structures and their final sizes. $\mathsf{PHONI}_{\text{std}}$ also takes less memory that MONI because, again, it does not need thresholds. Finally, Fig. 4(e) shows the maximum amount of memory requested for allocation during a query (including the indexing data structures).

For the larger datasets, MONI and $\mathsf{PHONI}_{\text{std}}$ seem to be the most practical solutions, with MONI being faster for small datasets but somewhat harder to build and needing more RAM when computing the matching statistics for long patterns. As stated in Section 1, for the full version of this paper we will test the maximum latency and the RAM usage when the solutions process queries in parallel.

#	$\|T\|$ [GB]	|SLP| [MB]	r [M]	LF%
16	0.96	36.11	32.40	78.88
32	1.92	37.86	32.83	79.11
64	3.85	39.49	33.34	79.36
100	6.01	41.02	33.78	79.56
256	15.39	47.38	35.62	80.34
512	30.78	57.98	39.24	81.96
1000	60.11	80.64	45.93	84.61

Table 1: Characteristics of the obtained matching statistics. The column # is the number of chr19 sequences stored in T, |SLP| the sizes of PHONI$_{\text{std}}$'s SLP grammar, and r the runs in BWT (in million). The column LF% is the percentage of how often Line 5 in Algo. 2 is true. This percentage increases with the number of samples since it becomes likelier for matches the longer the indexed text becomes. The average and maximum value of len in MS are roughly 81,558 and 3,100,685 for all instances.

Construction of PHONI. To construct PHONI, we used Rossi et al.'s construction algorithm for MONI to build the RLBWT and the SLP. The indexing data structures of MONI assume multiple sequences whose characters are drawn from the byte alphabet. We omit the construction of the thresholds from the index, as we do not need them.

msfast. Regarding times, we can see that the construction and query times grow linearly with the sample sizes. While its construction is among one of the slowest, it is faster than PHONI when it comes to queries. However, due to the linear dependency on the sample size, we expect that this solution eventually becomes slower than PHONI for larger sample sizes. For the space, we observe that the construction of msfast has a huge memory footprint. Its memory requirement already reached the maximum available memory of our machine at 64 samples. Due to this physical restriction, there are no evaluations of msfast for larger sample sets.

MONI. Regarding the measured times (Fig. 4(a) and (b)), we observe that MONI with the additional need of the thresholds is slower during the construction than PHONI$_1$, while excelling at the queries. However, for larger sample sizes, the threshold computation take a considerable amount of time while PHONI's query times become faster as the number of reducible positions (cf. 4) increases, making the time-expensive LCE queries less frequent. At 1000 samples, the overall times are competitive, and we expect to see that PHONI will take the lead with even more sequences. While the difference in the construction is only the thresholds computation, PHONI and MONI largely differ in the needed memory requirements when it comes to queries. Here, MONI needs to additionally keep the patterns, the thresholds, and the pos components of the matching statistics in memory, causing a large memory footprint observable in Fig. 4(e). Unlike MONI, PHONI can stream both the input patterns and the output, making it the most favorable option when memory is the computational limitation.

Open Problems

The high percentages of reducible positions in Table 4 suggests that the matching statistics by themselves are compressible: Whenever Line 5 in Algo. 2 is true, we only have to store a single bit indicating to use the previous statistic with pos and len shifted by one. As a plain array representation of MS takes a multiple of the space of P, it could be interesting to find a space-efficient representation of MS.

We note that we run msfast in the experiments single threaded although it allows for parallelization. As as a future step, we want to parallelize PHONI and conduct an evaluation of parallel algorithms computing MS.

References

[1] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, *Genome-Scale Algorithm Design*, Cambridge University Press, 2015.

[2] E. Ohlebusch, *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag, 2013.

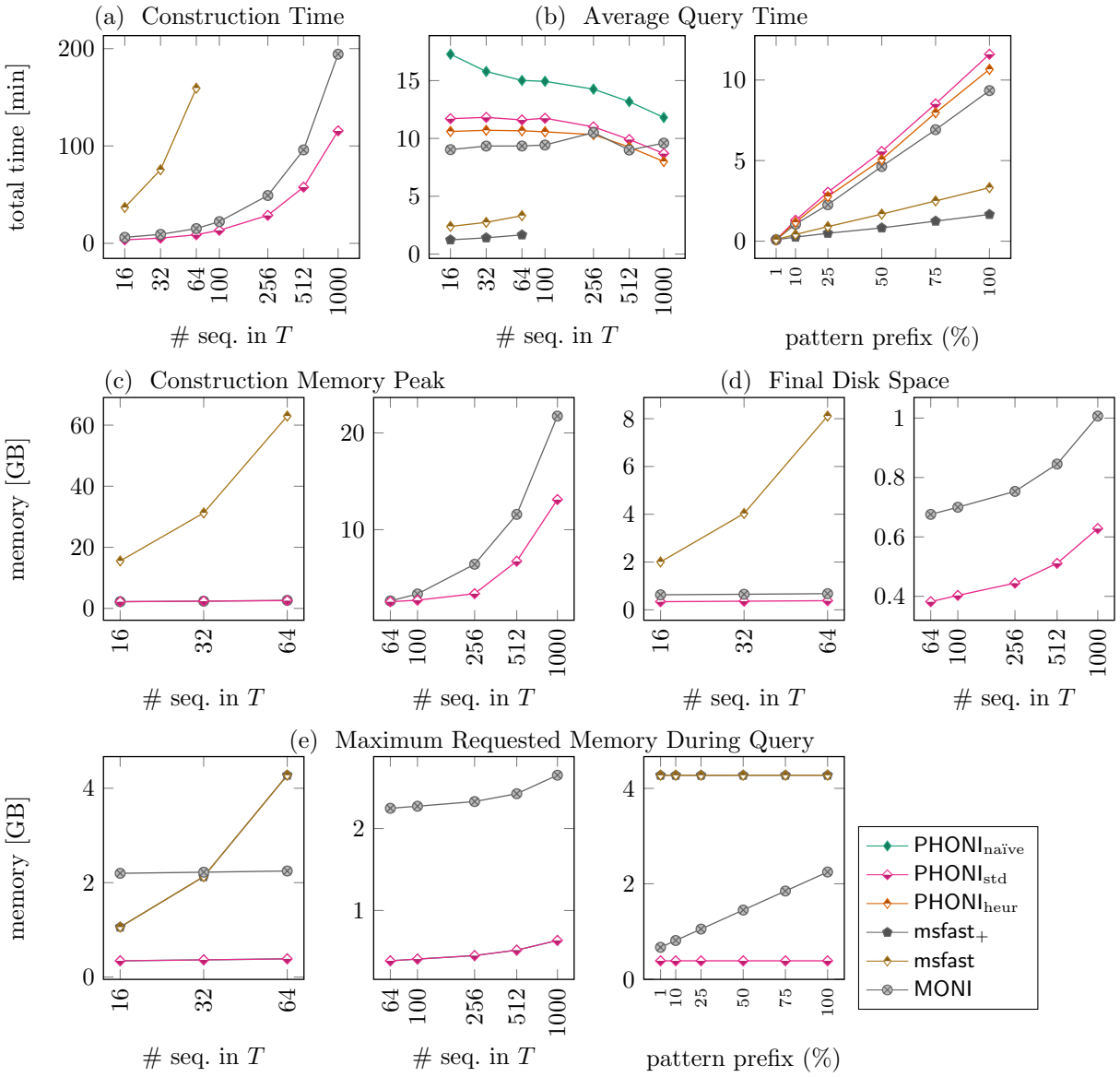

Figure 2: The time **(a)** and memory **(c)** needed to build PHONI$_{\text{naïve}}$, PHONI$_{\text{std}}$, MONI and msfast, and their final sizes **(d)**; the query times **(b)** as a function of dataset size and pattern length; and the memory used for queries **(e)**. The dataset used for testing prefixes of queries consisted of 64 chromosome 19s.

[3] H. Bannai, T. Gagie, and T. I, "Refining the r-index," *Theor. Comput. Sci.*, vol. 812, pp. 96–108, 2020.

[4] T. Gagie, G. Navarro, and N. Prezza, "Fully functional suffix trees and optimal text searching in BWT-runs bounded space," *J. ACM*, vol. 67, pp. 1–54, 2020.

[5] M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher, "MONI: A pangenomics index for finding MEMs," Submitted October 2020.

[6] M. Oliva, F. Milicchio, K. King, G. Benson, C. Boucher, and M. Prosperi, "Portable nanopore analytics: Are we there yet?," *Bioinformatics*, vol. 36, pp. 4399–4405, 2020.

[7] H. S. Edwards et al., "Real-time selective sequencing with RUBRIC: Read until with basecall and reference-informed criteria," *Sci. Rep*, vol. 9, pp. 1–11, 2019.

[8] S. Kovaka, Y. Fan, B. Ni, W. Timp, and M. C. Schatz, "Targeted nanopore sequencing by real-time mapping of raw electrical signal with UNCALLED," *BioRxiv*, 2020.

[9] A. M. Taxt et al., "Rapid identification of pathogens, antibiotic resistance genes and plasmids in blood cultures by nanopore sequencing," *Sci. Rep*, vol. 10, pp. 1–11, 2020.

[10] D. Belazzougui, F. Cunial, and O. Denas, "Fast matching statistics in small space," in *Proc. SEA*, 2018, pp. 17:1–17:14.

[11] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv:1303.3997*, 2013.

[12] G. Navarro and A. Ordóñez, "Faster compressed suffix trees for repetitive text collections," *ACM J. Exp. Algorithmics*, vol. 21, pp. article 1.8, 2016.

[13] W. Rytter, "Application of Lempel–Ziv factorization to the approximation of grammar-based compression," *Theor. Comput. Sci.*, vol. 302, pp. 211–222, 2003.

[14] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Trans. Inf. Theory*, vol. 51, pp. 2554–2576, 2005.

[15] N. J. Larsson and A. Moffat, "Offline dictionary-based compression," in *Proc. DCC*, 1999, pp. 296–305.

[16] T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, and Y. Takabatake, "Rpair: Rescaling RePair with rsync," in *Proc. SPIRE*, 2019, pp. 35–44.

[17] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun, "Prefix-free parsing for building big BWTs," *Algorithms Mol. Biol.*, vol. 14, pp. 13, 2019.

[18] A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, and G. Manzini, "Efficient construction of a complete index for pan-genomics read alignment," *J Comput. Biol.*, vol. 27, pp. 500–513, 2020.

[19] T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, L. Seelbach Benkner, and Y. Takabatake, "Practical random access to SLP-compressed texts," in *Proc. SPIRE*, 2020, pp. 221–231.

[20] P. Dinklage, J. Fischer, A. Herlez, T. Kociumaka, and F. Kurpicz, "Practical performance of space efficient data structures for longest common extensions," in *Proc. ESA*, 2020.

[21] L. Ilie, G. Navarro, and L. Tinta, "The longest common extension problem revisited and applications to approximate string searching," *J Discrete Algorithms*, vol. 8, pp. 418–428, 2010.

[22] J. Fischer, T. I, and Dominik Köppl, "Deterministic sparse suffix sorting in the restore model," *ACM Trans. Algorithms*, vol. 16, pp. 50:1–50:53, 2020.

[23] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda, "Fully dynamic data structure for LCE queries in compressed space," in *Proc. MFCS*, 2016.

[24] G. Navarro, *Compact data structures: A practical approach*, Cambridge University Press, 2016.