Fast Matching Statistics for Sets of Long Similar Strings

Zsuzsanna Lipták[1], Martina Lucà[1], Francesco Masillo[1], and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Verona, Italy
zsuzsanna.liptak@univr.it
martina.luca@studenti.univr.it
francesco.masillo@univr.it
[2] Department of Computer Science, University of Helsinki, Finland
simon.puglisi@helsinki.fi

Abstract. Matching statistics (*MS*) computation is at the heart of numerous bioinformatics applications, from read alignment to computing phylogenies of a set of genomes or even speeding up the computation of core data structures on collections of genomes. Many of these datasets have the property of being highly similar to the reference, which itself, however, may not be very repetitive. Some heuristics based on sequence-to-sequence similarity have already been studied in [Lipták et al., Alg. Mol. Biol. 2024], leading to a significant speedup in the computation of the matching statistics. In this paper, we introduce a new heuristic that further speeds MS computation. The core idea is to take advantage of existing similarities between the input sequences and the reference. We give an implementation making use of this heuristic, which also allows the use of multiple threads to parallelize *MS* computation. We give an experimental evaluation of our tool, `LRF-ms`, comparing it to other *MS* computation tools, on publicly available genomic datasets, and show that it is the fastest when the collection of genomes is highly similar to the reference string, while keeping a comparably low memory footprint.

Keywords: matching statistics, suffix array, parallel algorithms, LCP-array

1 Introduction

Given two strings S and R, over the same alphabet Σ, the matching statistics of S relative to R consists of an array of length $|S|$ whose ith entry contains the length of the longest substring of S starting in position i which has an occurrence also in R. Matching statistics were introduced by Chang and Lawler [11] in 1994 as an algorithmic tool for approximate pattern matching. It has since found many applications, for example, in DNA chip design [37], computation of string kernels [40,34], whole-genome phylogenies [12,44], and detection of SNVs or sequencing errors in read collections [35]. The classic book by Gusfield [21] contains several applications of matching statistics, among these longest common substrings, exact matches, longest prefix matching, and several others. Matching statistics have recently been used as a bridge to faster computation of the suffix array and Burrows-Wheeler Transform (BWT) [9] of string collections [27,28,30].

In contemporary genomics, a common type of dataset consists of many similar copies of essentially the same string. Most recent sequencing projects no longer aim to identify the genomic sequence of a species, but to identify the biological variation of individuals of a given species, such as the 100K Human Genome Project [43], the 1001 Arabidopsis Project [42], and the 3,000 Rice Genomes Project (3K RGP) [39]. Thus, the typical type of biological dataset no longer consists of one very long sequence

Zsuzsanna Lipták, Martina Lucà, Francesco Masillo, Simon J. Puglisi: Fast Matching Statistics for Sets of Long Similar Strings, pp. 3–15.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5 © Czech Technical University in Prague, Czech Republic

(or even one genome consisting of several chromosomes), but of a large number of highly-similar seqeuences.

Recently, a subset of the current authors introduced two heuristics for computing the matching statistics, which proved highly effective, and were therefore suggested to be of independent interest [27,28]. In this paper, we explore these heuristics and add another, which we show to be even more successful in speeding up computation. All of these heuristics exploit the fact that most datasets are highly repetitive *in a particular manner*, namely they consist of many similar copies of essentially the same string. Thus, while the collection of strings S is very similar to the reference string R, the reference string itself may not necessarily be very repetitive. This reflects itself on a fairly low maximal LCP-value (for precise definitions, see Section 2) on the reference string, while resulting in very long matches between S and R.

We implemented this heuristic in a tool for matching statistics computation. We compare our tool, `LRF-ms`, to other publicly available tools for matching statistics computation, on different datasets with varying properties. Surprisingly, we find that our tool outperforms the others by orders of magnitude w.r.t. running time, while also using less or comparable amount of peak memory.

The paper is organized as follows. In Section 2, we introduce the necessary definitions and terminology, in Section 3, we briefly recall matching statistics computation using the suffix array, followed by a presentation of the different heuristics. In Section 4, we give details of our implemention, followed by the results of the experiments in Section 5, comparing the different heuristics, and comparing our tool to competitor tools. We close with future work in Section 6.

2 Basics

Let Σ be an ordered alphabet of size σ. A string T over Σ is a finite sequence of characters from Σ. The ith character of T is denoted $T[i]$, its length is $|T| = n$, and $T[i..j]$ denotes the substring $T[i] \cdots T[j]$. If $i > j$, then $T[i..j]$ is the empty string ϵ. The suffix $T[i..] = T[i..n]$ is referred to as the ith suffix $suf_i(T)$, and $T[..i] = T[1..i]$ is the ith prefix $pref_i(T)$. When T is clear from the context, we write suf_i for $suf_i(T)$. We assume that the last character of T is the *sentinel character*, denoted \$, which does not occur elsewhere in the string and is assumed to be smaller than all characters from Σ. Note that we index strings from 1.

The *suffix array* SA of a string T is a permutation of the set $\{1, \ldots, n\}$ such that $SA[i] = j$ if $suf_j(T)$ is the ith in lexicographic order among all suffixes. For a substring Y of T, all suffixes prefixed by Y appear contiguously in SA; the interval $[s, e]$ of the SA containing all occurrences of Y is called *Y-interval* or *SA-range of Y*. It is well known that the suffix array can be computed in linear time [22,24,25] (see also the classic survey [36] and [3,7] for more recent overviews). Some recent algorithms include [32,31,4,19,26], with SA-IS [32] by far the most popular linear-time SACA, being both simple and fast in practice.

The *inverse suffix array* ISA is the inverse permutation of SA, namely, for all $1 \leq i \leq n$, $ISA[SA[i]] = i$. The *longest common prefix* (*lcp*) of two strings T and S is the longest string U which is a prefix of both T and S. The *longest-common-prefix array* LCP is another array closely related to the SA. It is given by: $LCP[1] = 0$, and for $i > 1$, $LCP[i]$ is the length of the longest common prefix of the two suffixes $suf_{SA[i-1]}$ and $suf_{SA[i]}$, which are consecutive in the SA. The LCP-array can be also be computed in linear time [23].

Let R and S be two strings, and let us denote the sentinel character of R as # and of S as \$, where # < \$. The *matching statistics* [11] MS of S w.r.t. R is an array of length $|S|$, whose entries are integer pairs defined as follows. For $1 \leq i \leq |S|$, let U_i be the longest prefix of suffix $suf_i(S)$ which occurs as a substring in R; we refer to U_i as the *matching factor* at position i. Then $MS[i] = (p_i, \ell_i)$, where $\ell_i = |U_i|$, and p_i is an occurrence of U_i in R if $U_i \neq \epsilon$, and $p_i = -1$ otherwise.

For an integer array A of length n and an index i, the *previous smaller values* PSV and *next smaller values* NSV are defined as follows: $PSV(A, i) = \max\{i' < i \ : \ A[i'] < A[i]\}$, $NSV(A, i) = \min\{i' > i \ : \ A[i'] < A[i]\}$, where $\min \emptyset = -\infty$ and $\max \emptyset = +\infty$. It is known that there is a data structure of size $n \log(3 + 2\sqrt{2}) + o(n)$ bits which answers both PSV and NSV queries in constant time and can be built in $\mathcal{O}(n)$ time [15]. A similar type of query, which we refer to as *extended PSV* and *extended NSV*, can be defined as follows: Let x be an integer and define $PSV(A, i, x) = \max\{i' < i \ : \ A[i'] < x\}$ and $NSV(A, i, x) = \min\{i' > i \ : \ A[i'] < x\}$, the previous respectively next smaller values *with respect to* x. As shown in [10], there is a data structure of size $16n/(2^B)$ bytes, which can be built in time $\mathcal{O}(n)$ and can answer these queries in $\mathcal{O}(B \log \frac{n}{B})$ time, where B is a parameter.

Given an integer array A of length n and two indices $1 \leq i \leq j \leq n$, a *range minimum query (RMQ)* on A is defined as $RMQ(A, i, j) = \arg\min\{A[k] \leq i \leq k \leq j\}$. Thus, an RMQ-query returns some position of the minimum value in the subarray $A[i..j]$. A data structure capable of answering RMQ-queries in constant time can be built in linear time and takes $2n + o(n)$ bits of space [14].

3 Matching Statistics computation

In the original publication introducing the matching statistics [11], Chang and Lawler gave a construction algorithm using the suffix tree of R, with space usage $\mathcal{O}(|R|)$ and running time $\mathcal{O}(|R| + |S| \log \sigma)$. Since then, it has been shown how to construct the matching statistics efficiently using compressed text indexes instead of the suffix tree [17,1,34,6].

In this section, we will first recall how to compute the matching statistics of a string S w.r.t. a reference string R using the suffix array. Then we will briefly explain two heuristics introduced in [27,28] (the MaxLCP and Block heuristics) and present a refinement which we refer to as LRF-heuristic. As we will see in Section 5, this latter heuristic far outperforms our previous heuristics, as well as all competing tools.

3.1 Plain version

In order to compute the matching statistics of S w.r.t. R, we will use SA_R, ISA_R, LCP_R, and the data structure of [10] (a heap-type tree) for extended *PSV-NSV*-queries on LCP_R (see Sec. 2). Constructing these data structures takes overall $\mathcal{O}(|R|)$ time and $\mathcal{O}(|R|)$ space.

In the following, we will use the terminology from [27,28]:

Definition 1 ([27,28]). *For a character c and a string Y, the computation of the Yc-interval from the Y-interval is called a* right extension *and the computation of the Y-interval from cY-interval is called a* left contraction.

A left contraction corresponds to following an implicit suffix link in the suffix tree of the text. In the following lemma, we show how to compute a left contraction using our data structures.[1]

Lemma 2. *Let cY be a substring of string T with SA-range $[s, e]$ and let $s' = ISA[SA[s] + 1]$ and $e' = ISA[SA[e] + 1]$. Then the SA-range of Y is $[x, y]$, where $x = PSV(LCP, s', |Y|)$ and $y = NSV(LCP, e', |Y|) - 1$.*

Proof. Let $i = SA[s]$ and $j = SA[e]$, thus both $suf_i(T)$ and $suf_j(T)$ have cY as prefix. Therefore, the following suffixes, $suf_{i+1}(T)$ and $suf_{j+1}(T)$ have Y as prefix, and thus both $s' = ISA[SA[s] + 1] = ISA[i + 1]$ and $e' = ISA[SA[e] + 1] = ISA[j + 1]$ lie in the SA-range $[x, y]$ of Y. Since $s \leq e$, thus $suf_i(T) \leq suf_j(T)$, and since both start with the same character c, therefore also $suf_{i+1}(T) \leq suf_{j+1}(T)$, implying $s' \leq e'$. Since both have prefix Y, it thus holds for all $s' \leq k \leq e'$ that $suf_k(T)$ has Y as prefix, i.e. k is an occurrence of Y. However, there could be other occurrences of Y whose corresponding suffixes are lexicographically smaller than suf_{i+1} resp. greater than suf_{j+1}. Now notice that for any two occurrences k, k' of Y, it holds that $lcp(suf_k(T), suf_{k'}(T)) \geq |Y|$. Therefore, the first position in which the LCP-value falls below $|Y|$ gives the extremities of the Y-interval of SA. Since $LCP[k]$ is defined as the *lcp* between the suffixes in position k and $k - 1$ of the SA, we get that $x = PSV(LCP, s', |Y|)$ gives us the position in the SA of the lexicographically smallest occurrence of Y, i.e. the one listed first in the SA, while $y = NSV(LCP, e', |Y|) - 1$ gives us the last one. □

The computation of each entry of the matching statistics can thus be seen as consisting of two distinct phases. Given entry $MS[i]$ and the U_i-interval $[s_i, e_i]$ of SA_R, we can split the computation of the next entry $MS[i + 1]$ into:

1. *left contraction:* Compute the SA-interval $[x, y]$ of U', where $U_i = cU'$ for character $c = S[i]$, and
2. *right extension:* Extend the factor U' to the right as long as the current prefix of $suf_i(S)$ occurs in R. The result is the range $[s_{i+1}, e_{i+1}]$ of the matching factor U_{i+1} (of which U' is a prefix).

Phase 1. Given the range $[s_i, e_i]$ for position i, we want to compute the range $[x, y]$ of factor U'. By Lemma 2, this interval equals

$$[x, y] = [PSV(LCP_R, s', |Y|), NSV(LCP_R, e', |Y|) - 1],$$

where $s' = ISA_R[SA_R[s_i] + 1]$ and $e' = ISA_R[SA_R[e_i] + 1]$.

Phase 2. The second phase can be done by searching for the longest prefix of $suf_i(S)$ which occurs in R, with two applications of binary search on SA_R, one for the left extremity of the interval, and one for the right one. The range is a pair of indices $[s_{i+1}, e_{i+1}]$, where $SA[k]$, for $s_{i+1} \leq k \leq e_{i+1}$, are all occurrences of U_{i+1} in R. We then set $MS[i + 1] = (s_{i+1}, |U_{i+1}|)$.

The total worst-case running time of the algorithm for computing the matching statistics is $\mathcal{O}(|S| \log |R|)$, due to the binary search in the right extension phase, and choosing the parameter $B = \mathcal{O}(1)$ for the extended PSV-NSV data structure.

[1] Note that this lemma is slightly different from the formula given in [27,28].

3.2 The *LRF*-heuristic

The two heuristics given in [27,28] both aimed at speeding up *MS*-computation using information stored in the *LCP*-array.

1. MaxLCP-heuristic: Compute $L = \max\{LCP[j] \mid 1 \leq j \leq |R|\}$, and for each $1 \leq i \leq |S|$ compare ℓ_i with L. If $\ell_i - 1 > L$, then this means that we only need a left contraction, i.e. $ISA[p_i + 1]$, and no right extension is necessary, leading to $MS[i+1] = (p_i + 1, \ell_i - 1)$. This is because the interval containing the matching factor U_{i+1} is a singleton interval in the SA_R (in other words, it can be viewed as a leaf branch in the suffix tree of R). Hence, there are no further occurrences in R of $S[i+1..\ell_i - 2]$ other than $p_i + 1$.
2. Block-heuristic: This is a further refinement of the MaxLCP-heuristic: Divide the *LCP*-array into blocks of size b and compute the maximum of each block. Locate the block that contains $SA[p_i]$ and compare ℓ_i to this value.

Here, we propose to further refine the second heuristic by noticing that by taking the extreme value of $b = 1$, one ends up having to query the *LCP* array directly, or rather, two consecutive values of the *LCP*-array (see Observation 1). But the access pattern to the *LCP* array for contiguous text positions is not cache-friendly, because the *lcp* information is in *SA* order rather than in text-order. Therefore, we introduce the following data structure:

Definition 3. *Let T be a string of length n over Σ. Define the* longest repeated factor array *LRF: $LRF[i] = \max\{|Y| : Y$ occurs both in position i and some $k \neq i$ in $T\}$.*

Note that the *LRF*-array is distinct from the well-known *LPF*-array [33], used, e.g. for computation of the LZ77 parsing of a string T, as that array contains the length of the longest *previous* factor, i.e. the longest factor occurring both in i and in some position $k < i$, while here, we want the longest factor occurring both in i and in *any* position $k \neq i$. It is easy to compute the *LRF*-array in linear time, using the following observation:

Observation 1 *For all i, $LRF[i] = \max\{LCP[ISA[i]], LCP[ISA[i]+1]\}$.*

Proof. Since $LCP[k]$ gives the longest common prefix of the suffix in position k and in position $k-1$, it follows that the longest repeated factor of the suffix which is position k is the maximum of $LCP[k]$ and $LCP[k+1]$, implying the claim. □

Since the *LCP*-array can be computed in linear time, e.g. using the algorithm by Kasai et al. [23], we can also compute the *LRF*-array in linear time. Alternatively, we can use a linear-time algorithm for computing the *PLCP*-array (*permutated LCP-array*) of [22], which is defined by $PLCP[i] = LCP[ISA[i]]$, i.e. the *LCP*-values given in text-order.

Now, checking whether we are in a singleton interval inside a leaf branch can be done by comparing $\ell_i - 1$ to $LRF[p_i + 1]$. The strength of this approach can be appreciated when we have consecutive positions in S that satisfy this check, leading to a very cache-efficient sequential access pattern in the *LRF*-array. A brief pseudocode for this heuristic is given in Algorithm 1.

An experimental evaluation of this heuristic against the other two showed that the *LRF*-heuristic far outperformed both the MaxLCP heuristic and the Block heuristic for all block sizes, see Figure 1.

Algorithm 1: LRF-heuristic

```
while ℓ_i − 1 > LRF[p_i + 1] do
    MS[i + 1] = (p_i + 1, ℓ_i − 1)
    i++
end
```

4 Implementation details

To compute SA_R, LCP_R, and $PLCP_R$ we use a highly engineered version of the SA-IS algorithm [32], `libsais` [20]. We implemented the data structure for *PSV-NSV* and range minimum queries on the LCP_R, based on the work of Cánovas and Navarro [10], setting $B = 7$. We preprocessed the data to have only characters in the set $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. We did this by assigning to IUPAC characters one among their possible corresponding nucleotides. We use this assumption in several of our experiments later described in this section, and to allow all tools to run without crashing (as some assume that all characters present in the dataset S are also present in the reference R).

We implemented the two heuristics introduced in [27,28] (see Sec. 3.2). For the second one, which involves choosing a block size for computing local maxima, we tested $b = 512, 1024, 2048, 4096$. In a further step, we parallelized our LRF-based implementation by splitting the collection to evenly distribute the number of sequences to give to each thread.

We tested various other approaches to further increase efficiency, including:

- a k-mer lookup table ($k = 8$), where we have a precomputed range for every possible k-mer in R. With the assumption of $\sigma = 4$, we can use just two bits to represent each letter of a k-mer, allowing us to use a lookup table with 2^{16} entries, when choosing $k = 8$. We access the lookup table when ℓ drops below the chosen value of k;
- a cache, implemented as a hashtable, storing resulting ranges during right-extensions. The hashtable has (sp, ep, l, c) as key, where sp and ep are the start, respectively the end, of the initial binary search range, l is the length of the currently matched prefix, and c is the character to be found after applying the binary search procedure. The values stored for each key are (sp', ep'), which are the start, respectively the end, of the new range of the binary search. We also tried to store in l the value of c in base 4, taking only the two most significant bits of l. We set 100000 elements as the maximum capacity of these hashtables;
- we attempted to trade off some running time by approximating the LCP array using only $\log \log n$ bits, so effectively reducing the integers to two bytes instead of four. This effect is also reflected on the RMQ and *PSV-NSV* data structures.
- we implemented the improved version of the binary search procedure on SA, which uses RMQ queries on LCP_R, allowing to compute right-extensions in overall $\mathcal{O}(|S| + \log |R|)$ time. For the RMQ data structure, we tested all the available plug-in options in the SDSL [18], plus the data structure based on [10]. The latter achieved the best performance in our tests, probably because it was already in memory due to suffix link traversal.

These four approaches did not lead to further significant improvement in time and/or space (data not shown).

5 Experiments

We implemented our algorithm for computing the *MS* in `C++`, resulting in the tool `LRF-ms`. Our implementation is available at `https://github.com/fmasillo/lrf-ms`. The experiments were conducted on a desktop equipped with 64GB of RAM DDR4-3200MHz and an Intel(R) Core(R) i9-11900 @ 2.50GHz (with turbo speed @ 5GHz) with 16 MB of cache. The operating system was Ubuntu 22.04 LTS, the compiler used was `g++` version 11.3.0 with options `-std=c++20 -O3 -march=native` enabled.

5.1 Datasets

In our experiments, we used four publicly available datasets. The first dataset, which we refer to as `sars-cov2`, contains copies of SARS-CoV2 genomes taken from the COVID-19 Data Portal. The second dataset (`chr19`) consists of copies of the Human Chromosome 19 from the 1000 Genomes Project [41]. The third dataset (`salmonella`) is a collection of assembled genomes of Salmonella enterica, downloaded from NCBI Pathogens website[2]. Lastly, the fourth dataset, which we refer to as `rice`, contains variants of Chromosome 1 of the Nipponbare reference sequence (rice) downloaded from RiceVarMap [45] as vcf files[3]. For each dataset we have a collection of sequences of length 1 GB. Some additional metadata can be found in Table 1.

name	no. sequences	ref. seq. length	max LCP_R	mean LCP_R
`sars-cov2`	36 201	29 783	17	6
`chr19`	17	59 126 939	3 099 999	81 379
`salmonella`	216	4 506 055	145	10
`rice`	23	43 992 113	12 893	27

Table 1. Datasets used in the experiments. In column 2 the number of sequences in the dataset, in column 3 the reference sequence length, in column 4 the maximum value in LCP_R, and in column 5 the average of the values in LCP_R. The total dataset has size 1 GB and σ is always 4.

5.2 Other tools

We compared our implementations to the following four tools:

1. `indexed_ms` [13], a tool computing a compact version of matching statistics. At its core, it uses a compressed suffix tree of the forward and reverse reference string along with some further annotation containing a list of nodes of the suffix tree topology that are maximal repeats. This tool outputs a compact version of matching statistics. We ran the experiments with the following flags enabled: `-load_cst 1 -load_maxrep 1 -lazy_wl 1 -nthreads 16`.
2. `MONI` [38], builds an enhanced r-index [16,5] on the reference string. Other than the Burrows-Wheeler Transform of R and the suffix array samples at run-boundaries, the tool also computes some additional $\mathcal{O}(r)$ space information, called thresholds, to enable fast recovery in case of a mismatch. It also makes use of a grammar to allow fast random access to the original text. We ran the experiments with the following flags enabled: `-t 16 -g shaped`.

[2] `https://www.ncbi.nlm.nih.gov/pathogens/isolates/` with isolate number PDS000065758.863

[3] `https://ricevarmap.ncpgr.cn/download/`

3. `PHONI` [8], is another tool based on the r-index, but it does not need thresholds. This is done via a refinement of the process of finding the length of the matches with a grammar enabling fast longest common extension (LCE) queries. This enables a single scan of the pattern to compute the matching statistics. We ran the experiments with the following flag enabled: `-g shaped`.
4. `AUG-PHONI` [29], a tool based on `PHONI`. It stores thresholds for LCE, bypassing queries to the grammar, resulting in a faster computation, with a slight increase in space. We ran the experiments with the following flag enabled: `-g shaped`.

5.3 Running time and peak memory results

In this section, we first compare the different implementations of our algorithm using different heuristics. Then, we will show how the best implementation of our tool compares to the other state-of-the-art tools for computing matching statistics.

Comparison of different heuristics. Looking at Figure 1, we can see that, overall, the fastest implementation is the one making use of the LRF heuristic. More in detail, in both `chr19` and `rice` datasets we can see a large improvement between using the LRF heuristic and the fastest of the block-based heuristic, taking approximately a third of the time. In `sars-cov2` and `salmonella` datasets, we can see that using even the simplest heuristic, i.e. taking only the maximum value in the LCP, leads to a big speedup w.r.t. not using a heuristic at all. This is because, as reported in Table 1, the max LCP value is quite small, enabling a lot of skipping in the sequence of left-contractions even without using the exact value of LCP/LRF associated with p_i. `sars-cov2` is the only dataset in which we have a slight worsening due to the overhead of making a cache-miss when accessing an entry of LRF instead of comparing to a variable holding the max LCP.

Comparison with other tools. We have divided the comparison of `LRF-ms` and the other state-of-the-art tools based on whether they support multi-threading when computing MS. In Figure 2, we compare `LRF-ms` with `indexed_ms` and `MONI`. Over the four datasets, `LRF-ms` is always the winner, followed by `indexed_ms`. On `chr19` and `rice` where the difference is largest, `LRF-ms` is around three times faster, while on `sars-cov2` and `salmonella` is around 1.5 to 2 times faster than `indexed_ms`. `MONI` takes, on average, 6.5 times more time than `LRF-ms`. Moving to Figure 3, we compare to `PHONI` and `AUG-PHONI`, which allow only for serial computation of MS. `AUG-PHONI` and `PHONI` are 32-52 respectively 44-382 times slower than `LRF-ms`.

The running results reported here are in line with the expectations on the indexes used for R. Our implementation makes use of mostly uncompressed data structures, so the overhead for each step of the computation is set to a minimum. `indexed_ms` uses suffix tree topology over the Burrows-Wheeler Transform (BWT) [9] of R, which is supposedly smaller than the total size of our data structures. Using compact data structures also implemented in SDSL, this tool is always between `LRF-ms` and the rest of the tools when comparing running time. `MONI`, `PHONI`, and `AUG-PHONI` are all based on the r-index, and therefore, due to the compressed nature of such index, the running time is affected negatively. In general, BWT-based approaches require a procedure called LF-mapping based on rank queries. LF-steps are known to be not very cache-efficient due to unpredictable jumps in the data structure. Here, extra work must be performed to recover from a mismatch during an LF-step, lacking the explicit suffix

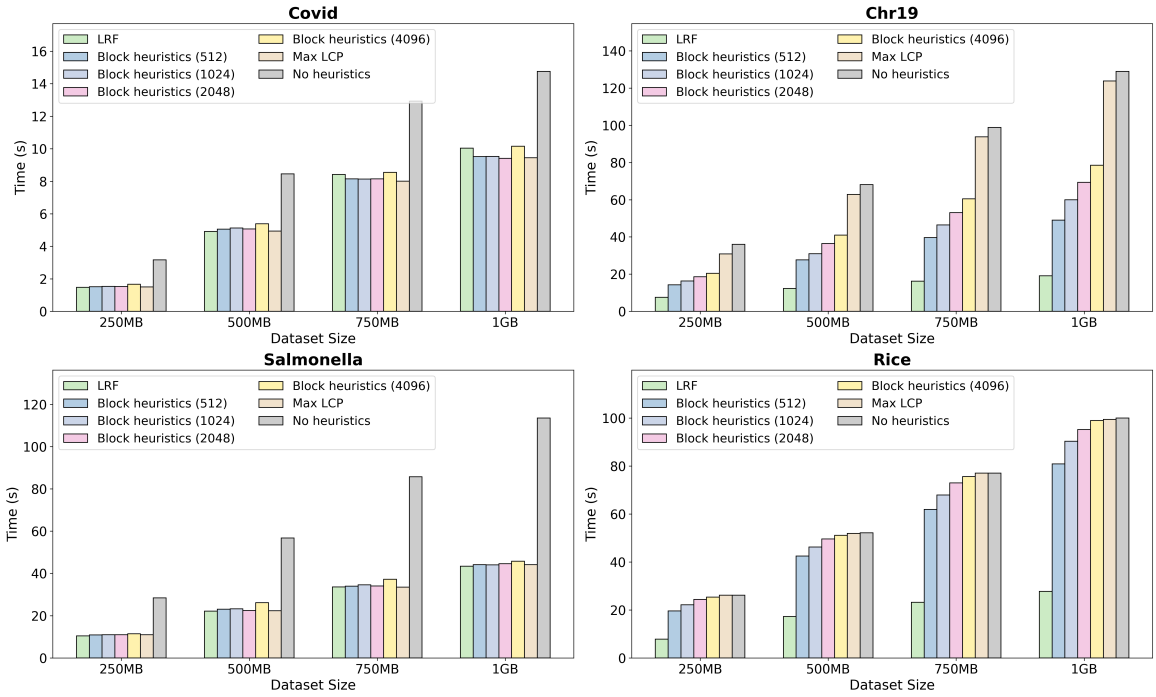

Figure 1. Comparison of different heuristics using as a base implementation the plain binary search version. Every subfigure represents a different dataset, and on the x-axis the size of the dataset is shown, while on the y-axis the running time is reported in seconds.

tree topology. On the other hand, both theoretically and experimentally, the index takes $\mathcal{O}(r)$ space, where r is the number of runs in the BWT of R, a well-known repetitiveness measure. (Here a *run* is defined as a maximal substring consisting of the same character.) This ensures the possibility of computing the index for extremely large references, as reported in each paper.

Looking at memory consumption, we divided the comparison into two phases: indexing of R and computation of *MS*. In Table 2, the peak memory in kilobytes is measured for the indexing phase. Across the four datasets, our tool uses the least memory overall. On `sars-cov2`, `LRF-ms` uses less than half the memory compared to the other tools due to $|R|$ being extremely small. Likewise, on `salmonella`, we take less space than the competitors, a third less in general. On `chr19` and `rice`, we take almost the same space as `indexed_ms` which is around a third less than the other three tools. Moving to Table 3, we recorded the peak memory of *MS* computation on 1GB of data for each dataset, using 16 threads if the tool can run in multi-threaded mode. We consistently use an additional 100MB of memory w.r.t. the indexing phase due to buffering I/O operations, which are the main bottleneck on `sars-cov2` and `chr19` datasets. On average, `indexed_ms` seems to be settling around 650 MB of internal memory, while `MONI` has a particular behaviour, probably related to the length of the individual sequences in the dataset. For example, on `chr19`, if we set the execution to single-threaded, the memory consumption drops to 1300 MB (data not shown) from around 13 GB when using 16 threads. Overall, in our setting, `LRF-ms` performs comparably well for space consumption, for example, being the lightest on `salmonella` and close to `indexed_ms` on `rice`.

In general, indexing a moderate-size reference with compressed or uncompressed data structures leads to similar memory consumption (due to the incompressibility of

the reference). When computing the *MS*, `indexed_ms` seems to take more space the more threads are allocated, as for `MONI`, but only around 40 MB per thread. Looking at r-index-based tools, even though the space is proportional to r, a single sequence is not so repetitive. The complex machinery used in these tools takes more space than our set of data structures when loaded in memory, as can be seen for every dataset except `sars-cov2`.

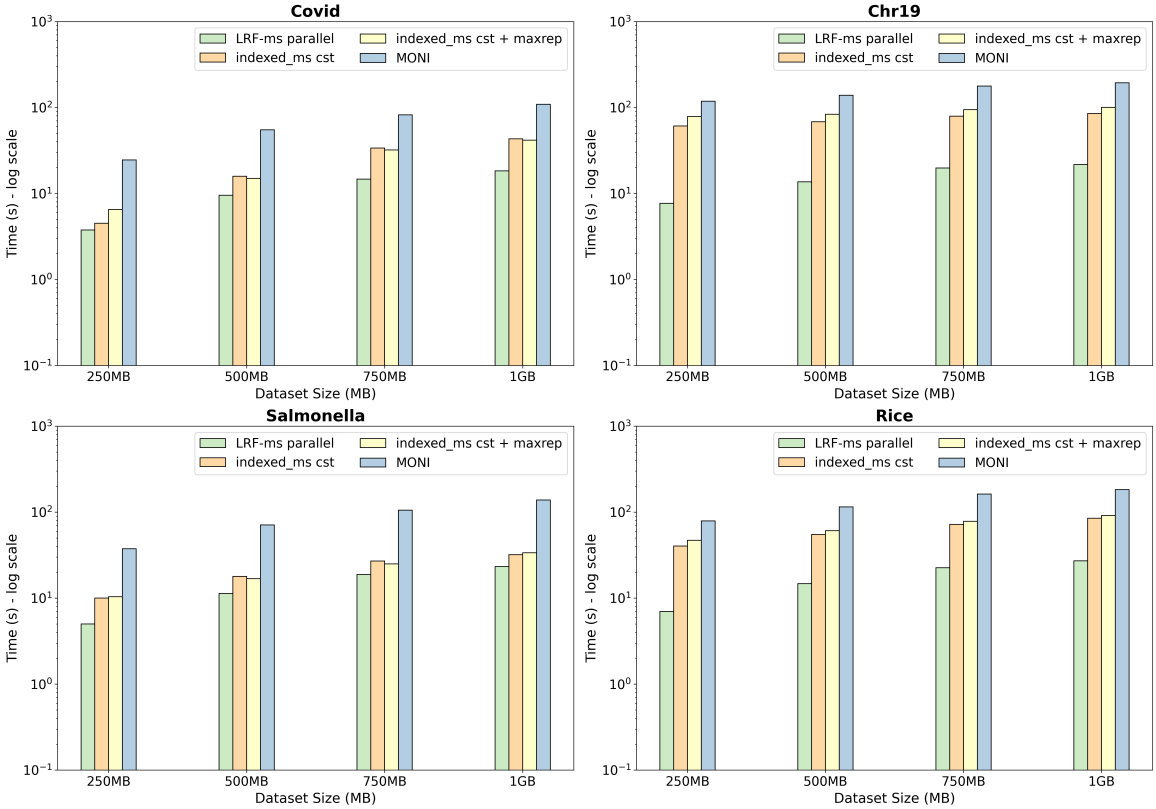

Figure 2. Comparison of different multi-threaded tools using 16 threads. Each subfigure represents a different dataset. We give the dataset size on the x-axis and the time (in seconds) on the y-axis (log-scale).

	LRF-ms	indexed_ms	MONI	PHONI	AUG-PHONI
sars-cov2	**4 992**	13 488	15 052	14 524	14 640
chr19	**991 240**	1 003 788	1 428 540	1 428 536	1 428 536
salmonella	**80 380**	102 996	112 760	112 760	112 760
rice	727 924	**709 864**	1 039 352	1 039 344	1 039 348

Table 2. Peak memory (in KB) of the different tools for indexing the reference string R. The lowest value for each row is highlighted in bold.

6 Conclusion

In this paper, we have presented a new simple and practical heuristic for speeding up the computation of matching statistics. This new heuristic exploits the similarity between the reference string and the individual strings in the collection. Compared to the previous best heuristic, our approach more often avoids the costly operation

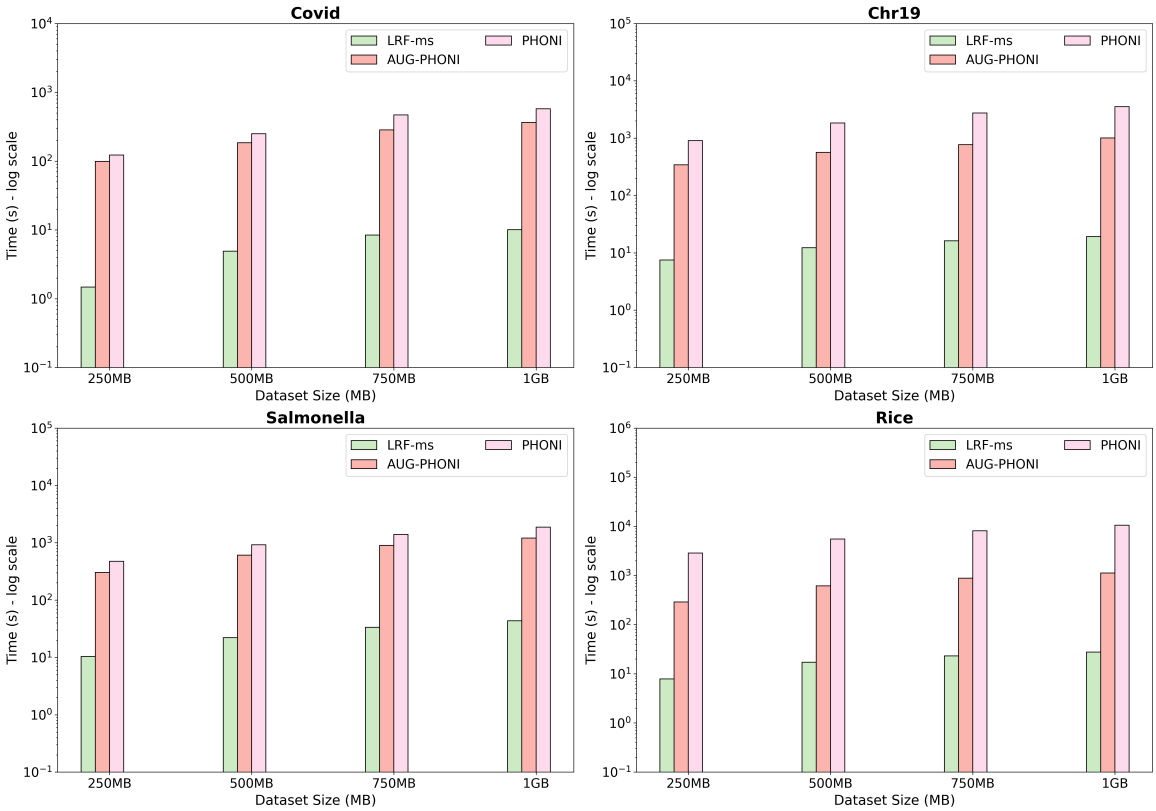

Figure 3. Comparison of different single-threaded tools. Each subfigure represents a different dataset. We give the dataset size on the x-axis and the time (in seconds) on the y-axis (log-scale).

	LRF-ms	indexed_ms	MONI	PHONI	AUG-PHONI
sars-cov2	90 716	617 820	17 044	**14 168**	14 180
chr19	1 077 920	**683 504**	13 078 520	2 175 716	2 393 404
salmonella	**166 552**	635 368	1 269 008	181 620	202 240
rice	814 900	**673 916**	10 859 492	1 628 136	1 816 560

Table 3. Peak memory (in KB) of the different tools for computing the matching statistics of the string collection w.r.t. R. The lowest value for each row is highlighted in bold.

of traversing a suffix link. The experimental evaluation against the state-of-the-art tools shows that our implementation `LRF-ms` is the winner in terms of running time, while having comparable space consumption.

Future work will focus on exploring the possibility of using the proposed heuristic on a more compact version of the text index of R, be it a compressed suffix tree or a variant of the r-index, called $\overline{r}$-index, which does not use backward search [2], possibly allowing a right-extension-like operation then followed by fast left-contractions.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. O. Ahmed, A. Baláz, N. K. Brown, L. Depuydt, A. Goga, A. Petescia, M. Zakeri, J. Fostier, T. Gagie, B. Langmead, G. Navarro, and N. Prezza: *r-indexing without backward searching.* CoRR, abs/2312.01359 2023.
3. J. Bahne, N. Bertram, M. Böcker, J. Bode, J. Fischer, H. Foot, F. Grieskamp, F. Kurpicz, M. Löbel, O. Magiera, R. Pink, D. Piper, and C. Poeplau: *Sacabench:*

Benchmarking suffix array construction, in Proc. of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019), vol. 11811 of Lecture Notes in Computer Science, Springer, 2019, pp. 407–416.

4. U. Baier: *Linear-time suffix sorting - A new approach for suffix array construction*, in Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), vol. 54 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 23:1–23:12.
5. H. Bannai, T. Gagie, and T. I: *Refining the* r*-index*. Theor. Comput. Sci., 812 2020, pp. 96–108.
6. D. Belazzougui, F. Cunial, and O. Denas: *Fast matching statistics in small space*, in Proc. 17th International Symposium on Experimental Algorithms (SEA), vol. 103 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 17:1–17:14.
7. T. Bingmann: *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools*, PhD thesis, Karlsruhe Institute of Technology, Germany, 2018.
8. C. Boucher, T. Gagie, T. I, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi: *PHONI: streamed matching statistics with multi-genome references*, in Proc. of the 31st Data Compression Conference (DCC 2021), IEEE, 2021, pp. 193–202.
9. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., DIGITAL System Research Center, 1994.
10. R. Cánovas and G. Navarro: *Practical compressed suffix trees*, in Proc. of the 9th International Symposium Experimental Algorithms, SEA 2010, vol. 6049 of LNCS, Springer, 2010, pp. 94–105.
11. W. I. Chang and E. L. Lawler: *Sublinear approximate string matching and biological applications*. Algorithmica, 12(4/5) 1994, pp. 327–344.
12. E. Cohen and B. Chor: *Detecting phylogenetic signals in eukaryotic whole genome sequences*. J. Comput. Biol., 19(8) 2012, pp. 945–956.
13. F. Cunial, O. Denas, and D. Belazzougui: *Fast and compact matching statistics analytics*. Bioinform., 38(7) 2022, pp. 1838–1845.
14. J. Fischer: *Optimal succinctness for range minimum queries*, in Proc. of the 9th Latin American Symposium on Theoretical Informatics (LATIN 2010), vol. 6034 of Lecture Notes in Computer Science, Springer, 2010, pp. 158–169.
15. J. Fischer: *Combined data structure for previous- and next-smaller-values*. Theor. Comput. Sci., 412(22) 2011, pp. 2451–2456.
16. T. Gagie, G. Navarro, and N. Prezza: *Fully functional suffix trees and optimal text searching in BWT-runs bounded space*. J. ACM, 67(1) 2020, pp. 2:1–2:54.
17. Y. Gao: *Computing matching statistics on repetitive texts*, in Proc. of the 32nd Data Compression Conference (DCC 2022), IEEE, 2022, pp. 73–82.
18. S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: Plug and play with succinct data structures*, in Proc. of the 13th International Symposium on Experimental Algorithms (SEA 2014), vol. 8504 of Lecture Notes in Computer Science, Springer, 2014, pp. 326–337.
19. K. Goto: *Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets*, in Proc. of the Prague Stringology Conference 2019, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019, pp. 111–125.
20. I. Grebnov: *Code for libsais*, `https://github.com/IlyaGrebnov/libsais`.
21. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
22. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction*. J. ACM, 53(6) 2006, pp. 918–936.
23. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings, vol. 2089 of Lecture Notes in Computer Science, Springer, 2001, pp. 181–192.
24. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time*. J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
25. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*. J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
26. Z. Li, J. Li, and H. Huo: *Optimal in-place suffix sorting*. Inf. Comput., 285(Part) 2022, p. 104818.

27. Zs. Lipták, F. Masillo, and S. J. Puglisi: *Suffix sorting via matching statistics*, in Proc. of the 22nd International Workshop on Algorithms in Bioinformatics (WABI 2022), vol. 242 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:15.
28. Zs. Lipták, F. Masillo, and S. J. Puglisi: *Suffix sorting via matching statistics.* Algorithms Mol. Biol., 19(1) 2024, pp. 11:1–11:18.
29. C. Martínez-Guardiola, N. K. Brown, F. Silva-Coira, D. Köppl, T. Gagie, and S. Ladra: *Augmented thresholds for MONI*, in Proc. of the 33rd Data Compression Conference (DCC 2023), IEEE, 2023, pp. 268–277.
30. F. Masillo: *Matching statistics speed up BWT construction*, in Proc. of the 31st Annual European Symposium on Algorithms (ESA 2023), vol. 274 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 83:1–83:15.
31. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, p. 15.
32. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction.* IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
33. E. Ohlebusch: *Bioinformatics Algorithms*, Oldenbusch Verlag, 2013.
34. E. Ohlebusch, S. Gog, and A. Kügel: *Computing matching statistics and maximal exact matches on compressed full-text indexes*, in Proc. of the 17th International Symposium on String Processing and Information Retrieval, SPIRE 2010, vol. 6393 of LNCS, Springer, 2010, pp. 347–358.
35. N. Philippe, M. Salson, T. Commes, and E. Rivals: *Crac: an integrated approach to the analysis of rna-seq reads.* Genome biology, 14 2013, pp. 1–16.
36. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39(2) 2007, p. 4.
37. S. Rahmann: *Fast and sensitive probe selection for DNA chips using jumps in matching statistics*, in Proc. of the 2nd IEEE Computer Society Bioinformatics Conference (CSB 2003), IEEE Computer Society, 2003, pp. 57–64.
38. M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher: *MONI: A pangenomic index for finding maximal exact matches.* J. Comput. Biol., 29(2) 2022, pp. 169–187.
39. C. Sun, Z. Hu, T. Zheng, K. Lu, Y. Zhao, W. Wang, J. Shi, C. Wang, J. Lu, D. Zhang, Z. Li, and C. Wei: *RPAN: rice pan-genome browser for 3000 rice genomes.* Nucleic Acids Research, 45(2) 2017, pp. 597–605.
40. C. H. Teo and S. V. N. Vishwanathan: *Fast and space efficient string kernels using suffix arrays*, in Proc. of the 23rd International Conference on Machine Learning (ICML 2006), vol. 148 of ACM International Conference Proceeding Series, ACM, 2006, pp. 929–936.
41. The 1000 Genomes Project Consortium: *A global reference for human genetic variation.* Nature, 526 2015, pp. 68–74.
42. The 1001 Genomes Consortium: *Epigenomic Diversity in a Global Collection of* Arabidopsis thaliana *Accessions.* Cell, 166(2) 2016, pp. 492–505.
43. C. Turnbull et al.: *The 100,000 genomes project: bringing whole genome sequencing to the NHS.* British Medical Journal, 361 2018.
44. I. Ulitsky, D. Burstein, T. Tuller, and B. Chor: *The average common substring approach to phylogenomic reconstruction.* J. Comput. Biol., 13(2) 2006, pp. 336–350.
45. H. Zhao, J. Li, L. Yang, G. Qin, C. Xia, X. Xu, Y. Su, Y. Liu, L. Ming, L.-L. Chen, et al.: *An inferred functional impact map of genetic variants in rice.* Molecular Plant, 14(9) 2021, pp. 1584–1599.