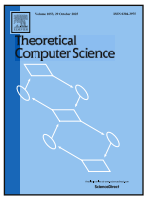
Theoretical Computer Science 1071 (2026) 115796

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs



Matching statistics—a survey

Zsuzsanna Lipták [a], Francesco Masillo [b], Simon J. Puglisi [c,*]

[a] Department of Informatics, University of Verona, Italy
[b] Department of Computer Science, Technical University of Dortmund, Germany
[c] Department of Computer Science, University of Helsinki, Finland

ARTICLE INFO

Section Editor: Seth Pettie
Handling Editor: Katie Harris

Keywords:
Matching statistics
Approximate string matching
String processing
Data structures
Data compression
Suffix tree

ABSTRACT

Given two strings S and R, the matching statistics of S with respect to R is an array of length $|S|$ whose ith entry encodes the longest prefix of the ith suffix of S that occurs in R. Introduced by Chang and Lawler in 1990 for approximate string matching, matching statistics have since found a variety of applications in computational biology, data compression, and string processing. In this article, we survey these applications, as well as the main ideas underlying the different algorithms for efficient construction of the matching statistics that have appeared in the last 30 years.

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 [1,2] as an algorithmic tool for approximate pattern matching. Since then, matching statistics have been applied to many other string processing tasks.[1] DNA chip design [3], computation of string kernels [4,5], whole-genome phylogenies [6,7], and detection of SNVs (single nucleotide variations) or sequencing errors in read collections [8]. Other applications to problems in string processing can be found in the classic book of Gusfield [9]: longest common substrings, exact matches, longest prefix matching, and several others. Recently, matching statistics have been used as a preprocessing step to achieve faster computation of the suffix array [10] and Burrows-Wheeler Transform (*BWT*) [11] of highly similar string collections [12–14]. Moreover, matching statistics are closely related to Lempel-Ziv factorization [15] and relative Lempel-Ziv factorization [16], and have been successfully applied there, as well.

In this survey, we give an overview of the different approaches to computing the matching statistics that have been developed in the last 30 years, and tour the applications mentioned above. The next section lays down notation and defines basic concepts that we use throughout the paper. In Section 3, we explain recent approaches to efficient storage of matching statistics, while in Section 4, we review different algorithms for matching statistics construction. In Sections 5 and 6, we describe how matching statistics can be applied to solving various string processing problems efficiently. We close with an overview and outlook in Section 7.

* Corresponding author.
E-mail address: puglisi@cs.helsinki.fi (S.J. Puglisi).

[1] We refer to Chang and Lawler's journal paper [2] as the main reference for matching statistics throughout, while noting here that the main ideas were already present in their earlier conference paper [1].

https://doi.org/10.1016/j.tcs.2026.115796
Received 5 August 2025; Received in revised form 8 January 2026; Accepted 29 January 2026
Available online 3 February 2026

2. Basics

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

The *rank* operation counts the number of occurrences of a specific character $c \in \Sigma$ up to a given position in an array A. Formally, $rank_c(A, i) = |\{j \in [1..i] \mid A[j] = c\}|$. The *select* operation is the inverse of the rank operation. It returns the position of the kth occurrence of a specific character. Formally, $select_c(A, k) = \min\{j \in [1..|A|] \mid rank_c(A, j) = k\}$. It is well known that *rank* and *select* queries can be answered in $\mathcal{O}(1)$ time on binary bitvectors (i.e. when $\sigma = 2$), after linear time preprocessing and using $o(n)$ bits of extra space [17,18].

Example 1. Let $B = 001010110111$, then $rank_1(B, 5) = 2$ and $rank_0(B, 5) = 3$, while $select_1(B, 4) = 8$ and $select_0(B, 3) = 4$.

The most commonly used data structure implementing $rank_c$ and $select_c$ on strings is the wavelet tree [19]. This data structure needs $n \log \sigma + o(n \log \sigma)$ bits of space in its plain representation, and allows answering both rank- and select-queries in $\mathcal{O}(\log \sigma)$ time.[2]

The suffix tree [20] is a classic data structure on texts which is able to efficiently answer many different kinds of string processing queries [9,21]. It uses linear space and can be built in linear time [20,22–24]. We give a brief summary; see Gusfield [9] for more details.

The *suffix tree* $ST(T)$ of a string T is the compact trie of the suffixes of T; it is a rooted tree whose edges are labeled by substrings of T (stored as two pointers into T), and every inner node of the tree has at least two children. The *label* $L(v)$ of a node v is the concatenation of the labels of the edges on the root-to-node path. There is a one-to-one correspondence between leaves of $ST(T)$ and suffixes of T, namely $leaf_i$ is the leaf whose label equals the ith suffix $T[i..]$ of T; i is called the *leaf number* of $leaf_i$. The *stringdepth* $sd(v)$ of a node v is the length of its label. Given a node u with parent v and an integer d s.t. $sd(v) < d \leq sd(u)$, one can further define the *locus* (u, d). There is a one-to-one correspondence between substrings of T and loci of $ST(T)$, namely, (u, d) corresponds to the unique string U which is the d-length prefix of $L(u)$. Every internal node except the root also has a so-called *suffix link*, which is a pointer to another node. If $L(v) = c\alpha$, with $c \in \Sigma$ and $\alpha \in \Sigma^*$, then $slink(v) = u$, with u the unique node such that $L(u) = \alpha$.

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 [15,25,26] (see also the classic survey [27] and [28,29] for more recent overviews). Some recent algorithms include [30–34], with SA-IS [30] by far the most popular linear-time suffix array construction algorithm, as it is both simple and fast in practice.

The *inverse suffix array* ISA is the inverse permutation of SA, i.e., for all $1 \leq i \leq n$, $ISA[SA[i]] = i$. The *longest common prefix* (*lcp*) of two strings S and T is the longest string U which is a prefix of both S and T. 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 also be computed in linear time [35].

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 [36]. 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 [37], 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. Other fast and compact data structures also exist for these types of queries [38,39].

The *Burrows-Wheeler Transform* (*BWT*) [11] is a reversible permutation of the input text T.[3] The BWT of T is defined as the concatenation of the last characters of the rotations of T, when the rotations are taken in lexicographic order. This is often visualized using the so-called *BW-matrix*, whose rows are the lexicographically sorted rotations of T, see Fig. 1. BWT_T is the last column of this matrix, read from top to bottom. Then the first column of the BW-matrix, commonly referred to as F (first), is the sorted list of the characters of T, while the last column L (last) is the BWT.

The BWT can also be defined via the SA as $BWT[i] = \varpi$ if $SA[i] = 1$, and $BWT[i] = T[SA[i] - 1]$ otherwise. In particular, it can be computed in linear time from the suffix array.

Two fundamental operations in the context of the BWT are the Last-to-First (LF) and the First-to-Last (FL) mappings, which we explain next.

For $c \in \Sigma$, let $C[c]$ be the number of occurrences of all characters c', such that $c' < c$ in T, i.e. $C[c] = |\{i \mid T[i] < c, 1 \leq i \leq n\}|$. Thus, $C[c] + 1$ is the position of the first occurrence of c in F. If $L[i] = c$ is the kth occurrence of character c in L, then $\mathrm{LF}(i) = j$ is the index j such that $F[j]$ is the kth occurrence of c in F. This can be computed using the formula: $\mathrm{LF}(i) = C[c] + rank_c(L, i)$, where

[2] All logarithms in this paper are base 2.

[3] Given that we are assuming a final sentinel character, the BWT is uniquely reversible. In other words, given a string W, there is at most one string T such that $BWT(T) = W$.

F								L
$	G	A	T	T	A	C	A	T
A	C	A	T	$	G	A	T	T
A	T	$	G	A	T	T	A	C
A	T	T	A	C	A	T	$	G
C	A	T	$	G	A	T	T	A
G	A	T	T	A	C	A	T	$
T	$	G	A	T	T	A	C	A
T	A	C	A	T	$	G	A	T
T	T	A	C	A	T	$	G	A

Fig. 1. The BW-matrix on $T = \texttt{GATTACAT\$}$, with $BWT(T) = \texttt{TTCGA\$ATA}$.

$c = L[i]$. Conversely, $\texttt{FL}(i)$ maps the ith character in F to its position in L. To compute it, we can use the formula $\texttt{FL}(i) = select_c(L, k)$, where $c = F[i]$ and $k = i - C[c]$.

The LF operation is at the core of *backward search*, the pattern matching procedure using the BWT [40,41]. The pattern P of length m is matched back-to-front: first, the BWT-range of $P[m]$ is found, that is, a range in the BWT where character $P[m]$ occurs. This corresponds to defining the BWT range as $[b, e] = [C[P[m]], C[P[m] + 1]]$, i.e., a contiguous range where $P[m]$ is found. Then, the match is extended to $P[m-1..m]$ and so on, further refining the BWT range of the matched pattern. For every new character of P, we perform a pair of LF-steps, using the following insight. Let $[i..j]$ be the SA-range of some string Y and $c \in \Sigma$. Then the SA-range of cY is $[i'..j']$, where $i' = C[c] + rank_c(BWT, i-1) + 1$, and $j' = C[c] + rank_c(BWT, j)$. Therefore, given the array C, the backward search procedure takes time $\mathcal{O}(|P| \cdot t_{rank})$, where t_{rank} is the time for one rank-query, and depends on the data structure used.

We are now ready to define the central data structure of this paper.

Definition 1 (Matching statistics)**.** Given two strings S and R, with distinct sentinel characters, the *matching statistics of S with respect to R* is an array $MS = MS_{S,R}$ of length $|S|$, defined as follows. Let U be the longest prefix of suffix $S[i..]$ which occurs in R as a substring. Then $MS[i] = (p_i, \ell_i)$, where $p_i = -1$ if $U = \varepsilon$, and p_i is an occurrence of U in R otherwise, and $\ell_i = |U|$. (Note that p_i is not unique in general.) We refer to U as the *matching factor*.

Note that our definition differs slightly from the original definition of Chang and Lawler [2]. There, the data structure consists of two arrays, one consisting of the lengths of the matching factors (i.e., the ℓ_is), and the other of pointers to nodes in the suffix tree: this pointer points to the locus of the matching factor (in our terminology) in the suffix tree of R. In particular, Chang and Lawler also store $ST(R)$. One consequence of this is that they do not need to fix a particular occurrence of the matching factor in R, but instead, via the suffix tree, essentially point to all occurrences of the matching factor simultaneously. Our definition is what is referred to as *extended matching statistics* in Gusfield [9, Sec. 7.8.3]. As the definition also notes, p_i is not necessarily unique—the choice of exactly which position to include in $MS[i]$ is largely application dependent.

Example 2.

Let $R = \underset{1\,2\,3\,4\,5\,6\,7\,8\,9}{\texttt{GATTACAT\#}}$ and $S = \texttt{GATTAGATTACATTA\$}$. Then the matching statistic of S w.r.t. R, $MS_{S,R}$, is as follows:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S	G	A	T	T	A	G	A	T	T	A	C	A	T	T	A	$
p_i	1	2	3	4	5	1	2	3	4	5	6	2	3	4	5	−1
ℓ_i	5	4	3	2	1	8	7	6	5	4	3	4	3	2	1	0

The following is a well-known property of the matching statistics, which will be used repeatedly in this paper.

Lemma 1 (Folklore)**.** *Let MS be the matching statistics of S w.r.t. R, and $MS[i] = (p_i, \ell_i)$, for $1 \le i \le |S|$. If $\ell_i > 0$, then $\ell_{i+1} \ge \ell_i - 1$.*

Proof. Let U be the matching factor in position i of S. Then $U[2..\ell_i]$ is a prefix of $S[i+1..]$, of length $\ell_i - 1$, which occurs in R. □

3. Storing the matching statistics

Let S and R be two strings over the alphabet Σ, and let MS denote the matching statistics of S with respect to R. The simplest way to store the matching statistics is as an array of length $|S|$, requiring space for two integers for every entry. More precisely, MS requires $2|S|\lceil \log |R| \rceil$ bits of space, given that both ℓ_i and p_i are in the range between 0 and $|R|$.

3.1. Compact matching statistics

Cunial et al. [42] present a *compact* representation of the matching statistics lengths that exploits Lemma 1. This is a bitvector ms of length $2|S|$ storing the differentially encoded lengths of the matching factors of S. For every position $i \in [1..|S|]$, the corresponding MS value stored is: $MS[i].\ell - MS[i-1].\ell + 1$ zeros followed by a single 1. $MS[0].\ell$ is set to 1. To retrieve the length of the matching statistics for position i, a single select operation is needed: $\ell_i = select_1(ms, i) - 2(i-1) - 1$.

Example 3. Continuing Example 2, we have $ms = 0000011111000000001\ 111110011111$. Then, $MS[6].\ell$ can be accessed via $select_1(ms, 6) - 2(i-1) - 1 = 19 - 10 - 1 = 8$. Another example: $MS[12].\ell = 27 - 22 - 1 = 4$.

The authors also present several ways of compressing the bitvector based on both lossless and lossy methods. For example, on some datasets consisting of similar genomes, using a run-length encoding strategy with a δ-coder yields a compression ratio of up to 20. When handling sets of dissimilar genomes coming from different species, this approach is not sufficient, and sometimes has a detrimental effect. The authors devise a modified version of MS that aims at filtering values below a predefined user threshold τ, which they call *thresholded matching statistics* MS_τ. This representation is defined as follows: $MS_\tau[i].\ell = MS[i].\ell$ if $MS[i].\ell \geq \tau$, or $MS_\tau[i].\ell$ is set to an arbitrary (possibly negative) value smaller than τ otherwise. Intervals in ms_τ corresponding to regions containing values below the threshold are processed to achieve better compression. The authors provide multiple schemes based on permuting the values in such ranges, and demonstrate empirically that, for reasonable values of τ, the space used by the resulting ms_τ can be as low as 2% of the space of the original uncompressed ms bitvector.

3.2. Compressed matching statistics

Another data structure for storing the matching statistics, which exploits Lemma 1 in a different way, and which is particularly useful when R and S are very similar, was presented by Lipták, Luca, Masillo, and Puglisi in [12,43].

Let us call a position j a *head* if $\ell_j > \ell_{j-1} - 1$, and a sequence of the form $(x, x-1, x-2, \ldots)$, of length at most $x-1$, a *decrement run*, i.e. each element is one less than the previous one. Using this terminology, we thus have that the sequence $L = (\ell_1, \ell_2, \ldots, \ell_n)$ is a concatenation of decrement runs, i.e. L has the form $(x_1, x_1-1, x_1-2, \ldots, x_2, x_2-1, x_2-2, \ldots, \ldots, x_k, x_k-1, x_k-2, \ldots)$, with each $x_i = \ell_j$ for some head j. We can therefore store the matching statistics in compressed form as follows:

Definition 2 (Compressed matching statistics)**.** Let R, S be two strings over Σ, and MS be the matching statistics of S w.r.t. R. The *compressed matching statistics (CMS) of S w.r.t. R* is a data structure storing $(j, MS[j])$ for each head j, and a predecessor data structure on the set of heads H.

For a set of integers $H = \{h_1, h_2, \ldots, h_m\}$, a *predecessor data structure for H* supports queries for the form $pred(j, H) = \max\{h_i \mid h_i < j\}$, and $pred(j, H) = -1$ if this set is empty (i.e., if j is smaller than all elements of H). There exists a large choice of predecessor data structures, with different space-time tradeoffs, see the recent survey by Navarro and Rojas-Ladesma [44].

We can use CMS to recover all values of MS:

Lemma 2. *Let $1 \leq i \leq |S|$. Then $MS[i] = (p_j + k, \ell_j - k)$, where $j = pred_H(i)$ and $k = i - j$.*

Proof. Let ℓ_i be the length of the matching factor of i. Since there is a matching factor of length ℓ_j starting in position j in S, this implies that $\ell_i \geq \max(0, \ell_j - k)$. If ℓ_i was strictly greater than $\ell_j - k$, this would imply the presence of another head between j and i, in contradiction to $j = pred_H(i)$. Since an occurrence of the matching factor U of j starts in position p_j of R, the matching factor $U' = U[k+1..\ell_j]$ of i has an occurrence at position $p_j + k$. □

Example 4. Continuing Example 2, the compressed matching statistics of S w.r.t. R is the following set of heads: $H = \{(1,1,5), (6,1,8), (12,2,4)\}$. To access $MS_{S,R}[10]$, we have to find the predecessor of 10 in H, which is 6, and then offset the position and the length of the predecessor item. This results in $p_{10} = p_6 + (10-6) = 1 + 4 = 5$ and $\ell_{10} = \ell_6 - (10-6) = 8 - 4 = 4$.

This representation is different from the solution proposed in [42], due to the fact that the CMS, and more prominently, its enhanced version $eCMS$ (see Section 6.2), includes more information than just the length of the matches. This additional information can then be used for other applications, such as suffix sorting (see Section 6.2).

4. Constructing the matching statistics

It is well known that the matching statistics of S w.r.t. R can be computed in time $\mathcal{O}(|R| + |S|)$ and $\mathcal{O}(|R|)$ space by using, for example, the suffix tree of R, as described in Chang and Lawler's original paper [2]. In this section, we first sketch this algorithm before going on to explain more recent MS-construction algorithms.

4.1. Constructing the MS using the suffix tree

Chang and Lawler [2] describe the following algorithm for MS construction: First, the suffix tree $ST(R)$ of R is built. The naive way to compute $MS[i]$, where i is some position in S, is to match the initial characters of $suf_i(S)$ against $ST(R)$ by following the unique path of character matches until no further matches are possible. The length of the matching path—the length of the matching prefix

Input: strings R and S and the suffix array SA_R of R
Output: the matching statistics M of S w.r.t. R

```
1:    ℓ ← 0
2:    [s, e] ← [0, |R|]
3:    for i ← 0 to |S| do
        — Extend match to the right until mismatch
4:      [s', e'] ← extendRight([s, e], ℓ, S[i + ℓ])
5:      while [s', e'] not empty do
6:        [s, e] ← [s', e']
7:        ℓ ← ℓ + 1
8:        [s', e'] ← extendRight([s, e], ℓ, S[i + ℓ])
        — Assign matching statistic for current position
9:      M[i] = (p_i, ℓ_i) ← (SA_R[s], ℓ)
10:     [s, e] ← contractLeft([s, e], ℓ)
        — SA_R[s, e] now contains all occurrences of S[p_i + 1..p_i + ℓ_i]
```

Fig. 2. A generic solution for computing matching statistics of S w.r.t. R using SA_R.

of $suf_i(S)$—is ℓ_i, while any leaf of the subtree below the path can be chosen for p_i. This algorithm has $\mathcal{O}(\sum_{i=1}^{|S|} MS[i]) = \mathcal{O}(|S||R|)$ time. To achieve linear time, suffix links can be used.

Suppose that the algorithm has just computed $MS[i-1]$ by following a matching path for position $i-1$, where $i \geq 2$. If the path is empty (i.e., $\ell_{i-1} = 0$) or it ends within the label of an edge outgoing from the root, then the search for $MS[i]$ starts at the root. Otherwise, the matching path either ends at an internal node v or within the label of an edge outgoing from an internal node v. To match the next suffix $suf_i(S)$ against $ST(R)$, one follows the suffix link $slink(v)$ from node v to node u. To efficiently find the locus of $S[i..i+\ell_{i-1}-1]$ in the suffix tree, the folklore technique called *skip and count* [9] can be used. Now the matching continues from the current locus until the first mismatch, giving ℓ_i. An amortized analysis shows that this algorithm runs in $\mathcal{O}(|S|+|R|)$ time, where the $\mathcal{O}(|R|)$ term is the time for building the suffix tree of R.

4.2. Constructing the MS using the suffix array

Several authors have considered using the suffix array, variously augmented, as a more practical approach than the suffix tree for computing matching statstics [13,43,45].

At a high level, these algorithms are essentially the same as the suffix-tree-based algorithm, and can be thought of as operating in a series of extendRight and contractLeft operations on the suffix array of R. Given an interval $SA[s,e]$ of SA containing all occurrences of string U', extendRight(s,e,c) returns the interval $SA[s',e']$ containing all occurrences of string Uc if it exists in R or an empty range otherwise. This is equivalent to traversing one symbol down an edge in the suffix tree. The operation contractLeft(s,e) returns the range $SA[s',e']$ that contains all occurrences of string $U[2..|U|]$. It is equivalent to suffix link traversal in the suffix tree.

Pseudocode is given in Fig. 2. At a generic step in the algorithm we have computed entry $MS[i] = (p_i, \ell_i)$ and know the SA_R-*range* $[s_i, e_i]$ of the corresponding matching factor U_i. To compute the next entry $MS[i+1]$ we first perform a contractLeft operation, obtaining the interval containing all occurrences of U', where $U_i = S[i]U'$. U' is a prefix of the matching factor U_{i+1}. We then perform at most $\ell_i + 1$ extendRight operations—when an empty interval is returned, we know $MS[i+1]$.

Abouelhoda et al. [45] use a so-called child table combined with the *LCP* array to implement extendRight in $\mathcal{O}(\sigma)$ time and a suffix link table to implement contractLeft in $\mathcal{O}(1)$ time. Maass [46] describes an alternative way to simulate suffix links on the suffix array.

To implement extendRight and contractLeft, Lipták et al. [13,43] use the arrays SA_R, ISA_R and LCP_R over the reference string R and data structures answering *extended PSV/NSV* queries on LCP_R (see Section 2). In particular, having computed entry $MS[i]$ and the SA_R-*range* $[s_i, e_i]$ of U_i, to compute the next entry $MS[i+1]$, they then:

1. Contract left: i.e., compute the SA_R-*range* $[s', e']$ of U', where $U_i = S[i]U'$. We have $s' = PSV(LCP_R, ISA_R[SA_R[s_i]+1]], |U'|)$ and $e' = NSV(LCP_R, ISA_R[SA_R[e_i]+1]], |U'|)-1$. This takes $\mathcal{O}(1)$ time.
2. Extend the factor U' to the right with character $c = S[i+1+|U'|]$ as long as $S[i+1..i+1+|U'|]$ occurs in R. This is done by searching for the longest common prefix of $S[i+1..]$ that occurs in R using two binary searches on SA_R to obtain the $U'c$-interval $[s,e]$ if it exists. This step takes $\mathcal{O}(\log|R|)$ time per matched symbol.

The authors of [43] further utilize heuristics that allow one to often skip the second step above and greatly improve running time in practice. The most significant heuristic is the *longest repeating factor*-array (LRF) of the string R, given by $LRF_R[i] =$

$\max\{LCP_R[i], LCP_R[i+1]\}$. Given the entry $MS[i] = (p_i, \ell_i)$, if $\ell_i - 1 > LRF_R[p_i + 1]$ then $MS[i+1] = (p_i + 1, \ell_i - 1)$, because there are no other occurrences of factor $R[p_i + 1..p_i + \ell_i]$ in R that need to be checked.

4.3. Constructing the MS using the Burrows-Wheeler transform

Ohlebusch et al. [5] give a solution for computing the matching statistics using the BWT and the LCP array. Their approach makes use of the concept *lcp intervals*, which correspond to nodes of the suffix tree.

Definition 3 (lcp-interval [45]). An interval $[i..j]$, $0 \le i < j \le n$, is called an *lcp-interval* of lcp-value ℓ if:

- $LCP[i] < \ell$,
- $LCP[k] \ge \ell$ for all k with $i + 1 \le k \le j$,
- $LCP[k] = \ell$ for at least one k with $i + 1 \le k \le j$, and
- $LCP[j + 1] < \ell$.

Ohlebusch et al. [5] store the *BWT* itself, in a data structure allowing *rank* and *select* operations, as well as the *LCP*-array along with a data structure that answers *parent*(i, j), where $[i..j]$ is an lcp-interval.

For data structures supporting *rank* and *select* operations, there are a variety of solutions from compressed text indexing (see, e.g., [19,47]). For example, using a wavelet tree built on the *BWT*, both operations *rank* and *select* can be performed in $\mathcal{O}(\log \sigma)$ time and using space $n \log \sigma + o(n \log \sigma)$ bits.

For data structures supporting the *parent*(i, j) query, one can use *PSV* and *NSV* queries on the *LCP* array (see Section 2).

Given $LCP[i] = p$ and $LCP[j] = q$, then:

$$parent(i, j) = \begin{cases} p - [PSV[i]..NSV[i] - 1] & \text{if } p \ge q, \\ q - [PSV[j + 1]..NSV[j + 1] - 1] & \text{if } p < q. \end{cases} \tag{1}$$

With these components, the matching statistics computation is performed by backward searching for S in R. Starting from the last character of S, LF-steps are performed until a mismatch occurs. At each step the length is increased by one. When the backward search step fails, calls to *parent*(i, j) are issued to compute the correct interval and matching length at which to resume the computation.

If we are only interested in the lengths of the matching statistics, this algorithm computes them in $\mathcal{O}(|R| + |S|)$ time and $\mathcal{O}(|R|)$ space. If positions are also required, the time becomes $\mathcal{O}(|R| + |S| t_{access(SA)})$, where $t_{access(SA)}$ is the time needed to access an entry of SA.

4.4. Constructing the MS using the run-length compressed BWT

Bannai, Gagie, and I in [48] showed how to compute the matching statistics in compressed space. More specifically, the authors use an augmented run-length compressed *BWT* taking space proportional to r_R, where r_R is the number of runs of the *BWT* of the reference string.

Prior to [48], Prezza and Policriti [49] had shown how to represent the *BWT* using $\mathcal{O}(r_R)$ space in such a way that a single occurrence of the search pattern could be determined in $\mathcal{O}(\log \log n)$ time after backward search, by storing suffix array entries at the beginning and at the end of *BWT* runs. This is also called the *toehold lemma*. Essentially the same technique is used by Kärkkäinen, Kempa, and Puglisi in [50].

In [48], some further augmentation is needed in order to be able to recover from backward steps that fail. This small additional data structure is called *thresholds*. It is used to compute the position of the longest prefix shared by some suffix of S and R when a mismatch occurs during the backward search for S on the *BWT*(R). Given an interval $[i..j]$ defining a run in the *BWT* and a character c different than the one of the *BWT* run in $[i..j]$, there exists a position k, $i \le k \le j$, called threshold, such that:

- for $i \le i' < k$, $cR[SA[i']..|R|]$ has a longer lcp with the occurrence of c preceding $BWT[i]$ and
- for $k \le j' \le j$, $cR[SA[j']..|R|]$ has a longer lcp with the occurrence of c following $BWT[i]$.

The authors show that it is possible to store the thresholds in $\mathcal{O}(r_R \sigma)$ words. This allows us to recover after a failed backward step using $\mathcal{O}(1)$ time and resume the computation at two possible positions, depending on the threshold stored at the *BWT* run:

1. $q = \max\{j : BWT[j] = c,\ j < i\}$
2. $q' = \min\{j : BWT[j] = c,\ j > i\}$

The main algorithm for computing $MS_{S,R}$ has two phases:

1. with a right-to-left pass of S, we can compute the positions p_i for every suffix i in S
2. with a left-to-right pass, we can compute the lengths ℓ_i for every suffix i in S.

Phase 1 is done by backward stepping each character of S in the augmented RLBWT of R with thresholds as recovery mechanism.

Phase 2 has to use some fast random access to the uncompressed reference R. In Bannai et al. [48], a relative Lempel-Ziv (RLZ) representation of R is used to allow $\mathcal{O}(\log \log n)$ time access to arbitrary positions in R. We start comparing the string S with the

reference R beginning at position p_1. We then extend the match by comparing the characters $S[i]$ with $R[p_1 + i - 1]$ for positions $1 < i \leq |S|$ until a mismatch is encountered. When a mismatch happens, instead of recomputing ℓ_2 from scratch (or in general ℓ_{i+1}), we exploit Lemma 1: since $\ell_{i+1} \geq \ell_i - 1$, we can jump directly to comparing the characters beyond the known matching prefix. This requires random access to R.

Overall, the time used to compute the matching statistics of S w.r.t. R is $\mathcal{O}(|S| \log\log |S| + |S| t_{access(R)})$, where $t_{access(R)}$ is the time for performing random access to R. The space usage is $\mathcal{O}(r_R \sigma + s_{access(R)})$, where $s_{access(R)}$ is the space required for the data structure supporting random access on R.

4.5. Constructing the MS using the r-index

Computing the matching statistics with the r-index of Gagie, Navarro, and Prezza [51] is closely related to the RLBWT procedure described in the previous section.

Bannai, Gagie, and I in [52] refine the method used to store thresholds, reducing the space complexity of the augmented RLBWT from $\mathcal{O}(r_R \sigma)$ to just $\mathcal{O}(r_R)$. The key observation is that, given two consecutive runs $BWT[i..j]$ and $BWT[k..l]$ of a character c, with $i \leq j < k \leq l$, we can find a position t such that the lcp between suffixes from position j to t is non-increasing, while the lcp between suffixes from t to k is non-decreasing. This means that we only need to store the local minimum of the lcp values in the range between consecutive runs of the same characters. By building an appropriate predecessor data structure [44] on the set of threshold positions, one can apply the recovery mechanism in $\mathcal{O}(\log\log n)$ time.

Overall, the time complexity stays the same $\mathcal{O}(|S| \log\log |S| + |S| t_{access(R)})$, while the total space is reduced to $\mathcal{O}(r_R + s_{access(R)})$.

5. Applications to string processing tasks

In this section, we give a non-exhaustive tour of applications of the matching statistics to problems in string processing.

5.1. Approximate pattern matching

Given a text T and a pattern P, P is said to be an *approximate match* in position i of T if P matches a substring of T which starts in position i, with some errors. This has to be made more precise; the two most common variants are the k-mismatch and k-difference problems. Both use a fixed positive integer k.

1. **k-mismatch problem:** Given text T and pattern P, we want to find all positions i of T such that the Hamming distance of P and $T[i..i + |P| - 1]$ is at most k.
2. **k-differences problem:** Given text T and pattern P, we want to find all positions i of T such that the edit distance of P and some prefix of $suf_i(T)$ is at most k.

In the k-differences problem therefore, errors may be single-character substitutions, insertions, or deletions, while in the k-mismatch problem, only substitutions are allowed. Our description of the algorithm follows [9]; the original can be found in Sec. 3.2 of [2].

First, a linear expected time algorithm (LET) is given in [2]:

1. compute a set of so-called "starting-positions" SP, where $SP_1 = 1$ and $SP_j = SP_{j-1} + \ell_{SP_{j-1}} + 1$ for $j > 1$;
2. if $SP_{j+k+2} - SP_j \geq |P| - k$, apply a worst-case $\mathcal{O}(k|T|)$ time dynamic-programming subroutine (e.g. Landau-Vishkin [53,54]) to the substring $T[SP_j..SP_{j+k+2}]$ to compute the correct set of positions matching the pattern (up to k mismatches/differences).

It is shown in [2] that, assuming that (a) the characters of T are drawn at random, and (b) $k \leq k^* = |P|/(\log_\sigma |P| + c_1) - c_2$ (where c_1 and c_2 are two constants), the expected work done at any position in SP is constant. This sums up to $\mathcal{O}(|T|)$ expected total time.

When $k \leq k^*/2 - 3$, the linear expected time algorithm can be further improved to sublinear expected time, as follows. The text T is split into blocks of size $(|P| - k)/2$. For each block, perform at most $k + 1$ "matching statistics jumps" from the starting position s of the block. For each jump, add to the current position the *MS* length ℓ_i of such positions. Computing the matching statistics value takes expected $\mathcal{O}(\log_\sigma |P|)$ time, due to T being randomly drawn from Σ. After $k + 1$ jumps, if the ending position p is within the end of the block, then discard the block. Otherwise, if p is outside of the block, apply the dynamic-programming subroutine to the substring $T[(s - (|P| + 3k)/2)..p]$. Chang and Lawler showed that on average the number of blocks on which we have to apply the subroutine is bounded by $\mathcal{O}(|T|/|P|)$. This approach combined with the LET algorithm, depending on the value of k, finally leads to the expected running time of $\mathcal{O}((|T|/|P|)k \log_\sigma |P|)$. The approach was subsequently refined by Takaoka [55] by simplifying the algorithm, relaxing the assumptions on the randomness of the reference sequence, and extending it to the two-dimensional scenario.

5.2. Exact string matching

It is easy to see that, given the matching statistics *MS* of S w.r.t. R, we also have a solution to the exact string matching problem: R occurs in S in position i if and only if $\ell_i = |R|$. Indeed, Chang and Lawler [2] note that their matching statistics construction algorithm also solves the exact pattern matching problem, in time $\mathcal{O}(|S| + |R|)$. In other words, the algorithm for exact string matching can be seen as a by-product of the *MS*-construction.

5.3. Longest common substring

The problem of *longest common substring* (also known as *longest common factor* or *LCF*) of two strings S and T was the original motivation for the introduction, by Peter Weiner in 1973, of the suffix tree data structure [20]. Weiner used what would nowadays be referred to as *generalized suffix tree* of S and T to compute the length of a longest common substring in time $\mathcal{O}(|S| + |T|)$. Although this is optimal with respect to running time, in practice the space overhead of a generalized suffix tree can be prohibitive.

As an alternative approach, described by Gusfield [9, Section 7.9], the matching statistics can be used to reduce space consumption. The procedure for solving the longest common substring problem now starts by constructing the suffix tree for the shorter of the two strings, say S. Then, by computing $MS_{T,S}$ via $ST(S)$, a simple scan to find the maximum ℓ_i in the MS gives the correct solution. Moreover, for any j such that $\ell_j = \max\{\ell_i \mid 1 \le i \le |T|\}$, the pair of positions j in T and p_j in S gives an occurrence of an LCF.

5.4. Longest common extension queries

Given two strings S and T, after a preprocessing step, we must answer a stream of queries, with each query consisting of a pair of indices (i, j). For each pair, we want to find the longest substring of S starting at position i that equals a substring in T starting at position j. This is called the *longest common extension* (LCE) of suffixes $S[i..]$ and $T[j..]$.

As with the longest common substring problem, a classical solution for longest common extension involves the use of a generalized suffix tree on S and T. The generalized suffix tree is augmented with the stringdepth information $sd(v)$ stored at each node v and additional machinery used to compute the lowest common ancestor (LCA) between leaves in constant time [56,57]. Computing the generalized suffix tree and the additional information can be done in $\mathcal{O}(|S| + |T|)$ time and space. Using this data structure, answering an *LCE* query is equivalent to an LCA query on the leaves containing positions i and j: Let v be the LCA of i and j, then $LCE(i, j) = sd(v)$. Therefore, *LCE* queries can be performed in $\mathcal{O}(1)$ time.

An alternative lightweight approach, described by Gusfield [9, Section 9.1.2], involves the use of matching statistics. Assuming that $|T| < |S|$, we can compute the matching statistics $MS_{S,T}$ using $ST(T)$. Then, $LCE(i, j)$ need to first find v, the LCA of p_i and j in $ST(T)$, and then take the minimum between ℓ_i and $sd(v)$. This procedure takes constant time just like the classical solution using a generalized suffix tree, but uses less space in practice.

5.5. Alignment-free whole-genome phylogenetics

Ulitsky, Burstein, Tuller, and Chor [7] use the matching statistics to improve the precision of phylogenetics reconstruction.

Whole-genome phylogenetics reconstruction involves determining the evolutionary relationships among species or genes by analyzing genome-wide data. Unlike conventional phylogenetics, which typically depends on a limited set of genes such as mitochondrial or ribosomal RNA genes, whole-genome phylogenetics uses extensive genomic information to construct more precise and reliable phylogenetic trees.

The authors exploit the matching statistics to compute a pairwise distance between sequences. To do this, they first define a non-symmetric function d as

$$d(S, T) = \log(|T|)/L(S, T) - \log(|S|)/L(S, S),$$

where $L(X, Y) = \sum_{i=1}^{n} \ell_i/|X|$ is the average length of the matching statistics of X w.r.t. Y. The actual distance function, called *average common substring distance*, is then given by $ACS(S, T) = (d(S, T) + d(T, S))/2$.

The authors show that $d(X, Y)$ approximates the cross-entropy between two distributions p, q, that is $H(p, q) = -E_p[\log q]$, where $E_p[\cdot]$ is the expected value of the argument with respect to p. Intuitively, this is the difference between two strings generated by two distinct Markovian processes: the distance is measured by the number of bits needed to describe one sequence given the other.

The distance matrix, containing the pairwise distances of the input sequences, is filled with pairwise ACS distances and then processed by some other algorithm, such as neighbor joining [58], to output the phylogenetic tree. This method was also used in [6]. Variants accounting for k-mismatches in the matching factor have been proposed in various studies [59–62].

5.6. Maximal exact matches (MEMs)

Given two strings S and T, a substring $S[i..i + \ell - 1]$ is a *Maximal Exact Match (MEM)* of S in T if:

- $S[i..i + \ell - 1]$ occurs exactly in T (i.e., $S[i..i + \ell - 1] = T[j..j + \ell - 1]$ for some index j),
- either $i = 1$ or $S[i - 1..i + \ell - 1]$ does not occur in T, and
- either $i + \ell - 1 = |S|$ or $S[i..i + \ell]$ does not occur in T.

MEMs are used to accelerate the alignment of read sets to a reference genome. One of the most used read aligners, `BWA-MEM` [63], first finds MEMs between the query read and the reference genome and then uses them as "seeds" to speed up the subsequent dynamic programming phase of the alignment procedure. In pangenomics, alignment of reads is performed on a collection of genomes, called a pangenome. Several recent tools, including `MONI` [64], `PHONI` [65], `AUG-PHONI` [66], `LAZY-PHONI` [67], `SPUMONI` [68], `SPUMONI2` [69], `SIGMONI` [70], and `MOVI` [71], make use of the r-index [51], to compute MEMs for large genome collections. MEMs are also used as anchors in seed-chain-extend-based algorithms and tools for (multiple) sequence alignment (see, e.g. [9,72,73]).

MEMs can be computed using the matching statistics via the following lemma:

Lemma 3. *[64] Given an input text $R[1..n]$ and a query string $S[1..m]$ with $m \leq n$, let $MS_{S,R}[1..m]$ be the matching statistics of S w.r.t. R. For all $1 < i \leq m$, $R[i..i + \ell - 1]$ is a maximal exact match of length ℓ in R if and only if $MS_{S,R}[i] = \ell$ and $MS_{S,R}[i-1] \leq MS_{S,R}[i]$.*

A linear time algorithm for computing MEMs between S and R simply iterates over $MS_{S,R}$ and outputs a MEM of length $MS_{S,R}[i]$ occurring at position i whenever the property given in the lemma holds.

We note that the computation of MEMs has been extended to multisets of cyclic strings [74], using the eBWT (extended BWT) of [75].

6. Applications in index construction and data compression

In this section, we will focus on the use of matching statistics in applications related to compression—in particular Lempel-Ziv and relative Lempel-Ziv parsing—and as a step in the construction of other data structures—particularly the SA and BWT.

6.1. Space-time tradeoff for LZ-parsing

The LZ77 factorization [76,77] uses the notion of a *longest previous factor* (LPF). The LPF at position i in S is a pair (p_i, ℓ_i) such that $p_i < i$, $S[p_i..p_i + \ell_i - 1] = S[i..i + \ell_i - 1]$, and $\ell_i > 0$ is maximal. In other words, $S[i..i + \ell_i - 1]$ is the longest prefix of $S[i..n]$ which also occurs at some position $p_i < i$ in S. If $S[i]$ is the leftmost occurrence of a symbol in S then such a pair does not exist. In this case we define $p_i = S[i]$ and $\ell_i = 0$. Note that there may be more than one potential p_i, and we do not care which one is used.

The LZ77 factorization (or LZ77 parsing) of a string S is then just a greedy, left-to-right parsing of S into longest previous factors. More precisely, if the jth LZ factor (or *phrase*) in the parsing is to start at position i, then we output (p_i, ℓ_i) (to represent the jth phrase), and then the $(j+1)$th phrase starts at position $i + \ell_i$, unless $\ell_i = 0$, in which case the next phrase starts at position $i + 1$. We call a factor (p_i, ℓ_i) *repeat* if it satisfies $\ell_i > 0$ and *literal* otherwise. The number of phrases in the factorization is denoted by z.

For the example string $S = zzzzzipzip$, the LZ77 factorization produces:

$$(z, 0), (1, 4), (i, 0), (p, 0), (5, 3).$$

The second and fifth factors are repeat, and the other three are literal.

The LPF array is a sequence of pairs that define the longest previous factor at each position in S. For our example string a valid LPF array is:

$$[(z, 0), (1, 4), (2, 4), (3, 4), (4, 4), (i, 0), (p, 0), (5, 3), (6, 2), (7, 1)].$$

Kärkkäinen et al. [50] describe an algorithm for LZ factorization or (LZ parsing) that allows space to be traded for running time. Their algorithm, called *LZscan*, divides the input string S into blocks of size b, where b is an input parameter. Then, via some computation that we sketch next, it is able to compute the optimal LZ parse over the total input string. In the following, we will use $B = S[kb + 1..(k+1)b]$ to refer to the currently processed block in S, and $A = S[1..kb]$ to refer to the already processed prefix of S, as the original paper. The LPF array for block B is denoted by LPF_B, while the LPF array for the concatenation of A and B is LPF_{AB}.

The procedure can be divided into four steps:

1. compute $MS_{A,B}$;
2. compute $MS_{B,A}$ via inverting $MS_{A,B}$ using data structures on B;
3. compute $\text{LPF}_{AB}[kb + 1..(k+1)b]$ using $MS_{B,A}$ and LPF_B;
4. factorize B using $\text{LPF}_{AB}[kb + 1..(k+1)b]$.

Step 1 is done via backward searching A in BWT_B using the standard algorithm outlined in Section 4.

Step 2 is performed via what the authors of [50] call inverting the matching statistics. It consist of running a scan on A and populating $MS_{B,A}$ using SA_B and LCP_B. This is done in order to avoid building heavy data structures on A, which can be large.

Step 3 is where the matching statistics are used. Consider computing $LPF_{AB}[i] = (p_i, \ell_i)$ for $i \in [kb + 1..(k+1)b]$ and assume $\ell_i > 0$. Either we have $p_i \leq kb$, in which case $\text{LPF}_{AB}[i] = MS_{B,A}[i]$, or $kb < p_i < i$ and then $\text{LPF}_{AB}[i] = (kb + p_B, \ell_B)$, where $(p_B, \ell_B) = \text{LPF}_B[i - kb]$.

A simple correction for cases where $p_i \leq kb$ but $p_i + \ell_i > kb + 1$ is to replace $MS_{A,B}$ with $MS_{AB,B}[1..kb]$ in all relevant steps.

6.2. Suffix sorting and BWT computation

Lipták et al. [12,13] introduced a suffix array construction algorithm for sets of highly repetitive biological collections that makes use of the fact that the matching statistics of such collections is small and can be computed fast.

They enhance the compressed representation of the matching statistics (see Section 3.2), resulting in the *enhanced CMS*, or *eCMS*, to contain the following information stored in the so-called *insert-heads*:

- p_i is substituted with $q_i = SA[ip(i)]$, where $ip(i)$ is the *insert-point* for position i. The insert point is the lexicographic rank, among all suffixes of R, of the next smaller occurrence of the matching factor U in R if such an occurrence exists, and of the smallest occurrence of U in R otherwise.
- the mismatch character c_i, the character immediately following U.

Input: the matching statistics M of S w.r.t. R
Output: the relative Lempel-Ziv parsing of S w.r.t. R

```
1:  i ← 1
2:  while i ≤ |S| do
3:      if ℓ_i > 0 then
4:          output factor (ℓ_i, p_i)
5:          i ← i + ℓ_i
6:      else
7:          output literal factor (S[i])
8:          i = i + 1
```

Fig. 3. Algorithm converting $MS_{S,R}$ into $RLZ_R(S)$.

- a boolean value x_i which tells us whether Uc_i is smaller (S) or greater (L) than $R[q_i..]$.

Example 5. Continuing Example 2, the enhanced compressed matching statistics of S w.r.t. R is the following set of insert-heads: $K = \{(1, 1, 5, \mathtt{G}, L), (6, 1, 8, \mathtt{T}, L), (12, 2, 4, \$, S), (16, -1, 0, \$, L)\}$.

The computation takes the same time as for *CMS*. The *eCMS* (after some further processing) allows us to perform suffix comparisons in $\mathcal{O}(1)$ time after retrieving the correct *MS* value for a pair of suffixes. Overall, the algorithm to compute the *SA* for a collection C follows this outline:

1. compute the *eCMS* of the collection C w.r.t. an "augmented" reference R (the augmentation is done to cover border-cases);
2. process *eCMS* to allow fast suffix sorting;
3. bucket S^*-suffixes by their insert-point[4];
4. fully sort S^*-suffixes bucket-by-bucket using comparisons based on the *eCMS*;
5. induce the full *SA* from the S^*-suffixes.

The resulting algorithm takes $\mathcal{O}(|R| + N \log |R| + N \log N)$ time, where N is the total length of the all the strings in C, and $\mathcal{O}(|R| + N + \kappa)$ space, where κ is the size of the *eCMS*.

Masillo [14] applied this approach to *BWT* construction for large collections of similar strings. In particular, the key observation is that insert-heads, which play a similar role for *eCMS* as heads for the *CMS*, induce run-breaks in the *BWT* of C. Starting from the *BWT* of R, it is only a matter of counting how many suffixes of C fall into the buckets defined from the insert-point information and sorting the insert-heads within the corresponding bucket, to find where new run-breaks appear. Counting the number of suffixes for each bucket is needed to "expand" the character from BWT_R into a longer run in BWT_C. The runs are then interrupted by the preceding character of the insert-heads, which is known to differ from that of the run they belong to. Sorting only suffixes w.r.t. insert-heads effectively reduces the number of suffix comparisons that have to be performed, leading to further time reduction.

The proposed algorithm runs in $\mathcal{O}(|R| + N \log |R| + N \log \kappa)$ time and $\mathcal{O}(|R| + \kappa)$ space.

6.3. Relative LZ compression

Relative Lempel-Ziv (RLZ) parsing, due to Kuruppu, Puglisi, and Zobel [16], compresses a text S relative to a reference R by expressing substrings of S as pointers to R. Although they arose independently, there is close relationship between the RLZ parse and the matching statistics: the RLZ parse of S w.r.t. R is a subsequence of $MS_{S,R}$. In particular, the RLZ parsing is a greedy, left-to-right parsing of S into factors occurring in R. More precisely, if the jth RLZ factor (or *phrase*) in the parsing is to start at position i, then we output $MS[i] = (p_i, \ell_i)$ (to represent the jth phrase), and then the $(j+1)$th phrase starts at position $i + \ell_i$, unless $\ell_i = 0$, in which case the next phrase starts at position $i + 1$.

We give the pseudocode in Fig. 3. Every character of S is either found in a literal factor (if it only appears in S), or in a repeat factor. The latter type of factors is correct due to the construction of the *MS*, being right-maximal matches between S and R.

RLZ has been studied as a compressor of genomic collections [16,78], collections of textual documents [79,80], and suffix arrays [81,82]. It is particularly interesting in each of those settings because it provides both powerful compression and support for fast random access to the underlying data by storing the phrase starting positions in a predecessor data structure. Bille, Gørtz, Puglisi, and Tarnow [83] describe a hierarchical version of the parsing suitable for compressing genomes of individuals of the same species. The parsing itself can also be augmented in various ways to obtain compressed indexes for pattern matching [84–86].

[4] These are the S^*-suffixes from the well-known SAIS algorithm of Nong et al. [31].

7. Conclusion

This survey has presented current and historical methods for computing and representing matching statistics, together with an overview of their many important applications. Much of this work is recent, and many interesting further directions for research on this fascinating data structure remain. One is to explore generalizations of matching statistics, following, for example, recent work on Wheeler DFAs [87]. Computing the matching statistics in space and/or time proportional to recent compressibility measures, such as the size of the smallest string attractor [88], is also of interest, particularly in the context of indexing. Finally, given its widespread applications, more efficient algorithms for computation of matching statistics, for example in massively parallel settings such as GPUs [89], would be of immediate practical impact.

CRediT authorship contribution statement

Zsuzsanna Lipták: Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization; **Francesco Masillo:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Formal analysis, Conceptualization; **Simon J. Puglisi:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Simon J. Puglisi reports financial support was provided by Research Council of Finland. Zsuzsanna Liptak reports financial support was provided by Francesco Severi National Institute of Higher Mathematics. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] W.I. Chang, E.L. Lawler, Approximate string matching in sublinear expected time, in: Proc. 31st Annual Symposium on Foundations of Computer Science (FOCS), 1, 1990, pp. 116–124. https://doi.org/10.1109/FSCS.1990.89530

[2] W.I. Chang, E.L. Lawler, Sublinear approximate string matching and biological applications, Algorithmica 12 (4/5) (1994) 327–344.

[3] S. Rahmann, Fast and sensitive probe selection for DNA chips using jumps in matching statistics, in: Proc. 2nd IEEE Computer Society Bioinformatics Conference (CSB), IEEE Computer Society, 2003, pp. 57–64.

[4] C.H. Teo, S.V.N. Vishwanathan, Fast and space efficient string kernels using suffix arrays, in: Proc. 23rd International Conference on Machine Learning (ICML), 148 of *ACM International Conference Proceeding Series*, ACM, 2006, pp. 929–936.

[5] E. Ohlebusch, S. Gog, A. Kügel, Computing matching statistics and maximal exact matches on compressed full-text indexes, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, Springer, 2010, pp. 347–358.

[6] E. Cohen, B. Chor, Detecting phylogenetic signals in eukaryotic whole genome sequences, J. Comput. Biol. 19 (8) (2012) 945–956.

[7] I. Ulitsky, D. Burstein, T. Tuller, B. Chor, The average common substring approach to phylogenomic reconstruction, J. Comput. Biol. 13 (2) (2006) 336–350.

[8] N. Philippe, M. Salson, T. Commes, E. Rivals, CRAC: an integrated approach to the analysis of RNA-seq reads, Genome Biol. 14 (2013) 1–16.

[9] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

[10] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.

[11] M. Burrows, D.J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, Technical Report, DIGITAL System Research Center, 1994.

[12] Zs. Lipták, F. Masillo, S.J. Puglisi, Suffix sorting via matching statistics, in: Proc. 22nd International Workshop on Algorithms in Bioinformatics (WABI), LIPIcs 242, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:15.

[13] Zs. Lipták, F. Masillo, S.J. Puglisi, Suffix sorting via matching statistics, Algorithms Mol. Biol. 19 (1) (2024) 11.

[14] F. Masillo, Matching statistics speed up BWT construction, in: Proc. 31st Annual European Symposium on Algorithms (ESA), LIPIcs 274, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 83:1–83:15.

[15] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.

[16] S. Kuruppu, S.J. Puglisi, J. Zobel, Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, Springer, 2010, pp. 201–206.

[17] D.R. Clark, Compact PAT Trees, Ph.D. thesis, University of Waterloo, Canada, 1996.

[18] J.I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.

[19] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual Symposium on Discrete Algorithms (SODA), ACM/SIAM, 2003, pp. 841–850.

[20] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 1–11.

[21] A. Apostolico, The myriad virtues of subword trees, in: Combinatorial Algorithms on Words, NATO ISI Series, Springer-Verlag, 1985, pp. 85–96.

[22] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.

[23] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1997, pp. 137–143.

[24] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[25] D.K. Kim, J.S. Sim, H. Park, K. Park, Constructing suffix arrays in linear time, J. Discr. Algorithms 3 (2-4) (2005) 126–142.

[26] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discr. Algorithms 3 (2-4) (2005) 143–156.

[27] S.J. Puglisi, W.F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2) (2007) 4.

[28] J. Bahne, N. Bertram, M. Böcker, J. Bode, J. Fischer, H. Foot, F. Grieskamp, F. Kurpicz, M. Löbel, O. Magiera, R. Pink, D. Piper, C. Poeplau, SACABench: benchmarking suffix array construction, in: Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 11811, Springer, 2019, pp. 407–416.

[29] T. Bingmann, Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools, Ph.D. thesis, Karlsruhe Institute of Technology, Germany, 2018. https://publikationen.bibliothek.kit.edu/1000085031.

[30] G. Nong, S. Zhang, W.H. Chan, Two efficient algorithms for linear time suffix array construction, IEEE Trans. Computers 60 (10) (2011) 1471–1484.

[31] G. Nong, Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets, ACM Trans. Inf. Syst. 31 (3) (2013) 15.

[32] U. Baier, Linear-time suffix sorting - a new approach for suffix array construction, in: Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM), LIPIcs 54, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 23:1–23:12.

[33] K. Goto, Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets, in: Proc. Prague Stringology Conference (PSC), Czech Technical University in Prague, 2019, pp. 111–125.
[34] Z. Li, J. Li, H. Huo, Optimal in-place suffix sorting, Inf. Comput. 285 (Part) (2022) 104818.
[35] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. 12th Annual Symposium Combinatorial Pattern Matching (CPM), LNCS 2089, Springer, 2001, pp. 181–192.
[36] J. Fischer, Combined data structure for previous- and next-smaller-values, Theor. Comput. Sci. 412 (22) (2011) 2451–2456.
[37] R. Cánovas, G. Navarro, Practical compressed suffix trees, in: Proc. of the 9th International Symposium Experimental Algorithms, SEA 2010, 6049 of *LNCS*, Springer, 2010, pp. 94–105.
[38] E. Ohlebusch, S. Gog, A compressed enhanced suffix array supporting fast string matching, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5721, Springer, 2009, pp. 51–62.
[39] J. Fischer, V. Mäkinen, G. Navarro, Faster entropy-bounded compressed suffix trees, Theor. Comput. Sci. 410 (51) (2009) 5354–5364.
[40] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 2000, pp. 390–398. https://doi.org/10.1109/SFCS.2000.892127
[41] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581. https://doi.org/10.1145/1082036.1082039
[42] F. Cunial, O. Denas, D. Belazzougui, Fast and compact matching statistics analytics, Bioinform. 38 (7) (2022) 1838–1845.
[43] Zs. Lipták, M. Lucà, F. Masillo, S.J. Puglisi, Fast matching statistics for sets of long similar strings, in: Proc. Prague Stringology Conference (PSC), Czech Technical University in Prague, 2024, pp. 3–15.
[44] G. Navarro, J. Rojas-Ledesma, Predecessor search, ACM Comput. Surv. 53 (5) (2020) .
[45] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discr. Algorithms 2 (1) (2004) 53–86.
[46] M.G. Maaß, Computing suffix links for suffix trees and arrays, Inf. Process. Lett. 101 (6) (2007) 250–254.
[47] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007) 2.
[48] H. Bannai, T. Gagie, I. Tomohiro, Online LZ77 parsing and matching statistics with RLBWTs, in: Proc. 29th Annual Symposium on Combinatorial Pattern Matching (CPM), LIPIcs 105, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 7:1–7:12.
[49] A. Policriti, N. Prezza, Computing LZ77 in run-compressed space, in: Proc. Data Compression Conference (DCC), IEEE, 2016, pp. 23–32.
[50] J. Kärkkäinen, D. Kempa, S.J. Puglisi, Lightweight Lempel-Ziv parsing, in: Proc. 12th International Symposium on Experimental Algorithms (SEA), LNCS 7933, Springer, 2013, pp. 139–150.
[51] T. Gagie, G. Navarro, N. Prezza, Fully functional suffix trees and optimal text searching in BWT-runs bounded space, J. ACM 67 (1) (2020) 2:1–2:54. https://doi.org/10.1145/3375890
[52] H. Bannai, T. Gagie, I. Tomohiro, Refining the *r*-index, Theor. Comput. Sci. 812 (2020) 96–108.
[53] G.M. Landau, U. Vishkin, Fast string matching with *k* differences, J. Comput. Syst. Sci. 37 (1) (1988) 63–78.
[54] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, J. Algorithms 10 (2) (1989) 157–169.
[55] T. Takaoka, Approximate pattern matching with samples, in: Proc. International Symposium on Algorithms and Computation (ISAAC), LNCS 834, Springer, 1994, pp. 234–242.
[56] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN), LNCS 1776, Springer, 2000, pp. 88–94.
[57] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[58] N. Saitou, M. Nei, The neighbor-joining method: a new method for reconstructing phylogenetic trees, Mol. Biol. Evol. 4 (4) (1987) 406–425.
[59] C. Leimeister, B. Morgenstern, kmacs: the *k*-mismatch average common substring approach to alignment-free sequence comparison, Bioinform. 30 (14) (2014) 2000–2008.
[60] A. Apostolico, C. Guerra, G.M. Landau, C. Pizzi, Sequence similarity measures based on bounded hamming distance, Theor. Comput. Sci. 638 (2016) 76–90.
[61] C. Pizzi, MissMax: alignment-free sequence comparison with mismatches through filtering and heuristics, Algorithms Mol. Biol. 11 (2016) 6.
[62] S.V. Thankachan, A. Apostolico, S. Aluru, A provably efficient algorithm for the *k*-mismatch average common substring problem, J. Comput. Biol. 23 (6) (2016) 472–482.
[63] H. Li, Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, 2013. https://arxiv.org/abs/1303.3997.
[64] M. Rossi, M. Oliva, B. Langmead, T. Gagie, C. Boucher, MONI: A pangenomic index for finding maximal exact matches, J. Comput. Biol. 29 (2) (2022) 169–187.
[65] C. Boucher, T. Gagie, I. Tomohiro, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, M. Rossi, PHONI: Streamed matching statistics with multi-genome references, in: Proc. 31st Data Compression Conference (DCC), IEEE, 2021, pp. 193–202.
[66] C. Martínez-Guardiola, N.K. Brown, F. Silva-Coira, D. Köppl, T. Gagie, S. Ladra, Augmented thresholds for MONI, in: Proc. 33rd Data Compression Conference (DCC), IEEE, 2023, pp. 268–277.
[67] A. Goga, L. Depuydt, N.K. Brown, J. Fostier, T. Gagie, G. Navarro, Faster maximal exact matches with lazy LCP evaluation, in: Proc. Data Compression Conference (DCC), IEEE, 2024, pp. 123–132.
[68] O. Ahmed, M. Rossi, S. Kovaka, M.C. Schatz, T. Gagie, C. Boucher, B. Langmead, Pan-genomic matching statistics for targeted nanopore sequencing, Iscience 24 (6) (2021).
[69] O.Y. Ahmed, M. Rossi, T. Gagie, C. Boucher, B. Langmead, SPUMONI 2: improved classification using a pangenome index of minimizer digests, Genome Biol. 24 (1) (2023) 122.
[70] V.S. Shivakumar, O.Y. Ahmed, S. Kovaka, M. Zakeri, B. Langmead, Sigmoni: classification of nanopore signal with a compressed pangenome index, Bioinformatics 40 (Supplement 1) (2024) i287–i296.
[71] M. Zakeri, N.K. Brown, O.Y. Ahmed, T. Gagie, B. Langmead, Movi: a fast and cache-efficient full-text pangenome index, iScience 27 (12) (2024).
[72] G. Marçais, A.L. Delcher, A.M. Phillippy, R. Coston, S.L. Salzberg, A. Zimin, Aleksey, MUMmer4: A fast and versatile genome alignment system, PLoS Comput. Biol. 14 (1) (2018) 1–14.
[73] V. Mäkinen, D. Belazzougui, F. Cunial, A. Tomescu, Genome Scale Algorithm Design, 2nd Edition, Cambridge University Press, 2023.
[74] C. Boucher, D. Cenzato, Zs. Lipták, M. Rossi, M. Sciortino, *r*-indexing the eBWT, Inf. Comput. 298 (2024) 105155. https://doi.org/10.1016/J.IC.2024.105155
[75] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows-Wheeler transform, Theor. Comput. Sci. 387 (3) (2007) 298–312. https://doi.org/10.1016/J.TCS.2007.07.014
[76] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inf. Theory 22 (1) (1976) 75–81.
[77] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (3) (1977) 337–343.
[78] S. Deorowicz, S. Grabowski, Robust relative compression of genomes with random access, Bioinform. 27 (21) (2011) 2979–2986.
[79] C. Hoobin, S.J. Puglisi, J. Zobel, Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections, Proc. VLDB Endow. 5 (3) (2011) 265–273.
[80] K. Liao, M. Petri, A. Moffat, A. Wirth, Effective construction of relative Lempel-Ziv dictionaries, in: Proc. 25t International Conference on World Wide Web (WWW), ACM, 2016, pp. 807–816.
[81] S.J. Puglisi, B. Zhukova, Relative Lempel-Ziv compression of suffix arrays, in: Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 12303, Springer, 2020, pp. 89–96.
[82] S.J. Puglisi, B. Zhukova, Document retrieval hacks, in: Proc. 19th International Symposium on Experimental Algorithms (SEA), LIPIcs 190, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 12:1–12:12.
[83] P. Bille, I.L. Gørtz, S.J. Puglisi, S.R. Tarnow, Hierarchical relative Lempel-Ziv compression, in: Proc. 21st International Symposium on Experimental Algorithms (SEA), LIPIcs 265, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 18:1–18:16.
[84] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, S.J. Puglisi, A faster grammar-based self-index, in: Proc. 6th International Conference on Language and Automata Theory and Applications (LATA), LNCS 7183, Springer, 2012, pp. 240–251.
[85] H.H. Do, J. Jansson, K. Sadakane, W. Sung, Fast relative Lempel-Ziv self-index for similar sequences, Theor. Comput. Sci. 532 (2014) 14–30.

[86] G. Navarro, V. Sepúlveda, Practical indexing of repetitive collections using relative Lempel-Ziv, in: Proc. 29th Data Compression Conference (DCC), 2019, pp. 201–210.
[87] A. Conte, N. Cotumaccio, T. Gagie, G. Manzini, N. Prezza, M. Sciortino, Computing matching statistics on wheeler DFAs, in: Proc. 33rd Data Compression Conference (DCC), 2023, pp. 150–159.
[88] Y. Gao, Computing matching statistics on repetitive texts, in: Proc. 32nd Data Compression Conference (DCC), IEEE, 2022, pp. 73–82.
[89] A.C. Diseth, K. Heljanko, S.J. Puglisi, Massively parallel computation of matching statistics, in: Proc. 32nd International Symposium on String Processing and Information Retrieval (SPIRE), 2025. to appear.