Computing Matching Statistics on Repetitive Texts

Younan Gao
Dalhousie University
Halifax, Canada
yn803382@dal.ca

Abstract
Computing the matching statistics of a string $P[1..m]$ with respect to a text $T[1..n]$ is a fundamental problem which has application to genome sequence comparison. In this paper, we study the problem of computing the matching statistics upon highly repetitive texts. We design three different data structures that are similar to LZ-compressed indexes. The space costs of all of them can be measured by $\gamma$, the size of the smallest string attractor [STOC’2018] and $\delta$, a better measure of repetitiveness [LATIN’2020].

Introduction

The matching statistics, MS, of a pattern $P[1..m]$ with respect to a text $T[1..n]$ is an array of $m$ integers such that the $i$-th entry $\text{MS}[i]$ stores the length of the longest prefix of $P[i..m]$ that occurs in $T$. For example, given that $T = “aaabbbcc”$ and $P = “ccabb”$, matching statistics MS of $P$ w.r.t. $T$ stores an array of 5 integers, $[2, 1, 3, 2, 1]$. Originally, the concept, matching statistics, was introduced by Chang and Lawler [1] to solve the approximate string matching problem, i.e., given text $T[1..n]$ and pattern $P[1..m]$, the problem asks for all the locations in the text where $P[1..m]$ appears, with at most $k$ differences (including substitutions, insertions, and deletions) being allowed, where $k$ is not necessarily a constant. The approximate string matching plays an important role in computational genomics. In terms of sequence alignment, reads might not match the genome exactly because of the sequencing error, natural variance (i.e., differences in DNA among individuals of the same species), etc.; for that reason, the algorithms for the exact string matching might not be sufficient, and the approximate string matching is needed. Matching statistics is also useful in a variety of other applications such as finding the longest common substring between $P$ and $T$ [2].

A textbook solution [2] shows that a suffix tree data structure augmented with suffix links on the tree nodes can be used to compute matching statistics in $O(m \log \sigma)$ time, where $\sigma$ represents the size of the alphabet that $T$ is drawn from; the data structure uses $O(n)$ words of space. Ohlebusch et al. [3] solved this problem using a fully compressed text indexes built upon $T$, which consist of a wavelet tree data structure that supports the LF-Mapping and the backward search, a LCP-array, and a data structure that supports fast-navigating on a LCP-interval tree. Their indexes occupy $n \log \sigma + 4n + o(n \log \sigma)$ bits of space and achieve the same computing time as of the textbook solution. In the genomic databases, texts are always massive and highly repetitive; however, the compressed indexes based on statistical entropy might not capture repetitiveness [4]. Bannai et al. [5] considered to compute MS for a highly
repetitive text. They augmented a run-length BWT with $O(r)$ words of space, where $r$ is the number of runs in the BWT for $T$, and their indexes support computing MS in $O(m \lg \lg n)$ time, assuming that each element in $T$ can be accessed in $O(\lg \lg n)$ time. Let $z$ denote the number of phrases in Lempel-Ziv parsing (LZ). It has been proved that $r = O(z \lg^2 n)$ holds for every text $T$ [6]. However, to our knowledge, LZ-based indexes on computing MS have not been known prior to this work.

Recently, new compressibility measures such as $\gamma$, the size of the smallest string attractor, and $\delta$, a better measure of repetitiveness have been proposed. Both new measures better capture the compressibility of repetitive strings. It has been proved that $\delta \leq \gamma \leq z = O(\delta \lg \frac{n}{\delta})$ [7,8]. In this paper, we design the first string attractor based indexes (which is also workable upon LZ-parsing) to support computing matching statistics; the space cost of the indexes is measured by $\gamma$ and $\delta$. The computation time using string attractor based indexes might not be as efficient as the one using BWT-based indexes, but the indexes in the prior category always have an advantage of space cost.

Our Results. The results can be summarized as Theorem 1. In the first solution, we apply a data structure similar to LZ-compressed indexes. Instead of using LZ parsing, we define the phrases upon the smallest string attractor. We store a Patricia tree for the reversed phrases and another one for the suffixes following the phrase boundaries. To access text $T[1..n]$ within compressed space, we apply the string indexing data structure by Kociumaka et al. [8], whose space cost is measured by $\delta$. We give a simple and practical algorithm that reduces the problem of computing MS into $O(m^2)$ times of 2D orthogonal range predecessor queries upon $\gamma$ points on a grid. In the second solution, we apply the data structure framework by Abedin et al. [9]. Originally, they used the framework to find the longest common substring (LCS) between $P$ and $T$. Naively, computing MS of $P[1..m]$ can be reduced to $m$ times of LCS queries. Given that each LCS query can be computed in $O(m \lg \gamma \lg \lg \gamma)$ time, the naive method would take $O(m \cdot m \cdot \lg \gamma \lg \lg \gamma)$ time. We adjust their framework to computing MS and improve the query time to be $O(m \cdot (m + \lg \gamma \lg \lg \gamma))$. Theorem 1 The matching statistics of a $P[1..m]$ with respect to a text $T[1..n]$ can be computed (i) in $O(m^2 \lg^\epsilon \gamma + m \lg n)$ time using an $O(\delta \lg \frac{n}{\delta})$ word space data structure, or (ii) in $O(m^2 + m \lg \gamma \lg \lg \gamma + m \lg n)$ time using $O(\gamma \lg \gamma + \delta \lg \frac{n}{\delta})$ word space data structure, in which $\epsilon$ is any small positive constant, $\gamma$ is the size of the smallest string attractor, and $\delta$ is $\max\{S(k)/k, 1 \leq k \leq n\}$, where $S(k)$ denotes the number of distinct $k$-length sub-strings of $T$.

If text $T[1..n]$ is drawn from constant-size alphabet, we can further improve the computation time to be $O(m^2 + m \lg n)$ time using an $O(\gamma \lg \gamma + \delta \lg \frac{n}{\delta})$ word data structure. The third solution can be achieved by combining the first and second solution: i) when $m = \Omega(\lg \gamma \lg \lg \gamma)$, we can directly apply the second solution to achieve the target bound for the query time; ii) otherwise, we update the first solution using the technique solving the ball inheritance problem [10] to improve the query
time without decreasing the space cost. Due to the space limitation, the proof of the third solution is deferred to the full version of this paper.

Preliminaries

This section introduces the notations and the previous results used throughout this paper. Let $\epsilon$ denote any small positive constant, and all problems are studied under the standard word RAM model.

Compressibility Measures. We give the precise definitions of the compressibility measures $\gamma$ and $\delta$ that are mentioned before.

**Definition 1** [7] A string attractor of a string $T[1..n]$ is a set of $\gamma'$ positions $\Gamma' = \{j_1, \cdots, j_{\gamma'}\}$ such that every substring $T[i..j]$ has an occurrence $T[i'..j'] = T[i..j]$ with $j_k \in [i', j']$ for some $j_k \in \Gamma'$.

Let $\Gamma^*$ denote $\{1, \Gamma, n\}$, where $\Gamma$ denotes the smallest size string attractor storing positions sorted increasingly. For each $2 \leq i \leq |\Gamma^*|$, we call substring $T[\Gamma^*[i-1]..\Gamma^*[i]]$ a parsing phrase. Let $\gamma$ denote the size of $\Gamma$. It follows that given any substring $T[i..j]$, there must be an occurrence $T[i'..j'] = T[i..j]$ such that $T[i'..j']$ crosses the phrase boundary. Kociumaka et al. [8] defined a new measure $\delta$, which is even smaller than $\gamma$. Furthermore, measure $\delta$, different from $\gamma$, can be computed in linear time.

**Definition 2** [8] Let $d_k(S)$ be the number of distinct length-$k$ sub-strings in $S$. Then $\delta = \max\{d_k(S)/k : k \in [1..n]\}$.

Induced-Check and Find Partner. Let $T_1$ and $T_2$ be two trees on the same set of $n$ leaves. A node from $T_1$ and a node from $T_2$ are induced together if they have a common leaf descendant [11]. The partner operation [9] is defined upon the inducing relationship.

**Definition 3** [9] Given a pair of trees $T_1$ and $T_2$, the partner of a node $x \in T_1$ w.r.t a node $y \in T_2$, denoted by $\text{partner}(x/y)$, is the lowest ancestor, $y'$, of $y$ such that $x$ and $y'$ are induced. Likewise, $\text{partner}(y/x)$ is the lowest ancestor, $x'$, of $x$ such that $x'$ and $y$ are induced.

It has been proved that the induced relationship can be solved using 2D orthogonal range emptiness queries, while finding the partner can be reduced to 2D orthogonal range predecessor/successor queries.

**Lemma 1** [9] (Induced-Check). Given two nodes $x, y$, where $x \in T_1$ and $y \in T_2$, we can check if they are induced or not i) in $O(\lg \lg n)$ time using an $O(n \lg \lg n)$ word space structure, or ii) in $O(\lg \epsilon n)$ time using an $O(n)$ word space structure.

**Lemma 2** [9] (Find Partner). Given two nodes $x, y$ where $x \in T_1$ and $y \in T_2$, we can find $\text{partner}(x/y)$ as well as $\text{partner}(y/x)$ i) in $O(\lg \lg n)$ time using an $O(n \lg \lg n)$ word space structure, or ii) in $O(\lg \epsilon n)$ time using an $O(n)$ word space structure.

1See https://arxiv.org/pdf/2111.00376.pdf for the full version of this paper.
String Indexing. Recently, Kociumaka et al. [8] showed that within \( O(\delta \lg \frac{n}{\delta}) \) words of space, one can represent and index a string of \( n \) characters.

**Lemma 3** [8] Given a string \( T[1..n] \) with measure \( \delta \), one can build an \( O(\delta \lg \frac{n}{\delta}) \) word space data structure in \( O(n \lg n) \) expected preprocessing time to support retrieving any substring \( T[i..i+\ell] \) in \( O(\ell + \lg n) \) worst-case time.

Computing MS within \( O(\delta \lg \frac{n}{\delta}) \) Words of Space

This section presents our most space-efficient solution for computing MS. The data structure is similar to LZ-compressed indexes used for computing Longest Common Substrings [11], but instead of using phrases in LZ parse, we use the parsing phrases defined upon the smallest size string attractor of \( T[1..n] \): we store one Patricia tree \( T_{rev} \) for the reversed parsing phrases; we store another \( T_{suf} \) for the suffixes of \( T \) starting at different phrase boundaries. Those \( \gamma \) phrases, \( T[i+1..j] \), and their corresponding suffixes, \( T[j+1..n] \), are sorted in the lexicographic order, respectively. We construct a \( \gamma \times \gamma \) grid: we add a point, \((x, y)\), on the grid iff the lexicographically \( x \)-th phrase \( T[i+1..j] \) is followed by the lexicographically \( y \)-th suffix \( T[j+1..n] \) in text \( T \); we assign the \( x \)-coordinate of the point to reversed phrase\(^2\) \( T[i+1..j]^{rev} \) and the \( y \)-coordinate of the point to the suffix \( T[j+1..n] \). It follows that all points on the grid are in rank space, i.e., they have coordinates on the integer grid \([\gamma]^2 = \{1, 2, \cdots, \gamma\}^2\).

We store a linear space data structure implemented by part ii) of Lemma 1 to support 2D orthogonal range emptiness queries as induced-check and a linear space data structure by part ii) of Lemma 2 to support 2D orthogonal range predecessor/successor queries as partner-finding upon the \( \gamma \) points on the grid, respectively. Finally, we need a data structure supporting substring queries in \( T \), implemented by Lemma 3. Storing the string attractor, both Patricia trees, and the data structures for orthogonal range searching uses \( O(\gamma) \) words of space, while the string indexing data structure by Lemma 3 requires \( O(\delta \lg \frac{n}{\delta}) \) words. Overall, the space cost is \( O(\delta \lg \frac{n}{\delta}) \) words, since \( \gamma = O(\delta \lg \frac{n}{\delta}) \) [8].

As the pattern has \( m \) entries, \( P \) can be partitioned into \( m-1 \) different prefix and suffix pairs—that is, \( P[1..i] \) and \( P[i+1..m] \), for each \( 1 \leq i \leq m-1 \). For each prefix and suffix pair, we can use Lemma 4 to find the loci of \( (P[1..i])^{rev} \) in \( T_{rev} \) and the loci of \( P[i+1..m] \) in \( T_{suf} \), respectively.

**Lemma 4** Given a pattern \( P[1..m] \), for all \( 1 \leq i \leq m-1 \), we can find the longest common prefix (LCP) of \( (P[1..i])^{rev} \) and the path label of the node where the search in \( T_{rev} \) terminates, and the LCP of \( P[i+1..m] \) and the path label of the node where the search in \( T_{suf} \) terminates in \( O(m^2 + m \lg n) \) time.

*Proof:* For each \( 1 \leq i \leq m-1 \), a query need to access \( T \) to check that the path labels of the nodes where the searches terminate are really prefixed by some prefixes of \( (P[1..i])^{rev} \) and \( P[i+1..m] \), which can be solved by the substring queries using Lemma 3. As there are \( m-1 \) different pairs of \( (P[1..i])^{rev} \) and \( P[i+1..m] \), and the

\(^2\)Given a string \( s = \text{"abcd"} \), the formula \( s^{rev} \) represents the string \( \text{"dcba"} \).
LCPs can be found in

at line 6 is operated

node; Otherwise,

on the path from

Given any tree node

P

of (T)
in

total number of characters that each pair of them contain is m, the searching time is

O(m^2 + m \log n).

Next, we present the query algorithm. For 1 ≤ i ≤ m − 1, we search for (P[1..i])^{rev}

in T_{rev} and for P[i + 1..m] in T_{suf}; access T to find the longest common prefix (LCP)
of (P[1..i])^{rev} and the path label of the node where the search in T_{rev} terminates, and the
LCP of P[i + 1..m] and the path label of the node where the search in T_{suf} terminates;
take \text{loci}_1 and \text{loci}_2 to be the loci of those LCPs. For each node, v, on the path from
\text{loci}_1 to the root node of T_{rev}, we retrieve the lowest ancestor, u, of \text{loci}_2 in T_{suf}
such that u is induced together with v, i.e., u = partner(v/\text{loci}_2).

Given any tree node w, we use str(w) to denote the path label of w.\(^3\) Observe that

\((\text{str}(v))^{rev} \cdot \text{str}(u)\) (resp. \(P[i - \text{len(str}(v)) + 1..i + \text{len(str}(u))]\)) might be the longest
prefix of \(P[i - \text{len(str}(v)) + 1..m]\) that occurs in T; thus, we set \(MS[i - \text{len(str}(v)) + 1]\)
to be \(\text{len(str}(v)) \cdot \text{str}(u)\) temporarily. For each \(k \in [1..m - 1]\) and \(k \leq j \leq m - 1\),
the longest prefix of \(P[k..m]\) might appear somewhere in T crossing the phrase boundary
whose immediately left phrase ends with \(P[j]\) and immediately right phrase starts with
\(P[j + 1]\); since there are at most \(m - k + 1\) different types of phrase boundaries, entry
\(MS[k]\) will finally store the length of the longest prefix of \(P[k..m]\) that appears in T
after at most \(m - k + 1\) times of updates. The algorithm is shown in ComputingMS1.

We analyze the query time of the algorithm: As shown in Lemma 4, all locus of
LCPs can be found in \(O(m^2 + m \log n)\) time; for each 1 ≤ i ≤ m − 1, the while loop
at line 6 is operated \(O(i)\) times, and all \(O(i)\) iterations will call totally \(O(i)\) times

\(^3\)If v (resp. u) is the loci, then the longest matched prefix of (P[1..i])^{rev} (resp. P[i + 1..m]) might be a proper substring of the path label of v (resp. u). In that case, we let str(v) (resp. str(u)) denote the longest matched prefix of (P[1..i])^{rev} (resp. P[i + 1..m]). And |str(root)| is always 0.

\(^4\)For example, “abc”. “efg” = “abcefg”.

\(^5\)W.l.o.g., we assume that \(m > 1\). MS[m] is set to 1, if the loci of P[m] on T_{suf} is a non-root
node; Otherwise, MS[m] is set to 0.

\[\text{\$ComputingMS1(P[1..m], T_{suf}, T_{rev})}\]

1: MS[1..m] \leftarrow \{0 \cdots 0\}
2: for i = 1, 2, \ldots, m − 1 do
3: Find \text{loci}_1 of (P[1..i])^{rev} in T_{rev}
4: Find \text{loci}_2 of (P[i + 1..m]) in T_{suf}
5: v \leftarrow \text{loci}_1
6: \textbf{while} v \text{ is not NULL do}
7: \quad u \leftarrow \text{partner}(v/\text{loci}_2)
8: \quad vp \leftarrow \text{parent}(v)
9: \quad j \leftarrow \text{len(str}(v))
10: \quad \textbf{while} j > \text{len(str}(vp)) \text{ do}
11: \quad \quad \ell \leftarrow j + \text{len(str}(u))
12: \quad \quad \textbf{if} \ \ell > MS[i - j + 1] \text{ then}
13: \quad \quad \quad MS[i - j + 1] \leftarrow \ell
14: \quad j \leftarrow j - 1
15: v \leftarrow vp

\[\text{\$ComputingMS2(S[1..m])}\]

1: MS[1..m] \leftarrow \{0 \cdots 0\}
2: max \leftarrow 0
3: for i = 1, 2, \ldots, m − 1 do
4: \quad \textbf{if} max \leq S[i] \text{ then}
5: \quad \quad max \leftarrow S[i]
6: MS[i] \leftarrow max
7: \quad \textbf{if} max > 0 \text{ then}
8: \quad \quad max \leftarrow max - 1
of partner-finding queries and fill at most \( i \) entries of \( MS \); if only \( O(\gamma) \) words of space is allowed, each partner-finding query requires \( O(\lg^\epsilon \gamma) \) time as shown in part ii) of Lemma 2. The overall query time is \( O(m^2 + m \lg n + \sum_{i=1}^{m-1} O(i \cdot \lg^\epsilon \gamma + i)) = O(m^2 \lg^\epsilon \gamma + m \lg n) \). The first solution completes.

Towards Improving the Computing Time

In this section, we are trying to get rid of factor \( \lg^\epsilon \gamma \) from term \( m^2 \lg^\epsilon \gamma \) shown in the computing time before. As a result, the space cost of the new data structure gets worse slightly. Before showing our new solutions, we introduce a new definition locally potential maximal exact matching (LPMEM):

**Definition 4** Given a phrase boundary \( k' \in \Gamma \), we refer to a substring \( P[i\ldots j] \) as a locally potential maximal exact matching (LPMEM) that crosses the phrase boundary at position \( k' \) if substring \( P[i\ldots j] \) with an occurrence \( T[i'\ldots j'] \) such that \( i' \leq k' < j' \) holds and the occurrence can neither be extended to the left nor to the right.

Let \( v \) (resp. \( u \)) denote an ancestor node of \( \text{loci}_1 \) (resp. \( \text{loci}_2 \)) in \( T_{\text{rev}} \) (resp. \( T_{\text{suf}} \)); let \( \text{Path}(v, \text{loci}_1, T_{\text{rev}}) \) denote the path between \( v \) and \( \text{loci}_1 \) on \( T_{\text{rev}} \); let \( \text{Path}(u, \text{loci}_2, T_{\text{suf}}) \) denote the path between \( u \) and \( \text{loci}_2 \) on \( T_{\text{suf}} \). It follows that if i) \( v \) and \( u \) are induced together; ii) the child of \( v \) on \( \text{Path}(v, \text{loci}_1, T_{\text{rev}}) \) does not induce with node \( u \) in \( T_{\text{suf}} \); iii) and the child of \( u \) on \( \text{Path}(u, \text{loci}_2, T_{\text{suf}}) \) does not induce with node \( v \) in \( T_{\text{rev}} \), then \( v \) and \( u \) together induce a LPMEM, which is \((\text{str}(v))_{\text{rev}}.\text{str}(u)\).

Basic Properties of LPMEMs. We discuss the properties of LPMEMs. Those properties will be useful for designing the data structures and the query algorithm for the second solution. For simplicity, we call all sub-strings of \( P[1..m] \) that appear as LPMEM’s in text \( T[1..n] \) the LPMEMs of \( P \).

**Lemma 5** Given a pattern \( P[1..m] \) and all \( \text{occ} \) LPMEMs of \( P \), the matching statistics of \( P \) can be computed in \( O(\text{occ} + m) \) time.

**Proof:** Assume that all \( \text{occ} \) LPMEMs have been found, and each LPMEM can be represented by its starting position, \( i \), in \( P \) and its length, \( \ell(i) \). Let \( S[1..m] \) be an array, in which entry \( S[i] \), for each \( 1 \leq i \leq m \), stores \( \ell(i) \) if \( P[i, i + \ell(i) - 1] \) is a LPMEM. If there are multiple LPMEMs sharing the same starting position, \( i \), in \( P \), \( S[i] \) stores the largest length. It follows that for each \( 1 \leq i \leq m \), \( MS[i] \) is equal to \( \max(MS[i - 1] - 1, S[i]) \), where \( MS[0] \) is 0. See ComputingMS2 for the algorithm. Computing array \( S[1..m] \) takes \( O(\text{occ} + m) \) time and ComputingMS2 requires \( m \) primitive steps; hence, the computation time is \( O(\text{occ} + m) \). Note that \( ^5 \text{occ} \leq m(m - 1)/2 \).

\(^5\)A pattern \( P \) with \( m \) characters can have at most \( \binom{m}{2} \) sub-strings that appear as LPMEM’s in text \( T[1..n] \).

78
We compute the heavy path decomposition \cite{12} of \( T_{rev} \) and \( T_{suf} \) mentioned before. For a node \( u \) on a heavy path \( H \), let \( hp_{\text{root}}(u) \) (resp. \( hp_{\text{leaf}}(u) \)) denote the highest (resp. lowest) node of \( H \). We call the highest node of each heavy path light. As \( T_{rev} \) and \( T_{suf} \) each has \( \gamma \) leaves, a path from the root to any leaf on \( T_{rev} \) or \( T_{suf} \) traverses at most \( O(\lg \gamma) \) light nodes. We give a new definition special skyline node list borrowing the ideas of skyline node list from \cite{11} and special nodes from \cite{9}.

Definition 5 For each light node, \( w \in T_{suf} \), we identify a set, \( \text{SpecialLeaves}(w) \), of leaf nodes in \( T_{rev} \) and a set, \( \text{SpecialSkylineList}(w) \), of internal nodes in \( T_{rev} \) as follows: leaf node \( l \in T_{rev} \) is special iff \( l \) and \( w \) are induced with each other; we define special skyline node \( v \in \text{SpecialSkylineList}(w) \) if (i) \( v \) is a proper ancestor of \( \text{lca}(x,y) \) for some special leaves \( x \) and \( y \), (ii) and the child of \( v \) on \( \text{Path}(v, \text{lca}(x,y), T_{rev}) \) does not induce with \( \text{partner}(v/hp_{\text{leaf}}(w)) \). Following \cite{9}, we identify set \( \text{Special}(w) \) of nodes in \( T_{rev} \), consisting of the special leaves of \( w \) and their lowest common ancestors.

As shown before, for each \( 1 \leq i \leq m - 1 \), we search for \( (P[1..i])^{rev} \) in \( T_{rev} \) and for \( P[i+1..m] \) in \( T_{suf} \); take \( \text{loci}_1(i) \) and \( \text{loci}_2(i) \) to be the locus of those LCPs. Let \( u \) and \( v \) denote some ancestors nodes of \( \text{loci}_1(i) \) and \( \text{loci}_2(i) \) on \( T_{rev} \) and \( T_{suf} \), respectively; let \( \text{root}_1 \) (resp. \( \text{root}_2 \)) denote the root node of \( T_{rev} \) (resp. \( T_{suf} \)). Henceforth, if \( u \) and \( v \) induce a LPMEM\(^6\), \( (\text{str}(u))^{rev}.\text{str}(v) \), then \( u \) and \( v \) are referred to as the the beginning and the ending nodes of that LPMEM. Observe that for the LPMEMs crossing the phrase boundary between \( P[i] \) and \( P[i+1] \), their beginning nodes stay on \( \text{Path}(\text{partner}(\text{loci}_2(i)/\text{loci}_1(i)), \text{loci}_1(i), T_{rev}) \), and their ending nodes stay on \( \text{Path}(\text{partner}(\text{loci}_1(i)/\text{loci}_2(i)), \text{loci}_2(i), T_{suf}) \)

Let \( w_1, \ldots, w_k \) denote a sequence of light nodes on \( \text{Path}(\text{root}_2, \text{loci}_2(i), T_{suf}) \), sorted increasingly by the node depths, such that \( \text{partner}(\text{loci}_1(i)/\text{loci}_2(i)) \) is contained in the heavy path rooted by \( w_1 \), and \( w_k \) is the lowest light node above \( \text{loci}_2(i) \). Similarly, let \( t_1, \ldots, t_k \) denote the nodes on \( \text{Path}(\text{root}_2, \text{loci}_2(i), T_{suf}) \) such that \( t_k = \text{loci}_2(i) \) and \( t_h = \text{parent}(w_{h+1}) \) for \( h < k \). Let \( \alpha_f \) be \( \text{partner}(t_f/\text{loci}_1(i)) \) and \( \beta_f \) be \( \text{partner}(w_f/\text{loci}_1(i)) \), for each \( 1 \leq f \leq k \).

Lemma 6 For each \( 1 \leq f \leq k \), \( \beta_f \) and \( \text{partner}(\beta_f/\text{loci}_1(i)) \) induce a LPMEM, and \( \alpha_f \) and \( \text{partner}(\alpha_f/\text{loci}_1(i)) \) induce a LPMEM.

\textbf{Proof:} Let \( u \) denote \( \text{partner}(\alpha_f/\text{loci}_1(i)) \) in \( T_{suf} \). Since \( \alpha_f \) in \( T_{rev} \) and \( t_f \) in \( T_{suf} \) are induced together, \( u \) must be in the sub-tree of \( t_f \). Suppose that child \( c \) of \( \alpha_f \) on \( \text{Path}(\alpha_f, \text{loci}_1(i), T_{rev}) \) is induced with \( u \). As \( t_f \) is an ancestor of \( u \), \( t_f \) in \( T_{suf} \) and \( c \) in \( T_{rev} \) must be induced together, which contradicts with the claim that \( \alpha_f \) is \( \text{partner}(t_f/\text{loci}_1(i)) \). Therefore, \( c \) can not be induced with \( u \) in \( T_{suf} \). Following the definition of \( \text{partner} \), the child of \( u \) on \( \text{Path}(u, \text{loci}_2(i), T_{suf}) \) cannot be induced with \( \alpha_f \); hence, \( u \) and \( \alpha_f \) induce a LPMEM. The claim that \( \text{partner}(\beta_f/\text{loci}_1(i)) \) and \( \beta_f \) induce a LPMEM follows a similar argument. \( \Box \)

\(^6\)To compute the matching statistics, reporting a LPMEM \( (\text{str}(u))^{rev}.\text{str}(v) \) verbatim is unnecessary. What we need are its starting position in \( P[1..m] \), which is \( i - \text{len} (\text{str}(u)) + 1 \), and the length of the LPMEM, which is \( \text{len}(\text{str}(u)) + \text{len}(\text{str}(v)) \).
Lemma 7  Given a node \( u \) on \( T_{rev} \) and a node \( v \) on \( T_{suf} \) such that \( u \) and \( v \) induce a LPMEM, if \( v \in \text{Path}(w_f,t_f,T_{suf}) \) for some \( 1 \leq f \leq k \), then \( u \in \text{Path}(\alpha_f,\beta_f,T_{rev}) \).

Proof:  The proof is similar to the one shown as [9, Lemma 12]. Suppose \( u \) is a proper ancestor of \( \alpha_f \). Since \( \alpha_f \) and \( t_f \) are induced together, node \( v \), as an ancestor of \( t_f \), is also induced with \( \alpha_f \). Due to this, \( u \) and \( v \) cannot induce a LPMEM, which generates a contradiction; therefore, \( u \) must be in the sub-tree of \( \alpha_f \). Suppose that \( u \) is in the proper sub-tree of \( \beta_f \). Since \( u \) and \( v \) are induced together, and since \( v \) is in the sub-tree rooted by \( w_f \), \( u \) and \( w_f \) are induced together, which contradicts with the claim that \( \beta_f \) is \( \text{partner}(w_f/\text{loci}_1(i)) \). Therefore, \( u \) must be an ancestor of \( \beta_f \). □

Lemma 8  [9, Lemma 14] For each \( 1 \leq f \leq k \) and any \( x \in \text{Path}(\alpha_f,\beta_f,T_{rev})/\{\alpha_f\} \), \( \text{partner}(x/\text{loci}_2(i)) = \text{partner}(x/t_f) = \text{partner}(x/\text{hp}_\text{leaf}(w_f)) \) always holds.

For any node \( x \in \{\text{SpecialSkylineList}(w_f)\}((\alpha_f \cup \beta_f)) \), it follows that \( x \) and \( \text{partner}(x/\text{hp}_\text{leaf}(w_f)) \) induce a LPMEM because of Lemma 8 and Definition 5.

Lemma 9  For each \( v \in \text{SpecialSkylineList}(w_f) \), it follows that \( v \in \text{Special}(w_f) \).

Proof:  Let \( x \) and \( y \) be a pair of special leaves under \( v \) such that child \( c \) of \( v \) on \( \text{Path}(v,\text{lca}(x,y),T_{rev}) \) does not induce with \( \text{partner}(v/\text{hp}_\text{leaf}(w_f)) \). Since \( \text{partner}(v/\text{hp}_\text{leaf}(w_f)) \) and \( v \) are induced together by some special leaf \( \ell \) under the sibling node of \( c \), it follows that \( \text{lca}(x,\ell) \) or \( \text{lca}(y,\ell) \) is \( v \), and \( v \in \text{Special}(w_f) \). □

Lemma 10  Given a node \( u \) on \( \text{Path}(\alpha_f,\beta_f,T_{rev}) \) and an ancestor node \( v \) of \( \text{loci}_2(i) \) on \( T_{suf} \), if \( u \) and \( v \) induce a LPMEM, and \( u \notin \{\text{SpecialSkylineList}(w_f)\}((\alpha_f \cup \beta_f)) \), then \( u = \text{lca}(\ell,\ell') \) for some pair of \( \ell,\ell' \in \text{SpecialLeaves}(w_f) \).

Proof:  Since \( \beta_f \) and \( w_f \) are induced together, and since \( u \) is a proper ancestor of \( \beta_f \), there is a special leaf \( \ell \) as the common descendant of \( \beta_f \) and \( u \). As \( u \) and \( v \) induce a LPMEM, and as \( u \) is not \( \alpha_f \), \( v = \text{partner}(u/\text{loci}_2(i)) = \text{partner}(u/t_f) \) by Lemma 8. Since \( u \) and \( w_f \) are induced together by \( \ell \), \( v \) must be a descendant of \( w_f \). There is at least one leaf \( \ell' \) under \( u \) that makes \( u \) and \( v \) induced with each other, and \( \ell' \) is a special leaf because it is induced with \( w_f \). If \( \ell' \) and \( \ell \) are the same, then \( \beta_f \) and \( v \) are induced together, which contradicts with the claim that \( u \) and \( v \) induce a LPMEM. There are at least two special leaves \( \ell \) and \( \ell' \) under \( u \). Since \( u \notin \{\text{SpecialSkylineList}(w_f)\} \), \( u = \text{lca}(\ell,\ell') \). The proof completes. □

The Second Solution.  We apply the induced sub-tree defined in [9, Definition 18] to support finding LPMEMs. An induced sub-tree \( T_{rev}(w) \) w.r.t a light node \( w \in T_{suf} \) is a tree having exactly \( |\text{Special}(w)| \) nodes such that i) each node \( l \in \text{Special}(w) \) has a corresponding node \( l' \) in \( T_{rev}(w) \); and ii) for each pair of \( \ell,\ell' \in \text{Special}(w) \), node \( \text{lca}(\ell,\ell') \) in \( T_{rev} \) has a corresponding node, as \( \text{lca}(\ell,\ell') \) in \( T_{rev}(w) \). To support finding LPMEMs, we revise the induced sub-trees as follows: For each internal node \( \hat{v} \) of \( T_{rev}(w) \), we maintain a pointer \( e_0 \) pointing to its lowest proper ancestor \( \hat{v}' \) (if exists) that belongs to \( \text{SpecialSkylineList}(w) \) and a pointer \( e_1 \) pointing to its
corresponding node \( v \) in \( T_{rev} \); for each special skyline node \( \hat{v} \) of \( T_{rev}(w) \), we maintain a pointer \( e_2 \) pointing to \( \text{partner}(v/\text{hp_leaf}(w)) \) in \( T_{suf} \).

Since \( \sum_w |\text{Special}(w)| = O(\gamma \lg \gamma) \) for all light nodes \( w \in T_{suf} \), all revised induced sub-trees totally use \( O(\gamma \lg \gamma) \) words of space. Abedin et al. [9, Lemma 19] showed that given a node \( \ell \in \text{Special}(w) \), one can find its corresponding node \( \ell' \) in \( T_{rev}(w) \) in \( O(\lg \gamma) \) time by maintaining an \( O(\gamma \lg \gamma) \) word data structure. In addition, the data structures introduced in the first solution are also required, occupying extra \( O(\delta \lg \frac{n}{\gamma}) \) words of space. The overall space cost is \( O(\delta \lg \frac{n}{\gamma} + O(\gamma \lg \gamma)) \) words.

We show how to find LPMEMs between \( P \) and \( T \). By Lemma 4, we can find locus \( \text{loci}_1(i) \) and \( \text{loci}_2(i) \) on \( T_{rev} \) and \( T_{suf} \) in \( O(m^2 + m \lg n) \) time for all \( 1 \leq i \leq m - 1 \). We compute \( \text{partner}(\text{loci}_1(i)/\text{loci}_1(i)) \) and \( \text{partner}(\text{loci}_1(i)/\text{loci}_2(i)) \).

Each \( \text{partner} \) operation takes \( O(\lg \gamma) \) time by part i) of Lemma 2. If \( \text{loci}_1(i) \) and \( \text{loci}_2(i) \) are induced with each other, then there is only one LPMEM crossing the phrase boundary between \( P[i] \) and \( P[i+1] \), which is \( \text{str}(\text{loci}_1(i))^{rev}.\text{str}(\text{loci}_2(i)) \), and we continue to search for LPMEMs crossing the phrase boundary between \( P[i+1] \) and \( P[i+2] \). Otherwise, we iterate through \( \text{Path}(\text{root}_2, \text{loci}_2(i), T_{suf}) \) to find the light nodes \( w_1, \ldots, w_k \) and nodes \( t_1, \ldots, t_k \) as described before; compute \( \alpha_f \) and \( \beta_f \) for all \( 1 \leq f \leq k \). Since \( \beta_1 = \text{partner}(w_1/\text{loci}_1(i)) = \text{loci}_1(i) \), and since \( \alpha_k = \text{partner}(t_k/\text{loci}_1(i)) = \text{partner}(\text{loci}_2(i)/\text{loci}_1(i)) \), each of those LPMEMs has its beginning node on \( \text{Path}(\beta_1, \alpha_k, T_{rev}) \). We traverse the path from \( \beta_1 \) to \( \alpha_k \). In general, the beginning nodes on the sub-path from \( \beta_f \) to \( \alpha_f \) consist of 3 parts: \( \alpha_f, \beta_f, \) and some special nodes between \( \alpha_f \) (excluding \( \beta_f \)) and \( \beta_f \) (excluding \( \beta_f \)). Finding LPMEMs induced by \( \alpha_f \) and \( \text{partner}(\alpha_f/\text{loci}_2(i)) \) or \( \beta_f \) and \( \text{partner}(\beta_f/\text{loci}_2(i)) \) is straightforward, taking \( O(\lg \gamma) \) time. There are \( 2k \) such LPMEMs, and finding all of them takes \( O(\lg \gamma) \) time, since \( k = O(\lg \gamma) \).

It remains to find the LPMEMs with their beginning nodes between \( \alpha_f \) (excluding \( \alpha_f \)) and \( \beta_f \) (excluding \( \beta_f \)). Given an internal tree node \( x \), we use \( \text{1Most}(x) \) (resp. \( \text{rMost}(x) \)) to denote the index of the leftmost (resp. rightmost) leaf descendant of \( x \). Since \( \beta_f \) is \( \text{partner}(w_f/\text{loci}_1(i)) \), there exists at least a special leaf of \( T_{rev} \) as a descendant of \( \beta_f \) that belongs to \( \text{SpecialLeaves}(w_f) \), and we use \( \ell \) to denote the leftmost one. Let \( \ell' \) denote the rightmost special leaf among the first \( \text{1Most}(x) - 1 \) leaves of \( T_{rev} \) and \( \ell'' \) denote the leftmost special leaf on the right-hand side of the \( \text{rMost}(w_f)-th \) leaf of \( T_{rev} \). It follows that the lowest special node above \( \beta_f \), denoted by \( v \), is the lower one between \( \text{lca}(\ell', \ell) \) and \( \text{lca}(\ell, \ell'') \). Once \( v \) is found, we check whether it is the beginning node of some LPMEM: If the child of \( v \) on \( \text{Path}(v, \text{loci}_1(i), T_{rev}) \) does not induce with \( \text{partner}(v/\text{loci}_1(i)) \), then we report a LPMEM induced by \( v \) and \( \text{partner}(v/\text{loci}_1(i)) \). Since \( v \) is a special node, we find its corresponding node \( \hat{v} \) on \( T_{rev}(w) \) in \( O(\lg \gamma) \) time by [9, Lemma 19]. Following the pointer \( e_0 \) stored at \( \hat{v} \), we can find the lowest special skyline node, \( \hat{v}' \), above \( \hat{v} \). Note that \( \hat{v}' \) is the beginning node of some other LPMEM. We can find that LPMEM following the pointers \( e_1 \) and \( e_2 \) stored at \( \hat{v}' \) in \( O(1) \) time. We repeat this procedure to iterate over each special skyline node above \( \hat{v} \) until a node whose pointer \( e_1 \) pointing to \( \alpha_f \) is found. Each of \( \ell, \ell', \) and \( \ell'' \) can be found in \( O(\lg \gamma) \) time by 2D orthogonal range successor
queries\textsuperscript{7}, e.g., the leaf index of $\ell$ is the $x$-coordinate of the leftmost point in the query range $[l_{\text{Most}}(\beta_f), r_{\text{Most}}(\beta_f)] \times [l_{\text{Most}}(w_f), r_{\text{Most}}(w_f)]$. After finding the lowest special node $\hat{v}$, reporting the LPMEMs associated with SpecialSkylineList($w_f$) takes $O(\text{occ}(w_f))$ time, where $\text{occ}(w_f)$ denotes the number of reported LPMEMs. As there are $k$ different such different SpecialSkylineList($w_f$), finding $\text{occ}_i$ LPMEMs between loci$_1(i)$ and loci$_2(i)$ requires $O(k \lg \lg \gamma + \text{occ}_i) = O(\lg \gamma \lg \lg \gamma + \text{occ}_i)$ time, since $k = O(\lg \gamma)$. Considering there are $m - 1$ different pairs of locus, loci$_1(i)$ and loci$_2(i)$, finding all LPMEMs between $P$ and $T$ takes $O(m \lg \gamma \lg \lg \gamma + \text{occ})$ time. The overall query time is $O(m^2 + m \lg \gamma \lg \lg \gamma + m \lg n)$, since $\text{occ} = O(m^2)$. After finding all the LPMEMs, we can use Algorithm 2 to compute the matching statistics.

**Acknowledgments.** The author would like to thank Travis Gagie and Meng He for discussing various topics related to the compact data structures, and especially thank Travis for sharing this research topic as a course project.

**References**

\begin{itemize}
  \item [1] William I. Chang and Eugene L. Lawler, “Sublinear approximate string matching and biological applications,” *Algorithmica*, vol. 12, no. 4, pp. 327–344, 1994.
  \item [2] Dan Gusfield, “Algorithms on strings, trees, and sequences: Computer science and computational biology,” *Acm Sigact News*, vol. 28, no. 4, pp. 41–60, 1997.
  \item [3] Enno Ohlebusch, Simon Gog, and Adrian Kügel, “Computing matching statistics and maximal exact matches on compressed full-text indexes,” in *International Symposium on String Processing and Information Retrieval*. Springer, 2010, pp. 347–358.
  \item [4] Gonzalo Navarro and Veli Mäkinen, “Compressed full-text indexes,” *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, pp. 2–es, 2007.
  \item [5] Hideo Bannai, Travis Gagie, and I Tomohiro, “Refining the r-index,” *Theoretical Computer Science*, vol. 812, pp. 96–108, 2020.
  \item [6] Dominik Kempa and Tomasz Kociumaka, “Resolution of the burrows-wheeler transform conjecture,” in *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2020, pp. 1002–1013.
  \item [7] Dominik Kempa and Nicola Prezza, “At the roots of dictionary compression: string attractors,” in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, 2018, pp. 827–840.
  \item [8] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza, “Towards a definitive measure of repetitiveness,” in *Latin American Symposium on Theoretical Informatics*. Springer, 2021, pp. 207–219.
  \item [9] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V Thankachan, “The heaviest induced ancestors problem revisited,” in *Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
  \item [10] Timothy M Chan, Kasper Green Larsen, and Mihai Pătrașcu, “Orthogonal range searching on the ram, revisited,” in *Proceedings of the twenty-seventh annual symposium on Computational geometry*, 2011, pp. 1–10.
  \item [11] Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich, “Heaviest induced ancestors and longest common substrings,” *arXiv preprint arXiv:1305.3164*, 2013.
  \item [12] Dov Harel and Robert Endre Tarjan, “Fast algorithms for finding nearest common ancestors,” *siam Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.
\end{itemize}

\textsuperscript{7}The 2D orthogonal range successor query is also used for answering the \texttt{partner} operation.