Longest Common Prefixes with k-Mismatches & Applications

Hayam Alamro, Lorraine A.K. Ayad, Panagiotis Charalampopoulos*, Costas S. Iliopoulos, and Solon P. Pissis

Department of Informatics, King's College London, London, UK
[hayam.alamro,lorraine.ayad,panagiotis.charalampopoulos,
costas.iliopoulos,solon.pissis]@kcl.ac.uk

Abstract. We propose a new algorithm for computing the longest prefix of each suffix of a given string of length n over a constant-sized alphabet of size σ that occurs elsewhere in the string with Hamming distance at most k. Specifically, we show that the proposed algorithm requires time $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$ on average, where $R = \lceil (k + 2)(\log_\sigma n + 1)\rceil$, and space $\mathcal{O}(n)$. This improves upon the state-of-the-art average-case time complexity for the case when $k = 1$ [Manzini, SPIRE 2015] by a factor of $\log n/\log^3 \log n$. In addition, we show how the proposed technique can be adapted and applied in order to compute the longest previous factors under the Hamming distance model within the same complexities. In terms of real-world applications, we show that our technique can be directly applied to the problem of genome mappability.

1 Introduction

The longest common prefix (LCP) array is a commonly used data structure alongside the suffix array (SA). The LCP array stores the length of the longest common prefix between two adjacent suffixes of a given string as they are stored (in lexicographical order) in the SA [22]. A typical use of the combination of the SA and the LCP array is to simulate the suffix tree [30] functionality using less space [1]. This use has inspired many researchers to focus on engineering the construction of the LCP array [17].

However, there are many practical scenarios where the LCP array may be applied without making use of the SA. The LCP array provides us with essential information regarding repetitiveness in a given string and is therefore a useful data structure for analysing textual data in areas such as molecular biology, musicology, or natural language processing.

It is also quite common to account for potential alterations within sequences. For example, they can be the result of DNA replication or sequencing errors in DNA sequences. Alterations may also be introduced in the scope of plagiarism attempts in natural languages. In this context, it is natural to define the longest

* Supported by the Graduate Teaching Scholarship scheme of the Department of Informatics at King's College London and by the A.G. Leventis Foundation.

common prefix with k-mismatches. Given a string $x[0\,..\,n-1]$, the LCP with k-mismatches for every suffix $x[i\,..\,n-1]$ is the length of the longest common prefix of $x[i\,..\,n-1]$ and any $x[j\,..\,n-1]$, where $j \neq i$, with up to k mismatches. Note that for the mismatches we make use of the Hamming distance model throughout.

Molecular Biology. Repeated sequences are a common characteristic of genomes. One type in particular, namely *interspersed repeats*, are known to occur in all eukaryotic genomes. These repeats have no repetitive pattern and appear irregularly within DNA sequences [20]. Occurrences of single nucleotide polymorphism exist within the human genome, where a mutation of a single nucleotide takes place during cell division. This results in the existence of interspersed repeats that are not identical [21]. Identifying the positions of where these repeats occur has been linked to genome folding locations and phylogenetic analysis [27].

Computational Musicology. Sequential patterns of varying length are a key feature of musical compositions. Chords are made up of three or more simultaneously played notes and a chord progression is made up of two or more chords, where the order plays a significant role in determining the tone of a musical piece. Minor changes can exist in the transitions between chords. The analysis of these harmonic progressions contributes to the analysis and categorisation of musical genres [6]. Another example is *ostinato*, a motif that persistently repeats in the same musical voice; it is used to define the tone of a piece of music [8]. It commonly refers to exact repetition, but also covers repetition with variations.

Natural Language Processing. Natural language text collections are increasing rapidly and massively, thus becoming a source of several analysis tasks such as text classification. For instance, an important task is to identify similar or duplicate documents in large text collections for detecting plagiarism or for obtaining more realistic text statistics. To this end, many repetition-based strategies have been suggested. One of these approaches is based on computing the common repeated lengths of a document's text in other documents of the collection to derive whether the document is repeated in the text collection or not [19].

State of the Art. The problem of computing the longest common prefixes with k-mismatches was introduced by Manzini in [23]; an algorithm was presented to solve the problem only for $k = 1$ in time $\mathcal{O}(nL_{\text{ave}} \log n / \log\log n)$ using $\mathcal{O}(n)$ extra space for a string of length n over a constant-sized alphabet, where L_{ave} is the average value in the LCP array. We show that this value is $\Omega(\log n)$ (see Lemma 3). This value is known to be $\mathcal{O}(\log n)$ [18] on average, and so the algorithm of [23] works in time $\mathcal{O}(n \log^2 n / \log\log n)$ on average for $k = 1$.

Our Contribution. Due to our motivational applications we focus on linear-space solutions. Given a string x of length n over a constant-sized alphabet of size σ and an integer $0 < k < n$, and setting $R = \lceil (k+2)(\log_\sigma n + 1) \rceil$, we make the following threefold contribution:

1. We improve upon the result of [23] by presenting an algorithm for computing the longest common prefixes with 1-mismatch requiring time $\mathcal{O}(n \log n \log^2 \log n)$ on average using $\mathcal{O}(n)$ extra space. In fact we show how our technique can be generalised to work for arbitrary k; the average-case time complexity then becomes $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$ using $\mathcal{O}(n)$ extra space.
2. We apply our technique to compute the related longest previous factor (LPF) with k-mismatches for every suffix of x within the same complexities. The LPF with k-mismatches of the suffix $x[i\,..\,n-1]$ is the length of the longest prefix of $x[i\,..\,n-1]$ that occurs before i in x with at most k mismatches.
3. We also apply our technique to construct a data structure of size $\mathcal{O}(n)$ in average-case time $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$ using $\mathcal{O}(n)$ extra space that answers queries of the following type in $\mathcal{O}(1)$ time per query: return the smallest m such that at least α of the substrings of x of length m do not occur more than once in x with at most k mismatches. This data structure is a more general solution to the genome mappability problem [12].

2 Preliminaries

2.1 Strings

We begin with some basic definitions and notation. Let $x = x[0]x[1]\dots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma = \mathcal{O}(1)$. For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\dots x[j]$ the *substring* (sometimes called *factor*) of x that starts at position i and ends at position j. By ε we denote the *empty string* of length 0. We recall that a *prefix* of x is a substring that starts at position 0 ($x[0\,..\,j]$) and a *suffix* of x is a substring that ends at position $n-1$ ($x[i\,..\,n-1]$).

Let y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of y in x, or, more simply, that y *occurs in* x, when y is a substring of x. Every occurrence of y can be characterised by a starting position in x. We thus say that y occurs at the *starting position* i in x when $y = x[i\,..\,i+m-1]$.

The *Hamming distance* between two strings x and y, with $|x| = |y|$, is defined as $d_H(x,y) = |\{i : x[i] \neq y[i],\ i = 0, 1, \dots, |x|-1\}|$. If $|x| \neq |y|$, we set $d_H(x,y) = \infty$. If two strings x and y are at Hamming distance at most k, we write $x \approx_k y$, and we say that x and y have *k-mismatches* or have *at most k mismatches*.

Let x be a string of length $n > 0$. The *suffix tree* $\mathcal{T}(x)$ of x is a compact trie representing all suffixes of x. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. Node v is a *terminal* node if its path-label is a suffix of x, that is, $\mathcal{L}(v) = x[i\,..\,n-1]$ for

some $0 \leq i < n$; here v is also labelled with index i. It should be clear that each substring of x is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(x)$, called its *locus*. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once $\mathcal{T}(x)$ is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v. The suffix tree of a string of length n can be computed in time and space $\mathcal{O}(n)$ [30].

We denote by SA the *suffix array* of x. SA is an integer array of size n storing the starting positions of all (lexicographically) sorted non-empty suffixes of x, i.e. for all $1 \leq r < n$ we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$ [22]. Let $\mathsf{lcp}(r,s)$ denote the length of the longest common prefix between $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$ for positions r, s on x. We denote by LCP the *longest common prefix* array of x defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1,r)$ for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. It is known that SA, iSA, and LCP of a string of length n, over a constant-sized alphabet, can be computed in time and space $\mathcal{O}(n)$ [26, 13].

The *permuted LCP array*, denoted by PLCP, has the same contents as the LCP array but in different order. Let i^- denote the starting position of the lexicographic predecessor of $x[i\,..\,n-1]$. For $i = 0, \ldots, n-1$, we define $\mathsf{PLCP}[i] = \mathsf{LCP}[\mathsf{iSA}[i]] = \mathsf{lcp}(\mathsf{iSA}[i^-], \mathsf{iSA}[i])$, that is, $\mathsf{PLCP}[i]$ is the length of the longest common prefix between $x[i\,..\,n-1]$ and its lexicographic predecessor. For the starting position j of the lexicographically smallest suffix we set $\mathsf{PLCP}[j] = 0$. For any $k \geq 0$, we define $\mathsf{lcp}_k(y,z)$ as the largest $\ell \geq 0$ such that $y[0\,..\,\ell-1]$ and $z[0\,..\,\ell-1]$ exist and are at Hamming distance at most k, that is, have at most k mismatches; note that lcp_k is defined for a pair of strings. We analogously define the *permuted LCP array with k-mismatches*, denoted by PLCP_k. For $i = 0, \ldots, n-1$, we have that

$$\mathsf{PLCP}_k[i] = \max_{j=0,\ldots,n-1,\ j \neq i} \mathsf{lcp}_k(x[i\,..\,n-1], x[j\,..\,n-1]).$$

The computational problem in scope can be formally stated as follows.

PLCP WITH k-MISMATCHES
Input: A string x of length n and an integer $0 < k < n$
Output: PLCP_k and P_k; $\mathsf{P}_k[i] \neq i$, for $i = 0, \ldots, n-1$, is such that $x[i\,..\,i+\ell-1] \approx_k x[\mathsf{P}_k[i]\,..\,\mathsf{P}_k[i]+\ell-1]$, where $\ell = \mathsf{PLCP}_k[i]$

Example 1. Consider the string `acababbac` and $k = 1$. The following table gives arrays PLCP_1 and P_1.

i	0	1	2	3	4	5	6	7	8
$\mathsf{PLCP}_1[i]$	4	3	4	3	3	3	3	2	1
$\mathsf{P}_1[i]$	2	3	0	1	2	2	3	0	1

Our Analysis Model. When we state average-case time complexities for our algorithms, we assume that the input is a string x of length n over a constant-sized alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ.

2.2 Advanced Data Structure Tools

Let $\mathcal{T}$ be a rooted tree of size n with integer weights on nodes each of magnitude at most n. We require that the root weight is zero and the weight of any other node is strictly larger than its parent's weight. A node v is a *weighted ancestor* of a node u at depth ℓ if v is the highest ancestor of u with weight at least ℓ.

Lemma 1 ([4]) *After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries of nodes of a tree $\mathcal{T}$ can be answered in $\mathcal{O}(\log\log n)$ time per query.*

The following corollary applies Lemma 1 to the suffix tree of a string x of length n.

Corollary 1. *After $\mathcal{O}(n)$-time preprocessing, the locus of any substring $x[i\,..\,j]$ in $\mathcal{T}(x)$ can be found in $\mathcal{O}(\log\log n)$ time.*

Definition 1. *Given a string x and a substring y of x, we denote by range(x, y) the range in the SA of x that represents the suffixes of x that have y as a prefix.*

Every node u in $\mathcal{T}(x)$ corresponds to an SA range *range*$(x, \mathcal{L}(u))$. We can precompute *range*$(x, \mathcal{L}(u))$ for all explicit nodes u in $\mathcal{T}(x)$ in $\mathcal{O}(n)$ time while performing a depth-first traversal of the tree. We also make use of the following lemma for a string x of length n.

Lemma 2 ([14]) *Let y and z be two substrings of x. Given the SA and the iSA of x, as well as range(x, y) and range(x, z), range(x, yz) can be computed in time $\mathcal{O}(\log\log n)$ after $\mathcal{O}(n\log\log n)$-time and $\mathcal{O}(n)$-space preprocessing.*

3 Longest Common Prefixes with k-Mismatches

We first show the following lower bound which is related to the time complexity of the algorithm in [23].

Lemma 3 *The average value in the LCP array of any string x of length n over an alphabet Σ of size σ is $\Omega(\log_\sigma n)$.*

Proof. First note that $\sum_i \mathsf{LCP}[i] \leq \sum_i \max\{\mathsf{LCP}[i], \mathsf{LCP}[i+1]\} \leq 2\sum_i \mathsf{LCP}[i]$. We thus consider $\max\{\mathsf{LCP}[i], \mathsf{LCP}[i+1]\}$ (i.e. the length of the longest common prefix of $x[i\,..\,n-1]$ with any other suffix of x) instead of $\mathsf{LCP}[i]$ to simplify the proof. We know that $\max\{\mathsf{LCP}[i], \mathsf{LCP}[i+1]\}$ is equal to the string-depth of the parent of the leaf with path-label $x[i\,..\,n-1]\$$, $\$ \notin \Sigma$, in the suffix tree of $x\$$.

Consider the suffix tree of $x\$$. Each node can have at most $\sigma+1$ leaves attached to it. We have at most σ^r non-leaf nodes at depth r. Hence we can obtain a brute force lower bound by assuming that we have a complete tree—of the required depth so that it has $n+1$ leaves in total—with $\sigma+1$ leaves in all of its nodes (note that this is impossible). This required depth is the smallest t such that $(\sigma+1)(1+\sigma+\ldots+\sigma^t) \geq n$; $t = \Omega(\log_\sigma n)$. It is then clear that nodes in the two deepest levels have attached to them at least half of the $n+1$ leaves; this concludes the proof. □

We next present an algorithm for the problem PLCP WITH 1-MISMATCH and then explain how it can be extended to solve problem PLCP WITH k-MISMATCHES. The proposed algorithm essentially consists of two different parts:

1. Computing *long* PLCPs in average-case time $\mathcal{O}(n)$;
2. Computing *short* PLCPs in worst-case time $\mathcal{O}(n \log n \log^2 \log n)$.

Notably, both parts use $\mathcal{O}(n)$ extra space for arbitrary k.

We initialize PLCP_1 and P_1 for each i based on the longest common prefix of $x[i\,..\,n-1]$ (i.e. not allowing any mismatches) that occurs elsewhere using the SA and the LCP array; this can be done in $\mathcal{O}(n)$ time.

Computing Long PLCPs. The first part is a slight modification of the algorithm presented in Section 3 of [3] for the problem of 1-mappability. In this problem, we are asked to compute for each substring of length m of a given string of length n the number of other occurrences of this substring in the string with at most 1 mismatch. The algorithm of [3] was shown to solve this problem in average-case time $\mathcal{O}(n)$ for values of m greater than or equal to $3\log_\sigma n + 3$ using space $\mathcal{O}(n)$.

The algorithm presented in [3] computes all pairs of suffixes that share a prefix of length at least m with at most 1 mismatch. When such a pair is considered, the algorithm has already precomputed enough information (using longest common extension queries) that allows us to retrieve the longest common prefix with 1-mismatch of these two suffixes in $\mathcal{O}(1)$ time. This is merely because a longest common extension query may extend beyond length m: it is interrupted only by the second mismatch (or the ends of the string).

Hence, if we set $m = R = \lceil 3\log_\sigma n \rceil + 3$, it is trivial to store, within the same complexities, $\mathsf{PLCP}_1[i]$ and $\mathsf{P}_1[i]$ for every i for which $x[i\,..\,i+R-1]$ has 1-mappability greater than 0 (i.e. $x[i\,..\,i+R-1]$ occurs elsewhere in x with at most 1 mismatch). Note that these are the positions i for which we have that $\mathsf{PLCP}_1[i] \geq R$. We thus arrive at the following lemma.

Lemma 4 *We can compute $\mathsf{PLCP}_1[i]$ and $\mathsf{P}_1[i]$ for each i for which $\mathsf{PLCP}_1[i] \geq R$ in average-case time $\mathcal{O}(n)$ using $\mathcal{O}(n)$ extra space.*

Computing Short PLCPs. Let S be the set of starting positions of m-length substrings that have 1-mappability 0 for $m = R$. For each $i \in S$, we have that $\mathsf{PLCP}_1[i] < R = \mathcal{O}(\log n)$. We proceed to compute $\mathsf{PLCP}_1[i]$ and $\mathsf{P}_1[i]$ for each $i \in S$ as follows; see also Figure 1 for an illustration.

We first locate the node v in $\mathcal{T}(x)$ with path-label $x[i\,..\,n-1]$—this is a terminal node. We then consider each explicit ancestor u of v in $\mathcal{T}(x)$; note that for each such u we have that $\mathcal{D}(u) < R-1$ since $\mathsf{PLCP}_1[i] < R$. For each such u we perform the following. Suppose that the first edge label of the outgoing edge from u that lies on the path from the root to v is $a \in \Sigma$. For each other outgoing edge from u, say with first edge label $b \in \Sigma$, $b \neq a$, we wish to find the longest prefix of $\mathcal{L}(u)bx[i+\mathcal{D}(u)+1\,..\,i+s]$, where $s = \min\{R-2, n-1-i\}$, that is a substring of x and the starting position of one of its occurrences. The longest of

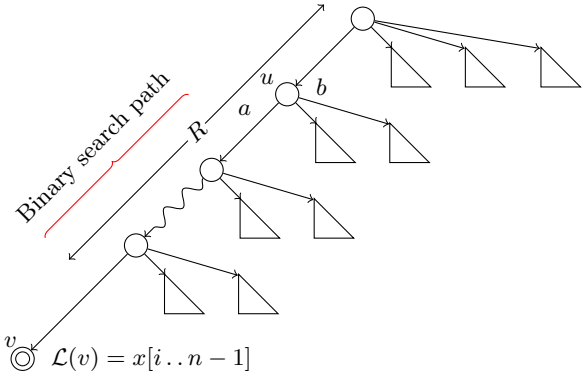

Fig. 1: Illustration; binary search along the shown path for the longest common prefix with 1 mismatch of $\mathcal{L}(v)$ and $\mathcal{L}(u)bx[i+\mathcal{D}(u)+1\,..\,i+\lfloor(R+\mathcal{D}(u)-1)/2\rfloor]$.

these strings is precisely the longest prefix of $x[i\,..\,n-1]$ that occurs elsewhere in x with at most 1 mismatch.

We will find this by performing binary search on the subpath of the path from the root to v that corresponds to $x[i+\mathcal{D}(u)+1\,..\,i+s]$ as follows. We first compute $\mathit{range}(x,y)$ for $y = \mathcal{L}(u)bx[i+\mathcal{D}(u)+1\,..\,i+\lfloor(s+\mathcal{D}(u)+1)/2\rfloor]$. If $\mathit{range}(x,y) = [p,q] \neq \emptyset$ we set $\mathsf{PLCP}_1[i] = \max\{\mathsf{PLCP}_1[i], |y|\}$ (and $\mathsf{P}_1[i] = \mathsf{SA}[p]$ if PLCP_1 has changed) and go down the path; else, the range is empty and we thus go up the path. We proceed with binary search in the same manner.

We can move along the path and find auxiliary ranges (e.g. $\mathit{range}(x, x[i+\mathcal{D}(u)+1\,..\,i+\lfloor(s+\mathcal{D}(u)+1)/2\rfloor])$) stored in explicit nodes of $\mathcal{T}(x)$ using weighted ancestor queries in time $\mathcal{O}(\log\log n)$ per query due to Corollary 1. Note that the range of an implicit node z along an edge (f,g) is equal to that of the explicit node g. We can then merge these ranges in time $\mathcal{O}(\log\log n)$ due to Lemma 2. Hence, each step of the binary search requires time $\mathcal{O}(\log\log n)$. The path-label of the considered path has length $\mathcal{O}(\log n)$ because $R = \lceil 3\log_\sigma n\rceil + 3$, and hence we do $\mathcal{O}(\log\log n)$ iterations for the binary search. Thus, each binary search takes time $\mathcal{O}(\log^2\log n)$ in total. We formalise the described algorithm in the pseudocode presented below.

Lemma 5 *Given the set of positions $S = \{i|\mathsf{PLCP}_1[i] < R\}$, we can compute $\mathsf{PLCP}_1[i]$ and $\mathsf{P}_1[i]$, for all $i \in S$, in worst-case time $\mathcal{O}(n\log n\log^2\log n)$ using $\mathcal{O}(n)$ extra space.*

Proof. The outer loop of PLCP_1Short iterates through positions of x that have 1-mappability 0 for $m = R$; these are at most n. For each of these, the algorithm considers its explicit ancestors; there are $\mathcal{O}(\log n)$ of them. For each such ancestor u it performs $\deg(u) = \mathcal{O}(\sigma) = \mathcal{O}(1)$ binary searches, each of which takes time $\mathcal{O}(\log^2\log n)$ as described above. Thus, algorithm PLCP_1Short takes worst-case time $\mathcal{O}(n\log n\log^2\log n)$. The extra space used is clearly $\mathcal{O}(n)$. □

```
PLCP_1SHORT(x, n, S, R)
 1  T(x) ← SUFFIXTREE(x)
 2  for each explicit node u ∈ T(x) do
 3      D(u) ← string-depth of u
 4      I(u) ← range(x, L(u))
 5  Preprocess T(x) for weighted ancestor queries
 6  for i ∈ S do
 7      v ← node with path-label x[i..n−1]
 8      for each explicit node u ancestor of v in T(x) do
 9          a ← x[i + D(u)]
10          for each outgoing edge from u with first edge label b ≠ a do
11              PATHBINARYSEARCH(i, u, b, D(u) + 1, min{R − 2, n − 1 − i})
```

```
PATHBINARYSEARCH(i, u, b, j_1, j_2)
 1  w ← NODE(i + D(u) + 1, i + ⌊(j_1 + j_2)/2⌋)
 2  [p, q] ← range(x, L(u)bL(w))
 3  if [p, q] ≠ ∅ then
 4      ℓ ← |L(u)bL(w)|
 5      if PLCP_1[i] < ℓ then
 6          PLCP_1[i] ← ℓ
 7          P_1[i] ← SA[p]
 8      if j_1 ≠ j_2 then
 9          PATHBINARYSEARCH(i, u, b, ⌊(j_1 + j_2)/2⌋, j_2)
10  else if j_1 ≠ j_2 then
11          PATHBINARYSEARCH(i, u, b, j_1, ⌊(j_1 + j_2)/2⌋)
```

By combining Lemmas 4 and 5 we arrive at the following result.

Theorem 6. *Problem* PLCP WITH 1-MISMATCH *can be solved in average-case time* $\mathcal{O}(n \log n \log^2 \log n)$ *using* $\mathcal{O}(n)$ *extra space.*

Theorem 6 improves upon the state-of-the-art average-case time complexity for the case where only 1 mismatch is allowed by a factor of $\log n / \log^3 \log n$ (see Lemma 3 and [23]). Notably, our technique can be extended to work for arbitrary k as follows. We first set $R = \lceil (k+2)(\log_\sigma n + 1) \rceil$ so that the adapted algorithm from [3] requires time $\mathcal{O}(kn)$ on average. Then for each position i with $\mathsf{PLCP}_k[i] < R$ we have $\mathcal{O}(k \log n)$ branching nodes for the first mismatch—similar to the above. Suppose that for some explicit node u, ancestor of v, where $\mathcal{L}(v) = x[i \,..\, n-1]$ and u has an outgoing edge with first edge label $b \neq x[i + \mathcal{D}(u)]$, the longest prefix of $\mathcal{L}(u)bx[i + \mathcal{D}(u) + 1 \,..\, i + s]$, where $s = \min\{R - 2, n - 1 - i\}$, that occurs in x is $x[p \,..\, q]$. In order to allow for a second mismatch, we consider every ancestor of node u_b, where $\mathcal{L}(u_b) = x[p \,..\, q]$, with string-depth larger than $\mathcal{D}(u) + 1$, and proceed in a similar fashion.

Each branching step allows for an extra mismatch. We only consider $\mathcal{T}(x)$ up to string-depth R and hence the possible branching options are $\mathcal{O}((\sigma R)^k)$. Each binary search is now performed on a path with path-label of length $\mathcal{O}(k \log n)$. Each iteration of the binary search takes time $\mathcal{O}(\log \log n)$ for the level ancestor query and for merging the ranges. We thus arrive at the following result.

Theorem 7. *Problem* PLCP WITH k-MISMATCHES *can be solved in average-case time* $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$*, where* $R = \lceil (k+2)(\log_\sigma n + 1)\rceil$*, using* $\mathcal{O}(n)$ *extra space.*

Remark 1. Alternatively, in the second part of our algorithm (Computing Short PLCPs), we can perform the binary search with the aid of the data structure presented by Cole et al in [9]. This data structure is of size $\mathcal{O}(n\frac{(c_1 \log n)^k}{k!})$ and can be built in time $\mathcal{O}(n\frac{(c_1 \log n)^k}{k!})$, where $c_1 > 1$ is a constant. We can then answer whether a given substring of x occurs elsewhere in x with at most k mismatches (as well as the starting position of one of its occurrences) in time $\mathcal{O}(\frac{(c_2 \log n)^k \log\log n}{k!})$, where $c_2 > 1$ is another constant. With this modification, the average-case time required by our algorithm becomes $\mathcal{O}(n\frac{(\log n)^k}{k!}(c_1^k + c_2^k(\log\log n(\log k + \log\log n))))$. The $\omega(n)$ space required for this data structure is however impractical for real-world datasets. Note that the efficient solutions of Thankachan et al for the related *longest common factor with k-mismatches* problem also require space $\omega(n)$ when $k = \omega(1)$ [28, 29].

4 Longest Previous Factors with k-Mismatches

The longest previous factor (LPF) array gives, for each position i in a string x, the length of the longest factor of x that occurs both at i and to the left of i in x. The LPF array is central in many text compression techniques as well as in the most efficient algorithms for detecting motifs and repetitions occurring in a text [11]. A time-space optimal (linear) algorithm that computes the LPF array is known for some time [10].

In this section, we present how the algorithm presented in Section 3 can be adapted to compute the LPF array with k-mismatches within the same complexities. We naturally define the *LPF array with k-mismatches*, denoted by LPF_k, as follows. We set $\mathsf{LPF}_k[0] = -1$ and for $i = 1, \ldots, n-1$, we have that

$$\mathsf{LPF}_k[i] = \max_{j=0,\ldots,i-1} \mathsf{lcp}_k(x[i\,..\,n-1], x[j\,..\,n-1]).$$

The problem in scope can be formally defined as follows.

LPF WITH k-MISMATCHES
Input: A string x of length n and an integer $0 < k < n$
Output: LPF_k and P_k; $\mathsf{P}_k[0] = -1$ and $\mathsf{P}_k[i] < i$, for $i = 1, \ldots, n-1$, is such that $x[i\,..\,i+\ell-1] \approx_k x[\mathsf{P}_k[i]\,..\,\mathsf{P}_k[i]+\ell-1]$, where $\ell = \mathsf{LPF}_k[i]$

Example 2. Consider the string `acababbac`. The following table gives arrays LPF_1 and P_1.

i	0	1	2	3	4	5	6	7	8
$\mathsf{LPF}_1[i]$	-1	1	4	3	3	3	3	2	1
$\mathsf{P}_1[i]$	-1	0	0	1	2	2	3	0	1

First, let us recall that a range minimum query (RMQ) data structure over an array of size n can be constructed in time and space $\mathcal{O}(n)$, and can then answer range minimum queries in $\mathcal{O}(1)$ time per query (see [7] for the details). The following two modifications to the algorithm presented in Section 3 suffice to transform it to an algorithm that solves problem LPF WITH k-MISMATCHES:

1. When running the average-case k-mappability algorithm and considering a pair of suffixes starting at positions r, q, $r < q$, $\mathsf{LPF}_k[r]$ remains unchanged, while we only update $\mathsf{LPF}_k[q]$ (along with $\mathsf{P}_k[q]$) if $\mathsf{LPF}_k[q] < \mathsf{lcp}_k(x[r\,..\,n-1], x[q\,..\,n-1])$.
2. In the second part of the algorithm described above, while processing the suffix starting at position i, if at any step of the algorithm the obtained SA range $[p, q]$ (corresponding to node z in $\mathcal{T}(x)$) is not empty, we also have to check whether it contains a position smaller than i. We do this by employing the RMQ data structure over the suffix array. If $\mathsf{SA}[\mathrm{RMQ}_{\mathsf{SA}}(p, q)] > i$, we treat the range as if it were empty and go up the path. If $\mathsf{SA}[\mathrm{RMQ}_{\mathsf{SA}}(p, q)] < i$, we go down the path after checking whether $\mathsf{LPF}_k[i] < \mathcal{D}(z)$; if yes, we set $\mathsf{LPF}_k[i] = \mathcal{D}(z)$ and $\mathsf{P}_k[i] = \mathsf{SA}[\mathrm{RMQ}_{\mathsf{SA}}(p, q)]$.

Theorem 8. *Problem* LPF WITH k-MISMATCHES *can be solved in average-case time* $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$, *where* $R = \lceil (k+2)(\log_\sigma n + 1)\rceil$, *using* $\mathcal{O}(n)$ *extra space.*

5 Application of LCP with k-Mismatches to Genome Mappability

The focus of this section is directly motivated by the well-known and challenging application of *genome re-sequencing*—the assembly of a genome directed by a reference sequence. New developments in sequencing technologies [24] allow whole-genome sequencing to be turned into a routine procedure, creating sequencing data in massive amounts. Short sequences, known as *reads*, are produced in huge amounts (tens of gigabytes); and in order to determine the part of the genome from which a read was derived, it must be mapped (aligned) back to some reference sequence that consists of a few gigabases. A wide variety of short-read alignment techniques and tools have been published in the past years to address the challenge of efficiently mapping tens of millions of reads to a genome, focusing on different aspects of the procedure: speed, sensitivity, and accuracy [15]. These tools allow for a small number of errors in the alignment.

The re-sequencing method starts with matching a *seed* of each read to the genome. This, for example, could be a short prefix of the read (the accuracy is usually higher in the prefix of the read). We then *extend* this match until the total number of errors exceeds a predefined threshold or until a match is found [2]. Considering errors is necessary due to genetic variation as well as sequencing errors; most of these errors are single-base substitutions [25]. It is suitable to allow for a small number k of errors in the seed part.

The *k-mappability* problem was first introduced in the context of genome analysis in [12] (and in some sense earlier in [5]), where a heuristic algorithm was proposed to approximate the solution. The aim from a biological perspective is to compute the mappability of each region of a genome sequence; i.e. for every substring of a given length of the sequence, we are asked to count how many other times it occurs in the genome with up to a given number of errors. This is particularly useful in the application of genome re-sequencing. By computing the mappability of the reference genome, we can then assemble the genome of an individual with greater confidence by first mapping the segments of the DNA that correspond to regions with low mappability. Interestingly, it has been shown that genome mappability varies greatly between species and gene classes [12].

Formally, we are given a string x of length n and integers $m < n$ and $k < m$, and we are asked to count, for each length-m substring y of x, the number *occ* of other length-m substrings of x that are at Hamming distance at most k from y. We then say that this substring has k-mappability equal to *occ*. Hence, a more general question to ask is the following: *What is the minimal value of m that forces at least α of the starting positions in the reference genome to have k-mappability equal to 0?* We formalise this question as a data structure problem.

Genome Mappability
Input: A string x of length n and an integer $0 < k < n$
Query: $\mathsf{LEN}_{x,k}(\alpha)$ that represents the smallest m such that at least $\alpha > 0$ of the substrings of x of length m do not occur more than once in x with at most k mismatches

We can solve this problem by first making the following crucial observation.

Observation 9 *A substring $x[i\,..\,i+m-1]$ of a string x does not occur more than once in x with at most k mismatches if and only if $m > \mathsf{PLCP}_k[i]$.*

Our construction works as follows. We first compute the PLCP_k array for x. We then sort its elements in ascending order using bucket sort in time $\mathcal{O}(n)$ and store them in a new array $\mathcal{A}_k$. Based on Observation 9, $\mathsf{LEN}_{x,k}(\alpha)$ is given by the smallest m for which $\mathcal{A}_k[\alpha+m-2]+1 \leq m$. We show the following property on the values of $\mathcal{A}_k$.

Property 1. $\mathcal{A}_k[i+1] \leq \mathcal{A}_k[i]+1$.

Proof. Note that either $\mathsf{PLCP}_k[i+1] \geq \mathsf{PLCP}_k[i]$ or $\mathsf{PLCP}_k[i+1] = \mathsf{PLCP}_k[i]-1$. Thus considering the values in the PLCP_k array from the left to the right they start from $\mathsf{PLCP}_k[0]$ and then as we move one position to the right, this value either increases or stays the same or drops by 1. $\mathsf{PLCP}_k[n-1]$ is equal to either 0 or 1. Hence, for every integer $d \in [\min_i\{\mathsf{PLCP}_k[i]\}, \max_i\{\mathsf{PLCP}_k[i]\}]$ there exists a j, $0 \leq j \leq n-1$, such that $\mathsf{PLCP}_k[j] = d$ and thus the lemma follows. (Note that $\mathcal{A}_k$ is just the sorted PLCP_k array.) □

For each α, $0 < \alpha \leq n-1$, we denote by $m_\alpha \in [k+1, n-\alpha+1]$ the smallest integer—if it exists—for which $\mathcal{A}_k[\alpha+m_\alpha-2]+1 \leq m_\alpha$ holds. In fact, we know

by Property 1 that if such an m_α exists, then

$$\mathcal{A}_k[\alpha + r - 2] + 1 \leq r, \text{ for all } m_\alpha \leq r \leq n - \alpha + 1.$$

Moreover, we have that $m_\alpha \leq m_{\alpha+1}$, since

$$\mathcal{A}_k[\alpha + m_{\alpha+1} - 2] + 1 \leq \mathcal{A}_k[\alpha + 1 + m_{\alpha+1} - 2] + 1 \leq m_{\alpha+1}.$$

We can thus precompute m_α, for all $0 < \alpha \leq n - 1$, and store them in an array $\mathcal{B}_k[\alpha] = m_\alpha$ in time $\mathcal{O}(n)$ while scanning array $\mathcal{A}_k$ from left to right: we start by computing m_1 and apply the inequality $m_\alpha \leq m_{\alpha+1}$ to obtain $m_2, \ldots, m_{n-1}$. If such an integer m_α does not exist, we set $\mathcal{B}_k[\alpha] = 0$. We can then answer query $\mathsf{LEN}_{x,k}(\alpha)$ in time $\mathcal{O}(1)$: the answer is $\mathcal{B}_k[\alpha]$.

Example 3. Consider the string $x = \texttt{acababbac}$ and $k = 1$. The following table gives arrays PLCP_1, $\mathcal{A}_1$, and $\mathcal{B}_1$. For $\alpha = 3$, we have that $\mathsf{LEN}_{x,k}(3) = \mathcal{B}_1[3] = 4$.

i	0	1	2	3	4	5	6	7	8
$\mathsf{PLCP}_1[i]$	4	3	4	3	3	3	3	2	1
$\mathcal{A}_1[i]$	1	2	3	3	3	3	3	4	4
$\mathcal{B}_1[i]$	-	4	4	4	4	5	0	0	0

We thus obtain the following result.

Theorem 10. *Array $\mathcal{B}_k$ can be computed in time $\mathcal{O}(n)$ from array PLCP_k.*

Corollary 2. *We can construct an $\mathcal{O}(n)$-sized data structure in average-case time $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$, where $R = \lceil (k+2)(\log_\sigma n + 1) \rceil$, using $\mathcal{O}(n)$ extra space that answers* Genome Mappability *queries in $\mathcal{O}(1)$ time per query.*

6 Final Remarks

We have presented a new algorithm for computing the longest prefix of each suffix of a given string of length n over a constant-sized alphabet of size σ that occurs elsewhere in the string with Hamming distance at most k. The proposed algorithm requires time $\mathcal{O}(n(\sigma R)^k \log\log n(\log k + \log\log n))$ on average, where $R = \lceil (k+2)(\log_\sigma n + 1) \rceil$, and $\mathcal{O}(n)$ extra space.

We have then shown that the proposed technique can be adapted and applied for computing the longest previous factors under the Hamming distance model within the same complexities. Finally, we have shown that our technique can be applied to construct an $\mathcal{O}(n)$-sized data structure that can answer queries related to genome mappability [12] in $\mathcal{O}(1)$ time per query.

We anticipate that this new technique would be applicable in several contexts where we have to compute longest repeating factors under the Hamming distance model subject to some additional requirements. For instance, one such problem is computing the longest factor of a string occurring in another string with k-mismatches, also known as the LCF with k-mismatches problem [16, 28].

References

1. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
2. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990.
3. Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, Jakub Radoszewski, and Wing-Kin Sung. Faster algorithms for 1-mappability of a sequence. In *COCOA*, LNCS. Springer International Publishing, 2017.
4. Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algor.*, 3(2):19, 2007.
5. Pavlos Antoniou, Jacqueline W. Daykin, Costas S. Iliopoulos, Derrick Kourie, Laurent Mouchard, and Solon P. Pissis. Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome. In *ITAB*, pages 1–4. IEEE Computer Society, 2009.
6. Mathieu Barthet, Mark D. Plumbley, Alexander Kachkaev, Jason Dykes, Daniel Wolff, and Tillman Weyde. Big chord data extraction and mining. In *CIM*, 2014.
7. Michael A. Bender and Martín Farach-Colton. The LCA problem revisited. In *LATIN*, volume 1776 of *LNCS*, pages 88–94. Springer-Verlag, 2000.
8. C. Bufe. *Understandable Guide to Music Theory: The Most Useful Aspects of Theory for Rock, Jazz, and Blues Musicians*. See Sharp Press, 1994.
9. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, STOC '04, pages 91–100. ACM, 2004.
10. Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26, 2013.
11. Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *DCC*, pages 482–488. IEEE Computer Society, 2008.
12. Thomas Derrien, Jordi Estellé, Santiago Marco Sola, David Knowles, Emanuele Raineri, Roderic Guigó, and Paolo Ribeca. Fast computation and applications of genome mappability. *PLoS ONE*, 7(1), 2012.
13. Johannes Fischer. Inducing the LCP-array. In *WADS*, volume 6844 of *LNCS*, pages 374–385. Springer-Verlag, 2011.
14. Johannes Fischer, Dominik Köppl, and Florian Kurpicz. On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching. In *CPM 2016*, volume 54 of *LIPIcs*, pages 26:1–26:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
15. Nuno A. Fonseca, Johan Rung, Alvis Brazma, and John C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24):3169–3177, 2012.
16. Szymon Grabowski. A note on the longest common substring with k-mismatches problem. *Inf. Process. Lett.*, 115(6-8):640–642, 2015.
17. Juha Kärkkäinen and Dominik Kempa. Faster external memory LCP array construction. In *ESA*, volume 57 of *LIPIcs*, pages 61:1–61:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
18. Samuel Karlin, Ghassan Ghandour, Friedemann Ost amd Simon Tavare, and Laurence J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 80(18):5660–5664, 1983.

19. Dmitry V. Khmelev and William J. Teahan. A repetition based measure for verification of text collections and for text categorization. In *ACM SIGIR*, SIGIR '03, pages 104–110. ACM, 2003.
20. Roman Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*, 31(13):3672–3678, 2003.
21. Kung-Hao Liang. *Bioinformatics for Biomedical Science and Clinical Applications*. Woodhead Publishing Series in Biomedicine. Woodhead Publishing, 2013.
22. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
23. Giovanni Manzini. Longest common prefix with mismatches. In *SPIRE*, volume 9309 of *LNCS*, pages 299–310. Springer, 2015.
24. Michael L. Metzker. Sequencing technologies – the next generation. *Nat. Rev. Genet.*, 11(1):31–46, 2010.
25. Claudine Médigue, Matthias Rose, Alain Viari, and Antoine Danchin. Detecting and analyzing dna sequencing errors: Toward a higher quality of the bacillus subtilis genome sequence. *Genome Research*, 9(11):1116–1127, 1999.
26. Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *DCC*, IEEE, pages 193–202, 2009.
27. Arian FA Smit. Interspersed repeats and other mementos of transposable elements in mammalian genomes. *Current Opinion in Genetics & Development*, 9(6):657–663, 1999.
28. Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016.
29. Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Alberto Apostolico, and Srinivas Aluru. ALFRED: A practical method for alignment-free distance computation. *Journal of Computational Biology*, 23(6):452–460, 2016.
30. Peter Weiner. Linear pattern matching algorithms. In *SWAT*, SWAT '73, pages 1–11. IEEE Computer Society, 1973.