Longest common substring with approximately k mismatches

Tatiana Starikovskaya

University of Bristol
Senate House, Tyndall Avenue, Bristol, BS8 1TH, United Kingdom
tat.starikovskaya@gmail.com

Abstract

In the longest common substring problem we are given two strings of length n and must find a substring of maximal length that occurs in both strings. It is well-known that the problem can be solved in linear time, but the solution is not robust and can vary greatly when the input strings are changed even by one letter. To circumvent this, Leimester and Morgenstern introduced the problem of the longest common substring with k mismatches. Lately, this problem has received a lot of attention in the literature, and several algorithms have been suggested. The running time of these algorithms is $n^{2-o(1)}$, and unfortunately, conditional lower bounds have been shown which imply that there is little hope to improve this bound.

In this paper we study a different but closely related problem of the longest common substring with approximately k mismatches and use computational geometry techniques to show that it admits a randomised solution with strongly subquadratic running time.

1998 ACM Subject Classification F.2.2. Nonnumerical Algorithms and Problems

Keywords and phrases Randomised algorithms, string similarity measures, longest common substring, sketching, locality-sensitive hashing

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Understanding how similar two strings are and what they share in common is a central task in stringology, the significance of which is witnessed for example by the 50,000+ citations of the paper introducing BLAST [3], a heuristic algorithmic tool for comparing biological sequences. This task can be formalised in many different ways, from the longest common substring problem to the edit distance problem. The longest common substring problem can be solved in optimal linear time and space, while the best known algorithms for the edit distance problem require $n^{2-o(1)}$ time, which makes the longest common substring problem an attractive choice for many practical applications. On the other hand, the longest common substring problem is not robust and its solution can vary greatly when the input strings are changed even by one letter. To overcome this issue, recently there has been introduced a new problem called the longest common substring with k mismatches. In this paper we continue this line of research.

1.1 Related work

Let us start with a precise statement of the longest common substring problem.

▶ **Problem 1** (The longest common substring). Given two strings T_1, T_2 of length n, find a substring of maximal length that occurs in T_1 and T_2 exactly.

Conference title on which this volume is based on.

The suffix tree of T_1 and T_2, a data structure containing all suffixes of T_1 and T_2, allows to solve this problem in linear time and space [31, 18, 20], which is optimal as any algorithm needs $\Omega(n)$ time to read and $\Omega(n)$ space to store the strings. However, if we only account for "additional" space, the space the algorithm uses apart from the space required to store the input, then the suffix tree-based solution is not optimal and has been improved in a series of publications [5, 24, 30].

The major disadvantage of the longest common substring problem is that its solution is not robust. Consider, for example, two pairs of strings: a^n, $a^{n-1}b$ and $a^{(n-1)/2}ba^{(n-1)/2}$, $a^{n-1}b$. (Assume for simplicity that $n-1 \geq 2$ is even.) The longest common substring of the first pair of strings is twice as long as the longest common substring of the second pair of strings, although we changed only one letter. This makes the longest common substring unsuitable to be used as a measure of similarity of two strings: Intuitively, changing one letter must not change the measure of similarity much. To overcome this issue, it is natural to allow the substring to occur in T_1 and T_2 not exactly but with a small number of mismatches.

▶ **Problem 2** (The longest common substring with k mismatches)**.** Given two strings T_1, T_2 of length n and an integer k, find a substring of maximal length that occurs in T_1 and T_2 with at most k mismatches.

The problem can be solved in quadratic time and space by a dynamic programming algorithm, but there have been also shown more efficient solutions. The longest common substring with one mismatch problem was first considered in [6], where an $\mathcal{O}(n^2)$-time and $\mathcal{O}(n)$-space solution was given. This result was further improved by Flouri et al. who showed an $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n)$ space solution to the problem [15]. For a general value of k, the problem was first considered by Leimeister and Morgenstern [27] who presented a greedy heuristic algorithm for the problem. Later Flouri et al. showed that the longest common substring with k mismatches admits a quadratic time and constant (additional) space algorithm [15]. Apart from that, Grabowski presented two output-dependent algorithms with running times $\mathcal{O}(n((k+1)(\ell_0+1))^k)$ and $\mathcal{O}(n^2 \ell_k / k)$, where ℓ_0 is the length of the longest common substring of T_1 and T_2 and ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2 [17]. Finally, Aluru et al. gave an $\mathcal{O}(2^k n)$-space, $\mathcal{O}(n(2 \log n)^{k+1})$-time algorithms. Yet, the worst-case running time of all these algorithms is still quadratic. Very recently, Abboud et al. [1] applied the polynomial method to develop a $k^{1.5} n^2 / 2^{\Omega(\sqrt{\frac{\log n}{k}})}$-time randomised solution to the problem.

The best algorithms for the edit distance problem and its variations (we do not give their precise statements here as it is not essential for the paper) also have $n^{2-o(1)}$ running time [29, 11, 28], and these bounds are tight under the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi and Zane: [7, 10]:

▶ **Hypothesis 1.** (SETH). For every $\delta > 0$ there exists an integer m such that SAT on m-CNF formulas on n variables cannot be solved in $m^{O(1)} 2^{(1-\delta)n}$ time.

1.2 Our contribution

In this paper we introduce a new problem called the longest common substring with approximately k mismatches, inspired by the work of Andoni and Indyk [4].

▶ **Problem 3** (The longest common substring with approximately k mismatches)**.** Given two strings T_1, T_2 of length n, an integer k, and a constant $\varepsilon > 0$. If ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2, return a substring of length at least ℓ_k that occurs in T_1 and T_2 with at most $(1+\varepsilon) \cdot k$ mismatches.

In their work Andoni and Indyk used the technique of locality-sensitive hashing to develop a space-efficient randomised index for a variant of the approximate pattern matching problem. We build up on their work with several new ideas in the construction and the analysis to develop a randomised subquadratic-time solution to Problem 3. Assume binary alphabet and let $0 < \varepsilon < 2$ be an arbitrary constant.

▶ **Theorem 1.** *The longest common substring with approximately k mismatches can be solved in $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space and $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time correctly with constant probability.*

If the alphabet is of constant size $\sigma > 2$, we can use a standard trick and encode T_1 and T_2 by replacing each letter a in them with a binary vector $0^{a-1}10^{\sigma-a}$. The Hamming distance (i.e. the number of mismatches) between any two substrings of T_1 and T_2 in the encoded form will be exactly twice as large as the Hamming distance between the original substrings, which allows to extend our solution naturally to this case as well at a cost of an additional constant factor.

We note that although the problem statement is not standard for stringology, it makes perfect sense from the practical point of view. Indeed, for most applications it is not important whether a returned substring occurs in T_1 and T_2 with for example 10 or $(1+\frac{1}{5}) \cdot 10 = 12$ mismatches. The result is also important from the theoretical point of view as it improves our understanding of the big picture of string comparison.

2 Overview

In this section we give an overview of the main ideas needed to prove Theorem 1. The classic solution to the longest common substring problem is based on two observations. The first observation is that the longest common substring of T_1 and T_2 is in fact the longest common prefix of some suffix of T_1 and some suffix of T_2. The second observation is that the maximal length of the longest common prefix of a fixed suffix S of T_1 and suffixes of T_2 is reached on the two suffixes of T_2 that are closest to S in the lexicographic order. This suggests the following algorithm: First, we build a suffix tree of T_1 and T_2, which contains all suffixes of T_1 and T_2 and orders them lexicographically. Secondly, we compute the longest common prefix of each suffix of T_1 and the two suffixes of T_2 closest to S in the lexicographic order, one from the left and one from the right. The problem of computing the longest common prefix has been extensively studied in the literature and a number of very efficient deterministic and randomised solutions exist [8, 9, 13, 21, 19], for example, one can use a Lowest Common Ancestor (LCA) data structure, which can be constructed in linear time and space and maintains longest common prefix queries in $\mathcal{O}(1)$ time [13, 19].

Our solution to the longest common substring with approximately k mismatches problem is somewhat similar. We will consider $\theta(n^{1+1/(1+\varepsilon)} \log n)$ orderings on the suffixes of T_1 and T_2 and will show that with high probability the length of the longest common substring with approximately k mismatches is the answer to a longest common prefix with approximately k mismatches ($\mathrm{LCP}_{\tilde{k}}$) query for some pair of suffixes that are close to each other in one of the orderings. In an $\mathrm{LCP}_{\tilde{k}}$ query we are given two suffixes S_1, S_2 of T_1 and T_2 and must output any integer in the interval $[\ell_k, \ell_{(1+\varepsilon)\cdot k}]$, where ℓ_k and $\ell_{(1+\varepsilon)\cdot k}$ are the lengths of the longest common prefixes of S_1 and S_2 with k and $(1+\varepsilon) \cdot k$ mismatches respectively. Note that $\mathrm{LCP}_{\tilde{k}}$ queries can be answered deterministically in $\mathcal{O}(k)$ time using the kangaroo method [26, 16], but for the purposes of this paper we give a faster randomised solution.

▶ **Theorem 2.** *After $\mathcal{O}(n \log^3 n)$ time and $\mathcal{O}(n \log^2 n)$ space preprocessing of T_1 and T_2, an* $\mathrm{LCP}_{\tilde{k}}$ *query can be answered in $\mathcal{O}(\log^2 n)$ time. The answer is correct with probability at* *least* $1 - \frac{1}{n^2}$.

The key idea is to compute sketches for all power-of-two length substrings of T_1 and T_2. The sketches will have logarithmic length (i.e., we will be able to compare them very fast) and the Hamming distance between them will be roughly equal to the Hamming distance between the original substrings. Once the sketches are computed, we can use a simple binary search to answer $\mathrm{LCP}_{\tilde{k}}$ queries in polylogarithmic time.

To define the orderings on suffixes of T_1 and T_2 we will use the locality-sensitive hashing technique, which was initially introduced for the needs of computational geometry [22]. In more detail, we will choose $\theta(n^{1+1/(1+\varepsilon)} \log n)$ hash functions, where each function can be considered as a projection of a string of length n onto a random subset of its positions. By choosing the size of the subset appropriately, we will be able to guarantee that the hash function is locality-sensitive: For any two strings at the Hamming distance at most k, the values of the hash functions on them will be equal with high probability, while the values of the hash functions on any pair of strings at the Hamming distance bigger than $(1+\varepsilon)\cdot k$ will be different with high probability. For each hash function we will sort the suffixes of T_1 and T_2 by the lexicographic order on their hash values. As a corollary of the locality-sensitive property, if two suffixes of T_1 and T_2 have a long common prefix with at most k mismatches, with high probability they will be close to each other in the ordering.

3 Proof of Theorem 2

In this section we show Theorem 2. During the preprocessing stage, we compute sketches [25] of all substrings of the strings T_1 and T_2 of lengths $\ell = 1, 2, \ldots, 2^{\lfloor \log n \rfloor}$, which can be defined in the following way. For a fixed ℓ choose $\lambda = 1.5 \ln n/\gamma^2$ binary vectors r_ℓ^i, where γ is a constant to be defined later and let

$$r_\ell^i[j] = \begin{cases} 1 & \text{with probability } \frac{1}{4k} \\ 0 & \text{with probability } 1 - \frac{1}{4k} \end{cases} \quad \text{for all } i = 1, 2, \ldots, \lambda \text{ and } j = 1, 2, \ldots, \ell$$

For a string x of length $\ell \in \{1, 2, \ldots, 2^{\lfloor \log n \rfloor}\}$ we define a sketch $\mathrm{sk}(x)$ to be a vector of length λ, where $\mathrm{sk}(x)[i] = r_\ell^i \cdot x \pmod 2$. In other words, to obtain $\mathrm{sk}(x)$ we sample each position of x with probability $\frac{1}{4k}$ and then sum the letters in the sampled positions modulo 2. All sketches can be computed in $\mathcal{O}(n \log^3 n)$ time by independently running the Fast Fourier Transform (FFT) algorithm for each of the vectors r_ℓ^i, and occupy $\mathcal{O}(n \log^2 n)$ space [14]. Each substring S can be decomposed uniquely as $x_1 x_2 \ldots x_r$, where $r \in \mathcal{O}(\log n)$ and $|x_1| > |x_2| > \ldots > |x_r|$ are powers of two. We define a sketch $\mathrm{sk}(S) = \sum_q \mathrm{sk}(x_q)$. Let $\delta_1 = \frac{1}{2}(1 - (1 - \frac{1}{4k})^k)$, $\delta_2 = \frac{1}{2}(1 - (1 - \frac{1}{4k})^{(1+\varepsilon)\cdot k})$, and $\Delta = \frac{(\delta_1 + \delta_2)}{2} \cdot \lambda$.

▶ **Lemma 3** ([25]). *For any i if the Hamming distance between S_1 and S_2 is at most k, then $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ with probability at most δ_1. If the Hamming distance between S_1 and S_2 is at least $(1+\varepsilon)\cdot k$, then $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ with probability at least δ_2.*

Proof. Let m be the Hamming distance between S_1 and S_2 and let $p_1, p_2, \ldots, p_m$ be the positions of the mismatches between them. If none of the positions $p_1, p_2, \ldots, p_m$ are sampled, then $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$, and otherwise for each way of sampling $p_1, p_2, \ldots, p_{m-1}$ exactly one of the two choices for p_m will give $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$. (Recall that the alphabet is binary.) Hence, the probability that $\mathrm{sk}(S_1)[i] \neq \mathrm{sk}(S_2)[i]$ is equal to $\frac{1}{2}(1 - (1 - \frac{1}{4k})^m)$, which is at most δ_1 if the Hamming distance between S_1 and S_2 is at most k, and at least δ_2 if the Hamming distance is greater than $(1+\varepsilon)\cdot k$. ◀

▶ **Lemma 4.** *If the Hamming distance between sketches* $\mathrm{sk}(S_1)$ *and* $\mathrm{sk}(S_2)$ *is bigger than* Δ*, then the Hamming distance between* S_1 *and* S_2 *is bigger than* k*. If the Hamming distance between sketches* $\mathrm{sk}(S_1)$ *and* $\mathrm{sk}(S_2)$ *is at most* Δ*, then the Hamming distance between* S_1 *and* S_2 *is at most* $(1+\varepsilon)\cdot k$*. Both claims are correct with probability at least* $1-\frac{1}{n^3}$*.*

Proof. Let χ_i be an indicator random variable that is equal to one if and only if $\mathrm{sk}(S_1)[i] = \mathrm{sk}(S_2)[i]$. The claim follows immediately from Lemma 3 and the following Chernoff bounds (see [2, Appendix A]). For λ independently and identically distributed binary variables $\chi_1, \chi_2, \dots, \chi_\lambda$, $\Pr[\sum_i \chi_i \geq (p+\gamma)] \leq e^{-2\lambda\gamma^2}$ and $\Pr[\sum_i \chi_i < (p-\gamma)] \leq e^{-2\lambda\gamma^2}$, where $p = \Pr[\chi_i = 1]$. We put $\gamma = \frac{(\delta_2-\delta_1)}{2}$ and obtain that the error probability is at most $e^{-2\lambda\gamma^2} < \frac{1}{n^3}$. (Note that $\gamma = \Theta(1-e^{\varepsilon/4})$ is a constant depending on ε.) ◀

Suppose we wish to answer an $\mathrm{LCP}_{\tilde{k}}$ query on two suffixes S_1, S_2. It suffices to find the longest prefixes of S_1, S_2 such that the Hamming distance between their sketches is at most Δ. As mentioned above, these prefixes can be represented uniquely as a concatenation of strings of power-of-two lengths $\ell_1 > \ell_2 > \ldots > \ell_r$. To compute ℓ_1, we initialise it with the biggest power of two not exceeding n and compute the Hamming distance between the sketches of corresponding substrings. If it is smaller than Δ, we have found ℓ_1, otherwise we divide ℓ_1 by two and continue. Suppose that we already know $\ell_1, \ell_2, \dots, \ell_i$ and let h_i be the Hamming distance between the sketches of prefixes of S_1 and S_2 of lengths $\ell_1+\ell_2+\ldots+\ell_i$. To compute ℓ_{i+1}, we initialise it with ℓ_i and then divide it by two until the Hamming distance between the corresponding substrings of length ℓ_{i+1} is at most $\Delta - h_i$. From above it follows that the algorithm is correct with probability at least $1-\frac{1}{n^2}$ (we estimate error probability by the union bound) and that the query time is $\mathcal{O}(\log^2 n)$. This completes the proof of Theorem 2.

4 Proof of Theorem 1

Recall that we are given two strings T_1, T_2 of length n, and if ℓ_k is the length of the longest common substring with k mismatches of T_1 and T_2, the objective is to return a substring of length at least ℓ_k that occurs in T_1 and T_2 with at most $(1+\varepsilon)\cdot k$ mismatches.

4.1 Algorithm

We start by preprocessing T_1 and T_2 as described in Theorem 2. The main phase of the algorithm is defined by three parameters t, w, and m to be specified later, and consists of $\theta(t! \log n)$ independent steps.

At each step we choose $\binom{w}{t}$ hash functions, where each hash function can be considered as a t-tuple of projections of strings of length n onto subsets of their positions of size m. Let $\mathcal{H}$ be a set of all projections of strings of length n onto a single position, i.e. the value of the i-th projection on a string of length n is simply its i-th letter. We start by choosing a set of w functions $u_r \in \mathcal{H}^m$, $r = 1, 2, \dots, w$, uniformly at random. Each hash function h is defined to be a t-tuple of distinct functions u_r. More formally, $h = (u_{r_1}, u_{r_2}, \dots, u_{r_t}) \in \mathcal{H}^{mt}$, where $1 \leq r_1 < r_2 < \ldots < r_t \leq w$. The fact that the hash functions are constructed from a small set of functions u_j will ensure faster running time for the algorithm.

Consider the set of all suffixes $S_1, S_2, \dots, S_{2n}$ of T_1 and T_2. We append each suffix in the set with an appropriate number of letters $\$ \notin \Sigma$ so that all suffixes have length n and build a trie on strings $h(S_1), h(S_2), \dots, h(S_{2n})$.

Algorithm 1 Longest common substring with approximately k mismatches.

```
 1: Preprocess $T_1, T_2$ for LCP$_{\tilde{k}}$ queries
 2: for $i = 1, 2, \ldots, \theta(t! \log n)$ do
 3:     for $r = 1, 2, \ldots, w$ do
 4:         Choose a function $u_r \in \mathcal{H}^m$ uniformly at random
 5:         Preprocess $u_r$
 6:     end for
 7:     for all $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t})$ do
 8:         Build a trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$
 9:         Augment the trie with an LCA data structure
10:     end for
11:     for all suffixes $S$ of $T_1$ do
12:         Find the largest $\ell$ s.t. the total size of the $\ell$-neighbourhoods of $S$ is $\geq 2\binom{w}{t}$
13:         Select a set $\mathcal{S}$ of $2\binom{w}{t}$ suffixes from the $\ell$-neighbourhoods of $S$
14:         for all suffixes $S' \in \mathcal{S}$ do
15:             Compute LCP$_{\tilde{k}}(S, S')$
16:             Update the longest common substring with approximately $k$ mismatches
17:         end for
18:     end for
19: end for
```

▶ **Theorem 5.** *After $\mathcal{O}(wn^{4/3} \log^{4/3} n)$-time and $\mathcal{O}(wn)$-space preprocessing of functions u_r, $r = 1, 2, \ldots, w$, for any hash function $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t})$ a trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$ can be built in $\mathcal{O}(tn \log n)$ time and linear space.*

Let us defer the proof of the theorem until we complete the description of the algorithm and show Theorem 1. The algorithm preprocesses $u_1, u_2, \ldots, u_w$, and for each hash function h builds a trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$. It then augments the trie with an LCA data structure, which can be done in linear time and space [13, 19]. Given two strings $h(S_i), h(S_j)$ the LCA data structure can find the length of their longest common prefix in constant time.

Consider a suffix S of T_1 and let h be a hash function projecting a string of length n onto a subset $\mathcal{P} \subseteq [1, n]$ of its positions. We define $h\big|_{[\ell]}$ to be a projection onto a subset $\mathcal{P} \cap [1, \ell]$ of positions. (In other words, $h\big|_{[\ell]}$ is a function h applied to a prefix of a string of length ℓ.) We further say that the ℓ-neighbourhood of S is the set of all suffixes S' of T_2 such that $h\big|_{[\ell]}(S) = h\big|_{[\ell]}(S')$. Note that for a fixed h and ℓ the size of the ℓ-neighbourhood of S is equal to the number of suffixes S' of T_2 such that the length of the longest common prefix of $h(S)$ and $h(S')$ is at least $|\mathcal{P} \cap [1, \ell]|$. From the properties of the lexicographic order it follows that if S', S'' are two suffixes of T_2 and $h(S'')$ is located between $h(S')$ and $h(S)$ in the trie for h, then the longest common prefix of $h(S')$ and $h(S)$ is no longer than the longest common prefix of $h(S'')$ and $h(S)$. Therefore, the larger ℓ is, the smaller the neighbourhood is. We use a simple binary search and the LCA data structures to find the largest ℓ such that the total size of ℓ-neighbourhoods for all hash functions is at least $2\binom{w}{t}$ in $\mathcal{O}(\binom{w}{t} \cdot \log^2 n)$ time. From the union of the ℓ-neighbourhoods we select a multiset $\mathcal{S}$ of $2\binom{w}{t}$ suffixes ensuring that all suffixes S' such that the longest common prefix of $h(S')$ and $h(S)$ has length at least $|\mathcal{P} \cap [1, \ell]| + 1$ are included. For each suffix $S' \in \mathcal{S}$ we compute the longest common prefix with approximately k mismatches of S' and S by one LCP$_{\tilde{k}}$ query (Theorem 2). The longest of all retrieved prefixes, over all suffixes S of T_1, is returned as an answer. The algorithm is summarised in the figure above. We will now proceed to its complexity and correctness.

4.2 Complexity and correctness

To ensure the complexity bounds and correctness of the algorithm, we must carefully choose the parameters t, w, and m. Let $p_1 = 1 - k/n$, and $p_2 = 1 - (1+\varepsilon) \cdot k/n$. The intuition behind p_1 and p_2 is that if S_1, S_2 are two suffixes of length n and the Hamming distance between S_1 and S_2 is at most k, then p_1 is a lower bound for the probability of two letters $S_1[i], S_2[i]$ to be equal. On the other hand, p_2 is an upper bound for the probability of two letters $S_1[i], S_2[i]$ to be equal if the Hamming distance between S_1 and S_2 is at least $(1+\varepsilon) \cdot k$. Let $\rho = \log p_1 / \log p_2$, and define

$$t = \left\lceil \sqrt{\frac{\rho \log n}{\ln \log n}} \right\rceil + 1,\ w = \lceil n^{\rho/t} \rceil,\ \text{and } m = \left\lceil \frac{1}{t} \log_{p_2} \frac{1}{n^2} \right\rceil.$$

4.2.1 Complexity

To show the time complexity of the algorithm, we will start from the following simple observation.

▶ **Observation 6.** $\rho \leq 1/(1+\varepsilon)$ and $2 \leq t \leq \sqrt{\log n}$.

Proof. By Bernoulli's inequality $(1-k/n)^{1+\varepsilon} \geq 1 - (1+\varepsilon) \cdot k/n$. Hence, we obtain that $\rho = \frac{\log(1-k/n)}{\log(1-(1+\varepsilon)k/n)} \leq \frac{1}{1+\varepsilon}$. The second part of the lemma follows. ◀

▶ **Lemma 7.** *One step of the algorithm takes* $\mathcal{O}(wn^{4/3} \log^{4/3} n + \binom{w}{t} \cdot n \log^2 n)$ *time.*

Proof. By Theorem 5, after $\mathcal{O}(wn^{4/3} \log^{4/3} n)$-time preprocessing we can build a trie and an LCA data structure on strings $h(S_1), h(S_2), \ldots, h(S_{2n})$ for a hash function h in $\mathcal{O}(tn \log n) = \mathcal{O}(n \log^{3/2} n)$ time and there are $\binom{w}{t}$ hash functions in total. For each suffix of T_1 we then run $2\binom{w}{t}$ LCP$_{\tilde{k}}$ queries, which takes $\mathcal{O}(\binom{w}{t} \cdot n \log^2 n)$ time. ◀

▶ **Corollary 8.** *The running time of the algorithm is* $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$.

Proof. Preprocessing T_1, T_2 for LCP$_{\tilde{k}}$ queries takes $\mathcal{O}(n \log^3 n / \varepsilon^2)$ time (see Theorem 2). Each step of the algorithm takes $\mathcal{O}(wn^{4/3} \log^{4/3} n + \binom{w}{t} \cdot n \log^2 n)$ time, and there are $\theta(t!\ \log n)$ steps in total. To estimate the total running time of the algorithm we notice that $wt! = \mathcal{O}(e^{\sqrt{\rho \log n \ln \log n}}) = \mathcal{O}(n^{o(1)})$ and $\binom{w}{t} \cdot t! \leq \frac{w^t}{t!} t! = n^\rho \leq n^{1/(1+\varepsilon)}$. Plugging these inequalities into the time bound for one step and recalling that $0 < \varepsilon < 2$, we obtain the claim. ◀

▶ **Lemma 9.** *The space complexity of the algorithm is* $\mathcal{O}(n^{1+1/(1+\varepsilon)})$.

Proof. The data structure for LCP$_{\tilde{k}}$ queries requires $\mathcal{O}(n \log^3 n)$ space. At each step of the algorithm, preprocessing functions u_j requires $\mathcal{O}(wn) = \mathcal{O}(n^{1+o(1)})$ space and the tries occupy $\mathcal{O}(\binom{w}{t} \cdot n) = \mathcal{O}(n^{1+1/(1+\varepsilon)})$ space. ◀

4.2.2 Correctness

Let S be a suffix of T_1. Consider a set of the longest common prefixes with k mismatches of S and suffixes of T_2 and let ℓ_k be the maximal length of a prefix in this set achieved on some suffix S' of T_2.

▶ **Lemma 10.** *For each step with probability $\geq \theta(1/t!)$ there exists a hash function h such that $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$.*

Proof. Consider strings $S[1, \ell_k]\$^{n-\ell_k}$ and $S'[1, \ell_k]\$^{n-\ell_k}$. The Hamming distance between them is equal to the Hamming distance between $S[1, \ell_k]$ and $S'[1, \ell_k]$, which is k. Moreover, for any hash function h we have that $h(S[1, \ell_k]\$^{n-\ell_k}) = h(S'[1, \ell_k]\$^{n-\ell_k})$ if and only if $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$. Remember that each hash function is a t-tuple of functions u_j. Consequently, if $h(S[1, \ell_k]\$^{n-\ell_k}) \neq h(S'[1, \ell_k]\$^{n-\ell_k})$ for all hash functions h, the strings collide on at most $t-1$ functions u_j. By [4, Lemma A.1] the probability of this event for two strings at the Hamming distance k is at most $1 - \theta(1/t!)$. ◀

As a corollary, we can choose the constant in the number of steps so that with probability $\geq 1 - 1/n^2$ there will exist a step of algorithm such that for at least one hash function we will have $h\big|_{[\ell_k]}(S) = h\big|_{[\ell_k]}(S')$. The set $\mathcal{S}$ of $2\binom{w}{t}$ suffixes that we sample from the ℓ-neighbourhoods of S might or might not include the suffix S'. If it does, then the LCP$_{\tilde{k}}$ query for S' and S will return a substring of length $\geq \ell_k$ with high probability. If it does not, then by the definition of neighbourhoods for each suffix $S'' \in \mathcal{S}$ belonging to the neighbourhood for a hash function g we have $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$. We will show that only for a small number of such suffixes an LCP$_{\tilde{k}}$ query can return a substring of length smaller than ℓ_k.

▶ **Lemma 11.** *With probability $\geq 1 - 2/n^2$ there are at most $\binom{w}{t}$ suffixes S'' such that $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$ but the* LCP$_{\tilde{k}}$ *query returns a substring shorter than ℓ_k.*

Proof. If the LCP$_{\tilde{k}}$ query for S and S'' returns a substring shorter than ℓ_k, then with high probability the Hamming distance between $S[1, \ell_k]$ and $S''[1, \ell_k]$ is at least $(1 + \varepsilon) \cdot k$. Remember that a hash function g can be considered as a projection onto a random subset of positions of size mt, and therefore we obtain

$$\Pr[g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')] = \Pr[g(S[1, \ell_k]\$^{n-\ell_k}) = g(S'')[1, \ell_k]\$^{n-\ell_k}] \leq (p_2)^{mt} = \frac{1}{n^2}$$

We can consider an indicator random variable that is equal to one if for a suffix S'' such that $g\big|_{[\ell_k]}(S) = g\big|_{[\ell_k]}(S'')$ the LCP$_{\tilde{k}}$ query returns a substring shorter than ℓ_k, and to zero otherwise. By linearity, the expectation of their sum is at most $\frac{2}{n^2} \cdot \binom{w}{t}$. The claim follows from Markov's inequality. ◀

From Lemmas 10 and 11 and Theorem 2 it follows that the algorithm correctly finds the value of ℓ for the suffix S of T_1 with error probability $\leq 3/n^2$. Applying the union bound, we obtain that the error probability of the algorithm is constant.

4.3 Proof of Theorem 5

Recall that h is a t-tuple of functions u_r, i.e $h = (u_{r_1}, u_{r_2}, \ldots, u_{r_t})$, where $1 \leq r_1 < r_2 < \ldots < r_t \leq w$. Below we will show that after the preprocessing of functions u_r we will be able to compute the longest common prefix of any two strings $u_r(S_i), u_r(S_j)$ in $\mathcal{O}(1)$ time. As a result, we will be able to compute the longest common prefix of $h(S_i), h(S_j)$ in $\mathcal{O}(t)$ time. It also follows that we will be able to compare any two strings $h(S_i), h(S_j)$ in $\mathcal{O}(t)$ time as their order is defined by the letter following the longest common prefix. Therefore, we can sort strings $h(S_1), h(S_2), \ldots, h(S_{2n})$ in $\mathcal{O}(tn \log n)$ time and $\mathcal{O}(n)$ space and then compute the longest common prefix of each two adjacent strings in $\mathcal{O}(tn)$ time. The trie on $h(S_1), h(S_2), \ldots, h(S_{2n})$ can then be built in $\mathcal{O}(n)$ time by imitating its depth-first traverse.

It remains to explain how we preprocess functions u_r, $r = 1, 2, \ldots, w$. For each function u_r it suffices to build a trie on strings $u_r(S_1), u_r(S_2), \ldots, u_r(S_{2n})$ and to augment it with an LCA data structure [13, 19]. We will consider two different methods for constructing the trie with time dependent on m. No matter what the value of m is, one of these methods will have $\mathcal{O}(n^{4/3} \log^{1/3} n)$ running time. Let u_r be a projection along a subset $\mathcal{P}$ of positions $1 \le p_1 \le p_2 \le \cdots \le p_m \le n$ and denote $T = T_1\$^n T_2\$^n$.

▶ **Lemma 12.** *The trie can be built in $\mathcal{O}(\sqrt{m}n \log n)$ time and $\mathcal{O}(n)$ space correctly with error probability at most $1/n^3$.*

Proof. We partition $\mathcal{P}$ into disjoint subsets $B_1, B_2, \ldots, B_{\sqrt{m}}$, where

$$B_\ell = \{p_{\ell,1}, p_{\ell,2}, \ldots, p_{\ell,\sqrt{m}}\} = \{p_{(\ell-1)\sqrt{m}+q} \mid q \in [1, \sqrt{m}]\}.$$

Now u_r can be represented as a $\sqrt{m}$-tuple of projections $b_1, b_2, \ldots, b_{\sqrt{m}}$ onto the subsets $B_1, B_2, \ldots, B_{\sqrt{m}}$ respectively. We will build the trie by layers to avoid space overhead. Suppose that we have built the trie for a function $(b_1, b_2, \ldots, b_{\ell-1})$ and we want to extend it to the trie for $(b_1, b_2, \ldots, b_{\ell-1}, b_\ell)$.

Let p be a random prime of value at most $n^{\mathcal{O}(1)}$. We create a vector χ of length n, where $\chi[p_{\ell,q}] = 2^{\sqrt{m}-1-q}$ and zero for all positions not in B_ℓ. We then run the FFT algorithm for χ and T in the field $\mathbb{Z}_p$ [14]. The output of the FFT algorithm will contain convolutions of χ and all suffixes $S_1, S_2, \ldots, S_{2n}$. The convolution of χ and a suffix S_i is the Karp-Rabin fingerprint [23] $\varphi_{\ell,i}$ of $b_\ell(S_i)$, where

$$\varphi_{\ell,i} = \sum_{q=1}^{\sqrt{m}} S_i[p_{\ell,q}] \cdot 2^{\sqrt{m}-1-q} \pmod p$$

If the fingerprints of $b_\ell(S_i)$ and $b_\ell(S_j)$ are equal, then $b_\ell(S_i)$ and $b_\ell(S_j)$ are equal with probability at least $1 - \frac{1}{n^4}$, and otherwise they differ. For a fixed leaf of the trie for $(b_1, b_2, \ldots, b_{\ell-1})$ we sort all suffixes that end in it by fingerprints $\varphi_{\ell,i}$, which takes $\mathcal{O}(n \log n)$ time in total. For each two suffixes S_i, S_j that end in the same leaf, adjacent and have $\varphi_{\ell,i} \neq \varphi_{\ell,j}$, we compare $b_\ell(S_i)$ and $b_\ell(S_j)$ letter-by-letter in $\mathcal{O}(\sqrt{m})$ time to find their longest common prefix. Note that this letter-by-letter comparison step will be executed at most n times, and therefore will take $\mathcal{O}(\sqrt{m}n)$ time in total. We then append each leaf with a trie on strings $b_\ell(S_i)$ that can be built by imitating its depth-first traverse, which takes $\mathcal{O}(n)$ time for a layer. ◀

The second method builds the trie in $\mathcal{O}(n^2 \log^2 n/m)$ time by the algorithm described in the first paragraph of this section, and we only need to give a method for comparing the longest common prefix of $u_r(S_i)$ and $u_r(S_j)$ (or, equivalently, the first position where $u_r(S_i)$ and $u_r(S_j)$ differ.)

▶ **Lemma 13** ([4]). *After $\mathcal{O}(n)$-time and space preprocessing the first position where two strings $u_r(S_i)$ and $u_r(S_j)$ differ can be found in $\mathcal{O}(n \log n/m)$ time correctly with error probability at most $1/n^3$.*

Proof. We start by building the suffix tree for the string T. The suffix tree can be built in $\mathcal{O}(n)$ time and space [31, 12, 18]. Furthermore, we augment the suffix tree with an LCA data structure in $\mathcal{O}(n)$ time [13, 19].

Let $\ell = 3n \log n/m$. We can find the first ℓ positions $q_1 < q_2 < \ldots < q_\ell$ where S_i and S_j differ in $\mathcal{O}(n \log n/m)$ time using the kangaroo method [26, 16]. The idea of the kangaroo method is as follows. We can find q_1 by one query to the LCA data structure in $\mathcal{O}(1)$ time. After removing the first q_1 positions of S_i, S_j, we obtain suffixes S_{i+q_1}, S_{j+q_1} and find q_2 by another query to the LCA data structure, and so on. If at least one of the positions $q_1, q_2, \ldots, q_\ell$ belongs to $\mathcal{P}$, then we return the first such position as an answer, and otherwise we say that $u_r(S_i) = u_r(S_j)$.

Let us show that if p is the first position where $u_r(S_i)$ and $u_r(S_j)$ differ, then p belongs to $\{q_1, q_2, \ldots, q_\ell\}$ with high probability. Because $q_1 < q_2 < \ldots < q_\ell$ are the first ℓ positions where S_i and S_j differ, it suffices to show that at least one of these positions belongs to $\mathcal{P}$. We rely on the fact $\mathcal{P}$ is a random subset of $[1, n]$. We have $Pr[q_1, q_2, \ldots, q_\ell \notin \mathcal{P}] = (1 - \ell/n)^m = (1 - 3\log n/m)^m \leq n^{-3}$. ◀

As a corollary of Lemmas 12 and 13, the trie on strings $u_r(S_1), u_r(S_2), \ldots, u_r(S_{2n})$ can be built in $\mathcal{O}(\min\{\sqrt{m}, n \log n/m\} \cdot n \log n) = \mathcal{O}(n^{4/3} \log^{4/3} n)$ time and $\mathcal{O}(n)$ space correctly with high probability which implies Theorem 5 as explained in the beginning of this section.

References

1 A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 218–230, 2015.

2 N. Alon and J. Spencer. *The Probabilistic Method.* John Wiley and Sons, 1992.

3 S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

4 A. Andoni and P. Indyk. Efficient algorithms for substring near neighbor problem. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1203–1212, 2006.

5 M. Babenko and T. Starikovskaya. Computing longest common substrings via suffix arrays. In *Proceedings of the 3rd International Computer Science Symposium in Russia*, pages 64–75, 2008.

6 M. Babenko and T. Starikovskaya. Computing the longest common substring with one mismatch. *Problems of Information Transmission*, 47(1):28–33, 2011.

7 A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (Unless SETH is false). In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing (STOC)*, pages 51–58, 2015.

8 P. Bille, I.L. Gørtz, and J. Kristensen. Longest common extensions via fingerprinting. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications*, pages 119–130, 2012.

9 P. Bille, I.L. Gørtz, B. Sach, and H.W. Vildhøj. Time-space trade-offs for longest common extensions. *J. of Discrete Algorithms*, 25:42–50, March 2014.

10 K. Bringmann and M. Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 79–97, 2015.

11 M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.

12 M. Farach-Colton. Optimal suffix tree construction with large alphabets. In *Procedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

13 J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Conference on Combinatorial Pattern Matching*, pages 36–48, 2006.

14 M. J. Fischer and M. S. Paterson. String-matching and other products. In *Complexity of Computation*, volume 7, pages 113–125. SIAM AMS, 1974.

15 T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6–8):643 – 647, 2015.

16 Z. Galil and R. Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science*, 51(3):341 – 348, 1987.

17 S. Grabowski. A note on the longest common substring with k-mismatches problem. *Information Processing Letters*, 115(6–8):640 – 642, 2015.

18 D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

19 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.

20 L.C.K. Hui. Color set size problem with applications to string matching. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture notes in computer science*, pages 230–243, 1992.

21 L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. of Discrete Algorithms*, 8(4):418–428, 2010.

22 P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

23 R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development - Mathematics and computing*, 31(2):249 –260, 1987.

24 T. Kociumaka, T. Starikovskaya, and Vildhøj H. W. Sublinear space algorithms for the longest common substring problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms*, volume 8737 of *Lecture notes in computer science*, pages 605–617, 2014.

25 E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 614–623, New York, NY, USA, 1998. ACM.

26 G.M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239 – 249, 1986.

27 C.A. Leimeister and B. Morgenstern. kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.

28 W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

29 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

30 T. Starikovskaya and H.W. Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proceedings of the 24th Annual Symposium in Combinatorial Pattern Matching*, volume 7922 of *Lecture notes in computer science*, pages 223–234, 2013.

31 P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.