Efficient Computation of Sequence Mappability

Panagiotis Charalampopoulos, Costas Iliopoulos, Tomasz Kociumaka, Solon Pissis, Jakub Radoszewski, Juliusz Straszyński

Efficient Computation of Sequence Mappability

Panagiotis Charalampopoulos[1], Costas S. Iliopoulos[2], Tomasz Kociumaka[3], Solon P. Pissis[4,5], Jakub Radoszewski*[6], and Juliusz Straszyński*[6]

[1]Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Herzliya, Israel, panagiotis.charalampopoulos@post.idc.ac.il
[2]Department of Informatics, King's College London, London, UK, c.iliopoulos@kcl.ac.uk
[3]University of California, Berkeley, USA, kociumaka@berkeley.edu
[4]CWI, Amsterdam, The Netherlands, solon.pissis@cwi.nl
[5]Vrije Universiteit, Amsterdam, The Netherlands
[6]Institute of Informatics, University of Warsaw, Warsaw, Poland, [jrad,jks]@mimuw.edu.pl

Abstract

Sequence mappability is an important task in genome resequencing. In the (k, m)-mappability problem, for a given sequence T of length n, the goal is to compute a table whose ith entry is the number of indices $j \neq i$ such that the length-m substrings of T starting at positions i and j have at most k mismatches. Previous works on this problem focused on heuristics computing a rough approximation of the result or on the case of $k = 1$. We present several efficient algorithms for the general case of the problem. Our main result is an algorithm that, for $k = \mathcal{O}(1)$, works in $\mathcal{O}(n)$ space and, with high probability, in $\mathcal{O}(n \cdot \min\{m^k, \log^k n\})$ time. Our algorithm requires a careful adaptation of the k-errata trees of Cole et al. [STOC 2004] to avoid multiple counting of pairs of substrings. Our technique can also be applied to solve the all-pairs Hamming distance problem introduced by Crochemore et al. [WABI 2017]. We further develop $\mathcal{O}(n^2)$-time algorithms to compute *all* (k, m)-mappability tables for a fixed m and all $k \in \{0, \ldots, m\}$ or a fixed k and all $m \in \{k, \ldots, n\}$. Finally, we show that, for $k, m = \Theta(\log n)$, the (k, m)-mappability problem cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis fails.

This is an improved and extended version of a paper that was presented at SPIRE 2018.

1 Introduction

The k-mappability problem. Analyzing data derived from massively parallel sequencing experiments often depends on the process of genome assembly via resequencing; namely, assembly with the help of a reference sequence. In this process, a large number of reads (or short sequences) derived from a DNA donor during these experiments must be mapped back to a reference sequence, comprising a few gigabases, to establish the section of the genome from which each read has been derived. An extensive number of short-read alignment techniques and tools have been introduced to address this challenge emphasizing on different aspects of the process [15].

In turn, the process of resequencing depends heavily on how mappable a genome is with respect to reads of some fixed length m. Thus, given a reference sequence, for every substring of length m in the sequence, we want to count how many additional times this substring appears in the sequence when allowing for a small number k of errors. This computational problem and a heuristic approach to approximate the solution were first proposed in [12] (see also [5]), where a great variance in genome mappability between species and gene classes was revealed.

*Supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

More formally, for a string T, let T_i^m denote the length-m substring of T that starts at position i. In the (k, m)-mappability problem, for a given string T of length n, we are asked to compute a table $\mathrm{A}_{\leq k}^m$ whose ith entry $\mathrm{A}_{\leq k}^m[i]$ is the number of indices $j \neq i$ such that the substrings T_i^m and T_j^m are at Hamming distance at most k. In the previous study [12], the assumed values of parameters were $k \leq 4$, $m \leq 100$, and the alphabet of T was $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Example 1. Consider a string $T = \mathtt{aababba}$ and $m = 3$. The following table shows the (k, m)-mappability counts for $k = 1$ and $k = 2$.

position	i	1	2	3	4	5
substring	T_i^3	aab	aba	bab	abb	bba
$(1,3)$**-mappability**	$\mathrm{A}_{\leq 1}^3[i]$	2	2	1	2	1
$(2,3)$**-mappability**	$\mathrm{A}_{\leq 2}^3[i]$	3	3	3	4	3
difference	$\mathrm{A}_{=2}^3[i]$	1	1	2	2	2

For instance, consider the position 1. The $(1, 3)$-mappability is 2 due to the occurrences of bab and abb at positions 3 and 4, respectively. The $(2, 3)$-mappability is 3 since only the substring bba, occurring at position 5, has three mismatches with aab.

For convenience, our algorithms compute an array $\mathrm{A}_{=k}^m$ whose ith entry $\mathrm{A}_{=k}^m[i]$ is the number of positions $j \neq i$ such that substrings T_i^m and T_j^m are at Hamming distance *exactly* k. Note that $\mathrm{A}_{\leq k}^m[i] = \sum_{\kappa=0}^{k} \mathrm{A}_{=\kappa}^m[i]$; see the "difference" row in the example above. Henceforth, we call this problem *the* (k, m)*-mappability problem*.

Solution	**Time complexity**
Manzini [29]	$\mathcal{O}(mn \log n / \log\log n)$
Alzamel et al. [3]	$\mathcal{O}(nm)$
Alzamel et al. [3]	$\mathcal{O}(n \log n \log\log n)$
Alzamel et al. [3]	$\mathcal{O}(n)$ on average for $m = \Omega(\log n)$
Amir et al. [4], Hooshmand et al. [20]	$\mathcal{O}(n \log n)$
Amir et al. [4]	$\mathcal{O}(n)$ for $m = \Omega(\sqrt{n})$

Table 1: Known algorithms for computing $(1, m)$-mappability for strings over constant-sized alphabets. All algorithms use $\mathcal{O}(n)$ space.

Using the suffix array and the LCP table [28, 25, 23], the $(0, m)$-mappability problem can be solved in $\mathcal{O}(n)$ time and space. Known solutions for computing $(1, m)$-mappability are shown in Table 1; the $\mathcal{O}(nm)$-time and the $\mathcal{O}(n)$-average-time solutions of Alzamel et al. [3] work also on strings over *integer alphabets* $\{1, \ldots, \sigma\}$ for $\sigma = n^{\mathcal{O}(1)}$. Moreover, the latter algorithm was shown to be generalizable to arbitrary k, requiring $\mathcal{O}(n)$ space and, on average, $\mathcal{O}(kn)$ time if $m = \Omega(k \log_\sigma n)$. In [1], the authors introduced an efficient construction of a *genome mappability array* B_k in which $B_k[\mu]$ is the smallest length m such that at least μ of the length-m substrings of T do not occur elsewhere in T with at most k mismatches. This construction was further improved in [6].

The all-pairs Hamming distance problem. The evolutionary relationships between different species or taxa are usually inferred through phylogenetic analysis techniques. Some of these techniques rely on the inference of phylogenetic trees. A first step of these techniques is to compute the distances between all pairs of sequences representing the set of species or taxa under study. This particular step, however, often dominates the running time of these methods. Depending on the application, the underlying model of evolution, and the optimality criterion, it may not be strictly necessary to be aware of the complete distance matrix (see [16, 11], for instance). Thus, in this preprocessing step, we are only interested in pairs with distances not exceeding a given threshold.

The computational problem can be formally defined as follows. Given a set $\mathbf{R}$ of r length-m strings and an integer $k \in \{0, \ldots, m\}$, return all pairs $(X_1, X_2) \in \mathbf{R} \times \mathbf{R}$, with $X_1 \neq X_2$, such that X_1 and X_2 are at Hamming distance at most k. This problem has been studied in the average-case model and efficient linear-time algorithms are known under some constraints on the value of k and some assumptions on the elements of $\mathbf{R}$ [11, 30, 19]. The indexing variant of the all-pairs Hamming distance problem has further applications in bioinformatics for querying typing databases [8] and in information retrieval for searching similar documents in a collection [18].

Intuitively, the connection between the (k, m)-mappability problem and the all-pairs Hamming distance problem is as follows. By first concatenating the r elements of $\mathbf{R}$ to construct a new string T of length $n = rm$, solving the former considering only the r substrings of T starting at positions i, with $i \bmod m = 1$, and summing up the resulting values, we would obtain the total size of the output of the latter.

Henceforth we assume, as in the mappability problem, that we are to compute all pairs at Hamming distance *exactly* k. In the end, we run the algorithm for all values of k up to a given threshold of interest.

Our contributions. We present several algorithms for the general case of the (k, m)-mappability problem. More specifically, our contributions are as follows:

1. In Section 3, we show a randomized Las-Vegas algorithm for the (k, m)-mappability problem that works in $\mathcal{O}(n\binom{\log n+k}{k}4^k k)$ time with high probability ($1 - n^{-c}$ for an arbitrarily large constant parameter $c > 0$) and $\mathcal{O}(n2^k k)$ space for a string over an ordered alphabet. It requires a careful adaptation of the technique of recursive heavy-path decompositions in a tree [10].

2. In Section 4, we show an algorithm to solve all-pairs Hamming distance problem in time $\mathcal{O}(rm + r\binom{\log r+k}{k}4^k k \log r + \mathsf{output}\, 2^k k \log r)$ and space $\mathcal{O}(rm + r2^k k \log r)$.

3. In Section 5, we show an algorithm for the (k, m)-mappability problem that works in $\mathcal{O}(nk \cdot (m+1)^k)$ time and $\mathcal{O}(n)$ space for a string over an integer alphabet. Together with the first result, this yields an $\mathcal{O}(n \cdot \min\{m^k, \log^k n\})$-time and $\mathcal{O}(n)$-space algorithm for $k = \mathcal{O}(1)$.

4. In Section 6, we show $\mathcal{O}(n^2)$-time algorithms to compute *all* (k, m)-mappability tables for a fixed m and all $k \in \{0, \ldots, m\}$, or for a fixed k and all $m \in \{k, \ldots, n\}$.

5. Finally, in Section 7, we prove that the (k, m)-mappability problem for $k, m = \Theta(\log n)$ cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis [22, 21] fails.

In contributions 1 and 5, we apply recent advances in the Longest Common Substring with k Mismatches problem that were presented in [9] and [26], respectively (see also [32]). In particular, compared to [9], our contribution 1 requires a careful counting of substring pairs to avoid multiple counting and a thorough analysis of the space usage. Technically this is the most involved contribution.

This work is an extended version of [2]. In comparison to the conference version, we improve the complexity of the main algorithm by a $\Theta(\log n)$-factor, remove the dependency on the alphabet size in contribution 3, and apply our techniques to solve the all-pairs Hamming distance problem (contribution 2).

2 Preliminaries

Let $T = T[1]T[2]\cdots T[n]$ be a *string* of length $|T| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma$. For two positions i and j on T, the *substring* (sometimes called *factor*) of T that starts at position i and ends at position j is $T[i]\cdots T[j]$ (it is of length 0 if $j < i$). A *prefix* of T is a substring that starts at position 1 and a *suffix* of T is a substring that ends at position n. We denote the suffix that starts at position i by T_i and its prefix of length m by T_i^m.

The *Hamming distance* between two strings S and T of the same length $|S| = |T|$ is defined as $d_H(S, T) = |\{i \in \{1, 2, \ldots, |S|\} : S[i] \neq T[i]\}|$. If $|S| \neq |T|$, we set $d_H(S, T) = \infty$.

By $\mathsf{lcp}(S,T)$ we denote the length of the longest common prefix of S and T. For a fixed string T, we also set $\mathsf{lce}(r,s) = \mathsf{lcp}(T_r, T_s)$. By $\mathsf{lce}_k(r,s)$ we denote the length of the longest common prefix of T_r and T_s when up to k mismatches are allowed, that is, the maximum ℓ such that $d_H(T_r^\ell, T_s^\ell) \le k$.

Compact trie. A *trie* of a collection of strings C is a labeled tree that contains a node for every distinct prefix of a string in C; the root node is ε; the set of *terminal* nodes is C; and edges are of the form $u \xrightarrow{c} uc$, where u and uc are nodes and $c \in \Sigma$. A compact trie $\mathbf{T}$ of a collection of strings C is obtained from the trie of C by dissolving all non-branching nodes, excluding the root and the terminals. The nodes of the trie which become nodes of $\mathbf{T}$ are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of $\mathbf{T}$ can be viewed as an upward maximal path of implicit nodes starting with an explicit node. The string label of an edge is a substring of one of the strings in C; the label of an edge is the first letter of the edge's string label. Each node of the trie can be represented in $\mathbf{T}$ 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 string labels of the edges along the path from the root to v. Additionally, $\mathbf{D}(v) = |\mathcal{L}(v)|$ is the *string-depth* of node v.

Suffix tree. The suffix tree of a string T is the compact trie representing all suffixes of T. The suffix tree of a string T of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time [14] and, after an $\mathcal{O}(n)$-time preprocessing [7], it can be used to answer $\mathsf{lce}(r,s)$ queries in $\mathcal{O}(1)$ time.

Hashing. We use perfect hashing to implement dynamic dictionaries supporting insertions and deletions of entries (key-value pairs), as well as to retrieve an arbitrary entry with a given key. Technically, we maintain a single global dictionary (which may simulate multiple local dictionaries) implemented using [13, Theorem 1.1]. In the preprocessing, we insert n dummy entries; this incurs extra $\mathcal{O}(n)$ terms in the time and space complexities, but also guarantees that the running time of every operation is $\mathcal{O}(1)$ with probability at least $1 - n^{-c}$, where $c > 0$ is a constant specified at initialization time. As long as the total number of dictionary operations is polynomial in n, we derive Las-Vegas algorithms whose running times bounds hold with high probability (rather than just in expectation). Whenever the time complexity of any algorithm in this work is superpolynomial in n (which may happen for large values of k), we resort to naive polynomial-time solutions.

When using strings as dictionary keys, we rely on Karp–Rabin fingerprints (polynomial hashing) [24] with collision probability bounded by n^{-C} for strings of length at most n (and a sufficiently large constant C). In order to obtain Las-Vegas algorithms, we provide mechanisms for detecting collisions and resort to naive polynomial-time solutions upon detecting any.

3 Computing Mappability in $\mathcal{O}(n \log^k n)$ Time and $\mathcal{O}(n)$ Space

Our algorithm operates on so-called *modified strings*. A modified string α is a string U with a set of modifications M. Each element of the set M is a pair of the form (i, c) which denotes a substitution "$U[i] := c$". We assume that no two pairs in M share the same index i. By $val(\alpha)$ we denote the string U after all the substitutions and by $M(\alpha)$ we denote the set M. The sets $M(\alpha)$ for modified strings are implemented as (functional) lists. Whenever a modified string β is obtained by introducing an extra modification to a modified string α, the head of $M(\beta)$ represents the new modification whereas the tail points to $M(\alpha)$. We always introduce modifications in the left-to-right order so that the lists $M(\alpha)$ are sorted according to the decreasing order of indices i.

The algorithm processes *modified substrings* of T that are modified strings originating from the substrings T_i^m. For a modified substring α originating from T_i^m, we denote $idx(\alpha) = i$.

Overview of the algorithm. Intuitively, the algorithm proceeds by efficiently simulating transformations of a compact trie of modified substrings, initially containing all substrings T_i^m. The elementary transformations are guided by the *smaller-to-larger* principle, and each of them consists in copying one subtree unto its sibling, with an appropriate modification introduced to each copied substring in order to match the label of

the edge leading to the sibling. This process effectively results in registering one mismatch for a large batch of substrings at once, and therefore lays a foundation to solve the main problem in the aforementioned time.

More precisely, the algorithm navigates a compact trie of modified substrings.[1] The trie is constructed top-down recursively, and the final set of modified substrings that are present in the trie is known only when all the leaves of the trie have been reached.

In a recursive step, a node v of the trie stores a set of modified substrings $MS(v)$. Initially, the root r stores all substrings T_i^m in its set $MS(r)$. The path-label $\mathcal{L}(v)$ is the longest common prefix of (the values of) all the modified substrings in $MS(v)$ and the string-depth $\mathbf{D}(v)$ is the length of this prefix. None of the strings in $MS(v)$ contains a modification at a position greater than $\mathbf{D}(v)$. The children of v are determined by subsets of $MS(v)$ that correspond to different letters at position $\mathbf{D}(v)+1$. Furthermore, additional modified substrings with modifications at position $\mathbf{D}(v)+1$ are created and inserted into the children's MS-sets. This corresponds to the intuition of copying subtrees unto their siblings.

The goal is to distribute the modified substrings into leaves and, by processing each leaf independently, register exactly once every pair of substrings (T_i^m, T_j^m) differing on exactly k positions.

Now, we will describe the recursive routine for visiting a node.

Processing an internal node. Assume that our node v has children $u_1, \ldots, u_a$. First, we distinguish a child of v with maximum-size set MS; let it be u_1. We will refer to this child as *heavy* and to every other as *light*. We will recursively branch into each child to take care of all pairs of modified substrings contained in any single subtree.

For this, we create an extra child u_{a+1} so that $MS(u_{a+1})$ contains all modified substrings from $MS(u_2) \cup \cdots \cup MS(u_a)$ with the letters at position $\mathbf{D}(v)+1$ replaced by a common wildcard character \$. By processing the subtree of u_{a+1}, we will consider pairs of modified substrings that originate from different light children.

Additionally, we insert all modified substrings from $MS(u_2) \cup \cdots \cup MS(u_a)$ into $MS(u_1)$, substituting the letter at position $\mathbf{D}(v)+1$ with the common letter at this position of modified substrings in $MS(u_1)$. This transformation will take care of pairs between the heavy child and the light ones.

Finally, the algorithm branches into the subtrees of $u_1, \ldots, u_{a+1}$. A pseudocode of this process is presented as Algorithm 1. Note that in the special case of a binary alphabet the child u_{a+1} need not be created. Moreover, since modified substrings with more than k substitutions are irrelevant for our algorithm, we refrain from creating them in the interest of time and space complexity.

Processing a leaf. Each modified substring α stores its index of origin $idx(\alpha)$ and the set of modifications $M(\alpha)$. As we have seen, the substitutions introduced in the recursion are of two types: of wildcard origin and of heavy origin. For a modified substring α, we introduce a partition $M(\alpha) = W(\alpha) \cup H(\alpha)$ into modifications of these kinds. For every leaf v, the modified substrings $\alpha \in MS(v)$ share the same value $val(\alpha)$, and hence $W(\alpha)$ is also the same. Finally, by $W^{-1}(\alpha)$ we denote the set $\{(j, T^m_{idx(\alpha)}[j]) : (j, \$) \in W(\alpha)\}$. We call modified substrings $\alpha, \beta \in MS(v)$ *compatible* if they satisfy the following condition:

$$H(\alpha) \cap H(\beta) = \emptyset, \quad W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset, \quad |H(\alpha)| + |H(\beta)| + |W(\alpha)| = k. \tag{1}$$

Intuitively, α and β are compatible only if the positions of modifications in $M(\alpha) \cup M(\beta)$ do not contain any position j such that $T^m_{idx(\alpha)}[j] = T^m_{idx(\beta)}[j]$. As proved in Lemma 4 below, for every $\alpha \in MS(v)$, we should increment $A^m_{=k}[idx(\alpha)]$ for each compatible $\beta \in MS(v)$. We next show how to efficiently count these modified substrings using the inclusion-exclusion principle and several precomputed values, as we cannot afford to count them naively.

For convenience, let $R(\alpha)$ denote the union of disjoint sets $H(\alpha)$ and $W^{-1}(\alpha)$. For a leaf v, let $Count(s, B)$ denote the number of modified substrings $\beta \in MS(v)$ such that $|H(\beta)| = s$ and $B \subseteq R(\beta)$. All the non-zero values are stored in a hash table. They can be generated by iterating through all the subsets of $R(\beta)$ for all modified substrings $\beta \in MS(v)$; this costs $\mathcal{O}(2^k k |MS(v)|)$ time and space. Finally, the result for a modified substring α can be computed using the following direct consequence of the inclusion-exclusion principle.

[1] The true course of the algorithm will not actually perform much of its operations on a compact trie, but the intuition is best conveyed by visualizing them this way.

Algorithm 1: A recursive procedure of processing a trie node

```
Procedure processNode(v)
    lcp(v): computes the longest common prefix of all the strings in MS(v)
    insert(v, α): inserts α into MS(v)
    splitByLetter(v, index): splits MS(v) into groups having the same index-th letter, returning a
                             list of sets of modified substrings

    depth ← lcp(v)
    if depth = m then
        processLeaf(v)
        return
    children ← splitByLetter(v, depth + 1)
    heavyChild ← findHeaviest(children)
    heavyLetter ← val(α)[depth+1] for some α ∈ heavyChild
    wildcardTree ← ∅
    foreach lightChild ∈ children \ {heavyChild} do
        foreach α ∈ lightChild do
            if |M(α)| < k then
                α' ← α
                α'[depth+1] ← $
                insert(wildcardTree, α')
                α'' ← α
                α''[depth+1] ← heavyLetter
                insert(heavyChild, α'')
    foreach child ∈ children ∪ {wildcardTree} do
        processNode(child)
```

Lemma 2. *The number of modified substrings $\beta \in MS(v)$ that are compatible with a modified substring $\alpha \in MS(v)$ is $\sum_{B \subseteq R(\alpha)} (-1)^{|B|} Count(k - |M(\alpha)|, B)$.*

Proof. First, let $h = k - |M(\alpha)|$. We want to count the modified substrings $\beta \in MS(v)$ that satisfy $|H(\beta)| = h$ and $R(\alpha) \cap R(\beta) = \emptyset$. For $(i, x) \in R(\alpha)$, let $A_{(i,x)} = \{\beta \in MS(v) : |H(\beta)| = h \text{ and } (i, x) \in R(\beta)\}$. Then, we want to compute $Count(h, \emptyset) - |\bigcup_{(i,x) \in R(\alpha)} A_{(i,x)}|$. By the inclusion-exclusion principle we have

$$\left| \bigcup_{(i,x) \in R(\alpha)} A_{(i,x)} \right| = \sum_{\emptyset \neq B \subseteq R(\alpha)} (-1)^{|B|+1} \left| \bigcap_{(i,x) \in B} A_{(i,x)} \right| = \sum_{\emptyset \neq B \subseteq R(\alpha)} (-1)^{|B|+1} Count(h, B),$$

which concludes the proof. □

Examples. Examples of the execution of the algorithm for a binary and a ternary string can be found in Figures 1 and 2, respectively.

Correctness. Let us start with an observation that lists some basic properties of our algorithm. Both parts can be shown by straightforward induction.

Observation 3. (a) *If a node v stores modified substrings $\alpha, \beta \in MS(v)$, then it has a descendant v' with $\mathbf{D}(v') = \mathsf{lcp}(val(\alpha), val(\beta))$ and $\alpha, \beta \in MS(v')$.*

(b) *Every node stores at most one modified substring originating from the same substring T_ℓ^m.*

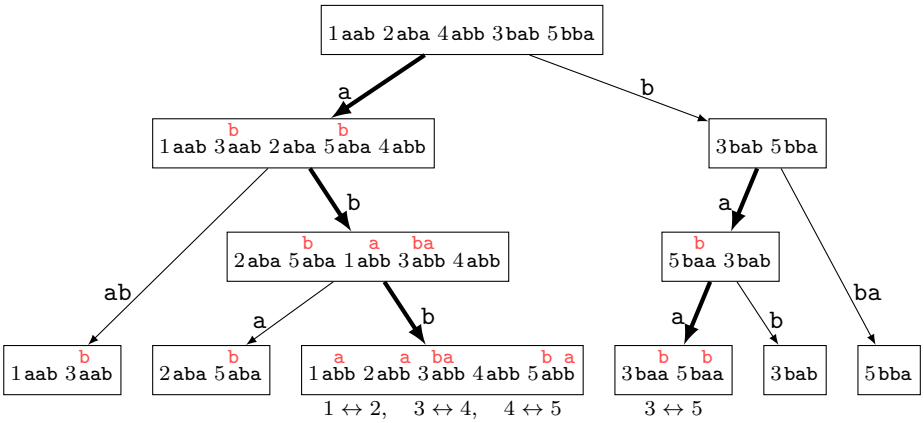

Figure 1: Computation of $(2,3)$-mappability for the string $T = \texttt{aababba}$ from Example 1. Note that the alphabet is binary in this case, so wildcard subtrees do not need to be introduced. Edges leading to heavy children are drawn in bold. The only substitutions are from a light child to a heavy child. The letters shown above are the original letters before the substitutions. The pairs of compatible modified substrings are indicated with arrows; in the end, $A^3_{=2}[1] = A^3_{=2}[2] = 1$ and $A^3_{=2}[3] = A^3_{=2}[4] = A^3_{=2}[5] = 2$ as expected.

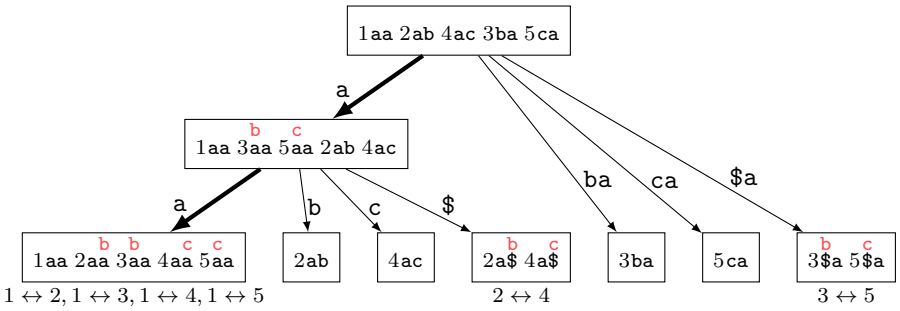

Figure 2: Computation of $(1,2)$-mappability for the string $T = \texttt{aabaca}$. This example illustrates the use of wildcard symbols. We have $A^2_{=1}[1] = 4$ and $A^2_{=1}[2] = A^2_{=1}[3] = A^2_{=1}[4] = A^2_{=1}[5] = 2$.

The following lemma shows that Algorithm 1 correctly computes the mappability table $\mathrm{A}^m_{=k}$.

Lemma 4. *If $d_H(T^m_i, T^m_j) = k$, then there is exactly one leaf v and exactly one pair of compatible modified substrings $\alpha, \beta \in MS(v)$ with $i = idx(\alpha)$ and $j = idx(\beta)$. Otherwise, there is no such leaf v and pair α, β.*

Proof. Suppose that $\alpha, \beta \in MS(v)$ are compatible, $i = idx(\alpha)$, and $j = idx(\beta)$. Since $W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset$, we conclude that T^m_i and T^m_j differ at positions of modifications in $W(\alpha) = W(\beta)$. They differ at positions of modifications in $H(\beta)$ since at the nodes corresponding to these positions, an ancestor of α (that is, the modified substring from which α originates) was in the heavy child and an ancestor of β originated from a light child (recall that (1) includes $H(\alpha) \cap H(\beta) = \emptyset$). Symmetrically, T^m_i and T^m_j differ at positions of modifications in $H(\alpha)$. In conclusion, they differ at positions of modifications in $H(\alpha) \cup H(\beta) \cup W(\alpha)$. The three sets are disjoint, so $|H(\alpha) \cup H(\beta) \cup W(\alpha)| = |H(\alpha)| + |H(\beta)| + |W(\alpha)| = k$ by (1). This shows that $d_H(T^m_i, T^m_j) \geq k$. With $val(\alpha) = val(\beta)$, we conclude that $d_H(T^m_i, T^m_j) = k$.

For a proof in the other direction, assume that $d_H(T^m_i, T^m_j) = k$ and let $1 \leq x_1 < x_2 < \cdots < x_k \leq m$ be the indices where the two substrings differ. Further let $x_{k+1} = m + 1$.

First of all, let us show that there is at least one leaf that contains compatible modified substrings α and

β with $idx(\alpha) = i$ and $idx(\beta) = j$.

Claim 5. *For every $p \in \{1, \ldots, k+1\}$, there exist a node v_p and modified substrings $\alpha_p, \beta_p \in MS(v_p)$ such that:*

- *$idx(\alpha_p) = i$ and $idx(\beta_p) = j$;*
- *$\mathsf{lcp}(val(\alpha_p), val(\beta_p)) = x_p - 1 = \mathbf{D}(v_p)$;*
- *for each position $x_1, \ldots, x_{p-1}$, both $M(\alpha_p)$ and $M(\beta_p)$ contain modifications of wildcard origin, or exactly one of these sets contains a modification of heavy origin;*
- *there are no other modifications in $M(\alpha_p)$ or $M(\beta_p)$.*

of Claim. The proof goes by induction on p. As α_1 and β_1, we take modified substrings such that $idx(\alpha_1) = i$, $idx(\beta_1) = j$, and $M(\alpha_1) = M(\beta_1) = \emptyset$. They are stored in the set $MS(r)$ for the root r, so Observation 3(a) guarantees the existence of a node v_1 with $\mathbf{D}(v_1) = \mathsf{lcp}(\alpha_1, \beta_1)$ and $\alpha_1, \beta_1 \in MS(v_1)$.

Let $p > 1$. By the inductive hypothesis, the set $MS(v_{p-1})$ contains modified substrings α_{p-1} and β_{p-1}. The node v_{p-1} has children w_1, w_2 corresponding to letters $T_i^m[x_{p-1}]$ and $T_j^m[x_{p-1}]$, respectively. If w_1 is the heavy child, then w_2 is a light child and a modified substring β' such that $idx(\beta') = j$ and $M(\beta') = M(\beta_{p-1}) \cup \{(x_{p-1}, T_i^m[x_{p-1}])\}$ is created for the recursive call in w_1. Then, we take $\alpha' = \alpha_{p-1}$. The case that w_2 is the heavy child is symmetric. Finally, if both w_1 and w_2 are light children, a child u of v_{p-1} is created along the wildcard symbol \$. There exist modified substrings $\alpha', \beta' \in MS(u)$ such that: $idx(\alpha') = i$, $idx(\beta') = j$, $M(\alpha') = M(\alpha_{p-1}) \cup \{(x_{p-1}, \$)\}$, and $M(\beta') = M(\beta_{p-1}) \cup \{(x_{p-1}, \$)\}$.

In either case, we have $\mathsf{lcp}(val(\alpha'), val(\beta')) = x_p - 1$. The set $(M(\alpha') \cup M(\beta')) \setminus (M(\alpha_{p-1}) \cup M(\beta_{p-1}))$ contains either a modification of heavy origin in one of the modified substrings or modifications of wildcard origin in both. Hence, by the inductive hypothesis, we can set $\alpha_p = \alpha'$ and $\beta_p = \beta'$. The node v_p with $\mathbf{D}(v_p) = \mathsf{lcp}(val(\alpha_p), val(\beta_p))$ and $\alpha_p, \beta_p \in MS(v_p)$ must exist due to Observation 3(a). □

Applied for $p = k+1$, the claim yields a leaf v_{k+1} that contains compatible modified substrings $\alpha = \alpha_{k+1}$ and $\beta = \beta_{k+1}$.

Now it suffices to check that there is no other pair of compatible modified substrings $(\alpha', \beta') \neq (\alpha, \beta)$ that would be present in some leaf u and satisfy $idx(\alpha') = i$ and $idx(\beta') = j$. Let us first note that $M(\alpha') \cup M(\beta')$ must contain modifications at positions $x_1, \ldots, x_k$ (since $val(\alpha') = val(\beta')$) and no modifications at other positions (otherwise, $|H(\alpha')| + |H(\beta')| + |W(\alpha')|$ would exceed k). Let p be the greatest index in $\{1, \ldots, k+1\}$ such that $x_p - 1 \leq \mathsf{lcp}(val(\alpha), val(\alpha'))$. By Observation 3(b), $u \neq v_{k+1}$, so $p \leq k$.

Thus, the node v_p is an ancestor of the leaf u, but the node v_{p+1} is not. Let us consider the children w_1, w_2 of v_p obtained by following edges with labels $T_i^m[x_p]$ and $T_j^m[x_p]$, respectively. If w_1 is the heavy child, β' must contain a modification of heavy origin at position x_p, so v_{p+1} is an ancestor of u; a contradiction. The same contradiction is obtained in the symmetric case that w_2 is the heavy child. Finally, if both w_1 and w_2 are light, then either both α' and β' contain a modification of wildcard origin at position x_p, which again gives a contradiction, or they both contain a modification of heavy origin, which contradicts the first part of condition (1). □

Remark 6. The recursive approach presented above is somewhat similar to the scheme used by Thankachan et al. [32] for computing the longest common substring with up to k mismatches of two strings. We attempted to adapt the approach of [32] to computing k-mappability, but failed due to multiple counting of substring pairs, e.g., for $T = \texttt{aabbab}$, $k = 2$, $m = 3$. Another virtue of our approach is that we obtain time complexity better by a factor of $k!$ for super-constant k.

Implementation and complexity. Our Algorithm 1, excluding the counting phase in the leaves, has exactly the same structure as Algorithm 1 in [9]. Proposition 13 from [9] provides a bound on the total number of the generated modified strings and an efficient implementation based on finger-search trees. We apply that proposition for a family $\mathbf{F}$ composed of substrings T_i^m to obtain the following bounds.

Fact 7 (see [9, Proposition 13]). *Algorithm 1 applied up to the leaves takes* $\mathcal{O}(n\binom{\log n+k+1}{k+1}2^k)$ *time and generates* $\mathcal{O}(n\binom{\log n+k}{k}2^k)$ *modified substrings.*

Let us further analyze the space complexity of the algorithm.

Lemma 8. *Algorithm 1 applied up to the leaves uses* $\mathcal{O}(nk)$ *working space.*

Proof. We assume that, upon termination, the procedure `processNode` discards the set $MS(v)$ and all the modified strings created during its execution. This way, the whole memory allocated within a given call to `processNode` is freed. Since `processNode` returns no output and its only side effects are updates of the array $A^k_=$, no information is lost through such garbage collection.

A call to `processNode`(v) for node v partitions the list $MS(v)$ into sublists corresponding to $u_1, \ldots, u_a$, creates $2(|MS(u_2)| + \cdots + |MS(u_a)|)$ new modified substrings (each requiring constant space to be stored), appends them to sublists corresponding to u_1 and u_{a+1}, and then recurses on the sublists. In particular, the elements of the original list $MS(v)$ are not copied but reused in the recursive call. The following observation provides further characterization of these elements:

Observation 9. *If a node v is a child of w, then every element of $MS(v)$ is either an element of $MS(w)$ or a modified substring originating from an element of $MS(w)$.*

Let us consider a root-to-leaf path ρ in the recursion. Each recursive call uses $\mathcal{O}(1)$ local variables, which take $\mathcal{O}(n)$ space overall. We also need to bound the total number of modified substrings created by calls to `processNode` for nodes on the path ρ.

By Observations 9 and 3(b), $|MS(v)|$ is non-increasing on ρ. Moreover, if v is a light child of its parent w, then $|MS(v)| \le |MS(w)|/2$. Let us consider all nodes w on ρ such that the unique child of w that is on ρ is a light child. The total number of modified strings created by the calls to `processNode`(w) for all such nodes w is $\mathcal{O}(n)$ since we can upper bound it by a geometric series that sums to $\mathcal{O}(n)$.

As for the calls to `processNode`(w) for the remaining nodes on ρ, for every two modified strings they create, they put one of them in the child of w that also belongs to ρ. Hence, it suffices to upper bound the total number of modified substrings originating from T^m_i for each position i that are in $MS(v)$ for some node v on ρ. For a given position i, let $\alpha_1, \ldots, \alpha_b$ be all such modified substrings originating from T^m_i. By Observation 9, we have $M(\alpha_1) \subsetneq M(\alpha_2) \subsetneq \cdots \subsetneq M(\alpha_b)$ and thus $b \le k$. In total, we create $\mathcal{O}(nk)$ modified substrings in calls to `processNode` on nodes of ρ. □

Next, we show how to improve the time complexity of Algorithm 1 by a relatively small change in its execution. Intuitively, we will take advantage of the fact that the modified substrings in a leaf of the recursion do not need to be sorted lexicographically.

Namely, whenever a modified substring β with exactly k modifications is created at a node v (i.e., $|M(\alpha)| = k - 1$ in the if-statement), we do not include β in the recursive call of wildcardTree or heavyChild. Instead, an entry $(val(\beta), \beta)$ is inserted into a global hash table. When processing a leaf v containing modified substrings with a common value $val(\alpha)$, we need to move all modified substrings with value $val(\alpha)$ from the global hash table to the set $MS(v)$. Finally, if any modified string β created while processing a given node v remains in the hash table upon completion of `processNode`(v), then β is removed from the hash table together with all other modified substrings with the value $val(\beta)$. At this moment, an artificial leaf of the recursion containing all these modified substrings is created and the standard routine is applied to process this leaf.

Recall that the hash table uses Karp–Rabin fingerprints to index strings and collisions could incur incorrect results in the algorithm. To tackle this issue, whenever a modified substring β is inserted to the hash table and there is another modified substring with the same hash in the table, we pick any one such modified substring α and check if $val(\alpha) = val(\beta)$ in $\mathcal{O}(k)$ time using lce queries on T with a method that resembles kangaroo jumping [17, 27] (it requires $\mathcal{O}(n)$-time preprocessing). By Lemma 8, the hash table contains up to $\mathcal{O}(nk)$ entries at any given time, so the collision probability is $\mathcal{O}(nk \cdot n^{-C}) = \mathcal{O}(n^{-C+2})$. Setting $C > c + 2$, we can make sure that this is dominated by the probability that the hash table fails to process the underlying insertion in $\mathcal{O}(1)$ time.

Let us call the resulting algorithm Algorithm 1'.

Lemma 10. *The outputs of Algorithms 1 and 1' are the same. Moreover, Algorithm 1' works in time* $\mathcal{O}(n\binom{\log n+k}{k}2^k k)$ *with high probability (up to the leaves) and uses the same amount of space as Algorithm 1.*

Proof. Let v be a leaf in the recursion of Algorithm 1. If $MS(v)$ contains at least one modified substring with up to $k-1$ modifications, v will be identified by the recursive procedure of Algorithm 1'. Then, all modified substrings with exactly k modifications that belong to v are populated from the global hash table. If $MS(v)$ does not contain any modified substring with less than k modifications, v will be identified upon a deletion from the global hash map at the lowest internal node u of the recursion in which a modified substring belonging to $MS(v)$ was created. Here, we use the fact that the path-labels $\mathcal{L}(u)$ of all nodes u of the recursion are different. This shows that indeed the leaves of the recursion of Algorithms 1 and 1' are the same.

As for the time complexity, the total number of modified substrings created by Algorithm 1' is the same as in Algorithm 1, i.e., $\mathcal{O}(n\binom{\log n+k}{k}2^k)$ by Fact 7. However, the time necessary to conduct the whole recursive procedure corresponds to the time complexity of Algorithm 1 that is run with $k-1$ instead of k, i.e., also $\mathcal{O}(n\binom{\log n+k}{k}2^k)$ by Fact 7. After $\mathcal{O}(n)$-time preprocessing, for each modified substring, we can compute its Karp–Rabin fingerprint and check collisions in $\mathcal{O}(k)$ time; this accounts for the additional factor k in the time complexity.

Finally, the space complexity stays the same because modified substrings with exactly k modifications are removed from the hash table at latest when the recursion rolls back. □

Lemmas 8 and 10 yield the complexity of Algorithm 1'. Note that, due to the application of the inclusion-exclusion principle in the leaves, we need to multiply the time complexity of the algorithm by 2^k and increase the space complexity by $\mathcal{O}(n2^k k)$.

Theorem 11. *There exists a Las-Vegas randomized algorithm that computes the* (k,m)*-mappability of a given length-*n *string in* $\mathcal{O}(n2^k k)$ *space and, with high probability, in* $\mathcal{O}(n\binom{\log n+k}{k}4^k k)$ *time. For* $k=\mathcal{O}(1)$*, the space is* $\mathcal{O}(n)$ *and the time becomes* $\mathcal{O}(n\log^k n)$*.*

4 All-Pairs Hamming Distance Problem

We will show how the previous algorithm can be modified to solve the all-pairs Hamming distance problem, at the cost of an additional $\log r$-factor in the complexity. We run the algorithm from the previous section for T being a concatenation of all the strings in $\mathbf{R}$ and only with substrings $\{T_i^m : i \bmod m = 1\}$ in the root. The algorithm needs to be updated only at the leaves of the compact trie. Henceforth, let us consider a trie leaf v with a set $MS(v)=\{\beta_1,\ldots,\beta_p\}$ of modified substrings. We will further denote this set as MS ($|MS|=p$). Our goal is to list, for every $\beta\in MS$, all $\beta'\in MS$ that are compatible with β.

Let us construct a static balanced binary search tree (BST) in which the leaves correspond to the modified substrings β_i. In this way, each node of the BST corresponds to a set of subsequent candidates from the leaves of its subtree. If $\beta_i,\ldots,\beta_j$ are the modified substrings in the leaves of the subtree of a BST node u, then we denote $set(u)=\{\beta_i,\ldots,\beta_j\}$. A leaf will be responsible for storing information only for itself and an internal node stores merged information of its children.

Our goal is to store information in each node u of the BST in such a way that for any modified substring $\alpha\in MS$ we will be able to answer if there is any other candidate in $set(u)$ that is compatible with α. Therefore, in each node u, we will compute all the required machinery for using the inclusion-exclusion principle on the modified substrings in $set(u)$, that is, a hashmap that stores all non-zero values of $Count(s,B)$ for modified substrings $\beta\in set(u)$. Since every $\beta\in MS$ is present in $\mathcal{O}(\log p)$ sets $set(u)$, precomputing all mentioned information can be done in $O(2^k kp\log p)$ time and space.

Our query algorithm for a given modified substring β is a recursive procedure starting at the root of the BST. Assume that the algorithm is at some BST node u. We use Lemma 2 and the hashmap for $set(u)$ to count the elements $\beta'\in set(u)$ that are compatible with β. If this number is positive, the algorithm recursively descends to the children of node u. In the end, modified substrings β' that are compatible with β will be listed at the leaves of the BST. The correctness of this algorithm follows from Lemma 4.

Every application of Lemma 2 takes $\mathcal{O}(2^k k)$ time. For each modified substring β' that is compatible with a modified substring β, the algorithm will visit $\mathcal{O}(\log p)$ BST nodes, which gives $\mathcal{O}(2^k k \log p)$ time for finding each compatible modified substring $\beta' \in MS$. Note that $p \leq r$ (see Observation 3(b)). Summing up over all trie nodes v and applying Lemmas 10 and 8, we obtain the following result. (Observe that [9, Proposition 13] is applied for a family $\mathbf{F}$ of size r rather than n.)

Theorem 12. *There exists a Las-Vegas randomized algorithm that, given a set of r length-m strings and an integer k, solves the all-pairs Hamming distance problem in $\mathcal{O}(rm + 2^k kr \log r)$ space and, with high probability, in $\mathcal{O}(rm + r\binom{\log r + k}{k} 4^k k \log r + \mathsf{output} \cdot 2^k k \log r)$ time. For $k = \mathcal{O}(1)$, the space is $\mathcal{O}(rm + r \log r)$ and the time becomes $\mathcal{O}(rm + r \log^{k+1} r + \mathsf{output} \cdot \log r)$.*

5 Computing Mappability in $\mathcal{O}(nm^k)$ Time and $\mathcal{O}(n)$ Space

In this section, we generalize the $\mathcal{O}(nm)$-time algorithm for $k = 1$ and integer alphabets from [3]. To this end, we make use of an approach from [6]. The high-level idea from [6] is to define a lexicographic order on the suffixes of T that ignores the same k fixed positions of every suffix. (In fact, the algorithm does the same for many such combinations of k positions.) It then uses the suffix tree of T to sort the modified suffixes according to this new lexicographic order. The focus of this algorithm is not on counting substrings that are at Hamming distance at most k, and so we adapt it with some extra care to avoid multiple counting.

We first generate all $\binom{m}{\leq k}$ subsets of $\{1, \ldots, m\}$ of size at most k. For each such subset F, we consider the length-m substrings of T with their f-th letter substituted with $\# \notin \Sigma$ for all $f \in F$. We sort each of these sets of strings in $\mathcal{O}(nk\binom{m}{\leq k})$ total time using the approach of [6], also obtaining the maximal blocks of equal strings in the sorted list.

We now briefly describe the algorithm for sorting one such set of strings in time $\mathcal{O}(nk)$ for the sake of completeness. Let us assume for simplicity that $F = \{f\}$ as the algorithm can be generalized trivially for larger sets. We first retrieve the sorted list of T_i^{f-1} for all i from the suffix tree. We then give ranks to these strings after we check equality of adjacent strings in the sorted list using lce queries. We similarly rank strings T_j^{m-f} for all j. Finally, we sort the ranks of the pairs $(T_i^{f-1}, T_{i+f+1}^{m-f})$ using bucket sort.

Prior to running the above algorithm, we initialize arrays D_K for $K \in \{1, \ldots, k\}$. For each maximal block, of size b, of equal strings obtained for some set F, we increment the b relevant entries of $D_{|F|}$ by $b-1$.

Note that if $d_H(T_i^m, T_j^m) = \kappa$, then this will contribute $\binom{m-\kappa}{K-\kappa}$ to each of $D_K[i]$ and $D_K[j]$ for $K \geq \kappa$, since there are these many size-K supersets of the set of mismatching positions in the power set of $\{1, \ldots, m\}$. We thus compute $\mathrm{A}^m_{=K}[i] = D_K[i] - \sum_{\kappa=0}^{K-1} \binom{m-\kappa}{K-\kappa} \mathrm{A}_{=\kappa}[i]$ in increasing order with respect to K and we are done. (We precompute all relevant binomial coefficients in $\mathcal{O}(k^2)$ time.)

Theorem 13. *Given a string of length n, the (k, m)-mappability problem can be solved in $\mathcal{O}(nk\binom{m}{\leq k})$ time and $\mathcal{O}(n)$ space. For $k = \mathcal{O}(1)$, the time becomes $\mathcal{O}(nm^k)$.*

Combining Theorems 11 and 13 gives the following result.

Corollary 14. *For every $k = \mathcal{O}(1)$, there exists a randomized algorithm that computes the (k, m)-mappability of a given length-n string in $\mathcal{O}(n)$ space and in $\mathcal{O}(n \cdot \min\{m^k, \log^k n\})$ time with high probability.*

6 Computing (k, m)-Mappability for All k or for All m

Theorem 15. *The (k, m)-mappability for a given m and all $k \in \{0, \ldots, m\}$ can be computed in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n)$ space.*

Proof. We first present an algorithm which solves the problem in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n^2)$ space and then show how to reduce the space usage to $\mathcal{O}(n)$.

We initialize an $n \times n$ matrix M in which $M[i,j]$ will store the Hamming distance between substrings T_i^m and T_j^m. Let us consider two letters $T[i] \neq T[j]$ of the input string, where $i < j$. Such a pair contributes to a mismatch between the following pairs of strings:

$$(T_{i-m+1}^m, T_{j-m+1}^m), (T_{i-m+2}^m, T_{j-m+2}^m), \ldots, (T_i^m, T_j^m).$$

This list of strings is represented by a diagonal interval in M, the entries of which we need to increment by 1. We process all $\mathcal{O}(n^2)$ pairs of letters and update the information on the respective intervals. Then $\mathrm{A}_{=k}^m[i] = |\{j : M[i,j] = k\}|$.

To achieve $\mathcal{O}(1)$ time for each single addition on a diagonal interval, we use a well-known trick from an analogous problem in one dimension. Suppose that we would like to add 1 on the diagonal interval from $M[x_1, y_1]$ to $M[x_2, y_2]$. Instead, we can simply add 1 to $M[x_1, y_1]$ and -1 to $M[x_2 + 1, y_2 + 1]$. Every cell will then represent the difference of its actual value to the actual value of its predecessor on the diagonal. After all such operations are performed, we can retrieve the actual values by computing prefix sums on each diagonal in a top-down manner.

To reduce the space usage to $\mathcal{O}(n)$, it suffices to observe that the value of $M[i,j]$ depends only on the value of $M[i-1, j-1]$ and at most two letter comparisons which can add $+1$ and/or -1 to the cell. Recall that $M[i,j] = d_H(T_i^m, T_j^m)$. We need to subtract 1 from the previous result if the first characters of the previous substrings were equal and add 1 if the last characters of the new substrings were different. Therefore, we can process the matrix row by row, from top to bottom, and compute the values $\mathrm{A}_{=0}^m[i], \ldots, \mathrm{A}_{=m}^m[i]$ while processing the ith row. □

Theorem 16. *The (k,m)-mappability for a given k and all $m \in \{k, \ldots, n\}$ can be computed in $\mathcal{O}(n^2)$ time and space.*

Proof. We first prove the following claim.

Claim 17. *The longest common prefixes with k mismatches for all pairs of suffixes of T can be computed in $\mathcal{O}(n^2)$ time.*

of Claim. We process the pairs in batches B_δ for $\delta \in \{1, 2, \ldots, n\}$ so that the pair (T_i, T_j), which we denote by (i,j), is in $B_{|j-i|}$. It now suffices to show how to process a single batch B_δ in $\mathcal{O}(n)$ time. We will do so by comparing pairs of letters of T at distance δ from left to right. We first compute $\mathsf{lce}_k(1, 1+\delta)$ naively. Then, given that $\mathsf{lce}_k(i,j) = \ell$, where $j - i = \delta$, we will retrieve $\mathsf{lce}_k(i+1, j+1)$ using the following simple observation: either $j+\ell-1 = n$, or T_i^ℓ and T_j^ℓ have exactly k mismatches and $T[i+\ell] \neq T[j+\ell]$. In the former case, we trivially have that $\mathsf{lce}_k(i+1, j+1) = \ell - 1$. In the latter case, we first check whether $T[i] = T[j]$, in which case $d_H(T_{i+1}^{\ell-1}, T_{j+1}^{\ell-1}) = k$ and hence $\mathsf{lce}_k(i+1, j+1) = \ell-1$. If $T[i] \neq T[j]$, then $d_H(T_{i+1}^{\ell-1}, T_{j+1}^{\ell-1}) = k-1$ and we perform letter comparisons to extend the match. The pairs of letters compared in this step have not been compared before; the complexity follows. □

We store the information on lce_k's as follows. We initialize an $n \times n$ matrix Q. Then, for a pair (i,j) such that $\mathsf{lce}_k(i,j) = \ell$, we increment by 1 the entries $Q[\ell, i]$ and $Q[\ell, j]$. Note that if $\mathsf{lce}_k(i,j) = \ell$, then i (resp. j) will contribute 1 to the (k,m)-mappability values $\mathrm{A}_{\leq k}^m[j]$ (resp. $\mathrm{A}_{\leq k}^m[i]$) for all $m \in \{k, \ldots, \ell\}$. Thus, starting from the last row of Q, we iteratively add row ℓ to row $\ell - 1$. In the end, by the above observation, row m stores the (k,m)-mappability array $\mathrm{A}_{\leq k}^m$. □

7 Conditional Hardness for $k, m = \Theta(\log n)$

We will show that (k,m)-mappability cannot be computed in strongly subquadratic time in case that the parameters are $\Theta(\log n)$, unless the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi and Zane [22, 21] fails. Our proof is based on the conditional hardness of the following decision version of the Longest Common Substring with k Mismatches problem.

Common Substring of Length d with k Mismatches
Input: Strings T_1, T_2 of length n over binary alphabet and integers k, d.
Output: Is there a factor of T_1 of length d that occurs in T_2 with k mismatches?

Lemma 18 ([26])**.** *Suppose there is $\varepsilon > 0$ such that Common Substring of Length d with k Mismatches can be solved in $\mathcal{O}(n^{2-\varepsilon})$ time on strings over binary alphabet for $k = \Theta(\log n)$ and $d = 21k$. Then SETH is false.*

Theorem 19. *If the (k, m)-mappability can be computed in $\mathcal{O}(n^{2-\varepsilon})$ time for binary strings, $k, m = \Theta(\log n)$, and some $\varepsilon > 0$, then SETH is false.*

Proof. We make a Turing reduction from Common Substring of Length d with k Mismatches. Let T_1 and T_2 be the input to the problem. We compute the (k, d)-mappabilities of strings $T_1 \cdot T_2$ and $T_1 \cdot T_2[1\,..\,d-1]$ and store them in arrays A and B, respectively. For each $i \in \{1, \ldots, n-d+1\}$, we subtract $B[i]$ from $A[i]$. Then, $A[i]$ holds the number of factors of T_2 of length d that are at Hamming distance k from $T_1[i\,..\,i+d-1]$. Hence, Common Substring of Length d with k Mismatches has a positive answer if and only if $A[i] > 0$ for any $i \in \{1, \ldots, n-d+1\}$.

By Lemma 18, an $\mathcal{O}(n^{2-\varepsilon})$-time algorithm for Common Substring of Length d with k Mismatches with $k = \Theta(\log n)$ and $d = 21k$ would refute SETH. By the shown reduction, an $\mathcal{O}(n^{2-\varepsilon})$-time algorithm for (k, m)-mappability with $k, m = \Theta(\log n)$ would also refute SETH. □

8 Final Remarks

Our main contribution is an $\mathcal{O}(n \cdot \min\{m^k, \log^k n\})$-time $\mathcal{O}(n)$-space algorithm for solving the (k, m)-mappability problem. Let us recall that genome mappability, as introduced in [12], counts the number of substrings that are at Hamming distance at most k from every length-m substring of the text. One may also be interested to consider mappability under the edit distance model. This question relates also to recent contributions on computing approximate longest common prefixes and substrings under edit distance [31, 6]. In the case of the edit distance, in particular, a decision needs to be made whether sufficiently similar substrings only of length exactly m or of all lengths between $m-k$ and $m+k$ should be counted. We leave the mappability problem under edit distance for future investigation.

References

[1] Hayam Alamro, Lorraine A. K. Ayad, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with k-mismatches and applications. In *Current Trends in Theory and Practice of Computer Science, SOFSEM 2018*, volume 10706 of *LNCS*, pages 636–649. Springer, 2018. URL: `https://doi.org/10.1007/978-3-319-73117-9_45`.

[2] Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, and Juliusz Straszyński. Efficient computation of sequence mappability. In *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018*, volume 11147 of *LNCS*, pages 12–26. Springer, 2018. URL: `https://doi.org/10.1007/978-3-030-00479-8_2`.

[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. *Theoretical Computer Science*, 812:2–12, 2020. URL: `https://doi.org/10.1016/j.tcs.2019.04.026`.

[4] Amihood Amir, Itai Boneh, and Eitan Kondratovsky. The k-mappability problem revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[5] Pavlos Antoniou, Jackie 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 *Information Technology and Applications in Biomedicine, ITAB 2009*. IEEE, 2009. URL: https://doi.org/10.1109/itab.2009.5394394.

[6] Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with k-errors and applications. In *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018*, volume 11147 of *LNCS*, pages 27–41. Springer, 2018. URL: https://doi.org/10.1007/978-3-030-00479-8_3.

[7] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. In *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium*, volume 2286 of *LNCS*, pages 508–515. Springer, 2002. URL: https://doi.org/10.1007/3-540-45995-2_44.

[8] João A. Carriço, Maxime Crochemore, Alexandre P. Francisco, Solon P. Pissis, Bruno Ribeiro-Gonçalves, and Cátia Vaz. Fast phylogenetic inference from typing data. *Algorithms for Molecular Biology*, 13(1):4, Feb 2018. URL: https://doi.org/10.1186/s13015-017-0119-7.

[9] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In *Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: https://doi.org/10.4230/LIPIcs.CPM.2018.23.

[10] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pages 91–100. ACM, 2004. URL: https://doi.org/10.1145/1007352.1007374.

[11] Maxime Crochemore, Alexandre P. Francisco, Solon P. Pissis, and Cátia Vaz. Towards Distance-Based Phylogenetic Inference in Average-Case Linear-Time. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. URL: https://doi.org/10.4230/LIPIcs.WABI.2017.9.

[12] Thomas Derrien, Jordi Estellé, Santiago Marco Sola, David G. Knowles, Emanuele Raineri, Roderic Guigó, and Paolo Ribeca. Fast computation and applications of genome mappability. *PLoS ONE*, 7(1):e30377, 2012. URL: https://doi.org/10.1371/journal.pone.0030377.

[13] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Automata, Languages and Programming, 17th International Colloquium, ICALP 1990*, volume 443 of *LNCS*, pages 6–19. Springer, 1990. URL: https://doi.org/10.1007/BFb0032018.

[14] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143. IEEE Computer Society, 1997. URL: https://doi.org/10.1109/SFCS.1997.646102.

[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. URL: https://doi.org/10.1093/bioinformatics/bts605.

[16] Alexandre P. Francisco, Miguel Bugalho, Mário Ramirez, and João A. Carriço. Global optimal eBURST analysis of multilocus typing data using a graphic matroid approach. *BMC Bioinformatics*, 10(1):152, May 2009. URL: https://doi.org/10.1186/1471-2105-10-152.

[17] Zvi Galil and Raffaele Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science*, 51:341–348, 1987. URL: https://doi.org/10.1016/0304-3975(87)90042-9.

[18] Simon Gog and Rossano Venturini. Fast and compact Hamming distance index. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2016*, pages 285–294. ACM, 2016. URL: https://doi.org/10.1145/2911451.2911523.

[19] Szymon Grabowski and Tomasz M. Kowalski. Algorithms for all-pairs hamming distance based similarity. *Software: Practice and Experience*, 2021. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2978.

[20] Sahar Hooshmand, Paniz Abedin, Daniel Gibney, Srinivas Aluru, and Sharma V. Thankachan. Faster computation of genome mappability with one mismatch. In *8th IEEE International Conference on Computational Advances in Bio and Medical Sciences, ICCABS 2018*, page 1. IEEE Computer Society, 2018. URL: https://doi.org/10.1109/ICCABS.2018.8541897.

[21] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. URL: https://doi.org/10.1006/jcss.2000.1727.

[22] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. URL: https://doi.org/10.1006/jcss.2001.1774.

[23] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. URL: https://doi.org/10.1145/1217856.1217858.

[24] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. URL: https://doi.org/10.1147/rd.312.0249.

[25] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, CPM 2001*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. URL: https://doi.org/10.1007/3-540-48194-X_17.

[26] Tomasz Kociumaka, Jakub Radoszewski, and Tatiana A. Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. URL: https://doi.org/10.1007/s00453-019-00548-x.

[27] Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. URL: https://doi.org/10.1016/0304-3975(86)90178-7.

[28] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. URL: https://doi.org/10.1137/0222058.

[29] Giovanni Manzini. Longest common prefix with mismatches. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval, SPIRE 2015*, volume 9309 of *LNCS*, pages 299–310. Springer, 2015. URL: https://doi.org/10.1007/978-3-319-23826-5_29.

[30] Veli Mäkinen and Tuukka Norri. Applying the positional Burrows–Wheeler transform to all-pairs Hamming distance. *Information Processing Letters*, 146:17–19, 2019. URL: https://doi.org/10.1016/j.ipl.2019.02.003.

[31] Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In *Research in Computational Molecular Biology, RECOMB 2018*, volume 10812 of *LNCS*, pages 211–224. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-89929-9_14.

[32] 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. URL: https://doi.org/10.1089/cmb.2015.0235.