Algorithmica
https://doi.org/10.1007/s00453-022-00934-y

# Efficient Computation of Sequence Mappability

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

Received: 29 March 2021 / Accepted: 10 December 2021

**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 $i$th 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 $k = O(1)$, works in $O(n)$ space and, with high probability, in $O(n \cdot \min\{m^k, \log^k n\})$ 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 $O(n^2)$-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 presented at SPIRE 2018.

**Keywords** Sequence mappability · $k$-errata tree · Hamming distance

This paper is part of the PANGAIA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 872539. This paper is also part of the ALPACA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 956229.

✉ Juliusz Straszyński
jks@mimuw.edu.pl

Extended author information available on the last page of the article

Published online: 02 February 2022

## 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 [16].

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.

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 $\mathrm{A}_{\le k}^m$ whose $i$th entry $\mathrm{A}_{\le k}^m[i]$ 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 $k \le 4$, $m \le 100$, and the alphabet of $T$ was $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

***Example 1.1*** 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** | $\mathrm{A}_{\le 1}^3[i]$ | 2 | 2 | 1 | 2 | 1 |
| $(2, 3)$-**mappability** | $\mathrm{A}_{\le 2}^3[i]$ | 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 $i$th 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 $\mathrm{A}_{\le k}^m[i] = \sum_{\kappa=0}^{k} \mathrm{A}_{=\kappa}^m[i]$; see the "difference" row in the example above. Henceforth, we call this problem *the* $(k, m)$*-mappability problem.*

Using the suffix array and the LCP table [24,26,30], the $(0, m)$-mappability problem can be solved in $O(n)$ time and space. Known solutions for computing $(1, m)$-mappability are shown in Table 1; the $O(nm)$-time and the $O(n)$-average-time

**Table 1** Known algorithms for computing $(1, m)$-mappability for strings over constant-sized alphabets

| Solution | Time complexity |
|---|---|
| Manzini [31] | $O(mn \log n / \log\log n)$ |
| Alzamel et al. [3] | $O(nm)$ |
| Alzamel et al. [3] | $O(n \log n \log\log n)$ |
| Alzamel et al. [3] | $O(n)$ on average for $m = \Omega(\log n)$ |
| Hooshmand et al. [21], Amir et al. [4] | $O(n \log n)$ |
| Amir et al. [4] | $O(n)$ for $m = \Omega(\sqrt{n})$ |

All algorithms use $O(n)$ space

solutions of Alzamel et al. [3] work also on strings over *integer alphabets* $\{1, \ldots, \sigma\}$ for $\sigma = n^{O(1)}$. Moreover, the latter algorithm was shown to be generalizable to arbitrary $k$, requiring $O(n)$ space and, on average, $O(kn)$ time if $m = \Omega(k \log_\sigma n)$. A practically fast algorithm for arbitrary $k$ was presented in [32]. 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 $\mu$ 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 [35]. 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 [11,17], 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 $\mathbf{R}$ [11,20,29]. In particular, these algorithms work in $O(rm)$ average-case time if $k < \frac{(m-k-1)\log\sigma}{\log rm}$ and the elements of $\mathbf{R}$ are over an integer alphabet $\Sigma$ of size $\sigma > 1$ with the letters of the strings being independent and identically distributed random variables uniformly distributed over $\Sigma$. 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 [19].

Intuitively, there is a connection between the $(k, m)$-mappability problem and the all-pairs Hamming distance problem that allows to transfer the technique used in the solution to the former to a solution to the latter (it is not a formal reduction between problems). The connection 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 Sect. 3, we show a randomized Las-Vegas algorithm for the $(k, m)$-mappability problem that works in $O(n\binom{\log n+k}{k}4^k k)$ time with high probability[1] and $O(n2^k k)$ space for a string over any ordered alphabet. It requires a careful adaptation of the technique of recursive heavy-path decompositions in a tree [10].
2. In Sect. 4, we show an algorithm to solve the all-pairs Hamming distance problem for strings over any ordered alphabet that works in $O(rm + r\binom{\log r+k}{k}4^k k \log r + \mathsf{output} \cdot 2^k k \log r)$ time and $O(rm + r2^k k \log r)$ space.
3. In Sect. 5, we show an algorithm for the $(k, m)$-mappability problem that works in $O(nk \cdot (m + 1)^k)$ time and $O(n)$ space for a string over an integer alphabet. Together with the first result, this yields an $O(n \cdot \min\{m^k, \log^k n\})$-time and $O(n)$-space algorithm for $k = O(1)$.
4. In Sect. 6, we show $O(n^2)$-time algorithms for a string over any ordered alphabet 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 Sect. 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 [22,23] fails.

In contributions 1 and 5, we apply recent advances in the Longest Common Substring with $k$ Mismatches problem that were presented in [9,27], respectively (see also [34]). 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. Contributions 1, 2, and 4 apply to strings over an arbitrary ordered alphabet; the running times of the respective algorithms are $\Omega(n \log n)$, which is sufficient to renumber the letters of the input text so that its alphabet becomes an integer alphabet.

This work is an extended version of [2]. In comparison to the conference version, in particular, 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).

[1] With probability at least $1 - n^{-c}$ for an arbitrarily large predefined constant parameter $c > 0$.

## 2 Preliminaries

Let $T = T[1]T[2]\cdots T[n]$ be a *string* of length $|T| = n$ over a finite ordered alphabet $\Sigma$ of size $|\Sigma| = \sigma$. The empty string is denoted by $\varepsilon$. In some algorithms we assume that the string is over an *integer alphabet*, i.e., $\Sigma = \{1, \dots, n^{O(1)}\}$. 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 $d_H(S, T) = |\{i \in \{1, 2, \dots, |S|\} : S[i] \neq T[i]\}|$. If $|S| \neq |T|$, we set $d_H(S, T) = \infty$.

By $\mathsf{lcp}(U, V)$ we denote the length of the longest common prefix of strings $U$ and $V$. For a fixed string $T$, we also set $\mathsf{lcp}(r, s) = \mathsf{lcp}(T_r, T_s)$.

*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 $\varepsilon$; 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, whereas 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 $\mathbf{L}(v)$ 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, $\mathbf{D}(v) = |\mathbf{L}(v)|$ 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 $O(n)$ time [14] and, after an $O(n)$-time preprocessing [7], it can be used to answer $\mathsf{lcp}(r, s)$ queries in $O(1)$ time.

*Hashing.* We use perfect hashing to implement dynamic dictionaries supporting insertions and deletions of entries (key-value pairs), as well as look-ups of entries with a given key. Technically, we maintain a single global dictionary (which may simulate multiple local dictionaries) implemented using the following result originating from the work of Dietzfelbinger and Meyer auf der Heide [13].

**Theorem 2.1** (see [13, Theorem 5.5]) *For any constant $c > 1$ and positive integer $n$, there is a data structure that maintains a dynamic dictionary $\mathbf{D}$ of size $|\mathbf{D}| \leq n$ with the following guarantees:*

1. *The data structure occupies $O(n)$ space.*

1. *Handling any $m \leq n^c$ operations (look-ups, insertions, and deletions) in an on-line fashion costs $O(n+m)$ time in total*[2] *with probability at least $1-n^{-c}$.*

*The constants in the time and space bounds depend on c.*

The original data structure of [13, Theorem5.5] supports $n$ operations in $O(n)$ time with probability at least $1-n^{-c}$. To allow $m \leq n^c$ operations, we use instances supporting $2n$ operations in $O(n)$ time with probability at least $1-n^{1-2c}$, and we build such an instance from scratch after completing every $n$ operations (using $|\mathbf{D}| \leq n$ insertions out of the allowance of $2n$ operations). By the union bound, all $m \leq n^c$ operations are thus handled in $O(n+m)$ total time with probability at least $1-n^{-c}$.

When using strings as dictionary keys, we rely on Karp–Rabin fingerprints (polynomial hashing) [25] 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 $O(n \log^k n)$ Time and $O(n)$ Space

Our algorithm operates on so-called *modified strings*. A modified string $\alpha$ is a pair $(U(\alpha), M(\alpha))$, where $U(\alpha)$ is a string and $M(\alpha)$ a set of modifications. Each element of the set $M(\alpha)$ is a pair of the form $(i, c)$ which denotes a substitution "$U(\alpha)[i] := c$". We assume that no two pairs in $M(\alpha)$ share the same index $i$. By $val(\alpha)$, we denote the string $U(\alpha)$ after all the substitutions. The sets $M(\alpha)$ for modified strings are implemented as (functional) lists. Whenever a modified string $\beta$ is obtained by introducing an extra modification to a modified string $\alpha$, 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 $T_i^m$. In this case, the strings $U(\alpha)$ are not stored explicitly. Instead, for a modified substring $\alpha$ originating from $T_i^m$, an index $idx(\alpha) = i$ is stored.

*Overview of the algorithm* Intuitively, the algorithm proceeds by efficiently simulating transformations of a compact trie of modified substrings, initially containing all substrings $T_i^m$.[3] 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.

[2] Through standard deamortization, we could achieve, with high probability, $O(1)$-time operations after $O(n)$-time initialization. However, this would not benefit the main results of this paper.

[3] 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.

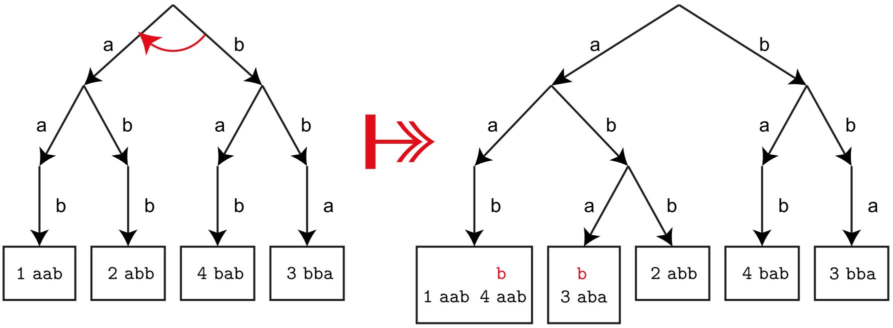

**Fig. 1** To the left: a trie of all length-3 substrings of aabbbab. To the right: an effect of copying the right subtree unto the left subtree, which corresponds to changing the first letter of all its substrings from b to a. In the original trie, there was exactly one pair of substrings from different subtrees of the root at Hamming distance 1; after the operation, there is a leaf containing modified substrings corresponding to these substrings. Such copy operations are performed in our algorithm top-down in the trie, making sure that each resulting modified substring has at most $k$ modifications

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.

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 $\mathbf{L}(v)$ 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; see Fig. 1.

The goal is to propagate the modified substrings to the 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 $u_1, \dots, u_a$. First, we distinguish a child of $v$ with maximum-size set $MS$, with ties being broken arbitrarily; 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.

As 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.

Finally, the algorithm branches into the subtrees of $u_1, \ldots, u_{a+1}$. A pseudocode of this process is presented as Algorithm 1.

Let us note that, in the special case of a binary alphabet, the child $u_{a+1}$ need not be created. Indeed, in this case, each node has at most two children, hence at most one light one, whereas when processing the subtree of $u_{a+1}$, we consider pairs of modified substrings that originate from different light children.

**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 $\{val(\alpha)\ :\ \alpha \in MS(v)\}$
    insert($v$, $\alpha$): inserts $\alpha$ 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 ← $val(\alpha)$[depth+1] for some $\alpha \in$ heavyChild
    wildcardTree ← ∅
    foreach lightChild ∈ children \ {heavyChild} do
        foreach $\alpha \in$ lightChild do
            if $|M(\alpha)| < k$ then
                $\alpha' \leftarrow \alpha$
                $\alpha'$[depth+1] ← $
                insert(wildcardTree, $\alpha'$)
                $\alpha'' \leftarrow \alpha$
                $\alpha''$[depth+1] ← heavyLetter
                insert(heavyChild, $\alpha''$)
    foreach child ∈ children ∪ {wildcardTree} do
        processNode(child)
```

*Processing a leaf* Each modified substring $\alpha$ 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 $\alpha$, 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:

$$H(\alpha) \cap H(\beta) = \emptyset, \quad W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset, \quad |H(\alpha)| + |H(\beta)| + |W(\alpha)| = k. \tag{3.1}$$

Lemma 3.3 below shows that if two modified substrings are compatible, then the original substrings were at Hamming distance at most $k$. Intuitively, $\alpha$ and $\beta$ are compatible only if the positions of modifications in $M(\alpha) \cup M(\beta)$ do not contain any position $j$ such that $T^m_{idx(\alpha)}[j] = T^m_{idx(\beta)}[j]$.

***Example 3.1*** Let us consider modified strings $\alpha_1, \ldots, \alpha_6$ with the following original strings and sets of modifications such that $val(\alpha_i) = \texttt{aba\$c}$ for all $i = 1, \ldots, 6$.

| $i$ | $U(\alpha_i)$ | $H(\alpha_i)$ | $W(\alpha_i)$ | $W^{-1}(\alpha_i)$ | $d_H(U(\alpha_1), U(\alpha_i))$ |
|---|---|---|---|---|---|
| 1 | aaabc | $\{(2, \mathrm{b})\}$ | $\{(4, \$)\}$ | $\{(4, \mathrm{b})\}$ | 0 |
| 2 | bbacc | $\{(1, \mathrm{a})\}$ | $\{(4, \$)\}$ | $\{(4, \mathrm{c})\}$ | 3 |
| 3 | abbcb | $\{(3, \mathrm{a}), (5, \mathrm{c})\}$ | $\{(4, \$)\}$ | $\{(4, \mathrm{c})\}$ | 4 |
| 4 | abacc | $\emptyset$ | $\{(4, \$)\}$ | $\{(4, \mathrm{c})\}$ | 2 |
| 5 | acacc | $\{(2, \mathrm{b})\}$ | $\{(4, \$)\}$ | $\{(4, \mathrm{c})\}$ | 2 |
| 6 | ababa | $\{(5, \mathrm{c})\}$ | $\{(4, \$)\}$ | $\{(4, \mathrm{b})\}$ | 2 |

Let us notice that $W(\alpha_i)$ is the same for all $i$. Let $k = 3$. The only modified string that is compatible with $\alpha_1$ is $\alpha_2$. Each of the remaining modified strings violates exactly one of the conditions from Eq. 3.1: $|H(\alpha_1)| + |H(\alpha_3)| + |W(\alpha_1)| = 4$, $|H(\alpha_1)| + |H(\alpha_4)| + |W(\alpha_1)| = 2$, $H(\alpha_1) \cap H(\alpha_5) = \{(2, \mathrm{b})\}$, $W^{-1}(\alpha_1) \cap W^{-1}(\alpha_6) = \{(4, \mathrm{b})\}$. Indeed, we have $d_H(U(\alpha_1), U(\alpha_2)) = 3$ and $d_H(U(\alpha_1), U(\alpha_i)) \neq 3$ for $i \in \{3, \ldots, 6\}$.

As proved in Lemma 3.8 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 $Count(\cdot, \cdot)$ 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 $O(2^k k|MS(v)|)$ time and space. Finally, the result for a modified substring $\alpha$ can be computed using the following direct consequence of the inclusion–exclusion principle.

**Lemma 3.2** *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

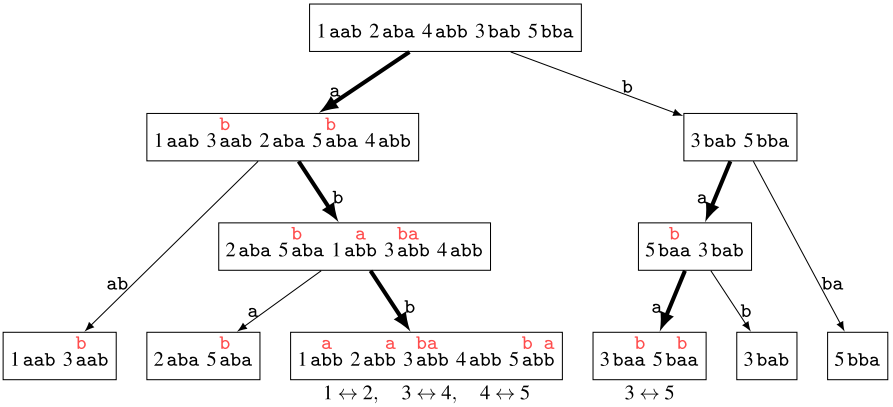

**Fig. 2** Computation of (2, 3)-mappability for the string $T = \texttt{aababba}$ from Example 1.1. Edges leading to heavy children are drawn in bold. Note that the alphabet is binary in this case, so wildcard subtrees do not need to be introduced; the only substitutions are from the (at most one) light child to the 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 binary case, 3.1 implies that these are substrings with modifications at different positions and exactly $k = 2$ modifications in total. 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

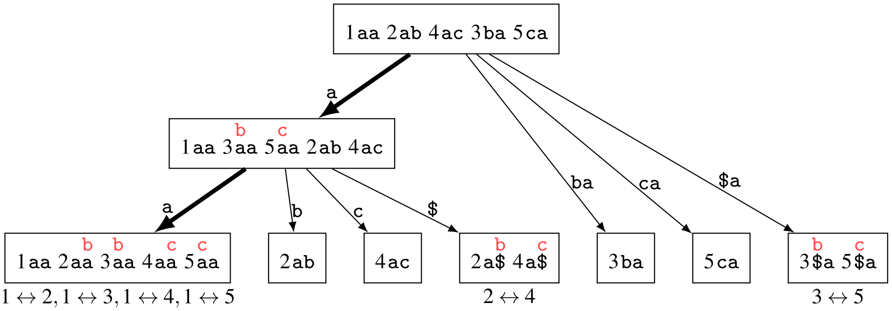

**Fig. 3** Computation of (1, 2)-mappability for the string $T = \texttt{aabaca}$. This example illustrates the need to use of wildcard symbols for a non-binary alphabet, as otherwise pairs from different light children of the same node would not be registered. In this case $k = 1$, so modified substrings are compatible if and only if at most one of them has a modification or both have a modification of the wildcard-type which originate from different letters. We have $A^2_{=1}[1] = 4$ and $A^2_{=1}[2] = A^2_{=1}[3] = A^2_{=1}[4] = A^2_{=1}[5] = 2$

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

which concludes the proof. □

*Examples* Examples of the execution of the algorithm for a binary and a ternary string can be found in Figs. 2 and 3, respectively.

*Correctness* We will show that it is enough to count pairs of modified substrings obtained in the leaves. First we show that a pair of compatible modified substrings implies a pair of length-$m$ substrings at Hamming distance at most $k$.

**Lemma 3.3** *If $\alpha, \beta \in MS(v)$ are compatible with $i = idx(\alpha)$, and $j = idx(\beta)$, then $d_H(T_i^m, T_j^m) = k$.*

***Proof*** By Eq. 3.1 we have $W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset$, so $T_i^m$ and $T_j^m$ differ at positions of modifications in $W(\alpha) = W(\beta)$. They also differ at positions of modifications in $H(\beta)$ since at the nodes corresponding to these positions, an ancestor of $\alpha$ (that is, the modified substring from which $\alpha$ originates) was in the heavy child and an ancestor of $\beta$ originated from a light child (recall that Eq. 3.1 includes $H(\alpha) \cap H(\beta) = \emptyset$). Symmetrically, $T_i^m$ and $T_j^m$ 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 Eq. 3.1. This shows that $d_H(T_i^m, T_j^m) \geq k$. With $val(\alpha) = val(\beta)$, we conclude that $d_H(T_i^m, T_j^m) = k$. □

We proceed with a proof that if two length-$m$ substrings are at distance at most $k$, then some leaf contains a pair of corresponding modified substrings that are compatible. Let us start with an observation that lists some basic properties of our algorithm. Both parts can be shown by straightforward induction.

**Observation 3.4** (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 $T_\ell^m$.*

We use the following auxiliary lemma.

**Lemma 3.5** *Assume that $d_H(T_i^m, T_j^m) = k$ and let $1 \leq x_1 < x_2 < \cdots < x_k \leq m$ be the indices where the two substrings differ. Further let $x_{k+1} = m + 1$. 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 $M(\beta_p)$.*

***Proof*** The proof goes by induction on $p$. As $\alpha_1$ and $\beta_1$, we take (un)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 3.4(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 $\alpha_{p-1}$ and $\beta_{p-1}$. 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 $\beta'$ such that $idx(\beta') = j$ and $M(\beta') = M(\beta_{p-1}) \cup \{(x_{p-1}, T_i^m[x_{p-1}])\}$ is inserted to $MS(w_1)$. 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 3.4(a). □

***Example 3.6*** Let us consider strings $\alpha = \texttt{aab}$ and $\beta = \texttt{aba}$ from Fig. 2 that differ at positions $x_1 = 2$ and $x_2 = 3$. The trie in the figure has a path that contains nodes storing the modified substrings from the following table.

| $p$ | $\alpha$ | $M(\alpha_p)$ | $val(\alpha_p)$ | $\beta$ | $M(\beta_p)$ | $val(\beta_p)$ | $\mathbf{D}(v_p)$ |
|---|---|---|---|---|---|---|---|
| 1 | aab | $\emptyset$ | aab | aba | $\emptyset$ | aba | 1 |
| 2 | aab | $\{(2, \mathrm{b})\}$ | abb | aba | $\emptyset$ | aba | 2 |
| 3 | aab | $\{(2, \mathrm{b})\}$ | abb | abb | $\{(3, \mathrm{b})\}$ | abb | 3 |

The following corollary is a direct consequence of Lemma 3.5.

**Corollary 3.7** *If $d_H(T_i^m, T_j^m) = k$, then there is a leaf $v$ and a pair of compatible modified substrings $\alpha, \beta \in MS(v)$ with $i = idx(\alpha)$ and $j = idx(\beta)$.*

***Proof*** Lemma 3.5, applied for $p = k + 1$, yields a leaf $v_{k+1}$ that contains compatible modified substrings $\alpha = \alpha_{k+1}$ and $\beta = \beta_{k+1}$ with $idx(\alpha) = i$ and $idx(\beta) = j$. □

The following lemma, a stronger version of the corollary, together with Lemma 3.3 shows that Algorithm 1 correctly computes the mappability table $\mathrm{A}^m_{=k}$.

**Lemma 3.8** *If $d_H(T_i^m, T_j^m) = k$, 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 $j = idx(\beta)$.*

***Proof*** Corollary 3.7 implies that there is at least one leaf that contains compatible modified substrings $\alpha$ and $\beta$ with $idx(\alpha) = i$ and $idx(\beta) = j$. 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 $idx(\beta') = j$.

We apply Lemma 3.5. 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 $\{1, \dots, k+1\}$ such that $x_p - 1 \leq \mathsf{lcp}(val(\alpha), val(\alpha'))$. By Observation 3.4(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, $\beta'$ 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 $\alpha'$ and $\beta'$ 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 Eq. 3.1. □

***Remark 3.9*** The recursive approach presented above is somewhat similar to the scheme used by Thankachan et al. [34] for computing the longest common substring with up to $k$ mismatches of two strings. We attempted to adapt the approach of [34] to computing $k$-mappability, but failed. 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]. This is verified in detail in "Appendix A". 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 3.10** (see [9, Proposition 13]) *Algorithm 1 applied up to the leaves takes* $O(n\binom{\log n+k+1}{k+1}2^k)$ *time and generates* $O(n\binom{\log n+k}{k}2^k)$ *modified substrings.*

Let us further analyze the space complexity of the algorithm.

**Observation 3.11** *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 $MS(w)$.*

**Lemma 3.12** *Algorithm 1 applied up to the leaves uses $O(nk)$ 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 $u_1, \dots, u_a$, creates $2(|MS(u_2)| + \dots + |MS(u_a)|)$ 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.

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

By Observations 3.11 and 3.4(b), $|MS(v)|$ is non-increasing on $\rho$. Moreover, if $v$ is a light child of its parent $w$, then $|MS(v)| \leq |MS(w)|/2$. Let us consider all nodes

$w$ on $\rho$ such that the unique child of $w$ that is on $\rho$ is a light child. The total number of modified strings created by the calls to `processNode`$(w)$ for all such nodes $w$ is $O(n)$ since we can bound it from above by a geometric series that sums to $O(n)$.

As for the calls to `processNode`$(w)$ for the remaining nodes on $\rho$, for every two modified strings they create, they put one of them in the child of $w$ that also belongs to $\rho$. Hence, it suffices to bound the total number of modified substrings originating from $T_i^m$ for each position $i$ that are in $MS(v)$ for some node $v$ on $\rho$. For a given position $i$, let $\alpha_1, \ldots, \alpha_b$ be all such modified substrings originating from $T_i^m$. By Observation 3.11, we have $M(\alpha_1) \subsetneq M(\alpha_2) \subsetneq \cdots \subsetneq M(\alpha_b)$ and thus $b \leq k$. In total, we create $O(nk)$ modified substrings in calls to `processNode` on nodes of $\rho$. □

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 $\beta$ with exactly $k$ modifications is created at a node $v$ (i.e., $|M(\alpha)| = k - 1$ in the if-statement), we do not include $\beta$ 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 $\beta$ created while processing a given node $v$ remains in the hash table upon completion of `processNode`$(v)$, then $\beta$ 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 $\beta$ is inserted to the hash table and there is another modified substring with the same hash in the table, we pick any such modified substring $\alpha$ and check if $val(\alpha) = val(\beta)$ in $O(k)$ time using lcp queries on $T$ with a method that resembles kangaroo jumping [18,28] (it requires $O(n)$-time preprocessing). By Lemma 3.12, the hash table contains up to $O(nk)$ entries at any given time, so the collision probability is $O(nk \cdot n^{-C}) = O(n^{-C+2})$. 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 $O(1)$ amortized time.

Let us call the resulting algorithm Algorithm 1'.

**Lemma 3.13** *The outputs of Algorithms 1 and 1' are the same. Moreover, Algorithm 1' works in $O(n\binom{\log n+k}{k}2^k k)$ time 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 $\mathbf{L}(u)$ 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., $O(n\binom{\log n+k}{k}2^k)$ by Fact 3.10. However, the time necessary to conduct the whole recursive procedure corresponds to the time complexity of Algorithm 1 if it had been executed with $k-1$ instead of $k$, i.e., also $O(n\binom{\log n+k}{k}2^k)$ by Fact 3.10. After $O(n)$-time preprocessing, for each modified substring, we can compute its Karp–Rabin fingerprint and check collisions in $O(k)$ 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 3.12 and 3.13 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 $O(n2^k k)$.

**Theorem 3.14** *There exists a Las-Vegas randomized algorithm that computes the $(k,m)$-mappability of a given length-n string in $O(n2^k k)$ space and, with high probability, in $O(n\binom{\log n+k}{k}4^k k)$ time. For $k=O(1)$, the space is $O(n)$ and the time becomes $O(n\log^k n)$.*

## 4 All-Pairs Hamming Distance Problem

Let us recall that in the all-pairs Hamming distance problem, given a set $\mathbf{R}$ of $r$ length-$m$ strings and an integer $k\in\{0,\ldots,m\}$, we are to 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 $k$. We will show how the algorithm from the previous section can be modified to solve this 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 $\beta$.

Let us construct a static balanced binary search tree (BST) in which the leaves correspond to the modified substrings $\beta_i$. 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 decide whether there is any other candidate in $set(u)$ that is compatible with $\alpha$. Therefore, in each node $u$, we will compute all the required machinery for using the inclusion-exclusion principle on the modified substrings in $\mathit{set}(u)$, that is, a dictionary that stores all non-zero values of $\mathit{Count}(s, B)$ for modified substrings $\beta \in \mathit{set}(u)$. Since every $\beta \in \mathit{MS}$ is present in $O(\log p)$ sets $\mathit{set}(u)$, 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 $\beta$ is a recursive procedure starting at the root of the BST. Assume that the algorithm is at some BST node $u$. We use Lemma 3.2 and the dictionary for $\mathit{set}(u)$ to count the elements $\beta' \in \mathit{set}(u)$ that are compatible with $\beta$. If this number is positive, the algorithm recursively descends to the children of node $u$. In the end, modified substrings $\beta'$ that are compatible with $\beta$ will be listed at the leaves of the BST. The correctness of this algorithm follows from Lemma 3.8.

Every application of Lemma 3.2 takes $O(2^k k)$ time. For each modified substring $\beta'$ that is compatible with a modified substring $\beta$, the algorithm will visit $O(\log p)$ BST nodes, which gives $O(2^k k \log p)$ time for finding each compatible modified substring $\beta' \in \mathit{MS}$. Note that $p \le r$ (see Observation 3.4(b)). Summing up over all trie nodes $v$ and applying Lemmas 3.13 and 3.12, we obtain the following result. (Observe that [9, Proposition 13] is applied for a family $\mathbf{F}$ of size $r$ rather than $n$.)

**Theorem 4.1** *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 $O(rm + 2^k kr \log r)$ space and, with high probability, in $O(rm + r\binom{\log r+k}{k}4^k k \log r + \mathsf{output} \cdot 2^k k \log r)$ time. For $k = O(1)$, the space is $O(rm + r \log r)$ and the time becomes $O(rm + r \log^{k+1} r + \mathsf{output} \cdot \log r)$.*

Notably, the algorithm underlying Theorem 4.1 works in $O(rm)$ time (with high probability) if $k = O(1)$, $m = \Omega(\log^{k+1} r)$, and $\mathsf{output} = O(rm/\log r)$.

## 5 Computing Mappability in $O(nm^k)$ Time and $O(n)$ Space

In this section, we generalize the $O(nm)$-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.) The algorithm then uses the suffix tree of $T$ to sort the modified suffixes according to this new lexicographic order. The focus of the original 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 $\binom{m}{\le k}$ subsets of $\{1, \dots, m\}$ 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 all these sets of strings in $O(nk\binom{m}{\le k})$ total time using the approach of [6], also obtaining the maximal blocks of equal strings in the sorted lists.

We now briefly describe the algorithm for sorting one such set of strings in time $O(nk)$ 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 lcp 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 $K \in \{1, \dots, k\}$. 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 this many size-$K$ supersets of the set of mismatching positions in the power set of $\{1, \dots, m\}$. We thus compute $\mathrm{A}_{=K}^{m}[i] = D_K[i] - \sum_{\kappa=0}^{K-1} \binom{m-\kappa}{K-\kappa} \mathrm{A}_{=\kappa}[i]$ in increasing order with respect to $K$, and we are done. (We precompute all relevant binomial coefficients in $O(k^2)$ time.)

**Theorem 5.1** *Given a string of length n, the $(k, m)$-mappability problem can be solved in $O(nk\binom{m}{\leq k})$ time and $O(n)$ space. For $k = O(1)$, the time becomes $O(nm^k)$.*

Combining Theorems 3.14 and 5.1 gives the following result.

**Corollary 5.2** *For every $k = O(1)$, there exists a randomized algorithm that computes the $(k, m)$-mappability of a given length-n string in $O(n)$ space and in $O(n \cdot \min\{m^k, \log^k n\})$ time with high probability.*

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

**Theorem 6.1** *The $(k, m)$-mappability for a given m and all $k \in \{0, \dots, m\}$ can be computed in $O(n^2)$ time using $O(n)$ space.*

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

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:

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

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 $O(n^2)$ pairs of letters and update the information on the respective intervals. Then $\mathrm{A}_{=k}^m[i] = |\{j : M[i, j] = k\}|$.

To achieve $O(1)$ 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 $O(n)$, 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 $\mathrm{A}^m_{=0}[i], \ldots, \mathrm{A}^m_{=m}[i]$ while processing the $i$th row. □

By $\mathsf{lcp}_k(i, j)$ we denote the length of the longest common prefix of $T_i$ and $T_j$ when up to $k$ mismatches are allowed, that is, the maximum $\ell$ such that $d_H(T_i^\ell, T_j^\ell) \le k$. Flouri et al. [15] proposed an $O(n^2)$-time algorithm to compute the longest common substring of two strings $S$, $T$ with at most $k$ mismatches. Their algorithm actually computes the lengths of the longest common prefixes with at most $k$ mismatches of every suffix of $S$ and $T$ and returns the maximum among them. Applied for $S = T$, it gives the following result.

**Lemma 6.2** [15] *For a string $T$ of length $n$, the values $\mathsf{lcp}_k(i, j)$ for all $i, j \in \{1, \ldots, n\}$ can be computed in $O(n^2)$ time.*

**Theorem 6.3** *The $(k, m)$-mappability for a given $k$ and all $m \in \{k, \ldots, n\}$ can be computed in $O(n^2)$ time and space.*

***Proof*** First we compute all the values $\mathsf{lcp}_k(i, j)$ using Lemma 6.2. We initialize an $n \times n$ matrix $Q$ setting all entries to 0. Then, for a pair $(i, j)$ such that $\mathsf{lcp}_k(i, j) = \ell$, we increment the entries $Q[\ell, i]$ and $Q[\ell, j]$. Note that if $\mathsf{lcp}_k(i, j) = \ell$, then $i$ (resp. $j$) will contribute 1 to the $(k, m)$-mappability values $\mathrm{A}^m_{\le k}[j]$ (resp. $\mathrm{A}^m_{\le k}[i]$) for all $m \in \{k, \ldots, \ell\}$. Thus, starting from the last row of $Q$, we iteratively add row $\ell$ to row $\ell - 1$. By the above observation, row $m$ ends up storing the $(k, m)$-mappability array $\mathrm{A}^m_{\le k}$. □

## 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 [22,23] 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 $S$, $T$ of length $n$ over binary alphabet and integers $k$, $d$.
**Output:** Is there a factor of $S$ of length $d$ that occurs in $T$ with $k$ mismatches?

**Lemma 7.1** [27] *Suppose there is $\epsilon > 0$ such that Common Substring of Length $d$ with k Mismatches can be solved in $O(n^{2-\epsilon})$ time on strings over binary alphabet for $k = \Theta(\log n)$ and $d = 21k$. Then SETH is false.*

**Theorem 7.2** *If the $(k, m)$-mappability can be computed in $O(n^{2-\epsilon})$ time for binary strings, $k, m = \Theta(\log n)$, and some $\epsilon > 0$, then SETH is false.*

***Proof*** We make a Turing reduction from Common Substring of Length $d$ with $k$ Mismatches. Let $S$ and $T$ be the input to the problem. We compute the $(k, d)$-mappabilities of strings $S \cdot T$ and $S \cdot T_1^{d-1}$ and store them in arrays $A$ and $B$, respectively. Henceforth, we consider only indices $i \in \{1, \ldots, n-d+1\}$ in the arrays. For each such index, $A[i]$ holds the number of length-$d$ factors of $S$, $X := S_{n-d+2}^{d-1} T_1^{d-1}$, and $T$ that are at Hamming distance $k$ from $S_i^d$, and $B[i]$ holds the number of length-$d$ factors of $S$ and $X$ that are at Hamming distance $k$ from $S_i^d$. For each $i$, we subtract $B[i]$ from $A[i]$. Then, $A[i]$ holds the number of length-$d$ factors of $T$ that are at Hamming distance $k$ from $S_i^d$. 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 7.1, an $O(n^{2-\epsilon})$-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 $O(n^{2-\epsilon})$-time algorithm for $(k, m)$-mappability with $k, m = \Theta(\log n)$ would also refute SETH. □

## 8 Final Remarks

Our main contribution is an $O(n \cdot \min\{m^k, \log^k n\})$-time $O(n)$-space algorithm for solving the $(k, m)$-mappability problem for a length-$n$ string over an integer alphabet. 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 to computing approximate longest common prefixes and substrings under edit distance [6,33]. 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.

**Acknowledgements** Panagiotis Charalampopoulos is supported by the Israel Science Foundation Grant 592/17. Tomasz Kociumaka is partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship. Jakub Radoszewski and Juliusz Straszyński are supported by the "Algorithms for text processing with errors and uncertainties"project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund, Project No. POIR.04.04.00-00-24BA/16, and by the Polish National Science Center, Grant Number 2018/31/D/ST6/03991.

## A Application of the Construction from [9]

In [9], a recursive procedure shown in Algorithm 2 was developed. This procedure takes as input a string $P$ and a family $\mathbf{F}_P$ that consists of tuples $(S, F, b)$ such that $F \in \mathbf{F}$ for some string family $\mathbf{F}$, $S$ is a suffix of $F$ of length $|S| = |F| - |P|$, and $b = k - d_H(F, PS) \geq 0$. Intuitively, the parameter $b$ can be seen as a "budget" of remaining letter substitutions that can be performed in the string ($PS$ obtained from $F$) that prevents exceeding the threshold $k$ of mismatches. In the first call, we have $P = \varepsilon$ and $\mathbf{F}_P = \{(F, F, k) : F \in \mathbf{F}\}$. For a non-empty string $S = aS'$, where $a \in \Sigma$, we denote $\mathsf{suf}(S) = S'$.

**Algorithm 2:** A recursive procedure inserting strings with prefix $P$ to sets $N(F)$.

```
Procedure Generate($P, \mathbf{F}_P$) is
    $h$ := a most frequent element of $\{S[1] : (S, F, b) \in \mathbf{F}_P$ and $S \neq \varepsilon\}$;
    foreach $(S, F, b) \in \mathbf{F}_P$ do                          // $b = k - d_H(F, PS) \geq 0$
        if $S = \varepsilon$ then $N(F) := N(F) \cup \{P\}$;
        else
            $c := S[1]$;
            $\mathbf{F}_{Pc} := \mathbf{F}_{Pc} \cup \{ (\mathsf{suf}(S), F, b) \}$;
            if $c \neq h$ and $b > 0$ then
                $\mathbf{F}_{Ph} := \mathbf{F}_{Ph} \cup \{ (\mathsf{suf}(S), F, b-1) \}$;
                $\mathbf{F}_{P\$} := \mathbf{F}_{P\$} \cup \{ (\mathsf{suf}(S), F, b-1) \}$;
    foreach $c \in \Sigma \cup \{\$\}$ such that $\mathbf{F}_{Pc} \neq \emptyset$ do
        Generate($Pc, \mathbf{F}_{Pc}$);
```

The following result from [9] shows that this abstract procedure can be implemented efficiently. In the statement below, we ignore the meaning of the resulting family of strings (which is important for computing the longest common substring of two strings with $k$ mismatches) and focus only on its size and the complexity of its construction.

**Theorem A.1** (see [9, Proposition 13]) *Let $\mathbf{F} \subseteq \Sigma^*$ be a finite family of strings and $k \geq 0$ be an integer. Then the family $\mathbf{F}' = \bigcup_{F \in \mathbf{F}} N(F)$ generated by Algorithm 2 has size at most $2^k \binom{\log|\mathbf{F}|+k}{k}|\mathbf{F}|$. Moreover, the compacted trie of $\mathbf{F}'$ can be constructed in $O(2^k|\mathbf{F}|\binom{\log|\mathbf{F}|+k+1}{k+1})$ time provided constant-time $\mathsf{lcp}$ queries for suffixes of the strings $F \in \mathbf{F}$.*

Let us now inspect how the recursive procedure `processNode` in Algorithm 1 translates one-to-one to procedure `Generate` in Algorithm 2 applied for the family $\mathbf{F} = \{T_i^m : i \in \{1, \ldots, n-m+1\}\}$ to showcase that they are indeed equivalent. For this string family, if $\mathbf{F}_P$ contains a triple $(\varepsilon, F, b)$ for some $F$ and $b$, then all the remaining triples have the first component equal to $\varepsilon$ as well as all strings in $\mathbf{F}$ are of the same length.

A node $v$ corresponds to string $P$ and the modified strings in $MS(v)$ correspond to the triples in $\mathbf{F}_P$ in such a way that the set $MS(v)$ (composed of pairs $(U(\alpha), M(\alpha))$) is

$$\{ (F,\ \{(i, P[i]) : i \in \{1, \ldots, |P|\},\ F[i] \neq P[i]\}) : (S, F, b) \in \mathbf{F}_P \}.$$

In both procedures, we construct light trees, a heavy tree (i.e., the call for $Ph$), and a wildcard tree (i.e., the call for $P\$$). These trees are constructed in the same manner. The modified strings $\alpha \in MS(v)$ that get copied to the heavy and wildcard trees in Algorithm 1 are those that satisfy $|M(\alpha)| < k$ and $U(\alpha)[\mathsf{depth}+1] \neq \mathsf{heavyLetter}$. In Algorithm 2, the triples $(S, F, b) \in \mathbf{F}_P$ that get copied to families $\mathbf{F}_{Ph}$ and $\mathbf{F}_{P\$}$ are those that satisfy $b > 0$ and $F[|P|+1] \neq h$. For $\alpha \in MS(v)$ corresponding to $(S, F, b) \in \mathbf{F}_P$, we have $|M(\alpha)| = k - b$, $|P| = \mathsf{depth}$, and $\mathsf{heavyLetter} = h$.[4] This yields a bijection between the family of copied modified strings and the family of copied triples. This concludes that indeed both constructions are equivalent.

Finally, the condition about constant-time $\mathsf{lcp}$-queries on strings $F \in \mathbf{F}$ from Theorem A.1, in this case being substrings of the text $T$, is satisfied with the aid of LCA queries on a suffix tree of $T$, so the theorem implies Fact 3.10.

## References

1. Alamro, H., Ayad, L.A.K., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with $k$-mismatches and applications. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) 44th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2018, LNCS, vol. 10706, pp. 636–649. Springer (2018). https://doi.org/10.1007/978-3-319-73117-9_45
2. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Straszyński, J.: Efficient computation of sequence mappability. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) 25th International Symposium on String Processing and Information Retrieval, SPIRE 2018, LNCS, vol. 11147, pp. 12–26. Springer (2018). https://doi.org/10.1007/978-3-030-00479-8_2
3. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P., Radoszewski, J., Sung, W.: Faster algorithms for 1-mappability of a sequence. Theor. Comput. Sci. **812**, 2–12 (2020). https://doi.org/10.1016/j.tcs.2019.04.026
4. Amir, A., Boneh, I., Kondratovsky, E.: The k-mappability problem revisited. In: Gawrychowski, P., Starikovskaya, T. (eds.) 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, LIPIcs, vol. 191, pp. 5:1–5:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CPM.2021.5
5. Antoniou, P., Daykin, J.W., Iliopoulos, C.S., Kourie, D., Mouchard, L., Pissis, S.P.: Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome. In: 9th International Conference on Information Technology and Applications in Biomedicine, ITAB 2009, pp. 1–4. IEEE (2009). https://doi.org/10.1109/itab.2009.5394394
6. Ayad, L.A.K., Barton, C., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with $k$-errors and applications. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) 25th International Symposium on String Processing and Information Retrieval, SPIRE 2018, LNCS, vol. 11147, pp. 27–41. Springer (2018). https://doi.org/10.1007/978-3-030-00479-8_3
7. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2), 75–94 (2005). https://doi.org/10.1016/j.jalgor.2005.08.001
8. Carriço, J.A., Crochemore, M., Francisco, A.P., Pissis, S.P., Ribeiro-Gonçalves, B., Vaz, C.: Fast phylogenetic inference from typing data. Algorithms Mol. Biol. **13**(1), 4 (2018). https://doi.org/10.1186/s13015-017-0119-7
9. Charalampopoulos, P., Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Linear-time algorithm for long LCF with $k$ mismatches. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) 29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018,

[4] Here, we assume that the two algorithms break ties for the heavy letter consistently, e.g., by choosing the lexicographically smallest letter.

LIPIcs, vol. 105, pp. 23:1–23:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.23

10. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Babai, L. (ed.) 36th Annual ACM Symposium on Theory of Computing, STOC 2004, pp. 91–100. ACM (2004). https://doi.org/10.1145/1007352.1007374
11. Crochemore, M., Francisco, A.P., Pissis, S.P., Vaz, C.: Towards distance-based phylogenetic inference in average-case linear-time. In: Schwartz, R., Reinert, K. (eds.) 17th International Workshop on Algorithms in Bioinformatics, WABI 2017, LIPIcs, vol. 88, pp. 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.WABI.2017.9
12. Derrien, T., Estellé, J., Sola, S.M., Knowles, D.G., Raineri, E., Guigó, R., Ribeca, P.: Fast computation and applications of genome mappability. PLoS ONE **7**(1), e30377 (2012). https://doi.org/10.1371/journal.pone.0030377
13. Dietzfelbinger, M., Meyer auf der Heide, F.: A new universal class of hash functions and dynamic hashing in real time. In: Paterson, M. (ed.) 17th International Colloquium on Automata, Languages and Programming, ICALP 1990, LNCS, vol. 443, pp. 6–19. Springer (1990). https://doi.org/10.1007/BFb0032018
14. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000). https://doi.org/10.1145/355541.355547
15. Flouri, T., Giaquinta, E., Kobert, K., Ukkonen, E.: Longest common substrings with $k$ mismatches. Inf. Process. Lett. **115**(6–8), 643–647 (2015). https://doi.org/10.1016/j.ipl.2015.03.006
16. Fonseca, N.A., Rung, J., Brazma, A., Marioni, J.C.: Tools for mapping high-throughput sequencing data. Bioinformatics **28**(24), 3169–3177 (2012). https://doi.org/10.1093/bioinformatics/bts605
17. Francisco, A.P., Bugalho, M., Ramirez, M., Carriço, J.A.: Global optimal eBURST analysis of multilocus typing data using a graphic matroid approach. BMC Bioinform. **10**(1), 152 (2009). https://doi.org/10.1186/1471-2105-10-152
18. Galil, Z., Giancarlo, R.: Parallel string matching with $k$ mismatches. Theor. Comput. Sci. **51**, 341–348 (1987). https://doi.org/10.1016/0304-3975(87)90042-9
19. Gog, S., Venturini, R.: Fast and compact Hamming distance index. In: Perego, R., Sebastiani, F., Aslam, J.A., Ruthven, I., Zobel, J. (eds.) 39th International ACM-SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2016, pp. 285–294. ACM (2016). https://doi.org/10.1145/2911451.2911523
20. Grabowski, S., Kowalski, T.M.: Algorithms for all-pairs Hamming distance based similarity. Softw. Pract. Exp. (2021). https://doi.org/10.1002/spe.2978
21. Hooshmand, S., Abedin, P., Gibney, D., Aluru, S., Thankachan, S.V.: Faster computation of genome mappability with one mismatch. In: 8th IEEE International Conference on Computational Advances in Bio and Medical Sciences, ICCABS 2018, p. 1. IEEE Computer Society (2018). https://doi.org/10.1109/ICCABS.2018.8541897
22. Impagliazzo, R., Paturi, R.: On the complexity of $k$-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001). https://doi.org/10.1006/jcss.2000.1727
23. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? J. Comput. Syst. Sci. **63**(4), 512–530 (2001). https://doi.org/10.1006/jcss.2001.1774
24. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006). https://doi.org/10.1145/1217856.1217858
25. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987). https://doi.org/10.1147/rd.312.0249
26. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, LNCS, vol. 2089, pp. 181–192. Springer (2001). https://doi.org/10.1007/3-540-48194-X_17
27. Kociumaka, T., Radoszewski, J., Starikovskaya, T.A.: Longest common substring with approximately $k$ mismatches. Algorithmica **81**(6), 2633–2652 (2019). https://doi.org/10.1007/s00453-019-00548-x
28. Landau, G.M., Vishkin, U.: Efficient string matching with $k$ mismatches. Theor. Comput. Sci. **43**, 239–249 (1986). https://doi.org/10.1016/0304-3975(86)90178-7
29. Mäkinen, V., Norri, T.: Applying the positional Burrows–Wheeler transform to all-pairs Hamming distance. Inf. Process. Lett. **146**, 17–19 (2019). https://doi.org/10.1016/j.ipl.2019.02.003
30. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058

31. Manzini, G.: Longest common prefix with mismatches. In: Iliopoulos, C.S., Puglisi, S.J., Yilmaz, E. (eds.) 22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015, LNCS, vol. 9309, pp. 299–310. Springer (2015). https://doi.org/10.1007/978-3-319-23826-5_29
32. Pockrandt, C., Alzamel, M., Iliopoulos, C.S., Reinert, K.: Genmap: ultra-fast computation of genome mappability. Bioinformatics **36**(12), 3687–3692 (2020). https://doi.org/10.1093/bioinformatics/btaa222
33. Thankachan, S.V., Aluru, C., Chockalingam, S.P., Aluru, S.: Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In: Raphael, B.J. (ed.) 22nd Annual International Conference on Research in Computational Molecular Biology, RECOMB 2018, LNCS, vol. 10812, pp. 211–224. Springer (2018). https://doi.org/10.1007/978-3-319-89929-9_14
34. Thankachan, S.V., Apostolico, A., Aluru, S.: A provably efficient algorithm for the k-mismatch average common substring problem. J. Comput. Biol. **23**(6), 472–482 (2016). https://doi.org/10.1089/cmb.2015.0235
35. Vaz, C., Nascimento, M., Carriço, J.A., Rocher, T., Francisco, A.P.: Distance-based phylogenetic inference from typing data: a unifying view. Brief. Bioinform. (2021). https://doi.org/10.1093/bib/bbaa147

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Authors and Affiliations

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

✉ Juliusz Straszyński
jks@mimuw.edu.pl

Panagiotis Charalampopoulos
panagiotis.charalampopoulos@post.idc.ac.il

Costas S. Iliopoulos
c.iliopoulos@kcl.ac.uk

Tomasz Kociumaka
kociumaka@berkeley.edu

Solon P. Pissis
solon.pissis@cwi.nl

Jakub Radoszewski
jrad@mimuw.edu.pl

[1] Reichman University, Herzliya, Israel

[2] Department of Informatics, King's College London, London, UK

[3] University of California, Berkeley, USA

[4] CWI, Amsterdam, The Netherlands

[5] Vrije Universiteit, Amsterdam, The Netherlands

[6] Institute of Informatics, University of Warsaw, Warsaw, Poland