Efficient Computation of Sequence Mappability

Mai Alzamel[1,2], Panagiotis Charalampopoulos[1], Costas S. Iliopoulos[1], Tomasz Kociumaka[3], Solon P. Pissis[1], Jakub Radoszewski[3], and Juliusz Straszyński[3(✉)]

[1] Department of Informatics, King's College London, London, UK
{mai.alzamel,panagiotis.charalampopoulos,
c.iliopoulos,solon.pissis}@kcl.ac.uk
[2] Computer Science Department, King Saud University, Riyadh, Saudi Arabia
[3] Institute of Informatics, University of Warsaw, Warsaw, Poland
{kociumaka,jrad,jks}@mimuw.edu.pl

Abstract. Sequence mappability is an important task in genome re-sequencing. In the (k, m)-mappability problem, for a given sequence T of length n, our goal is to compute a table whose ith entry is the number of indices $j \neq i$ such that length-m substrings of T starting at positions i and j have at most k mismatches. Previous works on this problem focused on heuristic approaches to compute 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 works in $\mathcal{O}(n \min\{m^k, \log^{k+1} n\})$ time and $\mathcal{O}(n)$ space for $k = \mathcal{O}(1)$. It requires a careful adaptation of the technique of Cole et al. [STOC 2004] to avoid multiple counting of pairs of substrings. We also show $\mathcal{O}(n^2)$-time algorithms to compute *all* results for a fixed m and all $k = 0, \ldots, m$ or a fixed k and all $m = k, \ldots, n-1$. Finally we show that the (k, m)-mappability problem cannot be solved in strongly subquadratic time for $k, m = \Theta(\log n)$ unless the Strong Exponential Time Hypothesis fails.

Keywords: Sequence mappability · Hamming distance
Genome sequencing · Longest common substring with k mismatches
Suffix tree

1 Introduction

Analyzing data derived from massively parallel sequencing experiments often depends on the process of genome assembly via re-sequencing; namely, assembly with the help of a reference sequence. In this process, a large number of reads (or

J. Radoszewski and J. Straszyński—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.

T. Gagie et al. (Eds.): SPIRE 2018, LNCS 11147, pp. 12–26, 2018.
https://doi.org/10.1007/978-3-030-00479-8_2

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 [10].

In turn, the process of re-sequencing depends heavily on how mappable a genome is given a set of 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 [7] (see also [3]). A great variance in genome mappability between species and gene classes was revealed in [7].

More formally, 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 $\mathrm{A}^m_{\leq k}$ whose ith entry $\mathrm{A}^m_{\leq k}[i]$ 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 [7] 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 the 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	$\mathrm{A}^3_{\leq 1}[i]$	2	2	1	2	1
$(2,3)$-mappability	$\mathrm{A}^3_{\leq 2}[i]$	3	3	3	4	3
Difference	$\mathrm{A}^3_{=2}[i]$	1	1	2	2	2

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

For convenience, in our algorithms we compute an array $\mathrm{A}^m_{=k}$ whose ith entry $\mathrm{A}^m_{=k}[i]$ 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 $\mathrm{A}^m_{\leq k}[i] = \sum_{k'=0}^{k} \mathrm{A}^m_{=k'}[i]$; see the "difference" row in the example above. Henceforth we refer to this modified problem as to the (k, m)-mappability problem.

Using the well-known LCP table [14,15,17], the $(0, m)$-mappability problem can be solved in $\mathcal{O}(n)$ time and space. Manzini [18] proposed an algorithm working in $\mathcal{O}(mn \log n / \log\log n)$ time and $\mathcal{O}(n)$ space for strings over a constant-sized alphabet for the case of $k = 1$. This was later improved in [2] with two

algorithms that require worst-case time $\mathcal{O}(mn)$ and $\mathcal{O}(n \log n \log\log n)$, respectively, and space $\mathcal{O}(n)$ for the case of $k = 1$. Moreover, the authors presented another algorithm requiring average-case time and space $\mathcal{O}(n)$ for uniformly random strings over a linearly-sortable integer alphabet of size σ if $k = 1$ and $m = \Omega(\log_\sigma n)$. In addition, they showed that their algorithm is generalizable for arbitrary k, requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ 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.

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 present an algorithm for the (k, m)-mappability problem that works in $\mathcal{O}(n\binom{\log n+k+1}{k+1}4^k k)$ time 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 [6].
2. In Sect. 4 we show an algorithm for the same problem that works in $\mathcal{O}(n\binom{m}{k}\sigma^k)$ time and $\mathcal{O}(n)$ space for a string over an integer alphabet. Together with the previous one, this yields an $\mathcal{O}(n \min\{m^k, \log^{k+1} n\})$-time and $\mathcal{O}(n)$-space algorithm for $\sigma, k = \mathcal{O}(1)$.
3. In Sect. 5 we describe $\mathcal{O}(n^2)$-time algorithms to compute *all* (k, m)-mappability results: for a fixed m and all $k = 0, \ldots, m$; or for a fixed k and all $m = k, \ldots, n-1$.
4. Finally, in Sect. 6 we show that the (k, m)-mappability problem cannot be solved in strongly subquadratic time for $k, m = \Theta(\log n)$ unless the Strong Exponential Time Hypothesis [12,13] fails.

In contributions 1 and 4 we apply very recent advances in the Longest Common Substring with k Mismatches problem that were presented in [5,16], respectively (see also [21]). In particular, in addition to [5], our contribution 1 requires careful counting of substring pairs to avoid multiple counting and a thorough analysis of the space usage. Technically this is the most involved contribution.

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, $T[i]\cdots T[j]$ is the *substring* (sometimes called *factor*) of T that starts at position i and ends at position 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 T and S, $|T| = |S|$, is defined as $d_H(T, S) = |\{i : T[i] \neq S[i],\ i = 1, 2, \ldots, |T|\}|$. If $|T| \neq |S|$, we set $d_H(T, S) = \infty$.

By $\mathsf{lcp}(S,T)$ we denote the length of the longest common prefix of S and T and by $\mathsf{lcp}(r,s)$ we denote $\mathsf{lcp}(T_r,T_s)$ for a fixed string T. By k-$\mathsf{lcp}(r,s)$ we denote the length of the longest common prefix of T_r and T_s when k mismatches are allowed, that is, the maximum ℓ such that $d_H(T_r^\ell,T_s^\ell)\leq k$.

Compact Trie. A compact trie $\mathcal{T}$ of a collection of strings C is obtained from the trie of C by removing all non-branching nodes, excluding the root and the leaves. The nodes of the trie which become nodes of $\mathcal{T}$ are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of $\mathcal{T}$ 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 its first letter. Each node of the trie can be represented in $\mathcal{T}$ 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 edge labels along the path from the root to v. Additionally, $\mathcal{D}(v)=|\mathcal{L}(v)|$ is the *string-depth* of node v.

Suffix Tree. The suffix tree $\mathcal{T}(T)$ of a string T is a compact trie representing all suffixes of T. A node v is a *terminal* node if its path-label is a suffix of T, that is, $\mathcal{L}(v)=T_i$ for some $1\leq i\leq n$; here v is also labeled with index i. Each substring of T is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(T)$. The *suffix link* of a node v with path-label $\mathcal{L}(v)=\alpha Y$ is a pointer to the node with path-label Y, where $\alpha\in\Sigma$ is a single letter and Y is a string. The suffix link of v exists if v is a non-root explicit node of $\mathcal{T}(T)$.

The suffix tree of a string of length n over an integer alphabet (together with the suffix links) can be computed in time and space $\mathcal{O}(n)$ [9]. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. For non-constant alphabets, in order to access the children of an explicit node by the first letter of their edge label, perfect hashing [11] can be used. Once $\mathcal{T}(T)$ is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v.

3 $\mathcal{O}(n\log^{k+1}n)$-Time and $\mathcal{O}(n)$-Space Algorithm

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 the first components of the pairs in M are pairwise distinct. By $val(\alpha)$ we denote the string U after all the substitutions and by $M(\alpha)$ we denote the set M.

The algorithm processes *modified substrings* of T that are modified strings originating from the substrings T_i^m. The index of origin of a modified substring α is denoted by $idx(\alpha)$ (that is, α is a modification of T_i^m for $i=idx(\alpha)$).

Overview of the Algorithm. Intuitively, the algorithm performs the task by efficiently simulating transformations of a compact trie initially containing all substrings T_i^m. The operation we would like to perform efficiently is copying one

subtree unto its sibling, changing the first letter on the appropriate label. This process effectively results in registering one mismatch for a large batch of substrings at once. Combining it together with the *smaller-to-larger* principle, this yields 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 all the modified substrings in $MS(v)$ and the string-depth $\mathcal{D}(v)$ is the length of this prefix. None of the strings in $MS(v)$ contains a modification at a position greater than $\mathcal{D}(v)$. The children of v are determined by subsets of $MS(v)$ that correspond to different letters at position $\mathcal{D}(v) + 1$. Furthermore, additional modified substrings with modifications at position $\mathcal{D}(v) + 1$ 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 put multiple appropriate modified substrings in a single leaf, where they will be processed in such way that every pair of substrings (T_i^m, T_j^m) differing on exactly k positions will be registered exactly once.

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 strings contained in any single subtree. We need to make sure that all relevant pairs satisfy this condition.

For this, we create an extra child u_{a+1} that contains all modified substrings from $MS(u_2) \cup \cdots \cup MS(u_a)$ with the letters at position $\mathcal{D}(v) + 1$ replaced by a common wildcard character \$. Note that each modified substring in u_{a+1} contains one more substitution compared to its source in one of the light subtrees. Hence, we refrain from copying any modified substring which already has k substitutions. This way, 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 $\mathcal{D}(v)+1$ 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.

Processing a Leaf. Each modified substring α stores its index of origin $idx(\alpha)$ and information about modified positions. As we have seen, the substitutions

[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 groups

    depth ← lcp(v)
    if depth = m then
        processLeaf(v)
        return
    children ← splitByLetter(v, depth + 1)
    heavyChild ← findHeaviest(children)
    heavyLetter ← least(heavyChild)[depth+1]
    wildcardTree ← ∅
    foreach lightChild ∈ children \ {heavyChild} do
        foreach α ∈ MS(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)
```

introduced in the recursion are of two types: of wildcard origin and heavy origin. For a modified substring α, we introduce a partition $M(\alpha) = W(\alpha) \cup H(\alpha)$ into modifications of these kinds. For all modified strings α in the same leaf, $val(\alpha)$ is the same and, hence, $W(\alpha)$ is the same. Finally, by $W^{-1}(\alpha)$ we denote the set $\{(j, T^m_{idx(\alpha)}[j]) : (j, \$) \in W(\alpha)\}$. In the end, we count the pairs of modified substrings (α, β) that satisfy the following conditions:

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

Modified substrings α and β that satisfy (1) are called *compatible*. For a given modified substring α, the number of compatible pairs (α, β) obtained in the same leaf is counted using the inclusion-exclusion principle as follows.

For convenience, let $R(\alpha)$ denote the disjoint union of $H(\alpha)$ and $W^{-1}(\alpha)$. 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 hashmap. They can be generated by iterating through all the subsets of $R(\beta)$ for all modified substrings $\beta \in MS(v)$, with a multiplicative $\mathcal{O}(2^k k)$ overhead in time and space.

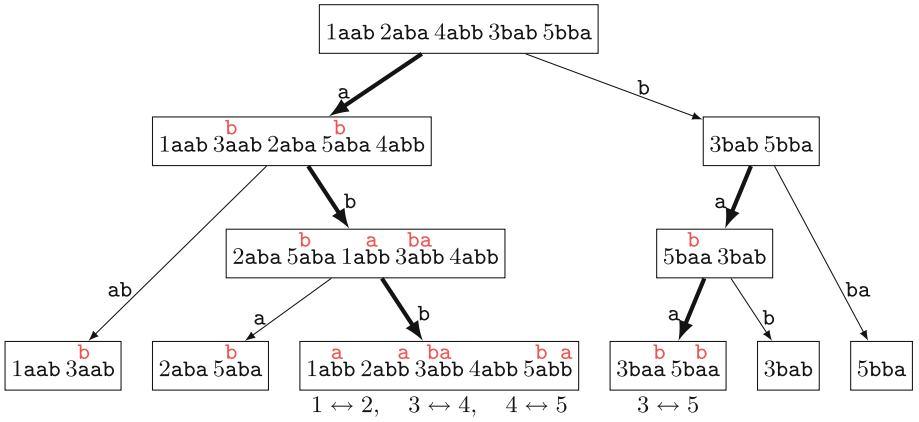

Fig. 1. Computation of $(2, 3)$-mappability for the string $T = \texttt{aababba}$ from Example 1. Note that in this case the alphabet is binary, so wildcard subtrees do not need to be introduced. Edges leading to heavy children are drawn in bold. The only substitutions are from a light child to a heavy child. The letters shown above are the original letters before the substitutions. The pairs of modified substrings are counted as shown; 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.

Finally, the result for a modified substring α—by which $A[idx(\alpha)]$ is increased—can be computed using the formula:

$$\sum_{B \subseteq R(\alpha)} (-1)^{|B|} \mathit{Count}(k - |M(\alpha)|, B).$$

Examples. Examples of the execution of the algorithm for a binary and a ternary string can be found in Figs. 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 2. *(a) If a node v stores modified substrings $\alpha, \beta \in MS(v)$, then it has a descendant v' with $\mathcal{D}(v') = \mathsf{lcp}(val(\alpha), val(\beta))$ and $\alpha, \beta \in MS(v')$.*
(b) Every node stores at most one modified substring with the same idx value.

The following lemma shows that the above approach correctly computes the (k, m)-mappability array $\mathrm{A}^m_{\leq k}$.

Lemma 3. *If $d_H(T^m_i, T^m_j) = k$, then there is exactly one leaf v and exactly one pair of compatible modified strings $\alpha, \beta \in MS(v)$ 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 in $W(\alpha) = W(\beta)$. They differ at positions in $H(\beta)$ since at the nodes corresponding to these positions, an ancestor of α (that is, the

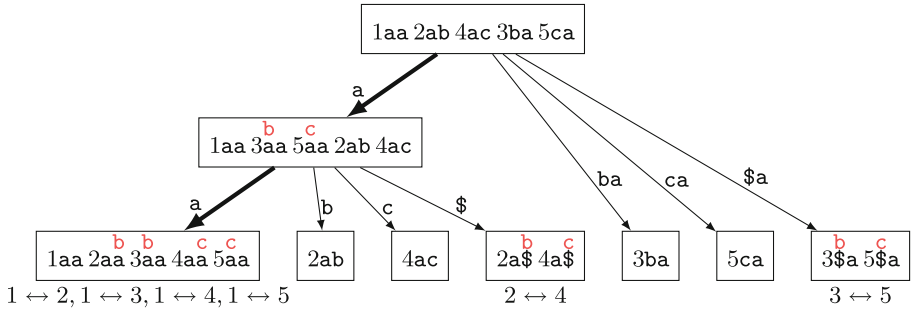

Fig. 2. Computation of $(1, 2)$-mappability for the string $T = \texttt{aabaca}$. This example shows how wildcard symbols are used in the algorithm. We have $A^2_{=1}[1] = 4$ and $A^2_{=1}[2] = A^2_{=1}[3] = A^2_{=1}[4] = A^2_{=1}[5] = 2$.

modified substring from which α originates) was in the heavy child (because $H(\alpha) \cap H(\beta) = \emptyset$ due to (1)) and an ancestor of β originated from a light child. Symmetrically, T_i^m and T_j^m differ at positions in $H(\alpha)$. In conclusion, they differ at positions 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 $d_H(T_i^m, T_j^m) \geq k$. With $val(\alpha) = val(\beta)$, we conclude that $d_H(T_i^m, T_j^m) = k$.

For a proof in the other direction, 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 $x_{k+1} = m + 1$.

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. For every $p = 1, \ldots, k+1$, there exists a node v_p and modified substrings $\alpha_p, \beta_p \in MS(v_p)$ such that:

- $idx(\alpha_p) = i$ and $idx(\beta_p) = j$;
- $\mathsf{lcp}(val(\alpha_p), val(\beta_p)) = x_p - 1 = \mathcal{D}(v_p)$;
- 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 (*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 2(a) guarantees existence of a node v_1 with $\mathcal{D}(v_1) = \mathsf{lcp}(\alpha_1, \beta_1)$ 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 string 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 $\alpha_p = \alpha'$, $\beta_p = \beta'$. The node v_p with $\mathcal{D}(v_p) = \mathsf{lcp}(val(\alpha_p), val(\beta_p))$ and $\alpha_p, \beta_p \in MS(v_p)$ must exist due to Observation 2(a). □

It suffices to apply the claim for $k = p + 1$. The node v_{k+1} is a leaf that contains compatible modified substrings $\alpha = \alpha_{k+1}$ and $\beta = \beta_{k+1}$.

Now, let us 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 the positions $x_1, \ldots, x_k$ (since $val(\alpha') = val(\beta')$) and no 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 $x_p \leq \mathsf{lcp}(val(\alpha), val(\alpha'))$. By Observation 2(b), $u \neq v_{k+1}$, so $p < k$.

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 corresponding to letters $T_i^m[x_{p-1}]$ and $T_j^m[x_{p-1}]$, respectively. If w_1 is the heavy child, β' must contain a modification of heavy origin at position x_{p+1}, 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+1}, which again gives a contradiction, or they both contain a modification of heavy origin, which contradicts the first part of condition (1). □

Remark 4. The authors also attempted to adapt the approach of [21] but failed due to multiple counting of substring pairs, e.g., for $T = \texttt{aabbab}$, $k = 2$, $m = 3$.

Implementation and Complexity. Our Algorithm 1, excluding the counting phase in the leaves, has exactly the same structure as Algorithm 1 in [5]. Proposition 13 from [5] provides a bound on the total size of the generated compact trie and an efficient implementation based on finger-search trees. We apply that proposition for a family $\mathcal{F}$ of size $\mathcal{O}(n)$ composed of substrings T_i^m to obtain the following bounds.

Fact 5 ([5]). *Algorithm 1 applied up to the leaves takes* $\mathcal{O}(n\binom{\log n+k+1}{k+1}2^k)$ *time.*

Let us further analyze the space complexity of the algorithm.

Lemma 6. *Algorithm 1 applied up to the leaves uses* $\mathcal{O}(nk)$ *working space.*

Proof. We inductively bound the working space of any recursive call. For a node v, let us define the potential

$$\Phi(v) = C \sum_{\alpha \in MS(v)} (k + 1 - |M(\alpha)|),$$

where C is a constant which depends on the implementation details.

We shall prove that the space consumption of a recursive call to v is bounded by $\Phi(v)$. We ignore the working space for the procedure processing leaves, so this is trivially true if v is a leaf. Next, let us analyze an internal node with children $u_1, \ldots, u_a, u_{a+1}$, where u_1 is the heavy child, $u_2, \ldots, u_a$ are the light children, and u_{a+1} corresponds to the wildcard character. Moreover, let $LS(v) = MS(u_2) \cup \cdots \cup MS(u_a)$.

Outside the recursive calls, the working space is $\mathcal{O}(|MS(v)|)$, which is below $\Phi(v)$ provided that C is large enough. Thus, let us analyze the space consumption during a recursive call to u_i. By the inductive hypothesis, the call uses $\Phi(u_i)$ working space. On top of that, we need to store the input for the remaining branches, which takes $\mathcal{O}(\sum_{i' \neq i} |MS(u_{i'})|)$ space.

If u_i is light, we observe that $\Phi(v) - \Phi(u_i) \geq C(|MS(v)| - |MS(u_i)|) \geq \frac{1}{2}C|MS(v)|$, which is sufficient to cover the total size $\mathcal{O}(|MS(v)|)$ of the input for the remaining branches. Similarly, if $i = a + 1$, then $\Phi(v) - \Phi(u_i) \geq C|MS(v)|$, because each modified string in $MS(u_i)$ has more changes than its original in $MS(v)$. Finally, to analyze the case $i = 1$ when u_i is heavy, we observe that $\sum_{i'>1} |MS(u_{i'})| \leq 2|LS(v)|$. However, $\Phi(v) - \Phi(u_1) \geq C|LS(v)|$, because each modified substring from $LS(v)$ inserted to $MS(u_1)$ has an additional substitution.

In the root call, we have $\Phi(r) = C \cdot (k + 1) \cdot |MS(r)| = \mathcal{O}(nk)$, as claimed. □

Fact 5 and Lemma 6 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 $2^k k$ and increase the space complexity by $\mathcal{O}(n2^k k)$.

Theorem 7. *Given a string of length n, the (k, m)-mappability problem can be solved in $\mathcal{O}(n\binom{\log n+k+1}{k+1}4^k k)$ time and $\mathcal{O}(n2^k k)$ space. For $k = \mathcal{O}(1)$, the time becomes $\mathcal{O}(n \log^{k+1} n)$ and the space is $\mathcal{O}(n)$.*

4 $\mathcal{O}(nm^k)$-Time and $\mathcal{O}(n)$-Space Algorithm

In this section we generalize the $\mathcal{O}(nm)$-time algorithm for $k = 1$ and integer alphabets from [2]. We start off with a simple $\mathcal{O}(nm\binom{m}{k}(\sigma-1)^k)$-time and $\mathcal{O}(n)$-space algorithm. We first construct the suffix tree $\mathcal{T}(T)$ in $\mathcal{O}(n)$ time. Within the same time complexity, we use a post-order traversal of $\mathcal{T}(T)$, to compute, for each explicit node v, a value $C(v)$ denoting the number of terminal nodes in the subtree rooted at v. For each T_i^m, we generate all possible $\binom{m}{k}$ combinations of substitution positions, create all $(\sigma - 1)^k$ distinct strings per combination, and then spell each created string from the root of $\mathcal{T}(T)$. Generating all combinations

can be done in $\mathcal{O}(\binom{m}{k})$ time [8] and creating and querying the strings can be done in $\mathcal{O}(m)$ time per string. If we successfully spell the whole string arriving at an explicit node v or an implicit node along an edge (u, v), we increment $A[i]$ by $C(v)$. The whole process takes $\mathcal{O}(nm\binom{m}{k}(\sigma-1)^k)$ time and $\mathcal{O}(n)$ space. For $k, \sigma = \mathcal{O}(1)$, the time becomes $\mathcal{O}(nm^{k+1})$. The counting of this algorithm is correct as we do the above for all $\binom{m}{k}(\sigma-1)^k$ pairwise distinct strings.

We next show how to shave a factor m from the time complexity. The main idea comes from observing that in the algorithm described above, after spelling from the root of $\mathcal{T}(T)$ a string of length m created by a combination of substitution positions, we start again from the root to spell a (potentially) completely different string. We can instead make use of the maximal match achieved in each spelling to query efficiently for another string. Intuitively, we construct $\sigma^k\binom{m}{k}$ strings of length n and spell them utilizing suffix links. When we reach string-depth m, we increment the respective counter if needed. Then the algorithm presented below correctly counts the number of times each length-m substring occurs in T with exactly k mismatches.

Consider a specific combination of k substitution positions with a sequence of k letters assigned to these k positions. We apply this "mask" to all non-overlapping length-m substrings of T (including, possibly, a suffix of length smaller than m) thus creating a new string S of length n. We start by spelling S from the root of $\mathcal{T}(T)$ until either we have a mismatch or we are at string-depth m. Let us denote the current depth by $d \leq m$. If $d < m$, we follow the suffix link of the last visited explicit node and traverse the edges down until we reach depth $\max\{d-1, 0\}$. If $d = m$, we have successfully spelled S_1^m arriving at an explicit node v or an implicit node along an edge (u, v). In this case, we increment $A[1]$ by $C(v)$ if and only if $d_H(S_1^m, T_1^m) = k$. If $\mathcal{D}(v) = m$, we follow its suffix link arriving by construction to a node of depth $m-1$; if not, we follow the suffix link of its parent u and traverse the edges down until we reach depth $m-1$. (Note that we know which edges we need to traverse by looking at S.) From this point onward, we process substring S_i^m, for all $2 \leq i \leq n-m+1$, analogously. Processing S takes time $\mathcal{O}(n)$ using an amortization argument analogous to the suffix tree construction of McCreight [19]. The working space is clearly $\mathcal{O}(n)$.

It remains to argue that for each length-m substring all different combinations with their different substitutions of k letters are induced by our construction of S. This is easy to see by considering a sliding window of length m running through S: it always contains k altered positions and these are uniquely determined by the combination used for the length-m substrings starting at positions equal to 1 *modulo* m. The final array A becomes $A_{=k}^m$. We arrive at the following result.

Theorem 8. *Given a string of length n over an integer alphabet, the (k, m)-mappability problem can be solved in $\mathcal{O}(n\binom{m}{k}\sigma^k)$ time and $\mathcal{O}(n)$ space. For $k, \sigma = \mathcal{O}(1)$, the time becomes $\mathcal{O}(nm^k)$.*

Combining Theorems 7 and 8 gives the following result for $\sigma, k = \mathcal{O}(1)$.

Corollary 9. *Given a string of length n over a constant-sized alphabet, the (k,m)-mappability problem can be solved in $\mathcal{O}(n \min\{m^k, \log^{k+1} n\})$ time and $\mathcal{O}(n)$ space for $k = \mathcal{O}(1)$.*

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

Theorem 10. *The (k,m)-mappability for a given m and all $k = 0,\ldots,m$ 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 of a 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 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 11. *The (k,m)-mappability for a given k and all $m = k,\ldots,n-1$ can be computed in $\mathcal{O}(n^2)$ time and space.*

Proof. We first prove the following claim.

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

Proof (*of Claim*). We process the pairs in batches B_δ for $\delta = 1,2,\ldots,n-1$ 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 $k\text{-}\mathsf{lcp}(1, 1+\delta)$ naively. Then, given that $k\text{-}\mathsf{lcp}(i,j) = \ell$, where $j - i = \delta$, we will retrieve $k\text{-}\mathsf{lcp}(i+1, j+1)$ 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 $k\text{-}\mathsf{lcp}(i+1, j+1) = \ell - 1$. 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 $k\text{-}\mathsf{lcp}(i+1, j+1) = \ell - 1$. 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 k-lcp's as follows. We initialize an $n \times n$ matrix Q. Then, for a pair (i, j) such that $k\text{-}\mathsf{lcp}(i, j) = \ell$, we increment by 1 the entries $Q[\ell, i]$ and $Q[\ell, j]$. Note that if $k\text{-}\mathsf{lcp}(i, j) = \ell$, then i (resp. j) will contribute 1 to the (k, m)-mappability values $\mathrm{A}^m_{\leq k}[j]$ (resp. $\mathrm{A}^m_{\leq k}[i]$) for all $1 \leq m \leq \ell$. 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 $\mathrm{A}^m_{\leq k}$. □

6 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 [12,13] 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 12 ([16]). *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 13. *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 $i = 1, \ldots, n-d+1$, we subtract $B[i]$ by $A[i]$. 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 $i = 1, \ldots, n-d+1$.

By Lemma 12, 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. □

7 Final Remarks

Our main contribution is an $\mathcal{O}(n \min\{m^k, \log^{k+1} n\})$-time and $\mathcal{O}(n)$-space algorithm for solving the (k, m)-mappability problem. Let us recall that genome mappability, as introduced in [7], 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 very recent contributions on approximate matching under edit distance [4,20]. 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. 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.) SOFSEM 2018. LNCS, vol. 10706, pp. 636–649. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73117-9_45
2. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P., Radoszewski, J., Sung, W.-K.: Faster algorithms for 1-mappability of a sequence. In: Gao, X., Du, H., Han, M. (eds.) COCOA 2017. LNCS, vol. 10628, pp. 109–121. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71147-8_8
3. 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: Information Technology and Applications in Biomedicine, ITAB 2009. IEEE (2009). https://doi.org/10.1109/itab.2009.5394394
4. 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., et al. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 27–41. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00479-8_3
5. Charalampopoulos, P., et al.: Linear-time algorithm for long LCF with k mismatches. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Combinatorial Pattern Matching, CPM 2018. LIPIcs, vol. 105, pp. 23:1–23:16. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.23
6. 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
7. Derrien, T.: Fast computation and applications of genome mappability. PLoS ONE **7**(1), e30377 (2012). https://doi.org/10.1371/journal.pone.0030377
8. Eades, P., McKay, B.D.: An algorithm for generating subsets of fixed size with a strong minimal change property. Inf. Process. Lett. **19**(3), 131–133 (1984). https://doi.org/10.1016/0020-0190(84)90091-7
9. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1997, pp. 137–143. IEEE Computer Society (1997). https://doi.org/10.1109/SFCS.1997.646102

10. 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
11. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $O(1)$ worst case access time. J. ACM **31**(3), 538–544 (1984). https://doi.org/10.1145/828.1884
12. 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
13. 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
14. 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
15. 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.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48194-X_17
16. Kociumaka, T., Radoszewski, J., Starikovskaya, T.A.: Longest common substring with approximately k mismatches (2017). arxiv.1712.08573
17. 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
18. Manzini, G.: Longest common prefix with mismatches. In: Iliopoulos, C., Puglisi, S., Yilmaz, E. (eds.) SPIRE 2015. LNCS, vol. 9309, pp. 299–310. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23826-5_29
19. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976). https://doi.org/10.1145/321941.321946
20. 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.) RECOMB 2018. LNCS, vol. 10812, pp. 211–224. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89929-9_14
21. 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