Algorithmic Framework for Approximate Matching Under Bounded Edits with Applications to Sequence Analysis

Sharma V. Thankachan[1(✉)], Chaitanya Aluru[2], Sriram P. Chockalingam[3], and Srinivas Aluru[3,4]

[1] Department of Computer Science, University of Central Florida, Orlando, FL, USA
sharma.thankachan@ucf.edu

[2] Department of Computer Science, Princeton University, Princeton, NJ, USA
caluru@princeton.edu

[3] School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA
srirampc@gatech.edu, aluru@cc.gatech.edu

[4] Institute for Data Engineering and Science, Georgia Institute of Technology, Atlanta, GA, USA

Abstract. We present a novel algorithmic framework for solving approximate sequence matching problems that permit a bounded total number k of mismatches, insertions, and deletions. The core of the framework relies on transforming an approximate matching problem into a corresponding exact matching problem on suitably edited string suffixes, while carefully controlling the required number of such edited suffixes to enable the design of efficient algorithms. For a total input size of n, our framework limits the number of generated edited suffixes to no more than a factor of $O(\log^k n)$ of the input size (for any constant k), and restricts the algorithm to linear space usage by overlapping the generation and processing of edited suffixes. Our framework improves the best known upper bound of $n^2k^{1.5}/2^{\Omega(\sqrt{\log n/k})}$ for the classic k-edit longest common substring problem [Abboud, Williams, and Yu; SODA 2015] to yield the first strictly sub-quadratic time algorithm that runs in $O(n \log^k n)$ time and $O(n)$ space for any constant k. We present similar subquadratic time and linear space algorithms for (i) computing the alignment-free distance between two genomes based on the k-edit average common substring measure, (ii) mapping reads/read fragments to a reference genome while allowing up to k edits, and (iii) computing all-pair maximal k-edit common substrings (also, suffix/prefix overlaps), which has applications in clustering and assembly. We expect our algorithmic framework to be a broadly applicable theoretical tool, and may inspire the design of practical heuristics and software.

1 Introduction

Numerous problems related to exact sequence matching can be solved efficiently, often within optimal time and space bounds, typically using versatile string

B. J. Raphael (Ed.): RECOMB 2018, LNBI 10812, pp. 211–224, 2018.
https://doi.org/10.1007/978-3-319-89929-9_14

data structures such as suffix trees and suffix arrays. However, variants of such sequence matching problems that permit a limited number of mismatches or edits (insertions/deletions/mismatches) are often challenging and many problems are still open. For example, the classic problem of finding the longest common substring (LCS[1]) between a pair of sequences is easily solvable in optimal *linear time* using suffix trees, a solution that dates back to the 70's [31]. However, when the sequences contain (an unbounded number of) wild-card characters, an $n^{2-o(1)}$ time conditional lower bound, based on the Strong Exponential Time Hypothesis (SETH), comes into play [2]. As for the k-edit LCS problem[2], the best known result is only slightly better than a straightforward dynamic programming solution. Specifically, the run time is $n^2k^{1.5}/2^{\Omega(\sqrt{\log n/k})}$ and the algorithm is randomized [1].

In recent times, there is renewed interest in approximate sequence matching problems due to their wide applicability in computational biology. Many fundamental problems between evolutionarily related genomes, or components to the solutions thereof, can be cast as edit distance problems, with bounded versions of significant practical interest. Short read sequencers sport low error rates, typically <1–3% of sequence length, which is within a few hundred bases. Many problems in relating such reads to each other, or to the source genomes they originate from, can be effectively modeled as bounded edit distance problems. While many such problems can be solved efficiently in practice via heuristics, their worst-case run times are often the same as alignment-based methods that allow unconstrained edit distance. Thus, an algorithmic framework for approximate sequence matching that can lead to the design of strictly subquadratic time algorithms for such problems is of significant theoretical and practical interest.

Our Contributions and Relation to Prior Work

In this work, we focus on multiple approximate sequencing matching problems under a bounded number k of edits. We expect k to be a small constant in practice. Our algorithms work for arbitrary values of k, but they are designed to be superior in both asymptotic and practical runtimes for small values of k. We first develop a novel algorithmic framework that is potentially applicable to a broad class of problems, including the four problems solved in this paper. The core of the framework is a transformation by which an approximate matching problem on exact suffixes can be converted into an exact matching counterpart on approximate suffixes, specifically, suffixes with at most k edits. The number of such *k-edited suffixes* generated is constrained to a *polylog* factor of the input size, through a non-trivial application of Sleator and Tarjan's classic heavy path tree decomposition technique [26]. As a result, the framework yields algorithms with runtime behavior of $O(n\ \mathit{poly}\log(n))$ while consuming only linear $O(n)$ space, for a total input of size n, marking a significant improvement over current

[1] In this paper, we use LCS to denote the longest common substring. Note that LCS is frequently used in literature to refer to the longest common subsequence instead.

[2] Find the longest substring of a sequence that matches with a substring of another sequence, allowing $\leq k$ edits.

worst-case runtimes that are quadratic or near quadratic. Using this framework, we propose asymptotically faster algorithms for three well known and widely applicable problems in biological sequence analysis, and derive the first strictly sub-quadratic time algorithm for the k-edit LCS problem as a corollary to one of these. As will become evident later, the design of appropriate k-edited suffixes and algorithms for processing them are specific to the problem at hand, leading to a rich algorithmic framework for tackling additional approximate sequence matching problems.

Our first result concerns alignment-free genomic distance based on the average common substring (ACS) measure, proposed by Burstein *et al.* [9]. The ACS between genomes X and Y is:

$$\mathsf{ACS}(\mathsf{X},\mathsf{Y}) = \frac{1}{|\mathsf{X}|}\sum_{i=1}^{|\mathsf{X}|} L[i], \text{ where } L[i] = \max_j |\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)|$$

Here X_i is the i-th longest suffix of X and LCP denotes the longest common prefix. The distance metric based on ACS is defined as

$$\mathsf{Dist}(\mathsf{X},\mathsf{Y}) = \frac{1}{2}\left(\frac{\log|\mathsf{Y}|}{\mathsf{ACS}(\mathsf{X},\mathsf{Y})} + \frac{\log|\mathsf{X}|}{\mathsf{ACS}(\mathsf{Y},\mathsf{X})}\right) - \frac{1}{2}\left(\frac{\log|\mathsf{X}|}{\mathsf{ACS}(\mathsf{X},\mathsf{X})} + \frac{\log|\mathsf{Y}|}{\mathsf{ACS}(\mathsf{Y},\mathsf{Y})}\right).$$

Since its introduction, ACS has proven to be useful in multiple applications including phylogeny reconstruction [4,6,10,12,13,15]. It was later observed that its approximate variants, k-mismatch and k-edit ACS, that are based on permitting k mismatches (or k edits, respectively) in the LCP computation, more accurately model genome evolution and lead to higher quality phylogenetic trees [18]. The ACS computation using exact substring composition, as described above, is straightforward to compute in linear time using suffix trees [9]. The k-mismatch ACS and the k-edit ACS can be computed by a trivial $O(n^2k)$ dynamic programming algorithm, which is prohibitively expensive for large genomes. Leimeister and Morgenstern [18] proposed algorithms that heuristically estimate k-mismatch ACS. Apostolico *et al.* [5] were the first to break the $O(n^2k)$ bound for an exact solution by proposing an $O(n^2/\log n)$ run time algorithm. However, the first strictly sub-quadratic algorithms are by Thankachan *et al.* [3,27], that run in $O(n\log^k n)$ time. Also see [22,24,28,29].

To date, there is no non-trivial solution for the k-edit ACS, beyond the straightforward $O(n^2k)$ algorithm. Unfortunately, the previous techniques for k-mismatch ACS do not easily extend to k-edit ACS. In comparison to mismatches, insertions and deletions are much harder to account for as they introduce a combinatorially larger number of possibilities (3^k ways of making modifications at k given locations) and also alter sequence lengths. Using the algorithmic framework presented in this paper, we present the first *strictly sub-quadratic* time algorithms for both k-edit ACS and k-edit LCS, that run in $O(n\log^k n)$ time and $O(n)$ space for any constant k[3].

[3] Throughout the analysis, we treat k as a constant for brevity. However, with a tighter analysis (deferred to full version), we can bound the time and space by $O(n(c\log n)^k/k!)$ and $O(c^k n)$, respectively for a constant c without making any such assumption on the value of k.

Theorem 1. *Given two sequences* X *and* Y *of* n *characters in total and a constant* k*, we can compute* $\forall i, L[i] = \max_j |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ *in* $O(n \log^k n)$ *time using* $O(n)$ *space. Here* $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ *is the length of the longest common prefix of* X_i *and* Y_j *after allowing* $\leq k$ *edits.*

In addition, we provide sub-quadratic algorithms for the following problems. Note that the size of the alphabet set is $O(1)$ in all these applications, however we make no such assumptions in the complexity analysis.

- **Read mapping:** A collection of m reads of length ℓ each can be mapped to a reference genome G while permitting at most k edits per read in $O((n + \mathsf{occ}) \log^k n)$ time using $O(n)$ space for any constant k. Here $n = |G| + m\ell$ is the input size and occ is the output size.
- **All-pair Maximal k-edit Common Substrings:** Given a collection of m reads of total length n, all pairwise k-edit maximal common substrings of length $\geq \tau$ can be computed in $O((n + \mathsf{occ}) \log^k n)$ time using $O(n)$ space for any constant k. Here occ is the output size.
- **All-pair Maximal k-edit suffix/prefix overlaps:** Given a collection of m reads of total length n and a length threshold τ, all pairwise k-edit maximal suffix/prefix overlaps of length $\geq \tau$ can be computed in $O((n + \mathsf{occ}) \log^k n)$ time using $O(n)$ space for any constant k. Here occ is the output size.

All of these are widely studied problems with excellent heuristic solutions and software availability. The read mapping problem is typically solved using seed-and-extend heuristics with exact matching or spaced seeds computed using a pre-built index of the genome such as BWT or FM-index (e.g. [17,19,21]; see [20] for a survey). Similarly, the other two problems are also solved through seed-and-extend type filtering solutions such as suffix filtering [16,30], spaced seeds filtering [8], and substring filtering [25]. Our goal is to present asymptotically efficient and sub-quadratic worst-case run-time algorithms for these commonly solved problems to improve upon their upper bounds. We remark that the algorithms presented here can also be used in conjunction with any existing seed-based heuristics by permitting seeds with bounded edit distance.

Roadmap. In Sect. 2, we present an overview of our framework and the key results, which are instrumental in achieving the above claimed worst-case run times. The proofs of the key results of our framework are described in detail in Sect. 3. We complete the proof of Theorem 1 in Sect. 4. In Sect. 5, we present our solutions to the other problems listed.

2 Our Algorithmic Framework

Our approximate sequence matching framework takes a collection of two or more sequences and a constant k as input. Then, a controlled number of changes (edits) are applied to the suffixes of all input sequences, so that an approximate sequence matching task over the input can now be transformed to an equivalent exact prefix matching over the newly generated edited-suffixes. We illustrate our

framework with a collection of two input sequences (X and Y of total length n). The framework relies on a Generalized Suffix Tree (GST), a compact trie representation of all suffixes of all input sequences. It takes $O(n)$ space for storage and $O(n)$ time for construction [23,31]. For any two suffixes X_i and Y_j, we can compute $|\mathsf{LCP}_0(\mathsf{X}_i, \mathsf{Y}_j)| = |\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)| = z$ in constant time using GST and $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ for any $k > 0$ in $O(3^k)$ time via the following recursion:

$$|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)| = z + \max \begin{cases} 1 + |\mathsf{LCP}_{k-1}(\mathsf{X}_{i+z+1}, \mathsf{Y}_{j+z+1})| & \text{(substitution)} \\ |\mathsf{LCP}_{k-1}(\mathsf{X}_{i+z+1}, \mathsf{Y}_{j+z})| & \text{(deletion in } \mathsf{X}_i) \\ |\mathsf{LCP}_{k-1}(\mathsf{X}_{i+z}, \mathsf{Y}_{j+z+1})| & \text{(deletion in } \mathsf{Y}_j) \end{cases}$$

Observe that while computing LCP_k, a substitution (in at least one suffix) is equivalent to deletions in both suffixes at the same location. For example, $\mathsf{X}_i = AATCGGT..$ and $\mathsf{Y}_j = AATGGTT..$ disagree at the 4th position. To make them agree more, we can either delete the 4th character from both suffixes, or change the 4th character in at least one suffix to match the 4th character of the other. Also, deletion in X_i (respectively, Y_j) is equivalent to an appropriate insertion in Y_j (respectively, X_i). Therefore, in general we have many possible (equivalent) ways of correcting the first k disagreements between X_i and Y_j. Note that the length of the resulting LCP_k may differ (slightly) as per our choice within the equivalent cases. However, the framework we propose exploits the fact that many of the equivalent cases will lead to the correct solution, and makes a suitable fixed choice.

Overview. A suffix after applying $\leq k$ edits is called a **k-edited suffix**. Let X'_i and Y'_j be k-edited suffixes derived from X_i and Y_j, respectively. Then, the value of $|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|$ can range anywhere between 0 and $|\mathsf{LCP}_{2k}(\mathsf{X}_i, \mathsf{Y}_j)|$. However, if the modifications turn **exactly** the first k disagreeing positions into agreements, then $|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|$ is precisely $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$. A **set of two** such edited suffixes is called an $(i, j)_k$-$\mathsf{maxpair}$. We call a collection of k-edited suffixes an order-k universe (denoted by U_k) if for all (i, j) pairs, $\exists (i, j)_k$-$\mathsf{maxpair} \subseteq U_k$. Note that U_0 is simply the set of all suffixes of X and Y. Trivially, there exists an order-k universe of size $\binom{n}{2}$. However, the core of our framework is a meticulous construction of an order k universe of size $O(n \log^k n)$, based on the *heavy path decomposition* strategy by Sleator and Tarjan [26] as in Cole *et al.* [11]. Various approximate sequence matching problems can then be solved via processing U_k in linear or near-linear time.

Representation of Edited Suffixes. Clearly, it is cumbersome to keep track of all edits applied on suffixes during the creation of edited-suffixes. However, we have the following crucial observation: *for each edited suffix, we do not need to keep track of all edits, but* **only substitutions** *and the* **total number of insertions and deletions**. Specifically, let X'_i be a k-edited suffix obtained via a combination of insertions, deletions and substitutions on X_i. Then, X'_i can be simply represented as a concatenation of a combination of $O(k)$ sub-strings of X and characters in the alphabet set, along with the following two satellite information.

– $\delta(\mathsf{X}'_i)$: **number** of insertions and deletions made to transform X_i to X'_i.
– $\Delta(\mathsf{X}'_i)$: **set** of positions in X'_i corresponding to substitutions in X_i.

Example: Let $\mathsf{X}_i = CATCATCATCAT$. We consider the following edits simultaneously on X_i: delete the 2nd and 10th character, change the 4th character to T and the 9th character to A, and insert G after position 6. Then, $\mathsf{X}'_i = CT\underline{T}ATGCA\underline{A}AT$, $\delta(\mathsf{X}'_i) = 3$ and $\Delta(\mathsf{X}'_i) = \{3, 9\}$.

Lemma 1. *Let X'_i (respectively, Y'_j) be obtained via at most k edits on X_i (respectively, Y_j). Then, the value of $|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|$ can range anywhere between 0 and $|\mathsf{LCP}_{2k}(\mathsf{X}_i, \mathsf{Y}_j)|$. However, if we impose the following condition, then $|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|$ is at most $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$.*

$$|\Delta(\mathsf{X}'_i) \cup \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j) \leq k.$$

Proof. If we allow k edits on each suffix, we can correct at most $2k$ disagreements. However, the condition limits the total number of insertions/deletions and distinct substitution positions. □

We now define the notion of $(i, j)_k$-maxpair in a formal way.

Definition 1. *Let X'_i be a k-edited suffix derived from X_i and Y'_j be a k-edited suffix derived from Y_j. Then, we call the set $\{\mathsf{X}'_i, \mathsf{Y}'_j\}$ an $(i, j)_k$-maxpair iff*

$$|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| = |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)| \text{ and } |\Delta(\mathsf{X}'_i) \cup \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j) \leq k.$$

Lemma 2. *Given two k-edited suffixes, we can compute the length of their longest common prefix (hence their lexicographic order) in $O(k)$ time via $O(k)$ number of $|\mathsf{LCP}|$ queries on the GST.*

3 Details of the Construction of U_k

We show how to construct the universe U_k in small parts (in linear work space). The **parts** of U_k, denoted by $\{\mathcal{P}^k_1, \mathcal{P}^k_2, \mathcal{P}^k_3, ...\}$ are its subsets (not necessary disjoint) such that the following properties are ensured.

1. $\max_f |\mathcal{P}^k_f| = O(n)$
2. $\sum_f |\mathcal{P}^k_f| = O(n \log^k n)$
3. for any (i, j), $\exists f$ such that a two-element subset of $\mathcal{P}^k_f$ is an $(i, j)_k$-maxpair

The construction procedure is recursive. We first construct U_0, then U_1 from U_0 and so on. The base case, i.e., an order 0 universe U_0 has exactly one part, the set of all suffixes of X and Y. We now proceed to the inductive step, where we assume the availability of order-h universe U_h (specifically, its parts $\mathcal{P}^h_1, \mathcal{P}^h_2, \dots$) for an $h \geq 0$ and the task is to obtain the parts $\mathcal{P}^{h+1}_1, \mathcal{P}^{h+1}_2, \dots$ of U_{h+1}. To do so, we apply the following steps on each $\mathcal{P}^h_f$. We describe the procedure first and prove its correctness later.

1. Let $m = |\mathcal{P}_f^h|$ and $\mathcal{T}$ be a compact trie of all h-edited suffixes in $\mathcal{P}_f^h$. Notice that $\mathcal{T}$ is GST when $h = 0$. Classify the nodes in $\mathcal{T}$ into **light** or **heavy**: the root is always light and any other node is heavy, if it is the *heaviest child*[4] of its parent. Furthermore, a maximal downward path starting from a light node where all other nodes on the path are heavy is called a **heavy path**. A key property is that the *number of heavy paths that intersect any root to leaf path is* $\leq \log m$ [11,26]. Equivalently, the number of light nodes on any root to leaf path is $\leq \log m$. Therefore the sum of subtree sizes of all light nodes in $\mathcal{T}$ is $\leq m \log m$, because each leaf contributes to at most $\log m$ light rooted subtree.
2. Corresponding to each internal *light* node u in $\mathcal{T}$, there will be a **part**, say $\mathcal{P}_t^{h+1}$. The steps involved in its construction are as follows. Let Q be the set of h-edited suffixes corresponding to the leaves in the subtree of u, α be the h-edited suffix corresponding to the particular leaf on the heavy path through u. Then,
$$\mathcal{P}_t^{h+1} = \{\alpha\} \bigcup_{\beta \in Q, \beta \neq \alpha} \{\beta, \beta^I, \beta^D, \beta^S\}$$
Here, β^I, β^D and β^S are $(h+1)$-edited suffixes, obtained by performing exactly one edit on β w.r.t. α as follows: Let $z = |\mathsf{LCP}(\alpha, \beta)|$ and σ be the $(z+1)$th character of α, then
   - β^I is obtained by *inserting* the character σ in β after the zth character.
   - β^D is obtained by *deleting* the $(z+1)$th character of β.
   - β^S is obtained by *substituting* the $(z+1)$th character of β by σ.

   See Fig. 1 for an illustration. We now prove that the parts created in the above manner satisfy the desired properties.

3.1 Correctness Proof (via Mathematical Induction)

All three properties hold true for $k = 0$ (base case). Assuming they are true for all values of k up to h, we now prove it for $h + 1$. From our construction procedure, $|\mathcal{P}_t^{h+1}| = 1 + 4(|Q| - 1) < 4|\mathcal{P}_f^h| = 4m$. Therefore, the maximum size of a part can be bounded by $\max_t |\mathcal{P}_t^{h+1}| < 4 \max_f |\mathcal{P}_f^h| = O(n)$. The total size of all pairs derived from $\mathcal{P}_f^h$ is $4 \sum_{u \text{ is light}} \text{subtree-size}(u) \leq 4m \log m$. Therefore, the total size of all parts in U_{h+1} is

$$\sum_t |\mathcal{P}_t^{h+1}| \leq 4 \sum_f |\mathcal{P}_f^h| \log |\mathcal{P}_f^h| < 4 \Big(\sum_f |\mathcal{P}_f^h| \Big) \Big(\log \sum_f |\mathcal{P}_f^h| \Big) = O(n \log^{h+1} n)$$

Next we prove the existence of an $(i, j)_{h+1}$-$\mathsf{maxpair}$ in at least one part, say $\mathcal{P}_t^{h+1}$. Without loss of generality, assume $\{\mathsf{X}'_i, \mathsf{Y}'_j\}$ is an $(i, j)_h$-$\mathsf{maxpair}$ and is a subset of $\mathcal{P}_f^h$. Then,

$$|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| = |\mathsf{LCP}_h(\mathsf{X}_i, \mathsf{Y}_j)| \quad \text{and} \quad |\Delta(\mathsf{X}'_i) \cup \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j) \leq h$$

[4] The child with the largest number of leaves in its subtree (ties broken arbitrarily) among its siblings.

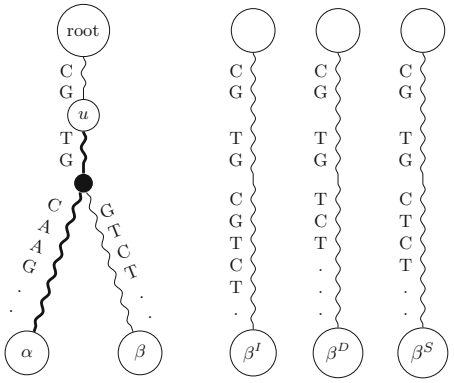

Fig. 1. Illustrates an edit operation along a suffix β, at the point where it diverges from a heavy path (shown as a thick wavy line). For insertion (β^I) and substitution (β^S), the modification is made to conform to the next character along the heavy path.

Let w be the lowest common ancestor of the leaves corresponding to X'_i and Y'_j in the trie $\mathcal{T}$, $l = |\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|$, σ be the leading character on the outgoing edge from w towards its heavy child, and $\mathcal{P}_t^{h+1}$ be the part created w.r.t. the heavy path through w. We now prove there exists an $(i,j)_{h+1}$-maxpair $\subseteq \mathcal{P}_t^{h+1}$. We have the following cases.

Case 1: At w, both X'_i *and* Y'_j *diverge from the heavy path through* w. Then the following edited suffixes, in addition to X'_i and Y'_j, are in $\mathcal{P}_f^{h+1}$. Let

- X''_i be the edited suffix obtained by deleting the $(l+1)$th character of X'_i.
- Y''_j be the edited suffix obtained by deleting the $(l+1)$th character of Y'_j.
- X'''_i be the edited suffix obtained by substituting the $(l+1)$th character of X'_i by σ.
- Y'''_j be the edited suffix obtained by substituting the $(l+1)$th character of Y'_j by σ.

It can be easily verified that one of the following subsets of $\mathcal{P}_t^{h+1}$ is an $(i,j)_{h+1}$-maxpair: $\{\mathsf{X}'_i, \mathsf{Y}''_j\}, \{\mathsf{X}''_i, \mathsf{Y}'_j\}, \{\mathsf{X}'''_i, \mathsf{Y}'''_j\}$.

Case 2: At w, exactly one among X'_i *and* Y'_j *diverges from the heavy path.* Without loss of generality, assume the diverging suffix is X'_i. Then the following edited suffixes, in addition to X'_i and Y'_j, are in $\mathcal{P}_f^{h+1}$. Let

- X''_i be the edited suffix obtained by deleting the $(l+1)$th character of X'_i.
- X'''_i be the edited suffix obtained by substituting the $(l+1)$th character of X'_i by σ.
- X''''_i be the edited suffix obtained by inserting σ in X'_i after l characters.

Here also, it can be easily verified that one of the following subset of $\mathcal{P}_t^{h+1}$ is an $(i,j)_{h+1}$-maxpair: $\{\mathsf{X}''_i, \mathsf{Y}'_j\}, \{\mathsf{X}'''_i, \mathsf{Y}'_j\}, \{\mathsf{X}''''_i, \mathsf{Y}'_j\}$. This completes the correctness proof.

3.2 Time and Space Complexity Analysis

First, we consider the recursive step of creating parts out of $\mathcal{P}_f^h$. The trie $\mathcal{T}$ can be constructed in $O(m \log m)$ time (recall $m = |\mathcal{P}_f^h|$) with the following steps.

1. First, sort the edited suffixes in $\mathcal{P}_f^h$ in time $O(m \log m)$ via merge sorting. Note that any two k-edited suffixes can be compared in $O(k)$ time (refer to Lemma 2).
2. Then, compute the LCP between every consecutive pair of edited suffixes in the sorted list and build the trie $\mathcal{T}$ using standard techniques from the suffix tree construction algorithms [14]. This step takes only $O(m)$ time.

Note that the part corresponding to each light node u can be obtained in time proportional to the subtree size of u. Therefore, the time for deriving parts from $\mathcal{P}_f^h$ is $O(m \log m)$. In other words, parts of U_{h+1} can be obtained from parts of U_h in $O(\log n \sum_f |\mathcal{P}_f^h|)$ time for $h = 0, 1, 2, \ldots, k-1$. Total time is $\log n \sum_{h=0}^{k-1} |\mathcal{P}_f^h| = O(n \log^k n)$.

The parts can be created (and processed) one at a time by keeping exactly one partition in each U_h for $h = 0, 1, 2, \ldots, k$. Therefore, the working space is $\sum_{h=0}^{k} \max_t |\mathcal{P}_t^h| = O(n)$.

Lemma 3. *The universe U_k can be created in parts in $O(n \log^k n)$ time using $O(n)$ space.*

3.3 Obtaining the Parts of U_k with Its Elements Sorted

We now present a more careful implementation of the above steps, so that the parts can be generated with their elements in sorted order without incurring additional comparison sorting costs. Specifically, we show how to process a light node u in $\mathcal{T}$ (the trie over all edited suffixes in $\mathcal{P}_f^h$) and construct the corresponding part $\mathcal{P}_t^{h+1}$ in U_{h+1} with its elements sorted. We use the classic result that two sorted lists of sizes p and q ($q \leq p$) in the form of balanced binary search trees (BSTs) can be merged using $O(q \log(p/q))$ comparisons [7]. Throughout the execution of our algorithm, we maintain edited suffixes in the form of a BST. Key steps are below.

1. Initialize BST with exactly one element α.
2. Visit the heavy internal nodes on the heavy path through u in a bottom up fashion. For each light child w of a heavy node v on the path (let l be the string depth of v), merge BST with BST_w, BST_w^I, BST_w^D and BST_w^S. Here,
   - BST_w is the set of all strings corresponding to the leaves in the subtree of w.
   - BST_w^I is BST_w after inserting the character $\alpha[l+1]$ after lth position of all strings in it.
   - BST_w^D is BST_w after deleting the $(l+1)$th character of all strings in it.

- BST_w^S is BST_w after replacing the $(l+1)$th character by $\alpha[l+1]$ for all its strings.

Note that BST_w can be created from $\mathcal{T}$ in time linear to its size. Since the LCP of any two strings in BST_w is at least $(l+1)$, we can generate BST_w^I, BST_w^D and BST_w^S also in time linear to their size. Therefore, the merging can be performed in time $O(size(w)\log(size(v)/size(w)))$ via fast merging.

The correctness is ensured as we are implementing the same algorithm described earlier. The time for processing all light nodes in $\mathcal{T}$ is the sum of $size(\cdot) \times \log(size(parent(\cdot))/size(\cdot))$ over all light nodes. This is the same as the sum of $\log(size(parent(\cdot))/size(\cdot))$ over all light ancestors of all leaves. However, sum of $\log(size(parent(\cdot))/size(\cdot))$ over all nodes on any root to leaf path is $\log(size(root))$. In summary, we have the following.

Lemma 4. *We can generate U_k with its parts sorted in $O(n\log^k n)$ time using $O(n)$ space.*

4 Our Algorithm for Computing the Array L

We can compute $\forall i, L[i] = \max_j |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ with the following procedure. First, initialize all entries in array L to 0 and then, process each part $\mathcal{P}_f^k$ one after another as follows:

$$\forall\ \mathsf{X}'_i, \mathsf{Y}'_j \in \mathcal{P}_f^k \text{ s.t. } |\Delta(\mathsf{X}'_i) \cup \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j) \leq k,$$

$$\text{update } L[i] \leftarrow \max\{L[i], |\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)|\}$$

After processing all the parts of U_k, we have $\max_j |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)| = L[i]$ for all values of i. Correctness follows from the fact that at some point during the execution of the algorithm, we will process a pair $\mathsf{X}_i^*, \mathsf{Y}_d^*$ corresponding an (i,d)-$\mathsf{maxpair}$ with $d = \arg\max_j |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ and update $L[i] \leftarrow |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_d)|$. However, we cannot afford to examine all the pairs.

Our Strategy. $\forall\ h, t \in [0,k]$ and set ϕ, generate all non-empty sets $S(h,t,\phi)$ from $\mathcal{P}_f^k$, such that $S(h,t,\phi) =$

$$\{\mathsf{X}'_i \mid \phi \subseteq \Delta(\mathsf{X}'_i) \textbf{ and } |\Delta(\mathsf{X}'_i)| + \delta(\mathsf{X}'_i) = h\} \cup \{\mathsf{Y}'_j \mid \phi \subseteq \Delta(\mathsf{Y}'_j) \textbf{ and } |\Delta(\mathsf{Y}'_j)| + \delta(\mathsf{Y}'_j) = t\}$$

Observe that $\forall\ \mathsf{X}'_i, \mathsf{Y}'_j \in S(h,t,\phi)$, $|\Delta(\mathsf{X}'_i) \cup \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j)$

$$\begin{aligned} &= |\Delta(\mathsf{X}'_i)| + |\Delta(\mathsf{Y}'_j)| - |\Delta(\mathsf{X}'_i) \cap \Delta(\mathsf{Y}'_j)| + \delta(\mathsf{X}'_i) + \delta(\mathsf{Y}'_j) \\ &= h + t - |\Delta(\mathsf{X}'_i) \cap \Delta(\mathsf{Y}'_j)| \\ &\leq h + t - |\phi| \end{aligned}$$

This in turn implies that $|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| \leq |\mathsf{LCP}_{h+t-|\phi|}(\mathsf{X}_i, \mathsf{Y}_j)|$. Therefore,

$$\forall\ \mathsf{X}'_i, \mathsf{Y}'_j \in S(h,t,\phi) \text{ with } h + t - |\phi| \leq k, |\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| \leq |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$$

Additionally, $\forall\ (i,j)$ pairs, there exists an $(i,j)_k$-$\mathsf{maxpair}$, say $\{\mathsf{X}''_i, \mathsf{Y}''_j\}$ and an f, such that $\mathsf{X}''_i, \mathsf{Y}''_j \in \mathcal{P}_f^k$. In other words, there exists a non-empty set $S(a,b,\mu)$, such that $\mathsf{X}''_i, \mathsf{Y}''_j \in S(a,b,\mu)$. Specifically, $a = |\Delta(\mathsf{X}''_i)| + \delta(\mathsf{X}''_i), b = |\Delta(\mathsf{Y}''_j)| + \delta(\mathsf{Y}''_j)$ and $\mu =$

$\Delta(\mathsf{X}''_i) \cap \Delta(\mathsf{Y}''_j)$. Since $\{\mathsf{X}''_i, \mathsf{Y}''_j\}$ is an $(i,j)_k$-maxpair, $|\mathsf{LCP}(\mathsf{X}''_i, \mathsf{Y}''_j)| = |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ and $a + b - |\mu| \leq k$. Therefore,

$$|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)| = \max\{|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| \mid \mathsf{X}'_i, \mathsf{Y}'_j \in S(h,t,\phi) \textbf{ and } (h + t - |\phi| \leq k)\}$$

$$L[i] = \max_j\{|\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_j)| \mid \mathsf{X}'_i, \mathsf{Y}'_j \in S(h,t,\phi) \textbf{ and } (h + t - |\phi| \leq k)\}$$

Note that there is no $|\mathsf{LCP}_k(\cdot,\cdot)|$ in the above equation. Equivalently, we have a new definition for $L[\cdot]$ using *exact matching over k-edited suffixes*. Therefore, the computation of L is straightforward.

Proposed Algorithm. Initialize $L[i] \leftarrow 0, \forall i$. Then, $\forall\ h,t \in [0,k]$ and set ϕ with $h + t - |\phi| \leq k$, process $S(h,t,\phi)$ as follows: sort all of its strings, and visit the strings in both ascending and descending order. For each X'_i visited, update $L[i] \leftarrow \max\{L[i], |\mathsf{LCP}(\mathsf{X}'_i, \mathsf{Y}'_l)|\}$, where Y'_l is the last visited k-edited suffix of Y. Correctness is immediate from the above discussions.

Space and Time Analysis. Since we process the parts $\mathcal{P}^k_f$ one after another, space is $O(n)$. W.r.t. time complexity, note that each $S(\cdot,\cdot,\cdot)$ can be processed in time linear plus the time for sorting its strings, which is $O(|S(\cdot,\cdot,\cdot)| \log(|S(\cdot,\cdot,\cdot)|))$ using Lemma 2. The sum of sizes of all $S(\cdot,\cdot,\cdot)$ generated from a particular $\mathcal{P}^k_f$ is at most $k \times 2^k \times |\mathcal{P}^k_f|$ i.e., $\sum |S(\cdot,\cdot,\cdot)| = O(n \log^k n)$. Total time is $\sum |S(\cdot,\cdot,\cdot)| \log(|S(\cdot,\cdot,\cdot)|) = O(n \log^{k+1} n)$.

To shave off an additional $\log n$ factor from the time complexity, we replace the merge sorting by integer sorting. Specifically, we generate all $\mathcal{P}^k_f$'s with their elements sorted using Lemma 4. We then process $\mathcal{P}^k_f$s after replacing each edited suffix within $\mathcal{P}^k_f$ by its lexicographic rank in $\mathcal{P}^k_f$. Essentially, we replace all string comparison tasks by integer comparison. Therefore, the main task now is the sorting of several sets of integers of total size $O(n \log^k n)$ and maximum size $O(n)$. On sets of size $\Theta(n)$, we employ counting sort. To sort smaller sets, we combine several of them up to a total size of $\Theta(n)$. Then, a counting sort is performed, followed by a stable sort with the id associated with the set in which each integer belongs to as the key. By scanning the output in linear time, we can segregate the individual sorted lists. The time in both cases is constant per element. By combining this with Lemma 4, we obtain the result in Theorem 1.

5 Solving Approximate Sequence Matching Problems

5.1 Computing the *k*-edit Average Common Substring

The computation of ACS_k from L is straightforward. We now demonstrate the applicability of our algorithmic framework to three other important problems. The general strategy is to begin with an order-0 universe U_0 with one part: the set of all suffixes of all input sequences. Then create U_k, in parts and process them one by one, using problem specific steps. In all three cases, the correctness (deferred to full version) can be obtained via straightforward adaptations of the correctness proof of Theorem 1.

5.2 Our Algorithm for the Mapping Problem

Let $\{R_1, R_2, .., R_m\}$ be the set of input reads and G be the reference genome. Our task is to report all (i, j) pairs, s.t. the edit distance between R_i and $G[j..(j + \ell - 1)]$ is $\leq k$, where ℓ is the read length. We use R'_i (resp., G'_j) for a k-edited copy of R_i (resp., G_j). Let $S(h, t, \phi)$ w.r.t. $\mathcal{P}_f^k$ be $\{R'_i \in \mathcal{P}_f^k \mid |\Delta(R'_i)| + \delta(R'_i) = h$ **and** $\phi \subseteq \Delta(R'_i)\} \cup \{G'_j \in \mathcal{P}_f^k \mid |\Delta(G'_j)| + \delta(G'_j) = t$ **and** $\phi \subseteq \Delta(G'_j)\}$. Then, $\forall\ h, t \in [0, k]$ and set ϕ with $h+t-|\phi| \leq k$, process all $S(h, t, \phi)$ as follows: $\forall\ R'_i, G'_j \in S(h, t, \phi)$ s.t. $|\mathsf{LCP}(R'_i, G'_j)| \geq \ell$, report (i, j). The following correctness argument can be easily verified: we report a pair (i, j) iff it is a valid output.

The processing of an $S(\cdot, \cdot, \cdot)$ is now an exact matching task. It can be implemented in time linear to its size and the number of pairs reported. Therefore, time over all $S(\cdot, \cdot, \cdot)$ is $O(n \log^k n)$ plus the total number of pairs reported. Note that our algorithm might report the same pair multiple times, but not more than $O(\log^k n)$ times, because for any (i, j) pair, the number of parts containing an R'_i and a G'_j is $O(\log^k n)$ (follows from our construction). Therefore total time is $O((n + occ) \log^k n)$.

5.3 All-Pair Maximal k-edit Common Substrings

Let $\{R_1, R_2, .., R_m\}$ be the set of input reads. Let $R_{i,x}$ denotes the xth longest suffix of R_i and let $R'_{i,x}$ denotes a k-edited copy of $R_{i,x}$. Our task is to report all tuples (i, x, j, y), s.t. $|\mathsf{LCP}_k(R_{i,x}, R_{j,y})| \geq \tau$, $i \neq j$ and $R_i[x - 1] \neq R_j[y - 1]$. Let $S(h, t, \phi)$ w.r.t. a part $\mathcal{P}_f^k$ is the union of the following two sets.

$$\{R'_{i,x} \in \mathcal{P}_f^k \mid |\Delta(R'_{i,x})| + \delta(R'_{i,x}) = h \textbf{ and } \phi \subseteq \Delta(R'_{i,x})\}$$

$$\{R'_{j,y} \in \mathcal{P}_f^k \mid |\Delta(R'_{j,y})| + \delta(R'_{j,y}) = t \textbf{ and } \phi \subseteq \Delta(R'_{j,y})\}$$

Then, $\forall\ h, t \in [0, k]$ and set ϕ with $h + t - |\phi| \leq k$, process $S(h, t, \phi)$ as follows: $\forall\ R'_{i,x}, R'_{j,y} \in S(h, t, \phi)$ s.t. $|\mathsf{LCP}(R'_{i,x}, R'_{j,y})| \geq \tau, |\Delta(R'_{i,x})| + \delta(R'_{i,x}) = h, |\Delta(R'_{j,y})| + \delta(R'_{j,y}) = t$, $R_i[x - 1] \neq R_j[y - 1]$ and $i \neq j$, report (i, x, j, y). This (exact matching) task can be easily implemented in time linear to $|S(h, t, \phi)|$ and the number of tuples generated using standard techniques (details deferred to full version). Also, we report a tuple iff it is a valid output and we report one only $O(\log^k n)$ times. Hence the total run time is $O((n + occ) \log^k n)$.

5.4 All-Pair Maximal k-edit Suffix/Prefix Overlaps

Borrowing from the terminologies defined in Sect. 5.3, the task here is to report all tuples (i, j, y), s.t. $i \neq j$ and $|\mathsf{LCP}_k(R_i, R_{j,y})| \geq (\ell - y + 1) \geq \tau$. To do so, we process all $S(h, t, \phi)$ with $h+t-|\phi| \leq k$ as follows: $\forall\ R'_i, R'_{j,y} \in S(h, t, \phi)$ s.t. $|\mathsf{LCP}(R'_i, R'_{j,y})| \geq \ell - y + 1 \geq \tau, |\Delta(R'_i)| + \delta(R'_i) = h, |\Delta(R'_{j,y})| + \delta(R'_{j,y}) = t$ and $i \neq j$, report (i, j, y). Again, this is an exact matching task, which can be easily implemented in time linear to $|S(h, t, \phi)|$ and the number of tuples generated using standard techniques (details deferred to full version). Also, we report a tuple iff it is a valid output and we report one only $O(\log^k n)$ times across all $S(\cdot, \cdot, \cdot)$'s, yielding $O((n + occ) \log^k n)$ total time.

Acknowledgments. This research is supported in part by the U.S. National Science Foundation under CCF-1704552 and CCF-1703489.

References

1. Abboud, A., Williams, R., Yu, H.: More applications of the polynomial method to algorithm design. In: Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 218–230 (2015)
2. Abboud, A., Williams, V.V., Weimann, O.: Consequences of faster alignment of sequences. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 39–51. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43948-7_4
3. Aluru, S., Apostolico, A., Thankachan, S.V.: Efficient alignment free sequence comparison with bounded mismatches. In: Przytycka, T.M. (ed.) RECOMB 2015. LNCS, vol. 9029, pp. 1–12. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16706-0_1
4. Apostolico, A.: Maximal words in sequence comparisons based on subword composition. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Algorithms and Applications. LNCS, vol. 6060, pp. 34–44. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12476-1_2
5. Apostolico, A., Guerra, C., Landau, G.M., Pizzi, C.: Sequence similarity measures based on bounded hamming distance. Theoret. Comput. Sci. **638**, 76–90 (2016)
6. Bonham-Carter, O., Steele, J., Bastola, D.: Alignment-free genetic sequence comparisons: a review of recent approaches by word analysis. Briefings Bioinform. **15**(6), 890–905 (2013)
7. Brown, M.R., Tarjan, R.E.: A fast merging algorithm. J. ACM **26**(2), 211–226 (1979)
8. Burkhardt, S., Kärkkäinen, J.: Better filtering with gapped q-grams. Fundam. Inform. **56**(1–2), 51–70 (2003)
9. Burstein, D., Ulitsky, I., Tuller, T., Chor, B.: Information theoretic approaches to whole genome phylogenies. In: Miyano, S., Mesirov, J., Kasif, S., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2005. LNCS, vol. 3500, pp. 283–295. Springer, Heidelberg (2005). https://doi.org/10.1007/11415770_22
10. Chang, G., Wang, T.: Phylogenetic analysis of protein sequences based on distribution of length about common substring. Protein J. **30**(3), 167–172 (2011)
11. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of the 36th Annual ACM Symposium on Theory of computing (STOC), pp. 91–100. ACM (2004)
12. Comin, M., Verzotto, D.: Alignment-free phylogeny of whole genomes using underlying subwords. Algorithms Mol. Biol. **7**(1), 1 (2012)
13. Domazet-Lošo, M., Haubold, B.: Efficient estimation of pairwise distances between genomes. Bioinformatics **25**(24), 3221–3227 (2009)
14. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
15. Guyon, F., Brochier-Armanet, C., Guénoche, A.: Comparison of alignment free string distances for complete genome phylogeny. Adv. Data Anal. Classif. **3**(2), 95–108 (2009)
16. Kucherov, G., Tsur, D.: Improved filters for the approximate suffix-prefix overlap problem. In: Moura, E., Crochemore, M. (eds.) SPIRE 2014. LNCS, vol. 8799, pp. 139–148. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11918-2_14
17. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**(4), 357–359 (2012)

18. Leimeister, C.-A., Morgenstern, B.: kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. Bioinformatics **30**(14), 2000–2008 (2014)
19. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
20. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Briefings Bioinform. **11**(5), 473–483 (2010)
21. Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)
22. 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
23. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM (JACM) **23**(2), 262–272 (1976)
24. Pizzi, C.: A filtering approach for alignment-free biosequences comparison with mismatches. In: Pop, M., Touzet, H. (eds.) WABI 2015. LNCS, vol. 9289, pp. 231–242. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48221-6_17
25. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Res. **22**(3), 549–556 (2012)
26. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
27. 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)
28. Thankachan, S.V., Chockalingam, S.P., Liu, Y., Apostolico, A., Aluru, S.: ALFRED: a practical method for alignment-free distance computation. J. Comput. Biol. **23**(6), 452–460 (2016)
29. Thankachan, S.V., Chockalingam, S.P., Liu, Y., Krishnan, A., Aluru, S.: A greedy alignment-free distance estimator for phylogenetic inference. In: Proceedings of 5th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS) (2015)
30. Välimäki, N., Ladra, S., Mäkinen, V.: Approximate all-pairs suffix/prefix overlaps. Inf. Comput. **213**, 49–58 (2012)
31. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (SWAT), pp. 1–11 (1973)