k-Approximate Quasiperiodicity under Hamming and Edit Distance

Aleksander Kędzierski
Institute of Informatics, University of Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
aa.kedzierski2@uw.edu.pl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
jrad@mimuw.edu.pl

Abstract

Quasiperiodicity in strings was introduced almost 30 years ago as an extension of string periodicity. The basic notions of quasiperiodicity are cover and seed. A cover of a text T is a string whose occurrences in T cover all positions of T. A seed of text T is a cover of a superstring of T. In various applications exact quasiperiodicity is still not sufficient due to the presence of errors. We consider approximate notions of quasiperiodicity, for which we allow approximate occurrences in T with a small Hamming, Levenshtein or weighted edit distance.

In previous work Sip et al. (2002) and Christodoulakis et al. (2005) showed that computing approximate covers and seeds, respectively, under weighted edit distance is NP-hard. They, therefore, considered restricted approximate covers and seeds which need to be factors of the original string T and presented polynomial-time algorithms for computing them. Further algorithms, considering approximate occurrences with Hamming distance bounded by k, were given in several contributions by Guth et al. They also studied relaxed approximate quasiperiods that do not need to cover all positions of T.

In case of large data the exponents in polynomial time complexity play a crucial role. We present more efficient algorithms for computing restricted approximate covers and seeds. In particular, we improve upon the complexities of many of the aforementioned algorithms, also for relaxed quasiperiods. Our solutions are especially efficient if the number (or total cost) of allowed errors is bounded. We also show NP-hardness of computing non-restricted approximate covers and seeds under Hamming distance.

Approximate covers were studied in three recent contributions at CPM over the last three years. However, these works consider a different definition of an approximate cover of T, that is, the shortest exact cover of a string T' with the smallest Hamming distance from T.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases approximate cover, approximate seed, enhanced cover, Hamming distance, edit distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.18

Related Version `https://arxiv.org/abs/2005.06329`

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

1 Introduction

Quasiperiodicity was introduced as an extension of periodicity [6]. Its aim is to capture repetitive structure of strings that do not have an exact period. The basic notions of quasiperiodicity are cover (also called quasiperiod) and seed. A *cover* of a string T is a string C whose occurrences cover all positions of T. A *seed* of string T is a cover of a superstring

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).
Editors: Inge Li Gørtz and Oren Weimann; Article No. 18; pp. 18:1–18:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of T. Covers and seeds were first considered in [7] and [21], respectively, and linear-time algorithms computing them are known; see [9, 21, 29, 30, 31] and [24].

A cover is necessarily a *border*, that is, a prefix and a suffix of the string. A seed C of T covers all positions of T by its occurrences or by left- or right-overhangs, that is, by suffixes of C being prefixes of T and prefixes of C being suffixes of T. In order to avoid extreme cases one usually assumes that covers C of T need to satisfy $|C| < |T|$ and seeds C need to satisfy $2|C| \leq |T|$ (so a seed needs to be a factor of T). Seeds, unlike covers, preserve an important property of periods that if T has a period or a seed, then every (sufficiently long) factor of T has the same period or seed, respectively.

The classic notions of quasiperiodicity may not capture repetitive structure of strings in practical settings; it was also confirmed by a recent experimental study [12]. In order to tackle this problem, further types of quasiperiodicity were studied that require that only a certain number of positions in a string are covered. This way notions of enhanced cover, partial cover and partial seed were introduced. A *partial cover* and *partial seed* are required to cover a given number of positions of a string, where for the partial seed overhangs are allowed, and an *enhanced cover* is a partial cover with an additional requirement of being a border of the string. $\mathcal{O}(n \log n)$-time algorithms for computing shortest partial covers and seeds were shown in [26] and [25], respectively, whereas a linear-time algorithm for computing a proper enhanced cover that covers the maximum number of positions in T was presented (among other variations of the problem) in [14].

Further study has lead to *approximate quasiperiodicity* in which approximate occurrences of a quasiperiod are allowed. In particular, Hamming, Levenshtein and weighted edit distance were considered. A *k-approximate cover* of string T is a string C whose approximate occurrences with distance at most k cover T. Similarly one can define a *k-approximate seed*, allowing overhangs. These notions were introduced by Sip et al. [33] and Christodoulakis et al. [10], respectively, who showed that the problem of checking if a string T has a k-approximate cover and k-approximate seed, respectively, for a given k is NP-complete under weighted edit distance. (Their proof used arbitrary integer weights and a constant-sized – 12 letters in the case of approximate seeds – alphabet.) Therefore, they considered a *restricted* version of the problem in which the approximate cover or seed is required to be a factor of T. Formally, the problem is to compute, for every factor of T, the smallest k for which it is a k-approximate cover or seed of T. For this version of the problem, they presented an $\mathcal{O}(n^3)$-time algorithm for the Hamming distance and an $\mathcal{O}(n^4)$-time algorithm for the edit distance[1]. The same problems under Hamming distance were considered by Guth et al. [19] and Guth and Melichar [18]. They studied a *k-restricted* version of the problems, in which we are only interested in factors of T being ℓ-approximate covers or seeds for $\ell \leq k$, and developed $\mathcal{O}(n^3(|\Sigma| + k))$-time and $\mathcal{O}(n^3|\Sigma|k)$-time automata-based algorithms for k-restricted approximate covers and seeds, respectively. Experimental evaluation of these algorithms was performed by Guth [16].

Recently, Guth [17] extended this study to *k-approximate restricted enhanced covers* under Hamming distance. In this problem, we search for a border of T whose k-approximate occurrences cover the maximum number of text positions. In another variant of the problem, which one could see as approximate partial cover problem, we only require the approximate

[1] In fact, they consider *relative* Hamming and Levenshtein distances which are inversely proportional to the length of the candidate factor and seek for an approximate cover/seed that minimizes such distance. However, their algorithms actually compute the minimum distance k for every factor of T under the standard distance definitions.

enhanced cover to be a k-approximate border of T, but still to be a factor of T. Guth [17] proposed $\mathcal{O}(n^2)$-time and $\mathcal{O}(n^3(|\Sigma|+k))$-time algorithms for the two respective variants.

We improve upon previous results on restricted approximate quasiperiodicity. We introduce a general notion of *k-coverage* of a string S in a string T, defined as the number of positions in T that are covered by k-approximate occurrences of S. Efficient algorithms computing the k-coverage for factors of T are presented. We also show NP-hardness for non-restricted approximate covers and seeds under the Hamming distance. A detailed list of our results is as follows.

1. The Hamming k-coverage for every prefix and for every factor of a string of length n can be computed in $\mathcal{O}(nk^{2/3}\log^{1/3} n \log k)$ time (for a string over an integer alphabet) and $\mathcal{O}(n^2)$ time, respectively. (See Section 3.)
   With this result we obtain algorithms with the same time complexities for the two versions of k-approximate restricted enhanced covers that were proposed by Guth [17] and an $\mathcal{O}(n^2k)$-time algorithm computing k-restricted approximate covers and seeds. Our algorithm for prefixes actually works in linear time assuming that a k-mismatch version of the PREF table [11] is given. Thus, as a by-product, for $k=0$, we obtain an alternative linear-time algorithm for computing all (exact) enhanced covers of a string. (A different linear-time algorithm for this problem was given in [14]).
   The complexities come from using tools of Kaplan et al. [22] and Flouri et al. [13], respectively.
2. The k-coverage under Levenshtein distance and weighted edit distance for every factor of a string of length n can be computed in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^3\sqrt{n\log n})$ time, respectively. (See Section 4.)
   We also show in Section 4 how our approach can be used to compute restricted approximate covers and seeds under weighted edit distance in $\mathcal{O}(n^3\sqrt{n\log n})$ time, thus improving upon the previous $\mathcal{O}(n^4)$-time algorithms of Sip et al. [33] and Christodoulakis et al. [10]. Our algorithm for Levenshtein distance uses incremental string comparison [27].
3. Under Hamming distance, it is NP-hard to check if a given string of length n has a k-approximate cover or a k-approximate seed of a given length c. This statement holds even for strings over a binary alphabet. (See Section 5.)
   This result extends the previous proofs of Sip et al. [33] and Christodoulakis et al. [10] which worked for the weighted edit distance.

A different notion of approximate cover, which we do not consider in this work, was recently studied in [1, 2, 3, 4, 5]. This work assumed that the string T may not have a cover, but it is at a small Hamming distance from a string T' that has a proper cover. They defined an approximate cover of T as the shortest cover of a string T' that is closest to T under Hamming distance. Interestingly, this problem was also shown to be NP-hard [2] and an $\mathcal{O}(n^4)$-time algorithm was developed for it in the restricted case that the approximate cover is a factor of the string T [4]. Our work can be viewed as complementary to this study as "the natural definition of an approximate repetition is not clear" [4].

2 Preliminaries

We consider strings over an alphabet Σ. The empty string is denoted by ε. For a string T, by $|T|$ we denote its length and by $T[0],\ldots,T[|T|-1]$ its subsequent letters. By $T[i,j]$ we denote the string $T[i]\ldots T[j]$ which we call a *factor* of T. If $i=0$, it is a *prefix* of T, and if $j=|T|-1$, it is a *suffix* of T. A string that is both a prefix and a suffix of T is called

a *border* of T. For a string $T = XY$ such that $|X| = b$, by $\mathsf{rot}_b(T)$ we denote YX, called a *cyclic shift* of T.

For equal-length strings U and V, by $Ham(U, V)$ we denote their *Hamming distance*, that is, the number of positions where they do not match. For strings U and V, by $ed(U, V)$ we denote their *edit distance*, that is, the minimum cost of edit operations (insertions, deletions, substitutions) that allow to transform U to V. Here the cost of an edit operation can vary depending both on the type of the operation and on the letters that take part in it. In case that all edit operations have unit cost, the edit distance is also called *Levenshtein distance* and denoted here as $Lev(U, V)$.

For two strings S and T and metric d, we denote by

$$\mathsf{Occ}_k^d(S, T) = \{[i, j] \,:\, d(S, T[i, j]) \leq k\}$$

the set of approximate occurrences of S in T, represented as intervals, under the metric d. We then denote by

$$\mathsf{Covered}_k^d(S, T) = |\bigcup \mathsf{Occ}_k^d(S, T)|$$

the k-*coverage* of S in T. In case of Hamming or Levenshtein distances, $k \leq n$, but for the weighted edit distance k can be arbitrarily large. Moreover, by $\mathsf{StartOcc}_k^d(S, T)$ we denote the set of left endpoints of the intervals in $\mathsf{Occ}_k^d(S, T)$.

▶ **Definition 1.** *Let d be a metric and T be a string. We say that string C, $|C| < |T|$, is a* k-approximate cover *of T under metric d if* $\mathsf{Covered}_k^d(C, T) = |T|$.

We say that string C, $2|C| \leq |T|$, is a k-*approximate seed* of T if it is a k-approximate cover of some string T' whose factor is T. Let $\diamondsuit$ be a wildcard symbol that matches every other symbol of the alphabet. Strings over $\Sigma \cup \{\diamondsuit\}$ are also called *partial words*. In order to compute k-approximate seeds, it suffices to consider k-approximate covers of $\diamondsuit^{|T|}T\diamondsuit^{|T|}$.

The main problems in scope can now be stated as follows.

General k-Approximate Cover/Seed
Input: String T of length n, metric d, integer $c \in \{1, \ldots, n-1\}$ and number k
Output: A string C of length c that is a k-approximate cover/seed of T under d

Prefix/Factor k-Coverage
Input: String T of length n, metric d and number k
Output: For every prefix/factor of T, compute its k-coverage under d

Restricted Approximate Covers/Seeds
Input: String T of length n and metric d
Output: Compute, for every factor C of T, the smallest k such that C is a k-approximate cover/seed of T under d

2.1 Algorithmic Toolbox for Hamming Distance

For a string T of length n, by $\mathsf{lcp}_k(i, j)$ we denote the length of the longest common prefix with at most k mismatches of the suffixes $T[i, n-1]$ and $T[j, n-1]$. Flouri et al. [13] proposed an $\mathcal{O}(n^2)$-time algorithm to compute the longest common factor of two strings T_1, T_2 with at most k mismatches. Their algorithm actually computes the lengths of the longest common prefixes with at most k mismatches of every two suffixes $T_1[i, |T_1| - 1]$ and $T_2[j, |T_2| - 1]$ and returns the maximum among them. Applied for $T_1 = T_2$, it gives the following result.

▶ **Lemma 2** ([13]). *For a string of length n, values $\mathsf{lcp}_k(i,j)$ for all $i,j=0,\ldots,n-1$ can be computed in $\mathcal{O}(n^2)$ time.*

We also use a table PREF_k such that $\mathsf{PREF}_k[i]=\mathsf{lcp}_k(0,i)$. LCP-queries with mismatches can be answered in $\mathcal{O}(k)$ time after linear-time preprocessing using the kangaroo method [28]. In particular, this allows to compute the PREF_k table in $\mathcal{O}(nk)$ time. Kaplan et al. [22] presented an algorithm that, given a pattern P of length m, a text T of length n over an integer alphabet $\Sigma\subseteq\{1,\ldots,n^{\mathcal{O}(1)}\}$, and an integer k, finds in $\mathcal{O}(nk^{2/3}\log^{1/3}m\log k)$ time for all positions j of T, the index of the k-th mismatch of P with the suffix $T[j,n-1]$. Applied for $P=T$, it gives the following result.

▶ **Lemma 3** ([22]). *The PREF_k table of a string of length n over an integer alphabet can be computed in $\mathcal{O}(nk^{2/3}\log^{1/3}n\log k)$ time.*

We say that strings U and V have a *k-mismatch prefix-suffix of length p* if U has a prefix U' of length p and V has a suffix V' of length p such that $Ham(U',V')\leq k$.

2.2 Algorithmic Toolbox for Edit Distance

For $x,y\in\Sigma$, let $c(x,y)$, $c(\varepsilon,x)$ and $c(x,\varepsilon)$ be the costs of substituting letter x by letter y (equal to 0 if $x=y$), inserting letter x and deleting letter x, respectively. They are usually specified by a penalty matrix c; it implies a metric if certain conditions are satisfied (identity of indiscernibles, symmetry, triangle inequality).

The classic dynamic programming solution to the edit distance problem (see [34]) for strings T_1 and T_2 uses the so-called D-table such that $D[i,j]$ is the edit distance between prefixes $T_1[0,i]$ and $T_2[0,j]$. Initially $D[-1,-1]=0$, $D[i,-1]=D[i-1,-1]+c(T_1[i],\varepsilon)$ for $i\geq 0$ and $D[-1,j]=D[-1,j-1]+c(\varepsilon,T_2[j])$ for $j\geq 0$. For $i,j\geq 0$, $D[i,j]$ can be computed as follows:

$$D[i,j]=\min(D[i-1,j-1]+c(T_1[i],T_2[j]),\ D[i,j-1]+c(\varepsilon,T_2[j]),\ D[i-1,j]+c(T_1[i],\varepsilon)).$$

Given a threshold h on the Levenshtein distance, Landau et al. [27] show how to compute the Levenshtein distance between T_1 and bT_2, for any $b\in\Sigma$, in $\mathcal{O}(h)$ time using previously computed solution for T_1 and T_2 (another solution was given later by Kim and Park [23]). They define an *h-wave* that contains indices of the last value h in diagonals of the D-table. Let $L^h(d)=\max\{i : D[i,i+d]=h\}$. Formally an h-wave is:

$$L^h=[L^h(-h),L^h(-h+1),\ldots,L^h(h-1),L^h(h)].$$

Landau et al. [27] show how to update the h-wave when string T_2 is prepended by a single letter in $\mathcal{O}(h)$ time. This method was introduced to approximate periodicity in [32].

3 Computing k-Coverage under Hamming Distance

Let T be a string of length n and assume that its PREF_k table is given. We will show a linear-time algorithm for computing the k-coverage of every prefix of T under the Hamming distance.

In the algorithm we consider all prefix lengths $\ell=1,\ldots,n$. At each step of the algorithm, a linked list $\mathcal{L}$ is stored that contains all positions i such that $\mathsf{PREF}_k[i]\geq\ell$ and a sentinel value n, in an increasing order. The list is stored together with a table $A(\mathcal{L})[0..n-1]$ such that $A(\mathcal{L})[i]$ is a link to the occurrence of i in $\mathcal{L}$ or **nil** if $i\notin\mathcal{L}$. It can be used to access and remove a given element of $\mathcal{L}$ in $\mathcal{O}(1)$ time. Before the start of the algorithm, $\mathcal{L}$ contains all numbers $0,\ldots,n$.

If $i \in \mathcal{L}$ and j is the successor of i in $\mathcal{L}$, then the approximate occurrence of $T[0, \ell-1]$ at position i accounts for $\min(\ell, j-i)$ positions that are covered in T. A pair of adjacent elements $i < j$ in $\mathcal{L}$ is called *overlapping* if $j - i < \ell$ and *non-overlapping* otherwise. Hence, each non-overlapping adjacent pair adds the same amount to the number of covered positions.

All pairs of adjacent elements of $\mathcal{L}$ are partitioned in two data structures, $\mathcal{D}_o$ and $\mathcal{D}_{no}$, that store overlapping and non-overlapping pairs, respectively. Data structure $\mathcal{D}_{no}$ stores non-overlapping pairs (i, j) in buckets that correspond to $j - i$, in a table $B(\mathcal{D}_{no})$ indexed from 1 to n. It also stores a table $A(\mathcal{D}_{no})$ indexed 0 through $n-1$ such that $A(\mathcal{D}_{no})[i]$ points to the location of (i, j) in its bucket, provided that such a pair exists for some j, or **nil** otherwise. Finally, it remembers the number $num(\mathcal{D}_{no})$ of stored adjacent pairs. $\mathcal{D}_o$ does not store the overlapping adjacent pairs (i, j) explicitly, just the sum of values $j - i$, as $sum(\mathcal{D}_o)$. Then

$$\mathsf{Covered}_k^{Ham}(T[0, \ell-1], T) = sum(\mathcal{D}_o) + num(\mathcal{D}_{no}) \cdot \ell. \tag{1}$$

Now we need to describe how the data structures are updated when ℓ is incremented.

In the algorithm we store a table $Q[0..n]$ of buckets containing pairs $(\mathsf{PREF}_k[i], i)$ grouped by the first component. When ℓ changes to $\ell + 1$, the second components of all pairs from $Q[\ell]$ are removed, one by one, from the list $\mathcal{L}$ (using the table $A(\mathcal{L})$).

Let us describe what happens when element q is removed from $\mathcal{L}$. Let q_1 and q_2 be its predecessor and successor in $\mathcal{L}$. (They exist because 0 and n are never removed from $\mathcal{L}$.) Then each of the pairs (q_1, q) and (q, q_2) is removed from the respective data structure $\mathcal{D}_o$ or $\mathcal{D}_{no}$, depending on the difference of elements. Removal of a pair (i, j) from $\mathcal{D}_o$ simply consists in decreasing $sum(\mathcal{D}_o)$ by $j - i$, whereas to remove (i, j) from $\mathcal{D}_{no}$ one needs to remove it from the right bucket (using the table $A(\mathcal{D}_{no})$) and decrement $num(\mathcal{D}_o)$. In the end, the pair (q_1, q_2) is inserted to $\mathcal{D}_o$ or to $\mathcal{D}_{no}$ depending on $q_2 - q_1$. Insertion to $\mathcal{D}_o$ and to $\mathcal{D}_{no}$ is symmetric to deletion.

When ℓ is incremented, non-overlapping pairs (i, j) with $j - i = \ell$ become overlapping. Thus, all pairs from the bucket $B(\mathcal{D}_{no})[\ell]$ are removed from $\mathcal{D}_{no}$ and inserted to $\mathcal{D}_o$.

This concludes the description of operations on the data structures. Correctness of the resulting algorithm follows from (1). We analyze its complexity in the following theorem.

▶ **Theorem 4.** *Let T be a string of length n. Assuming that the PREF_k table for string T is given, the k-coverage of every prefix of T under the Hamming distance can be computed in $\mathcal{O}(n)$ time.*

Proof. There are up to n removals from $\mathcal{L}$. Initially $\mathcal{L}$ contains n adjacent pairs. Every removal from $\mathcal{L}$ introduces one new adjacent pair, so the total number of adjacent pairs that are considered in the algorithm is $2n - 1$. Each adjacent pair is inserted to $\mathcal{D}_o$ or to $\mathcal{D}_{no}$, then it may be moved from $\mathcal{D}_{no}$ to $\mathcal{D}_o$, and finally it is removed from its data structure. In total, $\mathcal{O}(n)$ insertions and deletions are performed on the two data structures, in $\mathcal{O}(1)$ time each. This yields the desired time complexity of the algorithm. ◀

Let us note that in order to compute the k-coverage of all factors of T that start at a given position i, it suffices to use a table $[\mathsf{lcp}_k(i, 0), \ldots, \mathsf{lcp}_k(i, n-1)]$ instead of PREF_k. Together with Lemma 2 this gives the following result.

▶ **Corollary 5.** *Let T be a string of length n. The k-coverage of every factor of T under the Hamming distance can be computed in $\mathcal{O}(n^2)$ time.*

4 Computing k-Coverage under Edit Distance

Let us state an abstract problem that, to some extent, is a generalization of the k-mismatch lcp-queries to the edit distance.

Longest Approximate Prefix Problem
Input: A string T of length n, a metric d and a number k
Output: A table P_k^d such that $P_k^d[a,b,a']$ is the maximum $b' \geq a'-1$ such that $d(T[a,b],T[a',b']) \leq k$ or -1 if no such b' exists.

Having the table P_k^d, one can easily compute the k-coverage of a factor $T[a,b]$ under metric d as:

$$\mathsf{Covered}_k^d(T[a,b],T) = \left|\bigcup_{a'=0}^{n-1}[a',P_k^d[a,b,a']]\right|, \tag{2}$$

where an interval of the form $[a',b']$ for $b'<a'$ is considered to be empty. The size of the union of n intervals can be computed in $\mathcal{O}(n)$ time, which gives $\mathcal{O}(n^3)$ time over all factors.

In Section 4.1 and 4.2 we show how to compute the tables P_k^{Lev} and P_k^{ed} for a given threshold k in $\mathcal{O}(n^3)$ and $\mathcal{O}(n^3\sqrt{n\log n})$ time, respectively. Then in Section 4.3 we apply the techniques of Section 4.2 to obtain an $\mathcal{O}(n^3\sqrt{n\log n})$-time algorithm for computing restricted approximate covers and seeds under the edit distance.

4.1 Longest Approximate Prefix under Levenshtein Distance

Let $H_{i,j}$ be the h-wave for strings $T[i,n-1]$ and $T[j,n-1]$ and $h=k$. Then we can compute P_k^{Lev} with Algorithm 1. The algorithm basically takes the rightmost diagonal of D-table in which the value in row $b-a+1$ is less than or equal to k.

Algorithm 1 Computing P_k^{Lev} table.

```
1  for a' := n − 1 down to 0 do
2      Compute H_{n−1,a'};
3      for a := n − 1 down to 0 do
4          if a < n then
5              Compute H_{a,a'} from H_{a+1,a'};
6          d := k;
7          for b := a to n − 1 do
8              i := b − a + 1;
9              while d ≥ −k and H_{a,a'}(d) < i do
10                 d := d − 1;
11             if d < −k then P_k^{Lev}[a,b,a'] := −1;
12             else P_k^{Lev}[a,b,a'] := a' + i + d;
```

The while-loop can run up to $2k$ times for given a and a'. Computing $H_{n-1,a'}$ takes $\mathcal{O}(k^2)$ time and updating $H_{a,a'}$ takes $\mathcal{O}(k)$ time. It makes the algorithm run in $\mathcal{O}(n^3)$ time. Together with Equation (2) this yields the following result.

▶ **Proposition 6.** *Let T be a string of length n. The k-coverage of every factor of T under the Levenshtein distance can be computed in $\mathcal{O}(n^3)$ time.*

A similar method could be used in case of constant edit operation costs, by applying the work of [20]. In the following section we develop a solution for arbitrary costs.

4.2 Longest Approximate Prefix under Edit Distance

For indices $a, a' \in [0, n]$ we define a table $D_{a,a'}$ such that $D_{a,a'}[b, b']$ is the edit distance between $T[a, b]$ and $T[a', b']$, for $b \in [a-1, n-1]$ and $b' \in [a'-1, n-1]$. For other indices we set $D_{a,a'}[b, b'] = \infty$. The $D_{a,a'}$ table corresponds to the D-table for $T[a, n-1]$ and $T[a', n-1]$ and so it can be computed in $\mathcal{O}(n^2)$ time.

We say that pair (d, b) (Pareto-)dominates pair (d', b') if $(d, b) \neq (d', b')$, $d \leq d'$ and $b \geq b'$. Let us introduce a data structure $L_{a,a'}[b]$ being a table of all among pairs $(D_{a,a'}[b, b'], b')$ that are maximal in this sense (i.e., are not dominated by other pairs), sorted by increasing first component. Using a folklore stack-based algorithm (Algorithm 2), this data structure can be computed from $D_{a,a'}[b, a'-1], \ldots, D_{a,a'}[b, n-1]$ in linear time.

Algorithm 2 Computing $L_{a,a'}[b]$ from $D_{a,a'}[b, \cdot]$.

```
1 Q := empty stack;
2 for b' := a' − 1 to n − 1 do
3     d := D_{a,a'}[b, b'];
4     while Q not empty do
5         (d', x) := top(Q);
6         if d' ≥ d then pop(Q);
7         else break;
8     push(Q, (d, b'));
9 L_{a,a'}[b] := Q;
```

Every multiple of $M = \lfloor \sqrt{n / \log n} \rfloor$ will be called a *special point*. In our algorithm we first compute the following data structures:

(a) all $L_{a,a'}[b]$ lists where a or a' is a special point, for $a, a' \in [0, n-1]$ and $b \in [a-1, n-1]$ (if $a \geq n$ or $a' \geq n$, the list is empty); and

(b) all cells $D_{a,a'}[b, b']$ of all $D_{a,a'}$ tables for $a, a' \in [0, n]$ and $-1 \leq b - a, b' - a' < M - 1$.

Computing part (a) takes $\mathcal{O}(n^4/M) = \mathcal{O}(n^3\sqrt{n \log n})$ time, whereas part (b) can be computed in $\mathcal{O}(n^4/M^2) = \mathcal{O}(n^3 \log n)$ time. The intuition behind this data structure is shown in the following lemma.

▶ **Lemma 7.** *Assume that $b - a \geq M - 1$ or $b' - a' \geq M - 1$. Then there exists a pair of positions c, c' such that the following conditions hold:*

- *$a \leq c \leq b + 1$ and $a' \leq c' \leq b' + 1$, and*
- *$c - a, c' - a' < M$, and*
- *$ed(T[a, b], T[a', b']) = ed(T[a, c-1], T[a', c'-1]) + ed(T[c, b], T[c', b'])$, and*
- *at least one of c, c' is a special point.*

Moreover, if c (c') is the special point, then $c \leq b$ ($c' \leq b'$, respectively).

Proof. By the assumption, at least one of the intervals $[a, b]$ and $[a', b']$ contains a special point. Let $p \in [a, b]$ and $p' \in [a', b']$ be the smallest among them; we have $p - a, p' - a' < M$ provided that p or p' exists, respectively (otherwise p or p' is set to ∞). Let us consider the table $D_{a,a'}$ and how its cell $D_{a,a'}[b, b']$ is computed. We can trace the path of parents in the dynamic programming from $D_{a,a'}[b, b']$ to the origin ($D_{a,a'}[a-1, a'-1]$). Let us traverse this path in the reverse direction until the first dimension of the table reaches p or the second

dimension reaches p'. Say that just before this step we are at $D_{a,a'}[q,q']$. If $q+1=p$ and $q'<p'$, then we set $c=q+1$ and $c'=q'+1$. Indeed $c=p$ is a special point,

$$ed(T[a,b],T[a',b']) = ed(T[a,c-1],T[a',c'-1]) + ed(T[c,b],T[c',b'])$$

and $c-a, c'-a' < M$. Moreover, $q' \in [a'-1,b']$, so $c' \in [a',b'+1]$. The opposite case (that $q'+1=p'$) is symmetric. ◀

If $P_k^{ed}[a,b,a'] - a' < M-1$, then it can be computed using one the $M \times M$ prefix fragments of the $D_{a,a'}$ tables. Otherwise, according to the statement of the lemma, one of the $L_{c,c'}[b]$ lists can be used, where $c-a, c'-a' < M$, as shown in Algorithm 3. The algorithm uses a predecessor operation $Pred(x,L)$ which for a number x and a list $L = L_{c,c'}[b]$ returns the maximal pair whose first component does not exceed x, or (∞,∞) if no such pair exists. This operation can be implemented in $\mathcal{O}(\log n)$ time via binary search.

Algorithm 3 Computing $P_k^{ed}[a,b,a']$.

```
1  res := −1;
2  if b − a < M − 1 then
3      for b' := a' − 1 to a' + M − 2 do
4          if D_{a,a'}[b,b'] ≤ k then
5              res := b';
6  s := a + ((−a) mod M); s' := a' + ((−a') mod M);        // closest special pts
7  foreach (c,c') in ({s} × [a', a' + M − 1]) ∪ ([a, a + M − 1] × {s'}) do
8      (d',b') := Pred(k − D_{a,a'}[c − 1, c' − 1], L_{c,c'}[b]);
9      if d' ≠ ∞ then
10         res := max(res, b');
11 P_k^{ed}[a,b,a'] := res;
```

▶ **Theorem 8.** *Let T be a string of length n. The k-coverage of every factor of T under the edit distance can be computed in $\mathcal{O}(n^3\sqrt{n\log n})$ time.*

Proof. We want to show that Algorithm 3 correctly computes $P_k^{ed}[a,b,a']$. Let us first check that the result $b'=res$ of Algorithm 3 satisfies $D_{a,a'}[b,b'] \le k$. It is clear if the algorithm computes b' in line 5. Otherwise, it is computed in line 10. This means that $L_{c,c'}[b]$ contains a pair $(D_{c,c'}[b,b'],b')$ such that

$$k \;\ge\; D_{c,c'}[b,b'] + D_{a,a'}[c-1,c'-1] \;\ge\; D_{a,a'}[b,b'].$$

Now we show that the returned value res is at least $x = P_k^{ed}[a,b,a']$. If $b-a<M-1$ and $x-a'<M-1$, then the condition in line 4 holds for $b'=x$, so indeed $res \ge x$. Otherwise, the condition of Lemma 7 is satisfied. The lemma implies two positions c,c' such that at least one of them is special and that satisfy additional constraints.

If c is special, then the constraints $a \le c$ and $c-a<M$ imply that $c=s$, as defined in line 6. Additionally, $a' \le c' \le a'+M-1$, so (c,c') will be considered in the loop from line 7. By the lemma and the definition of x, we have

$$D_{c,c'}[b,x] \;=\; D_{a,a'}[b,x] - D_{a,a'}[c-1,c'-1] \;\le\; k - D_{a,a'}[c-1,c'-1]. \tag{3}$$

The list $L_{c,c'}[b]$ either contains the pair $(D_{c,c'}[b,x],x)$, or a pair $(D_{c,c'}[b,x'],x')$ such that $D_{c,c'}[b,x'] \leq D_{c,c'}[b,x]$ and $x' > x$. In the latter case by (3) we would have

$$k \;\geq\; D_{a,a'}[c-1,c'-1] + D_{c,c'}[b,x] \;\geq\; D_{a,a'}[c-1,c'-1] + D_{c,c'}[b,x'] \;\geq\; D_{a,a'}[b,x']$$

and $x' > x$. In both cases the predecessor computed in line 8 returns a value res such that $res \geq x$ and $res \neq \infty$. The case that c' is special admits an analogous argument.

Combining Algorithm 3 with Equation (2), we obtain correctness of the computation.

As for complexity, Algorithm 3 computes $P^{ed}_k[a,b,a']$ in $\mathcal{O}(M\log n) = \mathcal{O}(\sqrt{n\log n})$ time and the pre-computations take $\mathcal{O}(n^3\sqrt{n\log n})$ total time. ◀

4.3 Restricted Approximate Covers and Seeds under Edit Distance

The techniques that were developed in Section 4.2 can be used to improve upon the $\mathcal{O}(n^4)$ time complexity of the algorithms for computing the restricted approximate covers and seeds under the edit distance [10, 33]. We describe our solution only for restricted approximate covers; the solution for restricted approximate seeds follows by considering the text $\diamondsuit^{|T|}T\diamondsuit^{|T|}$.

Let us first note that the techniques from the previous subsection can be used as a black box to solve the problem in scope in $\mathcal{O}(n^3\sqrt{n\log n}\log(nw))$ time, where w is the maximum cost of an edit operation. Indeed, for every factor $T[a,b]$ we binary search for the smallest k for which $T[a,b]$ is a k-approximate cover of T. A given value k is tested by computing the tables $P^{ed}_k[a,b,a']$ for all $a' = 0,\ldots,n-1$ and checking if $\mathsf{Covered}^d_k(T[a,b],T) = n$ using Equation (2).

Now we proceed to a more efficient solution. Same as in the algorithms from [10, 33] we compute, for every factor $T[a,b]$, a table $Q_{a,b}[0..n]$ such that $Q_{a,b}[i]$ is the minimum edit distance threshold k for which $T[a,b]$ is a k-approximate cover of $T[i,n-1]$. In the end, all factors $T[a,b]$ for which $Q_{a,b}[0]$ is minimal need to be reported as restricted approximate covers of T. We will show how, given the data structures (a) and (b) of the previous section, we can compute this table in $\mathcal{O}(nM\log n)$ time.

A dynamic programming algorithm for computing the $Q_{a,b}$ table, similar to the one in [10], is shown in Algorithm 4. Computing $Q_{a,b}$ takes $\mathcal{O}(n^2)$ time provided that all $D_{a,b}$ arrays, of total size $\mathcal{O}(n^4)$, are available. The algorithm considers all possibilities for the approximate occurrence $T[i,j]$ of $T[a,b]$.

Algorithm 4 Computing $Q_{a,b}$ in quadratic time.

```
1 Q_{a,b}[n] := 0;
2 for i := n − 1 down to 0 do
3     Q_{a,b}[i] := ∞;
4     minQ := ∞;
5     for j := i to n − 1 do
6         minQ := min(minQ, Q_{a,b}[j + 1]);          // minQ = min Q_{a,b}[i + 1..j + 1]
7         Q_{a,b}[i] := min(Q_{a,b}[i], max(D_{a,i}[b, j], minQ));
```

During the computation of $Q_{a,b}$, we will compute a data structure for on-line range-minimum queries over the table. We can use the following simple data structure with $\mathcal{O}(n\log n)$ total construction time and $\mathcal{O}(1)$-time queries. For every position i and power of two 2^p, we store as $RM[i,p]$ the minimal value in the table $Q_{a,b}$ on the interval $[i,i+2^p-1]$. When a new value $Q_{a,b}[i]$ is computed, we compute $RM[i,0] = Q_{a,b}[i]$ and $RM[i,p]$ for all $0 < p \leq \log_2(n-i)$ using the formula $RM[i,p] = \min(RM[i,p-1], RM[i+2^{p-1},p-1])$.

Then a range-minimum query over an interval $[i, j]$ of $Q_{a,b}$ can be answered by inspecting up to two cells of the RM table for p such that $2^p \leq j - i + 1 < 2^{p+1}$.

Let us note that the variable $minQ$, which denotes the minimum of a growing segment in the $Q_{a,b}$ table, can only decrease. We would like to make the second argument of max in line 7 non-decreasing for increasing j. The values $ed(T[a, b], T[i, j]) = D_{a,i}[b, j]$ may increase or decrease as j grows. However, it is sufficient to consider only those values of j for which $(D_{a,i}[b, j], j)$ is not (Pareto-)dominated (as in Section 4.2), i.e., the elements of the list $L_{a,i}[b]$. For these values, $D_{a,i}[b, j]$ is indeed increasing for increasing j. The next observation follows from this monotonicity and the monotonicity of $\min Q_{a,b}[i + 1..j + 1]$.

▶ **Observation 9.** *Let $(D_{a,i}[b, j'], j')$ be the first element on the list $L_{a,i}[b]$ such that*

$$\min Q_{a,b}[i + 1..j' + 1] \leq D_{a,i}[b, j'].$$

If j' does not exist, we simply take the last element of $L_{a,i}[b]$. Further let $(D_{a,i}[b, j''], j'')$ be the predecessor of $(D_{a,i}[b, j'], j')$ in $L_{a,i}[b]$ (if it exists). Then $j \in \{j', j''\}$ minimizes the value of the expression $\max(\min Q_{a,b}[i + 1..j + 1], D_{a,i}[b, j])$.

If we had access to the list $L_{a,i}[b]$, we could use binary search to locate the index j' defined in the observation. However, we only store the lists $L_{a,i}[b]$ for a and i such that at least one of them is a special point. We can cope with this issue by separately considering all j such that $j < i + M - 1$ and then performing binary search on every of $\mathcal{O}(M)$ lists $L_{c,c'}[b]$ where $a \leq c < a + M$, $i \leq c' < i + M$ and at least one of c, c' is a special point, just as in Algorithm 3. A pseudocode of the resulting algorithm is given as Algorithm 5.

Algorithm 5 Computing $Q_{a,b}$ in $\mathcal{O}(n\sqrt{n \log n})$ time using pre-computed data structures.

```
1  Q_{a,b}[n] := 0;
2  for i := n − 1 down to 0 do
3      Q_{a,b}[i] := ∞;
4      minQ := ∞;
5      if b − a < M − 1 then
6          for j := i to i + M − 2 do
7              minQ := min(minQ, Q_{a,b}[j + 1]);
8              Q_{a,b}[i] := min(Q_{a,b}[i], max(D_{a,i}[b, j], minQ));
9      s := a + ((−a) mod M); s' := i + ((−i) mod M);
10     foreach (c, c') in ({s} × [i, i + M − 1]) ∪ ([a, a + M − 1] × {s'}) do
11         if L_{c,c'}[b] is empty then continue;
           /* Binary search                                              */
12         (d_{j'}, j') := the first pair in L_{c,c'}[b] such that
             min Q_{a,b}[i + 1..j' + 1] ≤ D_{a,i}[c − 1, c' − 1] + d_{j'} or the last pair;
13         (d_{j''}, j'') := predecessor of (d_{j'}, j') in L_{c,c'}[b] or (d_{j'}, j') if there is none;
14         foreach j in {j', j''} do
15             Q_{a,b}[i] := min(Q_{a,b}[i], max(D_{a,i}[c − 1, c' − 1] + d_j, min Q_{a,b}[i + 1..j + 1]));
```

Let us summarize the complexity of the algorithm. Pre-computation of auxiliary data structures requires $\mathcal{O}(n^3\sqrt{n \log n})$ time. Then for every factor $T[a, b]$ we compute the table $Q_{a,b}$. The data structure for constant-time range-minimum queries over the table costs only additional $\mathcal{O}(n \log n)$ space and computation time. When computing $Q_{a,b}[i]$ using dynamic programming, we may separately consider first $M - 1$ indices j, and then we perform a binary search in $\mathcal{O}(M)$ lists $L_{c,c'}[b]$. In total, the time to compute $Q_{a,b}[i]$ given a, b, i is $\mathcal{O}(M \log n) = \mathcal{O}(\sqrt{n \log n})$.

▶ **Theorem 10.** *Let T be a string of length n. All restricted approximate covers and seeds of T under the edit distance can be computed in $\mathcal{O}(n^3\sqrt{n\log n})$ time.*

The work of [10, 33] on approximate covers and seeds originates from a study of approximate periods [32]. Interestingly, while our algorithm improves upon the algorithms for computing approximate covers and seeds, it does not work for approximate periods.

5 NP-hardness of General Hamming k-Approximate Cover and Seed

We make a reduction from the following problem.

Hamming String Consensus
Input: Strings $S_1, \ldots, S_m$, each of length ℓ, and an integer $k < \ell$
Output: A string S, called consensus string, such that $Ham(S, S_i) \leq k$ for all $i = 1, \ldots, m$

The following fact is known.

▶ **Fact 11** ([15]). *Hamming String Consensus is NP-complete even for the binary alphabet.*

Let strings $S_1, \ldots, S_m$ of length ℓ over the alphabet $\Sigma = \{0, 1\}$ and integer k be an instance of Hamming String Consensus. We introduce a morphism ϕ such that

$$\phi(0) = 0^{2k+4}\,1010\,0^{2k+4}, \quad \phi(1) = 0^{2k+4}\,1011\,0^{2k+4}.$$

We will exploit the following simple property of this morphism.

▶ **Observation 12.** *For every string S, every length-$(2k+4)$ factor of $\phi(S)$ contains at most three ones.*

We set $\gamma_i = 1^{2k+4}\phi(S_i)$ and $T = \gamma_1 \ldots \gamma_m$. Further let $\psi(U)$ be an operation that reverses this encoding, i.e., $\psi(\gamma_i) = S_i$. Formally, it takes as input a string U and outputs $U[4k + 12 - 1]U[2 \cdot (4k + 12) - 1] \ldots U[(\ell - 1)(4k + 12) - 1]$.

▶ **Lemma 13.** *Strings γ_i and γ_j, for any $i, j \in \{1, \ldots, m\}$, have no $2k$-mismatch prefix-suffix of length $p \in \{2k + 4, \ldots, |\gamma_i| - 1\}$.*

Proof. We will show that the prefix U of γ_i of length p and the suffix V of γ_j of length p have at least $2k + 1$ mismatches. Let us note that U starts with 1^{2k+4}. The proof depends on the value $d = |\gamma_i| - p$; we have $1 \leq d \leq |\gamma_i| - 2k - 4$. Let us start with the following observation that can be readily verified.

▶ **Observation 14.** *For $A, B \in \{1010, 1011\}$, the strings $A0^4$ and 0^4B have no 1-mismatch prefix-suffix of length in $\{5, \ldots, 8\}$.*

If $1 \leq d \leq 4$, then U and V have a mismatch at position $2k + 4$ since V starts with $1^{2k+4-d}0$. Moreover, they have at least 2ℓ mismatches by the observation (applied for the prefix-suffix length $d + 4$). In total, $Ham(U, V) \geq 2\ell + 1 \geq 2k + 1$.

If $4 < d < 2k + 4$, then every block 1010 or 1011 in γ_i and in γ_j is matched against a block of zeroes in the other string, which gives at least 4ℓ mismatches. Hence, $Ham(U, V) \geq 4\ell \geq 2k + 1$.

Finally, if $2k + 4 \leq d \leq |\gamma_i| - 2k - 4$, then U starts with 1^{2k+4} and every factor of V of length $2k + 4$ has at most three ones (see Observation 12). Hence, $Ham(U, V) \geq 2k + 1$. ◀

The following lemma gives the reduction.

▶ **Lemma 15.** *If* Hamming String Consensus *for $S_1, \ldots, S_m$ has a positive answer, then the* General k-Approximate Cover *under Hamming distance for T, k, and $c = |\gamma_i|$ returns a k-approximate cover C such that $S = \psi(C)$ is a Hamming consensus string for $S_1, \ldots, S_m$.*

Proof. By Lemma 13, if C is a k-approximate cover of T of length c, then every position $a \in \mathsf{StartOcc}_k^H(C, T)$ satisfies $c \mid a$. Hence, $\mathsf{StartOcc}_k^H(C, T) = \{0, c, 2c, \ldots, (m-1)c\}$.

If Hamming String Consensus for $S_1, \ldots, S_m$ has a positive answer S, then $1^{2k+4}\phi(S)$ is a k-approximate cover of T of length c. Moreover, if T has a k-approximate cover C of length c, then for $S = \psi(C)$ and for each $i = 1, \ldots, m$, we have that

$$Ham(C, T[(i-1)c, ic-1]) \geq Ham(S, S_i),$$

so S is a consensus string for $S_1, \ldots, S_m$. This completes the proof. ◀

Lemma 15 and Fact 11 imply that computing k-approximate covers is NP-hard. Obviously, it is in NP.

▶ **Theorem 16.** *General k-Approximate Cover under the Hamming distance is NP-complete even over a binary alphabet.*

A lemma that is similar to Lemma 15 can be shown for approximate seeds. We leave its technical proof for the full version of the paper. Let $T' = \gamma_1\gamma_1 \ldots \gamma_m 1^{2k+4}\gamma_m 1^{2k+4}$.

▶ **Lemma 17.** *If Hamming String Consensus for $S_1, \ldots, S_m$ has a positive answer, then the General k-Approximate Seed under Hamming distance for T', k, and $c = |\gamma_1| + 2k + 4$ returns a k-approximate seed C such that $S = \psi(C')$ is a Hamming consensus string for $S_1, \ldots, S_m$ for some cyclic shift C' of C.*

▶ **Theorem 18.** *General k-Approximate Seed under the Hamming distance is NP-complete even over a binary alphabet.*

6 Conclusions

We have presented several polynomial-time algorithms for computing restricted approximate covers and seeds and k-coverage under Hamming, Levenshtein and weighted edit distances and shown NP-hardness of non-restricted variants of these problems under the Hamming distance. It is not clear if any of the algorithms are optimal. The only known related conditional lower bound shows hardness of computing the Levenshtein distance of two strings in strongly subquadratic time [8]; however, our algorithms for approximate covers under edit distance work in $\Omega(n^3)$ time. An interesting open problem is if restricted approximate covers or seeds under Hamming distance, as defined in [10, 33], can be computed in $\mathcal{O}(n^{3-\epsilon})$ time, for any $\epsilon > 0$. Here we have shown an efficient solution for k-restricted versions of these problems.

References

1 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 25:1–25:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.25.

2 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? *Algorithmica*, 81(7):2857–2875, 2019. doi:10.1007/s00453-019-00559-8.

3 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 26:1–26:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.26.

4 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. *Theoretical Computer Science*, 793:59–69, 2019. doi:10.1016/j.tcs.2019.05.020.

5 Amihood Amir, Avivit Levy, and Ely Porat. Quasi-periodicity under mismatch errors. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPIcs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.4.

6 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

7 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

8 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

9 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

10 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *Journal of Automata, Languages and Combinatorics*, 10(5/6):609–626, 2005. doi:10.25596/jalc-2005-609.

11 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2003. doi:10.1142/4838.

12 Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *CoRR (accepted to Theoretical Computer Science)*, 2019. arXiv:1909.11336.

13 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015. doi:10.1016/j.ipl.2015.03.006.

14 Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theoretical Computer Science*, 506:102–114, 2013. doi:10.1016/j.tcs.2013.08.013.

15 Moti Frances and Ami Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997. doi:10.1007/s002240000044.

16 Ondřej Guth. *Searching Regularities in Strings using Finite Automata*. PhD thesis, Czech Technical University in Prague, 2014. URL: https://fit.cvut.cz/sites/default/files/PhDThesis_Guth.pdf.

17 Ondřej Guth. On approximate enhanced covers under Hamming distance. *Discrete Applied Mathematics*, 274:67–80, 2020. doi:10.1016/j.dam.2019.01.015.

18 Ondřej Guth and Bořivoj Melichar. Using finite automata approach for searching approximate seeds of strings. In Xu Huang, Sio-Iong Ao, and Oscar Castillo, editors, *Intelligent Automation and Computer Engineering*, pages 347–360, Dordrecht, 2010. Springer Netherlands. doi:10.1007/978-90-481-3517-2_27.

19 Ondřej Guth, Bořivoj Melichar, and Miroslav Balík. Searching all approximate covers and their distance using finite automata. In Peter Vojtás, editor, *Proceedings of the Conference on Theory and Practice of Information Technologies, ITAT 2008*, volume 414 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL: http://ceur-ws.org/Vol-414/paper4.pdf.

20 Heikki Hyyrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *Journal of Discrete Algorithms*, 34:2–17, 2015. doi:10.1016/j.jda.2015.05.007.

21 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. doi:10.1007/BF01955677.

22 Haim Kaplan, Ely Porat, and Nira Shafrir. Finding the position of the k-mismatch and approximate tandem repeats. In Lars Arge and Rusins Freivalds, editors, *Algorithm Theory - SWAT 2006, 10th ScandinavianWorkshop on Algorithm Theory*, volume 4059 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2006. doi:10.1007/11785293_11.

23 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.

24 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):Article 27, April 2020. doi:10.1145/3386369.

25 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient algorithms for shortest partial seeds in words. *Theoretical Computer Science*, 710:139–147, 2018. doi:10.1016/j.tcs.2016.11.035.

26 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. doi:10.1007/s00453-014-9915-3.

27 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

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

29 Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002. doi:10.1007/s00453-001-0062-2.

30 Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

31 Dennis W. G. Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.

32 Jeong Seop Sim, Costas S. Iliopoulos, Kunsoo Park, and William F. Smyth. Approximate periods of strings. *Theoretical Computer Science*, 262(1):557–568, 2001. doi:10.1016/S0304-3975(00)00365-0.

33 Jeong Seop Sim, Kunsoo Park, Sung-Ryul Kim, and Jee-Soo Lee. Finding approximate covers of strings. *Journal of Korea Information Science Society*, 29(1):16–21, 2002. URL: http://www.koreascience.or.kr/article/ArticleFullRecord.jsp?cn=JBGHG6_2002_v29n1_16.

34 R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.