FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Approximate Covers of Strings
Student:	Vendula Švastalová
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

Study the notion of approximate cover of a string under Hamming and Levenshtein distance. Study, thoroughly describe, and experimentally evaluate the algorithm for computing approximate covers presented recently by Kędzierski et al. [1]

–

[1] KĘDZIERSKI, Aleksander; RADOSZEWSKI, Jakub. k-Approximate Quasiperiodicity under Hamming and Edit Distance. In: 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020). DOI 10.4230/LIPIcs.CPM.2020.18.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 14 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Approximate Covers of Strings

Vendula Švastalová

Department of Theoretical Computer Science
Supervisor: Ing. Ondřej Guth, Ph.D.

June 27, 2021

Acknowledgements

I would like to thank Ing. Ondřej Guth, Ph.D. for the supervision of this thesis, for his expertise, guidance and patience. I would also like to thank my partner and family for supporting me greatly during my studies.

Declaration

In Prague on June 27, 2021 …………………

Citation of this thesis

Švastalová, Vendula. *Approximate Covers of Strings.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce staví na publikaci Kędzierského and Radoszewského, kteří popsali vylepšené polynomiální algoritmy řešící problém k-přibližných pokrytí řetězců nad Hammingovou, Levenshteinovou a váženou editační vzdáleností. Práce tyto algoritmy důkladně popisuje, vysvětluje a poskytuje jiný úhel pohledu. Algoritmy jsou implementovány a problémy, které řesí jsou zasazeny do kontextu dalších pravidelností v řetězcích. Implementce je experimentálně otestována a jsou popsány hlavní implementační kroky.

Klíčová slova přibližné pravidelnosti, pravidelnost v řetězci, přibližné pokrytí, rozšířené pokrytí, Levenshteinova vzdálenost, Hammingova vzdálenost, vážená editační vzdálenost, kvaziperiodicita

Abstract

This thesis builds upon recent findings of Kędzierski and Radoszewski who presented improved polynomial time algorithms for computing k-approximate covers of strings under Hamming, Levenshtein and weighted edit distance. These algorithms are thoroughly described providing explanations from different point of view. The algorithms are implemented and the problems they solve are inset into the context of other string regularities. The implementation is experimentally evaluated alongside with the description of the main implementation decisions.

Keywords approximate regularities, string regularity, approximate cover, enhanced cover, Levenshtein distance, Hamming distance, weighted edit distance, quasiperiodicity

Contents

List of Figures

Introduction

Motivation

The study of regularities in strings is nowadays relevant to various fields of study including data compression, computer-assisted music analysis or bioinformatics.

During biological research large amounts of data is acquired. This data is further heavily studied therefore it is crucial to come up with faster and more efficient solutions for understanding and processing its volume. One of the ways to further analyse the data is to investigate its regularities. The findings might uncover important underlying relationships of the biologic structures, that produced the studied data, and can, for example, lead to new medical discoveries.

Aims and Objectives

The aim of this thesis is to describe, analyse and implement polynomial–time algorithms for computing regularities in a string. Particularly *k–coverage under Hamming* and *edit distance* as well as *restricted approximate covers under edit distance.* These algorithms are presented in a 2020 study [1] by Kędzierski and Radoszewski and improve upon complexities of former solutions.

Implementations are then experimentally evaluated using randomly generated strings with various parameters.

Lastly, the aim of this thesis is also to thoroughly explain all the terms and notions of regularities and various distance metrics — Hamming distance, Levenshtein distance, weighted edit distance — that this thesis builds upon.

Structure

The introduction states the main goals of the thesis and explains why it is crucial to be concerned about *string regularities*.

First chapter lists all the definitions of terms and notions that are used throughout the work. Namely various string regularities and metrics.

Second chapter presents different data structures that are the building blocks for understanding the implementation of the algorithms discussed in the thesis.

Third chapter describes the algorithms introduced by Kędzierski and Radoszewski [1].

Fourth chapter shows pseudocodes and discusses the implementation of the algorithms.

Fifth chapter deals with the experimental evaluation of the implemented algorithms.

The thesis concludes with propositions on future improvements.

Chapter 1

Definitions and Notions

This chapter lists definitions and notions that are essential to fully understand the concepts described in this work.

1.1 Strings

Definition 1.1.1 (Alphabet)**.** An *alphabet* is a non-empty finite set of symbols denoted by Σ.

Example. Alphabet consisting of three symbols $\Sigma = \{a, b, c\}$

Definition 1.1.2 (String)**.** A *string* S is an ordered sequence composed of symbols defined in an alphabet Σ. Length of string S is denoted by $|S|$ and determines the number of symbols that S is composed of. S is called an *empty string* when $|S| = 0$ and is denoted by ε. The consecutive symbols of S are denoted by $S[0], \ldots, S[n-1]$ where $n = |S|$.

Example. $S = aaabcabaabbaaab$, $|S| = 15$, $S[0]$ is the first symbol of S equal to a and $S[4]$ is the fifth symbol of S equal to c.

Definition 1.1.3 (Substring)**.** A *string* describing the subsequent symbols in S bounded by and including symbols $S[i]$ and $S[j]$ is called a *factor* or *substring* and is labeled as $S[i, j]$. For i and j it applies that $j \geq i$ and $i, j \in [0, n-1]$. Length of such substring is $|S[i, j]| = j - i + 1$.

Example. $S = aaabcabaabbaaab$ where $S[4, 8] = cabaa$ is substring of S.

Definition 1.1.4 (Prefix, suffix)**.** A factor of S is a *prefix* when $i = 0$ and a *suffix* when $j = n - 1$.

Example. $S = aaabcabaabbaaab$ with prefix $S[0, 4] = aaabc$ and suffix $S[10, 14] = baaab$.

Definition 1.1.5 (Border)**.** *Border* of S is a string that is both its *prefix* and *suffix.*

Example. $S = aaabcabaabbaaab$. Border $aaab$ of length 4.

1.2 Metrics

Definition 1.2.1 (Metric)**.** *Distance* or *metric* defined on set X is a function

$$d : X \times X \to [0, +\infty)$$

such that for each element $x, y, z \in X$ holds:

- **Positive definiteness:** $d(x, y) \geq 0$, $d(x, y) = 0 \iff x = y$,
- **Symmetry:** $d(x, y) = d(y, x)$,
- **Triangular inequality:** $d(x, y) + d(y, z) \geq d(x, z)$. [2]

Algorithms discussed and implemented in this thesis operate with several metrics that are stated by the following definitions.

Definition 1.2.2 (Hamming distance)**.** Given two strings S, T of equal lengths ($|S| = |T|$), *Hamming distance* $d(S, T)$ is defined as the number of positions x such that $S[x] \neq T[x]$. [2]

Example. For strings S and T the Hamming distance $d(S, T) = 3$.

$$S = a\ a\ a\ b\ c\ a\ b\ a\ a\ b\ b\ a\ a\ a\ b$$
$$T = a\ a\ a\ b\ b\ b\ b\ a\ a\ b\ b\ a\ c\ a\ b$$

Definition 1.2.3 (Weighted edit distance)**.** Given two strings S, T minimum cost of *edit operations* required to transform S into T is a *metric* called the *edit distance.* Given an alphabet Σ and an empty symbol ε let *cost* of an operation be a function c such that for each symbol $x, y \in \Sigma$

$$c : x, y \to \mathbb{R}_0^+$$

Edit operations that are most commonly used

- inserting symbol x — $c(\varepsilon, x)$,
- deleting symbol x — $c(x, \varepsilon)$,
- substituting symbol x by y — $c(x, y)$, for $x = y \implies c(x, y) = 0$. [3]

Costs of the operations for all symbols are given by a *penalty matrix*.

Assigning different weights to different types of operations depending on their likelihood can be convenient in practice — for example when substituting symbol **a** by symbol **s**. In this case the weight can be set higher than for substituting symbol **a** by symbol **p** as **a** and **s** are closer on the keyboard than **a** and **p** so it is more likely that **s** was mistaken for **a**.

A well–known *dynamic programming* algorithm, when given two strings S and T, can compute the weighted edit distance in $O(|S| \times |T|)$. This algorithm is further described in 2.1.1 and its implementation is presented in 4.2.2

Example. Given an alphabet $\Sigma = \{a, b, c\}$. The *cost* for substituting the symbol a with the symbol b is set to $c(a, b) = 3$, for inserting the symbol b is set to $c(\epsilon, b) = 5$ and for deleting the symbol b is set to $c(b, \epsilon) = 1$. For strings $S = abcca$ and $T = accbb$ the weighted edit distance is $d(S, T) = 9$.

$$S = abcc\ \ a$$
$$T = a\ \ ccbb$$

Definition 1.2.4 (Levenshtein distance)**.** Given two strings S, T, the *Levenshtein distance* is equivalent to the *edit distance* defined in 1.2.3, such that when given symbols $x, y \in \Sigma$ where $x \neq y$ the *cost* of all operations — inserting, deleting, substituting — is equal to 1. [3]

Example. For strings $S = abcca$ and $T = accbb$ the Levenshtein distance is $d(S, T) = 3$. One of the possible alignments is shown in this example.

$$S = abcc\ \ a$$
$$T = a\ \ ccbb$$

1.3 Regularities

Stringology concerns finding patterns and characterizing regularities in strings. String periodicity is a repeating instance of certain string inside another. Quasiperiodicity was introduced as an extension to string periodicity. It describes an irregular periodicity of a string. Basic notions of quasiperiodicity are *cover* and *seed*. This work is primarily focused on and works with covers, specifically approximate covers. Given a metric d, approximate quasiperiodicities are such regularities that allow approximate occurrences in strings with distances measured by d.

A string S is periodic when it has a period. Period is an initial string P usually consisting of less characters than S such that it can generate string S by repeating itself. String that has no period — can not be generated by any other string consisting of less symbols — is called *primitive*.

Definition 1.3.1 (Cover)**.** String C is a *cover* of string S if every position of S is covered by an occurrence of C in S. Therefore every string S is also its own cover and every cover of S is also its border that is defined in 1.1.5. Considering $C \neq S$ then S is quasiperiodic if C is its cover [4].

Example. All positions of string $S =$ `cabcabccabc` are covered by $C =$ `cabc` and therefore C is a cover of S.

A string is *quasiperiodic* when it has a cover. Cover is a generalization of period in such sense that unlike cover, period in a string cannot overlap. When there exists neither period nor quasiperiod in a string (the string is not coverable) such string is called *superprimitive*.

Definition 1.3.2 (Partial cover)**.** String C that is required to cover specified number l of positions of string S.

Example. For $l = 6$, `ab` is a partial cover of string `abcabbabb`.

Definition 1.3.3 (Enhanced cover)**.** *Enhanced cover* is a partial cover 1.3.2 which is required to also be a border of string S.

Example. For $l = 6$, `ab` is an enhanced cover of string `abcabbab`.

Definition 1.3.4 (Seed)**.** String C is a *seed* of string S if and only if C is a cover of a superstring of S — that is, a cover of string S with allowing left or right overhangs of C. A suffix of C can therefore be a prefix of S and prefix of C can be a suffix of S. [5]

Example. Seed of string $S =$ `baababaa` is string $C =$ `aba` with left overhang of a and right overhang of ba.

The notion of seed is more complicated as far as finding this quasiperiodicity in a string is concerned. Unlike for finding covers there is no known linear time algorithm for finding seeds of a string. Therefore variations of this problem can be considered for which there have been presented linear time algorithms [6] — such as left or right seed. These quasiperiodicities describe such covers for which only left or right overhangs are allowed.

Definition 1.3.5 (k-coverage)**.** A metric d, string S and number k are given. *k-coverage* of string S is the number of positions that are covered by k-approximate occurrences of C with distance under d of at most k. When k-coverage of C is equal to $|S|$ then C is a k-approximate cover of S (see 1.3.6).

All k-approximate occurrences under metric d of string S in string T are described by a set of all intervals of where the occurrence of S starts and ends in T. This is formally denoted by

$$Occ_k^d(S,T) = \{[i,j] : d(S,T[i,j]) \leq k\} \tag{1.1}$$

k-coverage of S in T is then computed as the size of the union of all occurrences of S in T. Formally

$$Coverage_k^d(S,T) = |\bigcup Occ_k^d(S,T)| \tag{1.2}$$

Example. 1-coverage of `ba` in $S =$ `abacabb` is 6. 2-coverage of `acc` in $T =$ `abacabb` is 7 and therefore `acc` is a 2-approximate cover of T because $|T| = 7$.

Definition 1.3.6 (k-approximate cover)**.** A metric d, string S and number k are given. String C is a k-approximate cover of S if all positions of S are covered by k-approximate occurrences of C with distance under d of at most k. Introduced by Sim et al. in [7]. Formally

$$Coverage_k^d(C,S) = |S|$$

Example. 1-approximate cover `aba` of `abacabb`. 2-approximate cover `baa` of `baabccaa`.

Definition 1.3.7 (Restricted approximate cover)**.** In the context of this thesis restricted approximate cover of string S is such approximate cover that is also a substring of S.

Restricted approximate covers are considered because it has been shown that computing non-restricted approximate covers under weighted edit distance is an NP-hard problem [7]. In [1] it is shown that also for Hamming distance it is an NP-hard problem to compute non-restricted approximate covers.

Definition 1.3.8 (Relaxed approximate cover)**.** String C that does not need to approximately cover the whole string S and therefore its k-approximate coverage does not have to be equal to $|S|$.

Example. `aba` is a relaxed 1-approximate cover of string `acabcbbba`.

CHAPTER 2

Data structures

This chapter lists and describes all data structures that are needed for implementation of the algorithms presented by Kędzierski and Radoszewski in [1].

2.1 Edit distance table

Definition 2.1.1 (D-table)**.** [8] D-table is a data structure needed for the dynamic programming solution of *edit distance* of two strings S_1 of lenght m and S_2 of length n. A cell of the table $D[i,j]$ determines the edit distance between prefixes $S_1[0,i]$, $S_2[0,j]$ of the two given strings.

Cost function c as explained in 1.2.3 and strings S_1, S_2 are given. The D-table is filled as follows: $D[-1,-1] = 0$, $D[i,-1] = D[i-1,-1] + c(S_1[i], \varepsilon)$ for $i \geq 0$, $D[-1,j] = D[-1,j-1] + c(\varepsilon, S_2[j])$ for $j \geq 0$ and for $i,j \geq 0$

$$\begin{aligned} D[i,j] = \min(&D[i-1,j-1] + c(S_1[i], S_2[j]), \\ &D[i,j1] + c(\varepsilon, S_2[j]), \\ &D[i1,j] + c(S_1[i], \varepsilon)) \end{aligned}$$

Example. $S_1 = abacabb$, $S_2 = ababa$ and the cost of all operations is 1. The edit distance of S_1 and S_2 can be found in cell $D[6,4] = 3$ in table 2.1.

2.1.1 Complexity

Using this dynamic programming approach, all the cells are computed from cells that have been computed in the previous iteration. Therefore the results can be obtained in $\mathcal{O}(mn)$ time.

		-1	0	1	2	3	4
		ε	a	b	a	b	a
-1	ε	0	1	2	3	4	5
0	a	1	0	1	2	3	4
1	b	2	1	0	1	2	3
2	a	3	2	1	0	1	2
3	c	4	3	2	1	1	2
4	a	5	4	3	2	2	1
5	b	6	5	4	3	2	2
6	b	7	6	5	4	3	3

Table 2.1: D-table for $S_1 = abacabb$ and $S_2 = ababa$

2.2 h-wave

In this section we consider D-tables that are computed under *Levenshtein distance* (1.2.4).

Definition 2.2.1 (diagonal)**.** A *diagonal* d is a list of all points (i, j) in the D-table for which $j = d + i$ applies. Bottom left diagonal is therefore labeled $d = -m$ and the top right diagonal $d = n$.

Lemma 2.2.1. [9, p. 4] All values on each diagonal of D are non-decreasing and always increase by at most one.

$$\forall(i, j) : D[i, j] - D[i - 1, j - 1] \in \{0, 1\}$$

Lemma 2.2.2. [9, p. 4] Difference between a value on diagonal d and an adjacent value on diagonal $d + 1$ (resp. $d - 1$) is at most one.

$$\forall(i, j) : D[i, j] - D[i - 1, j], D[i, j] - D[i, j - 1] \in \{-1, 0, 1\}$$

It was stated by Ukkonen [9] that the D-table is directly determined by the last values h on each diagonal d as 2.2.1 applies for all values in D.

Lemma 2.2.3. There are no h values outside of diagonals $[-h...h]$. [9]

Definition 2.2.2 (h-wave)**.** [10] Let $L^h(d)$ be the highest row index i of a point from D on a diagonal d with value h. This point is directly identified by i as $(i, i + d)$. Formally

$$L^h(d) = \max\{i : D[i, i + d] = h\}$$

Knowing 2.2.3 an *h-wave* L^h is defined as an ordered list of the $2h + 1$ furthest h-points on diagonals $-h$ to h.

$$L^h = [L^h(-h), L^h(-h + 1), ..., L^h(0), ..., L^h(h - 1), L^h(h)]$$

$L^h(d)$ is set to ∞ in several cases:

1. all values on the diagonal d are less than h or
2. $L^h(d) = m-1$, $D[m-1, m-1+d] = h$ and $D[m-1, m+d] = h-1$ or
3. $L^h(d) = n-1$, $D[n-1-d, n-1] = h$ and $D[n-d, n-1] = h-1$

Example. In the example D-table 2.2 the h-values of the following h-waves for strings $S_1 = acbbcaca$ $(m = 8)$, $S_2 = abbacacca$ $(n = 9)$ are highlighted.

- h-points of $L^1 = [3, 2, 0]$ are highlighted in green.
- h-points of $L^2 = [7, 4, 6, 2, 0]$ are highligted in blue.
- h-points of $L^3 = [\infty, \infty, 7, 7, 7, 2, 1]$ are highlighted in red.
  - $L^3(-3) = \infty$ because condition 2 applies here:

    $$L^3(-3) = 7 = m-1, D[7, 4] = 3 \text{ and } D[7, 5] = 2$$

  - $L^3(-2) = \infty$ because condition 1 applies here

	j	-1	0	1	2	3	4	5	6	7	8	∞
i		ε	a	b	b	a	c	a	c	c	a	
-1	ε	0	1	2	3	4	5	6	7	8	9	
0	a	1	0	1	2	3	4	5	6	7	8	
1	c	2	1	1	2	3	3	4	5	6	7	
2	b	3	2	1	1	2	3	4	5	6	7	
3	b	4	3	2	1	2	3	4	5	6	7	
4	c	5	4	3	2	2	2	3	4	5	6	
5	a	6	5	4	3	2	3	2	3	4	5	
6	c	7	6	5	4	3	2	3	2	3	4	
7	a	8	7	6	5	4	3	2	3	3	3	
∞								3	3			

Table 2.2: D-table with highlighted h-points

Note. An h-wave for strings $S[i, n-1]$ and $S[j, n-1]$ is denoted by $H_{i,j}$.

Definition 2.2.3 (Slide function)**.** Given diagonal d and starting row index i the *slide function* returns for two substrings of S_1 and S_2 the maximum index q for which all symbols of the substrings $S_1[i, q]$ and $S_2[i+d, q+d]$ are equal. Formally:

$$Slide_d(i) = \max\{q : S_1[i, q] = S_2[i+d, q+d]\}$$

Slide function essentially finds the index of the next h-point on diagonal d.

2.2.1 Computing h-wave

An h-wave is computed by induction from $(h-1)$-wave using the formula:

$$\forall h > 0 : L^h(d) = Slide_d\left(\max \begin{cases} L^{h-1}(d+1)+1, & if d < h \\ L^{h-1}(d)+1, & always \\ L^{h-1}(d-1), & if d > -h \end{cases}\right)$$

The initial step of the induction is simply computing the 0-wave consisting of a single point $L^0 = [Slide_0(0)]$.

2.2.2 Complexity

Complexity of the algorithm to compute wave L^k for a given k depends on the efficiency with which $Slide_d(i) = q$ is computed.

Comparing characters in each iteration results in complexity of the slide function being $\mathcal{O}(q-i)$ and therefore the resulting complexity being $\mathcal{O}(km)$.

However, there exists an improved approach resulting in faster algorithm. In this approach the $Slide_d(i)$ query is answered in $\mathcal{O}(1)$. It needs $\mathcal{O}(n)$ preprocessing in the form of a *generalized suffix tree* constructed for string S_1xS_2y where $x, y \notin \Sigma, x \neq y$ with LCA (lowest common ancestor) access. Then the complexity to construct an L^k wave from L^0 is $\mathcal{O}((k+1)^2) = \mathcal{O}(k^2)$ so the overall worst-case time complexity including preprocessing is $\mathcal{O}(k^2+n)$ [10].

2.3 Suffix trie, suffix tree and LCA

Suffix trie is a data structure represented by a rooted tree. It represents a string S with symbols from alphabet Σ. Each edge of the suffix trie is labeled with a single symbol from Σ. Each path from root to any node represents a suffix from S.

Suffix tree is a compressed suffix trie. It can be constructed in $\mathcal{O}(n)$ time [11]. An example of a suffix tree is shown in figure 2.1.

Lowest common ancestor queries can be made in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ time preprocessing of a given suffix tree [12]. In the example suffix tree 2.1 the lowest common ancestor of the two leaves 0 and 2 is the first vertex from the root by going along the edge b. Lowest common ancestor of 6 and 3 is the root itself.

2.4 RMQ (Range minimum query)

Definition 2.4.1 (Sparse table [13]). *Sparse table* is a data structure used for answering range queries, most of which can be answered in $\mathcal{O}(\log n)$ time.

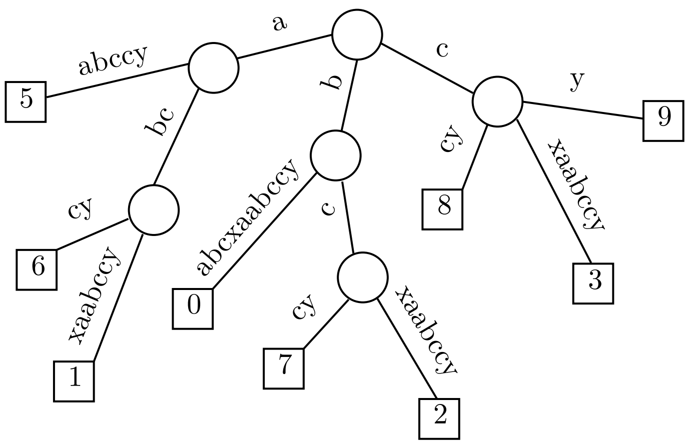

Figure 2.1: Suffix tree for string $S_1 x S_2 y$ where $S_1 = babc$ and $S_2 = aabcc$

This data structure can only be used on immutable data, meaning that if the source data changes at any time, the sparse table has to be recomputed.

However, for answering *range minimum queries*, sparse table can be even more efficient as it can answer these in $\mathcal{O}(1)$ time.

Given an array of n values and a range [L, R], range minimum query is answered as the minimum value in the range [L, R].

The sparse table is represented by a $2D$ array of size $N * (K + 1)$ where $K = \lfloor \log_2 N \rfloor$

Definition 2.4.2 (Range minimum query [13])**.** Any interval can be represented by a union of smaller intervals such that each of them has a length of a power of two. When working with RMQ on sparse table ST all the answers for queries with lengths of power of two are precomputed and then can be queried and combined to receive the final result.

Each cell $ST[i, j]$ represents a minimum for a range of length 2^j starting at i: $ARR[i, i + 2^j - 1]$.

Because all the minimums for every power of two long ranges are precomputed, it is possible to split the query range $[L, R]$ into two overlapping ranges of power of two lenghts and then query for each range and compute the minimum value out of the two ranges. This results in:

$$\min(ST[L, j], ST[R - 2^j + 1][j]) \text{ where } j = \lfloor \log_2(R - L + 1) \rfloor$$

Example. Sparse table for RMQ of $ARR = [16, 15, 19, 3, 20, 0, 7, 7, 13, 7, 16]$ is shown in table 2.3. Row indices are values of j and column indices are values of i The table is of size 11 * 4.

For query on interval $[5, 9]$ the closest lower power j is computed for the length of the interval $j = \lfloor \log_2 5 \rfloor = 2$ and then the result is found as the minimum of two intervals of length $2^j = 4$: $ARR[5, 8]$ and $ARR[6, 9]$

which can be found in $ST[5,2] = 0$ and $ST[6,2] = 7$ resulting in minimum $min(ARR[5,9]) = 0$.

$$\begin{tikzcd}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
0 & 16 \arrow[d] & 15 \arrow[d] \arrow[dl] & 19 \arrow[d] \arrow[dl] & 3 \arrow[d] \arrow[dl] & 20 \arrow[d] \arrow[dl] & 0 \arrow[d] \arrow[dl] & 7 \arrow[d] \arrow[dl] & 7 \arrow[d] \arrow[dl] & 13 \arrow[d] \arrow[dl] & 7 \arrow[d] \arrow[dl] & 16 \arrow[dl] \\
1 & 15 \arrow[d] & 15 \arrow[d] & 3 \arrow[d] \arrow[dll] & 3 \arrow[d] \arrow[dll] & 0 \arrow[d] \arrow[dll] & 0 \arrow[d] \arrow[dll] & 7 \arrow[d] \arrow[dll] & 7 \arrow[d] \arrow[dll] & 7 \arrow[dll] & 7 \arrow[dll] & \\
2 & 3 \arrow[d] & 3 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[dllll] & 0 \arrow[dllll] & 7 \arrow[dllll] & 7 \arrow[dllll] & & & \\
3 & 0 & 0 & 0 & 0 & & & & & & &
\end{tikzcd}$$

Table 2.3: Sparse table for RMQ

2.4.1 Complexity

Sparse table is precomputed in $\mathcal{O}(N \log N)$ and range minimum queries take $\mathcal{O}(1)$ time.

CHAPTER 3

Approximate covers problems

This chapter lists all the problems and their solutions presented by Kędzierski and Radoszewski in [1]. Their algorithms improve upon previous solutions on restricted quasiperiodicity. The implementation of the problems 3.1.1, 3.1.2, 3.3.1, 3.3.2 is then presented in chapter 4.

3.1 Approximate covers under weighted edit distance

For computing weighted edit distance we consider such D-table that is defined on a single string S of length n labeled as $D_{a,a'}$. This table characterizes a D-table computed for strings $S[a, n-1]$ and $S[a', n-1]$. For values $b \in [a, n-1]$ and $b' \in [a', n-1]$ the weighted edit distance between $S[a, b]$ and $S[a', b']$ is determined by $D_{a,a'}[b, b']$.

3.1.1 k-coverage of every factor

The algorithm described in this section computes the k-coverage of every factor of string S.

Definition 3.1.1 (Longest approximate prefix problem)**.** Element $P_k^{ed}[a, b, a']$ is defined as the furthest point $b' \geq a' - 1$ such that $ed(S[a, b], S[a', b']) \leq k$ or -1 if no such point exists.

In other words it is the largest index b' in the table $D_{a,a'}$ for which $D_{a,a'}[b, b'] \leq k$.

Example. Given string $S = abaca$, $a = 0$, $a' = 2$ and weighted edit distance is defined with unit costs. Examples are shown in table 3.1

- $k = 2$, $b = 2$: $P_{ed}^k[a, b, a'] = 4$. The point $D_{0,2}[2, 4]$ is shown in green box.

- $k = 1$, $b = 3$: $P^k_{ed}[a, b, a'] = -1$. There is no index b' that meets the condition $ed(S[a, b], S[a', b'] \leq k$ and therefore the P value is set to -1.
- $k = 1, b = 1$: $P^k_{ed}[a, b, a'] = 3$. The point $D_{0,2}[1, 3]$ is shown in blue box.

				a				
			-1	0	1	2	3	4
			ε	a	b	a	c	a
	1	ε	0	1	2	3	4	5
a'	2	a	1	0	1	2	3	4
	3	c	2	1	1	2	2	3
	4	a	3	2	2	1	2	2

Table 3.1: $D_{0,2}$ for string $S = abaca$

An interval $[a', P^{ed}_k[a, b, a']]$ then defines the occurrence of $S[a, b]$ in S as described by equation 1.1. It is possible that $P^{ed}_k[a, b, a'] = -1$ in such case when $P^{ed}_k[a, b, a'] < a'$ the interval is considered to be empty.

To calculate the k-coverage of $S[a, b]$ in S all occurrences of $S[a, b]$ must be found. Applying equation 1.2 results in the formula for calculating k-coverage of a factor $S[a, b]$ in S under weighted edit distance.

$$Coverage^{ed}_k(S[a, b], S) = |\bigcup_{a'=0}^{n-1} [a', P^{ed}_k[a, b, a']]|$$

Definition 3.1.2 (Patero-dominance of pairs)**.** In [1] Patero-dominance of two pair is defined as

$$\forall x, y, x', y' : (x, y) \neq (x', y'), x \leq x', y \geq y' \Leftrightarrow (x, y) \text{ patero-dominates } (x', y')$$

Patero-dominance is used in the construction of $L_{a,a'}[b]$ table which represents all pairs $(D_{a,a'}[b, b'], b')$ of values from column b and their b' indices that are **not dominated** by others. Pairs are stored in an increasing order sorted by the first element of the pair. $L_{a,a'}[b]$ is constructed from $D_{a,a'}[b, \cdot]$.

Example. Given string $S = abaca$, $a = 0$, $a' = 2$ and weighted edit distance is defined with unit costs. In table 3.2 examples of patero-dominated pairs are shown.

- for $b = 2$ there is only one pair $(1, 4)$ that is not dominated by others. Value from the pair is highlighted in green box. $L_{0,2}[2] = [(1, 4)]$
- for $b = 1$ there are 2 pairs that are not dominated by others. Values from the pairs are highlighted in blue box. $L_{0,2}[1] = [(1, 3), (2, 4)]$

- for all other columns $b \in \{-1, 0, 3, 4\}$ values from the not dominated pairs are highlighted in pink box
  - $L_{0,2}[-1] = [(0,1),(1,2),(2,3),(3,4)]$
  - $L_{0,2}[0] = [(0,2),(1,3),(2,4)]$
  - $L_{0,2}[3] = [(2,4)]$
  - $L_{0,2}[4] = [(2,4)]$

				a				
			-1	0	1	2	3	4
			ε	a	b	a	c	a
	1	ε	0	1	2	3	4	5
a'	2	a	1	0	1	2	3	4
	3	c	2	1	1	2	2	3
	4	a	3	2	2	1	2	2

Table 3.2: $L_{0,2}[b]$ examples for string $S = abaca$

The final algorithm uses multiples of a value $M = \lfloor \sqrt{n/\log n} \rfloor$ called **special points**.

These data structures must first be precomputed:

1. first $min(M, n - a' + 1)$ rows of all $D_{a,a'}$ for $a, a' \in [0, n]$,
2. all $L_{a,a'}$ lists for $a = kM \vee a' = kM\ \forall k \in \mathbb{N}_0$ (a, a' is a special point), $a, a' \in [0, n-1], b \in [a-1, n-1]$. For $a \geq n \vee a' \geq n$ the list $L_{a,a'}[b]$ is empty.

Definition 3.1.3 (Predecessor)**.** Predecessor operation is defined on $L_{a,a'}[b]$ of pairs ordered by their first component. It finds the element whose first component is less than or equal to the searched number x. If there is no such value, the operation returns pair (∞, ∞). Because the list is sorted by the first components the element can be found in $\mathcal{O}(\log n)$.

Example. Given string $S = abaca$ from example table 3.2

- For list $L_{0,2}[0]$ the predecessor of $x = 2$ is $(2, 3)$,
- For list $L_{0,2}[3]$ the predeccessor of $x = 1$ is (∞, ∞)

3.1.1.1 Complexity

Data structures for the first point are computed in $\mathcal{O}(n^4/M) = \mathcal{O}(n^3 \log n)$ time.

$L_{a,a'}[b]$ can be computed from $D_{a,a'}[b,\cdot]$ in $\mathcal{O}(n)$ time. So for all $\mathcal{O}(n^2)$ combinations of $a, a' \in [0, n-1]$ $\mathcal{O}(n/M)$ lists are computed in $\mathcal{O}(n)$ time. The resulting complexity for point 2 is

$$\frac{n^2}{M} \cdot n \cdot n = \frac{n^4}{M} = \frac{n^4}{n\sqrt{\frac{1}{n \log n}}} = \frac{n^3}{(n \log n)^{-\frac{1}{2}}} = n^3\sqrt{n \log n} \approx \mathcal{O}(n^3\sqrt{n \log n})$$

Computation of element $P_k^{ed}[a,b,a']$ takes at most M times search for predecessor in the list of pairs and therefore the complexity to compute it is $\mathcal{O}(M \log n) = \mathcal{O}(\sqrt{n \log n})$. Precomputations take $\mathcal{O}(n^3\sqrt{n \log n})$. Therefore the precomputations and computation of $P_k^{ed}[a,b,a']$ for all $a' \in [0, n-1]$ and for all factors (there are $\frac{n^2+n}{2} \approx \mathcal{O}(n^2)$ factors) gives the overall complexity for computing **all k-approximate covers of string S** is

$$\mathcal{O}(n^3\sqrt{n \log n} + \sqrt{n \log n} \cdot n \cdot \frac{n^2+n}{2}) \approx \mathcal{O}(n^3\sqrt{n \log n})$$

.

The implementation of this algorithm including pseudocode is discussed in Chapter 4.

3.1.2 Restricted approximate covers

The algorithm for computing restricted approximate covers finds for all factors of string S the k for which they are k-approximate covers of S. This algorithm builds upon the ideas described in section 3.1.1.

It computes a $Q_{a,b}$ array for each factor $S[a,b]$ such that at its i^{th} position it has the minimum weighted edit distance k for which the factor $S[a,b]$ is a k-approximate cover of $T[i, n-1]$. The k for the specified factor $S[a,b]$ are then located in $Q_{a,b}[0]$.

For fast retrieval of minimum on an interval, this algorithm uses sparse table representation of the $Q_{a,b}$ array for range minimum queries 2.3.

Pseudocode for computing the $Q_{a,b}$ array using precomputed data structures 3.1.1 is presented in 4.3.1.5.

3.1.2.1 Complexity

This algorithm again needs precomputed D-tables and L lists as described in 3.1.1. Constructing the sparse table from $Q_{a,b}$ elements takes $\mathcal{O}(n \log n)$ time. And the $Q_{a,b}[i]$ element can be computed in $\mathcal{O}(M \log n)$ when using dynamic programming with the use of M. The overall complexity including precomputations is then $\mathcal{O}(n^3\sqrt{n \log n})$.

The pseudocode of this algorithm is discussed in chapter 4.

3.2 Approximate covers under Levenshtein distance

3.2.1 k-coverage of every factor

Given string S and number k, compute the k-coverage of every factor $S[i, j]$ in S under Levenshtein distance.

As a preliminary to solve this problem it is needed to be able to compute an h-wave (see 2.2) $H_{i,j}$ for $h = k$.

To implement the h-wave efficiently the string S and its factor F are stored in a generalized suffix tree in the form $SxFy$. This suffix tree can be constructed in $\mathcal{O}(n)$ time. Then the suffix tree is preprocessed in $\mathcal{O}(n)$ time so that it can answer lowest common ancestor queries in $\mathcal{O}(1)$.

Definition 3.2.1 (Longest approximate prefix problem)**.** Element $P_k^{ld}[a, b, a']$ is defined as the furthest point $b' \geq a' - 1$ such that $ld(S[a, b], S[a', b']) \leq k$ or -1 if no such point exists.

The k-coverage of a factor $S[a, b]$ is then computed from all P^{ld} table consisting of all element $P_k^{ld}[a, b, a'], a \in [0, n-1]$ using the equation 1.2.

3.2.1.1 Complexity

The algorithm for computation of k-coverage under Levenshtein distance proposed in [1] computes this problem in $\mathcal{O}(n^3)$ time.

3.3 Approximate covers under Hamming distance

3.3.1 k-coverage of every prefix

Given string S of length n and number k, compute the k-coverage of every prefix under Hamming distance.

For the computation of the k coverage of every prefix under Hamming distance we must first compute a table of the longest common prefix with k mismatches for every suffix of string S where S is of length n.

Definition 3.3.1 (Longest common prefix algorithm)**.** The length of the longest common k-approximate prefix of two suffixes $S[i, n-1], S[j, n-1]$ is denoted by $lcp_k(i, j)$. In [1] it is suggested to use an algorithm described in [14] which computes the longest common factor with k mismatches of two strings. Internally this algorithm computes longest common suffix with at most k mismatches of every two prefixes and returns the length of the longest one. Although the algorithm presented in [14] computes the longest common suffix, it can be easily transformed to compute the longest common prefix by reversing the input string.

$PREF_k$ table is defined as the list of values $[lcp_k(0,0),\dots,lcp_k(0,n-1)]$.

The algorithm starts with a list $\mathcal{L} = [0,\dots,n]$ of all indices including n as a bounding value and at each step computes the k-coverage of a prefix of length $l = 1,\dots,n$.

The $\mathcal{L}$ linked list is kept updated with each change of l so that it always contains such indices i for which $PREF_k[i] \geq l$. This means that for each l the algorithm only accounts for such suffixes that have at least l positions covered by $T[0,l-1]$.

There is always a set of neighbouring indices in $\mathcal{L}$ that are stored as pairs and manipulated with respect to the current l. Initially $[(0,1),(1,2),\dots,(n-1,n)]$.

A pair (i,j) can be classified into one of two categories

- **overlapping** when $j - i < l$
- **non-overlapping** when $j - i \geq l$

In other words two factors of length l starting at positions i and j are *non-overlapping* when they are at least l positions apart and *overlapping* when there is less than l positions between them.

If a pair (i,j) is *non-overlapping*, then there is an occurrence starting on i of length l and the whole length l can be added to the coverage as the next occurrence starting on index j does not overlap it. If a pair is *overlapping* then only $j-i$ positions can be added to the coverage as the index j is also a start of a new occurrence and all positions must be accounted for only once.

The goal is to classify all pairs of neighboring indices (i,j) in $\mathcal{L}$ as either *overlapping* or *non-overlapping* and then for each pair (i,j) that is *overlapping* add $j-i$ to the sum sum_o and count the total number of *non-overlapping* pairs as num_{no}.

The final k-coverage of $S[0,l-1]$ in S is then

$$Covered_k^{Ham}(S[0,l-1],S) = sum_o + num_{no} \cdot l \tag{3.1}$$

Example. Let $S = ababbbbbab$, $k = 1$ and let $l = 3$. The $PREF_1$ table for S is $[10,1,6,2,2,2,3,1,2,1]$.

At the step when $l = 3$ the $\mathcal{L}$ list only contains such indices i for which $PREF_1[i] \geq 3$. So $\mathcal{L} = [0,2,6,10]$ meaning that there are occurrences of $S[0,2]$ starting on indices $i \in \{0,2,6\}$. The list of all neighboring positions is $[(0,2),(2,6),(6,10)]$ where $[(0,2)]$ is *overlapping* and $[(2,6),(6,10)]$ are *non-overlapping*. The overall coverage is then according to the equation

$$Covered_1^{Ham}(S[0,2],S) = (2-0) + 2 \cdot 3 = 8$$

The algorithm uses two data structures for storing *overlapping* (3.3.2) and *non-overlapping* (3.3.3) pairs. Both of these data structures are equipped with operations for *insertion* and *removal* of an element (i,j).

	0	1	2	3	4	5	6	7	8	9
S	a	b	a	b	b	b	b	b	a	b
$S[0,2]$	a	b	a				a	b	a	
			a	b	a					

Definition 3.3.2 (data structure $\mathcal{D}_o$)**.** $\mathcal{D}_o$ is used for storing pairs that are *overlapping*. It stores a variable sum_o.

Inserting pair (i, j) to $\mathcal{D}_o$ means adding $j - i$ to sum_o. *Removing* pair (i, j) from $\mathcal{D}_o$ subtracting $j - i$ from sum_o.

Definition 3.3.3 (data structure $\mathcal{D}_{no}$)**.** $\mathcal{D}_{no}$ is used for storing pairs that are *overlapping*. It stores an array of linked lists $\mathcal{B}$ indexed from 1 to n where list $\mathcal{B}[j - i]$ stores a pair (i, j). $\mathcal{D}_{no}$ also keeps a variable num_{no} that holds the current number of stored pairs in $\mathcal{B}$.

Inserting pair (i, j) to $\mathcal{D}_{no}$ means inserting it to $\mathcal{B}[j - i]$ and incrementing num_{no}. *Removing* pair (i, j) from $\mathcal{D}_{no}$ means removing it from $\mathcal{B}[j - i]$ and decrementing num_{no}.

Definition 3.3.4 (Q table)**.** Table Q indexed from 0 to n is initialized at the beginning of the algorithm. For each $i \in [0, \ldots, n]$ it adds the index i to the list $Q[PREF_k[i]]$. In other words table Q at index x stores all indices i from $PREF_k$ table for which $PREF_k[i] = x$.

Q is used for batch removal of indices from $\mathcal{L}$ that are no longer relevant for the current prefix length l.

For $\mathcal{O}(1)$ removal of indices from linked list $\mathcal{L}$, pointer to each index i in $\mathcal{L}$ is stored in an array $\mathcal{P}[0, \ldots, n - 1]$ under $\mathcal{P}[i]$. For $\mathcal{O}(1)$ removal of pairs from linked lists in $\mathcal{B}$, pointer to each pair (i, j) in $\mathcal{B}[j - i]$ is stored in an array $\mathcal{A}$ under $\mathcal{A}[i]$ as it is not possible to store more than one pair with i as the first component.

Values q_1 and q_2 are predecessor and successor of value q in $\mathcal{L}$. To remove q from $\mathcal{L}$ pairs (q_1, q) and (q, q_2) must be removed from and (q_1, q_2) must be added to the respective data structures for the current l.

The predecessor and successor of i in $\mathcal{L}$ always exist as n is never removed and 0 is also never removed because $PREF_k[0] = n$ which is the highest value of l.

Given all the data structures, operations and number l, the algorithm proceeds as follows for all l from 1 to n:

1. Remove all values stored in $Q[l - 1]$ from $\mathcal{L}$,
2. Move all pairs from list $\mathcal{B}[l]$ in $\mathcal{D}_{no}$ to $\mathcal{D}_o$
3. calculate k-coverage (equation 3.1)

3.3.1.1 Complexity

Given the $PREF_k$ table the time complexity to compute k-coverage of every prefix of string S under Hamming distance is $\mathcal{O}(n)$.

3.3.2 k-coverage of every factor

Given string S of length n and number k, compute the k-coverage of every factor under Hamming distance.

The approach for computing k-coverage of a string starting at index i is the same as in 3.3.1 with a few additional rules.

- An array $[lcp_k(i, 0), \ldots, lcp_k(i, n-1)]$ is used instead of $PREF_k$ table.
- It is now possible that value $q = 0$ will be removed from $\mathcal{L}$ for given l. $q = 0$ always has a successor q_2 in $\mathcal{L}$ (as n is never removed) but has no predecessor q_1. Therefore in case of removing $q = 0$ from $\mathcal{L}$ only the pair (q, q_2) is removed from the respective data structure.
- For starting index i the algorithm computes k-coverage for l from 1 to $n - i$.

To compute k-coverage of every factor of S, values for all $lcp_k(i, j)$ where $i, j \in [0, n-1]$, must be precomputed. Then the algorithm proceeds to compute k-coverage for each $i \in [0, n]$.

3.3.2.1 Complexity

The time complexity to compute k-coverage of every factor of string S under Hamming distance is $\mathcal{O}(n^2)$.

Chapter 4

Implementation

This chapter presents the implementation of problems stated in Chapter 3 along with the implementation of the supporting data structures presented in Chapter 2. Firstly, used technologies with which algorithms were implemented are stated. Secondly, this chapter presents pseudocodes of the algorithms implemented and it develops on the structural choices made during the implementation.

4.1 Used technologies

All algorithms are implemented using C++17 language. As the nature of implemented algorithms benefits greatly from the characteristics of compiled languages.

4.2 Supporting data structures

4.2.1 Penalty matrix

Costs for the operation of substitution used for computing edit distance are stored in `std::unordered_map` for $\mathcal{O}(1)$ access. The group of containers that store costs for deletion, insertion and substitution is represented by `class PenaltyMatrix`.

Costs that are different for different combination of symbols and operations are specified with an input file that is provided at launch of the program. Costs that are unique for each operation but the same for a combination of symbols can also be provided as an argument to the program. If no input file or costs per operation are specified unit costs are used for all combinations of operations and symbols. The different argument options and format of the input file are described in the documentation of the program.

4.2.2 D-table

Given string S, left and right bounding indices a, a', number of rows r, and penalty matrix PM, (4.2.1) the `class DTable` computes the $D_{a,a'}$ table for edit distance values of $S[a', a' + r - 1]$ and $S[a, |S| - 1]$.

If r is not specified it is set to $|S| - a' + 1$ for which the $D_{a,a'}$ table for $S[a', |S| - 1]$ and $S[a, |S| - 1]$ is computed.

Algorithm 1: compute $D_{a,a'}$

```
Input: $S$, $a$, $a'$, $r$, $PM$
for $i \leftarrow 0$ to $r$ do
    for $j \leftarrow 0$ to $|S| - a + 1$ do
        if $i$ is 0 and $j$ is 0 then
            $D_{a,a'}[i, j] = 0$
        else if $i$ is 0 then
            $D_{a,a'}[i, j] = D_{a,a'}[i, j - 1] + PM(\varepsilon, S[j + a - 1])$
        else if $j$ is 0 then
            $D_{a,a'}[i, j] = D_{a,a'}[i - 1, j] + PM(\varepsilon, S[i + a' - 1])$
        else
            $D_{a,a'}[i, j] = min($
                $D_{a,a'}[i - 1][j - 1] + PM(S[i + a' - 1], S[j + a - 1]),$
                $D_{a,a'}[i, j - 1] + PM(\varepsilon, S[j + a - 1]),$
                $D_{a,a'}[i - 1, j] + PM(\varepsilon, S[i + a' - 1])$
            $)$
    end
end
```

4.2.3 RMQ

Range minimum queries are implemented in `class RMTable`. For fast querying the $\lfloor \log_2 x \rfloor$ operation must also be fast. Therefore custom $\log_2$ table is computed on demand, meaning that when $\log_2$ values are missing, more are computed up to the currently accessed value and stored in `static std::vector<int> logCache`. [13].

In the definition of sparse table in 2.4.1 it was stated that sparse table must only be built from an immutable array of values. However, for the case in algorithm 7 new values are consecutively added to the `RMTable` from right to left and are computed from values already known, none of which is changed at any time. So there is no need to recompute the whole structure.

To picture the situation on the example table 2.3, values would be computed colum by colum from right to left.

The pseudocodes for computing the $\log_2$ table and updating sparse table are shown in algorithms 2 and 3.

```
Algorithm 2: compute $\lfloor \log_2 x \rfloor$
  Input  : value $x$, $logCache$
  Output: $\lfloor \log_2 x \rfloor$
1 if $x \geq |logCache|$ then
2     $i := |logCache|$
3     while $i \leq x$ do
4         $logCache[i] = logCache[i/2] + 1$
5         $i := i + 1$
6     end
7 end
8 return $logCache[x]$
```

```
Algorithm 3: update $ST$ at $i^{th}$ index from the end $n$ with $initialValue$
  Input: index $i$ from end $n$, $initialValue$
1 $ST[i][0] = initialValue$
2 for $p := 1$ to $\lfloor \log_2(n - i) \rfloor$ do
3     $ST[i][p] = min(ST[i][p-1], ST[i + 2^{p-1}][p-1])$
4 end
```

4.3 Approximate covers algorithms

4.3.1 Approximate covers under weighted edit distance

In the implementation algorithms and supporting data structures that are used for computing covers under weighted edit distance are defined under `namespace WeightedEditDistance`.

The algorithms for computing covers for weighted edit distance use data that are first precomputed from the initial string S as stated in 3.1.1. The precomputed data is represented by `class PrecomputedData`.

4.3.1.1 Table of patero-dominating pairs

The algorithm for computing patero dominating pairs described in 3.1.2 is presented in [1]. This algorithm is implemented in `class LTable` and uses `std::`

`dequeue` data structure to filter out all pairs that are dominated by (see 3.1.2) other pairs from the column of a D-table. Using this algorithm, the pairs are sorted by the first component in decreasing order, which was needed to into account in further implementation.

Algorithm 4: From a table $D_{a,a'}$ for each column b compute the list of pairs $(D_{a,a'}[b,b'], b')$ that are not dominated by others. [1]

```
for $b := a - 1$ to $n - 1$ do
    $column := b - a + 1$
    for $b' := a' - 1$ to $n - 1$ do
        $d := D_{a,a'}[b, b']$
        while $L_{a,a'}[b]$ is not empty do
            $(d', x) FRONT(L_{a,a'}[b])$
            if $d' \geq d$ then $POP\_FRONT(L_{a,a'}[b])$
            else break
        end
        $PUSH\_FRONT((d, b'), L_{a,a'}[b])$
    end
end
```

4.3.1.2 Predecessor

The predecessor operation is stated in 3.1.3. In an ordered list of pairs it finds such pair whose first component is less than or equal to x.

This operation is implemented in `class LTable` using binary search.

4.3.1.3 Longest common prefix

Algorithm 5 shows the pseudocode for computing $P_k^{ed}[a, b, a']$. The problem is described in 3.1.1 and the computation of $P_k^{ed}[a, b, a']$ is implemented in `class PElement`.

In algorithm 5 (line 8), the closest higher special point is obtained using the the ceil operation.

4.3.1.4 k-coverage for every factor

To get the k-coverage of every factor, the elements $P_k^{ed}[a, b, a']$ are used together with the equation 1.2. First, all the elements $P_k^{ed}[a, b, a']$ for $a' \in [0, n - 1]$ must be computed with algorithm 5. The pseudocode shown in the algorithm 6 illustrates the computation of the resulting union of intervals and retrieving the final k-coverage.

Algorithm 5: Compute $P_k^{ed}[a, b, a']$ [1]

```
if $b - a < M - 1$ then
    for $b' := a' - 1$ to $a' + M - 2$ do
        if $D_{a,a'}[b, b'] \leq k$ then
            $P_k^{ed}[a, b, a'] = b'$
        end
    end
end
$c = \lceil a/M \rceil * M$
for $c' := a'$ to $a' + M - 1$ do
    $(b', d') := L_{c,c'}[b].pred(k - D_{a,a'}[c - 1, c' - 1])$
    if $d' \neq \infty$ then $P_k^{ed}[a, b, a'] = \max(P_k^{ed}[a, b, a'], b')$
end
$c' = \lceil a/M \rceil * M$
for $c := a$ to $a + M - 1$ do
    $(b', d') := L_{c,c'}[b].pred(k - D_{a,a'}[c - 1, c' - 1])$
    if $d' \neq \infty$ then $P_k^{ed}[a, b, a'] = \max(P_k^{ed}[a, b, a'], b')$
end
```

Algorithm 6: Given table P_k^{ed} for string S of lenght n compute $Coverage_k^{ed}(S[a, b], S)$

```
$(l, r) := (-1, -1)$
$coverage := 0$
for $a' := 0$ to $n - 1$ do
    $b' := P_k^{ed}[a, b, a']$
    if $b' < a'$ then continue
    if $a' > r$ then $coverage := coverage + (b' - a' + 1)$
    else $coverage := coverage + (b' - r)$
    $(l, r) := (a', b')$
end
$Coverage_k^{ed}(S[a, b], S) := coverage$
```

4.3.1.5 Restricted approximate covers

Algorithm 7 shows the pseudocode for computing $Q_{a,b}$ table presented in 3.1.2. It uses range minimum queries supporting sparse table RMQ which for an interval $[x, y]$ returns the minimum of $Q_{a,b}[x, y]$. This data structure is defined in 2.4 and its implementation is presented in 4.2.3. $Q_{a,b}$ table is represented by `class QTable` in the implementation.

4.3.2 Approximate covers under Hamming distance

In the implementation, algorithms and supporting data structures used for computing covers under Hamming distance, are defined under `namespace Hamming`.

4.3.2.1 Longest common prefix

Algorithm 8 shows the pseudocode for computing the table of *k-approximate longest common prefixes* of string S. The problem is defined in 3.3.1.

This algorithm was stated in [14]. Originally it computes longest common suffixes of all prefixes with at most k mismatches, thus it had to be adapted for the purpose of computing longest common prefixes of all suffixes with at most k mismatches. This change is made on line 7 where characters of the string are compared in reverse order.

`class LongestCommonPrefixes` contains the implementation of this algorithm.

4.3.2.2 k-coverage for every prefix

Algorithm 9 shows the pseudocode for computing k-coverage of every prefix of string S. Insertion and removal of a pair to and from data structures $\mathcal{D}_o$ and $\mathcal{D}_{no}$ are described in 3.3.2 and 3.3.3 respectively.

Elements from $\mathcal{B}$ (resp. $\mathcal{L}$) are removed in $\mathcal{O}(1)$ time using pointers stored in $\mathcal{A}$ (resp. $\mathcal{P}$).

In the implementation, $\mathcal{D}_o$ is represented by `class Overlapping` and $\mathcal{D}_{no}$ is represented by `class NonOverlapping`. The Q table is represented by `class QTable`.

4.3.2.3 k-coverage for every factor

In algorithm 10 there are only a few differences from algorithm 9. It should be noted that the algorithm 10 solves the problem of finding k-coverage of every prefix (4.3.2.2) during the first iteration.

For each new i the data structures $\mathcal{L}$, $\mathcal{D}_o$, $\mathcal{D}_{no}$ need to be set to their initial state. Meaning that $\mathcal{L}$ must be filled with $[0, \ldots, n]$, $\mathcal{D}_{no}$ is filled with all adjacent pairs from $\mathcal{L}$ and $\mathcal{D}_o$ is emptied ($sum_o = 0$).

On line 12 a condition is added that solves the problem of removal of 0 from $\mathcal{L}$ because 0 has no predecessor in $\mathcal{L}$ and therefore only the pair $(0, q_2)$ is removed and then the algorithm continues in removing the remaining values from $\mathcal{L}$.

Algorithm 7: For given string S of length S, factor $S[a, b]$, $M = \lfloor\sqrt{n/\log n}\rfloor$ and precomputed D-tables and L lists compute the smallest k for which $S[a, b]$ is a k-approximate cover of S [1]

```
for $i := n - 1$ downto $0$ do
    $Q_{a,b}[i] := \infty$
    $minQ := \infty$
    if $b < a - 1 + M$ then
        for $j := i$ to $i - 2 + M$ do
            $minQ$ := minimum of $minQ$ and $Q_{a,b}[j + 1]$
            $Q_{a,b}[i]$ := minimum($Q_{a,b}[i]$, maximum($D_{a,i}[b, j]$, $minQ$)
            update $RMQ$ at index $i$ with $Q_{a,b}[i]$
        end
    end
    $s = \lceil a/M\rceil * M$
    for $s' := i$ to $i - 1 + M$ do
        if $L_{s,s'}[b]$ is empty then continue
        $(p1, p2)$ := binary search for the first pair in $L_{s,s'}[b]$ such that
          $p1 \geq RMQ[i + 1 \dots p2 + 1] - D_{a,i}[s - 1, s' - 1]$ or last pair in
          $L_{s,s'}[b]$)
        $(q1, q2)$ := Predecessor 4.3.1.2 of $p1$ in $L_{s,s'}[b]$ or $(p1, p2)$ if no
          predecessor exists
        $Q_{a,b}[i]$ := minimum($Q_{a,b}[i]$, maximum($D_{a,i}[s - 1, s' - 1] + p1$,
          $RMQ[i + 1 \dots j + 1]$))
        $Q_{a,b}[i]$ := minimum($Q_{a,b}[i]$, maximum($D_{a,i}[s - 1, s' - 1] + q1$,
          $RMQ[i + 1 \dots j + 1]$))
        update $RMQ$ at index $i$ with $Q_{a,b}[i]$
    end
    $c' = \lceil i/M\rceil * M$
    for $c := a$ to $a - 1 + M$ do
        if $L_{c,c'}[b]$ is empty then continue
        $(p1, p2)$ := binary search for the first pair in $L_{c,c'}[b]$ such that
          $p1 \geq RMQ[i + 1 \dots p2 + 1] - D_{a,i}[c - 1, c' - 1]$ or last pair in
          $L_{c,c'}[b]$)
        $(q1, q2)$ := Predecessor 4.3.1.2 of $p1$ in $L_{c,c'}[b]$ or $(p1, p2)$ if no
          predecessor exists
        $Q_{a,b}[i]$ := minimum($Q_{a,b}[i]$, maximum($D_{a,i}[c - 1, c' - 1] + p1$,
          $RMQ[i + 1 \dots j + 1]$))
        $Q_{a,b}[i]$ := minimum($Q_{a,b}[i]$, maximum($D_{a,i}[c - 1, c' - 1] + q1$,
          $RMQ[i + 1 \dots j + 1]$))
        update $RMQ$ at index $i$ with $Q_{a,b}[i]$
    end
end
```

Algorithm 8: For string S of lenght n and number k compute the lcp table where $lcp[i][j]$ is the length of the longest common k-approximate prefix of $S[i, n-1]$ and $S[j, n-1]$.

```
for $d := -n + 1$ to $n - 1$ do
    $i := max(-d, 0) + d$
    $j := max(-d, 0)$
    $s := 0$
    $q := \varnothing$ // empty queue
    for $p := 0$ to $min(n - i, n - j)$ do
        if $S[n - i - p - 1] \neq S[n - j - p - 1]$ then
            if $k$ is $0$ then
                $s := p + 1$
            else if $|q|$ is $k$ then
                $s := q.front() + 1$
                $q.pop()$
            end
            $q.push(p)$
        end
        $lcp[n - i - p - 1].push(p - s + 1)$
    end
end
```

Algorithm 9: Given Q table (see 3.3.4) constructed from $PREF_k$ table, compute the k-coverage of $S[0, l-1]$ in S

```
for $l := 1$ to $n$ do
    foreach $q \in Q[l-1]$ do
        $q_1 :=$ predecessor of $q$ in $\mathcal{L}$
        $q_2 :=$ successor of $q$ in $\mathcal{L}$
        remove $q$ from $\mathcal{L}$ using $\mathcal{P}[q]$
        if $q - q_1 < l$ then remove $(q_1, q)$ from $\mathcal{D}_o$
        else remove $(q_1, q)$ from $\mathcal{D}_{no}$
        if $q_2 - q < l$ then remove $(q, q_2)$ from $\mathcal{D}_o$
        else remove $(q, q_2)$ from $\mathcal{D}_{no}$
        if $q_2 - q_1 < l$ then insert $(q_1, q_2)$ into $\mathcal{D}_o$
        else insert $(q_1, q_2)$ into $\mathcal{D}_{no}$
    end
    foreach $(i, j) \in \mathcal{B}[l]$ do
        remove $(i, j)$ from $\mathcal{D}_{no}$
        add $(i, j)$ to $\mathcal{D}_o$
    end
    coverage of $S[0, l-1] := sum_o + num_{no} \cdot l$
end
```

Algorithm 10: Compute k-coverage of every factor of string S under Hamming distance

```
for $i := 0$ to $n - 1$ do
    compute $Q$ from $[lcp_k(i, 0), \ldots, lcp_k(i, n - 1)]$
    reset $\mathcal{L}$, $\mathcal{D}_o$, $\mathcal{D}_{no}$
    for $l := 1$ to $n - i$ do
        foreach $q \in Q[l - 1]$ do
            $q_1 :=$ predecessor of $q$ in $\mathcal{L}$
            $q_2 :=$ successor of $q$ in $\mathcal{L}$
            remove $q$ from $\mathcal{L}$ using $\mathcal{P}[q]$

            if $q_2 - q < l$ then remove $(q, q_2)$ from $\mathcal{D}_o$
            else remove $(q, q_2)$ from $\mathcal{D}_{no}$
            if $q = 0$ then continue
            if $q - q_1 < l$ then remove $(q_1, q)$ from $\mathcal{D}_o$
            else remove $(q_1, q)$ from $\mathcal{D}_{no}$
            if $q_2 - q_1 < l$ then insert $(q_1, q_2)$ into $\mathcal{D}_o$
            else insert $(q_1, q_2)$ into $\mathcal{D}_{no}$
        end
        foreach $(i, j) \in \mathcal{B}[l]$ do
            remove $(i, j)$ from $\mathcal{D}_{no}$
            add $(i, j)$ to $\mathcal{D}_o$
        end
        coverage of $S[i, l - 1] := sum_o + num_{no} \cdot l$
    end
end
```

Chapter 5

Experimental evaluation

In this chapter, algorithms presented in chapter 3 whose implementation is described in chapter 4 are evaluated.

A simple algorithm for generating random strings from a specified alphabet and length was used to generate random strings. The time complexity of the algorithms was tested and the results are visualized in graphs.

5.1 Testing environment

The implementation is tested on an operating system macOS Big Sur version 11.2.1 using:

- **CPU**: 2,3 GHz Intel Core i5
- **Memory**: 8GB 2133MHz

The implementation was compiled with Apple clang version 12.0.5 using -O3 -DNDEBUG options.

5.2 k-approximate covers under Hamming distance

In this section, the algorithm for computing all k-approximate covers under Hamming distance is experimentally evaluated with different parameters. In figure 5.1 the algorithm is tested on strings generated using two alphabets of different sizes (2 and 5). From the data acquired, it can be concluded, that the algorithm does not depend on the size of the alphabet Σ as the visualized dependency curves scale similarly. This behaviour was expected.

Results for strings generated using alphabet $\Sigma = \{a, b\}$ are shown in figure 5.2. From the data acquired, it can be concluded that k also plays no role in the algorithm performance.

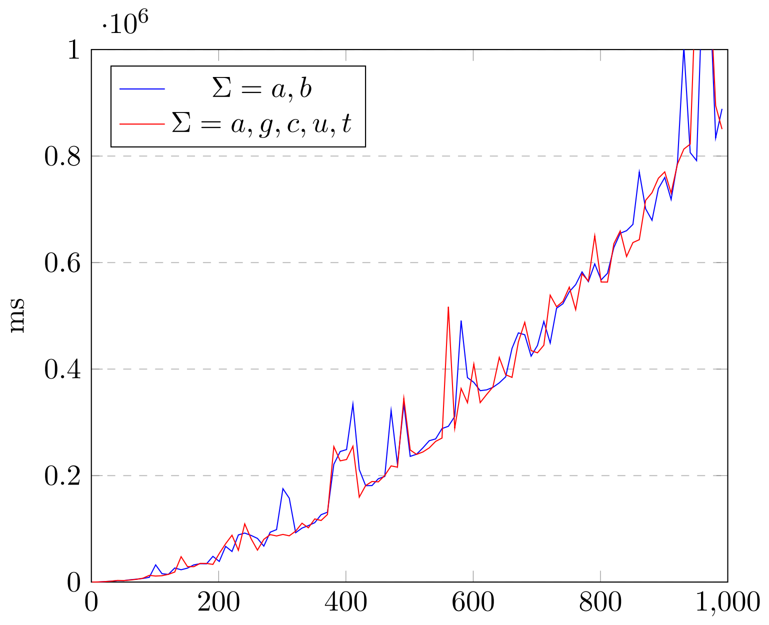

Figure 5.1: 1-approximate covers under Hamming distance

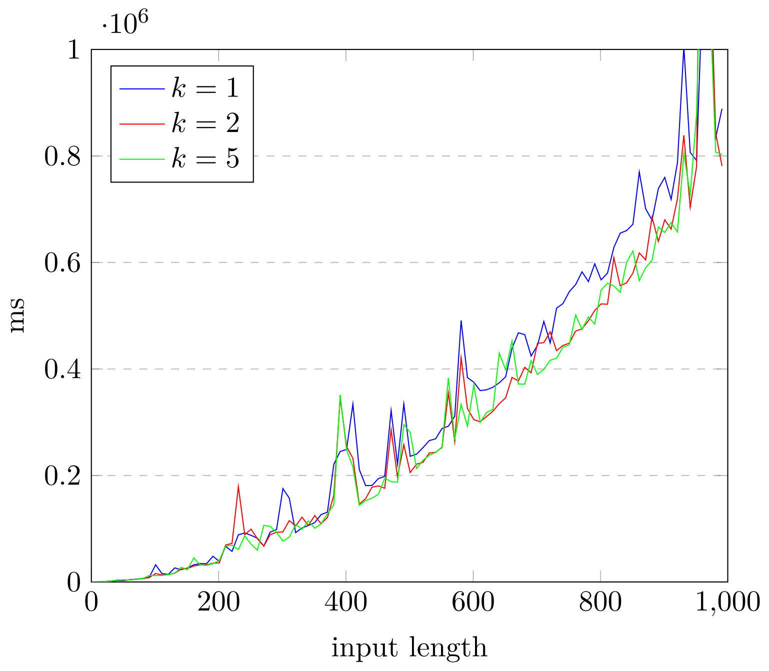

Figure 5.2: k-approximate covers under Hamming distance

5.3 k-approximate covers under weighted edit distance

In figure 5.3, the duration of execution of the algorithm for computing k-approximate covers under weighted edit distance was tested on randomly generated strings with the alphabet $\Sigma = \{a, b\}$ and $k = 5$. The acquired data confirm the stated complexity of the algorithm and that it only depends on the size of the input and not on the weights chosen.

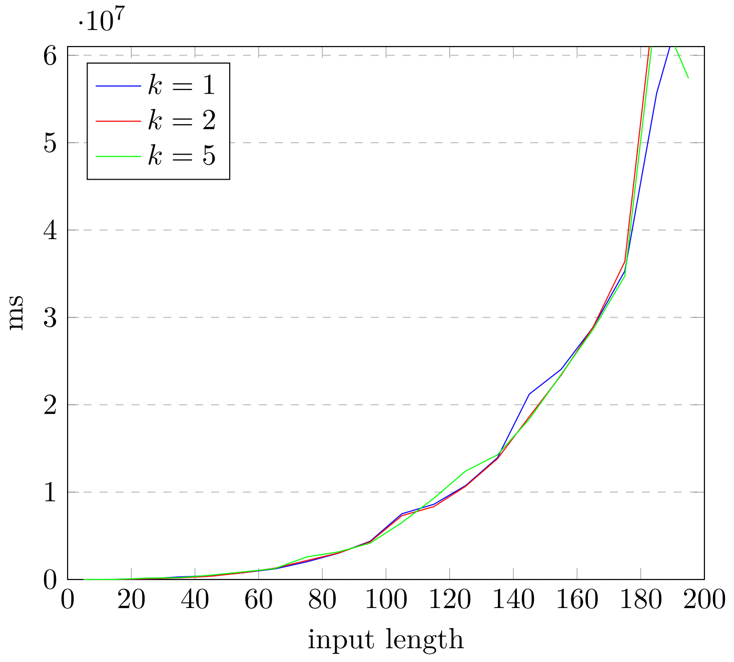

Figure 5.3: k-approximate covers under weighted edit distance with different k and unit costs

In figure 5.4 the same alphabet $\Sigma = \{a, b\}$ is used for generating each string but the weights — costs of operations delete, insert and substitute — are different in each test case. In one test car the weights are all of value 1 and in the other the weights are 3, 2, 1 respectively. These result confirm that the execution time does not depend on the weights provided.

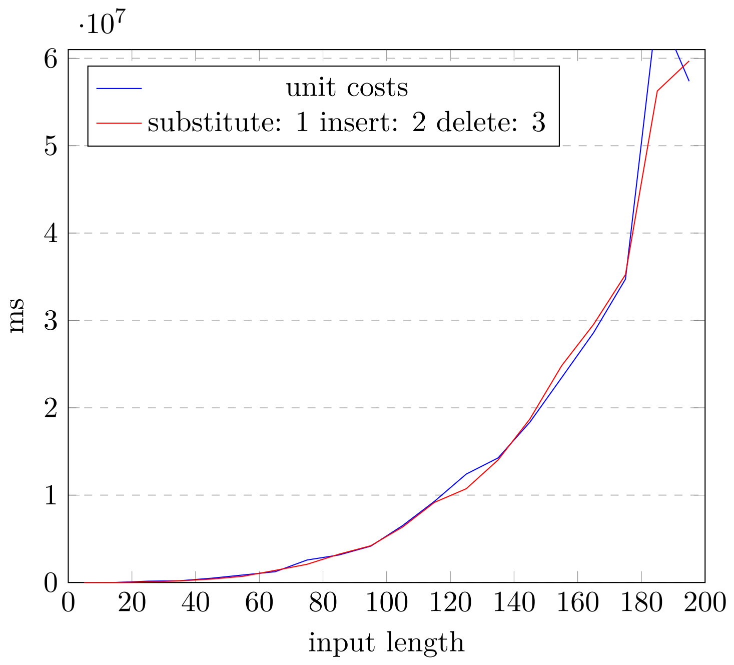

Figure 5.4: 5-approximate covers under weighted edit distance with unit costs and custom costs for different operations

5.4 Restricted approximate covers under weighted edit distance

Figure 5.5 shows the comparison of an algorithm computing k for each factor of S such that the factor is k-approximate cover of S. For each case, different alphabet was used. Again, it can be seen that there is no dependency on the alphabet and the results comply with the stated complexity.

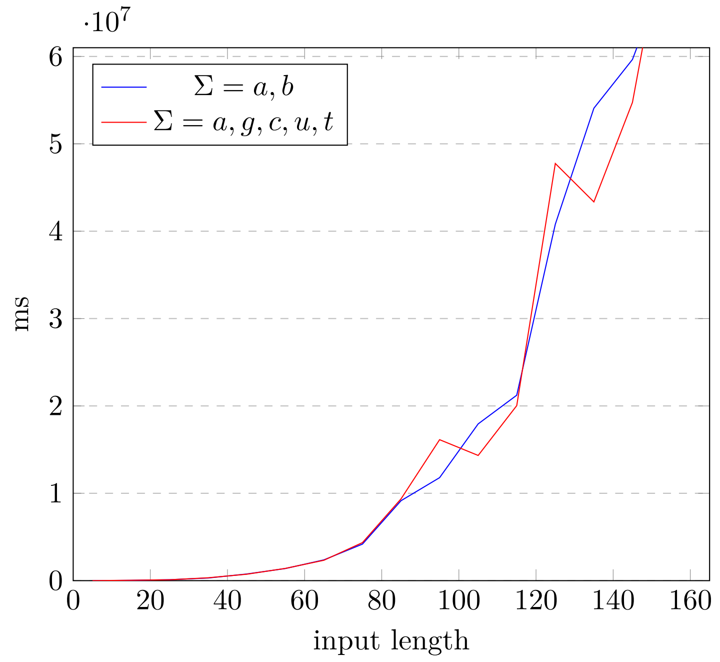

Figure 5.5: restricted approximate covers under weighted edit distance

Conclusion

The notion of covers was presented in the beginning of the work and was set in the context of other periodicities along with the description of different metrics under which approximate periodicities can be measured. The goal of this work was to also describe algorithms presented in [1] while adding examples and different views of perceiving the problems. Some of the described algorithms were also implemented in this work. The algorithms that were implemented are for computing k-approximate covers under Hamming and weighted edit distance along with an algorithm for computing the values k for all factors of a string S such that the factor is a k-approximate cover of S.

In the end, the implementations of these algorithms were tested with different parameters such as k, size of the alphabet and length of the input. The evaluation confirmed the complexities of these algorithms and that they only depend on the length of the input.

Possible future extensions

This work does not implement the algorithm for computing k-approximate covers under Levenshtein distance. For implementing this algorithm an effective implementation of h-waves construction is expected. This can be achieved using a suffix tree with linear construction and constant retrieval of lowest common ancestor after linear preprocessing of the suffix tree.

Apart from covers, the algorithms can also be extended to compute k-approximate seeds. [1].

Bibliography

1. KĘDZIERSKI, Aleksander; RADOSZEWSKI, Jakub. k-Approximate Quasiperiodicity under Hamming and Edit Distance. *arXiv preprint arXiv:2005.06329* [online]. 2020 [visited on 2021-04-15]. Available from: `https://arxiv.org/pdf/2005.06329.pdf`.
2. HAMMING, R. W. Error detecting and error correcting codes. *The Bell System Technical Journal*. 1950, vol. 29, no. 2, pp. 154–155. Available from DOI: `10.1002/j.1538-7305.1950.tb00463.x`.
3. SCHÜTZE, Hinrich; MANNING, Christopher D; RAGHAVAN, Prabhakar. *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008.
4. APOSTOLICO, Alberto; EHRENFEUCHT, Andrzej. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*. 1993, vol. 119, no. 2, pp. 247–265. ISSN 0304-3975. Available from DOI: `https://doi.org/10.1016/0304-3975(93)90159-Q`.
5. SMYTH, W.F. Computing regularities in strings: A survey. *European Journal of Combinatorics*. 2013, vol. 34, no. 1, pp. 3–14. ISSN 0195-6698. Available from DOI: `https://doi.org/10.1016/j.ejc.2012.07.010`. Combinatorics and Stringology.
6. CHRISTOU, M.; CROCHEMORE, M.; ILIOPOULOS, C.S.; KUBICA, M.; PISSIS, S.P.; RADOSZEWSKI, J.; RYTTER, W.; SZREDER, B.; WALEŃ, T. Efficient seed computation revisited. *Theoretical Computer Science*. 2013, vol. 483, pp. 171–181. ISSN 0304-3975. Available from DOI: `https://doi.org/10.1016/j.tcs.2011.12.078`. Special Issue Combinatorial Pattern Matching 2011.
7. SIM, Jeong-Seop; PARK, Kun-Soo; KIM, Sung-Ryul; LEE, Jee-Soo. Finding approximate covers of strings. *Journal of KIISE: Computer Systems and Theory*. 2002, vol. 29, no. 1, pp. 16–21. Available also from:

`http://www.koreascience.or.kr/article/JAKO200211920915073.page`.

8. WAGNER, Robert A.; FISCHER, Michael J. The String-to-String Correction Problem. *J. ACM*. 1974, vol. 21, no. 1, pp. 168–173. ISSN 0004-5411. Available from DOI: `10.1145/321796.321811`.
9. UKKONEN, Esko. Algorithms for approximate string matching. *Information and Control*. 1985, vol. 64, no. 1, pp. 100–118. ISSN 0019-9958. Available from DOI: `https://doi.org/10.1016/S0019-9958(85)80046-2`. International Conference on Foundations of Computation Theory.
10. LANDAU, Gad M; MYERS, Eugene W; SCHMIDT, Jeanette P. Incremental string comparison. *SIAM Journal on Computing*. 1998, vol. 27, no. 2, pp. 557–582. Available also from: `https://doi.org/10.1137/S0097539794264810`.
11. UKKONEN, Esko. On-line construction of suffix trees. *Algorithmica*. 1995, vol. 14, no. 3, pp. 249–260.
12. SCHIEBER, Baruch; VISHKIN, Uzi. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*. 1988, vol. 17, no. 6, pp. 1253–1262.
13. KOGLER, Jakob. *Sparse Table* [online]. 2020 [visited on 2021-04-15]. Available from: `https://cp-algorithms.com/data_structures/sparse-table.html`.
14. FLOURI, Tomas; GIAQUINTA, Emanuele; KOBERT, Kassian; UKKONEN, Esko. Longest common substrings with k mismatches. *Information Processing Letters*. 2015, vol. 115, no. 6, pp. 643–647. ISSN 0020-0190. Available from DOI: `https://doi.org/10.1016/j.ipl.2015.03.006`.

Appendix **A**

Contents of enclosed media

```
README.md .......................... file with description of the contents
├── implementation ........................... directory with source code
│   ├── README.md ........... file with the compilation and usage description
│   └── other source files and folders
└── text ....................................... folder with the thesis text
    ├── thesis.pdf ................................. thesis in PDF format
    └── source ..................... folder with the source code of the thesis
```