FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace algoritmů vyhledávání v řetězcích s konstantní pracovní pamětí navíc
Student:	Jan Jirák
Vedoucí:	Ing. Jan Trávníček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Nastudujte metody vyhledávání v řetězcích využívající při svém běhu pouze konstantní množství paměti navíc podle [1].
Navrhněte v C++ datové struktury, které tyto algoritmy používají.
Implementujte algoritmy z [1] v C++ v Algoritmové knihovně [2] s využitím a případnou úpravou dostupných datových struktur.
Otestujte implementaci pomocí Vámi vhodně generovaných náhodných řetězců a pomocí Vámi zvoleného standardního korpusu testovacích řetězců.

Seznam odborné literatury

[1] Cantone, Domenico, and Simone Faro. "IT'S ECONOMY, STUPID!": SEARCHING FOR A SUBSTRING WITH CONSTANT EXTRA SPACE COMPLEXITY.
[2] Trávníček, Jan, Pecka, Tomáš, et. al. Algorithms Library Toolkit. Dostupné online: https://gitlab.fit.cvut.cz/algorithms-library-toolkit

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 27. ledna 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementation of string searching algorithms with constant extra space complexity

Jan Jirák

Department of Computer science
Supervisor: Ing. Jan Trávníček, Ph.D.

May 25, 2020

Acknowledgements

Thanks to everyone who helped me on my way.

In the first place, I wish to express my sincere thanks to my supervisor Ing. Jan Trávníček, Ph.D. for providing me every advice, guidance needed, and the patience he had for me. I am also grateful to my family for encouraging me while I was finishing this work and for all the kind environment they created for me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 25, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Jirák, Jan. *Implementation of string searching algorithms with constant extra space complexity.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Cílem této práce je uvedení do problematiky přesného vyhledávání v řetězci s požadavkem použití pouze konstantního množství paměti. Práce se dále zabývá implementací probíraných algoritmů, které tento problém řeší. Tyto implementace budou integrovány do Algorithms Library Toolkit vyvíjeného na FIT ČVUT v Praze. Implementace jsou založeny čistě na pseudokódech z odkazované literatury.

Klíčová slova implementace algoritmů pro vyhledávání v řetězci, stringologie, přesné vyhledávání, konstantní pamět, Algorithms Library Toolkit

Abstract

The goal of this thesis is a theoretical introduction to searching in string exact match with only constant space memory given. Next the thesis considers the implementation of algorithms, which solve the problem. Algorithms will be integrated into Algorithms Library Toolkit developed at FIT CTU in Prague, where will be implemented based on pseudocodes from referred literature.

Keywords implementation of string-matching algorithms, stringology, exact matching, constant space matching, Algorithms Library Toolkit

Contents

List of Tables

Introduction

Given a text T and pattern P, the problem of finding all occurrences of P in T is called string matching. String matching is the base of the whole scientific field called stringology. In a lot of real-world situations string matching is related. To mention a few areas as data compression, computer vision, molecular biology, speech recognition, and others.

There exist a lot of variations of pattern matching, in this thesis, we will consider only problem called exact matching, the problem, where you get text and pattern, our task is to say whether the pattern occurs as a substring in the text. Other problems like approximate matching or matching based on regular expression will not be touched.

We will not only be looking for only exact matching algorithms that are fast (with linear time complexity), but also with the memory economic solutions (constant space complexity). We will show that it is not only possible to achieve these results, but that some of these algorithms are in practical use faster than algorithms with bigger memory consumption.

One question surely comes to your mind "Why to bother with the restriction of constant memory?". For this question are lots of answers. In this time when Iot is getting big, there is a huge need for small embedded devices, sometimes you just can't afford to have linear memory.

The main goal of this bachelor's thesis is to implement algorithms described in the thesis [1]. We will understand the theoretical background, then implement them. In the end we will test not only the correctness, but also do some speed benchmarks to be able to compare their performance in use on commons strings.

The second goal is to provide an understandable overview of the subject *effective constant space* string matching algorithm, where no other resources are needed.

In chapter 1 we start by introducing the basic definitions and properties of the string, then we proceed right to the algorithms description with their mathematical deduction. In chapter 2 we implement algorithms from chapter

1 to Algorithmic Library Toolkit based on previously presented pseudocodes. In chapter 3 we describe how was the implementation tested. In the appendix you can find the content of the enclosed CD where you can find built ALT.

Chapter 1

Theoretical part

In this chapter we gradually build the theoretical foundation around stringology and then apply it on specific algorithms mentioned in [1]. First we start with some general properties of the string and then we will discuss all implemented algorithms. In this section only **pseudocode** is considered.

1.1 Definitions

Let's start with a few basic definitions. These definitions are defined according to [2].

Definition 1.1.1. *Let Σ be a finite set of symbols, then we call Σ an input alphabet.*

Example. *Set of $\{0, 1\}$, UTF-8, greek alphabet, $\{a, b, c\}$.*

Definition 1.1.2. *String is a finite sequence over some alphabet Σ.*

Example. *"Hello" over $\{H, e, l, o\}$, "010011" over $\{0, 1\}$.*

Definition 1.1.3. *Let l be the number of characters in string s, then we call l to be the lenght of s, the lenght of s is denoted by $|s|$.*

Example. $|Hello| = 5$.

Definition 1.1.4. *Empty string (string of the lenght 0) is denoted by ϵ.*

Definition 1.1.5. *Let s be a string over some σ, then we denote its i-th position by $s[i]$ and we denote by $s[i..j]$ the factor $s[i+1]s[i+2]...s[j]$ of s.*

Example. *Let s be $Hello$, then $s[2..4] = ll$ and $s[2] = e$.*

Definition 1.1.6. *Word x is an factor of y iff exist i such that $x = y[i..i+m]$, where m is length of x. We say that x occurs on position i in y.*

Definition 1.1.7. ***Lexicographical order*** *is a way of ordering strings based on ordered alphabets of the string. We define that for strings x and y, $x < y$ iff $\exists i$, such $x[i] < y[i]$ and $\forall j, 1 \leq j < i$ holds, that $x[j] = y[j]$. If the two strings are not of the same lenght, then we artificially append characters \$ to end of the shorter string, such that $\forall \alpha \in \Sigma$ holds, that $\$ < \alpha$.*

Definition 1.1.8. *Let pattern and text of lengths m, n be two strings over the same alphabet.* ***String matching*** *is problem of deciding whether pattern occurs in text. Another version of this problem is its extencion from simple decidibility problem to problem, where you also have to report all occurences of pattern in text.*

Definition 1.1.9. *Let's say we have an text y and pattern x, if the characters $x[i]$ are aligned with $y[s+i]$, where $i \in \{1, ..., m\}$, we denote this alignment as shift of x in y. If $x = y[s..s+m-1]$, we say that shift s is valid.*

All string matching related algorithms rely on some mathematical properties of string, now we define a few such key properties.

Definition 1.1.10. ***Period*** *of a string s is a number p, $1 \leq p \leq |s|$, such that $s[i] = x[i+p]$ for all $i \in \{1, ..., |s| - p\}$. We denote by $period(s)$ the smallest period of s. If $period(s) \leq \frac{|s|}{2}$, then we say that s is periodic otherwise nonperiodic.*

Example. *Let $s = abcabcabcabc$, then all the periods of s are $3, 6, 9, 12$, $period(s) = 3$.*

Definition 1.1.11. ***Border*** *of a string s is a substring that is simultaneously a prefix and suffix of s. Note that ϵ and whole s are also borders of s. We denote by $Border(s)$ the longest nontrivial border of s.*

Example. *Take s from previous example, then borders are $abcabcabcabc, abcabcabc, abcabc, abc, \epsilon$. $Border(s) = abcabcabc$.*

Remark. Have you noticed the correlation between *borders* and *periods*? It is obvious that border is dual version of period. To every *period* corresponds *border* with lenght of $s - p$. From previous two examples:

Period	Border
3	$abcabcabc$
6	$abcabc$
9	abc
12	ϵ

Table 1.1: Corresponding borders

Lemma 1.1.1. ***Periodicity lemma*** *Let p and q be two periods of the string x. If $p + q < |x|$, then $gcd(p, q)$ is also a period of x.*

Proof. If $p = q$ the conclusion is trivially satisfied. W.l.o.g. let us assume that $p > q$. We show that condition $p + q < |x|$ implies that $p - q$ is also period of x. Let i be some position in x, then either $1 \leq i - q$ or $i + p \leq |x|$. In the first case $x[i] = x[i-q] = x[i+p-q]$, in the second case $x[i] = x[i+p] = x[i+p-q]$, so $p - q$ is also period of x. Rest of the proof is only showing the property that $gcd(p, q) = gcd(p - q, q)$ and using this in proper induction. □

Definition 1.1.12. *Let x be a string and l be an integer such that $0 \leq l \leq |x|$, then an integer r is called local period of x at position l iff $x[i] = x[i+r]$ for all i such that $1–r+1 < i < 1$ and such that both sides of the equality are defined. The local period of x at position l is the smallest local period at position l. It is denoted $r(x, l)$.*

Definition 1.1.13. *String z is a prefix period of w if it is basic and z^k is a prefix of w.*

Definition 1.1.14. *Let w be some string, then we define reach for $p \leq |w|$ as*

$$reach_w(p) = max(q \leq |w||[0, p]_w is a period of [0, q]_w)$$

.

1.2 Algorithms

A lot of string matching algorithms use some kind of pattern preprocessing. They create shift tables, which are used for computing minimal safe shift increments, that algorithm can perform in every step. Mostly used algorithms like Knuth–Morris–Pratt or Boyer–Moore, compute these tables in linear time and space, but for our purposes that not enough.

Next presented algorithms use all only constant space. They achieve that either by computing the table entry on the run, or by using some properties of string. Take into consideration that not all of the next algorithms are exactly running in the worst case in linear time, but in practice they are often shown as really fast.

The first six of these algorithms in this section are not designed by the authors of [1], the last two are.

In the following material we denote the text by y and pattern by x. The length of the text y is denoted by n and the length of the pattern x is denoted by m.

1.2.1 Not-So-Naive algorithm

The Not-So-Naive algorithm is a little bit smarter version of the Naive algorithm. Let us first introduce the Naive algorithm from [3].

Naive or Brute force algorithm checks all the position from 1 to $n - m$, it start with position 1, naively checks whether $x = y[1..m]$ and then shifts his sliding window by 1 to the right. It runs in $\mathcal{O}(nm)$ and needs no pattern preprocessing.

The Not-So-Naive works similarly, with the exception of two cases in matching phase, where the shift can be advanced by two positions instead of rigid one as seen in Naive. Cases:

1. Assume that $x[0] \neq x[1]$ and $x[0] = y[s] \wedge x[1] = y[s+1]$. When the matching phase of shift s ends, we can safety advance shift s by two positions. That is because $x[0] \neq x[1] = y[s+1]$.

2. Assume that $x[0] = x[1]$, then if we match $x[0] = y[s]$ and next character match fails ($x[1] \neq y[s+1]$), we can also advance shift s by two positions.

Not-So-Naive needs constant space and time preprocessing of $|x|$. On one side it can run in worst case $\mathcal{O}(nm)$, but in practise is deemed as in average fast algorithm. Note that comparisons are done in order $1, 2, ..., m-1, 0$.

Algorithm 1 Not-So-Naive

```
                                                  ▷ Preprocess pattern
if x[0] = x[1] then                               ▷ Case 2
    k ← 2
    ell ← 1
else                                              ▷ Case 1
    k ← 1
    ell ← 2
end if
while i ≤ n − m do
    if x[1] ≠ y[1] then
        i += case
    else
        if Check the rest of positions then Report match
        end if
        i += ell
    end if
end while
```

1.2.2 The Dogaru algorithm

Another very simple string matching algorithm is the Dogaru algorithm from the article [4]. This is another algorithm, which uses no preprocessing and in worst case run in $\mathcal{O}(nm)$.

It works similarly as the Naive algorithm, start searching in the text from left to right. If the pattern is matched, the shift is advanced by one position. However if lets say mismatch on position j between $x[j]$ and $y[s+j]$ is found, then algorithm searches for $x[j]$ in $y[s+j+1]$. If no occurrence of $x[j]$ is found, the algorithm terminates, else let's say that the occurrence was found on pos s'. Next algorithm naively checks left and right side of s', whether pattern starts on position $s'-j$. If not search for $x[j]$ is resumed on position $s'+1$.

Algorithm 2 The Dogaru

```
i, j = 0
while j < m ∧ x[j] = y[i] do                ▷ Look for the match at pos 0
    if j = m then
        Report match
        i = i − m + 1
        j = 0
    end if
end while
while i ≤ n − m + j ∧ x[j] ≠ y[i] do        ▷ Look for pos[j]
    i+ = 1
end while
```

Check naively position $i-j$ for match, if found report match and go to first *while* with $i = i-j+1, j = 0$, else go to second *while* and continue looking for $pos[j]$.

What is a little bit surprising is the fact that this algorithm does not find all occurrences of pattern. It is caused, because when the algorithm finds a valid shift in the text, it advances the window by the whole m. For example if you take $x = aa$ and $y = aaaaaa$, it finds a match at position 1 and then skip position 2. If m would be replaced by the 1, the algorithm would find all occurrences.

Example. $x = abc$, $y = abcabbbbcabcaa$
abcabbbbcabcaa
abc match
ab? mismatch at c
abc mismatch at a looking for next c
abc match

1.2.3 The Two-Way algorithm

In this section we will propose another algorithm, which has the following properties. It is linear in time $\mathcal{O}(|y| + |x|)$ and maximum number of comparisons is $2|y| + 5|x|$. Only constant memory is needed. This algorithm was designed in [5].

Let l be critical position (will be explained later in this section), then $x = x_l x_r$ where x_l is prefix of x and $|x| = l$. Searching for position t in y, where x begins takes two phases. In the first phase algorithms compares y with x_r from left to right, if succesful then compares r_l from right to left. In case of mismatch in x_r, sliding window is shifted by k, where k is number of characters matched in x_r. In case of mismatch in x_l, shift of $per(x)$ is performed. In case of full match, also shift of $per(x)$ is performed.

More formal description is given in pseudocode 3. The meaning of used variables pos, i, j, s is following. Pos is the position in y at which match is tested, i is an index in x_r, j is an index in x_l and s is a length of matching prefix of pattern with text. s is used in order to not check some comparisons again, when shift is performed as a result of mismatch in x_l.

Algorithm 3 Two-Way

```
1: per = per(x) and l < per is a critical position, both previously computed.
2: pos ←= 0, s ← 0
3: while pos + |x| ≤ |y| do
4:     i ← max(l, s)
5:     while i < |x| ∧ x[i] = y[pos + i] do i ← i + 1
6:     end while
7:     if i ≤ |x| then                                   ▷ Mismatch in x_r
8:         pos ← pos + max(i − l, s − p + 1)
9:         s ← 0
10:    else                                              ▷ Match of x_r, check x_l
11:        j ← l
12:        while s < j ∧ x[j] = y[pos + j] do j ← j − 1
13:        end while
14:        if j ≤ s then report match at pos
15:            pos ← pos + per
16:            s ← |x| − per
17:        end if
18:    end if
19: end while
```

In order to be able to understand how Two-Way algorithms works, first we have to build some theory. Let's start by introducing the concept of *self-maximal suffixes.*

Definition 1.2.1. *Let x be a string over alphabet Σ, then we denote by $MaxSuf(x)$, the lexicographically maximal suffix of x. We say that x is self-maximal iff $MaxSuf(x) = x$.*

Definition 1.2.2. ***Factorization*** *of a string x is any pair (u, v), such that $x = uv$.*

Theorem 1.2.1. ***Critical factorization theorem*** *says, that for each string x there exists a postion l, $0 \leq l < per(x)$ such that*

$$r(x, l) = per(x).$$

Position l such that $r(x, l) = per(x)$ is called a critical position of x.

Example. *Let $x = abaabaa$, $per(x) = 3$ and it has three critical positions $2, 4, 5$. Note that for the rest positions, there is alway smaller local period that 3.*

Lets now prove that algorithm 3 finishes in finite time and outputs correct results.

The algorithm will finish in finite time, because every run of the while statement at line 3 increments *pos* by a positive number. Either line 8 or line 15 in this while is executed and $per(x) > 0 \wedge i - l > 0$.

Now we should prove the correctness of the algorithm, but we will just refer the curious reader to the original article [5]. The second way to prove this is by using properties of *maximal suffix* from [2] and showing that the shifts are correct.

Next we need to prove it's linearity, so we will show that the algorithm 3 runs in $2|y|$ when we have the factorization $x = x_l x_r$ already precomputed.

First we prove, that every comparison done at line 3 strictly increases the value of $pos + i$. This is obvious if the letters $x[i]$ and $y[pos + i]$ coincide and $i < |x|$. If letters coincide but $x[i]$ is the last character of x, variable i is increased and at line 15 *pos* is increased by *per* and variable i is decreased by at most *per*, because of s at line 4. If mismatch occurs in the right part of the pattern, let i' be the value of the variable i, when this occurs. Then the variable *pos* is increased by at least $i' - l$ at line 8 and the variable i is in the next run of the loop decreased at most $i' - l - 1$ at line 2. So the number of comparisons at line 3 is at most $|y| - |x_l|$, because initial value of $pos + i$ is $|x_l| + 1$ and last value is $|y|$.

Secondly we take a look at line 12, where x_l is being compared. Because in at most $|x_l|$ steps will be performed shift of length *per* and variable *per* is more than $|x_l|$, at most $|y|$ comparisons are performed here.

This implies that the maximum number of character comparisons is $2|y|$.

Now let's take a look at the proof of theorem 1.2.1. This proof is mentioned here, because it will give us a method, how to practically compute the critical factorization. We will use proof using maximal suffixes of a string.

First we need to know following lemma.

Lemma 1.2.1. *Let v be the alphabetically maximal suffix of x and let $x = uv$. Then no nonempty string is both a suffix of u and a prefix of v. In other words, the string uv does not overlap.*

Proof. We will use the proof by contradiction. By the definition v is the maximal suffix of x and w to be the both a suffix of u and a prefix of v. We can denote $v = wt$. From the equation $wv < v$ we get $wwt < wt$, so if we omit starting w, we get $wt < t = v < t$, that leads to contradiction if w is nonempty string, because then t is some shorter maximal suffix of x. □

Lemma 1.2.2. *For the orderings $\leq$ and $\subseteq$, we have*

$$w \leq w' \wedge w \subseteq w'$$

iff w is a prefix of w'.

Now we can prove the theorem.

Theorem 1.2.2. *Let $\leq$ be an alphabetical ordering and let $\subseteq$ be the alphabetical ordering obtained by reversing the order $\leq$ on Σ. Let x be a nonempty string on Σ. Let v (resp., v') be the alphabetically maximal suffix of x according to the ordering $\leq$(resp., $\subseteq$). Let $x = uv = u'v'$.*

If $|v| \leq |v'|$, then (u, v) is a critical factorization of x. Otherwise, (u', v') is a critical factorization of x. Moreover, $|u| < per(x)$ and $|u'| < per(x)$.

Proof. In the case that x is a power single letter(case with $per(x) = 1$) any factorization is critical.

Lets now assume w.l.o.g. that $|v| \leq |v'|$. First let us prove that $u \neq \epsilon$. If $u = \epsilon$, then we have $x = v = v'$. This means, that $per(x) = 1$ and this case was already resolved.

Let r be the local period at (u, v). Thanks to 1.2.1 we cannot have $r \leq |u| \wedge r \leq |v|$, parts u and v would overlap. Since v is alphabeticaly maximal, it cannot be a factor of u, so $r > |u|$, else v would be foctor of u. Let z be the shortest string such that v is a prefix of zu or zu is a prefix of v. Then, $r = |zu|$. We now distinguish two cases according to $r > |v|$ or $r \leq |v|$.

Case $r > |v|$: In this situation, by the definition of r, the string u cannot be factor of v. The integet $|uz|$ is a period of uv since uv is a prefix of $uzuz$. The period of uz cannot be shorter than $|uz|$, because this quantity is the local period at (u, v). Hence, $per(uv) = |uz| = r$. So factorization (u, v) is critical.

Case $r \leq |v|$: The string u is a factor of v. We only need to show that $|zu|$ is a period of x. Remember $x = uv = u'v'$ for orderings $\leq$ and $\subseteq$. Let $u = u'u''$ and $v = zuz'$. By the definition of v', the suffix $u''z'$ of uv satisfies

$$u''z' \subseteq v' = u''v$$

hence $z' \subseteq v$. By the definition of v, we also have $z' \leq v$. By the observations made at the beqinning of the proof, these two inequalities imply that z' is a prefix of zuz'. Hence, z' is a prefix of a long enough repetion of zu's. Since $x = uzuz'$, this shows that uz is a period of x.

Now when we proved the theorem, we can reduce the computation of a critical factorization to the computation of two maximal suffixes for ordering $\leq$ and $\subseteq$. Such a computation of a critical factorization is sometimes called *magic decomposition.* □

The only things left are the computation of $per(x)$ and the computation of maximal suffixes for orderings $\leq$ and $\subseteq$.

We will now describe an algorithm for computation of the maximal suffix of a string, which is developed from its recursive version from [5]. For its correctness take a look into its original.

Algorithm 4 Maximal suffix

```
i ← 0, j ← 1, k ← 1, p ← 1
while j + k ≤ n do
    a' ← x[i + k], a ← x[j + k]
    if a < a' then j ← j + k, k ← 1, p ← j − i
    end if
    if a' = a then
        if k = p then j ← j + p, k ← 1
        else k ← k + 1
        end if
    end if
    if a > a' then i ← j, j ← i + 1, k ← 1, p ← 1
    end if
    return i, p
end while
```

Integer i is the starting position of current $max(x)$ being tested, j is the position of last occurence of $rest(x)$ and p is period of $max(x)$.

Theorem 1.2.3. *The algorithm 4 uses on its run less than $2n$ character comparisons, where n is the lenght of the input string.*

Proof. We will show that expression $i+j+k$ is increased after each comparison between letters a' and a at least by one. Since $i \leq n$ and $j + k \leq n + 1$, we have

$$2 \leq i + j + k \leq 2n + 1$$

which means that the number of comparisons is bounded by $2n$.

Lets prove now that $i + j + k$ is increased after each comparison. We distinquish three cases:

1. If $a < a'$, then $i + j + k$ is replaced by $i + j + k + 1$.
2. If $a = a'$, then $i + j + k$ is replaced also by $i + j + k + 1$.
3. If $a > a'$, then $i+j+k$ is replaced by $2j+2$, but since we have $i+k \leq j$, after adding j on both sides, we obtain $i + j + k \leq 2j$, so $i + j + k$ is increased in this case at least by two.

Hence the proof is complete. □

But how to compute the period of x. Sure we can use the Knuth-Morris-Pratt algorithm, but that would spoil our so far constant space algorithm. We will adjust our string matching algorithm, which will now use $per(x)$ only when the string x is periodic, if not we will present a simpler algorithm.

In the case, that $per(x) \leq \frac{x}{2}$(is *is periodic*), we use the algorithm, that produces $per(x)$ iff x is periodic, else lowerbound to $per(x)$. It is based on following theorem.

Theorem 1.2.4. *Let (u, v) be the critical factorization of the string x such that $|u| < per(x)$. Let $v = y^e z$ with $e \geq 1$ and $|y| = per(x)$, if $|u| < \frac{|x|}{2}$ and u is suffix of y, then $per(x) = per(v)$, otherwise $per(x) > max(|u|, |v|)$.*

Proof. If the condition $|u| < \frac{|x|}{2}$ is true, then the string x is a factor of y^{e+2}. Hence, $|y| = per(v)$ is a period of x, and since the period of x cannot be less than the period of v, we get $per(x) = per(v)$.

If $|u| \geq \frac{|x|}{2}$, then trivially holds $max(|u|, |v|) = |u|$ and $per(x) > max(|u|, |v|)$.

If $|u| < \frac{|x|}{2}$ and u is not a suffix of y. We show that there is no nonempty string w such that wu is a prefix of x. We will show this by contradiction. Assume that wu is a prefix of x. If w is nonempty, its lenght is a local period at (u, v), and then $|w| \geq per(x) \geq per(v)$. We cannot have $|w| = p(v)$ because u is not a suffix of y. We cannot either have $|w| > per(v)$ because this would lead to a local period at (u, v) strickly less than $per(v)$, a contradiction. This proves the assertion and also shows that the local period at (u, v) is strictly larger that $|v|$. Since $max(|u|, |v|) = v$, we get the conclusion: $per(x) > max(|u|, |v|)$. □

We showed that, when our algorithm has $|u| \leq \frac{|x|}{2}$, then we can simply we can simply use precomputed $per(v)$ of 4 as our $per(x)$. But what about larger periods? In the case of a larger period we will use an easier algorithm, that instead of shifting pattern by $per(x)$ to the right, shifts only by q with $q \leq per(x)$. When q is chosen well, the time complexity remains linear. How to choose q is explained below.

Theorem 1.2.5. *Let x and t be strings and let q be an integer such that $0 < q \leq per(x)$. Then, the function POSITION-BIS, which uses both the integer q and a critical position l such that $l < per(x)$, computes the set of position of x in t. If $q > max(l, |x| - l)$ the number of comparisons executed is at most $2|t|$.*

Note that function *POSITION-BIS* is not mentioned here. It is described in the original article [5]. The only difference between this algorithm and algorithm 3, is the change of shifts mentioned above.

Now we can finally describe the whole Two-Way algorithm. It is composed of the two procedures. First is 3 in the case when $l < \frac{|x|}{2}$ and $x[1..l]$ is a suffix of $x[l + 1..l + p]$. Second is *POSITION-BIS* with $q = max(l, |x| - l) + 1$. Proof that this combined algorithm works in linear time and constant space is consequence of correctness the previous two algorithms.

1.2.4 The Galil-Seiferas algorithm

The base idea of this algorithm was firstly presented in [6], where authors deduced almost linear time hybrid algorithm from KMP. It was shown that this algorithm for fixed k needs $\mathcal{O}(log(m))$ memory space and runs in $\mathcal{O}(|x|^{\epsilon}(|x||y|))$ for some small ϵ. They also showed that if we choose our k based on deliberately designed function, memory consumption of this algorithm can be torn down to only constant space, but at the cost of slowing down time complexity.

In [7] authors improve their previous algorithm with using string property called *prefix period* and develop a constant space algorithm, which makes at most $5n$ character comparisons with preprocessing in $\mathcal{O}(m)$.

Note that in this section we consider k to be some small fixed value, Galil and Seiferas suggested $k = 4$, because the time complexity of is the algorithm is proportional to k. In further text we will leave k to be unspecified to show its role in the analysis.

Lot of string-matching algorithms uses following scheme 5. We will use it as well. Variable p is position in text tested for a match, q maintains the lenght of currently matched part of pattern ($x[0, q] = y[p, p + q]$). Then next $p' > p$ and q' are computed, the shift is advanced. When $q = |x|$, valid shift at position p is reported.

For example Naive algorithm computes $p' = p + 1$, $q' = 0$. To achieve better performance, we need to use the knowledge that $x[0..q] = y[p..p + q]$. KMP uses this knowledge in a way, that he computes

$$shift_x(q) = min(shift > 0 | x[shift..q] = x[0..q - shift])$$

, and then applies $p' = p + shift_x(q)$ and $q' = q - shift_x(q)$. The *shift* function basically computes the shortest period of $x[0..q]$.

Algorithm 5 General scheme

$(p, q) \leftarrow (0, 0)$
ploop:
while $y[p + q + 1] = x[q + 1]$ **do** $q \leftarrow q + 1$
end while
$(p, q) \leftarrow (p', q')$
goto ploop

From this scheme we can easily deduce a hybrid algorithm, on which we will build our theory.

Algorithm 6 Hybrid

$(p, q) \leftarrow (0, 0)$
ploop:
while $y[p + q + 1] = x[q + 1]$ **do** $q \leftarrow q + 1$
end while
if $q = 0$ **then** $(p, q) \leftarrow (p + 1, 0)$
else if $q > 0 \wedge shift(q) \leq q/k$ **then**
    $(p, q) \leftarrow (p + shift(q), q - shift(q))$
else if $q > 0 \wedge shift(q) > q/k$ **then**
    $(p, q) \leftarrow (p + \lceil \frac{q}{k} \rceil, 0)$
end if
goto ploop

Its correctness is obvious. In each step p is incremented at least by 1, hence, it terminates, and no valid shift is skipped, because each advance performed is smaller than $shift(q)$ and it is (according to KMP) a smallest safe advance we can perform.

Running time of 6 is $\mathcal{O}(k|y| + |x|$, because the number $(k + 1)p + q$ is increased every step. So the time complexity is stricktly proporcional to k.

We also should mention the corollary of *periodicity lemma* which will be later used in proofs.

Theorem 1.2.6. *Distinct prefix periods of the same string differ in length by at least a factor of $k - 1$. In fact, if w has a prefix period of length p_1 and a basic prefix of length $p_2 \geq p_1$ with $reach_w(p_2) = k'p_2$ for any $k' \geq 2$, then $p_2 > (k - 1)p_1$.*

Next two lemmas give together the notion of *prefix period* and occurence of the first case $shift(q) \leq q/k$.

Lemma 1.2.3. *If $shift(q) \leq \frac{q}{k}$, then $x[0..shift(q)]$ is a prefix period of x.*

Proof. $Shift(q)$ is a shortest period of $x[0..q]$ and because $q \geq k * shift(q)$, $x[0..shift(q)]$ appers at least k-times in $x[0..q]$ it is prefix period. If it would

not be basic, then there would be some shorter period, which is a contradiction. □

Lemma 1.2.4. *If $x[0..shift]$ is a prefix period of x, then*

$$shift = shift(q) \leq q/k \leftrightarrow k * shift \leq q \leq reach(shift)$$

.

Proof. The way $\rightarrow$ is trivial, all you need is to realize the definition of prefix period and reach function. Backward implication $\leftarrow$ is a little bit harder. Lets prove it by contradiction. Consider two cases. If $shift(q) > shift$, then this claims that $shift(q)$ is the shortest period, hence, this could not happen, because shift is also period and is shorter. If $shift(q) < shift$, this contradicts the assumption that $x[0..shift]$ is basic. □

Now we will present *decomposition theorem*, which will give us an efficient way how to use scheme 6 for constant space string-matching algorithm.

Theorem 1.2.7 (Decomposition theorem)**.** *Each pattern x has a parse $x = uv$ such that v has at most one prefix period and $|u| = \mathcal{O}(shift_v(|v|))$.*

In order to be able to prove the theorem, first, we need to prove following lemma. This lemma will give us constructive proof which we will use in the preprocessing.

Lemma 1.2.5. *For each basic string w, there is a parse $w = w_1w_2$ such that, no matter what w' is, $w_2w^{k-1}w'$ has no prefix period shorter that $|w|$.*

Proof. To prove this lemma we will first show that w' does not matter. By the corollary 1.2.6 any prefix period shorter than $|w|$ would have to be shorter than $\frac{|w|}{k}$. When we realize that $|w|/(k-1) \leq (k-1)|w|/k \leq |w_2w^{k-1}|/k$, we see that w' is completely irrelevant because this shorter prefix is also a prefix period of just w_2w^{k-1} for any choice of parse $w = w_1w_2$.

Now let's assume the string w^∞ and seek this parse in it. We will repeatedly delete prefixes z such that remainder of w^∞ has a prefix z^k and $|z| < |w|$. Note that if this terminates, we have found our parse.

We claim that this sequence of deletions terminates. To show that we need to prove that every deleted z' is longer or of the same length as z deleted before and no same $|z|$ can loop forever. When z is deleted, then z^{k-1} remains a prefix of the rest. By the periodicity lemma, z'^k would have to be a prefix of zz' and also of z^k. This is a contradiction, hence $|z'| \geq |z|$ or $z' = z$. Since the w is basic, the periodicity lemma implies that no same z will be deleted forever. Hence, the termination of the sequence is guaranteed. □

Further, we will use a stronger version of this lemma that claims *Not even one full $|w|$ gets deleted.* Proof is omitted.

Proof of Decomposition theorem. We will prove this theorem by the construction of such decomposition. Consider this algorithm which produces $v = x[s..|x|]$, also called perfect factorization.

Algorithm 7 Perfect factorization

```
s ← 0
while x[s..|x|] has more than one prefix period do
    p2 ← a second shortest period
    Find s' such x[s'..|x|] has no prefix period shorter than p2
    s ← s'
end while
```

By the lemma 1.2.5 the algorithm is correct. Its property that $|u| = \mathcal{O}(shift_v(|v|))$ remains to be proved. We will just refer our reader to [7] where the proof by induction is completed.

□

If we would somehow find an algorithm that computes such decomposition in linear time, we get a linear time algorithm with constant space memory. To show that consider our scheme 6. We use this scheme to find all occurrences of v in text. If v has no prefix period then first case $shift(q) < q/k$ never happens and $(p, q) \leftarrow (p + \lceil \frac{q}{k} \rceil, 0)$ all the time. If v has a prefix period (note that it could not have more that one due to Decomposition theorem), then the first case occurs only when $kp_1 \leq q \leq reach(p_1)$ according to lemmas 1.2.3 and 1.2.4.

Time complexity of searching for every occurence of v is $\mathcal{O}(|v| + |y|)$ as shown earlier. Then the algorithm naively check whether there is an occurence of u before every v. Since v can occur at most $|y|/shift_v(|v|)$ in a text y and $|u| = \mathcal{O}(shift_v(|v|))$, hence, the total number of charecter comparisons while searching for u will be $\mathcal{O}(|u|)|y|/shift_v(|v|) = \mathcal{O}(|y|)$.

Complete complexity of these two checks is $\mathcal{O}(|x| + |y|)$.

Now last thing left is to show an algorithm for finding *perfect factorization* in a linear time. We will efficiently implement the algorithm 7. We have found a fast way how to compute the desired s'.

If we look at 1.2.5 and its stronger version, we can use:

`while` $x[s'..|x|]$ has a prefix shorter than p_2, delete shortest one.

Remains to find such a subroutine that will find p_1 and p_2 in linear time proportional to their lenghts. Such a subroutine is a matching the pattern against itself with $(p, q) \leftarrow (1, 0)$. Result of this algorithm will be that at each position i we have $shift(i) = per(x[0..i])$. Hence, we need to look for the first i such that $shift(i) \leq i/k$. If such a position i does not exist, the algorithm runs through whole w and therefor will have a complexity $\mathcal{O}(|w|)$. If p_1 is found its complexity is $\mathcal{O}(p_1)$.

To find a prefix period shorter than p_2 we will search for $i < p_2$.

The same procedure is used for finding p_2, the second longest prefix period. Suppose we have found an p_1, then we determine $reach_w(p_1)$ naively in time $\mathcal{O}(reach_w(p_1))$. When we have the value we just look for an occurence of $i > reach_w(p_1)$ where $shift(i) \leq i/k$ in time $\mathcal{O}(p_2)$ if p_2 exists or $\mathcal{O}(|w|)$.

Consider the time of using such subroutines for finding *perfect factorization*. Test for one failed entry $x[s..|x|]$ will take $\mathcal{O}(|v|)$. The time spend by computation of s' by deleting shortest prefix periods will be $\mathcal{O}(s'-s)+\mathcal{O}(p_2) = \mathcal{O}(p_2)$ and because each successive p_2 will be at least $k-2 \geq 2$ times the previous (according to periodicity lemma and $s' < s + p_2$) we get the total decomposition time

$$\mathcal{O}(|v|) + \mathcal{O}(|v|(1 + 1/2 + 1/4 + ...)) = \mathcal{O}(|v|) = \mathcal{O}(|x|)$$

, hence, we showed that *Galil-Seiferas* is indeed constant space algorithm with running time $\mathcal{O}(|x| + |y|)$.

1.2.5 The Sequential-Sampling algorithm

In [8] was presented another constant space and linear time, which is based upon the idea of *sampling*. In this article two versions of the algorithms were proposed, where the first one makes at $2n$ character comparisons and the second one makes $(1 + \epsilon)n + \mathcal{O}(n/m)$ comparisons. Preprocessing to our algorithm makes $(1 + \epsilon)m + \mathcal{O}(m/\epsilon)$ comparisons and can be implemented in constant space.

In this thesis we will attend only to the $2n$ version with no regard to preprocessing. We refer the curious reader to [8] for the whole understanding of the algorithm if needed.

Let start by introducing the definition of the *sample* of the pattern.

Definition 1.2.3. *If nonperiodic pattern has a periodic prefix, denote by π the longest one. Let $q-1$ be the lenght of π and per the shortest period of π. Let the sample of the prefix $y[1..q]$ be the set $S = \{p, q\}$. Secondly define the predicate $MatchSample(i, S) = (y[i + p] = x[p] \wedge y[i + q] = x[q]$.*

From this definition results the key property that claims that if $MatchSample(i, S)$ then no occurence of the pattern starts at any position in $y[i + 1..i + p]$. In another words, if $MatchSample(i, S)$ next safe shift can is at least p.

Example. *If $x = aaaab$ then $S = \{4, 5\}$ and if $MatchSample(i, S)$ the next safe shift is at least 4.*

Our simple algorithm differs in three cases.

1. All the patterns prefixes are nonperiodic.

2. The pattern is nonperiodic and has periodic prefix.

3. The pattern is periodic.

In the first case we can use the algorithm 8.

Algorithm 8 SimpleTextSearching

```
i ← 0
while i ≤ n − m do
    j ← max{k : y[i + 1..i + k] = x[1..k]}
    if j = m then Report match at i
    end if
    i ← i + ⌈(j+1)/2⌉
end while
```

Notice the shift $i \leftarrow i + \lceil \frac{j+1}{2} \rceil$. Why we can do so? Recall KMP shift, where $shift[i] = per(x[1..i+1])$. Because we know that every prefix of pattern is nonperiodic, the *per* of every prefix is larger than $|prefix|/2$, so this shift is correct. Because $j + 1 \leq 2\lceil \frac{j+1}{2} \rceil$, where $j + 1$ is number of comparisons in every stage, we get that 8 makes at most $2n$ comparisons.

In the second case the pattern has a sample $S = \{p, q\}$. For every i we try to match the sample S. If we succeed we try to match the whole occurrence of the pattern. In case of some mismatch we decide the next safe shift upon the fact, whether $j < q - 1$ or not. If it is, we use previous property and shift by p, in case $j \geq q - 1$ it means that the matched part is larger than the longest periodic prefix, therefore the shift by $\lceil \frac{j+1}{2} \rceil$ can be used. The time complexity of the second case is also $2n$, because if $j < q - 1$ we can amortize negative match by 2 comparisons on 1 shift, if not it is the same as in the first case.

Algorithm 9 SequentialSampling

```
                                  ▷ Required precomputed S = {p, q}
i ← 0
while i ≤ n − m do
    if not MatchSample(i, S) then i ← i + 1
    else
        j ← max{k : y[i + 1..i + k] = x[1..k]}
        if j = m then Report match at i
        end if
        if j < q − 1 then i ← i + p
        else
            i ← i + ⌈(j+1)/2⌉
        end if
    end if
end while
```

In the third case, when the pattern is periodic then $x = v^k v'$ for $k \geq 2$ where v is a prefix of length *per* and v' is a prefix of v. We will the observation

that the prefix vv^- where v^- is v without the last character is nonperiodic. Now we can search for vv^- in the text with one of the first two cases applied. If the part vv^- is found we start to match the next characters of the pattern. In case of a mismatch the number of comparisons made is $2 * per + k$ for some $0 \leq k \leq m - 2 * per + 1$, but the shift is $s = per + k$, because the matched prefix has the period per. Again even here the time complexity can be amortized to $2n$, because $2 * s \geq 2 * per + k$.

1.2.6 The CGR algorithm

The CGR algorithm presented in [9] is the first algorithm that tries to run in sublinear time. In average it runs in $o(n)$ (in worst case $\mathcal{O}(n)$ with only constant space complexity. The formal proof of these complexities is omitted, but can be found in the referenced paper.

The algorithm is based upon the key concept of *subword repetition*.

Definition 1.2.4. *Let w be a subword of x, then iff w has two disjoint occurrences in x, we say w is a repeated subword. Denote by $RepSize(x)$ the length of longest repeated subword of x.*

There also holds the following lemma which gives us information about the lower bound of a length of $RepSize(x)$.

Lemma 1.2.6. *Assume that the alphabet is constant. Then for any pattern x of size m $RepSize(x) = \Omega(m)$.*

For our algorithm we need to have precomputed this 4-tuple $REPET(x) = (w, r, p, q)$, where w is a longest repeated subword, r is the $RepSize(x)$, p, q are position in x of occurences of w. Let *window* be a part of y of length $r/2$ (r is a $RepSize(x)$) $x[i - r/2..i]$ for $i > r/2$. For a position i in text denote $CheckingArea(i)$ the union of two intervals

$$y[i + p..i + p + r/2] \cup y[i + q..i + q + r/2]$$

. A mismatch in $CheckingArea(i)$ is are positions p', q', where $p' - p = q' - q$, such that $y[p'] \neq y[q']$. We define $LeftMost_Mismatch(i)$ as first such a position in $CheckingArea(i)$ from the left, if no mismatch is found, return *nil*.

Now we can formally describe our algorithm.

Procedure *NaiveCheck* just naivelly check whether pattern x mathes text at position i_0.

Correctness of 10 is based upon following lemma.

Lemma 1.2.7. *If $LeftmostMismatch(i) \neq nil$ then no occurrence of the pattern starts in the window $y[i - r/2, i]$.*

Algorithm 10 CGR

```
r ← RepSize(x)
i ← r/2 + 1
while i ≤ n − m do
    if LeftMost_M ismatch(i) = nil then
        for i_0 ∈ [i − r/2..i] do NaiveCheck(i_0)
        end for
    end if
end while
```

Proof. Lets look at two positions p', q' in text, where mismatch was found. Now look at positions p'', q'' in the pattern, which should match p', q'. Because positions in the window are at most $r/2$ positions to the left of i, p'' and q'' should be the same. They are part of the repeating factor. This leads to a contradiction. □

Without correct proof we claim that the algorithm uses constant space and for nonperiodic pattern runs in $\mathcal{O}(n/RepSize(x)) = \mathcal{O}(n/\log(m))$ on the average, for periodic pattern runs even faster, in average in $\mathcal{O}(n/m)$.

1.2.7 The Quite-Naive algorithm

The Quite-Naive algorithm is an improvement of the Not-So-Naive algorithm. Its worse time complexity is $\mathcal{O}(nm)$ and it requires pattern preprocessing in $\mathcal{O}(m)$. This is a new algorithm proposed in [1].

In the preprocessing phase the algorithm computes two walues δ, γ, where

$$\delta = min(1 \leq j < m : x[m-1-j] = x[m-1]) \cup \{m\}$$

$$\gamma = min(1 \leq j < m : x[m-1-j] \neq x[m-1]) \cup \{m\}$$

. It is united with $\{m\}$, because you are not guaranteed, that there exist in either case such an index j. It is obvious that either δ or γ is one, the other is strictly bigger than one, hence the preprocessing phase takes only a single run of at most $m+1$ characters comparisons and requires constant space.

The matching phase is performed in the following way. The algorithm matches the text from right to left and every shift is by δ or γ. Assume that we are trying to match the text at position s. Now we distinguish two cases

Case 1: If the first comparison fails, namely $x[m-1] \neq y[s+m-1]$, then we advance the shift by γ position to the left.

Case 2: If the first comparison succeeded, we proceed further in the matching at position s, and then it doesn't matter whether we successfully matched pattern, we advance the shift by δ position to the left.

In practical cases The Quite-Naive performs slightly better than the Not-So-Naive.

1.2.8 The Tailed-Substring algorithm

The Tailed-Substring algorithm is another worst-case $\mathcal{O}(nm)$ algorithm, but in practice performs well, especially on longer patterns. This is also a new algorithm proposed in [1].

Definition 1.2.5. ***Tailed substring*** *is a substring sub of pattern, where holds that last character of sub has no other occurence in sub.* ***Maximal-tailed substring*** *is a tailed substring, which is also maximal in pattern.*

Its run consists of two phases. In the first phase the algorithm searches for x and simultaneously computes values δ and y such that $sub = x[k - \delta + 1..k]$ and sub is a maximal-tailed substring of pattern. Value δ is the length of sub and k is the last index of sub in pattern. When these values are correctly computed we can quicken our algorithm and proceed to phase 2, where are these values used for such purpose.

Algorithm 11 The Tailed-Substring

```
1:                                                          ▷ First phase
2: s ← 0
3: δ ← 1
4: i ← k ← m − 1
5: while s ≤ n − m ∧ i − δ ≥ 0 do
6:     if x[i] ≠ y[s + i] then s ← s + 1
7:     else
8:         j ← 0
9:         while j < m ∧ x[j] = y[s + j] do j ← j + 1
10:        end while
11:        if j = m then Report(s)
12:        end if
13:        h ← i − 1
14:        while 0 ≤ h ∧ x[h] ≠ x[i] do h ← h − 1
15:        end while
16:        if δ < i − h then
17:            δ ← i − h
18:            k ← i
19:        end if
20:        s ← s + i − h
21:        i ← i − 1
22:    end if
23: end while
24:                                                         ▷ Second phase
25: while s ≤ n − m do
26:    if x[k] ≠ y[s + k] then s ← s + 1
27:    else
28:        j ← 0
29:        while j < m ∧ x[j] = y[s + j] do j ← j + 1
30:        end while
31:        if j = m then Report(s)
32:        end if
33:        s ← s + δ
34:    end if
35: end while
```

Lets now describe the pseudocode 11. We start by with assigning $\delta \leftarrow 1$ and indices i, k are set to the last character of pattern, k represents currently maximal-tailed substring and i potencial candidate on k.

The first phase ends when $\delta \geq i$, because then we have already computed correct k and δ. First we find a shift s such that $x[i] = y[s + i]$, then we try to match this shift from left to right. After this we find an index h in

pattern such that $x[i] = x[h]$. If such an index is found, we advance the shift so position h is aligned with $y[s+i]$. Otherwise, we advance the shift by $i+1$ (Note that this is principally the bad character shift performed in BM). Both these cases are resolved at line 20, in the second case $h = -1$. Then on the line 16-18 we update if possible δ. It is obvious that after phase 1, we have the correct maximal tailed-substring of pattern.

In the second phase, we look for an occurrence of $x[k]$ in the text. If we find such a position, we naive check whether this is a valid shift and advance by δ positions.

Chapter 2

Implementation

In this chapter we will discuss the implementation and integration of algorithms described in the theoretical part. We will also mention the project of the Algorithms Library Toolkit itself in order to introduce the reader to our environment.

2.1 Algorithm Library Toolkit

Algorithms Library Toolkit(ALT) from [10] is a set of data structures and algorithms from the area of stringology. To name a few data structures ALT offers, various kinds of automata, grammars, trees, etc. All the data structures are written to be acting closest possible to its theoretical background.

The library is written in C++ and consists of a series of modules, command-line interpreter *aql2* and prototype of a GUI version *agui2*.

ALT was created mainly for educational purposes at the Faculty of Information Technology, Czech Technical University, and is there still under active development. We also could not forget its authors Jan Trávníček, Tomáš Pecka and others.

2.2 Algorithms

Our purpose was to implement every algorithm from [1]. Our implementation is based solely on descriptions in this article and the referenced bibliography. The implementation holds the structure of *pseudocodes* in these resources.

Note that the TwoWay algorithm was not implemented and will not be included in the rest of this thesis. It was because after sticking to its pseudocode and making minor adjustments, it kept failing in the integration tests.

All the implemented algorithms can be found in `automata-library/alib2algo/src/stringology/exact`. For their implementation were not used any outer libraries with the exception of `stl`. In some of these implementations were

needed other procedures as for example finding a longest prefix period etc. These additional procedures related to string properties can be found in `automata-library/alib2algo/src/string/properties`.

Also, note that the preprocessing for the CGR and the Sequential-Sampling were made naively, but because in the testing chapter we measure only the running phase, this has no impact on the final results.

2.3 Usage

You can either download ALT from its source on `https://alt.pecka.me/download/` or use enclosed CD, where is the library already precompiled with its code sources.

All you need to do, is to run the command line interpreter *aql2*. In case of using enclosed CD, navigate to `automata-library/debug/aql2`, in case of downloading ALT from source, it comes with prepared executable, that you can run from a terminal. Now you are in the interpreter and you can execute the algorithm you want to run by command `print algorithm input`. For example if you want to run Not-So-Naive algorithm you would type down `print stringology::exact::NotSoNaive abaaba aba`. It is also possible to redirect the output of a command to file, use standard UNIX `>` redirect. For quitting the interpreter use command `quit`.

Chapter 3

Testing

In this chapter we will attend to the important subject of testing our implementation described in previous chapter. This chapter is divided into two sections, *the correctness of implementation* and *speed benchmarkes*. Again note that TwoWay algorithm is ommited.

3.1 Testing the correctness of implementation

After implementing our algorithms, it is necessary to test its correctness. We did it by automatically generated test data.

ALT has already nicely written integration tests for all the data structures and algorithms contained, with *exact matching* included. For every algorithm we added line

```
std::make_tuple ( "Name of the algorithm",
 "stringology::exact::Name $subject $pattern", true ).
```

This registers the algorithm into testing on the automatically generated dataset. All these integration tests are run when the library is built.

3.2 Speed benchmarks

For testing the efficiency of our implementation we chose to conduct tests through the command line interpreter *aql2*. Average running time is measured. Also we will make comparisons with the BoyerMooreHorspool algorithm in order to get our data related to in practice used algorithms. BoyerMooreHorspool is in practice one of the most commonly used algorithms.

ATL has already prepared utilities for testing purposes. Namely it is the

```
string::generate::RandomStringFactory class
string::generate::RandomSubstringFactory class.
```

The first one is used for generating the text and the second one is used for generating the pattern.

Then we need to measure the running times of our algorithm. For this purpose we used the `measure` utility from ALT. For more details look at the bash script from the following text.

We wrote a simple bash script `measure.sh`(in *automata-library/debug/aql2*), which calls *aql2* with necessary arguments and inside the command line call all the measured algorithms with a randomly generated text and pattern. In the script there are three things parametrized. The length of the generated text, the length of the generated pattern, and the number of runs that will be used to compute average running time. The number of runs was decided to be the fixed number 10.

With this script we ran following test cases.

1. Short pattern
2. Long pattern

In the first case we chose $textsize = 10000$, $patternsize = 7$. The results were following. Note that all units are in ms.

SequentialSampling	QuiteNaive	TailedSubstring	CGR	Dogaru	GalilSeiferas	NotSoNaive	Horspool
3518	1426	1488	2291	2398	6843	3439	2183

Table 3.1: Rusults on a short pattern

In the second case we chose $textsize = 10000$, $patternsize = 30$. The results were following.

SequentialSampling	QuiteNaive	TailedSubstring	CGR	Dogaru	GalilSeiferas	NotSoNaive	Horspool
3657	704	1091	3646	2400	6997	2938	1869

Table 3.2: Rusults on a long pattern

From the above result we see that in both cases two best-performing algorithms are the QuiteNaive and the TailedSubstring followed by the Dogaru and the NotSoNaive. Although it seems that with shorter patterns QuiteNaive and TailedSubstring performed approximately the same, with longer pattern QuiteNaive is a little bit faster.

Note also that the time decrease in searching for longer patterns is caused by a lower density of pattern occurrences in the text.

There is also nicely demonstrated one thought from the [1], that "sometimes economical solutions are more efficient than unrestricted ones, it's economy, stupid! ". In both test cases Horspool performs worse than the two best algorithms. Although as mentioned in [1] Horspool, which is one of the

most efficient versions of BoyerMoore algorithm, performs better on bigger alphabets and not so short, we see that in some cases these in comparison to Horspool trivial algorithms with restricted memory can outperform a lot more sophisticated algorithms.

Conclusion

The goal of this thesis was to understand, implement and test algorithms from [1].

In this bachelor thesis you are presented with 8 constant space algorithms for the *string matching* problem. All of them except the Two-Way were implemented into the ALT library, tested and results of their performance were compared. The integration into the ALT ecosystem was successful.

In common literature on this theme, although there are enough books and articles about the algorithms described, we lack a complex overviewing text, which can give us all the information needed in one place. This thesis can serve as such a text. We described for the reader every algorithm presented in [1] with correctly proven properties which these algorithms are based on. Their implementation is in the enclosed CD. The algorithms were purposely implemented tightly according to pseudocodes mentioned here or in original articles to ease the understanding and to preserved their nature described here.

In chapter 3 we made performance testing on randomly generated strings and shown as said in [1] that in some cases more economical solutions can be even more efficient than those unrestricted. It was revealed that under the test circumstances two in practice the fastest algorithms were the Quite-Naive and the Tailed-Substring.

Bibliography

1. CANTONE, Domenico. " IT'S ECONOMY, STUPID! " : SEARCHING FOR A SUBSTRING WITH CONSTANT EXTRA SPACE COMPLEXITY. In: 2005.
2. M., Crochemore; W., Rytter. *Jewels of Stringology: Text Algorithms*. World Scientific, 2002. ISBN 9789810248970. Available also from: `https://books.google.cz/books?id=9NdohJXtIyYC`.
3. CHARRAS, Christian; LECROQ, Thierry. *EXACT STRING MATCHING ALGORITHMS* [online] [visited on 2020-04-20]. Available from: `http://www-igm.univ-mlv.fr/~lecroq/string/`.
4. DOGARU, O. C. On the All Occurrences of a Word in a Text. *Proc. of The Prague Stringology Conference*. 1998.
5. CROCHEMORE, Maxime; PERRIN, Dominique. Two-way string-matching. *Journal of the ACM (JACM)*. 1991, vol. 38, pp. 650–674. Available from DOI: `10.1145/116825.116845`.
6. GALIL, Zvi; SEIFERAS, Joel. Saving Space in Fast String-Matching. In: 1977, vol. 9, pp. 179 –188. Available from DOI: `10.1109/SFCS.1977.27`.
7. GALIL, Zvi; SEIFERAS, Joel. Time-space-optimal string matching. *Journal of Computer and System Sciences*. 1981, vol. 26, pp. 280–294. Available from DOI: `10.1016/0022-0000(83)90002-8`.
8. PLANDOWSKI, Wojciech; RYTTER, Wojciech; INFORMATKI, Instytut; WARSZAWSKI, Uniwersytet. Constant-Space String Matching With Smaller Number of Comparisons: Sequential Sampling. In: 1999. Available from DOI: `10.1007/3-540-60044-2_36`.
9. CROCHEMORE, Maxime; GASIENIEC, Leszek; RYTTER, Wojciech. Constant-space string-matching in sublinear average time. *Carpentieri, B.; De Santis, A.; Vaccaro, U.; Storer, J. A.: Compression and Complexity of SEQUENCES 1997, -, 230-239 (1998)*. 1999, vol. 218. Available from DOI: `10.1016/S0304-3975(98)00259-X`.

10. JAN, Trávníček; TOMÁŠ, Pecka; ET. *Algorithms Library Toolkit* [online] [visited on 2020-04-20]. Available from: `https:/gitlab.fit.cvut.cz/algorithms-library-toolkit`.

Appendix A

Acronyms

GUI Graphical user interface

XML Extensible markup language

iff If and only if

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `automata-library` ..... the directory with executables and source code
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis` ..... the directory of LaTeX source codes of the thesis