Algorithms and Data Structures

*Summer Term 2022*

*Bonus Problem*

*Chaitanya Mangra : 22105964 - Amelie Kammerer : 00816131*

*Alberto Gómez Conejero : 22109532 - Philipp Schmalhofer : 00736579*

**Introduction and Problem description**

In today's modern world, everything, from photos and emails to personal information, is digitalized. This phenomenon results in a humongous amount of data to be dealt with. This data is often stored in "string" form (sequence of characters). Although most of the data stored makes no sense to a human reader and has nothing to do with our natural language.

However, it is in some cases very useful to be able to search for a specific string pattern among this immense amount of data.

The following example shows a very simple human figure which has been pixeled in string form:

" BBBB

BBBBBB

ZBBBBBBZ

ZZZZ

ZZZZ

WWBBBBBB

BBBWWBBBBB

BBBBBWWBBB

BBBBBBBWWB

ZBBZZBBZ

BBB BBB

BBB BBB

BBB BBB

ZZZ ZZZ "

In this case the patterns "WW" represent 2 white pixels. If we wanted to change them to orange pixels "OO", we would have to either do it manually or run it through a pattern-matching algorithm which searches for the string pattern "WW" and replaces them with "OO". By replacing the pixels color code, we are improving/changing our figure and the way humans will see it.

This application example is a simple one, but in the real world, we sometimes even must deal with thousands of terabytes of data, when finding the right pattern-matching algorithm, that is able to handle this amount of data more efficiently, is crucial. This brings us to the three pattern-matching algorithms that we are going to take into consideration:

- The Brute-Force Algorithm

- The Boyer-Moore Algorithm

- The Knuth-Morris-Pratt Algorithm

**Presentation of the three algorithms and how they work**

**Brute-Force**

First of all, the Brute-Force pattern matching algorithm is one of the simplest algorithms for string matching. In order to find a string in a text document, the algorithm compares the first character of the pattern with the first character of the text. If there is a match, we try to match the second character and if there is a match again we try to match the third character and so on, up to the length of the pattern. When the code finds a mismatch it the pattern it slides one character to the right and starts again with the next character in the text.

Like this the algorithm browses throw the whole text document and then, if a match is found returns the starting location of the match. If there is no match in the text document the algorithm returns 0.

**Boyer-Moore**

Secondly, the Boyer-Moore-Algorithm is a little bit cleverer, since it skips some characters, where the algorithm can already be sure, that there will be no match. The Boyer-Moore-Algorithm compares the pattern not like the Brute-Force from left to right, but from right to left. Like this the Algorithm first compares the last character of the pattern to the character in the text that stands on the position of the length of the pattern. If the last character of the pattern matches the text, the algorithm now examines the character left of the last character. If the complete pattern matches the text, the starting location of the match is returned, but in case there is a mismatch the algorithm skips the alignment until either a mismatch becomes a match, or the pattern moves past the mismatched character.

On one hand that means if the mismatched character in the text occurs in the up to now not checked left hand side of the pattern, we shift the pattern to right until the mismatch becomes the match. On the other hand, if the mismatched character does not occur in the pattern, we shift the entire pattern to the right beyond the location of the mismatch.

In this project we only used a simplified version of the algorithm with the named rules called bad character heuristics, while a more complicated version also uses the already matched characters to skip even more parts of the text, the good suffix heuristics.

**Knuth-Morris-Pratt (KMP)**

Lastly, the Knuth-Morris-Pratt pattern matching algorithm, short KMP is an algorithm, which compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table, called “Prefix Table” to skip character comparisons.

To understand the algorithm, you need to know the terms proper prefix and suffix. A proper prefix is a subset of the pattern using the first index of the pattern and as many characters more as you want, as long as the last element is left out. A proper suffix would be a subset of the pattern with characters taken only from the right end of the pattern, also as many as wanted, except of the first character of the pattern.

In a pattern, there can be a proper prefix same as the suffix. For example, if you got a pattern “abcdfgabc”, “abc” would be in this category. The KMP algorithm observes this exact phenomenon, trying to find the LPS (longest proper prefix that is also a suffix) with a LPS table, which maps every character of the pattern to a value. The value mapped to each character, for example LPS[i] represents the length of the longest proper prefix that is also a suffix in the first i characters where 0 < i < length(pattern) – 1.

One example of a LSP table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | A | B | F | A | B | C |
| 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 |

While observing the pattern we will add ‘0’ to the characters that occurred for the first time, while if we encounter a character for another time, as on index 5 => ‘A’ and 6 => ‘B’, we write ‘1’ and ‘2’ because we have already got those characters at index 1 and 2. If we encounter a repeated character, that is also a part of a pattern (at index 8 in this example), we solve it as followed:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | A | B | C | A | D | A | A | B | E |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 0 |

The KMP algorithm now uses this table to skip partial matches and, as a consequence avoids unnecessary old comparisons on finding partial matches.

**Description of text documents that you use for testing your algorithm implementations**

*Shakespeare*

We decided to choose this text, not only because of its length, but also because of the old and existing words in it. In this case, words and sentences can be searched on a coherent flowing text. We selected some sentences and words with varying length to be searched, which knew that they would exist along the text. We do have selected patterns from all over the text in order to match it too with help from the algorithms.

*Random-letters-numbers*

In this case, we created with help from a computer, a no-human-involved text containing just random numbers and letters. It has been thought so that the outcomes could not be manipulated in any way, since patterns were also created randomly with also different length. However, we do not know if those patterns even exist in the text and also where. Therefore is it for working with these type of algorithms very convenient, not only because we can prove the efficiency of them, but also because we can know as fast as possible if a specific fragment is to found on a text.

**Presentation and discussion of experimental results:**

If we take a look at the results, we can clearly see, that the Brute Force Algorithm is, as expected, the slowest. It is highly remarkable, that all and each one from the different algorithms found the desired/searched patterns.

The rule we can clearly observe is that the Boyer Moore Algorithm is the fastest one looking at the average times, however, the KMP Algorithm could be faster in some specific cases, as we can observe at the results (timetable.txt).

**Conclusions and prospects: summary and what else you could do in order to further improve your project.**

The Boyer More Algorithm has resulted on being the fastest in every text document we have tested, so we can assume that the Boyer Moore is on an average the fastest algorithm, regardless of which text document we are willing to take/analyze.

What else we can do is to search for more different text documents, maybe different languages and also test if we can find patterns that can´t be found manually. What we definitely could try is to run the algorithms more often in different hardware and software environments in order to see if it changes something by the relative running times. By doing it, we are actually testing which algorithms take the most resources form our devices. In the results we are becoming in our work can we see exactly which percentage of resources have been using the different algorithms while running.