**Formalia**

Jonathan Tadesse

jtadesse@kth.se

A comparision between two text searching algorithms Naive method and Boyer Moore.

Table of Contents

[Summary 2](#_Toc133072200)

[Task description 2](#_Toc133072201)

[Algorithm Overview 2](#_Toc133072202)

[Method 2](#_Toc133072203)

[Results 3](#_Toc133072204)

[Analysis 3](#_Toc133072205)

[Appendix 4](#_Toc133072206)

# Summary

# Task description

The task entails comparing the performance of two string search algorithms: the Naive Search and the Boyer-Moore Search. The aim is to assess their efficiency in searching for a pattern within a text file, depending on the size of the input string.

# Algorithm Overview

Naive Search Algorithm:

The Naive Search algorithm is a simple string search algorithm that compares each character in the text with the pattern, moving one character at a time. It is also known as the Brute Force algorithm. The algorithm is simple to implement and works well for small input sizes but can be inefficient for larger input sizes, as it scales linearly with the size of the input text.

Boyer-Moore Algorithm:

The Boyer-Moore algorithm is a more advanced string search algorithm that takes advantage of two heuristics, the bad-character rule and the good-suffix rule, to skip sections of the text and perform fewer character comparisons. This allows the algorithm to achieve better performance, especially for larger input sizes, as it can skip over parts of the text that are irrelevant to the pattern being searched.

# Method

The performance of the two algorithms was measured using Python code, with the **timeit** library to record the running time of each algorithm. The input text was extracted from the two texts 'henrickIbsen.txt' which is a collection of novels by Henrik Ibsen and “DNA\_sequence.txt” which is a collection of 1000000 character random DNA sequences. Various input sizes were tested by extracting the different target sizes from the main text, ranging from 1,000 to the size of the text minus the last 50 character. For each input size, a pattern of 50 characters was extracted from the end of the input text, and the running time of the algorithm searching for this pattern was measured. The measurements were repeated 100 times for each input size, and the average running time was calculated.

The choice of input sizes and the pattern length was made to cover a wide range of input scenarios and to ensure that the performance trends of the algorithms would be clearly observable. The use of the **timeit** library allowed for accurate timing measurements by executing the algorithms multiple times and calculating the average running time, minimizing the effect of any outliers or variations in system performance.

The performance results were then tabulated and visualized using **prettytable** and **matplotlib.pyplot**, respectively. These libraries were chosen for their ease of use and their ability to create clear, informative visualizations that facilitate the analysis of the results.

Due to any potential discrepancy that might be caused by the text composition I choose to deploy two different texts in my testing as mentioned earlier, one containing a standard novel and one a sequence of random DNA characters.

# Results

The results of the performance are displayed in four tables and four graphs one quadruple containing the results for the naïve method deployed on DNA\_sequence.txt and henrickIbsen.txt and one quadruple containing the results for Boyer Moore method deployed on henrickIbsen.txt and DNA\_sequence.txt.

The attached results have a descriptive title entailing which algorithm it belongs to.

|  |  |
| --- | --- |
| Naïve Method on henrickIbsen.txt | |
| Input Size | Running Time(s) |
| 1000 | 0.006078808000893332 |
| 5000 | 0.03050083100242773 |
| 10000 | 0.06466610200004652 |
| 15000 | 0.0991999109974131 |
| 50000 | 0.3331241100022453 |
| 100000 | 0.7048181139980443 |
| 200000 | 1.355961125002068 |
| 395470 | 3.5367745569965336 |

|  |  |
| --- | --- |
| Boyer Moore on henricklbsen.txt | |
| Input Size | Running Time (s) |
| 1000 | 0.0003887650018441491 |
| 5000 | 0.0003878920033457689 |
| 10000 | 0.0009849569978541695 |
| 15000 | 0.00048614399565849453 |
| 50000 | 0.00044912900193594396 |
| 100000 | 0.0004061389990965836 |
| 200000 | 0.00035155299701727927 |
| 395470 | 0.0003636449982877821 |

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

|  |  |
| --- | --- |
| Naïve search on DNA\_sequence.txt | |
| Input Size | Running Time (s) |
| 1000 | 0.01389915499748895 |
| 5000 | 0.18331498299812665 |
| 10000 | 0.12810796999838203 |
| 15000 | 0.1422205179987941 |
| 50000 | 0.4797119290014962 |
| 100000 | 1.0600121300012688 |
| 200000 | 2.5217602769989753 |
| 1016667 | 8.326197389993467 |

|  |  |
| --- | --- |
| Boyer Moore DNA\_sequence.txt | |
| Input Size | Running Time (s) |
| 1000 | 0.0004427719977684319 |
| 5000 | 0.00045495799713535234 |
| 10000 | 0.00042985600157408044 |
| 15000 | 0.00043076599831692874 |
| 50000 | 0.0004354289994807914 |
| 100000 | 0.00043311400077072904 |
| 200000 | 0.00047513599565718323 |
| 1016667 | 0.0004351780007709749 |

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

# Analysis

# From the empirical data obtained, it is evident that the Boyer-Moore algorithm exhibits superior performance compared to the Naive Search algorithm across all input sizes tested, for both the 'henrickIbsen.txt' and the "DNA\_sequence.txt" files. The consistently low computational complexity of the Boyer-Moore algorithm, even as the input size increases, demonstrates its high efficiency and optimal time complexity in searching for patterns within large input strings. This efficiency can be attributed to the advanced heuristics employed by the Boyer-Moore algorithm, specifically the bad-character rule and the good-suffix rule, which allow it to perform fewer character comparisons and efficiently traverse sections of the text, leading to faster execution times.In contrast, the Naive Search algorithm exhibits linear growth in running time with respect to the input size, demonstrating its suboptimal computational complexity when dealing with larger input strings, as it compares each character in the text sequentially without any algorithmic optimizations. This disparity in performance is evident in both the 'henrickIbsen.txt' and "DNA\_sequence.txt" files, highlighting the general superiority of the Boyer-Moore algorithm across various text compositions.Based on these results, it can be concluded that the Boyer-Moore algorithm is a more sophisticated and efficient choice for pattern searching within large input strings, irrespective of the text composition. The enhanced performance of the Boyer-Moore algorithm can be attributed to its advanced skipping heuristics, which allow it to efficiently search for patterns while minimizing the number of character comparisons, resulting in a more favorable time complexity.

# Appendix

Code for Boyer Moore algorithm :

def boyerMooreSearch(pattern, text):

textLength, patternLength = len(text), len(pattern)

if patternLength == 0:

# trivial search for an empty string

return 0

# dictionayr to store the last index of characters in the pattern

last = {char: i for i, char in enumerate(pattern)}

i = patternLength - 1 # index into text

k = patternLength - 1 # index into pattern

while i < textLength:

if text[i] == pattern[k]:

if k == 0:

# match was found for all characters in pattern

return i

else:

# go the next left character in both text and pattern

i -= 1

k -= 1

else:

# if pattern and text do not match at certain index, get the next right most character in pattern

# that matches the text character,

j = last.get(text[i], -1)

i += textLength -min(k, j + 1)

k = textLength -1

return -1

Code for Naïve algorithm:

def naiveSearch(pattern, text):

textLength, patternLength = len(text), len(pattern)

# outer loop checks all possible index offsets on text

for i in range(textLength-patternLength + 1):

idx = 0

# inner loop checks if pattern matches corresponding sequence of text

while (idx < 0 and text[i + idx] == pattern[idx]):

idx += 1

if(idx == patternLength):

return i

return -1