Formalia
Jonathan Tadesse
9707294311
jtadesse@kth.se

Summary

In this study, the performance of two string searching algorithms, Naive Search and Boyer-Moore Search were tested and compared on their efficiency in searching for a pattern string within two text files of varying input sizes. The Boyer-Moore algorithm employs a bad match table which reduces unnecessary character comparisons making the average time complexity O(n/m). The Naive algorithm compares each character at a time with the pattern string, making the average time complexity O(n*m). Two texts were used “henricklbsen.txt” and “DNA_sequence.txt”, the first is a novel with a rich variety of character combinations while the latter is a sequence of 1 million DNA bases. Both text searching algorithms were tested with various target sizes as well as pattern sizes and their average running time was plotted and tabulated for each input size. The results indicated that the Boyer-Moore algorithm performed better across all input sizes as well as text types.

Task description

In this task a comparison has been done between two string search algorithms Naive search and the Boyer-Moore search on their performances when searching for a pattern string of various size and composition. The aim is to assess their efficiency when looking for a pattern in a given text.

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 method. The average time complexity of the Naïve algorithm is O(m*(n-m+1))≈O(m*n) where n is the target string and m is the pattern string.

Boyer-Moore Algorithm:

The Boyer-Moore algorithm utilize a bad match table lookup that is obtained by doing the following operation on the pattern string; for each character in the string, we obtain firstly its index position in the pattern and add a one to it. We then subtract this value from the total length of the pattern and pair the resulting value to the character. If there are reoccurrences of a character, we simply update the value of the previous paired value with the new one. The last index of the bad match table is left empty and is paired with the length of the whole pattern. Each paired number now represents how many shifts we need to make when there is a mismatch when checking from right to left.
The average time complexity of the Boyer Moore method is O(n/m) where m is the pattern and n the size of the string, however, in the worst-case scenario, the time complexity of the Boyer-Moore algorithm can degrade to O(nm), especially when the pattern has repeated characters. An example of a pattern and its bad match table.

“TEST” →

T	E	S	*
1	2	1	4

Method

Both text searching algorithms were implemented in python. A main script was created called metrics.py that used the **timeit** library to record the running time of each algorithm. Each algorithm was then tested with several pattern sizes directly extracted from two texts, “henricklbsen.txt” and “DNA_sequence.txt”. The first text is a collection of novels that contains 382307 characters and the second text “DNA_sequence.txt” contains 1 million characters. Various input sizes were tested by extracting different target sizes from the main text, ranging from 1,000 to the whole text, the corresponding pattern string was then obtained from the target string by removing the last 50 characters, the running time of the algorithms searching for this pattern was then 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 performance results were then tabulated and plotted using the module **prettytable** and **matplotlib.pyplot**. 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 another one a sequence of random DNA characters. The first text “henricklbsen.txt” would ensure that the testing would include a wide variety of character combinations, meanwhile the “DNA_sequence.txt” text would test the searching algorithms on limited character variations.

Results

The results of the performance are displayed in four tables and four graphs one quadruple containing the results for the Naïve algorithm deployed on DNA_sequence.txt and henricklbsen.txt and one quadruple containing the results for Boyer-Moore algorithm deployed on henricklbsen.txt and DNA_sequence.txt. The attached results have a descriptive title entailing which algorithm they belong to.

Table 1:
Naïve Method on henricklbsen.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

Table 2:
Boyer Moore on henricklbsen.txt

Input Size	Running Time (s)
1000	0.0003887650018441491
5000	0.0003878920033457689
10000	0.0004849569978541695
15000	0.0004861439956584945 3
50000	0.0004491290019359439 6
100000	0.0004061389990965836
200000	0.0003515529970172792 7
395470	0.0003436449982877821

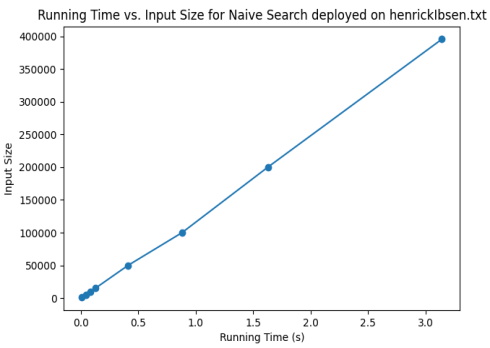
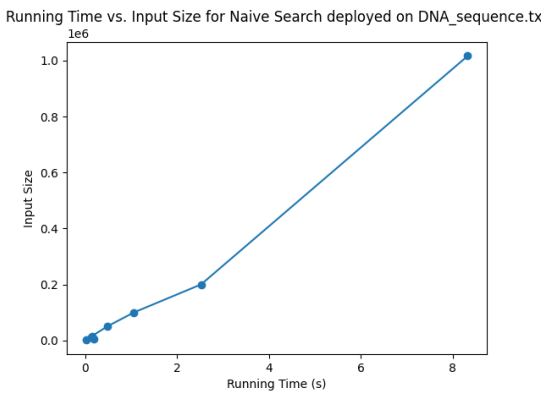
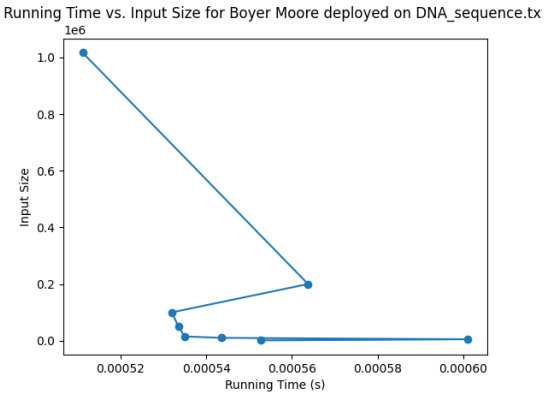


Table 3: Naïve search on DNA_sequence.txt	
Input Size	Running Time (s)
1000	0.01389915499748895
5000	0.18331498299812665
10000	0.12810796999838203
15000	0.1422205179987941
50000	0.4797119290014962
100000	1.0600121300012688
200000	2.5217602769989753
1016667	8.326197389993467

Table 4: 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	0.0003351780007709749





Analysis

Based on the results obtained, it is clear that the Boyer-Moore algorithm outperforms the Naive Search algorithm for all tested input sizes, in both the 'henrickIbsen.txt' and 'DNA_sequence.txt' files. As discussed in the algorithm overview, the best and worst-case time complexity for the Boyer-Moore algorithm is O(n/m) and O(n*m), respectively, where m represents the pattern string and n, the target string. The worst case arises when there are repeating characters in the target string, necessitating more character comparisons due to the skipping heuristics of the Boyer-Moore algorithm. This is evident in the results for the Boyer-Moore algorithm

applied to both 'henrickIbsen.txt' and 'DNA_sequence.txt' files.
Table 2 (Boyer-Moore deployed on “henricklbs.txt”) demonstrates an average running time of ≈0.00039 seconds, with a gradual decrease in running time as the input size increases, which aligns with the theoretical expectations. As the pattern string m increases, the time complexity O(n/m) decreases, resulting in fewer comparisons. On the other hand, Table 4 (Boyer-Moore deployed on “DNA_sequence.txt”) shows that the average running time was ≈0.000434 seconds. The slower running time could be attributed to the composition of the “DNA_sequence.txt” text, which consists only of the letters A, C, G, and T, forcing the Boyer-Moore algorithm to perform more comparisons and thus making the time complexity O(m*n). It is also observed that, unlike Table 2, there is a consistent running time across all input sizes in Table 4, except for the largest input size, where a significantly lower running time was recorded. This might be due to the recurring characters in the 'DNA_sequence.txt' file.
For the naive method as discussed in the algorithm overview, the average time complexity is O(m*(n-m+1) ≈ O(n*m), where n is the length of the target text and m is the length of the pattern string. The results for the Naive Search algorithm, when deployed on the 'henrickIbsen.txt' (Table 1) file, showed a linear relationship between the input size and running time. As the input size increased, the average running time was found to be proportional to the scaling factor between two consecutive input sizes. For instance, when the input size was n=1000, the running time was approximately t≈0.006 seconds, while for an input size of n=5000, the running time was approximately t≈0.03 seconds. This linear relationship can also be observed in the graph with the title "Naive Search Deployed on henrickIbsen.txt," which shows an almost linear increase in running time as the input size grows.
The average running time for Naive method on “henrickIbsen.txt” as calculated from table 1 is ≈ 0.77s. The average running time for Naive method on “DNA_sequence.txt” calculated from table 3 was shown to be ≈1.66s. The stark difference could be attributed the variety of character combinations in each of the text files, in the “henrickIbsen.txt” file we have more variety as its not restricted to the 4 letters A, C, G, and T like “DNA_sequence.txt” thus requiring less comparisons which explains the lower average running time.
We can conclude that Boyer-Moore search algorithm outperforms the Naïve search across all input sizes and type of texts. It can also be concluded that there is no area where the two methods have similar results and it’s reasonable to conclude that there is no scenario one would choose the Naïve algorithm over Boyer-Moore algorithm if the only parameter of judge is time complexity.

Sources

1. Goodrich, M.T., Goldwasser, M.H. and Tamassia, R. (2013) Data structures and Algortihms in python. New York: Wiley.

2. Ibsen and Archer (2023) The Collected Works of Henrik Ibsen, vol. 07 (of 11) by Henrik Ibsen, Project Gutenberg. Available at: https://www.gutenberg.org/ebooks/70566 (Accessed: April 21, 2023).

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
```