CS 412: Algorithms: Design and Analysis

Project Report

Group Members:

1. Muzammil Tariq
2. Syed Muhammed Abdullah Arif
3. Syed Muhammad Abbas Haider

Problem: String Matching Problem

Abstract:

String matching problem is a rather important topic in the world of computers. It is used widely in places like spam filtering, and plagiarism checking. The documents are converted into tokens and then matched with each other to find the relevant information. There are several types of approaches to this and for this project we will be talking about exact string matching. The algorithms we will be discussing in this are:

- Naïve Algorithm
- Knuth-Morris-Pratt Algorithm
- Boyer-Moore Algorithm

Introduction:

The <u>naïve</u> approach is the most basic approach to this and is important to understand the fundamentals of the string-matching process. The worst-case complexity of this is O(m*(n-m+1)), m over here is the size of the pattern that we are searching for and n is the size of the text that we are searching in. In every iteration the pattern is compared with a part of the text and in the next iteration it proceeds by matching it from the next element of the text. This causes a lot of useless comparisons since we would already have compared some of the elements of the text in our earlier comparisons.

The <u>Knuth-Morris-Pratt</u> (KMP) approach fixes this by creating an array beforehand which tells it which comparisons it can skip. An array is created which stores the number of comparisons that can be skipped after every mismatch. It does this by storing the maximum length it can go backwards in the pattern from the character where a mismatch occurs such that it forms a prefix of the pattern. Collectively this takes O(n + m) where O(n) is required for KMP and O(m) is required for the preprocess array creation.

The <u>Boyer-Moore</u> approach also works in similar ways to Knuth-Morris-Pratt. It can use two types of pre-computations, a bad character rule and good suffix rule, in our case we will only be looking into the prior rule. In the good suffix rule, we first generate an array, known as the mismatch shift table, that specifies the shortest distance from the right end of the pattern for each character in the pattern. Then, it compares and shifts the pattern from right to left like the naive approach but instead of matching from the start of the pattern it starts matching from the end of it, that is from the rightmost element. If a mismatch occurs during matching, with the help of the mismatch shift table it shifts the array to the right such that the first character same as the character mismatching in the text aligns with it to eliminate the mismatch. Then the matching is restarted from the right of the pattern. If the mismatched character does not exist in the pattern it shifts it by the length of the pattern. The complexity of this is O(nm + Σ) where O(nm) is required for the search and O(m + Σ) is required for the preprocessing where Σ is the English alphabet.

Procedure:

The three mentioned algorithms were tested and compared to each other by tracking their execution times with increasing size of the text and also in a separate experiment with increasing size of the pattern. The test with increasing size of text was conducted on two different samples of data one with large number of characters (all letters of the alphabet including both lower and upper case characters), and the other with small number of characters (binary numbers). The text and pattern data were generated from self-made random string generator functions. Also, as we were more interested in the growth of the execution time with increasing size rather than its absolute value, we measured them up to really large sizes (10,000) and applied an averaging function to only see the trend rather than the random fluctuations in the data. The result for each algorithm for a given test was plotted on the same graph to see how they performed in comparison to each other.

Results:

The results we achieved show that for text with several characters Boyer Moore had the smallest growth in execution time with increasing text size, whereas Knuth Morris had even larger growth than the Naive Algorithm, this can be seen in figure (1). When the experiment was repeated using binary text, refer to figure (2), it was found that Boyer Moore Algorithm performed the worst, having the largest growth in time. Whereas for the case of Knuth Morris algorithm, although it took more time than Naive algorithm for text sizes less than 8,000 characters, the Naive algorithm’s execution time surpasses that of the KMP algorithm on going a little beyond this limit. On increasing the ratio of the length of the pattern with respect to text size, see figure (3), it was found that the execution time of Knuth Morris algorithm grew, whereas Boyer Moore and Naive algorithm’s execution time decreased in a similar way to each other.

Conclusion and Discussion:

Through our research we found out that each of these implementations have specific advantages over the others. Theoretically naïve algorithm works best for short strings, Knuth Morris algorithm works best for binary strings, and Boyer Moore algorithm works best for long strings. Empirically we proved this by increasing the length of the text and pattern string and running the same algorithms repeatedly. Also, it was interesting to note that Knuth Morris Algorithm performs worse as the ratio of the pattern size to text increases, this might be because prepossessing time for the pattern increases with the pattern size. However, it has to be recognized that these tests were conducted only for data with uniformly randomly distributed characters. Future work can be done on data having a bias towards certain characters or key phrases, to see if they give off a different result.

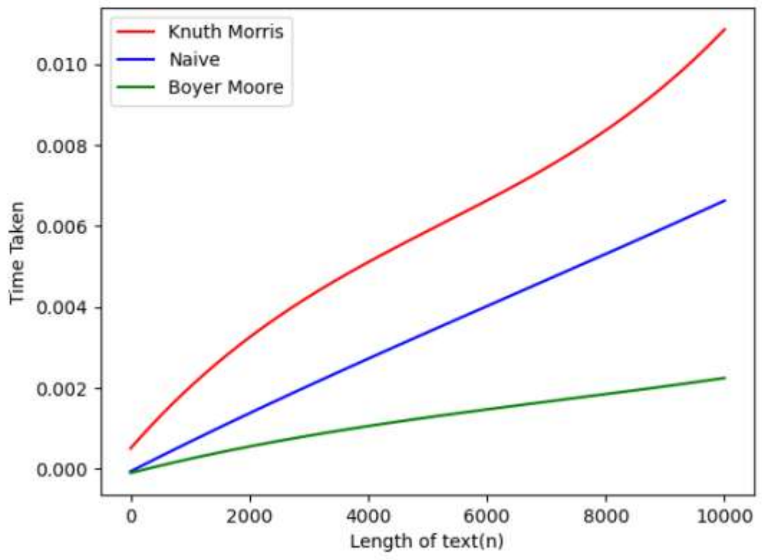

Figure 1: Text string with multiple letters i.e. not a binary string

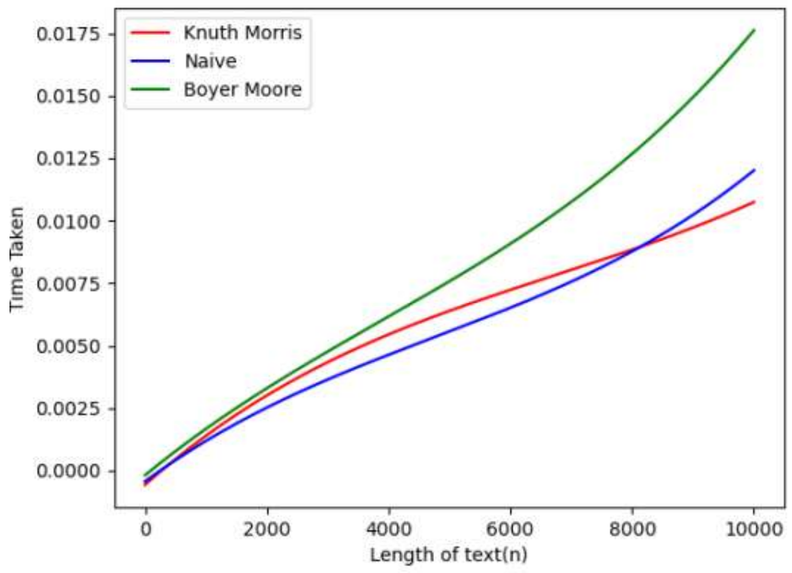

Figure 2: Text string with binary letters

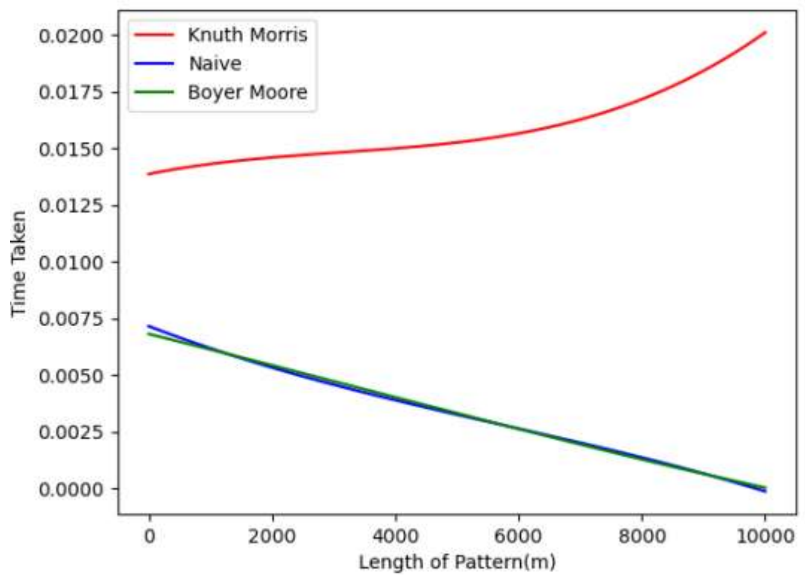

Figure 3: Testing with increasing length of the pattern that we have to search for

Complexity Table:

	Naïve Algorithm	Knuth Morris Pratt Algorithm	Boyer Moore Algorithm
Best Case Complexity	O(n)	O(n+m)	O(n/m+ Σ)
Average Case Complexity	O(n-m+1)	O(n+m)	O(n+m+ Σ)
Worst Case Complexity	O(m*(n-m+1))	O(n+m)	O(nm + Σ)

Note: n is the length of the text string and m is the length of the pattern string