CS 262P Project 4
Analyzing Exact Matching Algorithms

Submitted by: Pallavi Garg (87608678)

Contents

Problem Statement

Analysis of various algorithms used for exact pattern matching. The analysis involves analyzing the time complexity by measuring the time taken by each algorithm to find the first occurrence of the given pattern in the given text. The scenarios where pattern does not appear in the text are not considered as they won't tell us about the time taken (and number of comparisons) because pattern match would fail at the first comparison and the algorithms under consideration will move ahead in the text.

Algorithms

All the algorithms are implemented in C++ programming language as a single threaded program.

Brute Force

The brute force algorithm is the most naive algorithm to find exact matches in the string. It compares each character in the pattern with each character in the text sequentially. Starting by aligning the first character of the pattern with the first character of the text, it proceeds to check subsequent characters.
If a mismatch occurs, it moves to the next position in the text and restarts the comparison process with the first character of the pattern. This process continues until either a match is found or the end of the text is reached.

Knuth-Morris-Pratt (KMP)

KMP algorithm does preprocessing on the pattern first before starting to find the exact match.

Preprocessing

In this algorithm, a partial match table, called the failure function, is created. This function calculates the number of characters to skip in the pattern in case of a mismatch. It consists of an array of integers that represent the length of the longest proper prefix of the pattern that is also a suffix of its initial substring. In other words, for each position in the pattern, the failure function gives the length of the longest prefix that is also a suffix of the substring that ends at that position.

Search

The algorithm compares characters in the pattern with the corresponding characters of the text. If there is a mismatch, instead of restarting from the beginning of the pattern, it uses the information from the failure function table to determine the new starting position.This avoids unnecessary comparisons by skipping already matched characters and achieves linear time complexity.

Boyer-Moore-Horspool (BMH)

BMH algorithm does preprocessing on the pattern first before starting to find the exact match.

Preprocessing

The BMH algorithm also aims to reduce unnecessary character comparisons. It uses a bad character rule. It consists of an array of integers that represents the rightmost occurrence of the character in the pattern and uses this position to shift the pattern to the right.

Search

The algorithm compares characters of the pattern with the corresponding characters of the text from right to left. If a mismatch occurs, it consults the bad character rule to find the maximum shift possible based on the last occurrence of the mismatched character in the pattern.

Bitap

The Bitap algorithm is also known as the Shift-OR algorithm. It works by using a bitwise comparison of the pattern and the text, building a bitmap representation of the pattern. It scans the text from left to right, using bitwise operations to update the bitmap as it goes. It represents the pattern as a bitmask, where each bit position corresponds to a character and is set to 1 if the character is present in the pattern. The algorithm compares the bitmask of the pattern with a sliding window of the same length in the text, using bitwise operations to perform the comparison. If the bitwise AND operation between the pattern bitmask and the window equals the pattern bitmask itself, a match is found.

Rabin Karp

The Rabin Karp algorithm also does a minor preprocessing on the pattern.

Preprocessing

The Rabin-Karp algorithm uses hashing to quickly check for potential matches before performing detailed character comparisons. It precalculates a hash value for the pattern.

Search

The algorithm calculates the hash value of the characters appearing in the current window in the text using a hash function. The current window has length equal to the pattern length. If the hash values match, it performs a character-by-character comparison to confirm the match. To handle potential hash collisions, the algorithm employs a rolling hash technique, which efficiently updates the hash value as the window slides.

Test Setup

Each algorithm is run with the same text and pattern. Each iteration of the experiment is done on the same machine. I analyze the algorithms by considering two scenarios.

1. Random Strings
2. English strings and words

Random Strings Testing Setup

A random string of length 10,000 ASCII characters is characters. The initial test string is created without the letters - a, b, c, d, e, f, g, h, i, j.

The letters in the initial string at three sets of fixed position are then replaced with “abcdeabcdefghij”. The three position sets are: index 500 to 515, index 5000 to 5015, index 9000 to 9015. These ranges are chosen to analyze the algorithms where pattern is found in beginning, middle, and end of the large text.

The pattern to be searched ranges with length 2 - 10. For example “ab”, “abc”, “abcd”, and so on up to “abcdefghij”. The test has just once occurrence of each pattern.

The final results for each set of indices and each pattern length are then averaged in an effort to analyze the time complexity when the pattern is spread across the entire text.

English Strings and Words Testing Setup

In this setup, a long text is chosen. The patterns to be searched are of the length 2 - 10. For each length, 50 words are selected from the English dictionary. Each algorithm searches for the same pattern in the same text.

Observations

Random strings

Below is the chart for time taken by each of the algorithms to search a given random string of different length. X-axis represents the length of the patterns and Y-axis represents the time taken by the algorithm to find the first occurrence of the pattern in text.

For matching a random pattern, the KMP algorithm performs the best among the other algorithms in this test setting. Brute force and the Bitap algorithm also perform about the same in this test setting.

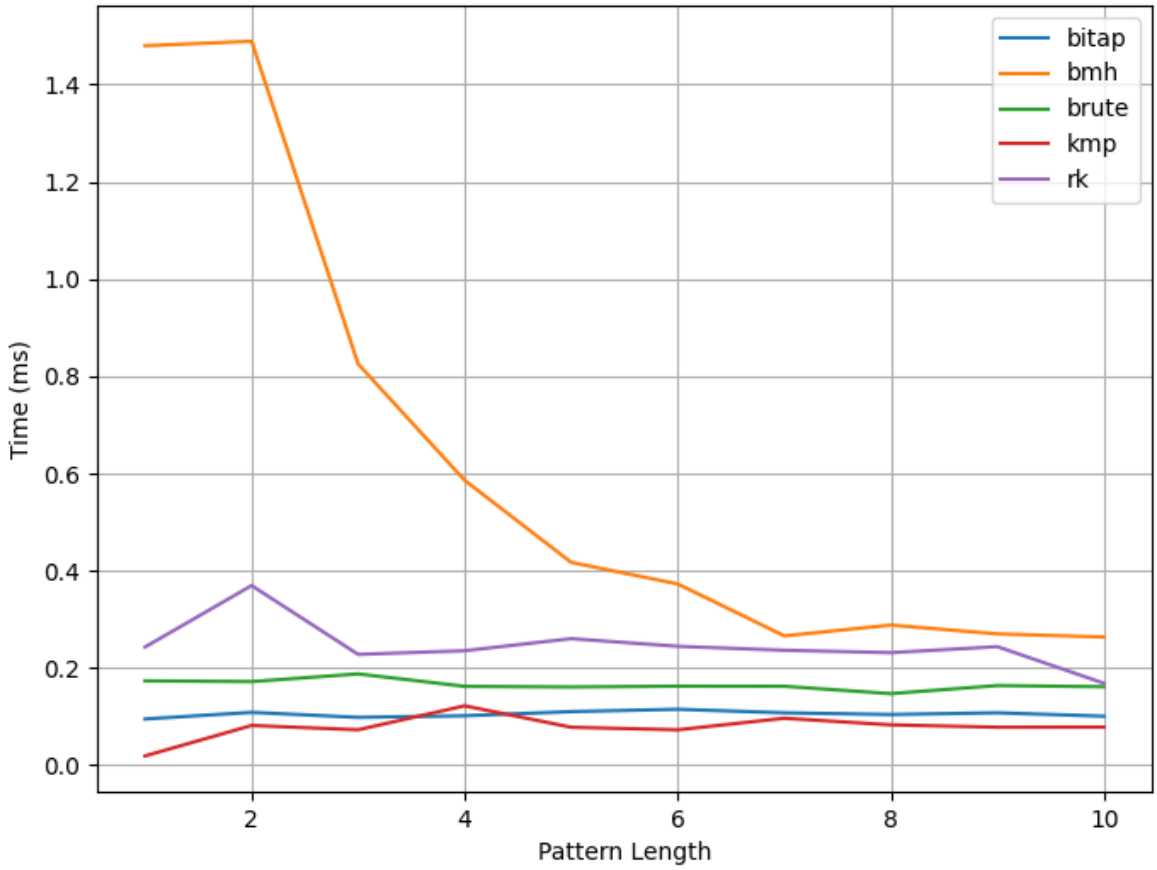

Chart 1: Random Strings

English Strings and Words

Below is the chart for time taken by each of the algorithms to search a given English word of different length.
For matching English patterns, the KMP algorithm performs the best among the other algorithms in this test setting. For pattern length > 3, the Bitap algorithm performance is quite close to that of the KMP algorithm.

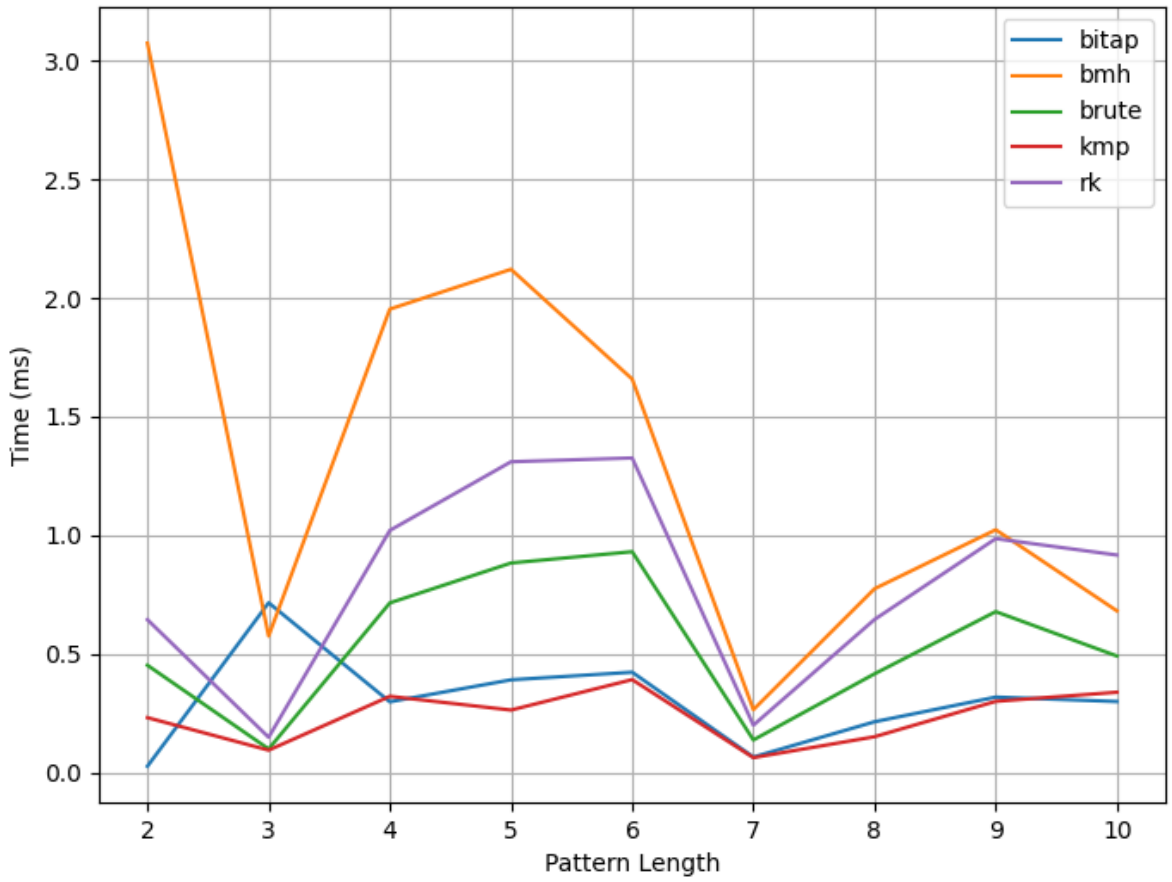

Chart 2: English Words

Conclusion

As per the observations, in both test settings KMP performs the best in both these test setups. The BMH algorithm performed poorly in this setup.
The reason for this observation is that:

1. The KMP uses the information about the previous comparisons which makes it fast for these scenarios. Because of the large degree of similarity among portions of pattern and text in English words setup, the BMH algorithm makes a lot of comparisons before it is able to find a match.
2. In both the test setups, text contains a large number of repeating characters and BMH algorithm relies on the “bad character shift” to determine how far to move the pattern when mismatch occurs. Which makes it shift the pattern by a small amount only, resulting in greater number of comparisons. On the other hand, the KMP algorithm uses the previous match information, which makes it much faster.

In general, it is noticeable from both the graphs, that brute force algorithm performance is better than both Rabin Karp and BMH algorithms. This shows that depending on the problem, brute force algorithm might be a better choice.

Overall, the performance of these algorithms depends on the characteristics of the specific problem being solved. It is important to consider the properties of the text and pattern when selecting an algorithm for string matching.