Pattern matching algorithms

Yakub Yakubov, Tarlan Sultanov, Daler Zakirov

April 2023

1. Introduction (Task 1 – Part A)

In this paper we approach the problem of finding pattern in a text. The problem is to implement Sunday, KMP, FSM, Rabin-Karp, Gustfield Z and Naive pattern matching algorithms, measure the performance of each and finally to do the modifications.

1.1. Naïve

Brute-force approach compares the first character of the pattern with the text, finding the match increase the pointer in both the strings, alternatively increasing the pointer of the text only. Time Complexity is $O(m * (n - m)$, where n is the length of the text and m is the length of the pattern.

1.2. Sunday

The algorithm works by scanning the text from left to right and attempting to match the pattern with the current position in the text. If the pattern is found, the algorithm reports the index of the first occurrence in the text. The algorithm precomputes a shift table for the pattern, which is a vector containing the shift values for each character in the ASCII table. The shift value represents the number of positions the pattern can be shifted to the right if the next character in the text does not match the current character in the pattern. The time complexity in average is $O(n + m)$ in the worst-case scenario is $O(n * m)$. The running time is affected by size of pattern and distribution of the characters in the text.

1.3. FSM

The finite automaton is built by constructing a table that stores the next state for each character in the alphabet for each state of the automaton. It first initializes the size of the automaton to larger than the length of the pattern. It then constructs the table by going through all possible states and characters in the alphabet. For each state and character, the transition is made to the next state by consuming the character. If the character does not match the current state, the automaton going to the previous state until a match is found, or the initial state is reached. The complexity of the time is $O(n)$, where n is the length of the text. The construction of table takes $O(m * 256)$, where m is the size of the pattern. So overall complexity is $O(n + m)$.

1.4. KMP

The Knuth-Morris-Pratt (KMP) algorithm is a pattern matching algorithm that finds all occurrences of a given pattern in a text string. It works by building a prefix-suffix table (also called the "failure function" or "LPS array") for the pattern, which enables efficient matching of the pattern with the text string without backtracking. The algorithm has a time complexity of $O(n + m)$, where n is the length of the text string and m is the length of the pattern.

1.5. Rabin-Karp

The Rabin-Karp algorithm is an algorithm that uses hashing. The algorithm hashes both the pattern and the first m characters of the text, where m is the length of the pattern. If the hashes match, it checks for a full match. If not, it slides the window of length m one character at a time, updating the hash value for the new window. If a match is found, it reports the starting position of the match in the text. The algorithm has a time complexity of $O(n + m)$ where n is the length of the text and m is the length of the pattern.

1.6. Gustfield Z

The algorithm is based on the Z algorithm, which is used to compute the Z values of a string. The Z values of a string represent the length of the longest substring starting from each position that is also a prefix of the string. The Gustfield Z algorithm first concatenates the pattern with the text using a separator character, such as '#'. Then, it computes the Z values of the concatenated string. If the Z value of a position is equal to the length of the pattern, then there is a match at that position. The algorithm has a time complexity of $O(n + m)$ where n is the length of the text and m is the length of the pattern. It is efficient for large texts and patterns.

2. Comparison (Task 1 – Part B)

2.1. Sunday and Gustfield Z comparison:

In general, the Sunday algorithm is known to perform well when the pattern and text have certain characteristics: particularly effective when the pattern being searched for has a small alphabet size and there are fewer potential matches in the text. In such cases, the Sunday algorithm can exhibit faster performance due to its ability to make larger skips in the text when a mismatch occurs. Gustfield Z is efficient when the pattern has a larger alphabet size or when there are more potential matches in the text.

For the empirical proof of that Sunday is twice faster in some cases, the file of the size 6.45 KB was used, and the pattern was the string of the 3 words. The time taken was measured several times and for the given scenario it took 11700, 10070, 11190 nanoseconds for the Sunday and 219190, 194640, 199020 for the Gusfield Z which proves that the former is twice as fast as latter.

2.2. KMP and Rabin Karp comparison:

KMP is known to be good for the repeated pattern in the text. Rabin Karp is particularly effective when the pattern and text have not a large alphabet size and the pattern length is small as hashing is used in this algorithm.

The same empirical prove is valid for the algorithms as the same text file and pattern string were used, which gave the time of 53420, 58300, 56030 for the former and 164630, 181790, 168850 for the latter, which proves that kmp is at least twice faster.

2.3. Rabin Karp and Sunday comparison:

For the last proof, the txt file of the size 451 KB was exploited, and the pattern was of size 193 KB giving the result of 11389460, 11697320, 11450770 for the Rabin Karp and 23917630, 24441510, 23976130 we can conclude that the statement that Rabin Karp is at least twice faster is also proved.

3. Methodology

Problem statement:

The goal of this project is to compare the performance of various string pattern matching algorithms and identify the situations where one algorithm is faster than the other, modify the algorithms to handle specific situations. Programming language used is C++. Algorithms to compare: Sunday, Gusfield Z, KMP, FSM, Rabin-Karp and Brute Force.

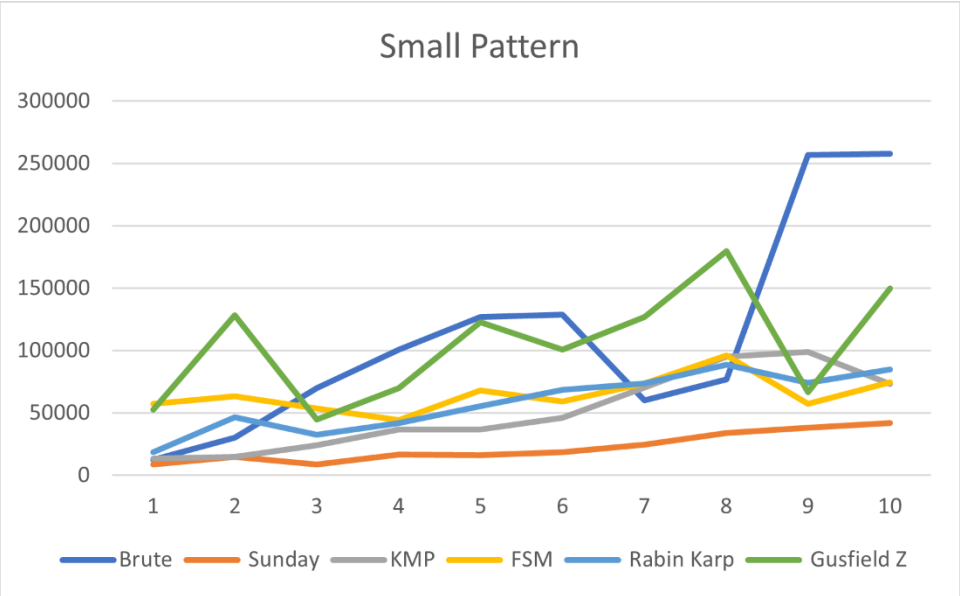
Book based strings of varying lengths will be generated for pattern and text. The lengths of pattern and text will be more than 100KB for the Empirical proof. The graphs will be plotted for the medium size pattern, small size pattern and big pattern. The generated data will be saved to txt files for reproducibility.

Experiment execution:

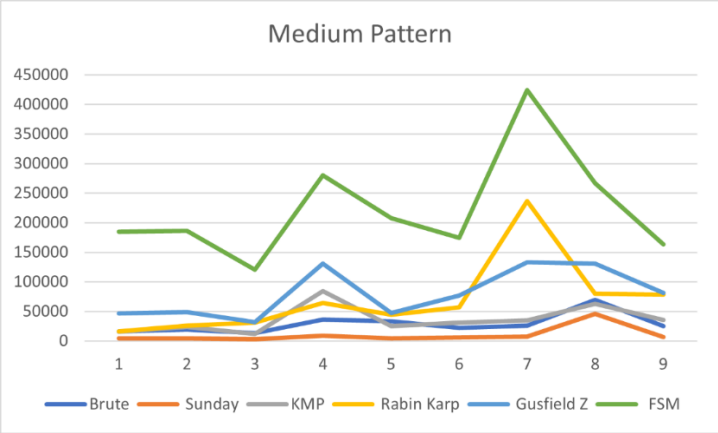
The experiments will be conducted in a Visual Studio Code. The performance metrics will be measured using the C++ time profiler libraries.

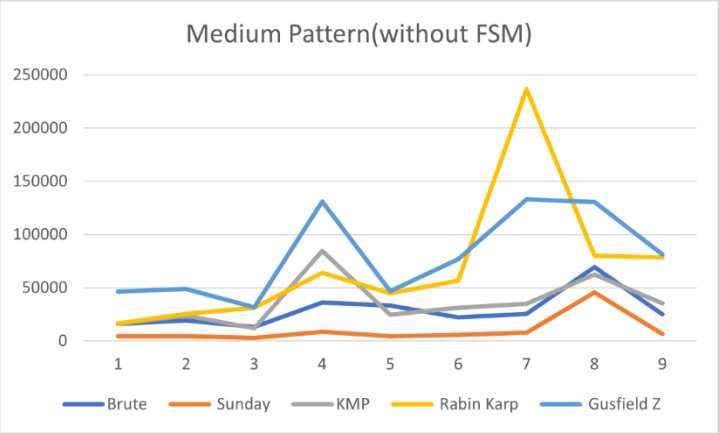
4. Results

The measurements were done by reading line by line from the txt format file, starting with 10 lines of the code, going up to 100 lines with the step of 10 lines. Firstly, the small pattern was introduced, and it is appeared that all the algorithms performed almost the same for it. All algorithms were showing linear growth, but Brute Force started to grow fast as soon as the size of the text came close to the 90 lines of the text. Moreover, Sunday was the fastest one among all the given.

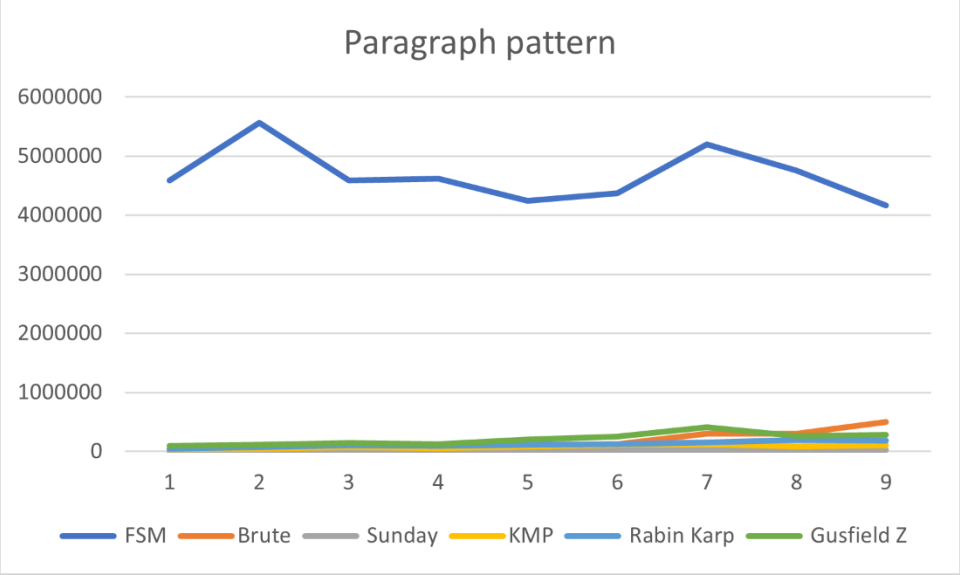


As the medium pattern of some sentences was introduced, the situation changed greatly for the FSM as algorithm will construct a transition table that has to be initialized for more characters in the alphabet, which means that the table will have a size proportional to the alphabet size. Then, the algorithm will iterate over the text characters one by one and perform constant-time transitions based on the current state and the next character until the pattern is either found or not. The other algorithms performing nearly equally along the way.

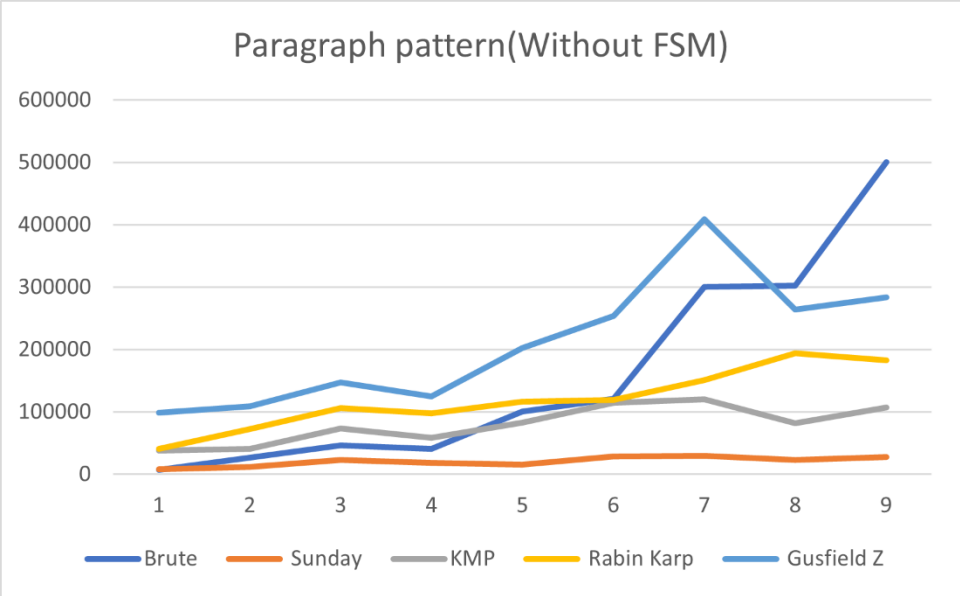




According to the measurements, FSM is the worst for the huge pattern being more than five time slower than the rest. As the difference among algorithms apart FSM is not clearly visible another picture was created.



It is apparent that there is a group of 3 (KMP, Robin-Karp, Sunday) produce the best results and with the time complexity growing slowly. As for the naïve and Gustfield Z, they took more time to finish being several times slower than the algo from the “group of 3”.



5. Modification to Naïve and Sunday to accept wildcards. (Task 2)

To extend the brute-force algorithm to handle wildcards, we modify the comparison process as follows. If the current character in the pattern string is a wildcard, we match it with any character in the text string. If the current character is a backslash, we ignore it and match the next character literally. We continue the comparison until we reach the end of the pattern string or find a mismatch.

To extend the Sunday algorithm to handle wildcards, we modify the shift table as follows. If the last character of the pattern string is a wildcard, we set the shift amount to 1, which means that we can shift the pattern string by only one character in case of a mismatch. If the last character is a backslash, we ignore it and use the previous character to compute the shift amount. During the comparison process, if the current character in the pattern string is a wildcard, we match it with any character in the text string. If the current character is a backslash, we ignore it and match the next character literally. We continue the comparison until we reach the end of the pattern string or find a mismatch.

6. “Jewish-style carp" (Task 3)

The task is to extend Rabin-Karp to handle 2D array or matrix of pixels. To remind, the algorithm is based on the observation that if two strings are equal, then their hash will also be equal. In the extension the algorithm starts by computing the hash values of the first row of the pattern and the last row of the pattern in the matrix. It then compares two hashes to check if they are the same, if so, the pattern is found at position (O, N-K), where N is the width of the picture, and K is the size of the sliding window, so it is the coordinate of top-left corner. If the hash values are different, the algorithm computes the scaling factor and then moves the pattern down the row at a time and computes hash value of the new pattern by subtracting the hash value of the top row of the previous pattern and adding the hash value of the bottom row of the new pattern. If the hash value of the new pattern matches the hash value of the pattern being searched for, then the algorithm returns true, indicating a match has been found. If the algorithm finishes searching the matrix without finding a match, it returns false. Overall, the code uses the rolling hash function to efficiently compute the hash values of the pattern and the substrings of the text.

To conclude, the string pattern matching algorithms were implemented, time complexity measured, graphs created, some algorithms were modified to accept wildcard and Robin-Karp was developed for searching in the matrix.