CS 412 - Project: String Matching Algorithms
Final Project Report

Team Members: Altaf Shaikh, Ma'aaz Shaheen, Neha Jafry, Qazi Talha
Student IDs: as05016, ms05176, nj05165, qt05099
Section: L2
Date: December, 2020
GitHub Link: https://github.com/Neha-Jafry/CS412---Algorithms-Project

1 Introduction

In layman terms, the string matching problem refers to finding all occurrences of a certain pattern within a given text. This means it looks for all the places where a specific string appears in a large body of text.
In formalizing this definition, we can say that:

- Let the text be represented by an array called T[1 . .n] of length n and let the pattern be represented by an array called P[1 . .m] of length m.
- The sequences in P and T are characters drawn from a finite alphabet A. For example, A = { a, b, c, . . . , z }.
- All occurrences of the pattern within the text are said to appear at a shift of S, where S is the index where the occurrence of the pattern begins in the text.

The string matching problem is considered important as there are several applications such as:

- Searching for words in a long text
- Search Engines
- Spell Check
- Plagiarism Check
- DNA Sequencing

For this reason it is important to come up with efficient algorithms in order to optimize the task. In our paper we shall look at the following algorithms:

- Naive String Matching Algorithm
- Rabin-Karp Algorithm
- Knuth-Morris-Pratt (KMP) Algorithm

2 Naive String Matching Algorithm

2.1 Approach

The naive approach to the string matching problem uses brute force to search for the pattern in the given text. For all indices of the text, from 0 to n-m, we check for matching in the pattern.

2.2 Time Complexity

Since one iterator is required for going over the text of length n, from 0 to n-m, where m is the length of the pattern, and for each index another iterator is required to make the actual comparison to the pattern of length m, then the worst case complexity of the naive algorithm, when all comparisons are needed, is:

$$O((n-m)*m) = O(nm)$$

In case the first character of the pattern is not present in the algorithm is not present in the text at all, the no further comparisons with the pattern are required for each index of the text. This means that only one comparison is made for each of the n-m indices of the text. This best case has the complexity:

$$O(n-m) = O(n)$$

3 Rabin-Karp Algorithm

3.1 Approach

The Rabin-Karp approach to the string matching problem uses hash values computed via a rolling hash function to first match the pattern to be search with the substring of the same length from the main string. If the hash value matches, only then does it compare the actual substring and pattern. If the substring does not match, it wastes no time in comparison, and moves on to the next substring, calculating its hash value in O(1) time due to the rolling hash function and re-does the comparision process.

3.2 Time Complexity

The time complexity of this algorithm is dependent largely upon the rolling hash function. To understand this, we must realise that there might be times when our hash values for the sub-string and the pattern might be the same, despite the pattern and the sub-string being different. This is known as a *spurious hit*

and adds to the time taken for the algorithm as it wastes more time checking the substring. As the algorithm runs from 0 to (n - m) iterations, same as the naive algorithm, the time complexity is dependent upon the number of spurious hits. Spurious hits can be reduced by the use of a better hash function, which would give different strings different hash values.

In the worst case, their will be a spurious hit at every iteration, in which case the pattern and substring will be compared everytime. So, this results in a special case of Rabin-Carp, which is the same as the naive algorithm i.e. substring checked for each iteration. So, worst case is:

$$O(n + m)$$

In case their are no spurious hits, our time complexity would then drop down to

$$O(n - m + 1)$$

as the substring would only be checked once when the hash value matches, and as it is not a spurious hit, would result in the correct comparison.

4 Knuth-Morris-Pratt (KMP) Algorithm

4.1 Approach

The intuition behind this algorithm came from identifying a key weakness in the algorithm of Naive approach. In a normal Naive approach, when a character mismatch occurs, the algorithm goes back to the character after from where the matching started. In many real world cases, this approach suffers heavily. This is the weakness that was targeted by KMP algorithm. The KMP algorithm prepares a table initially using the pattern. The table is of an array form known as 'longest proper prefix which is also a suffix' (lps). The lps array makes use of the pattern to be found in the text and prepares its lps array. The lps array holds the value for each index, representing the length of the pattern that has appeared previously in the pattern.

After preparing the lps array, we start comparing the text and the pattern normally. In the case that a mismatch occurs, we use the corresponding character's (the one that mismatched) value from the lps array and decide the next characters of the pattern to be matched. The idea is to not waste time matching a character that we know has repeated previously in the pattern and would match anyway. This reduces the time we are spending on comparing the pattern with the text.

4.2 Time Complexity

In order to prepare the lps array, the optimal algorithm of KMP traverses the pattern once. This gives us the time complexity of the preprocessing as:

$$O(m)$$

where m is the length of the pattern.

In the KMP algorithm, we bypass re-examination of previously matched characters using the information in the lps array. If the text is of length n and the patter is of length m then our worst case time complexity becomes:

$$O(n+m)$$

The m in the time complexity comes from the preprocessing. However, the other n comes from the matching algorithm itself. If we let t be the index traversing the text and p be the index traversing the pattern, we can note that during the algorithm, t is never decremented. The index p is only decremented when we encounter a mismatch after some matches. If we have 4 matches, then p could be decremented 4 times. In the absolute worst case, if we have n mismatches, that means that we had n matches too and must have traversed the entire text. This makes our worst case time complexity as:

$$O(n+n) = O(2n) = O(n)$$

5 Comparative Complexities

Naive and Rabin-Karp Algorithms have the same worst case complexity, which is $O(nm)$. Although the complexities of these two algorithms is comparable, the probability for the worst case for Rabin-Karp algorithm is much smaller than the probability of the worst case for the naive algorithm. Hence in general the Rabin-Karp algorithm is better than the naive approach. KMP Algorithm has the lowest time complexity, which is in linear time, $O(n+m)$.

6 Empirical Analysis

The string matching problem has a number of parameters that affect the resulting time complexities. These are the length of the test n and length of pattern m as well as the type of the string being considered. Surprisingly, the type of the string is a major factor that affects the time complexities. For the empirical analysis of these multiple parameters, we constrained our analysis to various cases that incorporated the type of strings and in few cases the length n.

6.1 Case 1: No matches

Table 1 shows the average runtime calculated on 100 runs for each algorithm for a text in which no matches for the pattern were found but the pattern and text had common substrings. The text and pattern used are given below:
text = brow $*$ 200000
pattern = brown

The results are as expected, KMP algorithm, has the lowest runtime, whereas naive and Rabin-Karp algorithms have comparable runtimes but naive algorithm is slower, this is due to the hash function used in Rabin-Karp which reduces the amount of comparisons made.

Table 1: Case 1: No matches

Algorithm	Average Runtime
Naive	0.6515625
Rabin-Karp	0.49765625
KMP	0.18921875

6.2 Case 2: Multiple matches

Table 2 shows the average runtime calculated on 100 runs for each algorithm for a text in which multiple matches for the pattern were found. The text and pattern used are given below:
text = the quick brown fox jumped over the lazy dog. $*$ 200000
pattern = over
The results are similar to the previous case, KMP algorithm, has the lowest runtime, whereas naive and Rabin-Karp algorithms have comparable runtimes but naive algorithm is slower.

Table 2: Case 2: Multiple matches

Algorithm	Average Runtime
Naive	5.5415625
Rabin-Karp	4.529375
KMP	1.8690625

6.3 Case 3: Many Matches

Table 3 shows the average runtime calculated on 100 runs for each algorithm for a text in which many matches for the pattern were found. The text and pattern used are given below:
text = A $*$ 200000
pattern = AAA
The results in this case are also expected, here too KMP algorithm, has the lowest runtime, however unlike the aforementioned two cases, here the Rabin-Karp algorithm performs worse that the naive algorithm. This is because in

this case the hash value gives a match each time so on top of calculating the hash values, the algorithm also has to make the caparisons to check for the matching pattern each time, whereas in the naive algorithm the same amount of comparisons are made but since it doesn't calculate hash values it performs better.

Table 3: Case 3: Many matches

Algorithm	Average Runtime
Naive	0.0746875
Rabin-Karp	0.13734375
KMP	0.03234375

6.4 Case 4: Real World Example

Table 4 shows the average runtime calculated on 100 runs for each algorithm for a text in which many matches for the pattern were found. The text used is a sample text from a random text generator and pattern used is some word in the text. The results in this case are again similar to the first two cases. KMP algorithm, has the lowest runtime and naive and Rabin-Karp Algorithms again have comparable runtimes, where naive algorithm is slower.

Table 4: Case 4: Real world example

Algorithm	Average Runtime
Naive	0.00359375
Rabin-Karp	0.00328125
KMP	0.0009375

7 Conclusion

The empirical analysis of Cases 1, 2 and 4 was in line with the theoretical complexities, however, Case 3 does not follow the norm, as the Rabin-Karp algorithm takes a higher time then naive.