A Mini Project Report

on

"IMPLEMENTATION OF NAIVE STRING-MATCHING ALGORITHM AND RABIN-KARP ALGORITHM FOR STRING MATCHING"

by

Samir Hasan Shaikh (.)

Kalyani Arjun Sansare (.)

Lina Pravin Birari (.)

Under the guidance of

Prof. S.S.Hinge

Department of Computer

Engineering

S.M.E.S. Sanghavi College

of Engineering,Nashik

SAVITRIBAI PHULE PUNE UNIVERSITY

2022_2023

Department of Computer Engineering

S.M.E.S. Sanghavi College of Engineering

Date:

CERTIFICATE

This is to certify that, Samir Hasan Shaikh (.) Kalyani Arjun Sansare (.) Lina Pravin Birari (.) of class **T.E Computer**; have successfully completed their mini project work on **"Implementation Of Naive String-Matching Algorithm And Rabin-Karp Algorithm For String Matching"**

at **Institute of Technology,Management & research, Nashik** in the partial fulfillment of the Graduate Degree course in **B.E** at the department of **Computer Engineering** in the academic Year 2022-2023 Semester – I as prescribed by the Savitribai Phule PuneUniversity.

{ Prof. S.S.Hinge}	**{ Prof. Puspendu Biswas }**
Project Guide	**Head of Department**

Acknowledgements

With deep sense of gratitude we would like to thanks all the people who have lit our path with their kind guidance. We are very grateful to these intellectuals who did their best to help during our project work.

It is our proud privilege to express deep sense of gratitude to **Prof. A.D. Lokhande,** Principal Sanghvi college of engineering, Nashik for his comments and kind permission to complete this project. We remain indebted to Prof. Puspendu Biswas, H.O.D. of Computer Engineering Department for his timely suggestion and valuable guidance.

The special gratitude goes to Prof. S. S. Hinge excellent and precious guidance in completion of this work .We thanks to all the colleagues for their appreciable help for our working project. With various industry owners or lab technicians to help, it has been our endeavor to throughout our work to cover the entire project work.

We also thankful to our parents who providing their wishful support for our project completion successfully .And lastly we thanks to our all friends and the people who are directly or indirectly related to our project work.

Samir Hasan Shaikh (.)

Kalyani Arjun Sansare (.)

Lina Pravin Birari (.)

TABLE OF CONTENTS

Chapter 1
Introduction

1.1 Abstract

String matching is a fundamental problem in computer science and has numerous real-world applications, including text search, data retrieval, and pattern recognition. This project presents the implementation of two widely-used string matching algorithms, the Naive String Matching Algorithm and the Rabin-Karp Algorithm, along with a comparative analysis of their performance.

The Naive String Matching Algorithm is a simple and intuitive approach that involves checking for a pattern's occurrence in a text by sliding the pattern over the text one character at a time. This algorithm serves as a baseline for comparing the efficiency and effectiveness of more advanced algorithms, such as Rabin-Karp.

The Rabin-Karp Algorithm is an advanced string matching technique that exploits the concept of hashing to achieve faster matching. It uses a rolling hash function to quickly identify potential matches and then employs additional checks to confirm them. Rabin-Karp is particularly efficient for searching patterns in large texts.

In this project, we aim to:

1. Implement the Naive String Matching Algorithm in Python.
2. Implement the Rabin-Karp Algorithm in Python.
3. Perform a comparative analysis of the two algorithms.

The implementation and comparative analysis will allow us to understand the strengths and weaknesses of these algorithms and make informed decisions about their use in different applications.

1.2 Introduction

String matching, the process of finding occurrences of a specific pattern within a given text, is a common problem in computer science and has wide-ranging applications. Whether it's searching for a keyword in a document, identifying plagiarism in academic papers, or parsing DNA sequences, efficient string matching algorithms are essential for computational tasks.

Two key algorithms for solving this problem are the Naive String Matching Algorithm and the Rabin-Karp Algorithm.

Naive String Matching Algorithm

The Naive String Matching Algorithm, also known as the Brute-Force algorithm, is a simple and straightforward approach to string matching. It operates by systematically sliding a fixed-size window (the pattern) over the text, character by character, comparing the pattern with the corresponding portion of the text. If a match is found, the algorithm reports the starting position of the match. While simple to understand and implement, the Naive Algorithm can be quite slow, especially for large texts or complex patterns.

Rabin-Karp Algorithm

The Rabin-Karp Algorithm is a more advanced string matching technique that leverages the concept of hashing to expedite the matching process. Instead of comparing the pattern with the text character by character, Rabin-Karp computes a hash value for the pattern and for each window of text it examines. If the hash values match, the algorithm performs an additional character-by-character check to confirm a match. The Rabin-Karp Algorithm is particularly useful when dealing with large texts or when multiple patterns need to be searched simultaneously.

1.3 Problem Definition and Scope

Problem Definition

The problem at hand is efficient string matching in a given text, which involves finding occurrences of a specific pattern within the text. This problem has wide-ranging applications in computer science and information retrieval, including text search, plagiarism detection, DNA sequence analysis, and more. The primary challenge is to design and implement string matching algorithms that balance accuracy and efficiency in different scenarios.

Scope of the Project

The scope of this project encompasses the implementation and analysis of two distinct string matching algorithms: the Naive String Matching Algorithm and the Rabin-Karp Algorithm.
The project aims to address the following aspects within this defined scope:

1. **Algorithm Implementation**

1.1 Naive String Matching Algorithm

- Implementation of the Naive String Matching Algorithm in Python.
- Verification of its correctness and functionality.

1.2 Rabin-Karp Algorithm

- Implementation of the Rabin-Karp Algorithm in Python, which includes the use of a rolling hash function.
- Validation of the algorithm's correctness.

2. **Comparative Analysis**

In summary, the project's scope is well-defined, covering the implementation, analysis, and practical considerations of two string matching algorithms, ultimately providing insights into their strengths, weaknesses, and use cases. The project aims to observe differences in the working of both algorithms for the same input.

Chapter 2

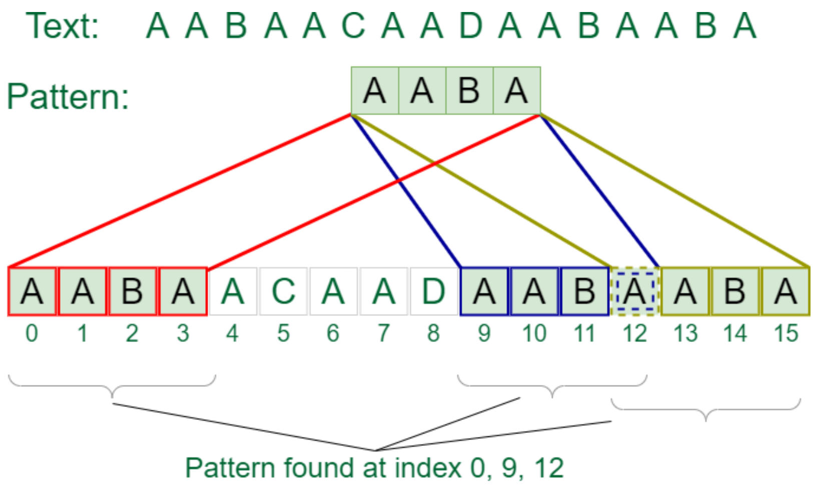
Methodology

2.1 Naïve Algorithm

- It is the simplest method which uses brute force approach.
- It is a straight forward approach of solving the problem.
- It compares first character of pattern with searchable text. If match is found, pointers in both strings are advanced. If match not found, pointer of text is incremented and pointer of pattern is reset. This process is repeated until the end of the text.
- It does not require any pre-processing. It directly starts comparing both strings character by character.

Algorithm for Naive string matching :

```
Algorithm - NAIVE_STRING_MATCHING (T, P)
for i←0 to n-m do
if P[1......m] == T[i+1.....i+m] then
print "Match Found"
end
end
```



Text: A A B A A C A A D A A B A A B A
Pattern: A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12

- **What is the best case of Naive algorithm for Pattern Searching?**

  The best case occurs when the first character of the pattern is not present in the text at all.

- **What is the worst case of Naive algorithm for Pattern Searching?**

  The worst case of Naive Pattern Searching occurs in the following scenarios.

  When all characters of the text and pattern are the same.

  Worst case also occurs when only the last character is different.

2.2 Rabin-Karp Algorithm

In the Naive String Matching algorithm, we check whether every substring of the text of the pattern's size is equal to the pattern or not one by one.

Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.

1. Pattern itself
2. All the substrings of the text of length m which is the size of pattern.

How is Hash Value calculated in Rabin-Karp?

Hash value is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.

Here's how the hash value is typically calculated in Rabin-Karp:

Step 1: Choose a suitable base and a modulus:

- Select a prime number 'p' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
- Choose a base 'b' (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value: Set an initial hash value 'hash' to 0.

Step 3: Calculate the initial hash value for the pattern:

- Iterate over each character in the pattern from left to right.
- For each character 'c' at position 'i', calculate its contribution to the hash value as 'c * (bpattern_length – i – 1) % p' and add it to 'hash'.
- This gives you the hash value for the entire pattern.

Step 4: Slide the pattern over the text: Start by calculating the hash value for the first substring of the text that is the same length as the pattern.

Step 5: Update the hash value for each subsequent substring:

- To slide the pattern one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position ‘i’ to ‘i+1’ is:

  hash = (hash - (text[i - pattern_length] * (bpattern_length - 1)) % p) * b + text[i]

Step 6: Compare hash values:

- When the hash value of a substring in the text matches the hash value of the pattern, it’s a potential match.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

Step-by-step approach:

- Initially calculate the hash value of the pattern.
- Start iterating from the starting of the string:
- Calculate the hash value of the current substring having length **m**.
- If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
- If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
- Return the starting indices as the required answer.

2.3 Naïve Algorithm Implementation

```
# IMPLEMENTATION OF NAIVE ALGORITHM

def search(pat, txt):
        M = len(pat)
        N = len(txt)

        # A loop to slide pat[] one by one
        for i in range(N - M + 1):
                j = 0

                # For current index i, check
                # for pattern match
                while(j < M):
                        if (txt[i + j] != pat[j]):
                                break
                        j += 1

                if (j == M):
                        print("Pattern found at index ", i)



# Driver Code

input_txt  =  str(input("Enter  the  Text  :  "))
input_pat = str(input("Enter the Pattern : "))
if_name__== '_main_':
        txt = input_txt
        pat = input_pat

        # Function Call
        search(pat, txt)
```

```
Enter the Text     : AABAACAADAABAABA
Enter the Pattern  : AABA

Pattern  found  at  index  0
Pattern  found  at  index  9
Pattern found at index  12
```

3.2 Rabin-Karp Implementation

```
# IMPLEMENTATION OF RABIN-KARP ALGORITHM

# d is the number of characters in the input alphabet
d = 256

# pat -> pattern
# txt  -> text
# q -> A prime number

def search(pat, txt, q):
        M = len(pat)
        N = len(txt)
        i = 0
        j = 0
        p = 0 # hash value for pattern
        t = 0 # hash value for txt
        h = 1

        # The value of h would be "pow(d, M-1)%q"
        for i in range(M-1):
                h = (h*d) % q

        # Calculate the hash value of pattern and first window of text
        for i in range(M):
                p = (d*p + ord(pat[i])) % q
                t = (d*t + ord(txt[i])) % q

        # Slide the pattern over text one by one
        for i in range(N-M+1):
                # Check the hash values of current window of text and
                # pattern if the hash values match then only check
                # for characters one by one
                if p == t:
                        # Check for characters one by one
                        for j in range(M):
                                if txt[i+j] != pat[j]:
                                break
                                else:
                                        j += 1

                        # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
                        if j == M:
                                print("Pattern found at index " + str(i))

                # Calculate hash value for next window of text: Remove
                # leading digit, add trailing digit
                if i < N-M:
                        t = (d*(t-ord(txt[i])*h) + ord(txt[i+M])) % q

                        # We might get negative values of t, converting it to
                        # positive
                        if t < 0:
                                t = t+q



# Driver Code
```

```
input_txt  =  str(input("Enter  the  Text  :  "))
input_pat = str(input("Enter the Pattern : "))
if_name__== '_main_':
        txt = input_txt
        pat = input_pat

        # A prime number
        q = 101

        # Function Call
        search(pat, txt, q)
```

```
Enter the Text     :  AABAACAADAABAABA
Enter the Pattern  : AABA

Pattern  found  at  index  0
Pattern  found  at  index  9
Pattern found at index 12
```

Chapter 4

Results and Analysis

The two approaches the Rabin-Karp algorithm and the Naive algorithm for string searching. Let's compare them and observe the output on passing the same input:

Rabin-Karp Algorithm:

The Rabin-Karp algorithm uses a rolling hash function to compare the hash values of the pattern with the hash values of overlapping substrings in the text. If the hash values match, it further compares the characters one by one to confirm a match. This algorithm is efficient for searching patterns in texts and is known for its average-case performance

Naive Algorithm:

The Naive algorithm, on the other hand, straightforwardly compares each character of the pattern with the corresponding character in the text, one by one, while sliding the pattern over the text. It has a time complexity of O(N*M), where N is the length of the text, and M is the length of the pattern, making it less efficient for large texts and patterns.

When these two algorithms are provided with the same input, they will generally get the same output, which is the indices where the pattern is found in the text. However, the key difference lies in the performance and efficiency of the algorithms, especially for larger texts and patterns.

For the same input, the Rabin-Karp algorithm is expected to be significantly faster than the Naive algorithm because it takes advantage of hash functions and can avoid unnecessary character comparisons. It works well when the text and pattern are large.

The Naive algorithm is simple and easy to understand but can be very slow for large inputs since it compares each character individually, even if the hash values don't match.

In practice, the choice of algorithm depends on the size of text and pattern, and the specific use case. If we have a large dataset, the Rabin-Karp algorithm is a better choice for efficiency.

Chapter 5

Conclusion

In conclusion, when comparing the Rabin-Karp algorithm and the Naive algorithm for string searching, we can make the following observations:

1. Rabin-Karp Algorithm:
   The Rabin-Karp algorithm is an efficient pattern-matching algorithm that uses a rolling hash function. Its time complexity in the worst case is O(M + N), where M is the length of the pattern, and N is the length of the text. It is particularly useful when dealing with large texts and patterns, where it outperforms the Naive algorithm.

2. Naive Algorithm:
   The Naive algorithm is a straightforward approach that compares characters one by one when sliding the pattern over the text. Its time complexity in the worst case is O(N * M), making it less efficient for larger texts and patterns. It is simple and easy to understand but can be slow for large datasets.

In practice, the choice between these algorithms depends on the specific use case and the size of the input data. If efficiency is a concern, the Rabin-Karp algorithm is generally a better choice for large texts and patterns due to its linear time complexity. However, for small datasets or simple implementations, the Naive algorithm may suffice.

Ultimately, understanding the strengths and weaknesses of both algorithms allows to select the most appropriate one for your particular scenario, optimizing both performance and simplicity as needed.

Chapter 6

Bibliography

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. "Algorithms" by Robert Sedgewick and Kevin Wayne.
3. GeeksforGeeks (https://www.geeksforgeeks.org)
4. Wikipedia (https://en.wikipedia.org)