

COSC 528 Design and Analysis of Algorithms

Project Title: String Searching Algorithms

Assignment Due: November 30th, 2021

Faculty: Dr. James Gil De Lamadrid

Team Members:

Syed Ali

Johnathan Ngaya

Contents

[Introduction 2](#_Toc88762303)

[Algorithms 2](#_Toc88762304)

[Analysis 5](#_Toc88762305)

[Test 6](#_Toc88762306)

[Results 6](#_Toc88762307)

[References: 7](#_Toc88762308)

# Introduction

The goal of string searching algorithms is to find the location of a specific text pattern within a larger body text. The larger body of text can be described either as a sentence, paragraph, or book. The topic of string searching has greatly influenced the computer science community and plays an important role in solving various real-world problems.

String matching algorithms are commonly used in plagiarism detection, where the documents to be compared are decomposed into string arrays or tokens and are compared with various string-matching algorithms such as Knuth-Morris-Pratt algorithms. String matching algorithms are also widely used in Bioinformatics and DNA Sequencing since these solve the problems of genetic sequences to find DNA patterns. Various string-matching algorithms are collectively used to find the occurrence of the pattern set. String matching algorithms are also widely used in search engines or content search in large databases, where string matching algorithms are used to categorize and organize the data efficiently. Categorization is done using search keywords, thus making it easier for the user to find information they are searching for. Some other common applications of string-matching algorithms are in digital forensics, spelling checker, spam filters, and intrusion detection system. In this paper we focus on two string matching algorithms which are Naïve algorithm for pattern searching (Brute-Force) and Knuth-Morris-Pratt algorithm.

# Algorithms

1. Naïve Pattern Search Algorithm (Brute Force): This algorithm compares the pattern to a text such as sentence. The algorithm compares one character at a time until a match is found. An example of this algorithm in action is shown below:

|  |  |  |
| --- | --- | --- |
| Characters: TWO ROADS ARE DIVERGED IN A YELLOW WOOD  Pattern: ROADS | | Pattern Found |
| Iteration 1 | |  |
| T | WO ROADS ARE DIVERGED IN A YELLOW WOOD | FALSE |
| R | OADS |
| TW | WO ROADS ARE DIVERGED IN A YELLOW WOOD | FALSE |
| RO | ADS |
| TWO  ROA | ROADS ARE DIVERGED IN A YELLOW WOOD  DS | FALSE |
| Iteration 2 | |  |
| R  R | OADS ARE DIVERGED IN A YELLOW WOOD  OADS | TRUE |

Table 1: Implementation of Naïve Algorithm (Brute Force)

The algorithm can be designed to either be stopped on the first occurrence or upon reaching the end of the text. A pseudo-code for this algorithm is provided below:

NaïvePatternSearch(Text, Pattern, m, n)

m = length(Text)

n = length(Pattern)

for( i = 0; i < n – m + 1; i++)

j = 0

while (j < m)

if (Text[ i + j] ≠ Pattern[j])

break

j = j + 1

if (j == m)

return(i, i + j – 1)

1. Knuth-Morris-Pratt Algorithm: Another popular algorithm that is used in pattern searching is called the Knuth-Morris-Pratt (KMP) Algorithm. The KMP algorithm differs from the Naïve pattern searching algorithm (brute-force) by keeping track of information gained from previous comparisons. The algorithm utilizes a preprocessing function that indicates how much of the last comparison can be reused if it fails. The preprocessing function is defined as the longest prefix of a pattern P[0,….,j] that is also a suffix of P[1,….,j]. An example of KMP’s preprocessing function is outlined below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| j | 0 | 1 | 2 | 3 | 4 | 5 |
| P[j] | a | b | a | b | a | c |
| f(j) | 0 | 0 | 1 | 2 | 3 | 0 |

Table 2: Implementation of Knuth-Morris-Pratt Algorithm’s preprocessing function

If we look at table 2, we can see how much of the beginning of a string matches up with the portion immediately preceding a failed comparison. We notice that if a comparison fails at say j = 4 then we know that a and b are in positions 2 and 3, which is identical to positions 0 and 1. A graphical representation of KMP algorithm is shown on the next page:

A picture containing diagram

Description automatically generated

Figure 1: Graphical representation of Knuth-Morris-Pratt Algorithm

A pseudo-code for KMP algorithm is defined below:

KMP(Text, Pattern, m, n)

K = []

n = -1

K.append(n)

for(k = 0; k < length(Pattern) + 1; k++)

while(n >= 0 ^^ Pattern[n] ≠ Pattern[k - 1])

n = K[n]

n = n + 1

K.append(n)

m = 0

for(i = 0; length(Text); i++)

while(m >= 0 and Pattern[m] ≠ Text[i])

m = K[m]

m = m + 1

if m = length(Pattern)

m = K[m]

return(i – m + 1, i)

# Analysis

1. Naïve Pattern Search Algorithm (Brute Force): To perform an analysis of the best case and worst-case time complexities for the Naïve Pattern Search algorithm we looked at a simple example below:

Best Case:

Given a table below which contains a pattern of m characters in length and a text of n characters in length. In this case, the best-case time complexity is found if there is no pattern, where there is always a mismatch on the first character, for example at m = 5:

|  |  |  |
| --- | --- | --- |
| Iterations | Text: AAAAAAAAAAAAAAAAAAAAAAAAAAH  Pattern: OOOOH | Comparisons |
| 1 | *A*AAAAAAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |
| 2 | A*A*AAAAAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |
| 3 | AA*A*AAAAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |
| 4 | AAA*A*AAAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |
| 5 | AAAA*A*AAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |
| …. | .... | …. |
| N | AAAAAAAAAAAAAAAAAAAAAAAAAAAH | 1 |
| *O*OOOH |

Table 3: Best case time complexity for a given length of m characters and text of n characters in length.

The total number of comparisons made is n and so therefore the best time complexity is O(n). Similarly, if the pattern was found, then the best time complexity would be O(m) since the number of comparisons would be m.

Worst Case:

Provided a table below which contains a pattern of m characters in length and a text of n characters in length. In this case, the worst case is found when the algorithm compares the pattern to each substring of text of length m, for example at m = 5:

|  |  |  |
| --- | --- | --- |
| Iterations | Text: AAAAAAAAAAAAAAAAAAAAAAAAAAH  Pattern: AAAAH | Comparisons |
| 1 | ***AAAAA***AAAAAAAAAAAAAAAAAAAAAAH | 5 |
| AAAAH |
| 2 | *A****AAAA***AAAAAAAAAAAAAAAAAAAAAAH | 5 |
| AAAAH |
| 3 | *AA****AAAAA***AAAAAAAAAAAAAAAAAAAAH | 5 |
| AAAAH |
| 4 | AAA***AAAAA***AAAAAAAAAAAAAAAAAAAH | 5 |
| AAAAH |
| 5 | AAAA***AAAAA***AAAAAAAAAAAAAAAAAAH | 5 |
| AAAAH |
| …. | .... | …. |
| N | AAAAAAAAAAAAAAAAAAAAAAA***AAAAH*** | 5 |
| AAAAH |

Table 4: Worst case time complexity for a given length of m characters and text of n characters in length.

So, in the worst case, the total number of comparisons made is m (n – m + 1) and so therefore the worst-case time complexity is O(mn).

1. Knuth-Morris-Pratt (KMP): Looking at the algorithm for KMP we notice that at every iteration through the while loop, we note the following three things happening:
2. If we let k to be n – m then, if Text[m] = Pattern[n], then n increases by 1 and m and k remains on the same position.
3. If Text[m] ≠ Pattern[n] and n > 0, then m does not change, and k increases by 1 since k is from m – n to m – f(n-1) where f is the preprocessing function.
4. If Text[m] ≠ Pattern[n] and n = 0, then n increases by 1 and k increases by 1 since m remains the same.

Therefore, each time through the loop either m or k increase by 1, so the greatest possible number of loops is 2n. Assuming that the preprocessing function has already been defined. In our case the pre-processing function is computed in the same way as the matching function therefore the time complexity argument is comparable. The time complexity of the preprocessing function is O(m) and the string-matching function has a time complexity of O(n) therefore the total time complexity for the KMP algorithm is O (m + n), where the worst case has a time complexity of O(n) which is a whole lot better when compared to the Naïve Pattern Search (brute-force) algorithm.

# Test

In this study we were asked to test using 3 characters a, b, and c, and then randomly generate them with a string size of 1000. We were then asked to search for the string abcba. Finally, we were asked to count the number of comparisons made in the Knuth- Morris Pratt (KMP) string search, and compare it with the Naïve Pattern Search (brute-force) result of m by n. We implemented both algorithms in Python.

## Results

After implementing the Naïve Pattern Search and Knuth-Morris-Pratt algorithms we obtained the following results:

Text, table

Description automatically generated

Figure 2: Results of Knuth-Morris-Pratt and Naïve Pattern Search algorithms

If we compare the results obtained using Knuth-Morris-Pratt and Naïve Pattern Search we notice that the number of comparisons made by KMP are identical to the Naïve Pattern Search, however, the time taken to perform Naïve Pattern Search is higher than the time taken to perform KMP search thus proving that the KMP algorithm is more efficient when compared to Naïve Pattern Search.

## References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf>

<https://www.codesdope.com/blog/article/kmp-algorithm/>

<https://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/kuthMP.htm>

<https://www.tutorialspoint.com/Knuth-Morris-Pratt-Algorithm>

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/