BOWIE STATE UNIVERSITY
1865

Syed Ali
Johnathan Ngaya

COSC 528 Design and Analysis of Algorithms

Project Title: String Matching Algorithms

Assignment Due: November 30th, 2021

Faculty: Dr. James Gil De Lamadrid

Contents

- Introduction
- Algorithm
- Analysis
- Test
- Results
- References

Introduction

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 as either 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, bioinformatics and DNA sequencing, search engines or content search in a large database, and among others.
- In this presentation we will focus on two string matching algorithms which are Naïve algorithm for pattern searching (Brute-Force) and Knuth-Morris-Pratt algorithm.

BOWIE STATE UNIVERSITY 1865

Algorithm

Algorithm

- 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.
- 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].

Algorithm

- Naïve Pattern Search Algorithm (Brute Force):

```
NaivePatternSearch(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)
```

Algorithm

- Knuth-Morris-Pratt Algorithm:

```
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

Analysis

- 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: AAAAAAAAAAAAAAAAAAAAAAAAAH Pattern: OOOOH	Comparisons
1	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1
2	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1
3	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1
4	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1
5	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1
….	….	….
N	AAAAAAAAAAAAAAAAAAAAAAAAAH OOOOH	1

- 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.

Analysis

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: AAAAAAAAAAAAAAAAAAAAAAAAAH Pattern: AAAAH	Comparisons
1	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5
2	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5
3	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5
4	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5
5	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5
….	….	….
N	AAAAAAAAAAAAAAAAAAAAAAAAAH AAAAH	5

- 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).

Analysis

- 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:

1. 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.
2. 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.
3. 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). 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

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

Results

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

```
KMP Algorithm Results
Pattern found at index: 274 278
Pattern found at index: 400 404
Pattern found at index: 614 618
Pattern found at index: 664 668
Pattern found at index: 775 779
Time taken to perform Knuth-Morris-Pratt Search: 0.0009691715240478516


Brute-Force Algorithm Results
Pattern found at index:  274 278
Pattern found at index:  400 404
Pattern found at index:  614 618
Pattern found at index:  664 668
Pattern found at index:  775 779
Time taken to perform Naive Pattern Search: 0.0009963512420654297
```

- 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

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/