CS 333 – Algorithm Analysis

Spring 2020

Project Report

Group members: Ahmet Erdem Gonul, Taci Ata Kucukpinar, Alp Bolukbasi, Ilhami Berker Gurcay

Title: Knuth Morris Pratt - Fast Pattern Matching In Strings

Abstract: Knuth Morris Pratt Algorithm is useful for finding a specific pattern in a given string. A pattern’s occurrences can be found in many different indexes. Ordinary string matching algorithms try every single condition to find out occurrences. On the other hand, Knuth Morris Pratt Algorithm skips some unnecessary operations.Thus, it has less time complexity and more efficiency.

1. Introduction

In computing applications, pattern searching algorithms are used to fetch a specific data and its index from several data structures. Searching algorithms are generally used in real world applications like Spell Checkers, Spam Filters, Intrusion Detection System, Search Engines, Plagiarism Detection and in Bioinformatics.

String matching is the classical problem of finding all occurrences of a pattern in a text. A real-time string matching algorithm takes worst-case constant-time to check if a pattern occurrence ends at each text location.

Generally, the most common way of searching a pattern is trying to find the initial element of the desired pattern and its afterwards elements. But this is an inefficient approach, especially when we are searching an occurrence such as "ccccccd" in "ccccccccccccccccccd". When the pattern is c'd and the text is c''d, we have to make lots of operations that are unnecessary compared to our algorithm. In this situation for brute force string matching algorithm, worst case time complexity becomes the multiplication of the length of the searched pattern and length of the given text.

KMP is a faster and more efficient approach to search a pattern and its occurrences when you compare it with many searching algorithms. This algorithm checks the text from left to right and when a matched pattern is found in the text , this algorithm uses that match to improve time complexity even in the worst case.

Approach of the algorithm is relatively simple, it can be imagined as a pattern is placed at the top of the beginning of the text. It compares the pattern with text and whenever there is mismatch, pattern slides to the left. If there is a match but it is up until some character then the pattern slides again but this time it slides right to the where the mismatch occurred.

KMP algorithm is used in real world applications where pattern matching is done in longer strings. A relevant example is the DNA alphabet, which has only 4 symbols (A,C,G,T). Imagine how KMP can work in a "DNA pattern matching problem": it is really suitable because many repetition of the same character allows many skips, and so less computation time complexity in algorithms.

Another algorithm we used to compare with the KMP pattern match algorithm, is the Boyer Moore pattern searching algorithm. Boyer Moore algorithm, unlike the KMP, starts matching the pattern from the last character of the pattern. There are two heuristics for this algorithm. We will use bad character heuristic. The approach is the last character of the pattern is matched and from left to right we look for mismatch. For the first mismatch, there are two cases.First case is the mismatched character in the text is in the pattern. Then we shift the pattern to match two characters and do the same operations whenever mismatch occurs. For the second case, if a mismatched character isn't in the pattern then we will shift the pattern until the pattern passes that character. The process continues to run until the pattern is matched.

In ~~the~~ sections 2 and 3, Knuth Morris Pratt and Boyer Moore algorithms are presented in detail and the implementations of the algorithms are described.

In the discussion part, analysis and runtime of the algorithm will be discussed with some calculations.

2. KMP Algorithm Overview

Knuth Morris Pratt Algorithm takes a string and a specific pattern as input and returns all occurrences of the pattern in the given string. It is a linear time algorithm that observes that every time a match or a mismatch happens, the pattern itself gets information to dictate where the new search index should begin from.

Mainly, the algorithm has two parts which are prefix function and precomputation function.Prefix function tells us what to do if a match occurs. This function is skipping some matched indexes which the iterating and checking is unnecessary. Prefix function shows us where the next matching index should begin from. Time complexity of the Prefix function is O(n) ~~which~~ n is the length of the text.

Precomputation function finds the occurrences of the pattern in the string. It uses the pre-computed prefix table to show us a new match can include previous matched characters.

Time Complexity of this function is O(m). In total, it takes O(m + n) time complexity which is more efficient than other popular pattern matching algorithms.

Matching the Pattern

Sample text = "**AAAA**ABAAABA"
Desired pattern = "**AAAA**"

We compare first characters of text with the pattern,

Text = "**AAAA**ABAAABA"
Pattern = "**AAAA**"

We found a match. This is the same as the Brute Force Algorithm. In the next step, we compare ~~afterwards~~ characters of the text with patterns.

Text = "A**AAAA**BAAABA"
Pattern = “**AAAA**"

This is the advantage of KMP over Brute Force Algorithm. In the second situation, we only compare the fourth A of the pattern with the fourth character of the text's current index to decide if the current Index matches or not. KMP already places the first three characters that match anyway, we skipped matching the first three characters. A pre-processing is necessary to know that. Preparing an integer array processes the count of skipped characters.

Preprocessing the Pattern

KMP algorithm preprocesses pattern and constructs an extraneous integer array of same size of pattern for skipping characters. This array shows the longest proper suffix which is prefix without the whole string shown. For a string “XYZ”, proper prefixes are “”, “X” and “XY”; proper suffixes are “”, “Z”, “YZ”.

We search for the longest proper suffix in sub-patterns. Obviously, we have to check the sub-strings of patterns that may be either prefix or suffix.

For each sub-pattern from 0 to i where i = 0 to the last element, longest proper suffix stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern 0 to i

Example:

Desired pattern = “XXXX”
For the text “XXXX”, integer array is [0, 1, 2, 3]
For the text “XYZTU”, integer array is [0, 0, 0, 0, 0]
For the pattern “XXYXXZXXYXX”, integer array is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
For the pattern “XXXZXXXXXZ”, integer array is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]
For the pattern “XXXYXXX”, integer array is [0, 1, 2, 0, 1, 2, 3]

Implementation Overview

In the implementation of the algorithm, there are 2 main functions called kmp and kmpAlgorithms. In the kmp function, a prefix table for the given string is created and returned to the main function.

The “kmp” function takes a parameter as “inputPattern” and this is the pattern we are searching in the text. Given string is assigned to the “pattern” variable and the length of this string is assigned to “lengthOfPattern”. An Array called “prefixTable” is created with the size of “lengthOfPattern”. Initially the first index of the array is set to 0 and “longestPrefix” is initially without any calculation. Then a “for loop” is iterating over the array. In each index of the pattern , the character at the specified index is getting checked with the character at the “longesPrefix” index. If it returns true, “longestPrefix” is incremented one and “longestPrefix” is assigned to “currentIndex” of “prefixTable” . If it returns false , the function goes into another condition. If “longestPrefix” is not equal to 0, prefixTable[longestPrefix-1] is assigned to longestPrefix and currentIndex is decremented one. If “longestPrefix” is equal to zero, prefixTable[currentIndex] becomes zero. At the end, for loops ends and the function returns prefixTable.

The “kmpAlgorithm” function takes two parameter as “text” and “pattern”. These are the pattern we are searching in the text and the given string which the pattern will be searched inside. Given string is assigned to the “pattern” variable and the length of this string is assigned to “lengthOfPattern”. The length of the text is assigned to “lengthOfPattern”. An Array called “prefixTable” is created with the return of the “kmp” function. An Array called “prefixTable” is created with the size of “text” length. For the while loop iterations, i , j and location variables initialized. While loop execution condition is the iterator “i” should be smaller than lengthOfText and if this condition is not satisfied while loop will terminate. If the “i” index of the string is equal to the “j” index of the pattern “i” and “j” is incremented one. After that if condition , new if condition starts. If “j” is equal to “lengthOfPattern” “i-j” is assigned to locationArray[location],location is incremented one and prefixArray[j-1] is assigned to “j”. “Else if” condition checks if i is smaller than “lengthOfText” and character at the “j” index of “pattern” is not equal to character at “i” index of “text”. Then inside the if new if starts and checks if j is not equal to zero. If the condition is satisfied, prefixArray[j-1] is assigned to “j”. Else, i is incremented one. After the “while” loop, locationArray is returned.

3. Boyer Moore Algorithm Overview

Just like the KMP algorithm Booyer Moore algorithm takes a text and a specific pattern to match as an input and returns the occurrence of the pattern in the given text.

Similar to KMP in this algorithm there is also the concept of sliding to pattern.However, in Booyer Moore algorithm the first character to be matched is the last character of the pattern.

There are two heuristics of Boyer Moore algorithm which are "Bad Character" and "Good Suffix". We will use bad character heuristic to compare with KMP algorithm

Bad Character is when the character of the text doesn't match with the current character of the pattern. When mismatch occurs, the pattern is shifted. However, there are two ways how it is shifted. In the first case, if the character of the text that mismatch occurred is in pattern then we shift the pattern until two of them match. In the second case, the mismatched character is not in the pattern so we skip it until its next character.

G A T T C G A G A C A C T G

G A C A C

T is where first mismatch occurs, and since T is not in the pattern, we shift the pattern until T is skipped.

G A T T C G A G A C A C T G

G A C A C

Now, A is the first character that is mismatched since A is in pattern we shifted until A is matched.

G A T T C G A G A C A C T G

G A C A C

Now, G is first one to mismatch so we shift the pattern until G is matched with G in the pattern.

G A T T C G A G A C A C T G

G A C A C

We find the pattern.

In future work we will implement Java code of Boyer Moore algorithm and compare both algorithms' efficiencies.

4. Discussion

In future work we will implement Java code of Boyer Moore algorithm and compare both algorithms' efficiencies.

5. Conclusion and Future Work

In future work we will implement Java code of Boyer Moore algorithm and compare both algorithms' efficiencies. Also we want to implement an brute force search algorithm (naive algorithm) to calculate the performance differences on lots of test cases. We will also convert this paper into the LaTeX in the final report.

6. References

- Boyer, Robert S., and J. Strother Moore. "A Fast String Searching Algorithm." *Communications of the ACM,* 1977.
- "KMP Algorithm for Pattern Searching." *GeeksforGeeks*, 20 May 2019, www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/.
- Knuth, Donald E., et al. "Fast Pattern Matching in Strings." *SIAM Journal on Computing*, 1977.

Appendix

```
public class Main {
  public static int[] kmp(String inputPattern){
    String pattern=inputPattern;
    int lengthOfPattern=pattern.length();
    int[] prefixTable=new int[lengthOfPattern];
    prefixTable[0]=0;
    int longestPrefix=0;
    for (int currentIndex = 1; currentIndex < lengthOfPattern; currentIndex++) {
        if(pattern.charAt(currentIndex)==pattern.charAt(longestPrefix)){
            longestPrefix++;
            prefixTable[currentIndex]=longestPrefix;
        }else{
            if(longestPrefix!=0){
                longestPrefix=prefixTable[longestPrefix-1];
                currentIndex--;
            }else
                prefixTable[currentIndex]=0;
        }
    }
    return prefixTable;
}
public static int[]  kmpAlgorithm(String text,String pattern){
    int lengthOfText=text.length();
    int lengthOfPattern=pattern.length();
    int[] prefixArray=kmp(pattern);
    int[] locationArray=new int[text.length()];
    int i=0,j=0,location=0;
    while(i<lengthOfText){
        if(text.charAt(i)==pattern.charAt(j)){
            i++;
            j++;
        }
        if(j==lengthOfPattern){
            locationArray[location]=i-j;
            location++;
            j=prefixArray[j-1];
        }else if(i<lengthOfText && pattern.charAt(j)!=text.charAt(i)){
            if(j!=0){
                j=prefixArray[j-1];
            }else
                i++;
        }
    }
    return locationArray;
}
public static void main(String[] args) {
    int[] result=Main.kmpAlgorithm("AAAABAAAAABBBAAAAB","AAAB");
    for (int i:result
         ) {
        if(i!=0)
        System.out.println("Pattern found at : " + i + " ");
    }
}}
```