SPRING 2016

Final Report

on

String Matching: Plagiarism detection

CSE – 5311 (Design and Analysis of Algorithms)

By:

Tasnim Makada (1001288916)

Deepak Verma (1001288915)

Chinthan Bhat (1001267683)

Anurag Uplanchiwar (1001231698)

INDEX

INTRODUCTION

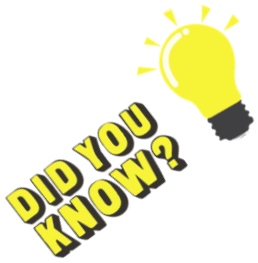
WHAT IS PLAGIARISM?

According to Wikipedia, plagiarism is the "wrongful appropriation" and "stealing and publication" of another author's "language, thoughts, ideas, or expressions" and the representation of them as one's own original work. But it can also be unintentional, using another author's language or work without citing them is also known as plagiarism. A detailed explanation of what plagiarism contains can be found below.

ALL OF THE FOLLOWING ARE CONSIDERED PLAGIARISM:

- turning in someone else's work as your own
- copying words or ideas from someone else without giving credit
- failing to put a quotation in quotation marks
- giving incorrect information about the source of a quotation
- changing words but copying the sentence structure of a source without giving credit
- copying so many words or ideas from a source that it makes up the majority of your work, whether you give credit or not.

With increasing amount of data that is coming on the internet, and journals on a daily basis, it becomes more and more difficult to check for plagiarism without tools. We have created a framework with multiple algorithms implemented to check for plagiarism in the document that is provided. Such tools make checking for plagiarism easier and more feasible. The mentioned framework is described further on in the report.



<u>The First "Plagiarists"</u>

The roman poet Martial lived from 40 AD to somewhere between 102 and 104 AD. Though he wasn't an immediate star, saying that he didn't care for many of his earlier works, by about 80 AD he started to enjoy success, which continued until much later in his life. However, Martial, like many poets in the era, found that his work was being copied and recited wholesale by other poets without attribution. This was a very common act during the time as poets, generally, were more expected to be able to recite and relay earlier works than create original ones.

But Martial wasn't content to stand by and let others take his work. Without copyright law or any legal recourse, he used the tool he had available to him, his words. He wrote several verses aimed at copycats, including a quip from an alleged plagiarist, Fidentinus.

FRAMEWORK DESCRIPTION

The framework created to check plagiarism takes 3 inputs, an algorithm option, a test file and a corpus of existing documents against which the test file will be checked for plagiarism. All the files are in text format. The first input, indicates which algorithm is used to match the files for plagiarism. Option 1 is for LCSS, 2 is for Naive string matching, 3 is for KMP, and 4 is for Boyer-Moore algorithms. Upon selecting an option, the respective algorithm is used for checking plagiarism and the result is accordingly displayed on the console.

There are two categories of algorithms that are implemented, longest common subsequence and substring matching. For finding the longest common subsequence (LCSS) dynamic programming approach is used. For substring matching, 3 algorithms are implemented, namely Naive, KMP and Boyer-Moore algorithms. These algorithms are then compared and contrasted for performance under the worst-case of substring matching.

For LCSS, the framework breaks the test file and existing documents into paragraphs. Each paragraph from the test file is then compared with each paragraph in the existing documents. If more than 50% of the paragraphs are found plagiarized from the existing documents, the test file is then marked as plagiarized and a message stating the same is displayed on the console. Also for each execution, for each plagiarized paragraph, the substring and existing document paragraph are displayed.

For naive substring matching, KMP and Boyer-Moore algorithms the execution style was a little different. For these algorithms, the test file and existing documents are broken into sentences based on "." regex. Each line from the test file is then checked with every line from the existing document using the algorithm that is given in the option. Every matched sentence is then counted, and it the number of lines in the test file that are plagiarized are more than 51% than significant plagiarism is displayed on the console. For these algorithms as well, all the matching sentences are displayed on the console.

MAIN FUNCTION

The main function is a simple function that takes the user input from the command line and parses it. The first input is the type of algorithm to use, based on this input, the respective function for each algorithm is executed. If the input option is 1, i.e. LCSS, the documents are divided into paragraphs and sent to the respective algorithm to check for plagiarism. For other inputs, i.e., 2, 3 and 4 the documents are divided into sentences using "." as the regex. Each sentence is then passed to the respective algorithm to check for plagiarism. The final plagiarism percentage is then obtained from each algorithm function which is displayed on the console.

LCSS - LONGEST COMMON SUBSEQUENCE

Unlike substring, a subsequence does not require characters to occupy consecutive positions. Finding the longest common subsequence (LCSS) for arbitrary number of sentences, the problem is NP-Hard. But for constant number of sentences, it can be performed in polynomial time using Dynamic Programming.

For 2 subsequences of length n and m the running time using Dynamic Programming is O(nm). Space complexity: O(nm)

The algorithm implemented for this project is also implemented using dynamic programming because it is one of the fastest ways of obtaining a subsequence. A 2D array was created of the dimension n x m, where n is the length of the string and m is the length of the pattern being matched. Once this was created, each character from the string is matched with the pattern and if they match, the value in the respective matrix block is incremented. The second part of the algorithm is obtaining the subsequence based on matrix, which is obtained in a reverse manner. Hence, the obtained string is reversed and then returned. It is then checked if the length of the returned subsequence is more than 51% of the length of the actual string. This ensures that matching of simple terms as “a”, “the”, “is” etc. are not considered as plagiarism. If the length of the subsequence is more than 51% of the actual string, the paragraph is marked as plagiarized.

The running time analysis of the lcss algorithm is displayed in figure 1 below:

```
public static String lcss(String pattern, String checkString) {
      int[][] lengths = new int[pattern.length()+1][checkString.length()+1];---O(1)
      StringBuffer buffer = new StringBuffer();-------------------------------O(1)

      for (int i=0; i<pattern.length(); i++)----------------------------------O(n)
        for (int j=0; j<checkString.length(); j++)----------------------------O(m)
            if (pattern.charAt(i) == checkString.charAt(j))-------------------O(1)
               lengths[i+1][j+1] = lengths[i][j] + 1;--------------------------O(1)
            else
               lengths[i+1][j+1] = Math.max(lengths[i+1][j], lengths[i][j+1]);-O(1)

      // Get the common subsequence
      for (int x=pattern.length(),y=checkString.length();((x!=0) && (y!=0));){O(nm)
        if (lengths[x][y] == lengths[x-1][y])-----------------------------------O(1)
           x--; ---------------------------------------------------------------O(1)
        else if (lengths[x][y] == lengths[x][y-1])------------------------------O(1)
           y--; ---------------------------------------------------------------O(1)
        else {
           if(pattern.charAt(x-1) == checkString.charAt(y-1)){ ----------------O(1)
              buffer.append(pattern.charAt(x-1)); ------------------------------O(1)
              x-=1;
              y-=1;
           }
       }
     }
     return buffer.reverse().toString(); ---------------------------------------O(n)
}
```

Figure 1. Analysis of LCSS algorithm

As mentioned earlier as well, the running time of the algorithm is O(*mn*) and the space complexity is also O(*nm*).

NAIVE ALGORITHM FOR SUBSTRING MATCHING

The most obvious solution to string matching is matching each character of a string to the other string, and move ahead by only one character when a mismatch is encountered. But this approach, though most intuitive is the slowest in terms of running time. Though for smaller inputs this algorithm will perform better than other algorithms because it does not need any pre-processing and no space is used for the same.

As explained above briefly, the naïve algorithm matches each character of the pattern to the actual string and moves ahead in matching the pattern once a match is found. If a mismatch occurs, it does not use any previous matching information to jump positions, but rather just moves by 1 character starting the whole process again.

Pseudo-code its analysis:

For loop to slide pattern one by one for at most n-m+1 times
-n-m+1 times

For current index i, check for pattern match for m
-m times

if (txt[i+j] != pat[j])

We can visual it as a sliding pattern of P[1..m] over the text T[1..n] taking O((n - m +1)m), which is clearly O(nm) in worst case. Worst case also occurs when only the last character is different or all characters are same.

Text ="AAAAAAAAAB" Pattern = "AAAAB"

Text ="AAAAAAAAA" Pattern = "AAAA"

KNUTH-MORRIS-PRATT (KMP) ALGORITHM

The Knuth–Morris–Pratt string searching algorithm (or KMP algorithm) consists of a string and a pattern and searches for occurrences of the pattern within the string by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The algorithm was conceived in 1970 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977.

This algorithm is best suited in scenarios where 'pattern to be searched' remains same whereas 'text to be searched' changes. The implementation of this algorithm is a simple one and hence runs much faster than naive algorithm which runs in O(mn) time. KMP runs in O(m+n) time.

The algorithm works by creating a Prefix Set in which for each position of i in pattern P, b[i] is defined such that it takes the 'length of the longest proper suffix of P[1…i-1]' that matches with the 'prefix of P'.

To calculate the running time of this algorithm we need to consider two parts. The pattern-matching part and the Prefix-set processing part. The pattern matching part takes O(n) and Prefix Set processing takes O(m). Hence running time is O(m+n). The figure below gives the running time analysis of each step present in the algorithm. An array is used to store the string, pattern and to calculate the prefix-set.

```
Pattern Matching part:

        while (i < textLen) {                                   -O(n)
            while (j >= 0 && text[i] != pattern[j])             -O(m)
                j = b[j];                                       -O(1)
            i++;                                                -O(1)
            j++;                                                -O(1)
            if (j == patternLen) {
                j = b[j];                                       -O(1)
                return true;
            }
        }
            return false;

PreProcessing PrefixSet part:

            while (i < patternLen) {                            -O(m)
                  while (j >= 0 && pattern[i] != pattern[j])    -O(m)
                     j = b[j];                                  -O(1)
                i++;                                            -O(1)
                j++;                                            -O(1)
                b[i] = j;                                       -O(1)
            }
            return b;                                           -O(1)
      }
```

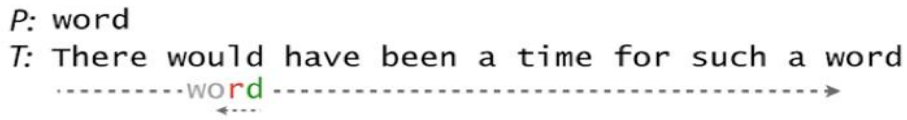
BOYER-MOORE ALGORITHM

Boyer–Moore string search algorithm is an efficient string searching algorithm that is the standard benchmark for practical string search literature. It was developed by Robert S. Boyer and J Strother Moore in 1977. The algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. The Boyer-Moore algorithm uses information gathered during the preprocess step to skip sections of the text, resulting in a lower constant factor than many other string search algorithms. In general, the algorithm runs faster as the pattern length increases. The key features of the algorithm are to match on the tail of the pattern rather than the head, and to skip along the text in jumps of multiple characters rather than searching every single character in the text.

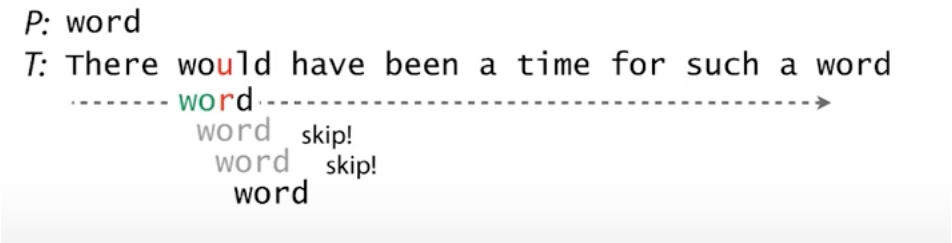
The way Boyer Moore Algorithm works is as follows:

•Boyer-Moore skips alignment when there is a mismatch between pattern and text.

For example,



Here unlike naïve algorithm, on a mismatch the number of comparison's skipped is not just 1 but greater value depending on the rules of Boyer Moore algorithm.



•To determine the number of alignments to skip it uses two rules:

1. Bad Character Rule: Upon mismatch skip alignments until:

    a. Mismatch becomes a match or

    b. Pattern moves past the mismatched character.

For example,

Here, for mismatch on character C of text, this algorithm checks if there is another C in pattern on the left of mismatched character. If so, it shifts pattern such that this C is aligned with C of

text, else it makes pattern pass across the mismatched character.

2. Good Suffix Rule:

If 't' is the length of substring matched before a mismatch, then skip until:

a. There are no mismatches between P and t or

b. Pattern moves past t.

For example,

Here, for mismatch on character C; the substring matched so far is 'TAC'. Algorithm search for this substring in of pattern. If found it aligns it with this substring of text, else skips the pattern across the mismatched character.

Implementation:

This algorithm was implemented using arrays as data structure. It creates two tables as part of preprocessing of pattern: character table & offset table. Character table is used for bad character rule, which stores the rightmost position of each character in pattern. Offset table is used for good suffix rule, i.e. to find positions of suffix in pattern which is also a prefix.

The comparison of pattern and text is made using two nested for loops. The counter value of outer for loop is incremented by taking maximum of values suggested by bad character rule and good suffix rule.

```
for(i=n-1; i<m;)
{

    for(j=n-1;pattern[j]==text[i]&&j>=0;j--,i--)
    {
        if(j==0)
            return true;
    }

        skip=Math.max(preProcOffsetTable[(n-1) - j], preProcCharTable[text[i]]);
    }
    i+=skip;
}
```

The time complexity of this algorithm turns out to be O(n+m) where n is the length of text and m is the length of pattern. Although this is a nested for loop, the counter of outer for loop is incremented by huge values using rules of boyer moore algorithm, which reduces the time complexity further.

COMPARISON OF ALL SUBSTRING MATCHING ALGORITHMS

The time-complexity for each of the algorithms above has been summarized in table 1 below for reference.

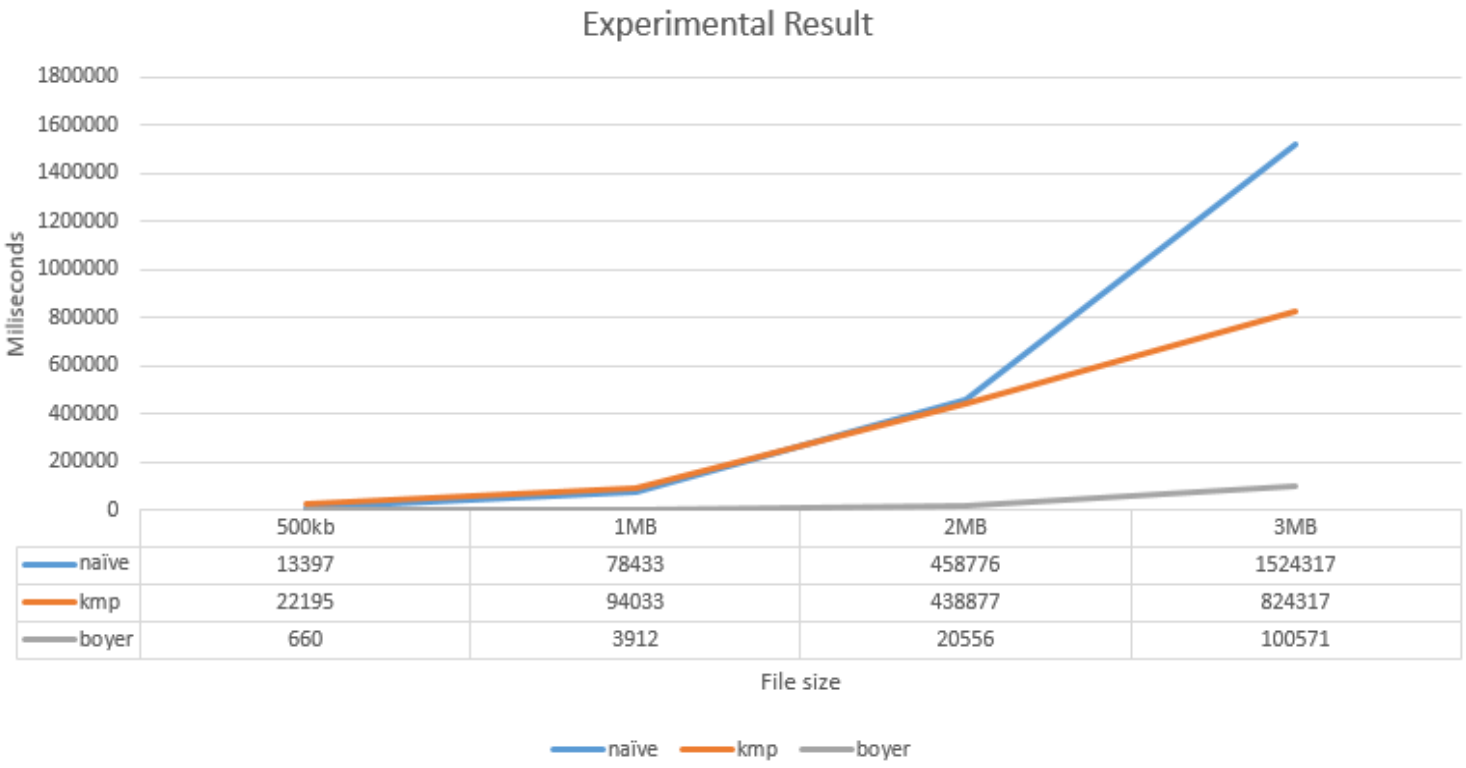
Algorithm	Preprocessing time	Matching time
Longest Common subsequence	0	Θ(nm)
Naïve string search	0	Θ(nm)
Knuth–Morris–Pratt	Θ(m)	Θ(n)
Boyer Moore	Θ(m)	Θ(nm)

Table 1. Time complexity for all the algorithms

EXPERIMENT

Test files with the worst possible test case was considered for testing. This test file consisted of sentences which matched the entire corresponding sentence in the existing files except for the last character on the sentence. File sizes of 500KB, 1MB, 2MB and 3MB were used for testing Naive search, KMP and Boyer Moore algorithms. Smaller files were chosen for LCSS, since its execution time and space requirements are higher which affected its execution time. We chose file sizes of 10KB, 25KB, 50KB, 100KB, 250KB and 500KB for LCSS.

The results were recorded and a graph was produced based on them. We observed that Boyer Moore was always the fastest algorithm for every case present. Initially Naive search was faster than KMP but as the file size got bigger KMP started giving results faster than Naive(2MB files onwards). These results can be clearly seen on the graph below. The running time of Boyer Moore is the fastest among the four and then KMP and then Naive search. Figure 2 shows the graph for naive, KMP and Boyer-Moore. LCSS is recorded in another graph, since it checks for subsequence and the rest of the algorithms check for substring. Figure 3 shows the results for LCSS.



	500kb	1MB	2MB	3MB
naïve	13397	78433	458776	1524317
kmp	22195	94033	438877	824317
boyer	660	3912	20556	100571

Figure 2. Running time for Naive, KMP and Boyer-Moore algorithm for worst case with different size input files

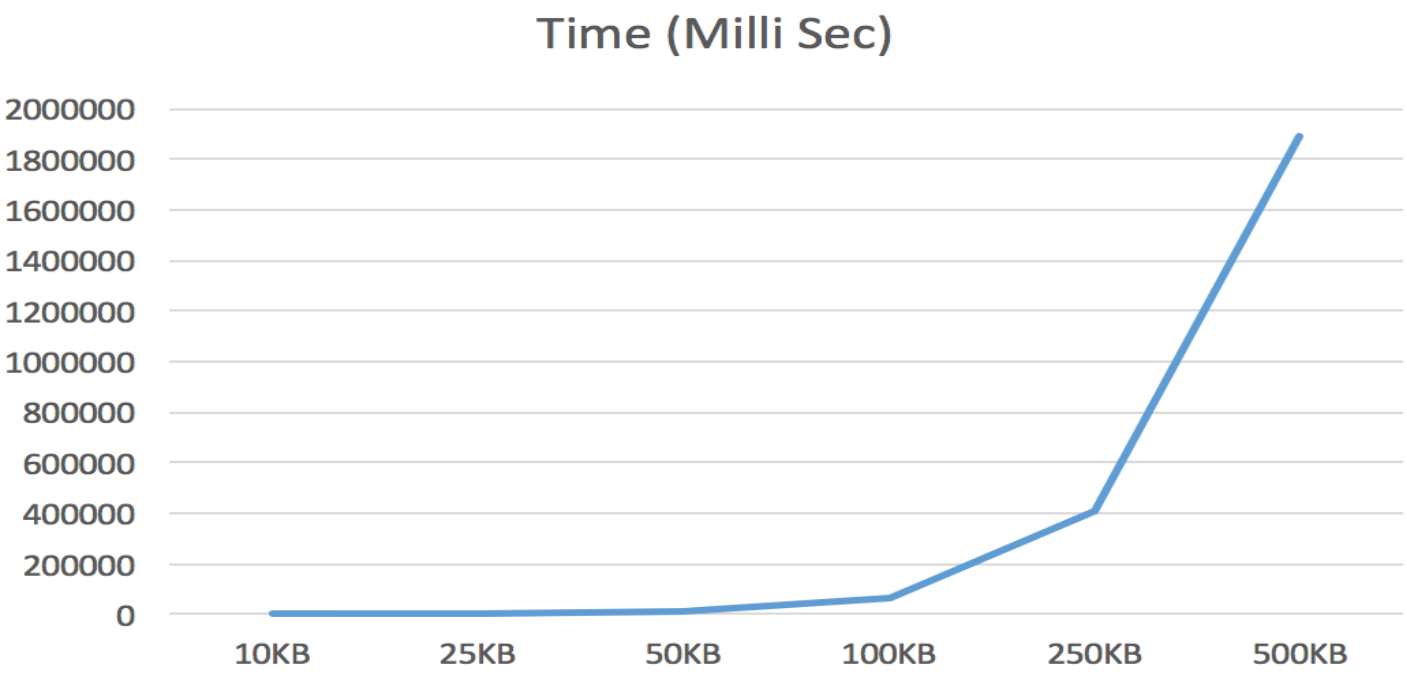


Figure 3. Running time for LCSS for worst case with different size input files

CONCLUSION

The implemented algorithms and their running time analysis is verified with the experiment and the graph produced by it. It can be clearly seen that the Naive algorithm executes faster than KMP for smaller inputs, but as the input time increases, KMP and Boyer-Moore algorithm out-perform Naive algorithm, which makes them a better choice for finding substrings. Their running time analysis shows the same.

LCSS is used for finding subsequence and can be used efficiently for plagiarism as well, but the system requirements in terms of space and time greatly affect its execution time. But for purposes of small to medium sized documents, LCSS would work well for finding plagiarism.

REFERENCES

- https://tekmarathon.com/2013/05/14/algorithm-to-find-substring-in-a-string-kmp-algorithm/
- https://www.youtube.com/watch?v=4Xyhb72LCX4
- https://en.wikipedia.org/wiki/Boyer_moore
- https://en.wikipedia.org/wiki/String_searching_algorithm
- http://www.plagiarism.org/plagiarism-101/what-is-plagiarism/
- https://en.wikipedia.org/wiki/Plagiarism