Project Report
CS 575 Fall 2018 Report

Plagiarism Detection using String Matching Algorithm

An approach using Longest Common Substring and Rabin Karp algorithm

By:
Antra Aruj
Akshi Sharma
CS-575-01

Table of Contents

Objective:

In today word copying something from other sources and claiming it as an own contribution is a crime. We have also seen it is major problem in academic where students of UG, PG or even at PhD level copying some part of original documents and publishing on own name without taking proper permission from author or developer. Many software tools in exist to find out and assist the monotonous and time consuming task of tracing plagiarism, because identifying the owner of that whole text is practically difficult and impossible for markers.The main objective of the project on development of plagiarism detection tool is to explore the idea of string/pattern matching. In computer science, string-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. In practice, how the string is encoded can affect the feasible string-search algorithms. In particular, if a variable-width encoding is in use then it may be slower to find the Nth character (perhaps requiring time proportional to N). One of many possible solutions is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed to avoid it.

The most basic case of string searching involves one (often very long) string, sometimes called the "haystack", and one (often very short) string, sometimes called the "needle". The goal is to find one or more occurrences of the "needle" within the "haystack". The basic idea of string matching is to compare the characters of the string in every sentence for an input file while comparing it with the existing files to show the level of similarity the documents may have. The idea of the string matching problem is that we want to find all occurrences of the pattern P in the given text T.

Basic classification of search algorithms:

The various algorithms can be classified by the number of patterns each uses.

1.Single-pattern algorithms

- Naïve string-search algorithm
- **Rabin–Karp algorithm**
- Knuth–Morris–Pratt algorithm
- Boyer–Moore string-search algorithm
- **Longest Common Substring algorithm**

2. Algorithms using a finite set of patterns

- Aho–Corasick string matching algorithm (extension of Knuth-Morris-Pratt)
- Commentz-Walter algorithm (extension of Boyer-Moore)
- Set-BOM (extension of Backward Oracle Matching)
- Rabin–Karp string search algorithm

Problem Statement:

In this project we will be focussing on two algorithms i.e. Longest Common Substrings and Rabin Karp Algorithm. The main focus is on algorithm development and comparison of time complexity to understand the area of usage of each algorithm undertaken.

Algorithm Description

Rabin Karp Algorithm Approach:

A string search algorithm which compares a string's hash values, rather than the strings themselves. We are taking a help of Karp-Rabin Algorithm. It uses fingerprints to find occurrences of one string into another string. Karp-Rabin Algorithm reduces time of comparison of two sequences by assigning hash value to each string and word. Without hash value, it takes too much time for comparison like if there is a word W and input string is S then word is compared with every string and sub string in program and hence it consumes more time. Karp-Rabin has introduced concept of Hash value to avoid time complexity O(m2). It assigns hash value by calculating to both word and string/substring. So hash of substring (S) matches with hash value of W then only we can say exact comparison is done.

Hash Values:

A hash value is a numeric value of a fixed length that uniquely identifies data. The most wonderful character of Hash Values is that they are highly unique. No two data can theoretically have same Hash Value.

Karp-Rabin algorithm preferred category from left to right comparison. Function of hash must able to find has value efficiently. When first time name would be hashing with the same hash it save the data causing yields a value which will be compared to at data is index with the value. It can deal with multiple pattern matching that's why people preferred this Karp-Rabin algorithm. Otherwise behavior of other algorithm is to perform basic pattern matching. Its having O(nm) complexity. Where n is length of text and m is length of pattern. It is little bit slow also due to we have to check every single character from the text. The steps followed for algorithm development are as follows:

- We have taken String s and an input file and the patterns taken are each sentence separated by delimiter full stop in String s.
- For each sentence we checked if the sentence is matched with any of the lines of input files.
- At last we have kept two counters :- m for total number of sentences in the string s and n for total number of matched sentences of string s.

```
for (int x = 0; x <= len2 - len1; x++)
{
    if ( hash_1 == hash_2 )
    {
        for (y = 0; y < len1; y++)
        {
            if (seq[y] != arr[x+y])
                break;
        }

        if (y == len1)
            {
                l++;
                cout << "Sentence " <<sent_line << " of the input text found at line " << line << " of output file"<<std::endl;
                break;
            }

    }

    else if ( x < len2-len1 )
    {
        int w = d* (hash_2 - arr[x]*h);
        hash_2 = (w + arr[x+len1])%m;

        if (hash_2 < 0)
        {
         hash_2 = (hash_2 + m);
        }

    }
}
return 1;
```

LCS Approach:

The LCS problem is the problem of finding the longest subsequence common to all sequences in a set of sequences. It differs from the longest common substring problem, unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. A subsequence is a sequence that appears in the same relative order, but not necessarily one after another in a contiguous manner.

LCS Algorithm Description:

The plagiarism checker implementation based on LCS can be understood as an extension of the original LCS algorithm. Therefore, we have implemented the LCS algorithm taught in the dynamic programming section of the class to accommodate the larger strings. In principle, we read the input file and parse it as a paragraph that need to be checked for plagiarism. Simultaneously we read a database of texts paragraph by paragraph and each paragraph is considered as a database file. In other words, the user can provide a single database file with multiple paragraphs and the code will consider each paragraph as one database file. While the database file is being read paragraph by paragraph we call an lcs_Check function to report the percentage of commonalities between input and current database paragraph.

```
// Read the input file
fgets(test_content, BUFFER_SIZE, testFP);
// Read the database file paragraph by paragraph
int size = 0;
while (fgets(source_content, BUFFER_SIZE, srcFP)) {
   printf("%s", source_content);
   // Call the lcs algorithm to check percentage of commonalities
   lcs_Check(source_content, test_content, resultFP);
}
```

The key components of the plagiarism checker are implemented inside the LCS function. Firstly, we create and initialize the 2D matrix to store the length of LCS and its relation with strings.

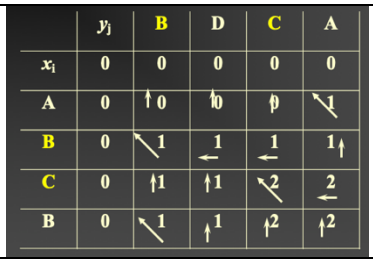
```
//step 1: Calculate their sizes and use them to allocate 2 dimensional matrix
int src_len = strlen(source_content);
int test_len = strlen(test_content);
//step 2: If size of any of the strings is zero report LCS = 0
if (src_len == 0 || test_len == 0) {
   printf("Size of LCS:: 0\n");
   return 0;
}
//step 3: Initialize the two i and j rows/column of the 2D matrix
int matrix[src_len + 1][test_len + 1]
for (int i = 0; i <= src_len; i++) {
   for (int j = 0; j <= test_len; j++) {
      matrix[i][j] = 0;
   }
}
```

Next, we apply the dynamic programming version of LCS to populate the matrix. We follow the principal that if there is a match between both strings then bring the element from diagonal and add one to it. If elements of string do not match then copy maximum of top or left element of the matrix to the current position. The working of the implementation is shown in the figure below.

```
for (int i = 1; i <= src_len; i++) {
    for (int j = 1; j <= test_len; j++) {
        if (source_content[i - 1] == test_content[j - 1]) {
            matrix[i][j] = matrix[i - 1][j - 1] + 1;
        } else {
            matrix[i][j] = max(matrix[i][j - 1], matrix[i - 1][j]);
        }

    }
}
```

(a) Implementation of algorithm to populate the LCS matrix



	y_j	B	D	C	A
x_i	0	0	0	0	0
A	0	↑0	↑0	↑0	↖1
B	0	↖1	←1	←1	↑1
C	0	↑1	↑1	↖2	←2
B	0	↖1	↑1	↑2	↑2

(b) Example of populated LCS matrix

Next, we traverse backward through the matrix staring the lower corner and if either the cell directly to the right or directly above contains a value to the value in the current cell, then we move to either of that cell. If both such cells have the values less that the value in current cell then we move diagonally up left and the output that character. Following this we will get the LCS in reverse order.

The resulting LCS at the end of this process contains truncated words and set of characters that should not be considered in the plagiarism check, therefore we compare the LCS with input file to extract the dictionary words and calculate the percentage.

```
char final_common_words[matrix[src_len][test_len]];
for (int i = 0; i < numword_lcs; i++) {
    for (int k = 0; k < numword_test_content; k++) {
        if (strcmp(test_splitStrings[k], LCS_splitStrings[i]) == 0) {
            strcat(final_common_words, " ");
            strcat(final_common_words, test_splitStrings[k]);
            break;
        }
    }
}
```

Steps to run the code:

The algorithm has been implemented in programming language “C”. Below are the steps to run the program.

- Make the binary using ***$make***
- Run the program using ***$./LCS test.txt database.txt***
- Visualize the results in the generated “***results.txt”*** file

Results:

- The **running time** of the Rabin-Karp algorithm in the worst-case scenario is **O(n*m)** but it has a good average-case running time.
- The time complexity of LCS algorithm is **O(m*n)** where m is the size of source and n is the size of input/test file.

Results from Rabin Karp approach:

Input file content: The input.txt is the file that has all the files to be tested for plagiarism

Test Strings: "The United States emerged from the thirteen British colonies established along the East Coast.The end of the Cold War and the collapse of the Soviet Union.United States has a very powerful army.North America migrated from Siberia by way of the Bering land bridge.The United States is a highly developed country, with the world's largest economy."

Pattern Match Found: Total matched sentence in the given test file content is 4 and total sentence in given test file is 5.

Percentage of Plagiarism: 80.0 %

As the plagiarism tolerence is set to 70 % , so a significant plagiarism is found.

Results from LCS approach:

Database file content: "The longest common subsequence LCS problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences)."

Test file content: "LCS problem is the problem of finding the longest subsequence common to all sequences, Shorter sequences are conveniently described using the term prefix"

Length of LCS Pattern: 109

LCS Found: LCS problem is the problem of finding the longest subsequence common to all sequences e sequences oen ust t e

Percentage of LCS pattern match: 71.0 %

In above LCS result, these characters are truncated or non-dictionary words: **e oen ust t e**. Therefore, we compare LCS Found with Test file content to output correct results. Below is the final plagiarism results.

Final common words length: 96

Final common words: LCS problem is the problem of finding the longest subsequence common to all sequences sequences

Percentage of Plagiarism: 62.0 %

Time Complexity Comparison:

- The worst case time complexity for Rabin-Karp algorithm is O(mn).
- The worst case time complexity for LCS algorithm is O(mn).

Which is better?

Though there is no comparison between algorithms when the question of implementation comes into picture. The only selection criterie used to chose which algorithm to apply solely depends on the type of problem in hand. Since the time complexity for both longest common substring and rabin karp algorithm is same to O(mn). But while exploring the input data and the outputs generated one common pattern seen was that while using longest common substring since there is each character comparison, so the matching patterns returned are the strings of characters without making any sense of the word.In other words the output generated is only the matched alphabet patterns which makes it difficult for one to read the patterns if one wishes to see what all is the similarity between the texts.

But, such is not the case for rabin karp algorithm,since this algorithm makes use of the hash value for entire pattern,so the entire string matches as a word with similar (but unique) hash values. The output generated is readable and clear to one's understanding as to what part of the text matches with each other in different documents. So, all in all both algorithms are an efficient output generator in terms of the accuracy but seems rabin-karp may win the race in terms of reliability since the outputs generated are more clear and user friendly such that the text patterns can be easily understood based on the similarity.