Plagiarism Detection

Project Report

Aarohi Dhireshkumar Patel (1001452956)
Krishna Joshi(1001488981)
Nishad Vijaykumar(1001380918)

University of Texas at Arlington
Computer Science and Engineering

ABSTRACT

Plagiarism is a serious problem in research. In this project, we implemented a simple plagiarism detector. Input is a corpus of existing documents and a potentially plagiarized document. Output is a set of documents from which the document was plagiarized. Following basic detectors were implemented and their performance were compared:

1. **LCSS: Implemented the LCSS algorithm for each sentence from the test file and existing corpus.**

2. **Naive String Search: Implemented the naive search algorithm that treats each sentence in the test file as a potential pattern and searches the pattern in all existing documents.**

3. **KMP: Given a test file, each sentence in the test file is treated as a potential pattern. Pattern was searched in the existing documents and matches were found.**

4. **Boyer-Moore algorithm: String matching using the Boyer-Moore algorithm was performed letter wise.**

Input: The input will be a directory containing a bunch of existing documents (for e.g., submissions from all students) and a single file that has to be evaluated.

Output: A set of documents and sentences from which the document was plagiarized.

For e.g., sentence x from input file is same as sentence y from file z and so on.
Project was implemented in Java.

1. INTRODUCTION

We thought of implementing plagiarism detector using string matching algorithms, because of the following reasons: string matching algorithms are widely used in various applications, like finding a friend through number or name, or editing documents by finding words that we need through collection of documents. Now with advanced technology it is also used in DNA sequence matching, intrusion detection in networking, text mining, etc. It is also widely used is plagiarism detection. There are numbers of algorithms are known to exist to solve string matching problem. The most basic algorithm was brute force algorithm where the pattern is compared with the collection of texts, one character at a time, until matching characters are found. For algorithm development we consider its complexity. This method might go up to the end of text files and after that can detect the pattern was not there. Different algorithms, which are more accurate in finding the occurrences, are explained below.

2. ALGORITHMS USED:

1. **LCSS**:

i. $$LCS[i,j] = \max(LCS[i-1,j], LCS[i,j-1]).$$

$$LCS[i,j] = 1 + LCS[i-1,j-1].$$

These two operations comprise all the steps necessary for LCSS algorithm. We can build a matrix out of these formulas and deduce what can be a longest common subsequence between given pattern and text.

ii. Time complexity: O(mn) and space complexity: minO(m,n)

2. NAÏVE

i. It simply test all possible placement of pattern p[1,2,..,n] relative to the text T[1,2,..,m].Shift s=0,1,..,m-n, successively and after each shift . Compare T[s+1,.., s+n] to P[1,2,..,s+m]. It acts like sliding window.
ii. For very short pattern this is preferable.
iii. Time complexity O(mn)

3. KMP

i. It differs from brute force algorithm by storing information gained from previous comparisons. It develops a prefix function at the beginning phase and whenever the string matching starts at some character and after some time it doesn't matches with the pattern. It searches through the index the prefix array. Hence it knows which characters to skip and saving lots cost of computation.

ii. For binary strings this is recommended.
iii. Time Complexity :O(m+n)

4. BOYER MOORE:

i. It is used for exact matching. The good thing about this algorithm is the longer the pattern becomes the fast the algorithm goes. Hence this algorithm is most efficient in usual application of string matching. For example, text editors.
ii. It preprocesses the pattern to skip as many alignments as possible.
iii. The working is as follows: Scanning characters of pattern from right to left . During testing of pattern P against text T, if a mismatch occurs at T[i] =k with the corresponding pattern character P[j] , if c is not present in the text then we will shift pattern p completely past T[i]. Else we will shift P until occurrence of character c in P gets aligned with T[i]
iv. For large alphabets and natural language search can be handled effectively through these method.
v. Time complexity: O(m+n)

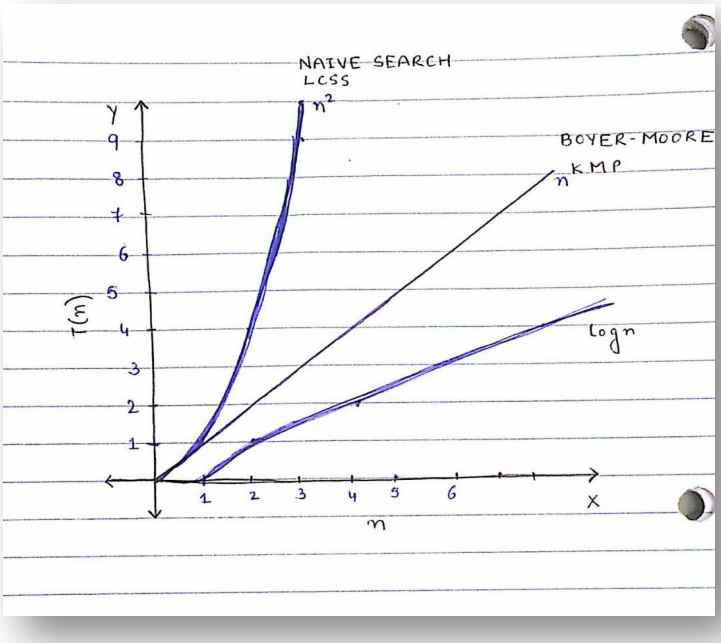


Fig 1.1 Time complexity comparisons

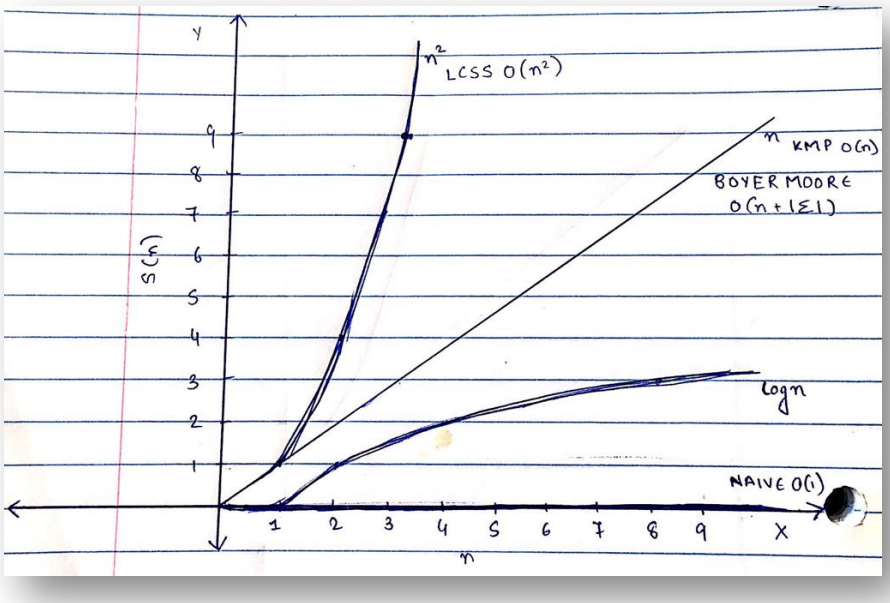
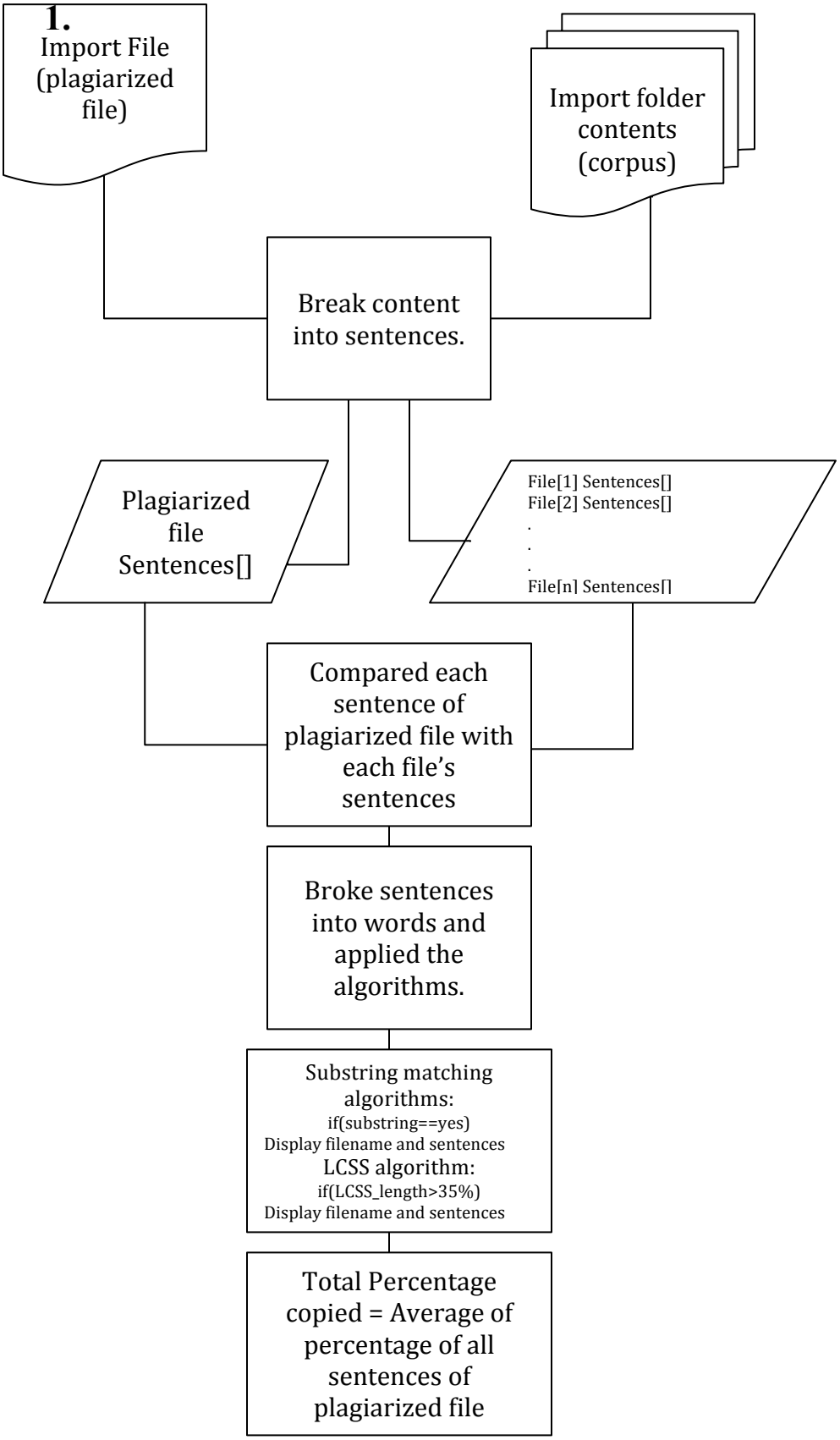


Fig 1.2 Space complexity comparison

3. SYSTEM DESIGN



4. User Interface:

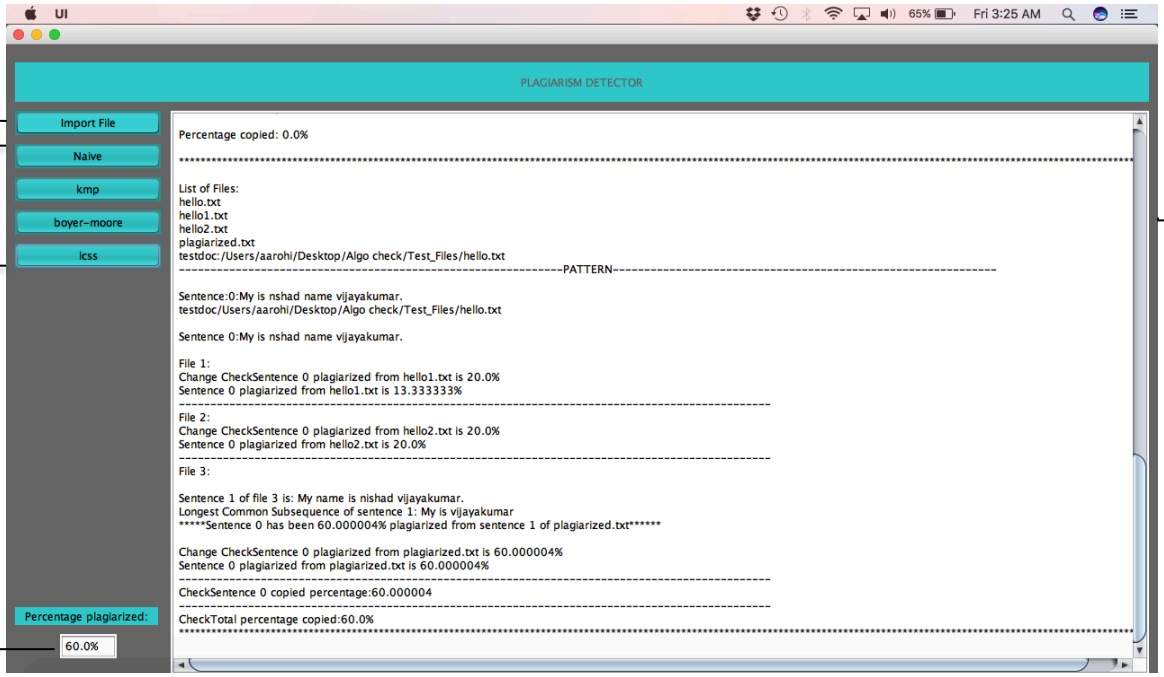


Fig 4.1 Snapshot of user interface

Final percentage copied, as analyzed by the algorithms

Buttons to select, which plagiarism algorithm to perform on the imported file

Import file on which check is to be performed

Display Area Showing summary of the Analysis of plagiarized file.

5. ANALYSIS

Which algorithm gives the exact solution of the problem?

Out of all the algorithms, LCSS provides the best solution to the plagiarism detection problem in text files.

a. KMP, Boyer Moore and Naïve Search returns true for substrings, which means that the text should be exactly copied for the detector to identify it as plagiarized.
b. However, in real world, people are smart enough to modify it and hence, LCSS which detects longest common subsequence can detect the plagiarized text even if the person has tried to modify it by removing certain sections.
c. In applications like code comparisons, we need to divide the codes into blocks and we need to compare the number of Ifs, FORs, WHILEs etc. Hence algorithms such as KMP and Boyer-Moore, which build an index that it can refer, run faster after the first time search and hence preferred.

Accuracy of Algorithms

Algorithms	Actual Percentage copied	Calculated by algorithm
LCSS	90%	86%
Naive	90%	60%
KMP	90%	60%
Boyer-Moore	90%	60%

Here, we can see that the difference between actual percentage and the calculated percentage is least in LCSS. Few words from the plagiarized file were deliberately deleted keeping the entire sentences same. But the substring matching algorithms could not identify those changes and hence did not give accurate results.

Letter wise vs. Word wise

a. In case of LCSS, letter wise is not practical for our project because, the characters can be repetitive throughout the source file but what we actually want to compare are the words and also the matrix size would grow drastically if we depend of characters.

b. For better performance (accuracy, time), in future we can divide the file into words and then clean them for e.g. by removing extra characters, representing words in singular form wherever necessary and removing prepositions.

1. **Input Length vs. Execution Time**

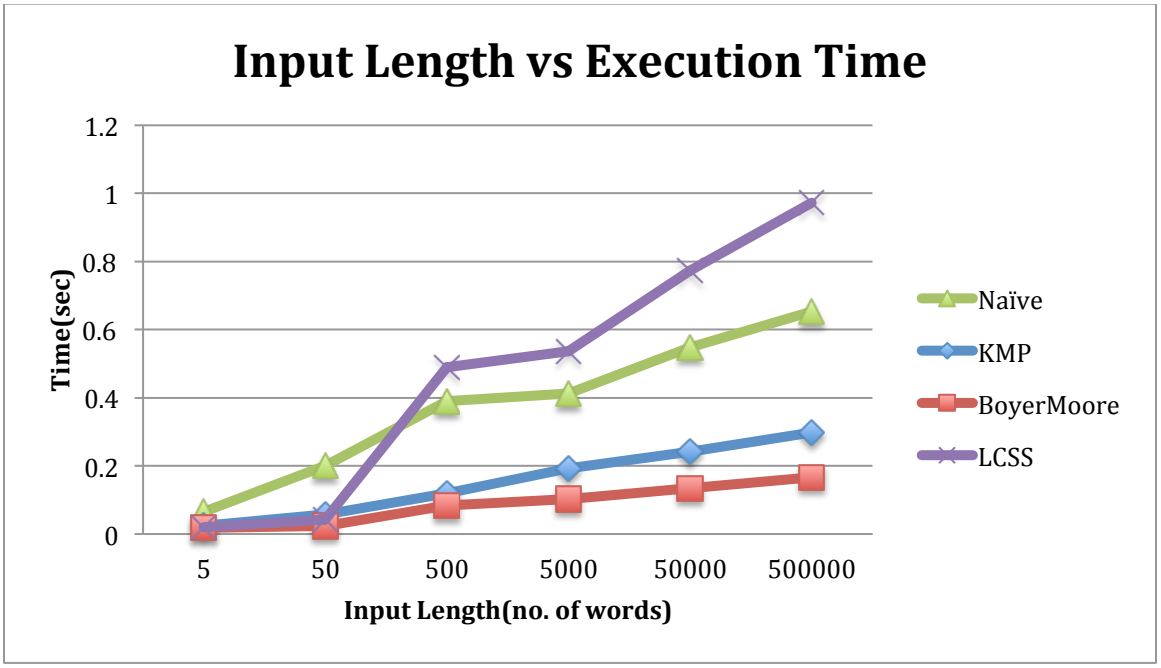


Fig 5.1

As the graph depicts, with the increase in input size Boyer Moore performance increases as compared to the other algorithms. The reason is that it works the fastest when the alphabet is moderately sized and the pattern is relatively long Boyer Moore performs the best. During the testing of a pattern P against text T, a mismatch of text character T[i] = a with the corresponding pattern character P[j] is handled as follows: If c is not contained anywhere in P, then shift the pattern P completely past[i]. Otherwise, shift P until an occurrence of character c in P gets aligned with T[i]. With smaller input size, LCSS performs good and with increase in size its execution time increases because each character in pattern is checked with each character in text.

Comparing algorithms based on Input Types

1. **Boyer-Moore** algorithm runs very fast on *large alphabet* (relative to the length of pattern). Text editors and applications that depend on output time, implement this algorithm.
2. **KMP** is recommended for Binary strings or DNA sequence matching problems where *the character set is small*. (As shown in graph.)
3. For very *short pattern size*, **brute-force** algorithm works better.
4. **Naïve string search** algorithm is ineffective when patterns are to be searched more than once. Other algorithms build an index that it can refer and can run faster after the first time search. Other algorithms are based on Deterministic-Finite-Automata approach where the algorithms learn by itself for further iterations.
5. In general, brute-force/naïve approach is a slower approach to problems as it lacks degree of definiteness.

2. Comparison for Binary Files

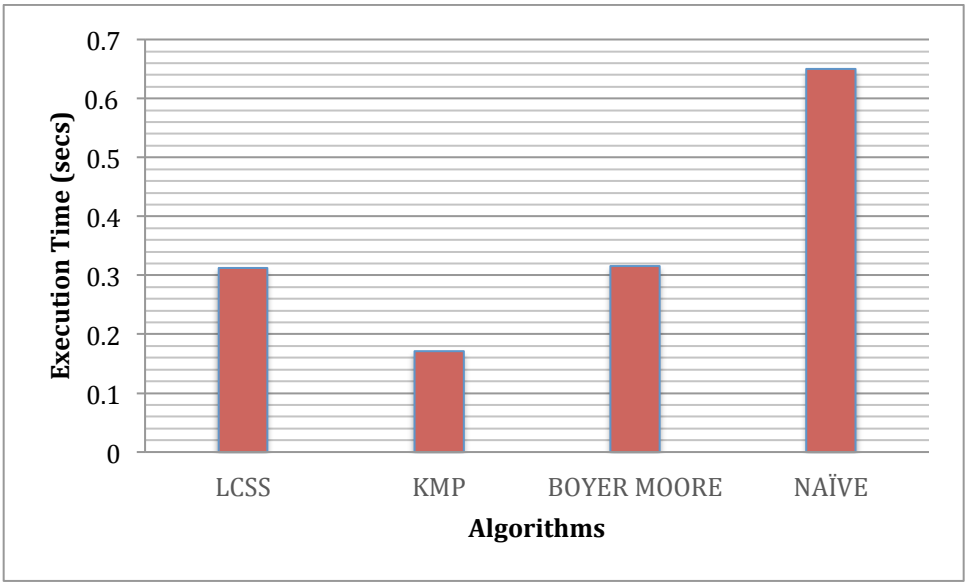


Fig 5.2

For the above experiments, files with 0s and 1s were created and tested. Execution time for KMP was least. It searches for occurrences of a pattern within text. When a mismatch occurs, the pattern itself contains sufficient information to determine where the next match could begin by creating a prefix array, and hence skipping comparing previously matched characters. Hence KMP recommended for Binary files.

3. Comparison of Execution times for different percentages of plagiarism

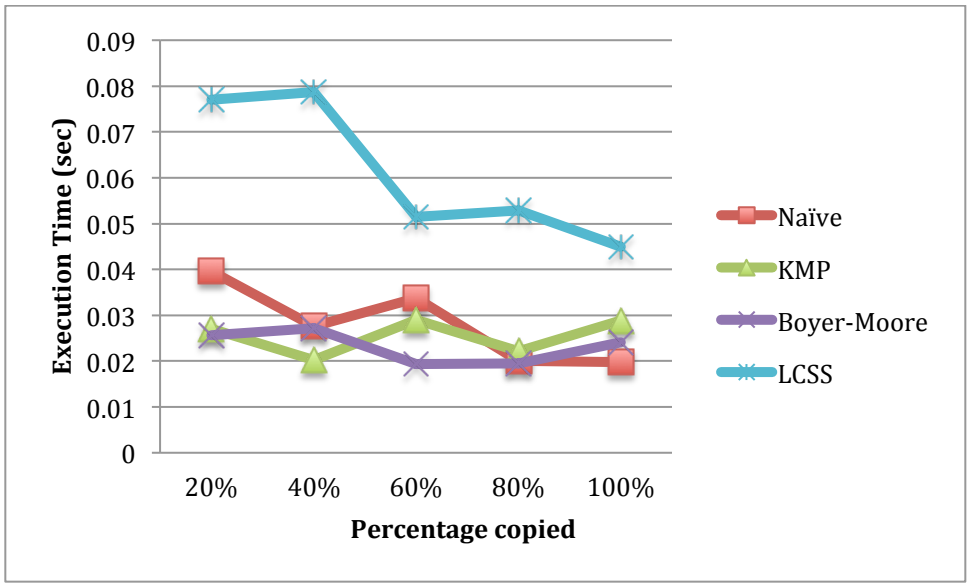


Fig 5.3

For KMP, the time does not vary much. LCSS performance increased as the math increased but still its execution time remained more than the rest of the algorithms.