CSCE 2202, 01 Project Report 2022

Simple Plagiarism Detection Utility using String Matching

Yehia Ragab 900204888

Youssef El Sherbiny 900213467

Omar Bahgat 900211747

Mohamed Ragab 900211252

CSCE 2202, 01

Department of Computer Science and Engineering, AUC

Table of Contents:

Introduction:

As plagiarism is a serious problem that has risen significantly in the past couple of years due to the ease of finding and copying information from the internet without consent, we decided to to design and implement a basic plagiarism detector using C++. The input of the program will be a set of documents (database) and a test file where we will be checking its plagiarism with respect to the documents in the database. The program will then calculate the plagiarism percentage as well as the documents from which the test file was plagiarized and it will output them on the screen.

Overview of how String Matching Works:

Regarding how the string matching process works, we will be having a test file where sentence in it will be treated as a potential pattern and will be compared against all the files in the database. The comparison part will be done through string matching where we are given a string of m characters called the pattern and a string of n characters $(m \leq n)$ called the text and we want to find a substring of the text that matches the pattern. There are several algorithms designed for the process of string matching and we will be using four of the most famous ones. The first one is Rabin-Karp algorithm which is based on hashing to find an exact match. The second one is Knuth-Morris-Pratt algorithm which is a very efficient algorithm capable of bypassing re-examination of previously matched characters. The third one is Boyer-Moore algorithm which is also an efficient algorithm capable of skipping sections of the text using data gathered during preprocessing. Lastly, there is Brute Force Matching using Hamming Distance which

compares character by character and thus yields the highest complexity among the four algorithms.

How Program Works:

To begin with, the program starts by initializing the test file and passing it four different times along with the numbers 1, 2, 3, or 4 as a second parameter. The numbers 1, 2, 3, 4 are hardcoded and signal which algorithm will be called. 1 signals Rabin-Karp algorithm, 2 signals Knuth-Morris-Pratt algorithm, 3 signals Boyer-Moore algorithm, and 4 signals Brute Force Matching using Hamming Distance. After the test file is passed along with the number to the function *plagiarism_detection,* the database is initialized and the test file is converted to a string using the function *separate_file_into_sentences.* This function reads the file with '.' acting as the delimiter and inserts each sentence into a vector called *sentences* of type string. The function then returns the vector *sentences* which contains all the sentences in the test file as strings. Next, a vector called *marked* of type integer is initalized and it's used is to mark whether the i_{th} sentence is plagiarized or not. The vector's default values are 0 and if a sentence is found to be plagiarized, it will be marked by 1 in the vector at its respective position. Afterwards, a switch statement calls the intended algorithm based on the number passed to the function. Then, each sentence in the vector *sentences* will be compared to the files in the database (after they have been converted to strings) and if a match is found, the plagiarized sentence will be marked by 1 and the file from which the sentence was plagiarized from will be inserted into a set. The reason I used a set was to avoid repeated files being displayed. For example, if sentence one and sentence two were copied from the same file, that file

would appear twice; however, this problem could be avoided by using a set. After the string matching process is done, the number of plagiarized sentences is counted by counting the total number of 1's in the vector *marked*. Next, the plagiarism percentage is calculated by dividing the number of (plagiarized sentences / total number of sentences). Finally, the plagiarism percentage and the set containing the documents from the sentences were plagiarized from is passed to the function *DisplayResults* to output the results. Using this approach, I have guranteed that each plagiarized sentence will be counted once even if it was found in multiple files. Also, it is guranteed that every file from which the sentence was plagiarized from will be inserted into the set and displayed once.

Specifications of Algorithms:

1) Rabin-Karp Algorithm:

For applications that seek to identify plagiarism, the Rabin-Karp string-matching method is utilized. This algorithm identifies string matches by adding the ASCII codes for the testing pattern and comparing them to the ASCII codes of the documents in the database; however, it has a flaw in that two words may have the same ASCII value but have different characters, as in the case of the words "abcdef," which have the same ASCII value of "ghabab" despite not being the same. As a result, I used a strong hashing function that employs a complicated equation to determine the Ascii code for each pattern to ensure that the algorithm is operating properly and prevent the hashing

collision. The complexity of this method, which looks for patterns inside text, ranges from O(n+m) in the best circumstances to O(nm) in the worst.

2) Knuth-Morris-Pratt Algorithm:

KMP is a string-searching algorithm that searches for occurrences of a string "pattern" within a larger string "text". This algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP is whenever a mismatch is detected, the algorithm is able to gather sufficient data and is able to determine where the possible next match could begin at. We take advantage of this information to bypass the re-examination of previously matched characters we already know will match anyways.

3) Boyer-Moore Algorithm:

The Boyer-Moore algorithm is a substring searching algorithm that compares each character of a substring to match it with the same characters of a string. This algorithm doesn't check all characters of a string, it gets to skip some characters using the Bad_Match_Table. The Bad_Match_Table is an indicator that assesses how many jumps it should move, which helps in the time complexity of the algorithm. The worst-case performance of the algorithm is O(mn), as n is the length of the string and m is the length of the substring. The average time is O(n). This algorithm is one of the best string-searching algorithms, which is commonly used for search bars, text labeling, and auto-correctors.

4) Brute Force Matching using Hamming Distance:

The hamming distance algorithm is used to compare two strings position by position and returns the number of mismatches between them. The time complexity of the hamming algorithm in a string matching context is $O(n^2)$.

Input Data Specifications and Output:

Regarding the input data, we created a test file consisting of several sentences and a database consisting of 3 different files each consisting of several sentences as well. Then, the program changes the test file and all files in the database to strings for comparison purposes. Each sentence within the test file is then compared to the files in the database to check if their exists a match. Regarding the results, the program displays the plagiarism percentage by dividing the (# of plagiarized sentences / total # of sentences). All the algorithms should output the same exact plagiarism percentage as well as the same files from which the sentences in the test file are plagiarized from. It is guranteed using our approach that each file will be displayed maximum once even if more than one sentence was copied from it. Also our approach gurantees accurate plagiarism percentage as a plagiarized sentence will be counted only once even if it is found in multiple files.

Conclusion:

In conclusion, our basic plagiarism detector utilizes several algorithms with different complexities in order to detect plagiarism through the process of string matching. However, all algorithms eventually yield the same results but at different costs

with some being better than the others. We have our test file where it will be compared to a set of files (database) and we check whether each sentence in our test file in exists in any of the files or not. By the end of the string matching process, we will know which of the sentences are plagiarized and the documents from which those sentences are taken from. Finally, the program outputs the name of each each algorithm utilized along with its detected plagiarism percentage and the documents from which the test file was plagiarized from.

References

1- Abdul Bari. (2018, March 30). 9.2 Rabin-Karp String Matching Algorithm. YouTube. https://www.youtube.com/watch?v=qQ8vS2btsxI

2- Abdul Bari. (2018a, March 25). 9.1 Knuth-Morris-Pratt KMP String Matching Algorithm. YouTube. https://www.youtube.com/watch?v=V5-7GzOfADQ

3- GeeksforGeeks. (2022a, September 23). Rabin-Karp Algorithm for Pattern Searching. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

4- GeeksforGeeks. (2022d, December 1). KMP Algorithm for Pattern Searching. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

5- GeeksforGeeks. (2022, November 9). Boyer Moore Algorithm for Pattern Searching. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

6- GeeksforGeeks. (2022a, July 7). Hamming Distance between two strings. https://www.geeksforgeeks.org/hamming-distance-two-strings/

Appendix

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

// function prototypes
long long unsigned hashing(string sentence);
bool Rabin_Karp_string_matching(string database, string sentence);
void calculate_longest_proper_prefix(string sentence, vector<int> &lpp);
bool Knuth_Morris_Pratt_string_matching(string database, string sentence);
void calculatePatrn(string str, int n, int patrn[]);
bool Boyer_Moore_string_matching(string str, string simlr);
int calculate_hamming_distance(string s1, string s2);
bool Hamming_Distance_string_matching(string database, string sentence);
string convert_file_to_string(string fileName);
vector<string> separate_file_into_sentences(string fileName);
void DisplayResults(double percentage, set<string> s);
void plagiarism_detection(string test_file, int algorithm);
```

```
// Rabin-Karp Algorithm
long long unsigned hashing(string sentence) {
    long long unsigned hashingCode = 0;
        // complex hashing equation to make sure that there will not be two words with the same ascii code
    for (int i = 0; i < sentence.length(); i++) {
        hashingCode = (hashingCode * 31 + sentence[i]) % 1000000007;
    }
    return hashingCode; // return the summation of the ascii code
}
bool Rabin_Karp_string_matching(string database, string sentence){
    int pattern_hash = hashing(sentence);
    for (int i = 0; i < database.length() - sentence.length() + 1; i++) {
        string substring = database.substr(i, sentence.length());
        if (pattern_hash == hashing(substring)) {
            for (int i = 0; i < sentence.size(); i++) {
                if (sentence[i] != substring[i])
                    return false;
            }
            return true;
        }
    }
    return false;
}
```

```
// Knuth-Morris-Pratt Algorithm
void calculate_longest_proper_prefix(string sentence, vector<int> &lpp) {
    int length = 0;
    lpp[0] = 0;
    int i = 1;
    while (i < sentence.length()) {
        if (sentence[i] == sentence[length]) {
            length++;
            lpp[i] = length;
            i++;
        }
        else {
            if (length != 0) {
                length = lpp[length - 1];
            }
            else {
                lpp[i] = 0;
                i++;
            }
        }
    }
}

bool Knuth_Morris_Pratt_string_matching(string database, string sentence){
    vector<int> lpp(sentence.length());
    calculate_longest_proper_prefix(sentence, lpp);
```

```
    int i = 0, j = 0;
    while (i < database.length()) {
        if (database[i] == sentence[j]) {
            i++, j++;
        }
        if (j == sentence.length()) {
            return true;
        }
        else if (i < database.length() && database[i] != sentence[j]) {
            if (j != 0) {
                j = lpp[j - 1];
            }
            else {
                i++;
            }
        }
    }
    return false;
}
```

// Boyer-Moore Algorithm

```
# define TotalChars 256
void calculatePatrn(string str, int n, int patrn[TotalChars]){
    for (int i = 0; i < TotalChars; i++){
        patrn[i] = -1;
    }
    for (int i = 0; i < n; i++){
```

```
        patrn[(int)str[i]] = i;
    }
}
bool Boyer_Moore_string_matching(string str, string simlr){
    int m = simlr.size();
    int n = str.size();
    int patrn[TotalChars];
    calculatePatrn(simlr, m, patrn);
    int shift = 0;
    while (shift <= (n - m)){
        int indicator = m - 1;
        while (indicator >= 0 && simlr[indicator] == str[shift + indicator]){
            indicator--;
        }
        if (indicator < 0){
            return true;
        }
        else{
            shift += max(1, indicator - patrn[str[shift + indicator]]);
        }
    }
    return false;
}
// Hamming-Distance Brute Force Algorithm
int calculate_hamming_distance(string s1, string s2){
    int distance = 0;
```

```
    for (int i = 0; i < s1.length(); i++){
        if (s1[i] != s2[i]){
            distance++;
        }
    }
    return distance;
}
bool Hamming_Distance_string_matching(string database, string sentence) {
    for (int i = 0; i < database.length() - sentence.length() + 1; i++){
        string substring = database.substr(i, sentence.length());
        if (calculate_hamming_distance(substring, sentence) == 0){
            return true;
        }
    }
    return false;
}
// file handling
string convert_file_to_string(string fileName){
    ifstream in;
    in.open(fileName);
    string sentence;
    string file = "";
    while (getline(in, sentence,'.')){
        file += sentence;
    }
    return file;
```

```
}
vector<string> separate_file_into_sentences(string fileName){
    ifstream in;
    in.open(fileName);
    string sentence;
    vector<string> sentences;
    getline(in, sentence, '.');
    sentences.push_back(sentence);
    while (getline(in, sentence, '.')){
        sentence.erase(sentence.begin());
        sentences.push_back(sentence);
    }
    sentences.pop_back();               // pop back last element of the vector as a dummy space is read
    return sentences;
}
// displaying results
void DisplayResults(double percentage, set<string> plagiarized_docs){
    cout << percentage << "%\n";
    cout << "Documents from which the test file was plagiarized: \n";
    int i = 1;
    set<string> :: iterator it;
    for(it = plagiarized_docs.begin(); it != plagiarized_docs.end(); it++){
        cout << i << ". " << (*it).substr((*it).size()-9) << "\n"; // to avoid printing full path. remove substr if
printing name
        i++;
    }
```

```
    for(int i = 0; i < 51; i++) cout << "-";
    cout << "\n";
}
void plagiarism_detection(string test_file, int algorithm){
    const int num_of_files = 3;
    string file1 = "/Users/omar_bahgat/Documents/College/College/College/file1.txt";
    string file2 = "/Users/omar_bahgat/Documents/College/College/College/file2.txt";
    string file3 = "/Users/omar_bahgat/Documents/College/College/College/file3.txt";
    // Database where we will compare our test file with
    string database[num_of_files] = {file1, file2, file3};
    string comparison_file; // database files which will be converted to strings when comparing
    vector<string> sentences = separate_file_into_sentences(test_file);
    vector<int> marked(sentences.size(),0); // vector indicating sentence is plagiarized or not

     set<string> plagiarized_documents; // will contain documents from which the potential document was
plagiarized
    // for each sentence, check if it exists in any of the files or not. If yes, we mark it by 1 and file to the set
    for(int i = 0; i < sentences.size(); i++){
        for(int j = 0; j < num_of_files; j++){
         comparison_file = convert_file_to_string(database[j]);  // convert file to string to be able to compare
            switch (algorithm){
                case 1: {
                    if(Rabin_Karp_string_matching(comparison_file, sentences[i]) == 1){
                        marked[i] = 1;
                        plagiarized_documents.insert(database[j]);
                    }
```

```
            }
            case 2: {
               if(Knuth_Morris_Pratt_string_matching(comparison_file, sentences[i]) == 1){
                  marked[i] = 1;
                  plagiarized_documents.insert(database[j]);
               }
            }
            case 3: {
               if(Boyer_Moore_string_matching(comparison_file, sentences[i])){
                  marked[i] = 1;
                  plagiarized_documents.insert(database[j]);
               }
            }
            case 4: {
               if( Hamming_Distance_string_matching(comparison_file, sentences[i]) == 1){
                  marked[i] = 1;
                  plagiarized_documents.insert(database[j]);
               }
            }
         }
      }
   }
double counter_matching = count(marked.begin(), marked.end(), 1); // if value = 1, then sentence is plagiarized
double percentage = ((counter_matching / sentences.size()) * 100); // divide # of plagiarized sentences / total # of sentences
```

```
 DisplayResults(percentage, plagiarized_documents);
}

int main(){
    string test_file = "/Users/omar_bahgat/Documents/College/College/College/testfile.txt";
    cout << "Rabin-Karp: ";
    plagiarism_detection(test_file, 1);

    cout << "Knuth-Morris-Pratt: ";
    plagiarism_detection(test_file, 2);

    cout << "Boyer-Moore: ";
    plagiarism_detection(test_file, 3);

    cout << "Hamming Distance: ";
    plagiarism_detection(test_file, 4);

    return 0;
}
```