Design and Development of Plagiarism Detection Software in C++

Muhammad Shumak Chouhan
School of Mechanical and Manufacturing Engineering
National University of Sciences and Technology
Islamabad, Pakistan
shumakchouhan789@gmail.com

Wali Ullah
School of Mechanical and Manufacturing Engineering
National University of Sciences and Technology
Islamabad, Pakistan
raiullah@gmail.com

Muhammad Sachal Israr
School of Mechanical and Manufacturing Engineering
National University of Sciences and Technology
Islamabad, Pakistan
israrsachal@gmail.com

Abdul Hadi
School of Mechanical and Manufacturing Engineering
National University of Sciences and Technology
Islamabad, Pakistan
hadijk1269@gmail.com

Muhammad Talha Irfan
School of Mechanical and Manufacturing Engineering
National University of Sciences and Technology
Islamabad, Pakistan
meinthami@gmail.com

Abstract* — *This document proposes a software written in C++ (C plus plus) programming language which accepts a text file, compares it against a pool of files for plagiarism of words and phrases, and prepares a similarity index report for the submitted file.

Keywords* — *C++, C plus plus, Programming, Program, Software, Plagiarism, Similarity, Detector.

I. Introduction

Plagiarism is copying someone else's work and presenting it as your own. It is a case where academics take large portions of text and do not credit or cite original owners, often unknowingly owing to lack of free-of-cost, no-restriction, and comprehensive plagiarism-checking application which they could use hassle-free.

In this document, we propose a solution to combat plagiarism, a software in the programming language C++ (C plus plus). Our developed program works by asking the user for the name of the file for which the plagiarism needs to be checked, which it then compares against a set of source files, taking count of any words, then phrases that are similar in our test file and each of the original source files. Using the data mentioned before, total words in the test file and a similarity index equation, the similarity index is calculated for the test file versus each source file. The program concludes with attaching a similarity report that contains essential data related to plagiarism, at the end of our test file itself.

II. Methods

A. File Input and File Streams: main( ) function

We experimented to find an optimal way to implement file name and file stream arrays and figured out we can make a string array to store file names and open an array of streams. We decided to make index 0 refer to our test file, and indices 1 to 5 for the source files 1 to 5 respectively. A *do-while* loop nested in a *for* loop allowed us to keep asking the user for the file names (if the file with the name provided fails to open in the stream) until each of the 6 files have their names inputted and the files opened successfully in their streams. A solitary output stream for printing similarity reports to the test file was also opened.

```
    int i;
    string file_names[1][6];
    ifstream file[6];
    for (i = 0; i <= 5; i++) {
        string text = (i == 0)? "test" : "source";
        do {
            cout << "Enter name of the " << text << "
file, file " << i << ": ";
            getline(cin, file_names[0][i]);
            file[i].open(file_names[0][i]);
            if (!file[i].is_open()) cout << "Failed
to open \"" << file_names[0][i] << "\".\nPlease try
again, making sure the file is in the same directory
as this software.\n\n";
            else { cout << "The " << text << " file
\"" << file_names[0][i] << "\" opened
successfully!\n\n"; }
        } while (!file[i].is_open()); }

    ofstream test_file(file_names[0][0], ios::app);
```

This reduced a lot of repetition in our code, which would have been the case if we opened streams separately to execute the same thing (as we tried initially):

```
    string source_file_1, source_file_2,
source_file_3, source_file_4, source_file_5,
test_file_name;
    cout << "Enter file name for test file:\n";
    getline(cin, test_file_name);
    cout << "Enter file name for source file 1:\n";
    getline(cin, source_file_1);
    cout << "Enter file name for source file 2:\n";
    getline(cin, source_file_2);
    cout << "Enter file name for source file 3:\n";
    getline(cin, source_file_3);
    cout << "Enter file name for source file 4:\n";
    getline(cin, source_file_4);
    cout << "Enter file name for source file 5:\n";
    getline(cin, source_file_5);
```

```
    ifstream file_1(source_file_1);
    ifstream file_2(source_file_2);
    ifstream file_3(source_file_3);
    ifstream file_4(source_file_4);
    ifstream file_5(source_file_5);
    fstream test_file(test_file_name, ios::app);
```

Another issue with this method is that the *fstream* will not give an error if the test file does not exist, and create an empty file.

We call the three user-defined functions we made:

```
    word_check();
    phrase_check();
    similarity_report(test_file);
```

Finally, we closed the file streams:

```
    for (i = 0; i <= 5; i++) { file[i].close(); }
    test_file.close();
```

B. *Data Storage and Arithmetic Functions: Implementation of the Header file containing a Class*

We created a Header file named *SimilarityReporter.h* containing a Class *SimilarityReporter* to go with our main code. It primarily functioned to store all numerical data and perform arithmetic calculations. The comments in our code generally explain what each line does:

```
class SimilarityReporter { // class declaration of
name SimilarityReporter
    private: // private stuff cannot be accessed
directly by main cpp file, but notice how
sum_word_count can be accessed when the public
sum_words() function is called
        int i; // simple integer i for loops
        int sum_word_count = 0;  // a simple sum of
word counts integer
    public: // public stuff can be accessed by main
cpp file, these variables have been called in it to
store values
        int word_count[6] = {0}; // index [0] is TEST
FILE word count, initialise all values with zero
        int similar_words[6] = {0}; // index [0] is
overall similar words, initialise all values with
zero
        double similarity_percentage[6] = {0}; //
index [0] is overall similarity percentage,
initialise all values with zero
        int similar_phrase_count = 0; // a simple
similar phrases counter integer
        int sum_words() { // a function to sum up
word count of all files
            sum_word_count = 0; // start with zero
each time the function is called, otherwise it'll be
having the sum previously calculated
            for (i = 0; i <= 5; i++) { sum_word_count
+= word_count[i]; } // count words from each file,
using array iteration
            return sum_word_count; // return the
value of the variable
        };
        int sum_similar_words() { // a function to
sum up similar word count of all files
            similar_words[0] = 0; // start with zero
each time the function is called (for reasons above),
will be storing the value in index zero of the array
            for (i = 1; i <= 5; i++) {
similar_words[0] += similar_words[i]; } // notice how
i starts with 1, the sum of 1 to 5 stores in index 0
            return similar_words[0]; // return the
value of 0 index of the variable
        };
        void calculate_similarity_index() { // a
function to calculate similarity index using
formulae, void data type so doesn't return anything,
just performs statements inside it
            similarity_percentage[0] = (
(double)sum_similar_words() / (double)sum_words()
)  * 100; // index zero again for overall similarity.
and notice how the above two functions are being
called in this function. and they are being
typecasted from integer datatype to double, for
accurate arithmetic.
            for (i = 1; i <= 5; i++) {
similarity_percentage[i] = ( (double)similar_words[i]
/ ( (double)word_count[0] + (double)word_count[i] ) )
* 100; } // a loop for individual similarities with
corresponding indices, the integer values are
typecasted to double again for arithmetic.
        }; // no, nothing is returned to the function
this function was called from. this one just performs
math

};
```

Here, we utilised the unused index 0 of the *similar_words* integer and the *similarity_percentage* double to store the overall values.

The *sum_words*() and the *sum_similar_words*() functions simply perform addition, and return the total number of words in the files, and similar words respectively.

The general equation in the *calculate_similarity_index*() function is as follows:

$$Similarity\ Index = \frac{Number\ of\ Shared\ Words}{Total\ Number\ of\ Words\ in\ the\ files} x\,100 \quad (1)$$

The integer values used in calculations are "typecasted" to double for accuracy in decimal points, otherwise the results are rounded down (treated like integers) and stored.

Our *SimilarityReporter* class can be used in our main file *PlagiarismDetector.cpp* by declaration as:

```
SimilarityReporter data_of_file;
```

We originally indexed *data_of_file* instead of the *word_count*, *similar_words* and *similarity_percentage* values in the class itself, but it prevented *for* loops in the arithmetic functions to work:

```
SimilarityReporter data_of_file[6];
```

C. *User-defined Functions*

In the main code three user-defined functions *word_check( )*, *phrase_check( )* and *similarity_report( )* are created before the *main( )* function, and called in it:

```
    word_check();
    phrase_check();
    similarity_report(test_file);
```

1. *Word Plagiarism Detection: word_check( )* function

The basic code body of word plagiarism detection comes out to be:

```
void word_check() {
    string word, test_word, source_word;
    // COUNT TOTAL WORDS OF EACH FILE
    for (i = 0; i <= 5; i++) {
        data_of_file.word_count[i] = 0;
        while(file[i] >> word) {
            data_of_file.word_count[i]++; }}
    // ACTUAL WORD PLAGIARISM CHECK
    for (i = 1; i <= 5; i++) {
            file[0].clear();
            file[0].seekg(0);
        while( file[0] >> test_word ) {
                file[i].clear();
                file[i].seekg(0);
            while ( file[i] >> source_word ) {
                if( source_word == test_word ) {
                     data_of_file.similar_words[i]++;
                                break; }}}}}
```

The total word counter for each file is simple enough: a for-loop for files 0 to 5 (test and 5 source files) and counting each word in the file while the words can be inputted.

For similar words, a word from test file is taken and is compared with the words from source file *i* (the running value of *i* from 1 to 5 during the loop) until the **string matches exactly** and increments the value of similar words. The *break;* terminates the source file *while* loop so that the test file loop takes in the next word and repeats the process until all words from the test file are done.

We improved this further to exclude symbol-only words through (*string::find_first_of* - C++ Reference, 2022) [1], adjust for small words, punctuation and similar words (such as past tense), implementing the "substring" *find* method (string::find - C++ Reference, 2022) [2] and string *length* [3]:

```
const string letters_numbers =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789";
int min_word_substring_len = 4;
int substring_len_diff_tolerance = 2;
void word_check() {
    string word, test_word, source_word;
    int test_substring_position,
source_substring_position;
    // COUNT TOTAL WORDS OF EACH FILE
    for (i = 0; i <= 5; i++) {
            data_of_file.word_count[i] = 0;
        while(file[i] >> word) {
            if ( word.find_first_of(letters_numbers)
!= string::npos ) {
                data_of_file.word_count[i]++;
                }}}
    // ACTUAL WORD PLAGIARISM CHECK
    for (i = 1; i <= 5; i++) {
            file[0].clear();
            file[0].seekg(ios::beg);
        while( file[0] >> test_word ) {
                file[i].clear();
                file[i].seekg(ios::beg);
            while ( file[i] >> source_word ) {
                    test_substring_position =
source_word.find(test_word);
                    source_substring_position =
test_word.find(source_word);
                if(test_substring_position !=
string::npos || source_substring_position !=
string::npos) {
                    // some conditions to meet: must
contain at least a letter or number, and length
adjustment, for punctuation, "too small vs too large"
substring exclusion and similar words (SEE KNOBS
ABOVE)
                    if (
test_word.find_first_of(letters_numbers) !=
string::npos &&
source_word.find_first_of(letters_numbers) !=
string::npos ) {
                        if ( test_word.length() >=
min_word_substring_len && source_word.length() >=
min_word_substring_len ) {
                            if ( test_word.length()
<= source_word.length() +
substring_len_diff_tolerance && test_word.length() >=
source_word.length() - substring_len_diff_tolerance )
{
                                data_of_file.similar_
words[i]++;
                                break;
    }}}}}}}}
```

This improvement resulted in better accuracy of similar words and hence similarity index.

We also tried a popular method of making a set list of words to ignore but opted against it as we can only put a select few words: such a list is always non-exhaustive.

2. *Phrase Plagiarism Detection: phrase_check( )* function

```
int min_phrase_len = 2;
string similar_phrase_data[2][10000];
void phrase_check() {
    string test_word, test_phrase, source_line,
temp_phrase;
    int test_substring_position, j, test_words_done;
    for (i = 1; i <= 5; i++) {
        file[0].clear();
        file[0].seekg(0);
        file[i].clear();
        file[i].seekg(0);
        test_words_done = 0;
        while ( file[0].good() ) {
            for ( j = 1; j <= test_words_done; j++ )
{ file[0] >> test_word; }
            if ( file[0].eof() ) break;
            else {
                file[0] >> test_word;
                test_phrase = test_word;
                if ( file[0].eof() ) break; }
        for ( j = 1; j <= min_phrase_len - 1; j++ ) {
                if ( !file[0].eof() ) {
                    file[0] >> test_word;
                    test_phrase = test_phrase + " " +
test_word; }}

            // ACTUAL PHRASE PLAGIARISM CHECK
            while ( !file[i].eof() ) {
                    getline(file[i], source_line);
                    test_substring_position =
source_line.find(test_phrase);
                if ( test_substring_position !=
string::npos ) {
```

```
                    test_words_done = test_words_done
+ min_phrase_len;
                    file[i].clear();
                    file[i].seekg(0);
                    break; }}
            if ( test_substring_position ==
string::npos ) {
               test_words_done = test_words_done + 1;
                file[0].clear();
                file[0].seekg(0);
                file[i].clear();
                file[i].seekg(0);
                continue; }
            while ( test_substring_position !=
string::npos ) {
                if ( file[0].eof() ) break;
                else {
                    file[0] >> test_word;
                    temp_phrase = test_phrase + " " +
test_word; }
                while ( !file[i].eof() ) {
                        getline(file[i],
source_line);
                        test_substring_position =
source_line.find(temp_phrase);
                    if ( test_substring_position !=
string::npos ) {
                        test_words_done =
test_words_done + 1;
                        test_phrase = temp_phrase;
                        file[i].clear();
                        file[i].seekg(0);
                        break;
            }}}
            data_of_file.similar_phrase_count++;
            similar_phrase_data[0][data_of_file.simil
ar_phrase_count] = to_string(i);
            similar_phrase_data[1][data_of_file.simil
ar_phrase_count] = test_phrase;
            file[0].clear();
            file[0].seekg(0);
            file[i].clear();
            file[i].seekg(0);
    }}
}
```

A 2D string array *similar_phrase_data* stores the similar phrases with their corresponding source file numbers.

Minimum number of words for the phrase are specified by *min_phrase_len*. There are multiple checks for cases where the test file ends, or when the test and source file positions need to seek to the beginning. The test file is able to progress word by word generally, by *test_words_done* and the loop that gets to the position of the required word for the next iteration of phrase search.

The source file is looped line by line to search for the phrase from the test file. If found, it keeps on adding words to the phrase until the phrase is not found in any of the lines of the source file. Now we obtain a similar phrase; the similar phrase count is incremented, and the phrase and the value of *i* (source file number) is stored in the 2d array.

The test file continues to get a phrase and loop (whether it finds a similar phrase or not) until it ends; in which case it seeks to beginning and the next source file is checked for similar phrases using the same procedure.

The phrase can have words of any length, small or large; it employs a simple “substring” *find* [2] method from a line.

3. *Similarity Reports: similarity_report( )* function

Our bare minimum code for *similarity_report( )* function is as follows:

```
#include <iomanip>
const string letters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
";
void similarity_report(ofstream& test_file) {
    cout << endl << endl << setw(0) << "Total number
of similar words found = " <<
data_of_file.sum_similar_words() << endl;
    for (i = 1; i <= 5; i++) {
        cout << setw(15) <<
data_of_file.similar_words[i] << " from file " << i
<< endl; }
    data_of_file.calculate_similarity_index();
    cout << endl << endl << setprecision(3) <<
"\t\tSimilarity Index = " <<
data_of_file.similarity_percentage[0] << "%" << endl;
    test_file << endl << endl << setprecision(3) <<
"\t\tSimilarity Index = " <<
data_of_file.similarity_percentage[0] << "%" << endl;
    for (i = 1; i <= 5; i ++) {
        cout << setprecision(3) << "Source " << i <<
"\t= " << data_of_file.similarity_percentage[i] <<
"%" << endl;
        test_file << setprecision(3) << "Source " <<
i << "\t= " << data_of_file.similarity_percentage[i]
<< "%" << endl; }
    test_file << endl << endl << "Total Number of
Similar Phrases = " <<
data_of_file.similar_phrase_count << endl <<
"    Similar Phrases/Clauses       Source File" <<
endl <<
"    \"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"
\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"";
    for (i = 1; i <=
data_of_file.similar_phrase_count; i++) {
        test_file << endl << "    " << letters.at( (i
- 1) % letters.size() ) << ")  ";
        test_file << setw(30) << std::left <<
similar_phrase_data[1][i];
        test_file << " " <<
similar_phrase_data[0][i]; }
}
```

The *iostream* library provides us with *endl* manipulator.

The *iomanip* library contains various other manipulators such as: *setprecision*, *setw* and *std::left*. The escape sequence *\t* represents a tab space, and *\”* allows us to print the *”* character. We used these manipulators and escape sequences to format our reports in a presentable form.

The function is of *void* datatype, meaning that like the other two user-defined functions, it does not return anything to the main function; but unlike them, it has an *ofstream* argument, the parameter of which is passed when the function is called in *main( )*, and is treated as a reference due to *&* rather than a copy.

a) The similar words report calls the *sum_similar_words( ) function* declared in the *SimilarityReporter* class.

b) The similarity index report takes calculated data from *similarity_percentage* variable declared in the class, and hence requires *calculate_similarity_index()* function in the class itself to be called, to store processed values in it.

c) The similar phrases report uses data from the two-dimensional array we stored phrases in. It lists them in an alphabetic list which was implemented by printing the character at – *"at( )" function* (string::at - C++ Reference, 2022) [4] - the appropriate index in the constant *letters* string.

III. Results

Our complete code (with descriptive comments and sample files) is available at (GitHub - meinthami/CS-114-Final-Project: Final Project for CS-114 course, Semester 1, SMME ME-13., 2022) [5] where it has been organised a bit for clarity and presentation, and optimised for detecting plagiarism in codes.

For simple text-based samples, we took a test file and 5 source files (available at [5]) with text written on a single theme (Plagiarism Detection). The similarity reports for our samples, printed in the console and at the end of the test file are given in Fig. 1 and 2, respectively:

```
Total number of similar words found = 101
            36 from file 1
            21 from file 2
            18 from file 3
             9 from file 4
            17 from file 5


                Similarity Index = 9.63%
Source 1        = 12.5%
Source 2        = 12.8%
Source 3        = 5.26%
Source 4        = 3.12%
Source 5        = 4.01%
```

Fig. 1. Output in a Console

```
      Similarity Index = 9.63%
Source 1    = 12.5%
Source 2    = 12.8%
Source 3    = 5.26%
Source 4    = 3.12%
Source 5    = 4.01%


Total Number of Similar Phrases = 17
    Similar Phrases/Clauses       Source File
    """""""""""""""""""""""""""""""""""""""""""""
    a)  is a                          1
    b)  against a set of              1
    c)  a similarity                  1
    d)  similarity report             1
    e)  of a                          1
    f)  source files.                 1
    g)  plagiarism detection software 2
    h)  against a                     2
    i)  a similarity                  2
    j)  plagiarism of                 2
    k)  text file                     2
    l)  in a                          3
    m)  of similar                    3
    n)  of a                          3
    o)  source files.                 3
    p)  in a                          4
    q)  the text                      5
```

Fig. 2. Output in our sample Test File

In the C++ and Python codes we tested respectively, it provides similarity/plagiarism reports with a greater accuracy than without optimisation.

IV. Discussion and Summary

Our plagiarism detection software has shown a high accuracy, in counting similar words, calculating percentage similarity index, and displaying similar phrases. We are immensely proud of our achievements in developing a versatile software that can be adjusted easily according to requirements and is robust in providing the desired results.

While our software highly meets our expectations, like all automation and programming it has a handful of assumptions and exceptions/edge-cases involved, the examples of some of which are:

- Our software is fully case-sensitive; "Plagiarism" and "plagiarism" would not be similar.
- "similarity/plagiarism" would be counted as a single word instead of two.
- small keywords like "cin" (in C++ code), "key", "end" and more would be ignored if we keep *min_word_substring_len* as 4, and words like "where", "when", "with" would be checked for similarity. There is no one-size-fits-all.
- The above principle also applies to *substring_len_diff_tolerance*, where we kept the substring length difference limits to 2. Words like "fast" and "fastest" would not be similar.
- In an exact match, similarity percentage formula used provides a 50% index when the test file and a source file are identical. When all 6 files are identical, the overall index remains 83.3%.
- Phrase check is "string-lenient" (substring check), but "punctuation-strict": punctuation is important.

These assumptions and minor exceptions are best treated as such and are a normal part of automation.

Acknowledgments

The authors owe, and would like to thank Mr. Ali Hassan, Lab Engineer at School of Mechanical and Manufacturing Engineering (SMME), National University of Sciences and Technology H12 (NUST H12), Islamabad, Pakistan; and Dr. Omer Gilani, Head Department of Biomedical Engineering & Sciences at SMME, NUST H12, Islamabad, Pakistan, for their valuable insight in the development of the plagiarism detection software. We also thank the authors of (cplusplus.com - The C++ Resources Network, 2022) [6] for a comprehensive documentation on standard C++ references and resources.

References

[1] "string::find_first_of - C++ Reference." [Online] Available at: https://www.cplusplus.com/reference/string/string/find_first_of/ [Accessed 16 February 2022].

[2] "string::find - C++ Reference." [Online] Available at: https://www.cplusplus.com/reference/string/string/find/ [Accessed 16 February 2022].

[3] "string::length - C++ Reference." [Online] Available at: https://www.cplusplus.com/reference/string/string/length/ [Accessed 16 February 2022].

[4] "string::at - C++ Reference." [Online] Available at: https://www.cplusplus.com/reference/string/string/at/ [Accessed 16 February 2022].

[5] "meinthami/CS-114-Final-Project: Final Project for CS-114 course, Semester 1, SMME ME-13," Accessed on Feb. 19, 2022. [Online]. Available: https://github.com/meinthami/CS-114-Final-Project

[6] "cplusplus.com - The C++ Resources Network." [Online] Available at: https://www.cplusplus.com/ [Accessed 16 February 2022].