Simple Plagiarism Detection Utility using String Matching

Samar Ahmed, Abdallah Ibrahim, and Bushra Alqhatani

Department of Computer Science and Engineering, The American University in Cairo

CSCE 1101, Section 2

Dr. Howaida Ismail

5/10/2023

Table of Contents

Abstract

This paper will contain information about 5 main string-matching algorithms that have been used to check for plagiarism. The five algorithms are hammingDistance, KMP, Rabin-Karp, Brute-Force, and Boyer-Mooyer. Each has its own method of checking between two different strings. The main method used in this project which will be referred to in this paper is the text inputted by the user will be split into words so that each word can be compared with the other texts from other documents. Counters are used in all of these algorithms to count the number of plagiarized words and create a percentage, furthermore, the vector UniqueWords is used to display all of the plagiarized words. The main methodology for this paper was to check and compare the accuracy of each algorithm. It was then found that Brute-Force and Boyer-Mooyer are less accurate since they check using characters rather than words. This paper will present an introduction to a string-matching algorithm, following the problem definition we present and the methodology leading to the data specification, the input of the user showing the results, and finally a critical analysis of the project.

Introduction

String-matching algorithms, also known as string-matching algorithms, are a significant family of string algorithms used in computer science. These algorithms look for the location of one or more strings (also known as patterns) inside a larger string or text. Even though the practical applications fall under the purview of computer science research, string matching is a classic and ongoing problem. Within a well-built string or "Text" in this domain, one or more strings called " Patterns" must be searched. Algorithms or strategies for string matching play a crucial part in a variety of applications or challenges in the real world. Spell checkers, spam filters, intrusion detection systems, search engines, plagiarism detectors, bioinformatics, digital forensics, information retrieval systems, and others are only a few of its crucial applications. The string-searching algorithms would align the pattern with the beginning of the text and proceed until a match was found or the text's end was reached. The most typical method will operate as follows: the text will be scanned and its size set to m. By comparing the characters in the text to the characters in the pattern, it will be possible to determine if the pattern appears in each segment of the text that depends on m. There are numerous string-matching methods; however, the issues are numerous and challenging to investigate and solve (Al-Khamaiseh & ALShagarin, 2014). The Naive String Matching Algorithm, the Rabin-Karp Algorithm, Finite Automata, the Knuth-Morris-Pratt Algorithm, and the Boyer-Moore Algorithm are some well-known and often employed string matching techniques.

Assuming that each action requires the same amount of time, the time complexity is the total number of operations required to complete a task via an algorithm. In terms of time complexity, the algorithm that completes the task with the fewest operations is regarded as the most effective one. In this study, we compare the accuracy of four different string-matching algorithms: hamming distance, KMP search, Rabin Karp, Boyer Moore, and Brute force. String matching techniques vary in time complexity and accuracy.

Problem definition

Today, string-matching algorithms are widely used to find patterns in large amounts of text. These algorithms have a wide range of applications, such as intrusion detection systems, plagiarism detection, and bioinformatics (Al-Khamaiseh & ALShagarin, 2014). Several published papers have been used to compare the running times of the string-matching algorithms. An integer k, a pattern string, and a text string make up the three inputs. The experiment's goal was to locate every occurrence of the pattern in the text with at most k changes (JOKINEN et al., 1996). This research will take into account five different algorithms. It would also be revealed that the methodologies' accuracy can vary significantly from one another.

Methodology

Finding the most accurate algorithm out of the five is the goal of this research. We have determined that comparing the outputs of all of the selected algorithms is the most effective technique to carry out this research. There are several steps in the process of comparing the algorithms. First, we run each algorithm through a plagiarism test. To ensure the accuracy of the results, we also utilize identical text to check for plagiarism. We also perform other types of plagiarism checks.

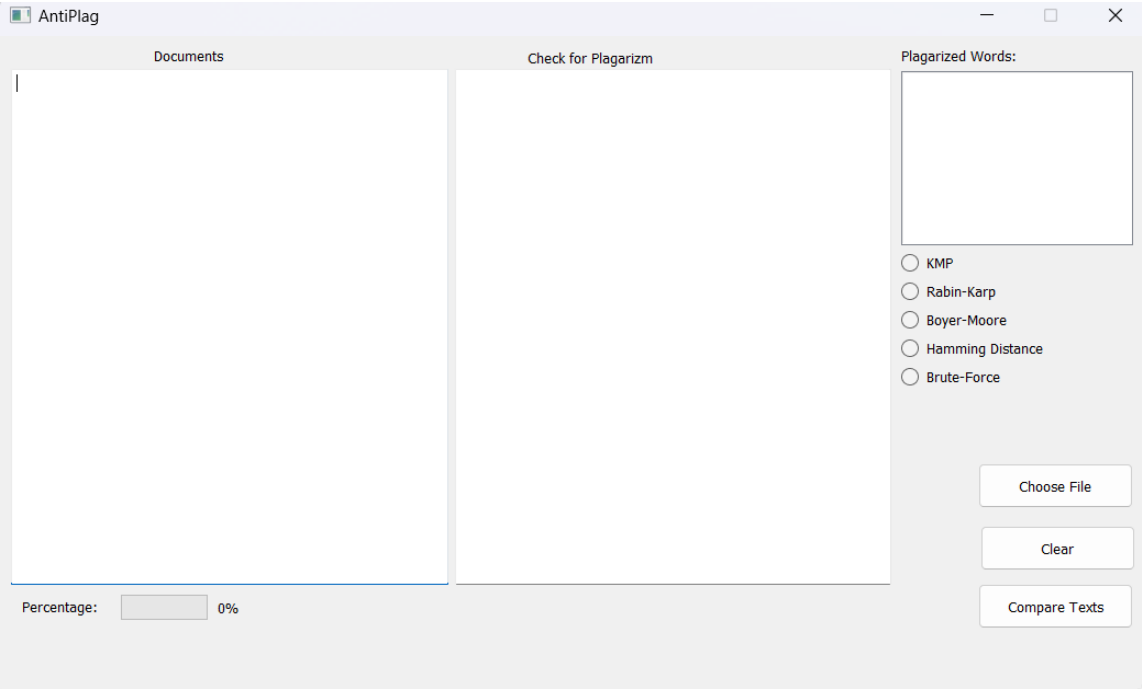
One method involves entering a text as a user and comparing it to files that have already been stored. We also perform other types of plagiarism checks. One method involves entering a text as a user and comparing it to files that have already been stored.

Another method involves entering a text file, after which the algorithms can examine it for similarities by comparing it to stored files. In addition to this, entering two texts and comparing them to see if there are any similarities could be another method. We needed to test each algorithm to look for variances in accuracy, so this approach to research was chosen.

Specification of Algorithms to be Used

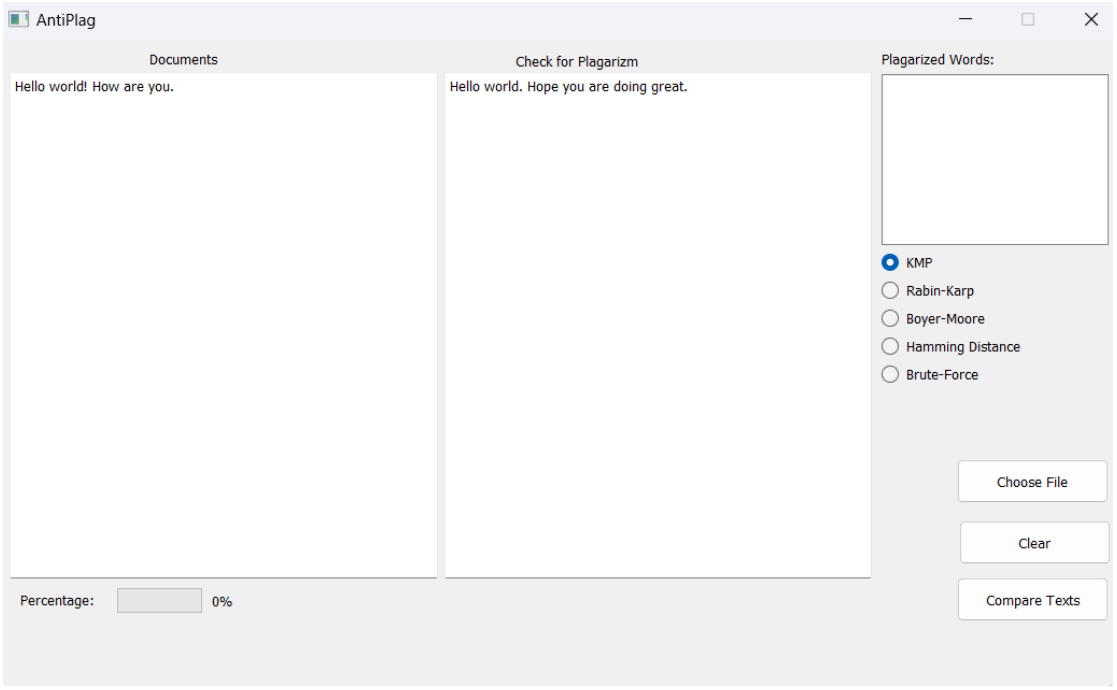
In this study, the algorithms hamming distance, KMP search, Rabin Karp, Boyermoore, and brute force will be applied. These algorithms were chosen because they are the most widely used and popular ones for study.

Data Specifications



The user will input the documents they want by pressing the choose file option or they can input the data through the keyboard. They will then enter the information they want to check for plagiarism on the right-hand side. They will then pick one of the radio buttons on the right side, showing the type of algorithm they want to use to check for plagiarism. Once they have done those three steps all they will have to do left is press the compare texts button. This will check which algorithm they chose and output the results by doing so. It will then output the words that are plagiarised on the top left box. If they want to check using a different algorithm or erase the content on both text edits, this can be done by pressing the clear button. Finally, the percentage of plagiarism will appear at the bottom left bar.

Here is an example:

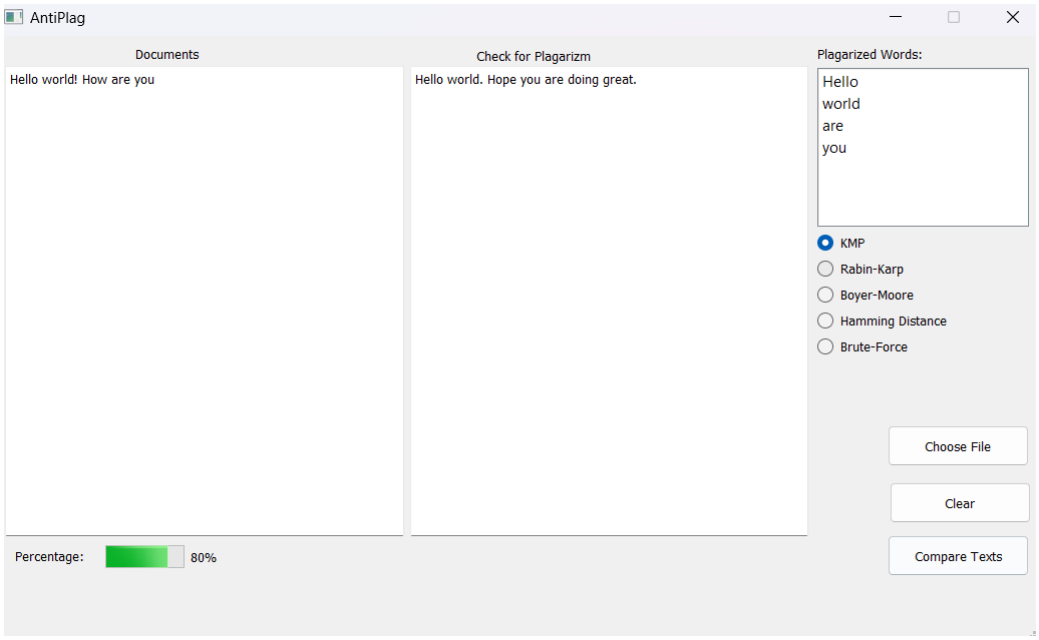


This example uses the KMP search algorithm to check for plagiarism. On both sides of the comparative boxes. It was entered by the user's keyboard. Syntax such as ‘,’ ‘.’ ‘;’ and ‘/n’

are not counted as plagiarism. They are removed before they are checked for plagiarism using a function that will be further discussed later on in this paper. Furthermore, this example shows user input through the keyboard. However, the user may also input data from a file on their desktop.

Experimental Results

The main aim was to show the percentage of plagiarism to a user. It would also be a method of comparison to see which algorithm could be the most accurate when it comes to this calculation. Following that the user would pick the algorithm. The common function that is used in all the algorithms is to split the words. Since all of the string-matching algorithms would loop around a string character by character. This would cause an inaccuracy in the percentage and when displaying the words. Therefore, all of the characters are split into words and added int a vector. Which is then used to compare between a different vectors. One is called the text and the other is the pattern. To check for similarities. If there is a similarity it will increment a counter. Later on, this counter is used to calculate the percentage over the total size of the text vector. And similar words are added to another vector which is output in the top right corner of the interface. Here is an example of the results from the user's input:



Analysis of the algorithm

File compilation:

File_compilation.h:

```
class File_compilation
{
public:
QVector<QString> splitStringIntoWords(QString str);
  QVector<int> computeLPS(QString pattern);
   int kmpSearch(QString pattern, QString text);
   int rabinKarp(QString pat, QString txt);
void badCharHeuristic(QVector<QString> str, int size,int badchar[NO_OF_CHARS]);
 double boyerMoore(QString pat, QString txt);
 void search(QString pattern, QString text, int &percentage);
int hammingDistance(QString text, QString pattern);
   QStringList UniqueWords;
   int percentageMatch = 0;
};
```

The "File_compilation" class contains all of the main algorithms used by this user interface. The *SplitStringIntoWord(string str)* function. This function receives an input of type string, also known as QString on QT. It would then return a vector. The second function is *computeLPS(string pattern)* which takes a pattern string and returns a vector containing the longest proper prefix that is also a suffix for each position in the pattern. The following function is *kmpSearch(string pattern, string text)*, which takes a pattern and text string and uses the Knuth-Morris-Pratt (KMP) algorithm to search for the pattern within the text. The following function is *rabinKarp(string pat, string txt)* which takes a pattern and text string and uses the Rabin-Karp algorithm to search for the pattern within the text. The fifth function is *badCharHeuristic(vector<string str,int size,int badchar[NO_OF_CHARS]),* which takes a string and its size and computes the bad character heuristic for use in the Boyer-Moore algorithm for string search. The final algorithm used is the *bruteForceStringMatch(string pattern, string text),* it would receive input from the user interface, then this would be inputted into a vector to be used for a for loop and allow comparison between both vectors. It mainly uses flags. Finally, the file_compilation class contains two main attributes. First is the QStringList UniqueWords, which acts like a vector to input all of the words that are similar in both vectors to be outputted later on. The other is percentageMatch, which is of type int. It would be found in all of the algorithms to calculate the percentage so that if the radio button is pressed it would use that one variable to output the percentage on a commonly used output bar.

QVector<QString> splitStringIntoWords(QString str): This function contains three variables the first is *QVector<QChars> unwantedChars,* which contains the list of characters that should be removed before the sentences are checked for plagiarism. The second is *QVector<QString> words.* It is the main vector that is used to push all of the words formed into it. First, a for loop is formed depending on the size of the text or pattern. Furthermore, an if statement is created to check if there are spaces for any unwanted carters specified. If it is neither an empty space of an unwanted character it is input into the third variable known as the currentWord of type string. This will then be pushed into the vector words. The currentWord variable will constantly be cleared to accept other words. After the currentWord is empty and the loop has ended the function will return the vector words.

QVector<int> computeLPS(QString pattern): It returns a vector of type int which represents the longest prefix and suffix values of the input pattern string. The function would first initialize the int variable “m” to the length of the pattern string inputted by the user. This would be used to create a vector of type int known as “lps” of size “m”. The function would then calculate the LPS values of the input pattern string using the KMP algorithm. This algorithm would initially iterate character by character. If the current character would match the previous one then the LPS value would be incremented. However, if the characters do not match then the LPS value is updated using the value of the previous character in the pattern.

int kmpSearch(QString pattern, QString text): The KMP search algorithm used for this project was slightly altered. Instead of checking character by character in the algorithm, it is checking word by word using the splitStringIntoWords that was referred to in the beginning. After forming two new vectors of type string one known as patternWords, and the other as textWords, there will

also be two variables of type int containing the sizes of these vectors. Furthermore, the vector UniqueWords will also be used. However, before it is used it would always be cleared to prevent any errors. The computerLPS function would first be called for, furthermore, the algorithm uses embedded for loops to check for plagiarism by checking the first index of the textWords with all of the indexes of the patternWords vector. If a similarity is found then the text is pushed into the UniqueWords vector and the counter wordMatchCounter is incremented. This is later used to calculate the percentage depending on the counter divided by the size of the text and multiplied by 100. The return value is an int and the percentage.

int RabinKarp (QString pat, QString txt): Similar to the KMP it also uses similar variables to wordMAtchCounter and the percentageMatch variables. It contains multiple other variables to be used to calculate the hash value of the pattern and the text. The first is q which contains the maximum value of an integer. After the strings have been split into words. The function would calculate the hash values. After doing so, the hash value of each word in the pattern and compares with the hash value of the corresponding words found in the text. If the same hash values have been found. Then it would increment the wordMatchCounter, and add the word to the UniqueWords. It would then calculate the percentage using the counter and the size of the words in a text to be inputted.

void badCharHeuristic(QVector<QString> str, int size, int badchar[NO_OF_CHARS]): This function takes three inputs, a vector of type string known as str, the integer size, and an integer array called badChar. It represents the bad character heuristic in a given set of characters. This technique is often used to speed up the string-matching process. It would skip characters in the text string that have already been matched therefore, it will reduce the number of

comparisons required to find a match. It would first initialize the badChar to -1 for all characters. It would then iterate on the str inputted by the user and store the last index of the last occur. If a character occurs multiple times the index of the last occurrence is stored. This function uses ASCII values for each character.

double boyerMoore(QString pat, QString txt): This function will take two variables of type string first being the text and the other being the pattern. It will return a double value as the percentage of plagiarism in the text. Similar to KMP and RabinKarp this function will also initialize the variables wordMatchCounter and percentageMatch to zero. It will then create an array to store the pattern and the text words using the function splitStringtoWords. The function will preprocess the input by calling the badCharHeuritsitc function. It will literate each word in the text and compare it to each word in the pattern. If the words are the same, it will push into the vector UniqueWords. The main method of this algorithm is that it will start by initializing the variable “s” to zero. It will then compare the pattern with the text. If the pattern is found, the variable wordMatchCounter will be incremented. Using this variable it will also calculate the percentage to be returned to the main.

Void search(QString pattern, QString text, int &percentage): The fourth string-matching algorithm used in this project is the Boyer-Moore algorithm. This is the only algorithm that is a void function and the only algorithm that searches for plagiarism character by character rather than word by word. The function would still split the pattern and the text string into words. The function will iterate through each word in the text string and compare it with the pattern string. If there is a match, then the variable “count” is incremented.

Furthermore, the word is pushed into the vector UniqueWords. It would finally calculate the percentage by dividing the count by the number of words in the text and multiplying it by 100. This is passed on by reference to be used in the main.

int hammingDistance(QString text, QString pattern): The final string matching algorithm used in this project is hamming Distance algorithm. The function would take two inputs first being the pattern and the second being the text. It would then return the percentage as an integer value. Similar to the other functions. The text in this function would be split into words to be compared using strings not characters. The function would then iterate through each text string and compare it with the pattern string, if it is found then the variable wordMatchCounter is incremented, which is used to calculate the percentage as explained before. This function, however, may not accurately represent the hamming distance algorithm due to the alterations that have been made which is one of the limitations in this project.

Overall these functions are used simultaneously to create a user interface that allows the user to check for plagiarism by checking which algorithm they would like to use. All the algorithms use the same attributes such as wordMatchCounter, percentageMatch, and UniqueWords to output the variables onto the interface.

MainWindow.h:

```
class MainWindow : public QMainWindow
{
   Q_OBJECT


public:
   MainWindow(QWidget *parent = nullptr);
   ~MainWindow();


private:
   Ui::MainWindow *ui;
   QStringListModel* m_listModel;
   int percentage = 0;
  int uniqueWordsCount;
 int  uniqueElementsCount;


private slots:
   void compareTexts();
   void on_pushButton_clicked();
   void on_pushButtonClear_clicked();
};
```

The attributes used in the mainWindow are the same as those used in the file_compilation. The main use for mainWindow.cpp is to connect the clicked buttons, radio buttons, and lineEdits to each other and to the algorithms.

Void compareTexts() contains the code that would take the information from the lineEdits and input it to a string to be used in the algorithms. It also contains five different radio buttons. Depending on which radio button is checked, it will call for its specific algorithm in the file_compilation.cpp to check for the percentage. In the end, the UniqueWords vector is called for to be displayed on the screen. And the percentage is added to a bar.

void pushButton_clicked(): it is the button used to open the file on the browser and to input it onto the left LineEdit.

void on_pushButtonClear_clicked(): It is the button that is used to clear all the vectors, line edits, bars, and variables to ensure that if the interface wants to be used again it will not overwrite the previous values and create an error.

Conclusion

Finally, despite having the same goal in mind, different string-matching algorithms are constructed differently. This tool uses string matching to check or find plagiarism. This study used a variety of tests and experiments to determine which algorithm is best accurate in detecting plagiarism. After doing all of these tests, it was determined that each algorithm produced results with an identical overall proportion of plagiarism. We do believe that as more and deeper searches are performed, the results may alter even though the findings were the same and there were no differences in the accuracy across the algorithms. Therefore, even though there are algorithms such as the brute-force algorithm and the Boyer-Moore algorithm that were slightly less accurate in showing the percentage of plagiarism compared to the other algorithms, more research is required to determine which algorithm would be the most accurate.

References

Al-Khamaiseh, K., & ALShagarin, S. (2014). A Survey of String Matching Algorithms. Journal of Engineering Research and Applications, 4(7), pp.144-156. https://www.researchgate.net/profile/Shadi-Alshagarin/publication/279442892_A_Survey_of_String_Matching_Algorithms/links/5592883108ae49b1fd6f0a67/A-Survey-of-String-Matching-Algorithms.pdf

GeekforGeeks. (2011a, April 3). *KMP Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

GeekforGeeks. (2011b, May 18). *Rabin-Karp Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

GeekforGeeks. (2012, May 26). *Boyer Moore Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

GeeksforGeeks. (2011, April 1). *Naive algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

GeeksforGeeks. (2018, October 4). *Hamming distance between two Integers*. GeeksforGeeks. https://www.geeksforgeeks.org/hamming-distance-between-two-integers/

JOKINEN, P., TARHIO, J., & UKKONEN, E. (1996). A Comparison of Approximate String Matching Algorithms. Software: Practice and Experience, 26(12), 1439–1458. https://doi.org/10.1002/(sici)1097-024x(199612)26:12%3C1439::aid-spe71%3E3.0.co;2-1

Appendix

File compilation

```
#include "file_compilation.h"
#include <regex>
#include <iostream>
#include <set>
#include <iostream>
#include <string>
#include <vector>
#include<fstream>
#include "document.h"
#include <QFile>
#include <QTextStream>
#include <QVector>
#include <QString>
#include <QDebug>
#include <QDir>
using namespace std;

QVector<QString> File_compilation::splitStringIntoWords(QString str) {
   QVector<QChar> unwantedChars = {',', '\n', '.', ';', ':', '!'};
   QVector<QString> words;
   QString currentWord;
   for (int i = 0; i < str.length(); i++) {
      QChar currentChar = str[i];
      if (currentChar.isSpace() || unwantedChars.contains(currentChar)) {
         if (!currentWord.isEmpty()) {
            words.push_back(currentWord);
            currentWord.clear();
         }
      } else {
         currentWord.append(currentChar);
      }
   }
   if (!currentWord.isEmpty()) {
      words.push_back(currentWord);
   }
   return words;
}
   // function to calculate Hamming distance
   int File_compilation::hammingDistance(QString text, QString pattern)
   {
      QVector<QString> patternWords = splitStringIntoWords(pattern);
```

```
    QVector<QString> textWords = splitStringIntoWords(text);
    double n = textWords.size();
    double m = patternWords.size();
    UniqueWords.clear();
    int wordMatchCounter = 0;
    for (int i = 0; i < n; i++) {
       int counter = 0;
       for (int j = 0; j < m; j++) {
          if (textWords[i] == patternWords[j]) {
           counter++;
         UniqueWords.push_back(textWords[i]);
          wordMatchCounter++;
          }
       }
       if (counter == n) {
          for (int j = 0; j < m; j++) {
          }
       }
    }
    percentageMatch = (wordMatchCounter / n) * 100.0;
    return percentageMatch;
}

// Fills lps[] for given pattern pat[0..M-1]
QVector<int> File_compilation::computeLPS(QString pattern) {
    int m = pattern.length();
    QVector<int> lps(m, 0);
    int len = 0;
    int i = 1;
    while (i < m) {
       if (pattern[i] == pattern[len]) {
          len++;
          lps[i] = len;
          i++;
       }
       else {
          if (len != 0) {
             len = lps[len - 1];
          }
          else {
             lps[i] = 0;
             i++;
          }
       }
    }
    return lps;
```

```
    }
    int File_compilation::kmpSearch(QString pattern, QString text) {
        QVector<QString> patternWords = splitStringIntoWords(pattern);
        QVector<QString> textWords = splitStringIntoWords(text);
        int n = textWords.size();
        int m = patternWords.size();
     //   int percentageMatch=0;
        UniqueWords.clear();
        int wordMatchCounter = 0;
  //     int i=0, j=0;
       QVector<int> lps = computeLPS(pattern);
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
            if (textWords[i] == patternWords[j]) {
                //QString z = text.mid(i, 1);
                UniqueWords.push_back(textWords[i]);
                wordMatchCounter++;
            }
            }
        }
   /*       if (j != 0) {
              j = lps[j - 1];
            }
            else {
                i++;
            }
*/
        percentageMatch = ((double)wordMatchCounter/n) *100.0;
        return percentageMatch;
    }

    // d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
    txt -> text
    q -> A prime number
*/
#define d 256
int File_compilation::rabinKarp(QString pat, QString txt) {
    int wordMatchCounter = 0;
   // int percentageMatch = 0;
    int q = INT_MAX;
    QVector<QString> patWords = splitStringIntoWords(pat);
```

```
    QVector<QString> txtWords = splitStringIntoWords(txt);
    int M = patWords.size();
    int N = txtWords.size();
    int i = 0;// , j = 0;
    int j = 0;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;
    QString tempPat = "";
    QString tempTxt = "";

        for (i = 0; i < M - 1; i++) {
            h = (h * d) % q;
        }
    for (i = 0; i < N ; i++)
    {
        p = 0;
        t = 0;
        for (j = 0; j < M; j++)
        {
            tempPat.clear();
            tempTxt.clear();
            p = 0;
            t = 0;
            p = (d * p + hash<QString>{}(patWords[j])) % q;
            tempPat = patWords[j];
            t = (d * t + hash<QString>{}(txtWords[i])) % q;
            tempTxt = txtWords[i];
            if (p == t)
            {
                  wordMatchCounter++;
                  UniqueWords.push_back(txtWords[i]);
            }
        }
    }
    percentageMatch = (wordMatchCounter / (double)N) * 100.0;
    return percentageMatch;
}


#define NO_OF_CHARS 256
void File_compilation::badCharHeuristic(QVector<QString> str, int size,int
badchar[NO_OF_CHARS])
{
    int i;
```

```
    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int)str[i][0].unicode()] = i;
}
/* A pattern searching function that uses Bad
Character Heuristic of Boyer Moore Algorithm */
double File_compilation ::boyerMoore(QString pat, QString text)
{

    QVector<QString> patternWords = splitStringIntoWords(pat);
    QVector<QString> textWords = splitStringIntoWords(text);
    double wordMatchCounter = 0.0;
    int m = textWords.size();
    int n = patternWords.size();
    int badchar[NO_OF_CHARS];
    UniqueWords.clear();
    /* Fill the bad character array by calling
    the preprocessing function badCharHeuristic()
    for given pattern */
    badCharHeuristic(textWords, m, badchar);
    int s = 0; // s is shift of the pattern with
          for (int i = 0; i < m; i++) {
             for (int k = 0; k < n; k++) {
                if (textWords[i] == patternWords[k])
                   UniqueWords.push_back(textWords[s + i]);
             }
          }
    // respect to text
    while (s <= (n - m))
    {
        int j = m - 1;
        while (j >= 0 && textWords[j] == patternWords[s + j])
            j--;

        if (j < 0)
        {
            s += (s + m < n) ? m - badchar[patternWords[s + m][0].unicode()] : 1;
        }

        else
            s += max(1, j - badchar[patternWords[s + j][0].unicode()]);
```

```
            wordMatchCounter++;
        }
        percentageMatch = ((double)wordMatchCounter / m)*100.0;
        return percentageMatch;
}


void File_compilation :: search(QString pat, QString txt, int &percentage) {
        QVector<QString> patWords = splitStringIntoWords(pat);
        QVector<QString> txtWords =splitStringIntoWords(txt);
        UniqueWords.clear();
        int N = txtWords.size();
        int M = patWords.size();

        int count = 0;

        for (int i = 0; i <= N - M; i++) {
            int j;
            for (j = 0; j < M; j++) {
                if (txtWords[i + j] != patWords[j]) {
                    break;
                }
            }
            if (j == M) {
              // cout << "Plagiarized part: ";
               for (int k = 0; k < M; k++) {
                 count++;
                   UniqueWords.push_back(txtWords[i + k]);

                }
             //   cout << endl;
            }
        }

        percentage = ((double)count /N)*100.0;

      //  cout << "Plagiarism percentage: " << percentage << "%" << endl;
}
```

Main Window

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <regex>
#include <iostream>
#include <set>
#include <QString>
#include <QFileDialog>
```

```
#include <QMessageBox>
#include<QDebug>
#include <iostream>
#include <fstream>
#include <string>
#include "file_compilation.h"

using namespace std;

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
   , ui(new Ui::MainWindow)
{
   ui->setupUi(this);
   m_listModel = new QStringListModel(this);
   ui->listOfWords->setModel(m_listModel);
   connect(ui->btCompare, SIGNAL(released()), this, SLOT(compareTexts()));

}

MainWindow::~MainWindow()
{
   delete m_listModel;
   delete ui;
}
int percentageMatch;
File_compilation run;
void MainWindow::compareTexts()
{
     QString text1 = ui->textEdit1->toPlainText().simplified();
     QString text2 = ui->textEdit2->toPlainText().simplified();
   if(ui->radioButton->isChecked()){
    percentageMatch = run.kmpSearch(text2, text1);
   }
   else if(ui->radioButton2->isChecked()){
        percentageMatch = run.rabinKarp(text2,text1);
   }
   else if(ui->radioButton3->isChecked()){
    percentageMatch= run.boyerMoore(text2, text1);
   }
   else if(ui->radioButton4->isChecked()){
     run.search(text2,text1,percentageMatch);
   }
    else if(ui->radioButton5->isChecked()){
      percentageMatch= run.hammingDistance(text2,text1);
   }
```

```
  m_listModel->setStringList(run.UniqueWords);
  ui->coincidenceBar->setValue(percentageMatch);
}

void MainWindow::on_pushButton_clicked()
{
   QString fileName = QFileDialog::getOpenFileName(this,
      tr("Open File"), "", tr("Text Files (*.txt)"));
   if (fileName.isEmpty()) {
      return;
   }

   QFile file(fileName);
   if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
      QMessageBox::warning(this, tr("Error"), tr("Could not open file."));
      return;
   }

   QTextStream in(&file);
   ui->textEdit1->setText(in.readAll());
   file.close();
}

void MainWindow::on_pushButtonClear_clicked()
{
   ui->textEdit1->clear();
   ui->textEdit2->clear();
   ui->coincidenceBar->setValue(0);
   m_listModel->setStringList(QStringList());
  // percentage=0;
   uniqueWordsCount = 0;
   uniqueElementsCount=0;
  run.UniqueWords.clear();
  percentageMatch=0;
}
```