Simple Plagiarism Detection Utility using String Matching

Muhammad Azzazy, Ramy Badras,
Shahd Elmahallawy, Youssif Abuzeid
Department of Computer Science and Engineering, AUC

Abstract: To preserve academic integrity, one has to be able to detect plagiarism in papers, assignments, exams and more. This is the topic of this project, a simple plagiarism detector that uses string matching to detect similarities between different documents. This project report should include the methodologies of plagiarism detection and the algorithms used for string matching. An analysis of the data used to achieve experimental results, the test run results and a comparison between all different algorithms used. A complete explanation of the program used to create the test runs is also included among the smaller auxiliary functions used to make the test runs easier and equally fair to the different algorithms.

Keywords: Plagiarism detection, Hamming Distance, KMP, Booyer Moore, Time complexity, Space complexity.

1 Introduction

Plagiarism has been a serious problem that affects academic integrity in universities and schools. Manual plagiarism detection is a time-consuming process, and it is not efficient at all. Instead, universities and research magazines depend on computers to detect plagiarism. Plagiarism detection tools date back to the 20th century. The building blocks of plagiarism detection tools are the string matching algorithms. There are various string matching algorithms. However, we will only use four of them in this project which are Hamming distance, Knuth–Morris–Pratt (KMP), Rabin-Karp, and Boyer-Moore algorithms. All of these methods are concerned with string matching, but they do it in different approaches. The difference in their methods makes them different in efficiency, time, and space complexity. All of these algorithms accept two main inputs, which are the pattern or the sentence we want to detect and a text. Their output is whether the pattern exists or not and the number of occurrences. Our project accepts a possibly plagiarized file as an input, and it works with a corpus or a collection of files that the input file may have plagiarized from. The program will use the four algorithms in parallel to detect plagiarism. Then the program will store the plagiarized sentences along with the directory of the file they plagiarized from. Then, we will calculate the time each algorithm takes to check for plagiarism. Based on these results, we will do our analysis which will include a comparison between the four algorithms.

2 Problem Definition

Since we have stated that plagiarism detection is important we need to be able to complete our task in the most efficient way possible, which is the point of this project. For example, let's say we have hundreds of thousands of documents and are given one to detect any plagiarism it might contain from our already existing large number of documents. We need to be able to detect this plagiarism in a small amount of time and without using many resources because more than thousands of academic papers are being written every day. Having slow or non cost efficient algorithms for plagiarism detection is not very practical. Hence why this project simulates the real world, but only by a small amount of documents but not too small so that we can get average test run results that can give us an idea as to how these algorithms will perform when it comes to detecting plagiarism among thousands of documents.

3 Methodology

The target of our project is to check whether a file entered by the user is plagiarized from a set of other files which are collected in a folder called the corpus. Implemented classes in the project: At first, we divided the program into four main classes, which are named Document, PotentiallyPlagiarized-Document, Corpus, and program_run. A brief description of each class is given below:

At first, we divided the program into four main classes, which are the Document class, Possible Plagiarized Document class, the Corpus Class, and the program_run class. A brief description of each class is given below:

1. Document: This class stores the directory of a given document and contains a vector that stores the sentences of this document. There is a string variable called text that stores all of the sentences of the document in a single variable. Also, it has several functions. The first one is the function called populate_sentences which gets the sentences from the text file. The second function is called adjust_sentences which removes all unnecessary spaces at the beginning or at the end of the sentence. The last one is called fill_in_text, which collects all sentences in the text variable.

2. Possible Plagiarized Document: This class inherits from the class called Document. We created this class to know that it is the class of the document that we should check whether it is plagiarized or not. This distinction helped us to determine how to use the function in a suitable way.

3. Corpus: This class stores all the documents that exist in the corpus folder. Documents in the class named Corpus are of the class called Document. All of them are stored in a vector. The main function in this class is the initialize_docs function which initializes all the corpus documents and fills their text attribute. The other functions are the four string matching functions. Each of them takes a sentence and a text as an input and returns whether the sentence exists in this text or not.

4. Program_run: This class has Corpus and Potentially Plagiarized Docu-

ment as attributes. Also, it has four functions that check whether there is a match between the possibly plagiarized document and the documents in the corpus. Each one of the four functions corresponds to one of the four string matching documents. Also, there are four integer attributes that store the running time of each algorithm which helps us to compare the efficiency of the four string matching algorithms.

How the program detects plagiarism

1. The program starts by asking the user to enter the directory of the file, which is possibly plagiarized.

2. The program goes to the file that the user entered in its directory and then extracts its sentences and adjusts them.

3. After this, the program initializes the documents in the corpus by getting the sentences from them and then adding them to a single text variable.

4. Next, the program uses the four-string matching algorithms: Hamming distance, Boyer-Moore, KMP, Rabin-Karp. In a loop, each sentence in the document the user entered is checked across all the documents in the corpus. The pattern here is the sentence and the text compared with is the text of the documents in the corpus. If there is a match, the sentence is added to a map alongside the directory of the doc it was plagiarized from. This process is done with the four matching algorithms to compare their efficiency in terms of time, space, and accuracy in detecting plagiarism.

5. Finally, the list of the plagiarized sentences is displayed on the screen alongside the documents they were plagiarized from.

4 Specifications of Algorithms to be used

1. *HammingDistance*
   The hamming distance algorithm represents the most straightforward string matching algorithm. It sees every character as a possible start for the pattern we are searching for. Starting from this character, it calculates the number of matches between the text and the pattern. If the number of matches equals the size of the pattern. Then it returns true. Otherwise, it returns false. **Algorithm** $HammingDistance(text, pattern)$

```
    for $i \leftarrow 0$ to text.size do
    $Counter \leftarrow 0$
    $K \leftarrow i$
    for $j \leftarrow 0$ to pattern.size do
      if $pattern[j] = text[k]$
        $Counter \leftarrow Counter + 1$
      if $Counter = pattern.size$
        return true
    return false
```

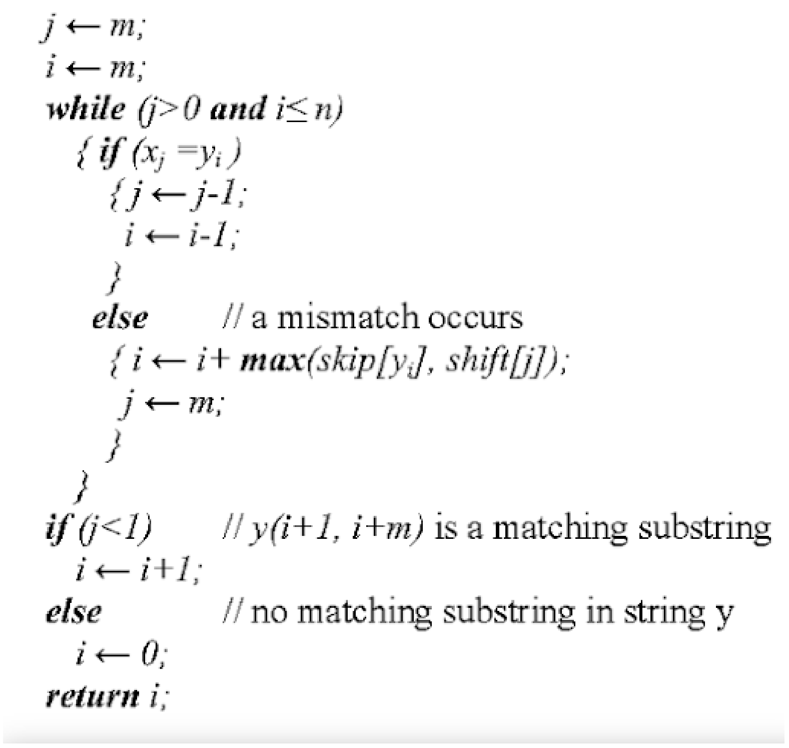
   The time complexity is $O(n*m)$, and the space complexity is $O(n+m)$ where n is the size of the text and m is the size of the pattern [1].

2. *KMP*
   The advantages of the KMP algorithm is that it keeps track of the mismatches and stores their positions to reduce the time needed for the search.
   ALGORITHM $KMP(text, pattern)$

```
    $PreifxAarray \leftarrow ComputeArray(pattern)$
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < text.size$ do
      while $j >= 0$ and $text[i]! = pattern[j]$
        $j \leftarrow PrefixArray[j]$
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
      if $j = pattern.size$
        return true
```

   ComputeArray is another algorithm used in this program that has complexity of $O(m)$ which is the length of the pattern. Time complexity is $O(n)$ in the average and the worst case where n is the size of the text and m is the size of the pattern [2]

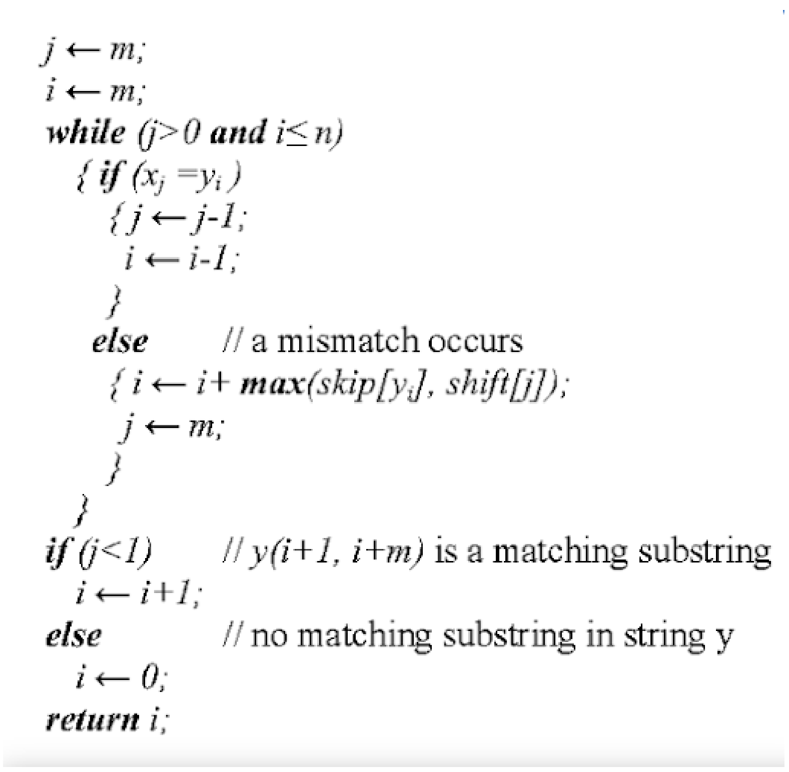
3. *RabinKarp*
   The Rabin-Karp algorithm use the principle of hashing to lower the time required to do the string matching process.



```
j ← m;
i ← m;
while (j>0 and i≤ n)
  { if ($x_j$ =$y_i$ )
     { j ← j-1;
       i ← i-1;
     }
    else        // a mismatch occurs
     { i ← i+ max(skip[$y_i$], shift[j]);
       j ← m;
     }
  }
if (j<1)        // y(i+1, i+m) is a matching substring
  i ← i+1;
else            // no matching substring in string y
  i ← 0;
return i;
```

   The time complexity of the algorithm is $O(n + m)$ in the average case and $O(n * m)$ in the worst case, and the space complexity is $O(n * m)$ where n is the size of the text and m is the size of the pattern. [3]

4. *BoyerMoore*
   The Boyer-Moore algorithm does pre-processing to lower the time complexity of the string matching process.



```
j ← m;
i ← m;
while (j>0 and i≤n)
  { if (x_j = y_i )
    { j ← j-1;
      i ← i-1;
    }
    else        // a mismatch occurs
    { i ← i+ max(skip[y_i], shift[j]);
      j ← m;
    }
  }
if (j<1)        // y(i+1, i+m) is a matching substring
  i ← i+1;
else            // no matching substring in string y
  i ← 0;
return i;
```

The average time complexity is $O(n)$, while the worst time complexity $O(n * m)$, and the space complexity is $O(n + m)$ where n and m are the sizes of the text and pattern respectively. [4]

5. *PopulateSentences*

This algorithm extracts the sentences from the text files. These sentences will be adjusted later and will be used as patterns for string matching.

ALGORITHM PopulateSentences(Directory)

```
    Open the file using the directory
    Iterate over the lines of the file
    $i \leftarrow 0$
    Loop over the sentences of the line
    if ($i = 0$ and flag)
        sentences.pushback($LastSentence + +sentence$)
        $LastSentence \leftarrow sentence$
    else
        sentences.pushback(sentence)
        $LastSentence \leftarrow sentence$
        $i \leftarrow i + 1$
    **if** last character is not a full stop
        $flag \leftarrow true$
        Remove the last sentence.
```

6. *AdjustSentences*

   This algorithm removes unnecessary spaces from sentences.It guarantees the efficiency of comparison process. Extra spaces might make plagiarized sentences undetectable.

```
**ALGORITHM** $AdjustSentences()$
Iterate over the sentences of the document
$temp \leftarrow$
$flag \leftarrow false$
for $i \leftarrow 0$ to the size of the sentence
**if** sentences[i, j] ' ' and flag = false
        $flag \leftarrow true$
**if** (flag)
        $temp \leftarrow sentences[i, j]$
        Make the current sentence = temp
```

   Time complexity is $O(l + s)$ and space complexity is $O(s)$ where l is the number of lines and s is the number of sentences in each line.

7. *FillInText*

   This algorithm adds all of the sentences of a given file in a single string variable. This is important to make it the string used for the matching process.

```
**ALGORITHM** $FillInText()$
    Apply the PopulateSentences and AdjustSentences algorithms
    $text \leftarrow$""
    **for** $i \leftarrow 0$ **to** $sentences.size()$ **do**
        $text \leftarrow sentences[i]$+" "
```

   The time complexity of the FillInText algorithm is $O(s)$, and the space complexity of the FillInText algorithm is $O(s)$ where s is the number of sentences.

8. *CheckMatchesKMP*

   This algorithm checks all possible matches in all patterns in the entered file across the corpus using the KMP algorithm.

```
ALGORITHM $CheckMatchesKMP()$
    for $i \leftarrow 0$ to sentences.size) do
        for $j \leftarrow 0$ to the number of documents in the corpus do
            if C.KMP(doc.sentences[i] , C.docs[j]
                Add sentences[i] to the list of plagiarized sentences.
                Set the directory of the document the sentence is
                plagiarized from to C.docs[j].directory.
```

Time complexity of *CheckMatchesKMP* is $O(d * s * n)$, and the space complexity is $O(s)$ where d is the number of documents, s is the number of sentences, and n is the number of characters in each document.

9. *CheckMatchesRabinKarp*
   This algorithm checks all possible matches in all patterns in the entered file across the corpus using the Rabin-Karp algorithm. **ALGORITHM** $CheckMatchesRabinKarp()$

```
    for $i \leftarrow 0$ to $doc.sentences.size$ do
        for $j \leftarrow 0$ to the number of documents in the corpus do
            if C.RabinKarp(doc.sentences[i], C.docs[j])
                Add sentences[i] to the list of plagiarized sentences.
                Set the directory of the document the sentence is
                plagiarized from to C.docs[j].directory.
```

   The time complexity is $O(d * s * (n + m))$ in the average case and $O(d * s * n * m)$ and the space complexity is $O(s)$ in the worst case, where d is the number of documents, s is the number of sentences, n is the number of characters in each document, and m is the number of characters in each sentence.

10. *CheckMatchesBoyer*
    This algorithm checks all possible matches in all patterns in the entered file across the corpus using the Boyer-Moore algorithm.

```
ALGORITHM $CheckMatchesBoyerMoore()$
    for $i \leftarrow 0$ to doc.sentences.size do
        for $j \leftarrow 0$ to the number of documents in the corpus do
            if C.BoyerMoore(doc.sentences[i] , C.docs[j])
                Add sentences[i] to the list of plagiarized sentences
                Set the directory of the document the sentence is
                plagiarized
        from to C.docs[j].directory
```

The time complexity is $O(d*s*n)$ in the average case and $O(d*s*n*m)$ in the worst case, space complexity is $O(S)$ where d is the number of documents, s is the number of sentences, n is the number of characters in each document, and m is the number of characters in each sentence.

11. *CheckMatchesHamming* This algorithm checks all possible matches in all patterns in the entered file across the corpus using the Hamming distance algorithm.

```
ALGORITHM CheckMatchesHamming()
    for $i \leftarrow 0$ to doc.sentences.size do
    for $j \leftarrow 0$ to the number of documents in the corpus do
      if C.Hamming(doc.sentences[i] , C.docs[j]
        Add sentences[i] to the list of plagiarized sentences
        Set the directory of the document the sentence is plagiarized
        from to C.docs[j].directory
```

The time complexity is $O(d*s*n*m)$ in all of the cases, and the space complexity $O(s)$ where d is the number of documents, s is the number of sentences, n is the number of characters in each document, and m is the number of characters in each sentence.

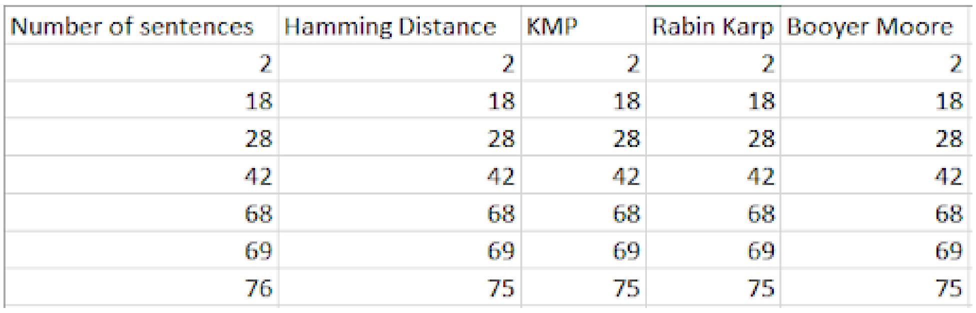
5 Data Specifications

Basically, there are two types of input data to the program:

1. The directory of the file that the user wants to detect plagiarism in.
2. The files in the corpus. These files the program is responsible for loading them at the beginning of the program. The files in the corpus are in the same folder of the program. However, the user might want to edit them in the way that fits his need.

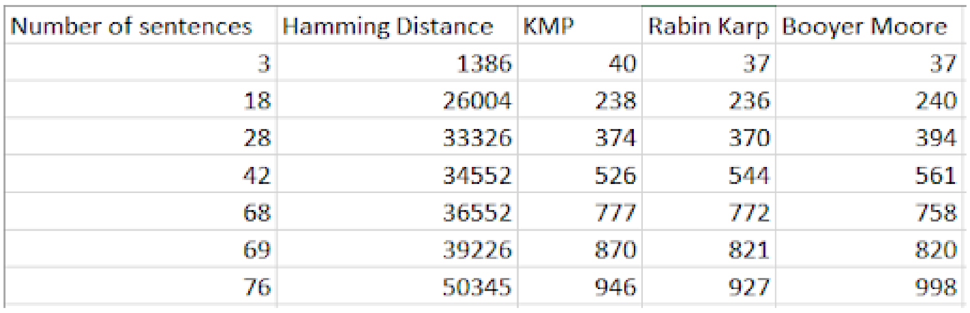
6 Experimental Results

1. Accuracy of algorithms test. At first, we tested the accuracy of the four algorithms in detecting plagiarism. To do this, we ran the program against a test file which included a predetermined number of plagiarized sentences. Every run, we changed the number of sentences to yield more meaningful results. The results of this experiment are mentioned in the table below.

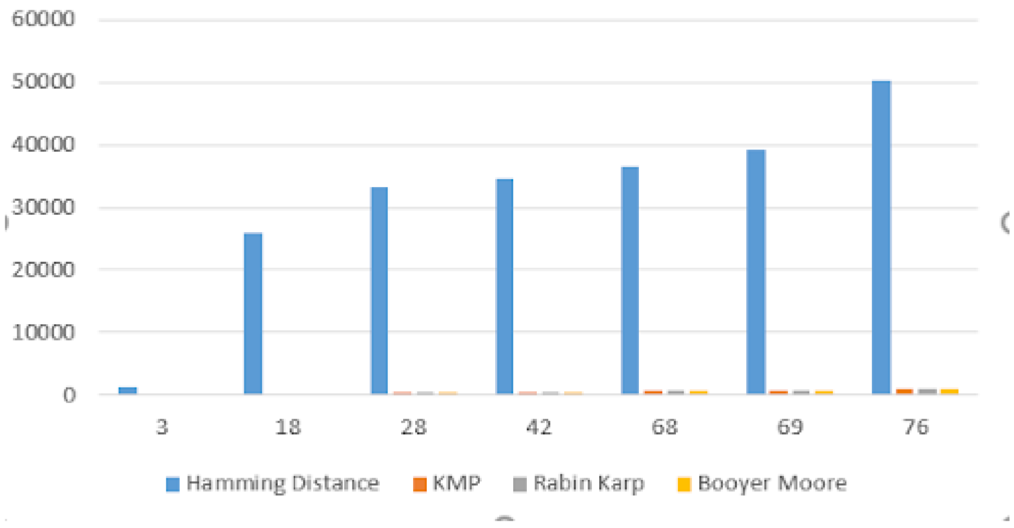


Number of sentences	Hamming Distance	KMP	Rabin Karp	Booyer Moore
2	2	2	2	2
18	18	18	18	18
28	28	28	28	28
42	42	42	42	42
68	68	68	68	68
69	69	69	69	69
76	75	75	75	75

2. Time complexity of Algorithms test. In this test, we ran the program against sample files of different sizes and we measured the time taken by each algorithm to detect plagiarism in this file in microseconds. Results of this test are in the table and the graph below.



Number of sentences	Hamming Distance	KMP	Rabin Karp	Booyer Moore
3	1386	40	37	37
18	26004	238	236	240
28	33326	374	370	394
42	34552	526	544	561
68	36552	777	772	758
69	39226	870	821	820
76	50345	946	927	998


60000
50000
40000
30000
20000
10000
0
3
18
28
42
68
69
76
Hamming Distance
KMP
Rabin Karp
Booyer Moore

7 Analysis and Critique

The efficiency of the program in plagiarism detection is acceptable as there is a small margin of error when the tested document has a high number of sentences. For instance, the program succeeded in detecting 75 out of 76 sentences. We can say that the undetected sentence is due to the format of the documents. When talking about time complexity, all the algorithms except the hamming distance algorithm finished the task in less than 1 second. We can say that it is not the best thing, but it is acceptable. According to the experimental results above, it is clear that all the algorithms have nearly the same efficiency in detecting plagiarism. However, it is clear that the hamming distance algorithm took much more time compared to the other algorithms. The reason is that the hamming distance is a brute force algorithm that makes every possible comparison which makes the time much higher. In addition, from the results of the time test, we can deduce that the experimental results are in agreement with the theoretical time complexities for the four algorithms. Only the hamming distance algorithm had a maximum time complexity of $O(d * s * n * m)$ where d is the number of documents, s is the number of sentences, n is the number of characters in each document, and m is the number of characters in each sentence. From the code and pseudocode of each algorithm, we can distinguish between their types. Clearly, the hamming distance algorithm is a brute force. On the other hand, the KMP algorithm is a dynamic programming algorithm as it uses extra space to yield more optimal results. Furthermore, the Rabin-Karp algorithm is a special type of brute force algorithm as it makes comparisons but skips some of the comparisons when possible. In other words, the Rabin-Karp algorithm is an optimized version of the hamming distance algorithm. Finally, the Boyer-Moore algorithm lowers its complexity by pre-processing the text. Much of the time used by the program is because the algorithms search for every document in the corpus. However, this process might be optimized by selecting some of the documents that have the same topic as the paper. This could be done by getting some distinctive keywords from each document and comparing them with the keywords of the tested files. In addition, good formatting of the text will ensure the high accuracy of the plagiarism detection process. For instance, extra spaces between words in the words of the same sentences might make a plagiarized sentence undetected by the program. Finally, we recommend that either of the KMP, Rabin-Karp, or

Boyer-Moore algorithms be used in string matching tasks as they have the same efficiency but with lower time complexity than the hamming distance algorithm.

8 Conclusions

We can say that our program is efficient to a good extent in terms of detecting plagiarism. In addition, according to the comparison we did between the four algorithms, the expected theoretical time complexity fits with our experimental results. The KMP, Boyer-Moore, Rabin-Karp algorithms have similar time complexities. Also, the hamming distance algorithm has higher complexity than the other three. Our recommendations are to find a method to select some files in the corpus to check for plagiarism in. This would reduce complexity as not all files will be used. Also, we recommend to depend mainly on KMP, Boyer-Moore, or Rabin-Karp algorithms for plagiarism detection rather than the Hamming distance algorithm.

Acknowledgments

At the end we thank God for helping us to produce this project. Also, we are grateful for the efforts of Professor Amr Goneid for his clear instructions about the project and its topic. Furthermore, we appreciate the support of his graduate and undergraduate teaching assistants all over the semester and their recommendations about the project.

References

[1] “Hamming Distance between two strings,” GeeksforGeeks, Feb. 26, 2017. https://www.geeksforgeeks.org/hamming-distance-two-strings/ (accessed May 14, 2022).

[2] “KMP Algorithm for Pattern Searching,” GeeksforGeeks, Apr. 03, 2011. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[3] “Rabin Karp Algorithm for Pattern Searching,” GeeksforGeeks, May 18, 2011. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/ (accessed May 14, 2022).

[4] “Boyer Moore Algorithm for Pattern Searching,” GeeksforGeeks, May 26, 2012. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Appendix:

Below is a list of the major algorithms used in the project.

1. Hamming distance algorithm.

```
bool Hamming_distance(string pattern, string text) {

        for (int i = 0; i < text.size(); i++) {
                int counter = 0;
                int k = i;
                for (int j = 0; j < pattern.size(); j++) {
                        if (pattern[j] == text[k]) {
                                counter++;
                                k++;
                        }
                }
                if (counter == pattern.size()) {
                        return true;
                }
        }
        return false;
}
```

2. KMP algorithm.

```
bool KMP(string pattern, string text) {
        int patLen = pattern.length();
        int textLen = text.length();
        int *lps = new int[patLen];
        lps[0] = 0;
        int len = 0;
        int i = 1;
        int index = -1;
        while (i < patLen){
                if (pattern[i] == pattern[len]) {
                        len++;
                        lps[i] = len;
                        i++;}
                else {
                        if (len != 0) {
                                len = lps[len - 1];}
                        else {
                                lps[i] = 0;
                                i++;}}}
        i = 0;
        int j = 0;
        while (i < textLen) {
                if (pattern[j] == text[i]) {
                        i++;
                        j++;}}
                if (j == patLen) {
                        index = i - j;
```

```
                    //cout << "Found pattern at: " << i - j << endl;
                    j = lps[j - 1];}
                else if (i < textLen && pattern[j] != text[i]) {
                    if (j != 0) {
                        j = lps[j - 1];}
                    else {
                        i++;
                    }
                }
            }
            if (index >= 0) {
                return true;
            }
            else {
                return false;
            }
        }
```

3. Rabin Karp algorithm.

```
        bool Corpus:: RabinKarpStringMatching(string pattern, string text)
        {
            int q = 101;
            int j;
            int m = pattern.length();
            int n = text.length();
            int hashcode = 0;
            int hashCode = 0;
            int h = 1;
            for (int i = 0; i < m - 1; i++)
            {
                h = (h * d) % q;
            }
            for (int i = 0; i < m; i++)
            {
                hashcode = (d * hashcode + pattern[i]) % q;
                hashCode = (d * hashCode + text[i]) % q;
            }
            for (int i = 0; i <= n - m; i++)
            {
                if (hashcode == hashCode)     {
                    bool flag = true;
                    for (j = 0; j < m; j++)
                    {
                        if (text[i + j] != pattern[j])
                        {
                            flag = false;
                            break;
                        }
                    }
                    if (j == m)
                    {
                        return true;
                    }
                }
                if (i < n - m)
                {
                    hashCode = (d * (hashCode - text[i] * h) + text[i + m]) % q;
                    if (hashCode < 0)
                    {
                        hashCode = (hashCode + q);
                    }
                }
            }
            return false;
        }
```

4. Booyer Moore Algorithm.

```
        bool Corpus::Boyer_Moore(string str, string sentence) {
            int m = sentence.size();
            int n = str.size();
            int frequency = 0;
            int badchar[2000];
            badCharHeuristic(sentence, m, badchar);
            int s = 0;
            while (s <= (n - m))
            {
                int j = m - 1;
                while (j >= 0 && sentence[j] == str[s + j])
                    j--;
                if (j < 0)
                {
                    frequency++;
                    s += (s + m < n) ? m - badchar[str[s + m]] : 1;
                }
                else
                    s += max(1, j - badchar[str[s + j]]);
            }
            if (frequency > 0) {
                return true;
            }
            else {
                return false;
            }
        }
```

5. Populate sentences algorithm.

```
        Void populate_sentences() {
            fstream fin;
            fin.open(directory, ios::in);
            string temp, line, sentence;
        string last_sentence;
        bool notdot = false;
        while (getline(fin, temp))
        {
                stringstream S(temp);
                int counter = 0;
                while (getline(S, sentence, '.'))
                {
                        if (counter == 0 && notdot) {
                                sentences.push_back(last_sentence + " " +sentence);
                                last_sentence = sentence;
                        }
                        else {
                                sentences.push_back(sentence);
                                last_sentence = sentence;
                                counter++;
                        }
                }
                if (temp.size() > 0) {
                        if (temp[temp.size() - 1] != '.') {
                                notdot = true;
                                sentences.pop_back();
                        }
                }
        }
}
```

6. Adjust sentences algorithm.

```
void adjust_sentences() {
        for (int i = 0; i < sentences.size(); i++) {
                string temp = "";
                bool flag = false;
                for (int j = 0; j < sentences[i].size(); j++) {
                        if (sentences[i][j] != ' ' && !flag) {
                                flag = true;
                        }
                        if (flag) {
                                temp += sentences[i][j];
                        }
                }
                sentences[i] = temp;
        }
}
```

7. Fill in text algorithm.

```
void fill_in_text() {
        populate_sentences();
        adjust_sentences();
        text = "";
        for (int i = 0; i < sentences.size(); i++) {
                text += sentences[i] + " ";
        }
}
```

8. Check match hamming algorithm.

```
void check_match_hamming() {
        auto started = std::chrono::steady_clock::now();
        for (int i = 0; i < doc.sentences.size(); i++) {
                for (int j = 0; j < C.docs.size(); j++) {
                        if (C.Hamming_distance(doc.sentences[i], C.docs[j].text)) {
                                plagiarized_sentence S;
                                S.content = doc.sentences[i];
                                S.doc_dir = C.docs[j].directory;
                                results_Hamming[S.doc_dir]++;
                                Hamming_result.push_back(S);
                        }
                }
        }
        auto done = std::chrono::steady_clock::now();
        time_Hamming = std::chrono::duration_cast<std::chrono::milliseconds>(done -
started).count();
}
```

9. Check match kmp algorithm.

```
void check_match_kmp() {
auto started = std::chrono::steady_clock::now();
for (int i = 0; i < doc.sentences.size(); i++) {
        for (int j = 0; j < C.docs.size(); j++) {
                if (C.KMP(doc.sentences[i], C.docs[j].text)) {
                        plagiarized_sentence S;
                        S.content = doc.sentences[i];
                        S.doc_dir = C.docs[j].directory;
                        results_KMP[S.doc_dir]++;
                        KMP_result.push_back(S);
                }
        }
}
auto done = std::chrono::steady_clock::now();
time_KMP = std::chrono::duration_cast<std::chrono::milliseconds>(done -
started).count();
}
```

10. Check match rabin algorithm.

```
void Program_run::check_match_rabin() {
        auto started = std::chrono::steady_clock::now();
        for (int i = 0; i < doc.sentences.size(); i++) {
                for (int j = 0; j < C.docs.size(); j++) {
                        if (C.KMP(doc.sentences[i], C.docs[j].text)) {
                                plagiarized_sentence S;
                                S.content = doc.sentences[i];
                                S.doc_dir = C.docs[j].directory;
                                results_Rabin[S.doc_dir]++;
                                Rabin_result.push_back(S);
                        }
                }
        }
        auto done = std::chrono::steady_clock::now();
        time_Rabin = std::chrono::duration_cast<std::chrono::milliseconds>(done -
started).count();
}
```

11. Check match booyer algorithm.

```
void Program_run::check_match_Booyer() {
        auto started = std::chrono::steady_clock::now();
        for (int i = 0; i < doc.sentences.size(); i++) {
                for (int j = 0; j < C.docs.size(); j++) {
                        if (C.KMP(doc.sentences[i], C.docs[j].text)) {
                                plagiarized_sentence S;
                                S.content = doc.sentences[i];
                                S.doc_dir = C.docs[j].directory;
                                results_Booyer[S.doc_dir]++;
                                Booyer_result.push_back(S);
                        }
                }
        }
        auto done = std::chrono::steady_clock::now();
        time_Booyer = std::chrono::duration_cast<std::chrono::milliseconds>(done -
started).count();
}
```