The Application of String-Searching Algorithm and Dynamic Programming in Multilingual Plagiarism Checker

Ignatius Timothy Manullang - 13517044
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
ignatiustimothymanullang@gmail.com

***Abstract*—Translation Plagiarism is a new way of plagiarism in which many people are trying to do because some Plagiarism Checkers do not have the ability to detect plagiarism in more than 1 language. In order to counteract Translation Plagiarism, Mullingual Plagiarism checker, which can check plagiarism in many languages can be used. This paper will discuss about the implementation of String-Searching Algorithm and Minimum Edit Distance Dynamic Programming to Multilingual Plagiarism Checker, which can detect plagiarized texts translated in many languages.**

***Keywords*—translation plagiarism, string matching, dynamic programming, multilingual plagiarism checker**

I. Introduction

Today, many people have access to many media as resource for their work, including, but not limited to, books, internet, television, so on and so forth. It's easier than ever for people to just take those resources and claim it as their own.

Computers of this generation can execute processes, including, but not limited to, cut, which removes a selected data and save it to the clipboard for a later use, copy, which is saving a selected data to the clipboard, and paste, which is placing the data chosen from the clipboard into a document at the flashing cursor's location. By using cut, copy, and paste, many people have taken other people's works as their own.

Plagiarism is a way to take others' works as their own. It also involves using others' works without crediting the source. This is a very big issue in the world, since taking others' ideas and claiming as own work is a lie and not ethical.

There are many types of plagiarism. A modern-day concern is translation plagiarism. It is a new way of plagiarism that uses content and translating it to another language so that plagiarism checkers might not be able to detect it as plagiarized.

Plagiarism checker itself is an automatic plagiarism detection tool. It checks for similarities in a text, by using databases of articles, and if it detects that the text is similar to another text, then it will output that the text is a plagiarized text, in which the text actually came from other people's ideas and works.

Plagiarism checkers can detect an exact text or similarities of a text to another text, mostly in the same language. However, some plagiarism checkers will not be able to detect a translated text as plagiarized because those plagiarism checkers do not deal with situations when the text are translated to different languages. As a result, many people got away with translation plagiarism.

One way to detect plagiarism is by checking a text and compare it by using string matching algorithm. However, that can only detect if the text is exactly the same, which means it cannot detect when a text is copied into a translator and instantly translated to other languages.

Therefore, the solution that the author of this paper thought of, is by using database of translations, thesauruses and text database in order to detect translation plagiarized text.

II. Basic Theory

A. Pattern Matching

Pattern Matching is the action of checking whether a pattern exists in a data, or not. It is also commonly referred to as String Searching.
It is an important problem in Computer Science. Pattern matching is used to output search results based on a search query.

By definition, if we are given a text T, which is a string that has a length of n characters, and a pattern P, which is another string that has a length of m characters and is going to be searched in the text. The problem is that we have to locate the first location in which the text matches the pattern.
There are some common ways in which the problem can be solved, which are:

1. Brute-force String Searching Algorithm
2. Knuth-Morris-Pratt String Searching Algorithm
3. Boyer-Moore String Searching Algorithm

B. Brute-Force Algorithm

Brute-force algorithm, which is also commonly referred to as the naïve algorithm, is an algorithm that uses straightforward ways to solve problems. It is straightforward because it refers to the problem statement and the definition of concept which is involved in the problem. Brute-force algorithm also breaks down problems in a very simple, direct and obvious way.

In string searching, the brute-force algorithm will slide the pattern over the text one-by-one and check each position in the text T to see if the pattern P starts in that position. If the pattern P starts in that position, it will check the 2nd character in the pattern P (if it exists) with the next character in the text T, and so on, until every character in the pattern is checked and matches a part of the text. If the pattern P doesn't start in that position, or a character which is currently checked in the pattern P doesn't match the character which is also currently checked in the text T, the position for the pattern P is moved 1 character to the right. The string searching will continue until every character in the pattern is checked and matches a part of the text, or every character in the text T is checked.

Using the example before, the brute-force string algorithm will find the solution as follows:

Text : "The shimmering star"
Pattern: "ring"

	T	h	e		s	h	i	m	m	e	r	i	n	g		s	t	a	r
1	r	i	n	g															
2		r	i	n	g														
3			r	i	n	g													
4				r	i	n	g												
5					r	i	n	g											
6						r	i	n	g										
7							r	i	n	g									
8								r	i	n	g								
9									r	i	n	g							
10										r	i	n	g						
11											r	i	n	g					

The best case for this algorithm happens when the first character of the pattern P doesn't appear in the text T at all, which means the maximal number of character checking is n times (n being the number of characters that the text T has).
An example of this case is:

Text: "Best string case yyy"
Pattern: "yyy"

The time complexity for the best case of brute-force string matching algorithm is O(n).

The average case for this algorithm takes O(m+n) time complexity, in which m is the number of characters that the pattern P has, and n is the number of characters that the text T has.
An example of this case is:

Text: "Average case analysis"
Pattern: "Any"

The worst case for this algorithm happens when all the characters of the text T and pattern P are the same, which is shown below.

Text: "ZZZZZZZZZZZZZZZZZZZZZ";
Pattern: "ZZZ"

The worst case also happens when only the last character is different, which is shown below.

Text: "ZZZZZZZZZZZZZZZZZZZZY";
Pattern: "ZZY".

The time complexity for the worst case of brute-force string matching algorithm is O(m*(n-m+1)) = O(mn).
The brute-force algorithm in Java is shown below:

```
public static int brute(String text,String pattern) {
        int n = text.length(); // n is length of text
        int m = pattern.length(); // m is length of pattern
        int j;
        for(int i=0; i <= (n-m); i++) {
                j = 0;
                while ((j < m) && (text.charAt(i+j)==
                pattern.charAt(j)) ) {
                        j++;
                }
                if (j == m)
                        return i; // match at i
        }
        return -1; // no match
} // end of brute()
```

(Source: Slides on Pattern Matching by Dr. Andrew Davidson and updated by Dr. Rinaldi Munir)

C. Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt Algorithm, which was conceived by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. It searches the pattern P in the text T from left-to-right. However, it shifts the pattern P better than the brute force algorithm. Whenever the Knuth-Morris-Pratt Algorithm detects a mismatch, some of the characters in the text of the next window is already known. The algorithm finds the most we can shift the pattern to avoid wasteful comparisons, which is the largest proper prefix of P[0 .. j-1] that is also a suffix of P[1 .. j-1].
An example of this case is:

Text:	.	.	c	d	c	c	d	y	.	.	.	.	.
Pattern:			c	d	c	c	d	c					
						c	d	c	c	d	c		

Find the largest prefix (start) of "**cd**ccd" which is also the suffix(end) of "cdc**cd**". We find that the answer is "cd". The number of shifts is the length of pattern subtracted by the length of the largest prefix of pattern that matches the text which is also the suffix of pattern that matches the text. In this case, the number of shifts is 3 (since 5 - 2 = 3).

Knuth-Morris-Pratt Algorithm uses a Border Function in which the algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself. First, we initialize the variable j with the mismatch position in pattern P, initialize the variable k with the position before the mismatch (k = j - 1). The border function b(k) is the size of the largest prefix of P[0 .. k] that is also a suffix of P[1 .. k]. Another name for the border function of Knuth-Morris-Pratt algorithm is failure function. An example of the use of Knuth-Morris-Pratt Border Function is

Pattern: cdcccd
j = 012345

j	0	1	2	3	4	5

P[j]	c	d	c	c	d	c
k	-	0	1	2	3	4
b(k)	-	0	0	1	1	2

An example of the usage of Knuth-Morris-Pratt algorithm is:

Text: bcbdbbcbdbc
Pattern: bcbdbc

b	c	b	d	b	b	c	b	d	b	c
1	2	3	4	5	6					
b	c	b	d	b	c					
					7					
				b	c	b	d	b	c	
					8	9	10	11	12	13
					b	c	b	d	b	c

This means there are 13 comparisons until the word is found. Below is the result which is generated by the border function.

j	0	1	2	3	4	5
P[j]	0	0	1	0	1	0

Parts of the Knuth-Morris-Pratt algorithm has a time complexity of:

- Calculating the border function: O(m)
- String searching O(n)

Therefore, the Knuth-Morris-Pratt Algorithm has a time complexity of O(m+n), which is faster than brute-force (which has a time complexity of O(mn).

The Knuth-Morris-Pratt Algorithm in Java is shown below:

```
public static int kmpMatch(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int fail[] = computeFail(pattern);
        int i=0;
        int j=0;
        while (i < n) {
                if (pattern.charAt(j) == text.charAt(i)) {
                        if (j == m - 1)
                                return i - m + 1; // match
                        i++;
                        j++;
                } else if (j > 0)
                        j = fail[j-1];
                else
                        i++;
        }
        return -1; // no match
} // end of kmpMatch()

public static int[] computeFail(String pattern) {
        int fail[] = new int[pattern.length()];
        fail[0] = 0;
        int m = pattern.length();
        int j = 0;
        int i = 1;
        while (i < m) {
                if (pattern.charAt(j) == pattern.charAt(i)) {
                //j+1 chars match
                        fail[i] = j + 1;
                        i++;
                        j++;
                } else if (j > 0) // j follows matching prefix
                        j = fail[j-1];
                else { // no match
                        fail[i] = 0;
                        i++;
                }
        }
```

(Source: Slides on Pattern Matching by Dr. Andrew Davidson and updated by Dr. Rinaldi Munir)

D. Boyer-Moore Algorithm

The Boyer-Moore pattern matching algorithm is a pattern matching algorithm made by Robert S. Boyer and J Sthrother Moore. It involves two techniques, the looking-glass technique, which is done by finding the pattern P in the text T by checking with the pattern P from right to left, starting from the end of the text T, and the character-jump technique, which is done when a mismatch occurs at T[i] == x, in which the character in the pattern P[j] isn't the same as T[i]. There are 3 possible cases:

1. If the pattern P has x, then shift the pattern until x in the pattern and in the text T is parallel to each other.
2. If the pattern P has x, however if a shift right to the last occurrence isn't possible, then the pattern P is shifted right by 1 character.
3. If both previous cases didn't work, then jump pattern so that it passes x.

Boyer-Moore string matching algorithm also has a Last Occurrence Function L() to preprocess the pattern P and the alphabet A to map all the letters in A to integers.

An example of the usage of Boyer-Moore Algorithm is as follows:

Text: abacaabacab
Pattern: abacab

a	b	a	c	a	a	b	a	d	a	b	a	c	a	d
					1									
a	b	a	c	a	b									
				4	3	2								
	a	b	a	c	a	b								
							5		12	11	10	9	8	7
		a	b	a	c	a	b		a	b	a	c	a	d
								6						
			a	b	a	c	a	b						

There are 12 comparisons in total.

x	a	b	c	d
L(x)	4	5	3	-1

In an average case, shown above, the time complexity of Boyer-Moore algorithm is O(m + n), which is equal to the time complexity of Knutt-Morris-Pratt algorithm. However, the Boyer-Moore algorithm has a worst case running time of O(mn+A), which happens when the first character in the pattern P doesn't exist in text T and text T only contain the same character as the last characters in pattern P, since it will match from the last character until the first, and failing to match the text every time, and will continue to do so until the pattern goes to the first string.

Text: “zzzzzzzz”
Pattern: “yzzzzz”

z	z	z	z	z	z	z	z	z
6	5	4	3	2	1			
y	z	z	z	z	z			
	12	11	10	9	8	7		
	y	z	z	z	z	z		
		18	17	16	15	14	13	
		y	z	z	z	z	z	
			24	23	22	21	20	19
			y	z	z	z	z	z

The best case of Boyer-Moore algorithm is O(n/m), which happens when the algorithm found a character that appears in text and doesn’t exist in the pattern before that position. Then, because the character doesn’t exist in the pattern before that position, the algorithm shifts the pattern past to the position and then instantly gets a perfect match of a pattern.
An example of a case is:

Text: defgcababa
Pattern:ababa

d	e	f	g	c	a	b	a	b	a
				1					
a	b	a	b	a					
					6	5	4	3	2
					a	b	a	b	a

The Boyer-Moore Algorithm in Java is shown below:

```
public static int bmMatch(String text, String pattern) {
        int last[] = buildLast(pattern);
        int n = text.length();
        int m = pattern.length();
        int i = m-1;
        if (i > n-1) return -1; // no match if pattern is
                        // longer than text
        int j = m-1;
        do {
                if (pattern.charAt(j) == text.charAt(i))
                        if (j == 0)
                                return i; // match
                        else { // looking-glass technique
                                i--;
                                 j--;
                        }
                else { // character jump technique
                        int lo = last[text.charAt(i)]; //last
                                                        //occ
                        i = i + m - Math.min(j, 1+lo);
                        j = m - 1;
                }
        } while (i <= n-1);
        return -1; // no match
} // end of bmMatch()
```

(Source: Slides on Pattern Matching by Dr. Andrew Davidson and updated by Dr. Rinaldi Munir)

E. Dynamic Programming

Dynamic Programming is an optimization of recursion. Optimization can be done if there is a recursive solution that has repeated calls for inputs. Dynamic programming involves storing results of subproblems so that recompute isn’t needed. Dynamic programming is able to reduce time complexities from exponential to polynomial. The characteristics for problems that can be solved using Dynamic Programming, are, the problem can be divided into a number of stages, which in every stage, only one decision can be made, and each stage consists of states, which are various possible inputs, that are connected to that stage.
One common problem that can be solved using Dynamic Programming is the similarity between strings, which can be used to detect plagiarism. A method that can be used is edit distance

F. Edit Distance Dynamic Programming Algorithm

Edit distance algorithm aims to find the minimum edit distance between two strings. In this algorithm, in order to search for the sequence of edits from the start string to the final string, there are four important points.

- Initial State, which is the word that is going to be transformed
- Operators, which include:
    - Insertion of character
        - Cost = D(m , n) = D(m, n – 1) + 1
    - Deletion of character
        - Cost = D(m, n) = D(m – 1, j) + 1
    - Substitution of character
        - Cost = D(m, n) = D(m – 1, n – 1) + 1
- Final state, which is the word that is the output of transformation
- The cost produced to change one string to become the other string is minimum.

The idea is that if we are given 2 strings m and n

- If the last characters of the two strings match, nothing is changed and the program recurs for length m-1 and n-1
- Else, we compute the minimum cost of insert, delete and substitute and take the minimum of the three values.

An example is comparing “cart” to “march”.

EditDistance	Ø	M	A	R	C	H
Ø	0	1	2	3	4	5
C	1	1	2	3	3	5
A	2	2	1	2	3	4
R	3	3	2	1	2	3
T	4	4	3	2	2	3

Assume Cell is labeled (X,Y), with X is horizontal (corresponds to ØMARCH) and Y is vertical (corresponds to ØCART)

1st Column
Cell (Ø, Ø) = 0
Cell (Ø, C) = Cell (Ø, Ø) + 1 = 1

Cell (Ø, A) = Cell (Ø, C) + 1 = 2
Cell (Ø, R) = Cell (Ø, A) + 1 = 3
Cell (Ø, T) = Cell(Ø, R) + 1 = 4
1st Row
Cell (M, Ø) = Cell(Ø, Ø) + 1 = 1
Cell (A, Ø) = Cell(M, Ø) + 1 = 2
Cell (R, Ø) = Cell(A, Ø) + 1 = 3
Cell (C, Ø) = Cell(R, Ø) + 1 = 4
Cell (H, Ø) = Cell(C, Ø) + 1 = 5
2nd Row
Cell (M, C) = 1
Cell (A, C) = Cell (M, C) + 1 = 2
Cell (R, C) = Cell (A, C) + 1 = 3
Cell (C, C) = Cell (R, C) + 0 = 3
Cell (H, C) = Cell (C, C) + 1 = 4
3rd Row
Cell (M, A) = Cell(M, C) + 1 = 2
Cell (A, A) = Cell (M, C) + 0 = 1
Cell (R, A) = Cell (A, A) + 1 = 2
Cell (C, A) = Cell (R, A) + 1 = 3
Cell (H, A) = Cell (C, A) + 1 = 4
4th Row
Cell (M, R) = Cell (M, A) + 1 = 3
Cell (A, R) = Cell (A, A) + 1 = 2
Cell (R, R) = Cell (A, A) + 0 = 1
Cell (C, R) = Cell (R, R) + 1 = 2
Cell (H, R) = Cell (C, R) + 1 = 3
5th Row
Cell (M, T) = Cell (M, R) + 1 = 4
Cell (A, T) = Cell (A, R) + 1 = 3
Cell (R, T) = Cell (R, R) + 1 = 2
Cell (C, T) = Cell (R, R) + 1 = 2
Cell (H, T) = Cell (C, R) + 1 = 3

The time complexity of this algorithm is O (m x n) where m is the character length of one string and n is the character length of the other string.

III. Application of String-Searching Algorithm for Exact String Matching in Multilingual Plagiarism Checker

I will demonstrate the application of a Bilingual Plagiarism Checker, using a test case for Brute-Force Algorithm, Knuth-Morris-Pratt Algorithm and Boyer Moore Algorithm. As for the languages, this time I will use Indonesian and English. The Plagiarism checker will use a database of translations and thesaurus. The programming language used to create this application is Python 3.7.3.

A. Problem Definition

Given an input string pattern P, we have to determine whether a text includes an exact copy or an exact translation copy of a text T in the database.
In this problem, the text in the database is

```
Meskipun Joker direncanakan untuk dibunuh selama penampilan awalnya, ia terhindar dari intervensi editorial, yang memungkinkan karakter untuk bertahan sebagai musuh utama Batman."
```

An example input is

```
"Although the Joker was planned to be killed off during his initial appearance, he was spared by editorial intervention, allowing the character to endure as the archenemy of the Batman."
```

B. Solving the Problem

First, the language in the database has to be found out in order for the translator to be able to translate into the appropriate language. By using the googletrans translator database in python, we can identify the language of the text.

```
from googletrans import Translator
translator = Translator()
inputText = "Although the Joker was planned to be killed off during his initial appearance, he was spared by editorial intervention, allowing the character to endure as the archenemy of the Batman."

print(translator.detect(inputText))
```

(Python code to use googletrans translator database to detect language)
The output should be as follows:

```
Detected(lang=id, confidence=0.86180246)
```

Since the input text is in a different language from the text in the database, a translator must be used before the string matching algorithm. In order to do that, I will use the translation library in order to translate the text to Indonesian, which is the language that is used by the text in the database.

```
translation = translator.translate(dest='id')
```

Using the python code, the inputted text will be translated to Indonesian, and the result is as follows:

```
"Meskipun Joker direncanakan untuk dibunuh selama penampilan awalnya, ia terhindar dari intervensi editorial, yang memungkinkan karakter untuk bertahan sebagai musuh utama Batman."
```

After translating, we can then use any implementation of the string-searching algorithm to find the text in the database that exactly matches the input text.
The implementation is as follows.

Brute-Force String-Searching Algorithm

```
def BruteForceSearch(text, pattern):
    M = len(pattern)
    N = len(text)
    for i in range(N - M + 1):
        j = 0
        for j in range(0, M):
            if (text[i + j] != pattern[j]):
                break
        if (j == M - 1):
            print("Pattern found at index ", i)
```

Knutt-Morris-Pratt String-Searching Algorithm

```
def KMPMatch(pattern, text):
    M = len(pattern)
    N = len(text)
    fail = [0]*M
    j = 0
    computeFail(pattern, M, fail)
    i = 0
    while i < N:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == M:
            print("Pattern found at index " + str(i-j))
            j = fail[j-1]
        elif i < N and pattern[j] != text[i]:
            if j != 0:
                j = fail[j-1]
            else:
                i += 1

def computeFail(pattern, M, fail):
    fail[0] = 0
    i = 1
    j = 0
    while i < M:
        if pattern[i]== pattern[j]:
            j += 1
            fail[i] = j
            i += 1
        else:
            if j != 0:
                j = fail[j-1]
            else:
                fail[i] = 0
                i += 1
```

Boyer-Moore String-Searching Algorithm

```
def bmMatch(text, pattern):
    last = buildLast(pattern)
    n = len(text)
    m = len(pattern)
    i = m-1;

    if (i > n-1):
        return -1 # no match if pattern is
                  # longer than text

    j = m-1;
    while (i < n):
        if (pattern[j] == text[i]):
            if (j == 0):
                return (“Pattern found at index “ + str(i)) #match
            else : #looking glass technique
                i -=1
                j -=1
        else : #character juml algoritma
            lo = last[text[i]]
            i = i + m - min(j, 1 + lo)

def buildLast(pattern):
        #/* Return array storing index of last occurrence of each ASCII char in pattern. */

        last = {} #ASCII char set
        for i in range(128):
            last[i] = -1 #initialize array

        for i in range(len(pattern)):
            last[pattern[i]] = i

        return last
```

These string-matching algorithms are sufficient to solve if the plagiarizer just copied the text into the translator. However, the problem is that these algorithms couldn’t match the whole string in the database, since if the database contains a word that is different, the pattern cannot fully match. Therefore, another way to check if the input text is a plagiarized text is to use a different algorithm strategy, which is Dynamic Programming.

IV. Application of Edit Distance Dynamic Programming Algorithm in Multilingual Plagiarism Checker

I will demonstrate the application of a Multilingual Plagiarism Checker, using a test case for Minimum Edit Dis. As for the languages, this time I will use Indonesian and German. The Plagiarism checker will use a database of translations and thesaurus. The programming language used to create this application is Python 3.7.3.

A. Problem Description

Given an input string pattern P, we have to determine whether a text is similar to the text T.

In this problem, the text in the database is :

"Joker tidak memiliki kemampuan manusia super, alih-alih menggunakan keahliannya di bidang teknik kimia untuk mengembangkan ramuan beracun atau mematikan, dan persenjataan tematik, termasuk kartu bermain berujung silet, buzzers yang mematikan, dan bunga kerah yang menyemburkan asam."

And the input is

“Der Joker verfügt nicht über übermenschliche Fähigkeiten. Stattdessen nutzt er sein Fachwissen in der chemischen Verfahrenstechnik, um giftige oder tödliche Inhaltsstoffe und thematische Waffen zu entwickeln, darunter Spielkarten mit Rasiermessern, tödliche Summer und Kragenblüten mit Säure. Obwohl geplant war, dass der Joker während seines ersten Auftritts getötet wurde, überlebte er die redaktionelle Intervention, wodurch der Charakter als Batman Hauptfeind überleben konnte.”

B. Solving the Problem

First, we use the translator database to detect the language of the word in the database.

```
from googletrans import Translator
translator = Translator()
test_Database = "Joker tidak memiliki kemampuan manusia super, alih-alih menggunakan keahliannya di bidang teknik kimia untuk mengembangkan ramuan beracun atau mematikan, dan persenjataan tematik, termasuk kartu bermain berujung silet, buzzers yang mematikan, dan bunga kerah yang menyemburkan asam."

print(translator.detect(testDatabase))
```

The result will be as follows:

```
Detected(lang=id, confidence=0.86244613)
```

Which means the language of the text in the database is Indonesian, because the confidence is about 86%.

In order to know how similar the input text is, we can use Minimum Edit Distance. It is implemented in python with this algorithm. a is length of pattern P and b is length of text T.

```
def editDistance(pattern, text, a, b):
    table = [[0 for x in range(b+1)] for x in range(a+1)]
    for i in range(a+1):
        for j in range(b+1):
            if i == 0:
                table[i][j] = j
            elif j == 0:
                table[i][j] = i
            elif str1[i-1] == str2[j-1]:
                table[i][j] = table[i-1][j-1]
            else:
                table[i][j] = 1 + min(dp[i][j-1],  # Insert
                                  table[i-1][j],        # Remove
                                  table[i-1][j-1])     # Substitute
    return table[a][b]
```

Then we can use the algorithm

```
print((editDistance(test_Database,
testText.text,len(test_Database),
len(testText.text)))/len(testText.text))
```

Which is an implementation of

The smallest edit distance divided by the length of the database text, in order to find the similarity of the string.

The result is

```
0.7365145228215768
```

Which means 73% similarity. It means that the input text is quite plagiarized since it has quite a lot of similarity to the text.

V. Conclusion

Therefore, the translator database, along with String-Matching Algorithm, such as Brute-Force String-Matching Algorithm, Knutt-Morris-Pratt String-Matching Algorithm and Boyer-Moore String-Matching Algorithm can be used to detect translation plagiarism if the string is exact-matching after being translated, and Minimum Edit Distance Dynamic Programming Algorithm can be used to find similarities between two strings, which can be used to find plagiarism if the similarity that is found after the input text has already been translated is high.

VI. Acknowledgement

The author is grateful to The One Almighty God for the blessing that has been given so that this paper can be finished successfully. The author is also grateful to Dr. Masayu Leylia Khodra ST,MT, Dr. Ir. Rinaldi Munir, MT. and Dr. Nur Ulfa Maulidevi, ST, M.Sc. our lecturers in Algorithm Strategies Course, for the knowledge and the time which are shared with the college students attending the course. The author is also grateful for the support from the author's family and friends.

References

[1] Davison Andrew, Pattern Matching, 2006 (updated by Rinaldi Munir), diakses 26 April 2019

[2] https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/ diakses 26 April 2019

[3] https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ diakses 26 April 2019

[4] https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/ diakses 26 April 2019

[5] https://www.geeksforgeeks.org/dynamic-programming/ diakses 26 April 2019

[6] https://web.stanford.edu/class/cs124/lec/med.pdf diakses 26 April 2019

[7] https://smallbusiness.chron.com/cut-copy-paste-mean-word-processing-66264.html diakses 26 April 2019

[8] https://www.geeksforgeeks.org/edit-distance-dp-5/ diakses 26 April 2019

[9] https://www.plagiarism.org/article/what-is-plagiarism diakes 26 April 2019

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2019

[illegible]

Ignatius Timothy Manullang - 13517044