Levenshtein Distance Calculation Using Dynamic Programming for Source Code Plagiarism Checking

Tito D. Kesumo Siregar[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13511018@std.stei.itb.ac.id

Abstract **-- The aim of this paper is to determine whether Levenshtein distance is a good enough value for plagiarism checking. In order to determine it, an experiment is made using an implementation of Wagner-Fischer algorithm to calculate Levenshtein distance. From the Levenshtein distance, a similarity value is calculated to determine whether two source codes are similar or not, thus detecting the plagiarism attempt.**

Index Terms **-- plagiarism, levenshtein, edit distance, dynamic programming, wagner-fischer.**

I. INTRODUCTION

Computers are machines able to perform various kinds of tasks, algorithms, and calculations. These tasks, algorithms, and calculations are described for the computer using a collection of computer instructions called source code, usually as text. Source codes are specially designed to allow computer programmers to express computer tasks and calculations in a simpler and easier to read text. A source code will be compiled by a compiler program into a lower-level machine code understood by the computer. Alternatively, a source code can be interpreted to perform the tasks described by the source code on the fly.

Building computer programs and applications requires the computer programmers to create source codes to express what computers should do to perform a certain task, algorithm, or calculation. If a person owns the source code for a particular application, the person will be able to build the computer program expressed by the source code. This is a delicate issue for several entities working with source codes, such as businesses and academic entities. Businesses that work with computer programs need to have the source codes they created to be protected against plagiarism, while academic entities need to avoid plagiarism attempts of a source code.

To detect plagiarism of a source code, an algorithm to compare two similar source code is needed. There exists algorithms to compare strings that can be used to compare text, and compare source codes. However, source code comparison is not a trivial task because of various reasons, such as the presence of white space characters and attempts to avoid detection by modifying the source code variable names.

The aim of this paper is to determine whether Levenshtein distance is a good enough value for plagiarism checking. If it is good enough, a lower bound of value whether two file compared is considered a plagiarism should be able to be determined.

II. BASE THEORIES

2.1. Dynamic Programming

Dynamic programming is a technique widely used in solving computational problems by reducing the solution to a several computable steps. A distinguishing characteristic of dynamic programming is creating recursive function to solve a range in the problem, which then will be used to solve the whole problem set.

Dynamic programming can dramatically reduces the runtime of some algorithms (but not all problems has dynamic programming characteristics) from exponential to polynomial. Many (and still increasing) real world problems are only solvable within reasonable time using dynamic programming.

To be able to use dynamic programming, the original problem must have: (1) optimal sub-structure property: optimal solution to the problem contains within its optimal solution to sub-problems, and (2) overlapping sub-problems property: to recalculate the same problem twice or more. By using the optimality principle, if the total solution is optimal, then the sub-solutions leading to the total solution is optimal, too.

The common dynamic programming implementation methods are top-down dynamic programming and bottom-up dynamic programming. The top-down dynamic programming often used a recursive function with memorization, while the bottom-up dynamic programming often used an array or a matrix.

2.2. Levenshtein Distance

Levenshtein distance is a measure of difference between two strings. Informally, the Levenshtein distance between two words is the minimum number of single-character edits required to change a word into another word. The measurement was first considered by

Vladimir Levenshtein, and his name is used as the distance name. The distance may also be referred as edit distance, although the edit distance term actually refers to a larger set of string difference measure, such as longest common subsequence (LCS) and Hamming distance.

Formally defined, given two strings and its character set, the edit distance is the minimum-weight series of edit operations that transforms the first string into the second string. One of the simplest sets of edit operations defined by Levenshtein in 1966 are: (1) insertion of a single symbol, (2) deletion of a single symbol, and (3) substitution of a single symbol. The original definition has a unit cost, so the Levenshtein distance is equal to the minimum number of steps to change the first string into the second string. This definition is referred as Levenshtein distance.

For those who are inclined to read the mathematical definition, the Levenshtein distance between two strings a, b is given by $lev\big(length(a), length(b)\big)$ where:

$$lev(i,j) = \begin{cases} \max(i,j), \min(i,j) = 0 \\ \min \begin{cases} lev(i-1,j)+1 \\ lev(i,j-1)+1 \\ lev(i-1,j-1)+eq(i,j) \end{cases}, \min(i,j) \neq 0 \end{cases}$$

where $eq(i,j)$ returns 1 if $a[i] = b[j]$ and 0 otherwise.

For example, the Levenshtein distance between the string "kitten" and "sitting" is 3. The steps to change "kitten" into "sitting" are: (1) replace "k" with "s", (2) replace "e" with "i", and (3) insert "g" at the end.

There exists several simple upper and lower bounds of Levenshtein distance, such as: (1) it is always at least the difference of the sizes of the two strings, (2) it is at most the length of the longer string, (3) it is zero if and only if the strings are equal, (4) the Hamming distance is an upper bound of Levenshtein distance, and (5) the triangle inequality (Levenshtein distance between two strings is no greater than the sum of the Levenshtein distances of the two strings from a third string).

2.3. Wagner-Fischer Algorithm

An algorithm to compute Levenshtein distance exists by simply following the definition. The result is a straightforward but inefficient algorithm because the algorithm will recalculate Levenshtein distance of same substrings many times. However, using dynamic programming, the calculation can be computed using a two-dimensional matrix. The resulting algorithm is known as Wagner-Fischer algorithm. The pseudo code for the algorithm is given below.

```
int levenshteindistance(
    char a[1..m],
    char b[1..n]
)
    // dp[i][j] contains levenshtein distance
    // between a[0..i] and b[0..j]
    dp: array of int[0..m][0..n]

    // the levenshtein distance of a string and
    // an empty string is equal with the length
    // of the string
    for i in [0..m] dp[i][0] = i
    for j in [0..n] dp[0][j] = j

    for j in [1..n]
        for i in [1..m]
            if a[i] == b[j]
                // no operation is required
                dp[i][j] = dp[i-1][j-1]
            else
                // select minimum levenshtein
                // distance between deletion,
                // insertion, and substitution
                dp[i][j] = min(
                    dp[i-1][j  ] + 1,
                    dp[i  ][j-1] + 1,
                    dp[i-1][j-1] + 1

    // the levenshtein distance of the whole
    // two string is stored in the bottom-right
    // cell of the dp table
    return dp[m][n]
```

The pseudo code described above assumes that every actions' cost is 1. Because the algorithm traverses a 2D matrix, the algorithm will run in the speed of $O(mn)$ with the required space of $O(mn)$. A possible improvement is to reduce the space complexity from $O(mn)$ to $O(m)$, observing that the algorithm only requires that the previous row and current row can be stored at any one time.

2.4. Source Code Comparison

It is desirable to have a single value to represents the similarity of a given text. In order to do that, one should be able to give the similarity by using the Levenshtein distance value. Furthermore, the similarity value should fulfill these requirements:

1. The similarity of two completely different text should be 0.
2. The similarity of two completely identical text should be 1.
3. The similarity of two text should be a value between 0 and 1, inclusive.

Given two strings a and b, we can determine the Levenshtein distance of the strings L_{ab} by using Wagner-Fischer algorithm. To construct a percentage value, we can compare L_{ab} with some value.

A lower bound of L_{ab} is 0, which is when a and b are identical. An upper bound of L_{ab} is $max(l(a), l(b))$, where $l(x)$ represents the length of the string x. From it,

we are able to derive the following equation to determine the similarity value of two strings:

$$S_{ab} = 1 - \frac{L_{ab}}{\max(l(a), l(b))}$$

where S_{ab} is the similarity value between two strings a and b, L_{ab} is the Levenshtein distance between two strings a and b, and $l(a)$ and $l(b)$ represents the length of the strings a and b, respectively.

III. Implementation and Experiment

3.1. Implementation

For the experiment, an implementation of the Wagner-Fischer has been written in C++. Below are the implementation of the algorithm.

```
int levenshtein(string a, string b) {
    int dp[MAX_LENGTH + 1][MAX_LENGTH + 1];
    int len_a = a.size();
    int len_b = b.size();

    // pad the strings to make it 1-based
    a = "#" + a;
    b = "#" + b;

    // dp(string, empty) = length of string
    for (int i = 0; i < len_a; i++) {
        dp[i][0] = i;
    }
    for (int j = 0; j < len_b; j++) {
        dp[0][j] = j;
    }

    // dp table building steps
    for (int i = 1; i < len_a; i++) {
        for (int j = 1; j < len_b; j++) {
            if (a[i] == b[j]) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = min3(
                    dp[i-1][j] + 1,
                    dp[i][j-1] + 1,
                    dp[i-1][j-1] + 1
                );
            }
        }
    }

    // print the dp table
    for (int i = 0; i < len_a; i++) {
        for (int j = 0; j < len_b; j++) {
            cout << dp[i][j];
            if (j + 1 == len_b) {
                cout << endl;
            } else {
                cout << " ";
            }
        }
    }

    return dp[len_a - 1][len_b - 1];
}
```

3.2. Source Code Preprocessing

A problem with the above implementation of Wagner-Fischer algorithm is that the length of the string to be compared is limited by `MAX_LENGTH`. The size of `MAX_LENGTH` determines the size of the matrix, which cannot be larger than `512` in a standard computer.

To make the way around, an observation of source code reveals that more than half of the characters in the source code are whitespaces. These whitespaces can be safely ignored, reducing the length of the source code string.

Below are the code for source code preprocessing.

```
string readfile(string filename) {
    string retval;
    ifstream is(filename.c_str());
    while (is.good()) {
        char c = is.get();
        if (!isspace(c)) retval += c;
    }
    is.close();
    return retval;
}

int main(int argc, char** argv) {

    if (argc < 3) {
        string name(argv[0]);
        cout << "usage: " << name << " <file1>
<file2>" << endl;
    } else {
        string file1(argv[1]);
        string file2(argv[2]);

        string a = readfile(file1);
        string b = readfile(file2);

        if (a.length() >= MAX_LENGTH ||
b.length() >= MAX_LENGTH) {
            cout << "Length of a=" <<
a.length() << " or b=" << b.length() << " is
too long." << endl;
        } else {
            cout << levenshtein(a, b) << endl;
        }
    }
}
```

3.3. Experiment

The test cases for the experiment is taken from several source codes from assignments of a C++ course, thus making all test case source codes are written in C++. However, the source code comparison program should work for all kinds of source code (the implementation provided can only read a 512-character length file at most). All test cases compared are written for a particular assignment problem. There are several test cases to be tested:

1. Test case where the source code is very similar, differing only in several syntax and variable declaration.
2. Test case where the source code for a same problem is written by two different people,

with the result expected should be totally different.

3. Test case where a plagiarism attempt occurs, where the source code variable names are modified.
4. Test case where a plagiarism attempt occurs, where several string literals for the program are modified.
5. Test case where a plagiarism attempt occurs, where parts of the source code are further modified by rewriting parts of the code.

The test case (1) and (2) serves as a control for a source code without plagiarism, while test case (3) and (4) serves as the variable for plagiarism detection.

Table III.1 below shows the input for the test cases, with length of each file after preprocessed. Note that the preprocessed file is no longer be able to be compiled by a C++ compiler because of the removal of all whitespaces, ignoring the semantics of the whitespace (whitespace needed to separate variable type and name such as `int` and `argc`, for example).

Test case	Case	Length of first file	Length of second file
1	Very little difference	688	706
2	Different source code	700	716
3	Plagiarism: modified variable names	706	706
4	Plagiarism: modified literals and constants	706	714
5	Plagiarism: rewriting parts of the code	706	674

Table III.1 The input length of each test cases

Table III.2 below shows the expected similarity value of the test cases and the output of the source code comparison program (the Levenshtein distance and the similarity value). The expected similarity of test case (1), (3), and (4) is high because the source code of (1) is very similar and (3) and (4) is a plagiarism attempt. The expected similarity of test case (2) is low because of the test case represents a source code for same problem but are written by two different people. The expected similarity of test case (5) is low because while the case is a plagiarism attempt, the rewriting of the source code and movement of several source code blocks is deemed 'good' enough to fool the source code comparison program. The similarity value is calculated using the previous equation, while the Levenshtein distance is calculated by using Wegner-Fischer algorithm implementation written above.

Test case	Expected similarity value	Levenshtein distance	Similarity value
1	High	22	96.9%
2	Low	474	33.8%
3	High	41	94.2%
4	High	44	93.8%
5	Low	205	71%

Table III.2 The result of each test cases

IV. ANALYSIS

4.1. Similarity Analysis

Test case (1) reveals the algorithm works as expected, with the similarity value of 96.9% for two very similar files (only differing in small details, such as variable declaration).

Test case (2) reveals that the similarity value of two different files are actually pretty small (33.8%). Furthermore, there is a considerable difference between the smallest similarity value test case with this test case (71% compared to 33.8%). This shows that the error margin for the algorithm to throw a false detection (detecting a non-plagiarism file as a plagiarism file) are small, thus the algorithm are quite reliable for the purpose.

Test case (3) and (4) reflects a small plagiarism attempt by modifying the variable names, literals, and constants. The algorithm manages to detect the similarity, showing that the algorithm works. These cases have actually happened on the selected C++ courses used as a test case sources.

Test case (5) is a more sophisticated plagiarism attempt by rewriting the code, while still having the same logic. This is done by moving several movable blocks such as variable declaration and rewriting several blocks. The test case is expected to break the algorithm, by providing a false pass (detecting the plagiarism code as legal). However, the result is quite a twist; the similarity value is still relativity high compared to (2). This shows that Levenshtein distance is a good measurement to detect plagiarism. Furthermore, a bound of 70% is deemed enough to distinguish between plagiarism and non-plagiarism source code.

4.2. Breaking the Algorithm

The experiments conducted shows that the Levenshtein distance is a good measurement for plagiarism detection, and a bound of 70% is deemed enough to serve as a bound between plagiarism and non-plagiarism code. However, the algorithm is not foolproof. Several strategies to force a false detection or a false pass exists, such as:

1. Adding trash codes and characters to the code.
2. Using a different procedure or function to produce similar results.

Adding trash codes and characters to the code works by increasing the amount of insertion, thus increasing the Levenshtein distance. In C++, this can be done by adding extra semicolon ending (;), whitespaces, or adding useless program blocks. While this strategy will easily break the algorithm, such strategies are easily seen by a human, and creating useless program blocks is not a relativity trivial task in a constrained environment (in a 120-minutes laboratorium assignment, for example). Furthermore, this strategy may not work well on programming languages that do not ignore whitespaces (such as Python and Ruby).

Rewriting program blocks to use a different procedure or function works by observing that many programming languages offers various alternatives for certain tasks. For example, C++ offers standard input `cin` and `cout`, but also still provides the `scanf()` and `printf()` function from C. This increases the amount of subsistution needed, increasing the Levenshtein distance and reducing the similarity value. This strategy may be avoided by forcing to use a certain function to do something, for example by forcing to use `cin` and `cout` instead of `scanf()` and `printf()`.

4.3. Algorithm Performance

The time complexity of Wagner-Fischer algorithm is $O(mn)$, where m, n represents the string length of its respective input. However, the hidden constants is small because there are no preprocessing for the algorithm for work and the main loop for matrix traversal consists of a single if-else statement. During the experiment, attempts to measure the speed of algorithm resulted in the speed of 0.002 ms to 0.003 ms on a modern processor. The algorithm is fast enough to be implemented on a processing-heavy environment.

The space complexity of Wagner-Fischer algorithm is $O(mn)$, where m, n represents the string length of its respective input. However, this limitation is a problem even for modern computers; during the experiment, the maximum input length is limited to only 719 characters. While this maximum input length varies over the computers, many source codes submitted for the selected C++ courses are twice longer (around 1400 to 2000 characters). Thus, 'hacks' to make the input string shorter are needed; for the experiment, the whitespaces are omitted. This proved to be serendipitous, because large amount of whitespace can increase the Levenshtein distance value and reducing the similarity value. Nevertheless, optimization for the algorithm can be made, such as by observing that the algorithm only requires that the previous row and current row can be stored at any one time. Using the fact, the space complexity can be reduced to $O(n)$.

V. Conclusion

The original aim to determine whether Levenshtein distance is a good enough value for plagiarism checking results in a positive; Levenshtein distance is a good enough value for plagiarism. The lower bound to determine whether a file is a plagiarism file is deemed to be 70%. There exists strategies to fool the algorithm but the strategies can be avoided by using several restrictions.

For the performance of the algorithm, the Wagner-Fischer algorithm to calculate the Levenshtein distance is deemed fast enough for processing-heavy environment such as web servers. However, the original implementation of the algorithm has a weakness in the high space complexity, limiting the source code size able to be checked. This limitation can be avoided by observing that the algorithm only requires that the previous row and current row can be stored at any one time. Using the fact, the space complexity can be reduced to $O(n)$.

VI. Reference

[1] Munir, Rinaldi. 2009. *Diktat Kuliah IF2211 Strategi Algoritma*. Program Studi Teknik Informatika ITB.

[2] Arefin, Shamsul Ahmed. 2009. *Art of Programming Contest 2nd Edition*.

[3] Halim, Steven and Halim, Felix. 2010. *Competitive Programming: Increasing the Lower Bound of Programming Contests*.

[4] *Minimum Edit Distance*. http://www.stanford.edu/class/cs124/lec/med.pdf. Accessed at May 18th, 2014.

VII. Acknowledgement

This paper is written by Tito D. Kesumo Siregar, intended for an assignment in IF2211 Algorithm Strategy course in Bandung Institute of Technology at 2014.

The writer wants to express his gratitude to Dr. Ir. Rinaldi Munir, M.T. and Masayu Leylia Khodra, S.T., M.T. as his lecturers on the course. The writer also express his gratitude to the family and the family of HMIF ITB (*Himpunan Mahasiswa Informatika ITB*).

VIII. NOTES

A repository containing the source code for the experiments, along with the test cases and a digital copy of this document is available on the Internet at https://github.com/tkesgar/paper-stima-levenshtein.

IX. 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, 29 April 2010

Tito D. Kesumo Siregar
NIM. 13511018