

# Dynamic Programming on Plagiarism Detecting Application

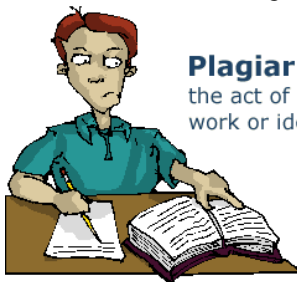
Edwin Lunando/13509024  
Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia  
edwinlunando@gmail.com

**Abstract**—Plagiarism in text or document is an issue that strongly concerned by the academic community. Nowadays, the most common text plagiarism occurs by making a set of minor alterations that include the insertion, deletion, or replacing words. The paper will present two plagiarism detection method. The first one is derived from Levenshtein Distance and the other one is derived from Longest Common Subsequence that was a classical tool in the indentification of local similarities in biological sequences. Based on the results, Both algorithm give significant improvement from the brute force method. In the future, it would be interesting to explore some heuristics method to improve the efficiency.

**Index Terms**—dynamic programming, levenshtein distance, longest common subsequence, plagiarism.

## I. INTRODUCTION

In this days, there are numerous amount of academic paper and journal that published everyday. Since every academic will have to write at least one paper in their entire academic life, there will be a huge amount of people who share the same topic of their paper. Moreover, in this era, people would search at their favorite search engine to find their paper material. Because their keywords is basically similar to each other, they would seen the same page. There are a lot of people would copy those materials without giving any credits to the writer. This action was highly concerned by the academic community, but since there were numerous amount of papers that were published, checking each paper manually or using the brute force method would be infeasible. We need a more efficient algorithm to improve the work performance to check the integrity of one paper.



**Plagiarism:**  
the act of presenting another's  
work or ideas as your own.

Figure 1 The plagiarism

The purpose of this paper is to give a solution for people that having a hard time to check the integrity of numerous paper. After reading this paper, the writer hope that the readers will understand a better method to check plagiarism and save a lot of time while checking a paper. The paper will also give the analysis of the complexity of all algorithm that would be used in the problem in details. And finally, the paper will show that plagiarism checking problem could be solved efficiently with both proposed algorithm, the Levenshtein distance and Longest Common Subsequence(LCS). An example java application is created to analyzed the algorithm.

## II. THEORY

### II.I Dynamic Programming

**Dynamic Programming(DP)** is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure (described below). When applicable, the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

### II.II Levenshtein Distance

In information theory and computer science, the Levenshtein Distance is a string metric for measuring the

amount difference between two sequences. The term edit distance is often used to refer specifically to Levenshtein distance.

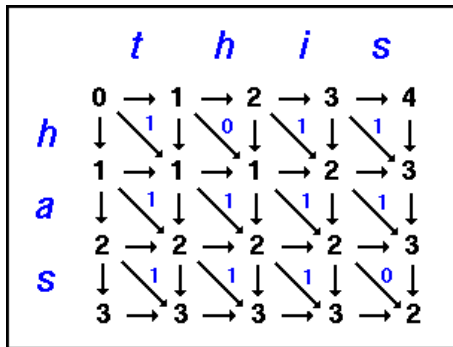


Figure 2 Example of Levenshtein Distance

The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character. It is named after Vladimir Levenshtein, who considered this distance in 1965.

### II.III Longest Common Subsequence

The longest common subsequence problem is to find the longest subsequence common to all sequence in a set of sequence. Note that subsequence is different from substring. It is a classic computer science problem, the basis of file comparison programs such as diff, and has applications in bioinformatics.



Figure 3 Example of LCS

Biological applications often need to compare the DNA of two different organism. A strand of DNA consists of a string of molecules called bases. One reason to compare two strands of DNA is to determine how similar the two strands are, as some measure of how closely related the two organisms are.

## III. ALGORITHM ANALYSIS

### III.I The Naïve Method

At the first time, the brute force method will be analyzed because any other algorithm will be measured by brute force algorithm. The main problem is to calculate

the similarity or difference of a document with other document in percent. It means we need to calculate the number of different and similarity of both document. In order to achieve that, we need to compare all combination of strings that appear in the document with the other one. Clearly a solution like this, that creating all permutations of substrings in a document will lead to factorial complexity  $O(n!)$ .

The solution is not appropriate because most document has more than 200 words. Clearly this solution is unfeasible because it would take forever to compare two documents. Clearly, we need a better solution.

### III.II The Levenshtein Distance Method

The Levenshtein method is using dynamic programming approach. Computing the Levenshtein distance is based on the observation that if we reserve a matrix to keep the Levenshtein distance between all prefixes of the first string and all prefixes of the second, then we can calculate the values in the matrix by flood filling the matrix, and then we can find the distance between two strings by the time the last value is computed. This algorithm, an example of bottom-up dynamic programming.

The matrix is filled from the upper left to the lower right corner. Each jump horizontally or vertically corresponds to an insert or delete, respectively. The cost is normally set to 1 for each of the operations. The diagonal jump can cost either one, if the two characters in the row and column do not match or 0, if they do. Each cell always minimize locally because this problem has an optimal substructure. This way, the number in the lower right corner is the Levenshtein distance between both words. This is an example code of Levenshtein distance in java programming language.

```

public int LDistance(String f, String s) {
    int n = f.length();
    int m = s.length();
    int cost = 0;
    int[][] dp = new int[n + 1][m + 1];
    for (int x = 0; x <= n; x++) {
        dp[x][0] = x;
    }
    for (int x = 0; x <= m; x++) {
        dp[0][x] = x;
    }
    for (int x = 1; x <= n; x++) {
        for (int y = 1; y <= m; y++) {
            if (f.charAt(x - 1) == s.charAt(y - 1)) {
                cost = 0;
            } else {
                cost = 1;
            }
            dp[x][y] = Minimum((dp[x - 1][y] + 1),
                (dp[x][y - 1] + 1), (dp[x - 1][y - 1] + cost));
        }
    }
    return dp[n][m];
}

public int Minimum(int a, int b, int c) {
    int mi = a;
}

```

```

    if (b < mi) {
        mi = b;
    }
    if (c < mi) {
        mi = c;
    }
    return mi;
}

```

$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}), x_i & \text{if } x_i = y_j \\ \text{longest}(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

Figure 4 Longest Common Subsequence

This is an example code of the LCS in java programming language.

This method calculate the number of differences of two strings. Due to two nested loops, we could conclude that the computational complexity of this algorithm is polinomial  $O(N^2)$ , much more efficient that the brute force algorithm that has factorial complexity  $O(n!)$ .

After we obtain the number of differences between two documents, by using simple formula, we could calculate the percentage of the differences between those two documents.

$$P = L \text{ distance} / \text{Number of characters} \times 100\%$$

By this result we could determine differences efficiently. There are a number of improvement to this algorithm:

- We can adapt the algorithm to use less space,  $2 \times O(\min(n, m))$  instead of  $O(mn)$ , since it only requires the previous row and current row be stored at any one time.
- We can store the number of insetions, deletions, and replacement seperately, or even the posistion at which they occur.
- By examining diagonal instead of rows, and by using lazy evaluation, we can find the Levenshtein distance in  $O(m(1 + d))$  time, which is much faster that the regular dynamic programming if the distance was small.

III.III The LCS Method

The LCS method is using the dynamic programming approach. The LCS problem has an optimal substructure, which means the problem can be broken down into smaller subproblem until the solution becomes trivial.

The subproblem become simpler as the sequences become shorter. Shorter sequences are conviniently described using the term prefix. A prefix of sequence id the sequence with the end of cut off. This method relies on the following two properties.

The first one is, suppose that two sequences both end in the same element. To find their LCS, shorten each sequence by removing the last element, find the LCS of the shortened sequences, and to that LCS append the removed element.

The second one is, Suppose that the two sequences X and Y do not end in the same symbol. Then the LCS of X and Y is the longer of the two sequence. By this two properties, now we could define the LCS function easily.

Let two sequences be define as follows:  $X = (x_1, x_2, \dots, x_m)$  and  $Y = (y_1, y_2, \dots, y_n)$  then, let  $LCS(X_i, Y_j)$  represent the set of longest common subsequence of prefixes  $X_i$  and  $Y_j$ .

```

public int lcs(String x, String y) {
    int m = x.length(), n = y.length();
    int[][] b = new int[m + 1][n + 1];
    for (int q = 0; q < m; q++) {
        for (int p = 0; p < n; p++) {
            if (x.charAt(q) == y.charAt(p)) {
                b[q + 1][p + 1] = b[q][p] + 1;
            } else {
                b[q + 1][p + 1] =
                java.lang.Math.max(b[q + 1][p], b[q][p + 1]);
            }
        }
    }
    return b[m][n];
}

```

While Levenstein distance calculate the number of differences between two document, the LCS calculate the similarities between two documents. Due to two nested loops, the computational complexity of this algorithm is polinomial  $O(N^2)$ , the same with Levenstein distance that was much more efficient than the naïve method.

$$P = LCS / \text{Number of characters} \times 100\%$$

Within this formula, we could easily see the similarity percentage between two documents with efficient resources compared to the naïve method.

Most of the time taken by the naïve algorithm is spent performing comparisons between items in the sequences. For textual sequence such as source code, you want to view lines as the sequence elements instead of single characters. This can mean comparisons of relatively long strings for each step in the algorithm. Two optimization can be made that can help to reduce the time these comparisons consume.

A hash function can be used to reduce the size of the strings in the sequences. That is, for source code where the average line is 60 or more characters long, the hash for that line might be only 8 to 40 characters long. Additionally, the randomized nature of hashes would guarantee that comparisons would be faster, as lines of source code will rarely changed at the beginning.

Like the Levenshtein method, we can reduce the required space into  $2 \times \min(m, n)$  as the dynamic programming approach only needs the current and previous collumns in the matrix.

IV PLAGIARISM ANALYSIS WITHIN TWO ALGORITHM

Since both algorithm had the same complexity and similar constant, the running time of both algorithm should be similar to each other. A java application is made to test and analyze the validity and the efficiency of

the algorithm. Moreover, with this test, we could analyze the pattern on plagiaristic documents.

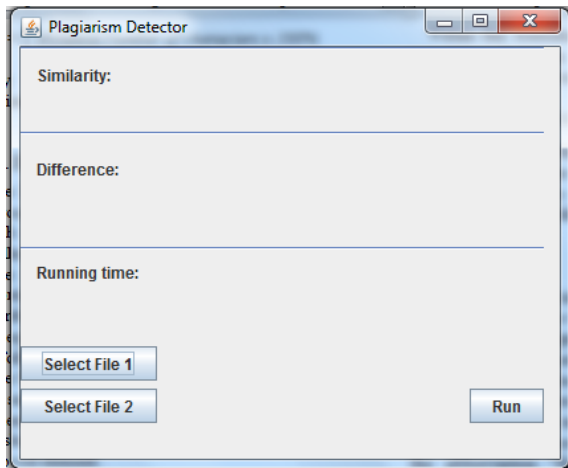


Figure 5 Plagiarism Detecting Application

The similarity will be calculated by LCS algorithm and the difference will be calculated by the Levenshtein algorithm. The following figure will compare two file texts with slight modification from the other one.

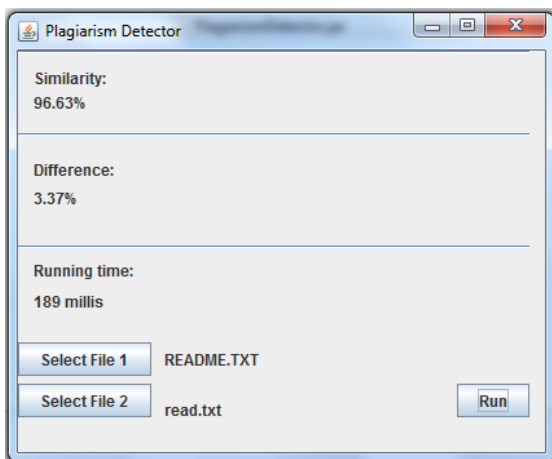


Figure 6 Test simple case

Clearly from the result we could conclude that those two files are almost identical. In order to know the standard of plagiaristic document that use find and replace technique, we need to conduct research on those kind of document. After testing the application for some plagiaristic document, there are some pattern that we could see the difference between plagiaristic document and the one that not.

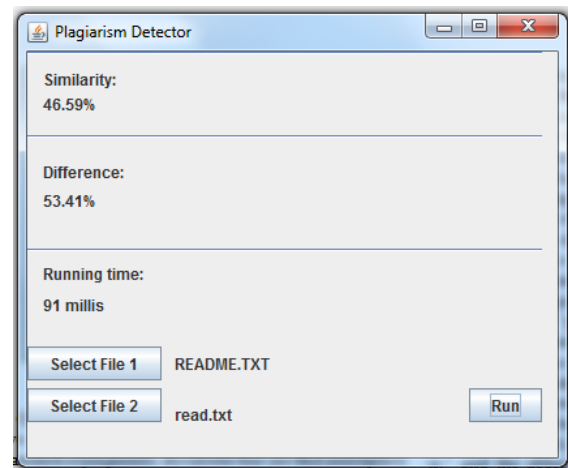


Figure 7 Plagiaristic Document

Most plagiaristic document has the similarity range between 30%-60% depend on the number of insertion, deletion, and replacement. After analyzed the document, we could found that plagiaristic document would only replace some important words and left the other as it is. Document with more than 60% similarity will be considered almost identical due to high percentage of similarity, while unplagiaristic document got less than 20% of similarity. Most of the document, even though they are has the same topic, there are a lot of difference in the document. Intiutively, we could think that every people has its own way of writing so that the words that used in the document are the writer preference, only the main topic keywords that increase the similarity.

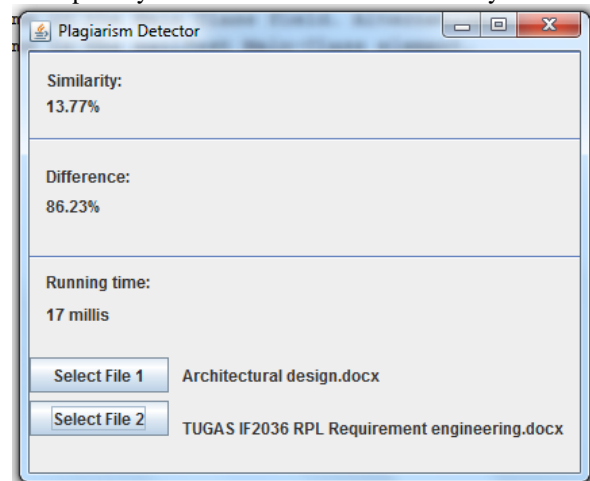


Figure 8 Unplagiaristic Document

After condunting all the research on both type of document, actually we could indentify the difference between plagiaristic document and not. Both algorithm is suffice to indentify the similarity or the difference between two document.

## V. CONCLUSION AND ADVICE

So, instead of using the naïve method that could that takes forever to find the number of differences on two documents, we have two alternate more efficient algorithm to calculate the similarity or difference of two documents.

From the research, we could conclude that unplagiaristic documents has the similarity percentage below 20% due to different writing types for each person, while plagiaristic documents has the range 30%-60%.

For the future, we could use the heuristic search to improve the efficiency of the algorithm because with polinomial complexity there are some limit that a normal personal computer could calculate in time. There are a lot of type of heuristic we could use to lower the running time of the algorithm.

## REFERENCES

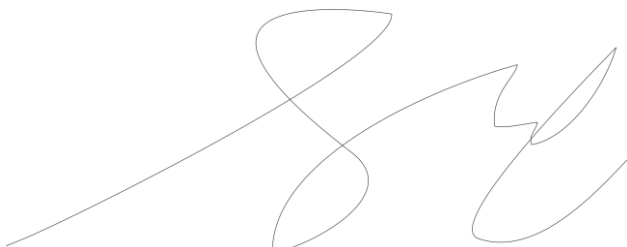
- [1] Che, Xin, 2003. *Shared Information and Program Plagiarism Detection*. [Online] (updated 13 May 2003). Available at: <http://monod.uwaterloo.ca/papers/> [Accessed 8 December 2011]
- [2] Su, Xhan, 2008. *Plagiarism Detection with Levenshtein Distance and Smith-Waterman Algorithm*. [Online] (updated 8 June 2008). Available at: <http://www.mendeley.com/research/plagiarism-detection-using-the-levenshtein-distance-and-smithwaterman-algorithm> [Accessed 8 December 2011]
- [3] Rouch, Erick, 2009. *Dynamic Programming*. [Online] (updated 10 January 2009). Available at: <http://www.avatar.se/molbioinfo2001/dynprog/dynamic.html> [Accessed 8 December 2011]

## 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, 8 Desember 2011

ttd



Edwin Lunando/13509024