Enhanced Levenshtein Edit Distance Method functioning as a String-to-String Similarity Measure

Prof. Dr. Abbas M. Al-Bakry
University of Information Technology and Communications
Baghdad, Iraq
Abbasm.albakry@uoitc.edu.iq

Marwa K. Al-Rikaby
College of Information Technology
Babylon University
Babylon, Iraq
marwaalrikaby@gmail.com

***Abstract*__ Levenshtein is a Minimum Edit Distance method; it is usually used in spell checking applications for generating candidates. The method computes the number of the required edit operations to transform one string to another and it can recognize three types of edit operations: deletion, insertion, and substitution of one letter. Damerau modified the Levenshtein method to consider another type of edit operations, the transposition of two adjacent letters, in addition to the considered three types. However, the modification suffers from the time complexity which was added to the original quadratic time complexity of the original method. In this paper, we proposed a modification for the original Levenshtein to consider the same four types using very small number of matching operations which resulted in a shorter execution time and a similarity measure is also achieved to exploit the resulted distance from any Edit Distance method for finding the amount of similarity between two given strings.**

***Keywords*_ Minimum Edit Distance, Similarity, Levenshtein method, Damerau's errors types.**

I. Introduction

Spell checking and correction is a common challenge in the area of language technology. It is one of the oldest most researched applications, started from the 1950s, and described as a challenge problem rather than a science. [8][4]

The spell checking task involves two main subtasks: error detection and error correction. The first deals with detecting mistakes in the given text (query, document, or even an isolated word) where several approaches are invented and varied in their efficiency and accuracy depending on the application environment and the available resources. [9][2]

The second subtask, error correction, involves generating the alternatives (candidates) for the misspelled word (or token) which is previously detected as erroneous, and suggesting those candidates as an output to the user (sometimes, a computer). The process of generating candidates is really a challenge problem till our days because the generation process is fully dependent on a set of factors like the underlying context, application environment, users' experience, the size of the lexicon, foundation of probabilistic and statistics information and its accuracy, and fundamentally on the method of selecting candidates, i.e. the way of computing the similarity (or reversely, the distance) between the source misspelled token and every alternative token.[6]

There are two well known types for error correction techniques: minimum edit distance and similarity based techniques; both of the two are usually independent of error detection technique used in the underhand application.

In this paper we are focusing on the Levenshtein method, which is a minimum edit distance technique, therefore, a short overview about these techniques is shown below:

Minimum edit distance is the minimum number of operations (insertion, deletion, substitution, and transposition) required for editing and transforming one string to another string. This technique is the most widely used in correcting spelling errors. [3] It takes a given string and matching it with a list of *M* words and returns the candidates with the minimum edit distances as correction suggestions. [1]

Different algorithms are invented in this technique field; Levenshtein, Hamming, and the longest common subsequence are examples of them. [1] Levenshtein algorithm is efficient compared with other methods because:

- It works with any kind of symbols in the input strings (binary, decimal, alphabetic ...).
- It accepts strings of different lengths (unlike Hamming).
- It gives accurately and can specify precisely (if preferred) what type of operation is required for transforming between the two input strings.

The Levenshtein was proposed by the Russian *Vladimir I. Levenshtein* in 1966 [6]; the algorithm computes the difference between any two string sequences by assigning each required edit operation a cost of 1 [10]. It is used in many different text correction applications, such as the post correction of Optical Character Recognition (OCR) [7], the dictionary looking up technique for candidates generation [5], and combined with other methods as an optimization tool [10].

II. Motivation

The obvious drawback of the Levenshtein algorithm is that it considers only three types of edit operations; it accounts a distance of 1 operation for each deletion, insertion, or a substitution operation but not the transposition of two adjacent symbols. Instead, it deals with this type of operations as two consecutive substitution operations and therefore accounts distance of 2 rather than 1.

In 1964, Damerau found in his research that the three types of errors considered by Vladimir in addition to the transposition error, altogether, caused 80% to 90% of misspellings; the research accounted only misspellings with at most one edit operation [10]. For these results, Damerau modified Levenshtein method in the followed years to consider all of the four types of errors. [4] Damerau's modification added more complexity to the quadratic time complexity method (N_1*N_2 is the complexity of the original Levenshtein method, where N_1 and N_2 are the lengths of $String_1$ and $String_2$, consecutively) because it consumes another comparison with every matching operation in the original algorithm to check if there is a transposition of two adjacent symbols. Specifically, the modification of Damerau multiplied the complexity by a factor of 2.

Typically, the algorithm is used to generate candidates from huge sized dictionaries; however, any additional time complexity has negative effects on the performance of the correction system. This overhead can be reduced if the transposition check is made with more sophisticated way using the same idea of Levenshtein. In this paper, we've proposed an alternative modification to the original Levenshtein method to consider the four types of errors with a time complexity close to the original time complexity.

III. The Levenshtein Method (the original algorithm)

The Levenshtein method is a minimum edit distance technique; it receives two strings of symbols of any type as inputs, and compares each symbol in the first string to every symbol in the second one for checking the difference between the two strings in terms of edit operations.

It can recognize three different types of edit operations: insertion, deletion, and substitution. Algorithm1 shows the method.

```
Algorithm1: Levenshtein Edit Distance
Input: $String_1$, $String_2$
Output: Edit Operations Number
Step1: Declaration
      distance(length of $String_1$, Length of $String_2$)=0, $min_1=0$, $min_2=0$, $min_3=0$, cost=0
Step2: Calculate Distance
if $String_1$ is NULL return Length of $String_2$
if $String_2$ is NULL return Length of $String_1$
for each symbol x in $String_1$ do
    for each symbol y in $String_2$ do
   begin
        if  $x = y$
           $cost = 0$
       else
           $cost = 1$
        r=index of x, c=index of y
        $min_1 = (distance(r - 1, c) + 1)$ // deletion
       $min_2 = (distance(r, c - 1) + 1)$ //insertion
       $min_3 = (distance(r - 1,c - 1) + cost)$ //substitution
       distance( r , c )=minimum($min_1$ ,$min_2$ ,$min_3$)
  end
Step3: return the value of the last cell in the distance matrix
 return distance (Length of $String_1$,Length of $String_2$)
End.
```

The method works by examining each symbol in the first input string ($String_1$) against each symbol in the second input string ($String_2$), the matching action requires a quadratic time complexity since it is performed by two nested loops.

Computing the distance between the two strings "Babylon" and "Babbly on" is an example for exploring how the method is working:

1. Define the dimensions of the distance matrix:
   No. of rows= length of the first string= ||Babylon||= 7
   No. of columns=length of the second string=||Babbly on||=9
2. Initialize the first column and the first row:
   Initial values of the distance matrix are:
   - An additional row contains the symbols of the second string.
   - An additional column contains the symbols of the first string.
   - Additional row and column contains numbers from (0 to 9) and (0 to 7), consecutively.

		B	a	b	b	l	y	␣	o	n
	0	1	2	3	4	5	6	7	8	9

B	1									
a	2									
b	3									
y	4									
l	5									
o	6									
n	7									

3. Start applying the matching:
   - Perform the following actions on each symbol in the first string (which represents the contents of the most left column):
     - Compare the letter "B" to every symbol in the top row; if matched set the variable cost to zero, otherwise set cost to one.
       Since "B"="B", the cost=0
     - Among the three values which stored in the cells that are surrounding the current cell (the cell under-consideration where we want to fill in order to complete the distance matrix) select the minimum according to the conditions in the method at step2.

The three cells are: the next on the left, the next on at the top, and the nearest at the left-top corner.

In terms of coordinates; for the cell at (i, j) select the minimum among:

{(i-1, j)+1, (i, j-1)+1, (i-1, j-1)+cost}.

And in our example:

		B	a	b	b	l	y	␣	o	n
	0	1	2	3	4	5	6	7	8	9
B	1									
a	2									
b	3									
y	4									
l	5									
o	6									
n	7									

The minimum is 0.

		B	a	b	b	l	y	␣	o	n
	0	1	2	3	4	5	6	7	8	9
B	1	0								

a	2									
b	3									
y	4									
l	5									
o	6									
n	7									

- Repeat the process until filling the first row, totally.

		B	a	b	b	l	y	␣	o	n
	0	1	2	3	4	5	6	7	8	9
B	1	0	1	2	3	4	5	6	7	8
a	2									
b	3									
y	4									
l	5									
o	6									
n	7									

- The distance matrix after completing calculating the distances row by row starting from the top row as shown in previous steps is:

		B	a	b	b	l	y	␣	o	n
	0	1	2	3	4	5	6	7	8	9
B	1	0	1	2	3	4	5	6	7	8
a	2	1	0	1	2	3	4	5	6	7
b	3	2	1	0	1	2	3	4	5	6
y	4	3	2	1	1	2	2	3	4	5
l	5	4	3	2	2	1	2	3	4	5
o	6	5	4	3	3	2	2	3	3	4
n	7	6	5	4	4	3	3	3	4	3

- The distance between the input strings is the value of the most right bottom cell (the last cell in the matrix) which holds the number 3.

The method expressed the difference between "Babylon" and "Babbly on" by three edit operations:

- Substituting 'y' by 'b'.
- Inserting 'y' after the 'l'
- Inserting '␣' after the inserted 'y'.

Notice, the method can indicate the difference of substrings, but constrained by necessary starting from the beginning of both the two strings; every (i, j) cell in the distance matrix holds the difference between the subsequence from the index (1 to i) from String$_1$ and the subsequence from the index (1 to j) from String$_2$.

Examples, consider the cell at (3,4) which means that the distance between "Bab" and "Babb" is 1 (inserting 'b' at the end); the cell at (4,4) holds a value of 1 and means that the difference between "Baby" and "Babb" is one edit operation (substituting 'y' by 'b').

Therefore, the type of the edit operations, if it is necessary to be detected, is differing according to the length of the subsequence taken from the string.

It is an interested feature in the Levenshtein method to find the ***minimum*** edit distance in this flexible manner instead of expressing the difference sequentially as: inserting 'b', substituting 'y' by 'l', substituting 'l' by 'y', and finally inserting white space which resulting in a distance of 4.

The weakness of this method appears in cases where the error is resulted from transposing two adjacent symbols, like the case of the 'l' and 'y'; it accounts two consecutive substitutions instead of one transposition. This idea is obvious in this example:

According to Levenshtein the distance between "Babylon" and "Bablyon" is 2 (substituting 'l' by 'y', and 'y' by 'l') but in fact we require only one edit operation (transposing 'y' and 'l') to transform the two strings the one to the other.

		B	**a**	**b**	**l**	**y**	**o**	**n**
	0	**1**	**2**	**3**	**5**	**6**	**8**	**9**
B	**1**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
a	**2**	**1**	**0**	**1**	**2**	**3**	**4**	**5**
b	**3**	**2**	**1**	**0**	**1**	**2**	**3**	**4**
y	**4**	**3**	**2**	**1**	**1**	**1**	**2**	**3**
l	**5**	**4**	**3**	**2**	**1**	**2**	**2**	**3**
o	**6**	**5**	**4**	**3**	**2**	**2**	**2**	**3**
n	**7**	**6**	**5**	**4**	**3**	**3**	**3**	**2**

In section IV, we will discuss the modification of Damerau on this method for overcoming its weakness.

IV. DAMERAU-LEVENSHTEIN DISTANCE

The idea of Damerau [5] to check whether a transposition is found was by matching every two consecutive symbols in one string with the mirror of every two consecutive symbols in the other string. In another word; to check if a symbol X was transposed with an adjacent symbol Y, the method must match the sequence XY with every two consecutive symbols WZ in the other string. If YX matched WZ then a transposition is found; otherwise, it is not.

The matching process is repeated in times equal to the multiplication of the lengths of the two input strings because it is associated with every basic matching operation; therefore, the method requires longer time to find the total distance.

```
1. Algorithm2: Damerau-Levenshtein Distance
2. Input: $String_1$, $String_2$
3. Output: Damerau Edit Operations Number
4. Step1: Declaration
5.  distance(length of $String_1$,Length of $String_2$)=0, $min_1=0$, $min_2=0$,  $min_3=0$, cost=0
6. Step2: Calculate Distance
7. if $String_1$ is NULL return Length of $String_2$
8. if $String_2$ is NULL return Length of $String_1$
9. for each symbol x in $String_1$  do
10.     for each symbol y in $String_2$ do
11.   begin
12.         if    x = y
13.             cost = 0
14.        else
15.              cost = 1
16.        r=index of x, c=index of y
17.        $min_1$ = (distance(r - 1, c) + 1) // deletion
18.        $min_2$ = (distance(r, c - 1) + 1) //insertion
19.        $min_3$ = (distance(r - 1,c - 1) + cost) //substitution
20.        distance( r , c )=minimum($min_1$ ,$min_2$ ,$min_3$)
21.        if     not($String_1$ starts with x) and not ($String_2$ starts with y) then
22.             if (the symbol preceding x= y) and (the symbol preceding y=x) then
23.                     distance(r,c)=minimum(distance(r,c), distance(r-2,c-2)+cost)
24.   end
25. Step3: return the value of the last cell in the distance matrix
26.         return distance(Length of $String_1$,Length of $String_2$)
27. End.
```

Although the modification gave accurate results, it increased the time complexity. Such additional complexity must be avoided in situations when the method is used for candidates generation where a source string should be matched with every token in a huge dictionary.

V. ENHANCED LEVENSHTEIN METHOD

The modification on the Levenshtein method can be performed by extending the standard matching step at line.12 to check the foundation of a transposition case. The idea rises from the fact that no transposition case may be found without finding a matching success between at least two symbols in the examined strings; and more precisely the transposition can be discovered using minimum number of operations by considering two facts:

- Two adjacent symbols can never be mirrored by other two adjacent symbols in another string unless the first symbol in the first set matches the second in the second set.
- Instead of manipulating the transposition occurrence separately, the algorithm can modify the under-processing cell in the distance matrix directly and the next matching steps will do the work.

The first point served in avoiding the trying of all possibilities as it was presented in Damerau's modification at lines 20 and 21 where each symbol is matched to every symbol in the second string regardless to the availability of a transposition operation happen by adding additional matching statements to the original one at line 12.

On the other hand, the second point announces another side of processing; the distance matrix is filled sequentially row by row from the top most left corner to the bottom right corner (where the total distance is held). Using one step to process both cases (transposition happen case and the not case) is a good way to minimize the number of operations required to accurately compute the distance.

In our modification, the distance matrix is updated directly by one step and the next steps (selecting the minimum and filling the underhand cell) are continued normally as it was done in the original algorithm; such action abstracted the step at line 22 in Algorithm2 which uses more than one operation to complete.

How modifying the Levenshtein method reduced the time and enhanced the candidates generation process is that the modification exploited point1 to make the algorithm avoids checking the cases that are leading to a failure situation, unlike Damerau-Levenshtein modification which makes no difference between the two situations; this is presented in lines 15 and 16.

The directly updated distance matrix (line 17) in the enhanced algorithm has accurately adjusted the distance without any more additional processing; it is simply an assignment.

```
1. Algorithm3: Enhanced Levenshtein Distance
2. Input: String$_1$, String$_2$
3. Output: Damerau Edit Operations Number
4. Step1: Declaration
5.     distance(length of String$_1$,Length of String$_2$)=0, min$_1$=0, min$_2$=0, min$_3$=0, cost=0
6. Step2: Calculate Distance
7. if String$_1$ is NULL return Length of String$_2$
8. if String$_2$ is NULL return Length of String$_1$
9. for each symbol x in String$_1$ do
10.     for each symbol y in String$_2$ do
11.    begin
12.         if  x = y
13.       begin
14.            cost = 0
15.               if x is not the start symbol of String$_1$ then
16.                   if (the symbol preceding x=the symbol following y) and (x is not duplicated) then
17.                    decrease distance (index(x)-1,index(y)) by 1 // transposed
18.       end
19.       else  cost = 1
20.        r=index of x, c=index of y
21.        min$_1$ = (distance(r - 1, c) + 1) // deletion
22.       min$_2$ = (distance(r, c - 1) + 1) //insertion
23.       min$_3$ = (distance(r - 1,c - 1) + cost) //substitution
24.       distance( r , c )=minimum(min$_1$ ,min$_2$ ,min$_3$)
25.   end
26. Step3: return the value of the last cell in the distance matrix
27.         return distance(Length of String$_1$,Length of String$_2$)
28. End.
```

Obviously, the time complexity is related to the real distance between the input strings. However, as the strings becomes more different, the steps at lines 15, 16 and 17 in Algorithm3 are rarely executed which saving time; in turn, this property is preferred in the cases where the algorithm is used for generating candidates.

Candidates should be as similar as possible to the source token (usually, a mistaken word) and the relativity of the additional steps (lines 15, 16 and 17) in the enhanced algorithm made the consumed time to generate candidates is useful (or not wasted) from the view point that those steps are only executed when there is a matching with the source token and they are more executed as the source word being more matched with the target word which means that it is a good candidate.

VI. CASE STUDY

An experiment for testing the real implementation of the three algorithms (Levenshtein, Damerau-Levenshtein and Enhanced-Levenshtein) and showing the variance of time complexity, we have used an English dictionary containing more than $3x10^5$ tokens and a list of 15 misspelled words.

For each misspelled word, we have shown the average time of finding the nearest candidates using each of the previously mentioned three algorithms measured in seconds.

Figure.1 shows the variance in which the Damerau modification consumed longer time than both of original and the enhanced Levenshtein did.

Figure.1 also shows that the enhanced algorithm has a time complexity close to (or on the boundaries of) the original Levenshtein algorithm and this is the goal of the modification. The enhancement performed the task of the Damerau's modification in a time complexity closer to the original algorithm, i.e. $O(N_1.N_2)$.

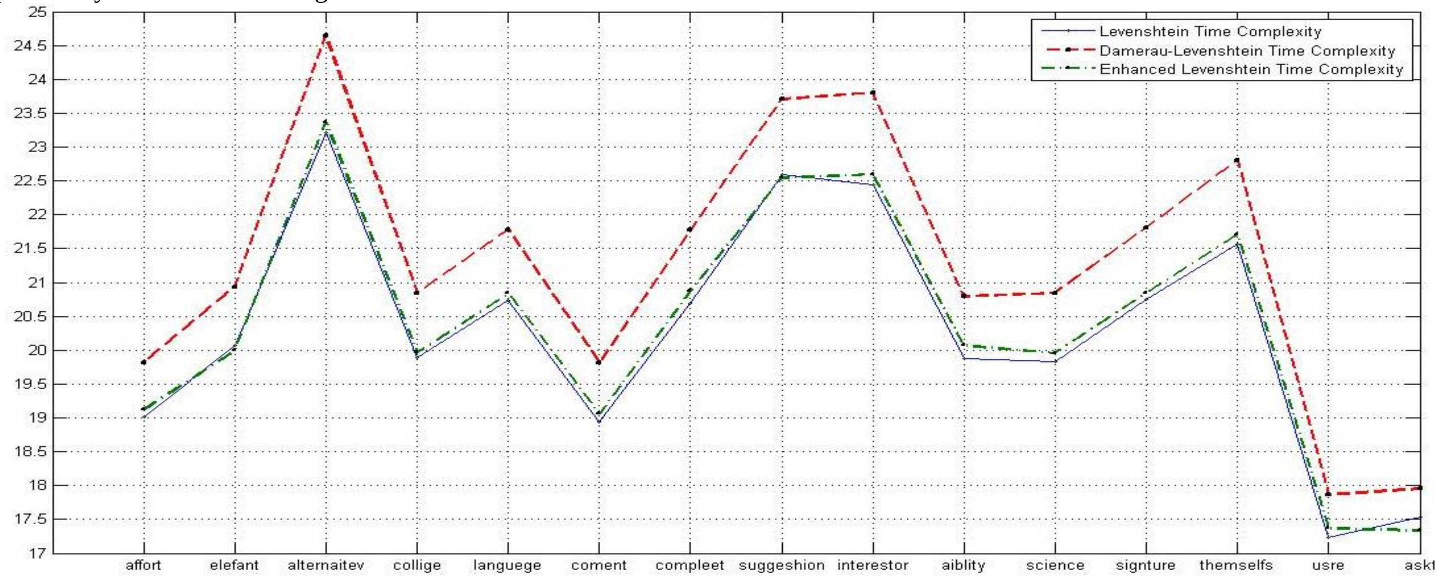

Fig.1: Time complexity variance of Levenshtein, Damerau-Levenshtein, and Enhanced Levenshtein [Y axis represents the consumed time measured in seconds, the X axis shows the samples used for testing]

VII. USING DISTANCE AS A SIMILARITY MEASURE

Minimum Edit Distance methods count the number of edit operations required to convert on string to another in that they are able to find the absolute difference between two strings and therefore they can't find the similarity amount between them.

As an example: the distance between "a" and "b" =1, but the similarity =0; whereas, distance between "Similar" and "Similer" is also 1, but the similarity =6/7 which means that there are six letters matched among seven. However, the difference is the same from the view of Minimum Edit Techniques, just one edit operation.

Another example showing the accuracy of selecting a candidate for the word "correcte", both "correct" and "corrected" are of the same distance (one edit operation: deleting 'e' to generate the first or inserting 'd' at the end to generate the second). This ambiguity makes ranking task more complicated, the similarity can solve the problem by showing how the two candidates "correct" and "corrected" share the misspelled word "correcte" some of its letters.

"correct" shares only 7 letters, while "corrected" shares 8 letters; this variance must give the second candidate higher ranking score because a similarity of 7/8 is smaller than the similarity of 8/9.

Strings lengths should be taken into account when computing the edit distance, then the resulted value is used as a similarity measure. Since the absolute difference between any two strings is added to the total mismatched symbols since it is considered as the number of deleted symbols from the shorter string. The similarity measure must depend on the maximum length between the two.

The absolute difference is directly computed by applying an edit distance method, in this paper the term "distance" refers to any difference value that is received from such methods:

Absolute_Difference=distance(St_1,St_2) … (1)

The relative distance is another view for the difference where a consideration for the foundation of ration between the number of edit operations required to make the matching and the total letters found in both input strings.

Hence, relative distance is computed by:

R_Dist(St_1,St_2)=
Absolute_Difference* / *max(length(St_1),length(St_2)) … (2)

Relative distance is a value within the interval (0,1) where completely different strings have a relative distance of 1; and as its value decreases, the difference is also decreases until reaching the value of 0 when the two strings are identical.

Since the similarity and difference are complements to each other, the similarity can be computed by:

Similarity (St_1, St_2)=1- R_Dist(St_1,St_2) … (3)

And the later is the measure of similarity used in the candidates' generation for this work.

VIII. CONCLUSION

Using minimum edit distance techniques for error correction is an efficient way specially in the fields of isolated words correction since they are fully dependent on performing the matching on the source word and a list of alternatives without any considerations for further constraints (such as context, position within sentence, frequency, …). Levenshtein method is one of those techniques which can identify three types of edit operations (deletion, insertion and substitution) but not the fourth type: the transposition of two adjacent symbols.

In this paper, a modification on the Levenshtein method was done to complete its work within a time complexity close to the unmodified method. Because of the algorithm suitability, it is used for candidates generation and therefore a modulation was required to convert the difference measure into a similarity measure. The resulted measure is suitable for every distance method specifically for those which work with strings.

Although the modified method showed an accepted execution time, it is still of a quadratic complexity. In the future, there is a necessity for further enhancing the method to predict the exact edit distance without performing all the matching steps.

REFERENCES

[1] R. Mishra and N. Kaur, "A Survey of Spelling Error Detection and Correction Techniques," *International Journal of Computer Trends and Technology,* vol. 4, no. 3, pp. 372-374, 2013.

[2] Manning, Raghavan and Schútze, "An Introduction to Information Retrieval," Cambridge University Press, 2008.

[3] L. Salifou and H. Naroua, "Design of A Spell Corrector For Hausa Language," *International Journal of Computational Linguistics (IJCL), Volume.5 : Issue 2,* pp. 14-26, 2014.

[4] F. J. Damerau, "A Technique for Computer detection and Correction of Spelling Errors," ACM, New York, 1964.

[5] R. Haldar and D. Mukhopadhyay, "Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach," ACM, New York, 2011.

[6] V. I. Levenshtein, "BINARY CODES CAPABLE OF CORRECTING DELETIONS, INSERTIONS, AND REVERSALS," *SOVIET PHYSICS - DOCKLADY,* vol. 10, no. 8, pp. 707 - 710, February 1966.

[7] S. Mihov, S. Koeva and others, "Precise and Efficient Text Correction using Levenshtein Automata,Dynamic Web Dictionaries and Optimized Correction Models," 2004.

[8] K. L. Tommi A. Pirinen, "Finite-State Spell-Checking with Weighted Language and Error Models—Building and Evaluating Spell-Checkers with Wikipedia as Corpus," in *proceedings of LREC 2010 workshop on creation and use of basic lexical resources for less-resourced languages*, 2010.

[9] Y. Bassil, "Parallel Spell-Checking Algorithm Based on Yahoo! N-Grams Dataset," *International Journal of Research and Reviews in Computer Science (IJRRCS),* vol. 3, no. 1, pp. ISSN: 2079-2557, 2012.

[10] I. Setiadi, "Damerau-Levenshtein Algorithm and Bayes Theorem for Spell Checker Optimization," Makalah IF2211 Strategi Algoritma, Bundang, 2014.