 http://dx.doi.org/10.14500/aro.10180

A Comparative Study for String Metrics and the Feasibility of Joining them as Combined Text Similarity Measures

Safa S. Abdul-Jabbar and Loay E. George

Department of Computer Science, College of Science
Baghdad University, Baghdad, Iraq

***Abstract*–This paper aims to introduce an optimized Damerau–Levenshtein and dice-coefficients using enumeration operations (ODADNEN) for providing fast string similarity measure with maintaining the results accuracy; searching to find specific words within a large text is a hard job which takes a lot of time and efforts. The string similarity measure plays a critical role in many searching problems. In this paper, different experiments were conducted to handle some spelling mistakes. An enhanced algorithm for string similarity assessment was proposed. This algorithm is a combined set of well-known algorithms with some improvements (e.g. the dice-coefficient was modified to deal with numbers instead of characters using certain conditions). These algorithms were adopted after conducting on a number of experimental tests to check its suitability. The ODADNN algorithm was tested using real data; its performance was compared with the original similarity measure. The results indicated that the most convincing measure is the proposed hybrid measure, which uses the Damerau–Levenshtein and dice-distance based on n-gram of each word to handle; also, it requires less processing time in comparison with the standard algorithms. Furthermore, it provides efficient results to assess the similarity between two words without the need to restrict the word length.**

***Index Terms*–Word classification, Word clustering, String distance, String matching operation, and String similarity metric.**

I. Introduction

To find the similarity ratio between strings, many comparing operations should be used, this subject considered as a basic task in natural language processing (NLP), as well as other disciplines such as computational biology. In NLP, the sequences of symbols are composed of a number of sentences, consisting of words. In the first approximation (such as applications in speech recognition), sentences are considered to be more similar to the more words they share and the reordering is no consideration. While in the second approximation (such as Grammar induction), the reordering of single words and blocks between two sentences can be expected (Leusch, et al., 2003; Mohri, 2003).

Many applications require string search with errors possibility. These applications should use a matching function to the user entry (which may contain an incorrect spelling) in the database. This operation should be done in milliseconds (Fenz, et al., 2012).

The problem underlying the searching operation, measuring the similarity or dissimilarity of two strings, had been a powerful topic of research for over five decades, ranging from early operations to modern machine learning and data analysis. Each method uses different aspects and characteristics of the data (Rieck and Wressnegger, 2016).

Various similarity measures were proposed for use in various fields: Damerau and Levenshtein introduced a method named Damerau–Levenshtein that used as a string metric between two strings. By counting the minimum number of operations needed to transform one string to the other, through measuring the substitution operations of a single character besides the insertion, deletion, or transposition operation of two adjacent characters (provided by the Levenshtein distance) (Damerau, 1964). These measures are based on probabilistic modeling for a particular applied instance. For example, in error correction of noisy sentences (Kashiap and Oommen, 1984; Oommen, 1987) and in recognition tasks (Marzal and Vidal, 1993; Bunke and Bühler, 1992; Cortelazzo, et al., 1996; Cortelazzo, et al., 1994; Peng and Chen, 1997); Winkler had proposed an enhancement to the Jaro metric based on the observation that spelling errors may occur commonly at the end of a string (Winkler, 1999). While the N-gram techniques can determine the similarity between strings from given text sequence by computing the similarity, on the basis of the distance between each character in the compared two strings. This distance is computed by dividing the number of similar grams by the maximal number of n-grams (Alberto, et al, 2010).

Sehgal, et al. (2006) compared three string similarity measures on a data integration task; they referred that edit distance is better than Jaccard and Jaro-Winkler when mapping between two sets of place names in Afghanistan. Martins (2011) used machine learning to classify gazetteer records as

ARO-The Scientific Journal of Koya University
Volume V, No 2(2017), Article ID: ARO.10198, 13 pages
DOI: 10.14500/aro.10180
Received 03 December 2016; Accepted 09 September 2017
Regular research paper: Published 21 October 2017
Corresponding author's e-mail: safasami1988@scbaghdad.edu.iq

duplicates or non-duplicates and compared the importance of several feature types, including eight string similarity measures. The experimental results show that using feature vectors which combined from (place names, semantic relations, place types, and geospatial footprints) leads to an increase in the results accuracy. Wang, et al. (2014) proposed a new hybrid similarity metrics, called "fuzzy token matching based similarity," which extends token-based similarity functions (e.g., Jaccard similarity and Cosine similarity) by allowing the fuzzy match between two tokens. They considered as new signature schemes and develop effective techniques to improve the performance.

Different measures of distance or similarity are convenient for different types of analysis:

1- String Similarity: Defines a similarity between two strings (0 means strings are completely different, 1 means strings are identical) like Sorensen–Dice coefficient (Dice, 1945).
2- String Distance: Defines a distance between two strings (0 means strings are identical), like Damerau–Levenshtein. The maximum distance value depends on each algorithm (Sellers, 1980; Hall and Dowling, 1980).

In this paper, many measures were implemented to make a decision about which one was more suitable to use. The implemented algorithms are listed in Table I. While in Table II, the comparison of different similarity metric methods was described in the context of their advantages and weak points.

These methods can be merged to provide fast retrieval systems, using the symbols enumeration operation for handling the string operations as a sequence of numbers instead of a sequence of characters to reduce the hidden cost of the string operations; this will reduce the memory, time, and CPU consumption.

Table I
The Studied String Metric Methods

Method name	Type
Levenshtein	Distance
Damerau–Levenshtein	Distance
Longest common subsequence	Distance
Jaro–Winkler	Similarity measures
N-Gram	Distance
Dice coefficient	Similarity measures
Matching coefficient	Similarity measures
Overlap coefficient	Similarity measures

II. Materials and Methods

In this paper, many different metrics were explored to decide which one is suitable for string-matching purpose depending on the elapsed time with respect to the result accuracy. Furthermore, a set of hybrid algorithms was made up using several existing measures with a simple modification. According to the conducted comparisons between eight string distance/similarity for evaluating them in terms of the consumed time; a brief summary of each one is presented in this section.

The experiments for these algorithms were involved with words of length (1-16) characters only. The conducted statistical analysis of the used datasets showed that approximately 99% of the overall words in each dataset are available in this range of word length; as depicted in Table III.

The number of comparisons for each given word will be reduced using a specific threshold based on the word length. The process of selecting the threshold was treated as follows:

- For words that have length ≤5, the threshold=1) the comparison operations were made with only words that have the length equal ±1 to the length of the given word).
- For words that have length ≥6, the threshold=2) the comparison operations were made with words that have the length equal ±2 to the length of the given word).

Which means the types that are processed in the proposed system are limited in two types: Words with length ≤5 have the possibility of one error only, whereas words with length >5 allow errors with two letters as the maximum probability. Then, to get the best system performance, the proposed system used the integrated number of similarity measures which have proven successful through experiments. These

Table II
Comparison of Different Similarity metrics methods

Method name	Advantage	Disadvantage
Levenshtein and Damerau–Levenshtein	Gives the best result in case of short string and it is fast and best suited for strings similarity (Pradhan, et al., 2015; Patel, 2016)	In case of long string cost of Levenshtein distance is same as the length of string and considered it is not order of sequence of characters while comparing (Pradhan, et al., 2015; Patel, 2016)
Longest common subsequence	-	Uses the recursion approach which uses stack that takes lots of space (Pradhan, et al., 2015)
Jaro–Winkler	Gives better result in case of hybrid method (Pradhan, et al., 2015)	If the data size is too much large, then Jaro distance similarity not gives efficient results (Pradhan, et al., 2015)
N-gram	Similarity technique is high (Pradhan, et al., 2015)	They are not suitable at multilingual environment, and the accuracy is very less (Pande, et al., 2013; Pradhan, et al., 2015)
Dice coefficient	Obtain satisfactory results and used to consider the sizes of the two words and the similarity score will be normalized into [0,1] (Pradhan, et al., 2015)	-
Matching coefficient	Very simple vector-based approach which simply counts the number of similar terms (dimensions) (Gomaa and Fahmy, 2013)	If one of these dimensions is zero, this method cannot work efficiently (Gomaa and Fahmy, 2013)
Overlap coefficient	Similar to the Dice's coefficient, but considers two strings a full match if one is a subset of the other (Gomaa and Fahmy, 2013)	-

Table III
The Words Count within the Considered 4 Datasets of Complete Words whose Lengths bounded between [1,16] Characters

Dataset#	No. of words in the overall dataset	No. of words from 1 to 16 char.
Dataset 1	530421873 (%100)	529051745 (%99.74)
Dataset 2	63948272 (%100)	63944562 (%99.99)
Dataset 3	246650908 (%100)	246598321 (%99.97)
Dataset 4	3455357163 (%100)	3452403297 (%99.88)

combined measures were used for measuring the string distance between pairs of strings. The considered measures are:

- Dice coefficient and N-gram (DN).
- Dice coefficient, N-gram, and Damerau–Levenshtein (DADN).
- Damerau–Levenshtein and longest common subsequence (DAL).

A. DN Measure

It is obtained by integrating the N-gram measure with the dice-coefficient measure to increase the similarity results accuracy. Algorithm (1) illustrates the implementation steps for DN.

Algorithm (1): DN Algorithm

Objectives: Measuring the similarity between two given strings.
Input: Variable number of input words ($string_1$), Text File.
Output: Integer value, words List contain the same number of input words, which is the most similar words from files.

```
Step1:
  String [] s1 → string1.Split(' ')  //Read the given string
                                     and split it to words array
                                     using the space delimiter
  Double len → 0, Double total_len → 0
  Double temp → -1, Int g→0
Step2:
  StreamReader sr →new StreamReader(ss)
//Read                                          the   file
content
  line = sr.ReadToEnd();
  string[] words = line.Split(' ');  //Split the file content in
                                     to words using the space delimiter
  For i = 0 to s1.Length-1 step 1 do
    For j = 0 to words.Length-1 step 1 do
      If (((s1[i].Length >= 1) & (s1[i].Length <= 5)) &
      ((s1[i].Length <= words[j].Length + 1) && (s1[i].
      Length >= words[j].Length-1))) then//For words
      that have length0-5 the threshold of error is 1 char.
      temp → DiceCoefficient(s1[i], words[j])
      If (len == 0) then len=temp, g = j
      Else If (temp > len) then len = temp, g = j
      End If
  Else If (((s1[i].Length >= 6) & (s1[i].Length <= 18)) &
  ((s1[i].Length <= words[j].Length + 2) & (s1 [i].Length >=
  words[j].Length - 2))) then//For words that more than 5
  characters the threshold of error is 2 char.
  temp → DiceCoefficient(s1[i], words[j])
      If (len == 0) then len=temp, g = j
      Else If (temp > len) then len = temp, g = j
      End If
    End If
    temp →0
  End For
  total_len = total_len + len//For collect distance of all
                                  words in the given string
  list1.Items.Add(words[g]);
  len → -1, temp → -1
  End for
Step3:
  Int Result_Distance= total_len/s1.Length
End;
```

```
Function1: Double DiceCoefficient(string stOne, string stTwo)//
    For strings instead of char comparing using words of two
    characters
  List<string> nx, ny; string temp = ""
  For i = 0 to stOne.Length – 2 step 1 do
      temp = "" + stOne[i] + stOne[i + 1]: nx.Add(temp)
  End For
  For j = 0 to stTwo.Length – 2 step 1 do
      temp = "" + stTwo[j] + stTwo[j + 1]: ny.Add(temp)
  End For
  If (stOne.Length == 1)//For handling words with one
  character
      temp = "" + stone, nx.Add(temp)
  End if
  If (stTwo.Length == 1)//For handling words with one
  character
      temp = "" + stTwo, ny.Add(temp)
  End if
  HashSet<string> intersection = new HashSet<string>(nx)
  intersection.IntersectWith(ny), double dbOne =
  intersection.Count//Determine the intersection between
  words
Return (2 * dbOne/(nx.Count + ny.Count))
```

B. DADN Measure

It is obtained by integrating the previous mentioned DN algorithm with Damerau–Levenshtein distance measure to increase the result accuracy; this integration aimed to take the advantage of Damerau–Levenshtein efficacy and speed. Then, handling the situation of equal single character movement results with a DN measure to decide which string is more similar to a given one. Algorithm (2) illustrates the implemented steps for DADN.

Algorithm (2): DADN Algorithm

Objectives: Measuring the similarity between two given strings.
Input: Variable number of input words ($string_1$), Text File.

Output: Integer value, List of words contains the same number of input words, which is the most similar words from files.

Step1:
String [] s_1 → $string_1$.Split(' ')//Read the given string and split it to words array using the space delimiter
Double len → 0, **Double** total_len → 0//Define the variables
Double temp → -1, **Int** g →0
DamerauLevensteinMetric da=newDamerauLevenstein Metric()//Define the object da as class of Damerau Levenste in Metric
Step2:
StreamReader sr →**new** StreamReader(ss), line = sr.ReadToEnd()//Read the file content
String[] words = line.Split(' ')//Split the file content in to words using the space delimiter
For i = 0 **to** s_1.Length-1 **step** 1 **do**
For j = 0 **to** words.Length-1 **step** 1 **do**
If ((s_1[i].Length >=1) & (s_1[i].Length<=5)) & ((s_1[i].Length<=words[j].Length+1) & (s_1 [i].Length>=words[j].Length - 1))) **then**//
For words that have length 0-5 the threshold of error is 1 character
temp → da.GetDistance(s1[i], words[j], 100)
If (len == 0) **then** len → temp, g → j
If (temp < len) **then** len → temp, g → j
else If ((temp==len) & (j!=g) & (len!=0)) **then**
Double one → DiceCoefficient(s1[i], words[g]), **Double** two → DiceCoefficient(s1[i], words[j])
If (one > two) **then** g = j
End If
else If ((s_1[i].Length>=6) & (s_1[i].Length<=18)) & ((s_1[i].Length<=words[j].Length+2) & (s_1 [i].Length >= words[j].Length - 2))) **then**//For words that have more than 5 char. the threshold of error is 2 char.
temp → da.GetDistance(s1[i], words[j], 100)
If (len == 0) **then** len=temp, g = j
If (temp < len) **then** len = temp, g = j
else If ((temp == len) & (j!= g) & (len != 0)) **then**
Double one → DiceCoefficient(s1[i], words[g]), **Double** two → DiceCoefficient(s1[i], words[j])
If (one > two) **then** g = j
End If
End If
temp →0
End For
total_len = total_len + len//For collect distance of all words in the given string
list1.Items.Add(words[g]), len → -1, temp → -1
End for
Step3:
Int Result_Distance= total_len/s_1.Length
End;

Function1: Double DiceCoefficient(string stOne, string stTwo)// The Same Function Steps (used to compare between two strings using Dice-Coefficient and N-gram) as in Algorithm(1)
Class DamerauLevensteinMetric
Const Int DEFAULT_LENGTH → 255
Int [] currentRow, previousRow, transpositionRow//Define the variables
Double GetDistance(**String** first, **String** second, **Int** max)//Max is the threshold of movements number
Int maxLength → DEFAULT_LENGTH//Maximum number of characters in each word
currentRow → **new Int**[maxLength +1], previousRow → **new Int**[maxLength +1],
transpositionRow →**new Int**[maxLength +1]
Int firstLength → first.Length, **Int** secondLength → second. Length//2Variable to store the length of string1 &2
If (firstLength == 0) **then Return** secondLength//If string1 was empty return the number of char. in string2
If (secondLength==0) **then Return** firstLength//If string2 was empty return the string1 length
If (firstLength > secondLength) **then**//Swap between string to make the second string with larger length and swap length
Swap (first, second), firstLength → secondLength
End If
If (secondLength - firstLength > max) **Return** max + 1// If the different is larger than threshold the return threshold+1
If (firstLength > _currentRow.Length) **then**
currentRow = **new Int** [firstLength + 1], previousRow = **new Int** [firstLength + 1],
transpositionRow = **new Int**[firstLength + 1]
End If
For i = 0 **to** firstLength **step** 1 **do**//As an initial value store the counter value then use this array to store the value of movements for each step(character)
previousRow[i] → i
End For
Char lastSecondCh → '\0'//The last used char in the second string
For i = 1 **to** secondLength **step** 1 **do**
Char secondCh → second[i - 1], currentRow[0] → i//Compute only diagonal stripe of width 2*(max+1)
Int from → Max(i - max - 1, 1), **Int** to → Min(i + max + 1, firstLength)//The start & end positions for checking process
Char lastFirstCh → '\0'//The last used char from first string
For j = from **to** to **step** 1 **do**
Char firstCh = first[j - 1]//Compute minimal cost of state change to current state from previous states of deletion, insertion and swapping
Int cost = 0
If (!(firstCh == secondCh)) **then** cost = 1
Int value = Min(Min(currentRow[j -1] + 1, previousRow[j]+1), previousRow[j -1]+ cost)
//If there was transposition, take in account its cost only if the transposed characters are adjacent
If (firstCh == lastSecondCh && secondCh ==

```
            lastFirstCh) then
    value = Min(value, _transpositionRow[j - 2] + cost),
    currentRow[j] = value, lastFirstCh = firstCh
    End if
End For
    lastSecondCh = secondCh, Int[] tempRow =
    transpositionRow, transpositionRow = previousRow
    previousRow = currentRow, currentRow = tempRow
  End For
  Return previousRow[firstLength]
End Class
```

C. DAL Measure

It is obtained by integrating Damerau–Levenshtein distance measure with the longest common subsequence measure. The longest common subsequence algorithm is used to handle the advantage of Damerau–Levenshtein efficacy and speed. It handles the cases of similar results for several words in the file to a given word, and hence, to decide which string is more similar to a given one. This measure checks the similar character sequence and takes the word with larger values of the sequence. Algorithm (3) presents the implementation steps for DAL.

Algorithm (3): DAL Algorithm

Objectives: Measuring the similarity between two given strings.
Input: Variable number of input words ($string_1$), Text File.
Output: Integer value, List of words containing the same number of input words, which is the most similar words from files.

```
Step1:
    String [] s1 → string1.Split(' ')//Read the given string and
    split it to words array using the space delimiter
    Double len → 0,Double total_len → 0, Double temp
    → -1,Int g→0//define the variables
    DamerauLevensteinMetric da=new DamerauLevenstein
    Metric()//define the object da as class of Damerau
    LevensteinMetric
Step2: StreamReader sr →new StreamReader(ss), line =
sr.ReadToEnd()//Read the file content
    String[] words = line.Split(' ')//Split the file content in to
    words using the space delimiter
    For i = 0 to s1.Length-1 step 1 do
    For j = 0 to words.Length-1 step 1 do If((s1[i].
    Length>=1)&(s1[i].Length<=5))&((s1[i].Length<=words[j].
    Length+1)&
    (s1[i].Length>=words[j].Length-1)) then//For words that
    have length 0-5 the threshold of error is 1char
    temp → da.GetDistance(s1[i], words[j], 100)
    If (len == 0) then len → temp, g → j
    If (temp > len) then len → temp, g → j
    Else If ((temp==len) & (j!=g) & (len! 0)) then//For words
    that have length 0-5 the threshold of error is 1 char.
            Double one → LongestCommonSubsequence(s1[i],
            words[g])
            Double two → LongestCommonSubsequence (s1[i],
            words[j])
            If (one > two) then g = j
    End If
Else If(((s1[i].Length>=6) & (s1[i].Length<=18)) & ((s1[i].
        Length<=words[j].Length+2) &
        (s1 [i].Length>=words[j].Length - 2))) then//For words
        that more than 5 char. the threshold of error is 2 char.
            temp → da.GetDistance(s1[i], words[j], 100)
            If (len == 0) then len=temp, g = j
            If (temp > len) then len = temp, g = j
            Else If ((temp == len) & (j!= g) & (len != 0)) then
            //For words that have length 0-5 the threshold of
            error is 1 char.
            Double one → LongestCommonSubsequence (s1[i],
            words[g])
            Double two → LongestCommonSubsequence (s1[i],
            words[j])
            If (one > two) then g = j
    End If
End If
    temp →0
    End For
    total_len = total_len + len//For collect distance of all
    words in the given string
    list1.Items.Add(words[g]), len → -1, temp → -1
    End for
Step3:
    Int Result_Distance= total_len/s1.Length, list1.show
End;
```

```
    Function1: Int LongestCommonSubsequence (String str1,
                String str2)
      String sequence → ""
      If ((str1.Length == 0) | (str2.Length == 0)) then Return
      0
      Int [,] num = new int[str1.Length, str2.Length]//Array
      used for count the number of identical char. in the given
      strings
      Int maxlen → 0, Int lastSubsBegin → 0, String
      sequencestring → ""
      For i = 0 to str1.Length-1 step 1 do
        For j = 0 to str2.Length step 1 do
          If (str1[i] != str2[j]) then num[i, j] = 0
          else If ((i == 0) || (j == 0)) then num[i, j] = 1//
          Every time check characters from it arrived to the
          end of char start from 1 for counter
          else num[i, j] → 1 + num[i - 1, j - 1]
          End If
    If (num[i, j] > maxlen)
      maxlen → num[i, j]
      Int thisSubsBegin → i - num[i, j] + 1
    If (lastSubsBegin == thisSubsBegin)
      sequencestring → sequencestring + str1[i]
    End If
    else//This block resets the string builder if a different LCS
          is found
    lastSubsBegin → thisSubsBegin, sequencestring → ""//Clear it
    sequencestring → sequencestring + str1. Subsequence
    (lastSubsBegin, (i + 1) - lastSubsBegin)
```

```
      End If
    End If
  End For
End For
sequence → sequencestring
Return maxlen
Class DamerauLevensteinMetric//The Same Class Steps as in
                                Algorithm(2)
```

After identifying the best algorithm, DADN, we modified it to deal with numbers rather than strings where this modification will produce a new method entitled dice coefficient, N-gram, and Damerau–Levenshtein using enumeration method (DADNEN). The test results of this algorithm indicated that the modified algorithm has a positive impact on the results for words with length ranging between 1 and 13, but has no effect on the words with the length which is equal or larger than 14. Hence, some conditions were used in the modified algorithms to control the performance of this algorithm with a wide range of word lengths (i.e., making the system work flexibly with all word lengths as much as possible). Algorithm (4) illustrates the implemented steps of the modified optimized DADNEN (ODADNEN) algorithm.

Algorithm (4): ODADNN Algorithm

Objectives: Measuring the similarity between two given strings.
Input: Variable number of input words ($string_1$), Text File.
Output: Integer value, List of words contains the same number of input words, which is the most similar words from files.

```
Step1: Define The Variables
  Byte[] bytes = Get Bytes(string1)//Read the given string
        and split it to words array using the space delimiter
  Int word_length → 0//Start and end and length for each
                      word in the input strings
  Int start → 0, Int end→0, Double len → -1//Number of
                movements required for each two strings
  Double total_len → 0//The total number of movements
for                                  all the given strings
  DamerauLevensteinMetricen da = new DamerauLevenste
  inMetricen()
                  //Define the object da as class of Damerau
                    Levenstein Metric
Step2: Read text files as blocks of bytes.//For each Text file
    read its content as blocks of bytes with size about 4 MB for
    each block till reaching the end of file
Step3: Count Length for each word.//Determine the start (s)
    and the end (e) of each word (array of bytes) for each word
    in the given string (bytes)
Step4:
  For Each Word in bytes
    For j = 0 to bytes2.Length-1 step 1 do
      While (end2 < bytes2.Length & (bytes2[end2] != 32))
        end2++
  End while
  wordfile_length[no] → end2 - start2 - 1
  If (wordfile_length[no] <= 18)
  (C, b)→ array of bytes [wordfile_length[no] + 1]
  Buffer.BlockCopy(bytes2, start2, b, (18 * no)
  wordfile_length[no] + 1)//To determine the start of each row
  Buffer.BlockCopy(bytes2, start2, c, 0, wordfile_length[no] + 1
  If ((word_length<=5) & (word_length<=wordfile_
  length[no]+1) & (word_length>= wordfile_length[no]-1))
  then//for words that have length 0-5 the threshold of error
  is 1 char.
  temp → da.GetDistance(a, c, 100)
  If (len == -1) then len →temp, g →no
  If (temp < len) then len → temp, g → no
  Else If ((temp == len) & (no != g) & (len != 0)) then
  Double one → DiceCoefficient(a, c), Double two →
  DiceCoefficient(a, c)
  If (one > two) then g = no
  End If
Else If (((word_length >= 6) & (word_length <= 18)) &
    ((word_length <= wordfile_length[no] + 2) & (word_length
    >= wordfile_length[no] - 2))) then
  //for words that have length larger 5 the threshold of error
is 2 char.
  If (s1[i].Length < 14) then//For decide if there is a need to
  use the enum methods according to the words length.
    Byte[] bytes1 = GetBytes(s1[i]), byte[] bytes2 =
    GetBytes(words[j])
    temp = da.GetDistanceen(bytes1, bytes2, 100)
    If (len == -1) then len = temp, g = j
    If (temp < len) then len = temp,g = j
    else if ((temp == len) & (j != g) & (len != 0)) then
        double one = DiceCoefficient(s1[i], words[g])
        double two = DiceCoefficient(s1[i], words[j])
        If (one > two) then g = j
  end if
else              //For words with length equal or larger than 14
    temp = da.GetDistance(s1[i], words[j], 100)
    If (len == -1) then len = temp, g = j
    If (temp < len) then len = temp,g = j
  else if ((temp == len) & (j != g) & (len != 0)) then
        double one = DiceCoefficient(s1[i], words[g])
        double two = DiceCoefficient(s1[i], words[no])
        If)one > two) then g = j
     end If
   end If
end If
  temp → -1, start2 → end2 + 1, j →start2, end2 → start2, no++
Next j
total_len → total_len + len, Int s → 0
For h = 0 to wordfile_length[g] step 1 do
  s → s + Convert.ToChar(b[g, h])
  Next h
  list1.Items.Add(s), len → -1, temp → -1, start → end + 1,
  i → start, end→ start
Step5:
  Int Result_Distance= total_len/s1.Length
End;
```

Function1: Double DiceCoefficient(string stOne, string stTwo)//The Same Function Steps (used to compare between

http://dx.doi.org/10.14500/aro.10180

two strings using Dice-Coefficient and N-gram) as in Algorithm(1)

Class DamerauLevensteinMetric//The Same Class Steps as in Algorithm(2) with one different, which convert all Strings to an array of bytes and deal with it on that basis

III. Results and Discussion

In this paper, all algorithms were implemented using C sharp 2015 programming language and applied on CPU 2.60 GHz with 16 GB RAM. For measuring the distance between two strings, many algorithms were tested to determine the most efficient one according to the elapsed time for each one. To test the system performance a text file that has size 171 KB which containing 15593 non-repeated words with lengths ranging from 1 to 16 characters; it was extracted from Oxford University Text Archive. This archive was designed to represent a wide cross-section of current British English (Burnard, 1976). In this paper, some of the non-repeated words were extracted from this dataset to test the system performance by typing ten words with different lengths for each user query.

The conducted test includes words have lengths ranging from 1 to 16, but for saving the space of this article only two lists of results are presented. Table IV lists the elapsed time for each method tested to the string distance measuring process; these methods were tested on 10 words with the same length in each time ranging from 1, 2, and 3. Table V lists the elapsed time for each method tested to the string distance measuring process; these methods were tested on 10 words with same length=16. Furthermore, for comparison purpose, the list given in Table VI shows the improved elapsed processing time ratio between the similar types of methods to find the best one for each type; it was computed using the following equation:

$$\text{Speed ratio}=100*(T_1-T_2)/T_1 \qquad (1)$$

Where T_1 is the elapsed time for first method and T_2 for the second one.

Then, the measures showed best similarity results are combined to increase the accuracy by overcoming the cases of similar comparison results for many words. These combined measures were used for measuring the string distance between pairs of strings. The considered measures are:

- DN: It was obtained by integrating the N-gram method with the dice coefficient method to increase the results accuracy by making use of the sequence of letters in the given words.
- DADN: It was obtained by integrating the previous mentioned DN method with Damerau–Levenshtein distance method to increase the result accuracy; this integration is aimed to take the advantage of Damerau–Levenshtein efficiency and speed. Then, handling the situation of equal single character movement results with a DN method to decide which string is more similar to a given one.
- DAL: It is obtained by integrating Damerau–Levenshtein distance with the longest common subsequence. The longest common subsequence is used to handle the advantage of Damerau–Levenshtein efficiency and speed. It handles

Table IV
The 8 of the String Metrics Method Tested using 10 Words with Different Lengths (1, 2 and 3 Characters) and Multiple Changes

Method name	Word length	1		2				3				
	Input status	Complete words	Complete words	Delete one char.	Exchange two char's	Error typing in one char.	Insert one char.	Complete words	Delete one char.	Exchange two char's	Error typing in one char.	Insert one char.
Jaro–Winkler distance*	Time (µs)	8017.9	8004.5	6004.3	7003.8	9023.7	6003.9	9991.7	7004.2	7005.8	8018.7	9006
	Distance	0	0	0.015	0.033	0.015	0.0178	0	0.0089	0.0175	0.0178	0.0067
Longest common substring**	Time (µs)	7003.8	8003.7	8006.1	6003.5	13009.2	8004.9	11007	12027.5	10973.9	12007	11005.1
	Similarity	10	20	19	19	19	20	30	29	29	29	29
Levenshtein distance***	Time (µs)	7023.5	11007.8	11026.8	12009	10007.1	10006.7	14009.9	12027.5	14010.4	13995	13009.2
	Movements	0	0	1	1	1	1	0	1	2	1	1
Damerau–Levenshtein distance***	Time (µs)	9005.2	11007	10007.5	18012.8	11007.8	14009.9	16011.3	23016.4	14993.3	17013	16013
	Movements	0	0	1	1	1	1	0	1	1	1	1
N-Gram****	Time (µs)	5003.2	11007.4	9988.5	7002.6	6003.4	5007.1	10005.5	10005.5	9006.4	10025	7996.6
	Similarity	0	10	9	9	9	10	20	20	19	19	19
Dice coefficient*****	Time (µs)	4985	8004.9	7004.6	7022.7	9002	8005.3	9006	16011.7	10006.8	7003.8	9023
	Distance	1	1	0.9667	0.95	0.95	0.9667	1	0.98	0.9333	0.9667	0.95
Overlap coefficient*****	Time (µs)	6003.5	7005.3	9006	7005.8	8005.3	10024.5	11006.6	13007.6	10025.3	13010	10008.3
	Distance	Division by zero	1	Division by zero	0.95	1	0.967	1	1	0.967	0.967	0.967
Matching coefficient******	Time (µs)	5001.6	9008	8005.3	12008.1	9006.8	8024.3	10006.7	13004.9	9006	9023	9006
	Distance	0.5	0.5	0.483	0.475	0.483	0.483	0.5	0.49	0.479	0.483	0.479

*Match=0, Not match=1, **Match=No. of all input char. For all words, No Match=0, ***Match=0, No Match=No. of all input char. for all words, ****Match=No of identical pairs=Σ(each word length−1), No match=0, *****Match=1, No Match=0, ******Match=0.5, No Match=0

the cases of similar results for several words in the file to a given word, and hence, to decide which string is more similar to a given one. This measure checks the similar character sequence and takes the word with larger values of the sequence.

The suggested measures were tested on 10 words with the same length in each time; they range from 1 to 16, as shown in Figs. 1-5, for finding the most appropriate algorithm for using in the next steps. The results showed that applying the DADN algorithm is the fastest one with maintaining the similarity results accuracy; the best algorithm can be selected by calculating the accumulative values for all execution times which obtained in the tests; that is, DDN=12032518.4 (μs), DADN=9673223 (μs), and DAL=9915123 (μs). The accumulative results showed that DADN algorithm has the smallest execution time. After identifying DADN as the best algorithm, we have modified it to deal with numbers rather than strings which we called it DADNEN for speeding up the process. The test results of this algorithm indicated that the modified algorithm has the positive impact on the results for words with length ranging between 1 and 13 but has no effect on the words with length more that. Hence, some conditions were used in the modified algorithms to control the performance of this algorithm, ODADNEN, with a wide range of word lengths (i.e., making the system works flexible with all word lengths as much as possible). The elapsed time for the tested results for (DADNEN, ODADNEN) algorithms, which select the most similar word for each word from the given word, are shown in Table VII and Table VIII.

Table V
The 8 of the Distance String Metrics Methods Tested using 10 Words 16 Characters and Multiple Changes

Method name	Word length	16				
	Input status	Complete words	Delete one char.	Exchange two char's	Error typing in one char.	Insert one char.
Jaro–Winkler distance*	Time (μs)	76053.3	73032.2	76071	73051.1	71050.5
	Distance	0	0.0013	0.0013	0.0013	0.0132
Longest common substring**	Time (μs)	129091	143120.3	137078.1	139098.5	116082.6
	Similarity	160	153	152	152	158
Levenshtein distance***	Time (μs)	140079.9	146105.4	134087.1	147121.2	127108.1
	Movements	0	1	2	1	1
Damerau–Levenshtein distance***	Time (μs)	160113.8	172122.7	171121.2	169119	142080.9
	Movements	0	1	1	1	1
N-gram****	Time (μs)	41032.7	42030.7	40026.5	39033.3	37026.7
	Similarity	150	140	147	142	139
Dice coefficient*****	Time (μs)	40046.6	41046.9	41027.2	42012.1	48033.4
	Distance	1	0.92	0.938	0.92	0.9235
Overlap coefficient*****	Time (μs)	41045.3	41033.1	40045	42028.7	46031.6
	Distance	1	0.943	0.988	0.953	0.941
Matching coefficient******	Time (μs)	43048.7	56033.9	39013.9	40027.7	44049.4
	Distance	0.5	0.471	0.494	0.476	0.471

*Match=0, Not match=1, **Match=No. of all input char. For all words, No Match=0, ***Match=0, No Match=No. of all input char. for all words, ****Match=No of identical pairs=Σ(each word length−1), No match=0, *****Match=1, No Match=0, ******Match=0.5, No Match=0

Table VI
The Improvement Ratio of the Elapsed Processing Time Measured between the Previously Tested Methods using 10 Words with Different Lengths (1, 2, 3, 16 Characters) and Multiple Changes

The speed ratio between methods		Levenshtein and Damerau–Levenshtein (%)	Dice and overlap (%)	Dice and matching (%)
W.L.	Input status			
1	Complete words	28.22	20.44	0.34
2	Complete words	−0.01	−12.49	12.54
	Delete one char.	−9.25	28.58	14.29
	Exchange two char's	50	−0.25	70.99
	Error typing in one char.	10	−11.08	0.06
	Insert one char.	40.01	25.23	0.24
3	Complete words	14.29	22.22	11.12
	Delete one char.	91.37	−18.77	−18.78
	Exchange two char's	7.02	0.19	−10.01
	Error typing in one char.	21.57	85.77	28.84
	Insert one char.	23.09	10.92	−0.19
16	Complete words	14.31	89.92	7.5
	Delete one char.	17.81	77.93	36.52
	Exchange two char's	27.62	85.42	−4.91
	Error typing in one char.	14.96	73.89	−4.73
	Insert one char.	11.78	47.92	−8.3

http://dx.doi.org/10.14500/aro.10180

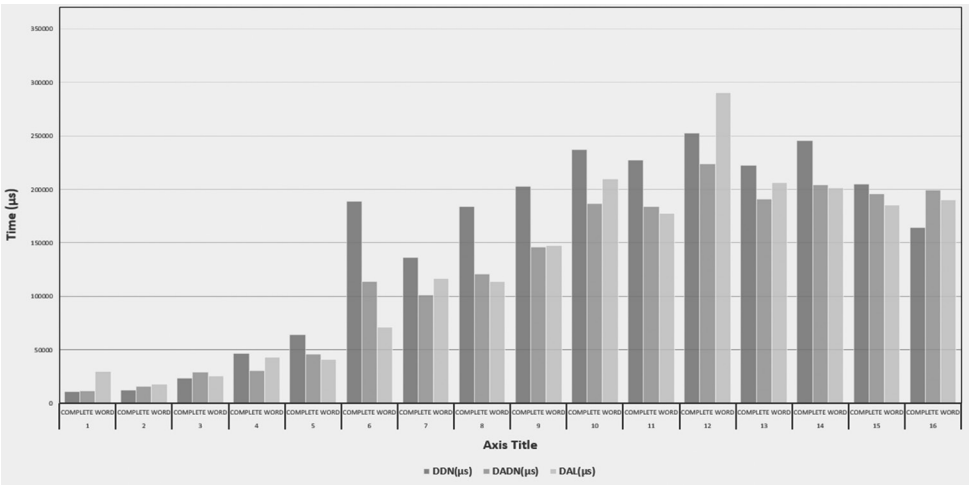

Fig. 1. The new algorithms tested using 10 complete words with different lengths ranging from 1 to 6 characters

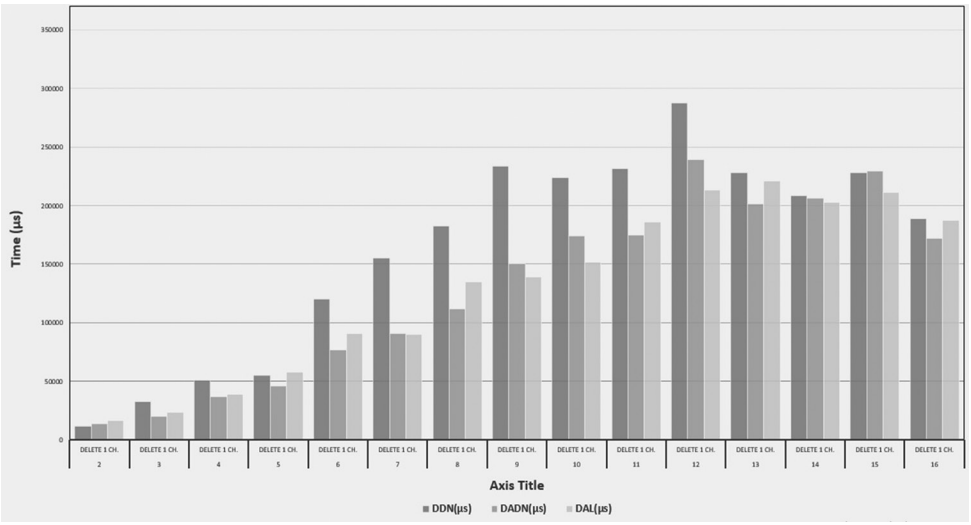

Fig. 2. The new algorithms tested using 10 words with different lengths ranging from 1 to 6 characters with deleting one character

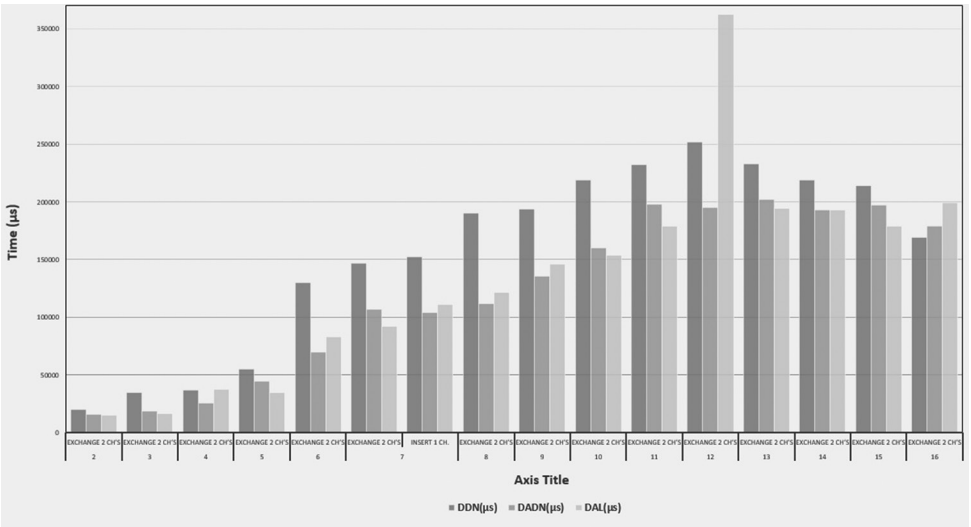

Fig. 3. The new algorithms tested using 10 words with different lengths ranging from 1 to 6 characters with exchange two characters

The test results showed that the proposed implementation of similarity measures reduces the processing time when compared with the commonly implemented methods while maintaining the results accuracy. Furthermore, it can be noticed that the baseline of the execution time is increased dramatically with the increase of words length (i.e., number of characters

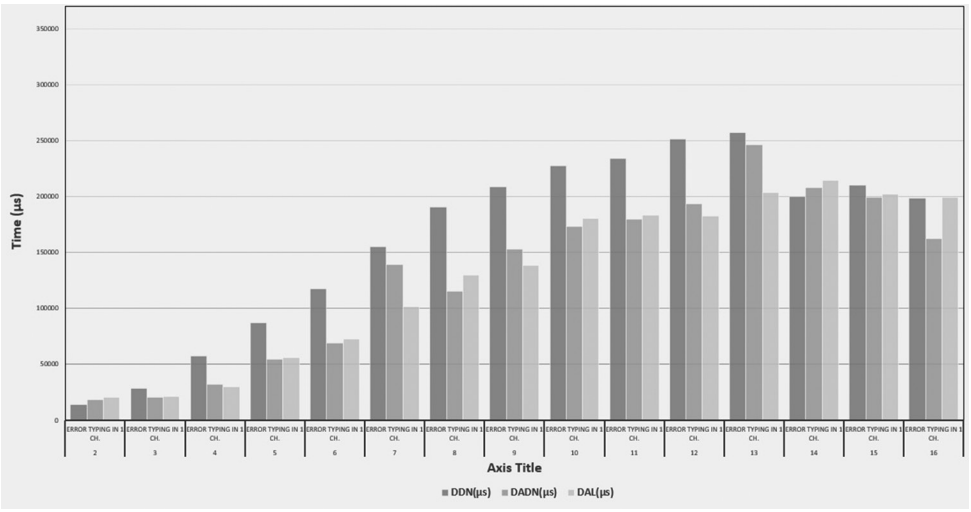

Fig. 4. The new algorithms tested using 10 words with different lengths ranging from 1 to 6 characters with one incorrect character

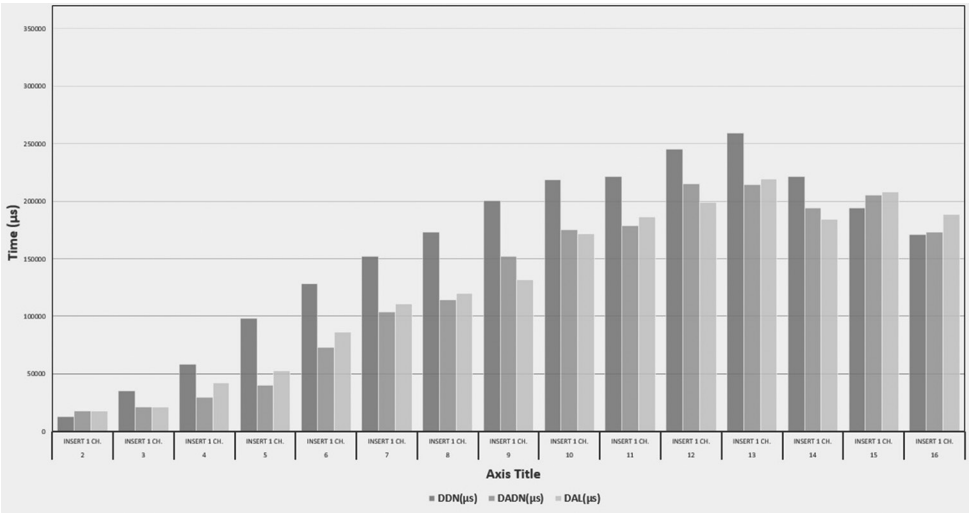

Fig. 5. The new algorithms tested using 10 words with different lengths ranging from 1 to 6 characters with insert one character

in each word) this due to the increase of matching operations that required in each execution. The presented results in Tables VII and VIII indicate that the elapsed time was slightly improved for words have lengths ranging between [1...5 and 14...16]. While the time improvements are relatively large with lengths between [6...13]; it depends on the processing time ratio of DNDA algorithm and DNDAEN algorithm. Hence, to achieve better performance results, the ODNDAEN algorithm is provided; it is used to process words according to some conditions depending on using the suggested enumeration methods. For clarification, the results of processing time improvement were explained using the processing time ratio for the DNDAEN and ODNDAEN algorithms.

IV. Conclusion

A new set of measures was introduced in this article for dealing with strings; it is based on combining some string similarity measures beside to using the string enumeration methodology. It reduces the elapsed time of each string matching operation; this remark can be simply noticed in the listed test results. A set of three combined similarity measures was suggested, and their performance was tested; these combined sets consist of some well-known similarity measures. The test results indicated significant performance improvements are attained when using these combined measures to overcome the lack of accuracy and to save spend time in the matching of misspelled words using the length threshold of words matching.

Furthermore, in this paper, a simple enumeration method was used together with the suggested combined measures to get a new scheme that offers more nearly stable and fast text similarity assessment; this scheme can be used for a wide range of word length. Because these three combined algorithms cause an enhancement in the processing results accuracy by dealing with many different cases that produce similar results (i.e. the same number of movements number of the given compared words). This number was used for measuring the distance between the given words. Hence, to identify the closest word between the given words with similar distances the modified Dice and N-gram algorithm were used.

ARO p-ISSN: 2410-9355, e-ISSN: 2307-549X

http://dx.doi.org/10.14500/aro.10180

Table VII
The Modified DADNEN Tested Using 10 Words with Different Lengths Ranging from 1 to 9 Characters with Multiple Changes

Method name		DADNEN (μs)	Speed ration between DADNEN (μs) and DANA (%)	DADNENs (μs)	Speed ration between DADENs (μs) and DANA (%)	Method name		DADNEN (μs)	Speed ration between DADNEN (μs) and DANA (%)	DADNENs (μs)	Speed ration between ODADNEN (μs) and DANA (%)
W.L.	Input status					W.L	Input Status				
1	Complete words	29038	141.82	12008.9	0.01	6	Complete words	83548.2	−26.77	115081	0.88
2	Complete words	37670.6	135.28	16011	−0.01		Delete one char.	62727.1	−18.6	91064.2	18.19
	Delete one char.	34023.8	140.69	14011.5	−0.89		Exchange two char's	62773.3	−10.39	122087	74.29
	Exchange two char's	41034.3	156.4	11994.7	−25.06		Error typing in one char.	65606.8	−4.99	100073	44.93
	Error typing in one char.	57040.2	216.67	13010.8	−27.77		Insert one char.	66307.9	−9.24	98068.4	34.24
	Insert one char.	43012	138.77	12023.5	−33.26	7	Complete words	130107	28.73	153107	51.48
3	Complete words	49279.6	69.92	18012.8	−37.9		Delete one char.	100552	10.4	139096	52.72
	Delete one char.	44041.2	114.04	17021.2	−17.28		Exchange two char's	102964	−3.84	135096	26.18
	Exchange two char's	42586.5	123.94	17011.6	−10.55		Error typing in one char.	131092	−5.77	149107	7.19
	Error typing in one char.	40029.3	96.78	22030.6	8.3		Insert one char.	120084	15.39	134096	28.85
	Insert one char.	43038.5	104.81	20013.8	−4.77	8	Complete words	124598	2.91	135098	11.58
4	Complete words	46519.1	49.96	36027.5	16.14		Delete one char.	126108	12.7	133094	18.95
	Delete one char.	56056.9	51.4	30021.7	−18.92		Exchange two char's	174124	55.71	146104	30.66
	Exchange two char's	49052.7	88.66	25037.1	−3.71		Error typing in one char.	136784	19	136080	18.39
	Error typing in one char.	55219.2	72.45	30040.3	−6.19		Insert one char.	145103	27.06	138096	20.92
	Insert one char.	46771.4	55.79	27020.3	−10.01	9	Complete words	204146	39.59	164070	12.19
5	Complete words	85060.1	84.79	43032.1	−6.52		Delete one char.	168126	11.66	134081	−10.96
	Delete one char.	55053.8	19.6	45030.8	−2.18		Exchange two char's	175971	29.54	159095	17.12
	Exchange two char's	78039.7	73.3	46038.2	2.24		Error typing in one char.	141642	−7.41	163096	6.63
	Error typing in one char.	55654.6	3	47049.9	−12.94		Insert one char	138238	−9.28	128090	−15.94
	Insert one char.	59425.3	48.55	48020.6	20.04		.				

DADNEN: Dice coefficient, N-gram, and Damerau–Levenshtein using enumeration, ODADNEN: Optimized dice coefficient, N-grame and Damerau–Levenshtein using enumeration, DANA: Dice Ccoefficient, N-grame and Damerau–Levenshtein Measure

Table VIII
The Modified DADNEN Tested Using 10 Words with Different Lengths Ranging from 10 to 16 Characters with Multiple Changes

Method name		DADNEN (μs)	Speed ration between DADNEN (μs) and DANA (%)	DADNENs (μs)	Speed ration between DADENs (μs) and DANA (%)	Method name		DADNEN (μs)	Speed ration between DADNEN (μs) and DANA (%)	DADNENs (μs)	Speed ration between DADENs (μs) and DANA (%)
W.L.	Input status					W.L.	Input Status				
10	Complete words	180127	−3.64	180111	−3.65	14	Complete words	213873	4.77	204127	−8.74
	Delete one char.	177144	1.92	140081	−19.42		Delete one char.	223007	8.18	188132	7.78
	Exchange two char's	199124	24.5	188133	17.63		Exchange two char's	241172	24.87	208148	−0.49
	Error typing in one char.	157921	−8.87	167136	−3.56		Error typing in one char.	227161	9.14	207148	−2.59
	Insert one char.	172124	−1.67	174124	−0.52		Insert one char.	206145	6.2	189115	−11.22
11	Complete words	198179	7.84	160131	−12.88	15	Complete words	200512	2.23	174142	−14.2
	Delete one char.	191103	9.29	161113	−7.87		Delete one char.	206454	−9.91	196620	12.22
	Exchange two char's	177125	−10.44	157128	−20.55		Exchange two char's	199274	1.09	221225	−3.05

(*Contd...*)

Table VIII
(Continued)

Method name		DADNEN (µs)	Speed ration between DADNEN (µs) and DANA (%)	DADNENs (µs)	Speed ration between DADENEs (µs) and DANA (%)	Method name		DADNEN (µs)	Speed ration between DADNEN (µs) and DANA (%)	DADNENs (µs)	Speed ration between DADENEs (µs) and DANA (%)
W.L.	Input status					W.L.	Input Status				
12	Error typing in one char.	170121	−5.33	186147	3.6	16	Error typing in one char.	230162	15.55	193136	−16.11
	Insert one char.	186132	4.2	181111	1.39		Insert one char.	212151	3.41	172122	-18.1
	Complete words	184130	−17.72	191120	−14.6		Complete words	209149	5.03	163115	−1.17
	Delete one char.	218155	−8.92	180127	−24.8		Delete one char.	194139	12.8	170122	2.81
	Exchange two char's	242169	23.97	200140	2.46		Exchange two char's	199141	11.18	184148	9.87
	Error typing in one char.	233164	20.6	209148	8.18		Error typing in one char.	188134	16.04	178120	4.45
	Insert one char.	222781	3.55	198143	−7.91		Insert one char.	188860	8.9	181143	−8.74
13	Complete words	225161	17.8	208152	8.9						
	Delete one char.	192155	−4.47	202140	0.5						
	Exchange two char's	216155	6.94	213150	5.45						
	Error typing in one char.	203981	−17.14	197139	−19.92						
	Insert one char.	234167	9.34	183128	−14.5						

DADNEN: Dice coefficient, N-gram and Damerau–Levenshtein using enumeration, DANA: Dice Ccoefficient, N-grame and Damerau–Levenshtein Measure

References

Alberto, B., Paolo, R., Eneko, A. and Gorka, L., 2010. Plagiarism detection across distant language Pairs. In: *Proceedings of the 23rd International Conference on Computational Linguistics*. pp.37-45.

Bunke, H. and Bühler, U., 1992. Invariant shape recognition using string matching. In: *Proceedings of 2nd International Conference on Automation*. Robotics and Computer Vision, Singapore.

Burnard, L., 1976. *The University of Oxford Text Archive University of Oxford*. Available from: http://www.ota.ox.ac.uk/catalogue/index.html.

Cortelazzo, G., Deretta, G., Mian, G.A. and Zamperoni, P., 1996. Normalized weighted Levensthein distance and triangle inequality in the context of similarity discrimination of bilevel images. Pattern Recognition Letters, 17(5), pp.431-436.

Cortelazzo, G., Mian, G.A., Vezzi, G. and Zamperoni, P., 1994. Trademark shapes description by string-matching techniques. Pattern Recognition, 27(8), pp.1005-1018.

Damerau, F.J., 1964. A technique for computer detection and correction of spelling errors. Communications of the ACM, 7(3), pp.171-176.

Dice, L.R., 1945. Measures of the amount of ecologic association between species. Ecology, 26(3), pp.297-302.

Fenz, D., Lange, D., Rheinländer, A., Naumann, F. and Leser, U., 2012. Efficient similarity search in very large string sets. In: *International Conference on Scientific and Statistical Database Management*. Springer Berlin, Heidelberg, pp262-279.

Gomaa, W.H. and Fahmy, A.A., 2013. A survey of text similarity approaches. International Journal of Computer Applications, 68(13), 13-18.

Hall, P.A. and Dowling, G.R., 1980. Approximate string matching. ACM Computing Surveys (CSUR), 12(4), pp.381-402.

Kashiap, R.L. and Oommen, B.J., 1984. String correction using probabilistic models. Pattern Recognition Letters, 2, pp.147-154.

Leusch, G., Ueffing, N. and Ney, H., 2003. A novel string-to-string distance measure with applications to machine translation evaluation. In: *Proceedings of MT Summit IX*, pp.240-247.

Martins, B., 2011. A supervised machine learning approach for duplicate detection over gazetteer records. In: *Proceedings of the 4th International Conference on Geospatial Semantics*. Springer, Berlin Heidelberg, pp.34-51.

Marzal, A. and Vidal, E., 1993. Computation of normalized edit distance and applications. IEEE Transactions on Pattern Analysis and Machine Intelligence, 15(9), pp.926-932.

Mohri, M., 2003. Edit-distance of weighted automata: General definitions and algorithms. International Journal of Foundations of Computer Science, 14(6), pp.957-982.

Oommen, B.J., 1987. Recognition of noisy subsequences using constrained edit distances. IEEE Transactions on Pattern Analysis and Machine Intelligence,5, pp.676-685.

Pande, B.P., Pawan, T. and Dhami, H.S., 2013. Generation, implementation and appraisal of an N-gram based stemming algorithm. ArXivpreprint arXiv 1312-4824. Available from: https://arxiv.org/ftp/arxiv/papers/1312/1312.4824.pdf.

Patel, D., 2016. Study of distance measurement techniques in context to prediction model of web caching and web prefetching. International Journal on Soft Computing, Artificial Intelligence and Applications (IJSCAI), 5(1), 1-8.

Peng, H.L. and Chen, S.Y., 1997. Trademark shape recognition using closed contours. Pattern Recognition Letters, 18(8), pp.791-803.

Pradhan, N., Gyanchandan, M. and Wadhvani, R., 2015. A review on text similarity technique used in IR and its application. International Journal of Computer Applications, 120(9), pp.29-34.

Rieck, K. and Wressnegger, C., 2016. Harry: A tool for measuring string similarity. Journal of Machine Learning Research, 17(9), pp.1-5.

Sehgal, V., Getoor, L. and Viechnicki, P.D., 2006. Entity resolution in geospatial data integration. In: *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems.* ACM, New York, NY, pp.83-90.

Sellers, P.H., 1980. The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms, 1(4), pp.359-373.

Wang, J., Li, G. and Feng, J., 2014. Extending string similarity join to tolerant fuzzy token matching. ACM Transaction Database Systems, 39(1), pp.1-45.

Winkler, W.E., 1999. The state of record linkage and current research problems. In: *Statistical Research Division, US Census Bureau*. Available from: http://www.census.gov/srd/www/byname.html.