Available online at www.interscience.in

Efficient Algorithm for Auto Correction Using n-gram Indexing

Mahesh Lalwani, Nitesh Bagmar & Saurin Parikh
Nirma University, Ahmedabad, India
E-mail : mahesh21688@gmail.com, nitesh89bagmar@gmail.com, Saurin.parikh@nirmauni.ac.in

***Abstract* -** Auto correction functionality is very popular in search portals. Its principal purpose is to correct common spelling or typing errors, saving time for the user. However, when there are millions of strings in a dictionary, it takes considerable amount of time to find the nearest matching string. Various approaches have been proposed for efficiently implementing auto correction functionality. All of these approaches focus on using suitable data structure and few heuristics to solve the problems. Here, we propose a new idea which eliminates the need for calculating edit distance with each string in the dictionary. It uses the concept of Ngram based indexing and hashing to filter out irrelevant strings from dictionary. Experiments suggest that proposed algorithm provides both efficient and accurate results.

***Keywords* -** *Edit distance; Ngram; trigram; String searching; Pattern matching.*

I. INTRODUCTION

Nowadays Auto correction feature is used at many places which automatically corrects the string entered by user. For example in Google search engine if user has entered a wrong string, it automatically corrects the string and shows the result of corrected string with the message of showing result of corrected string instead of showing result of string entered by user.

This auto correction functionality is usually implemented by finding the string that is most similar to the given search string. The difference between two strings or minimum operation required to transform one string to another string is called edit distance between two strings. So edit distance is calculated between each string in database and given search string, the string having minimum edit distance cost is selected for the result.

In many applications auto correction functionality is being used. However calculating edit distance for each pair of stings will require lot of time in case of having large dictionary of words. It becomes more inefficient and expensive in case of the application which is deployed on cloud like Google because in google cloud machine level API is not allowed hence all of the functionality is done through higher level API only. So calculating edit distance for each string in dictionary will take lot of time which is inefficient for use as it should be done in real time. Secondly as most of cloud computing service provider uses pay per user policy, so calculating edit distance for each string in dictionary will take lot of processing time as well which results in greater expenses because auto correction functionality is frequently used by the user.

Related to this problem jong yong kim and john shawe-taylor[1] had proposed an algorithm for DNA sequence. Another method proposed by Klaus U. Schulz and Stoyan Mihov uses finite state automata with Levenshtein Distance algorithm.[2] Victoria J. Hodge and Jim Austin has proposed an hybrid methodology integrating Hamming distance and n-gram algorithms particularly for spell checking user queries in a search engine.[3] But hamming distance algorithm requires equal length strings to calculate the edit distance algorithms. Zhao, Zuo-Peng, Yin, Zhi-Min, Wang, Qian-Pin, Xu, Xin-Zheng, Jiang, Hai-Feng and Jisuanji Yingyong suggest a method to improve the existing Levenshtein Distance algorithm. [4]

But instead of improving existing Levenshtein Distance algorithm we tried to solve the problem with different approach. We look for the solution such that we calculate edit distance for certain strings only instead of all the strings in dictionary. In this paper we proposed an algorithm which provides such solution and filters the string before calculating its edit distance. The new proposed algorithm provides most striking performance with reduced execution time.

II. EDIT DISTANCE

Edit distance between two strings is the minimum number of *edit operations* required to transform one string into another. The edit operations can be insert, delete, replace and transpose which depend on algorithm of edit distance used. There are various algorithms available to find edit distance between two strings. Algorithms related to the edit distance may be used in spelling correctors. If a text contains a word, W, that is not in the dictionary, a `close' word, i.e. one with a small edit distance to W, may be suggested as a correction. Below given are few algorithms used to find edit distance between two strings.

Hamming Distance
Levenshtein Distance
Damerau Kevenshtein Distance
Jaro-Winkler Distance
Ukkonen’s algorithm

From these available algorithms we start working on Levenshtein Distance algorithm.

Levenshtein Distance (LD) is a measure of the similarity between two strings. 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 the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965.

III. PROBLEM IN LEVENSHTEIN DISTANCE ALGORITHM

The running time-complexity of the algorithm is O(|S1|*|S2|), i.e. $O(N^2)$ if the lengths of both strings is about ‘N’. The space-complexity is also $O(N^2)$

With such complexity we can not develop an application with auto correction functionality having large dictionary of strings. Because Levenshtein Distance algorithm having time complexity of $O(N^2)$ will be executed for each string in the dictionary. From all these strings the string having minimum edit distance will be a resultant string. So for M number of strings the algorithm will be executed M times result in time complexity of $O(M*N^2)$ that means for 1 million or 1 billion strings the algorithm will be executed 1 million or 1 billion times respectively. As the value of M increases the solution becomes more and more impractical.

IV. N-GRAM INDEXING

N-gram is a subsequence of n items from a given sequence. The items in question can be phonemes, syllables, letters, words or base pairs according to the application. An n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram"; and size 4 or more is simply called an "n-gram". An n-gram model is a type of probabilistic model for predicting the next item in such a sequence. Ngram models are used in various areas of statistical natural language processing and genetic sequence analysis.

N-gram Index stores sequences of length of data to support other types of retrieval or text mining.

V. PROPOSED ALGORITHM

As above mentioned problem of using Levenshtein Distance algorithm that if we have large dictionary of strings and finding minimum edit distance from all the string in dictionary will take lot of time.

Levenshtein Distance has running time complexity $O(N^2)$ and executing Levenshtein Distance algorithm for each string in the dictionary will result in $O(M* N^2)$ and that will consume lot of time. So we look for the approach such that we can execute Levenshtein Distance algorithm for certain strings in the dictionary only, instead of executing it for each string in dictionary such that we can reduce the execution time.

We then prepared an algorithm which allows executing Levenshtein Distance algorithm for the strings which is most related to the search string instead of executing Levenshtein Distance algorithm for each string in dictionary.

The algorithm prepared by us is shown below in Table 1

```
Step 1:  [Creating Ascii list for search string]
         searchAscii=createList(searchTerm)
         j=0
Step 2:  [Repeat through step 4 for each string in
         database]
         for i=0 to number of string in database
Step 3:  [Get the length difference between string in
         database and search string]
         Diff=abs(len(lstAscii[i])-len(searchAscii))
Step 4:  [if length difference less then 4 then only
         process further for that string]
         If diff<threshold_min
         merge[i]=mergeList(lstAscii[i],searchAscii)
          if len(merge[i])< len(searchAscii) +
                                   threshold_max
                 stringToPass[j]=i
                 j++
            end if
          endif
          [End of  loop]
Step 5:  [Initialize min]
         min=0
Step 6:  [Repeat through step 8]
         for i=0 to len(stringToPass)
Step 7: [Call the Levenshtein Distance algorithm]
         cost[i]=
         LevenshteinDistance(words[stringToPass[i]],se
         archTerm)
```

```
Step 8:  [Check for minimum cost]
         if cost[min]>cost[i]
                 min=stringToPass[i]
         end if
         [End Loop]
```

A. Algorithm

We have prepared an algorithm shown in Table 1. It uses the concept of Ngram indexing. The main idea behind the algorithm is to divide each string in the dictionary in the form of trigrams and also to divide the search string in the form of trigrams. Then it will find out the unmatched trigrams between strings in dictionary and search string.

Depends on the number of unmatched trigrams the selection of string for calculating Levenshtein distance is done. Note that algorithm does not compare two trigrams of string as it will again result in time complexity of O(N^2). Instead for each trigram of string sum of Asciii value of each character is calculated. Then comparison is done on sum of Ascii values of characters as it will have time complexity O(N).

The reason behind using trigrams instead of unigram and bigram is in case of large string it will create many subsequences which also increase the processing time. And if we create subsequence of length more than three then it will decrease the accuracy of resultant string. Using trigram the results obtained are both efficient in terms of processing time and accurate with the desired result.

B. Algorithm Description

- Words[m][n] is variable contains list of strings
- searchTerm is search string that user has entered
  - lstAscii[m][n-2] is ordered list contains Ascii value of trigram for each string in database
  - searchAscii[] is ordered list contains Ascii value of trigram for search string
- merge[m][] is order list after merging two list
  - stringToPass[] contains string that will be passed to Levenshtein Distance algorithm

The algorithm shown in Table1 uses database of strings and ordered list having sum of ASCII value of each trigram of each string in database as creating trigram list for each string in database is one time cost and it is then stored into database. Algorithm also has string as search string entered by user. Algorithm calculates the sum of ASCII value of trigrams of search string and stores that in ordered list called searchAscii.

Then it will compare the length difference of ordered list of search string and ordered list of each string in database. If difference is less then the threshold_min then only for those strings it will perform merge operation. In case study we used value of threshold_min is four because there is least chance of the string having length difference more then three to become resultant string.

After merging two lists again it will check the length difference between merge list and searchAscii list and for the strings having difference less then threshold_max will pass to Levenshtein Distance algorithm. We have used threshold_max value nine in case study because for the mistake of single character will affect three trigram. So if we allow three character mistakes then at max it will affects nine trigrams

C. Complexity of Algorithm

The algorithm we proposed is useful for the application with Auto Correction feature and having large database of strings. As mentioned above that time and space complexity of Levenshtein Distance algorithm is O(N^2) which is not at all feasible for such application.

So to solve this problem we are proposing algorithm having running time complexity lesser then O(N^2) because in proposed algorithm Levenshtein Distance algorithm is executed for certain number of strings in database only.

Running Time Complexity of proposed algorithm is as follows.

The function createList() will be executed only once for search string and having complexity of O(N).

Then the outer loop will be an executed m time which is nothing but number of stings in database, hence time to execute this loop may vary depends on value of m.

The statement in step-3 to get difference between two lists will have complexity O(N) with respect to value of M, where M is number of strings in database.

Inside step-4 the if statement will take same execution time always, so if statement have complexity O(1) which is ignorable. Inside the if statement first statement merges the two list which have complexity of O(N). The second statement is again if statement hence, it has also complexity O(1) and inside this if statement again there is two statement having complexity of O(1) each which is ignorable.

Step-3 and step-4 will be executed M times where M is number of strings in database. So complexity of step-2 to step-4 is (O(M*N) + O(M*N))

The step-7 will execute Levenshtein Distance algorithm for certain K strings only having complexity of O(N^2).

Step-5 and step-8 will be executed K times where K is number of strings in database. So complexity of step-5 to step-8 is O(K*N^2)

So Total Running time complexity is:

O(N)+O(M*N)+O(M*N)+O(K*N2)

O(2*M*N)+ O(K*N2)

O(M*N)+O(K*N2)

Where K << M

D. Case study

We have tested this algorithm for the application mentioned earlier. We have used this algorithm for the application deployed on Google cloud also we compared it with the application using only Levenshtein Distance algorithm.

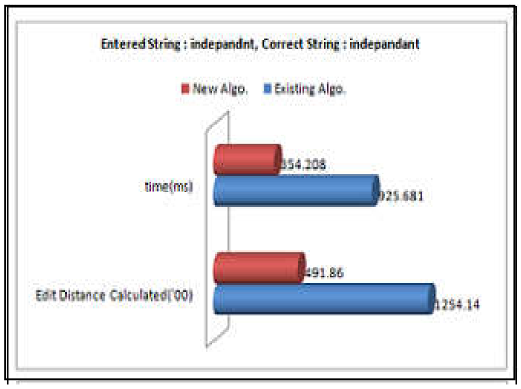
The result of case study is shown in fig-1. We get most striking performance in results. In the case study we measured the time taken by proposed algorithm and also the time take by using only Levenshtein Distance algorithm.

As shown in fig-1 the new proposed algorithm reduced the running time by almost 3 times comparing to using only using existing Levenshtein Distance algorithm.

In fig-1 comparison is shown for three strings and for each comparison it is shown that for how many strings edit distance is calculated and time taken by each.

Fig-1 also shows that the string entered by user and the string that is corrected by algorithm.

In existing algorithm the Levenshtein Distance algorithm was executed 125414 times while in new proposed algorithm Levenshtein Distance algorithm were executed 48512,49186 and 51232 times only.



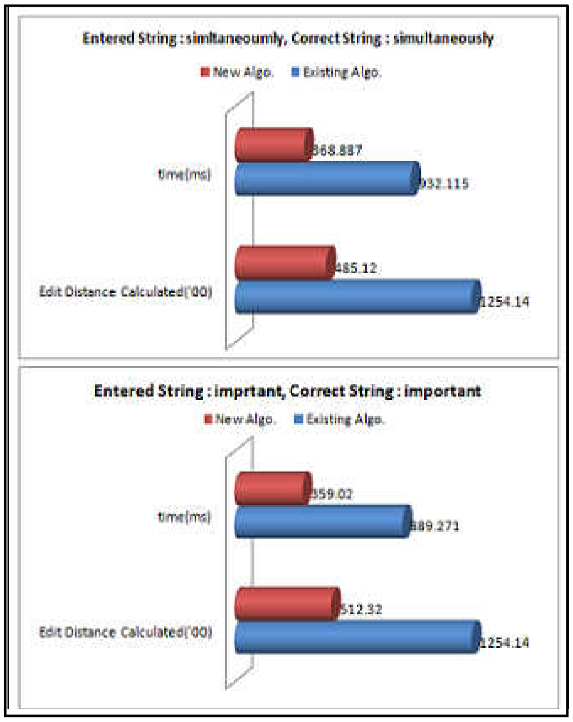


Figure 1: PERFORMANCE COMPARISON

VI. CONCLUSION AND FUTURE WORK

Auto Correction is one of the features which will be used in most of the today's web applications. Auto Correction feature can be implemented using existing Levenshtein Distance algorithm as well but it will lake into performance because the running time complexity becomes O(M*N^2) where M is the number of strings in database.

So proposed algorithm which uses n-gram indexing will improve the performance of the web application as it has running time complexity O(M*N)+O(K*N2) Where K<<M.

So we conclude that the proposed algorithm is more suitable for the web applications to implement the Auto Correction feature.

We also plan to explore randomized algorithms and approximation algorithms in future to make it more efficient.

REFERENCES

[1] Jong Yong Kim and John Shawe-Taylor. ``Fast string matching using an n-gram algorithm." Software-Practice and Experience 24(1):79-88. January 1994.

[2] Klaus U. Schulz and Stoyan Mihov “Fast string correction with Levenshtein automata” INTERNATIONAL JOURNAL ON DOCUMENT ANALYSIS AND RECOGNITION, Volume 5, Number 1, 67-85, DOI: 10.1007/s10032-002-0082-8,2003

[3] Ref: V. Hodge and J. Austin, “A Comparison of a Novel Spell Checker and Standard Spell Checking Algorithms Pattern Recognition”, vol. 15 no. 5,pp. 1073-1081, 2003.

[4] Zhao, Zuo-Peng, Yin, Zhi-Min, Wang, Qian-Pin, Xu, Xin-Zheng, Jiang, Hai-Feng and Jisuanji Yingyong “An improved algorithm of Levenshtein Distance and its application in data processing” Journal of Computer Applications. Vol. 29, no. 2, pp. 424-426. Feb. 2009 independant.

❑❑❑