Project Report

On

Analysis & Modification of Levenshtein Algorithm and Understanding the Linguistic Differences

Submitted By

1. **Shivam Tayal** - 150001034
2. **Utkarsh Kumar Singh** - 150001037

Computer Science and Engineering

2nd year

Under the Guidance Of

Dr. Kapil Ahuja

Department Of Computer Science and Engineering

Indian Institute Of Technology, Indore

Spring 2017

(1.) Analysis and Modifications of Levenshtein Algorithm

Introduction

Levenshtein Algorithm is used to find the similarity between two strings. It is also referred to as *Edit Distance Algorithm.* More specifically, Levenshtein Distance tells about the total number of edits (insertions, deletions and substitutions) needed to make in one string to convert it to another.
For example, the Edit Distance between “monkey” and “donkey” is 1 (‘m’ will be substituted by ‘n’ or vice versa).

Objective

We have a real life client – SUMMENTOR PRO which is a startup that acts as a channel between builders and their clients. For its website, the client wanted us to design a search portal. So our project would revolve around designing an efficient searching algorithm based upon ***Levenshtein distance.***
Apart from that, we wish to propose changes to the existing algorithm. The current algorithm doesn’t keep into account the position of the characters where changes are required. However, we have seen that more often, the initial positions are more relevant and hence, should be given more priority. We did the same for our version of Levenshtein Algorithm.

Pseudo Code

This is a dynamic programming methodology.

Suppose we have two string **a** and **b** of lengths **n** and m respectively. Let **d[i,j]** represents minimum no. of edits (substitution, deletion, addition) required to convert **a**(upto ith character) into **b**(upto jth character). Then,

```
If n = 0,  d[0,j] = j        for all j<=m
If m = 0, d[i,0] = i       for all i<=n
Else,    d[i,j] = min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+(a[i]==b[i]?0:1))
The result is d[n,m].
```

This algorithm works in O(n*m) time. Further, we incorporate weighted edit distance and our own modification.

Algorithm Design & Implementation

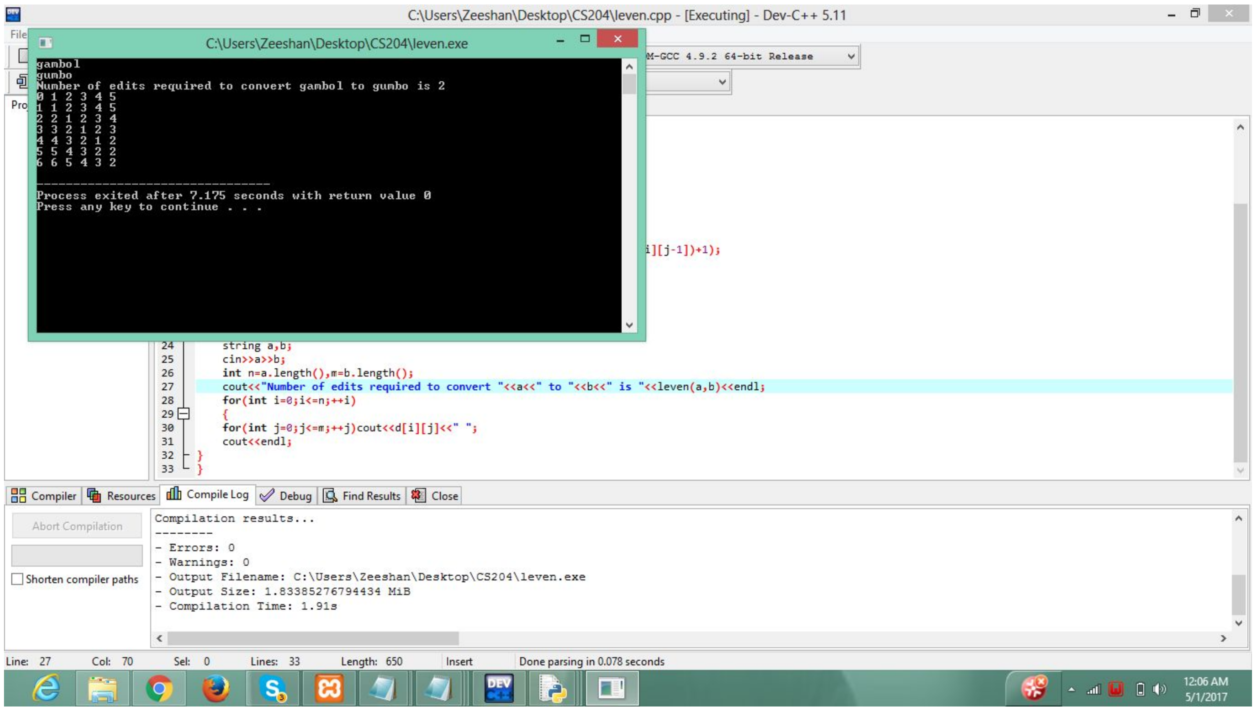
- Implementation of Levenshtein Distance -

```
 #include<bits/stdc++.h>
using namespace std;
int d[100][100]={0};
int leven(string a,string b)
{
        int n=a.length();
        int m=b.length();
       d[0][0]=0;
        for(int i=1;i<=n;++i)d[i][0]=i;
        for(int j=1;j<=m;++j)d[0][j]=j;
        for(int i=1;i<=n;++i)
        {
                for(int j=1;j<=m;++j)
                {

                        d[i][j]=min(d[i-1][j-1]+(a[i]==b[j]?0:1), min(d[i-1][j],d[i][j-1])+1);
                }
        }
  return d[n][m];
}
int main()
{
        string a,b;
        cin>>a>>b;
        int n=a.length(),m=b.length();
        cout<<"Number of edits to convert "<<a<<" to "<<b<<" is "<<leven(a,b)<<endl;
        for(int i=0;i<=n;++i)
        {
        for(int j=0;j<=m;++j)cout<<d[i][j]<<" ";
```

```
            cout<<endl;
    }
    }
```

This is the code to calculate Levenshtein Distance between two words.

E.g Shown below is the result for the words for “gambol” and “gumbo”.



C:\Users\Zeeshan\Desktop\CS204\leven.cpp - [Executing] - Dev-C++ 5.11
C:\Users\Zeeshan\Desktop\CS204\leven.exe
M-GCC 4.9.2 64-bit Release
Compiler
Resources
Compile Log
Debug
Find Results
Close
Abort Compilation
Shorten compiler paths
Line: 27 Col: 70 Sel: 0 Lines: 33 Length: 650 Insert Done parsing in 0.078 seconds
12:06 AM
5/1/2017

```
gambol
gumbo
Number of edits required to convert gambol to gumbo is 2
0 1 2 3 4 5
1 1 2 3 4 5
2 2 1 2 3 4
3 3 2 1 2 3
4 4 3 2 1 2
5 5 4 3 2 2
6 6 5 4 3 2

--------------------------------
Process exited after 7.175 seconds with return value 0
Press any key to continue . . .
```

```
i][j-1])+1);
    string a,b;
    cin>>a>>b;
    int n=a.length(),m=b.length();
    cout<<"Number of edits required to convert "<<a<<" to "<<b<<" is "<<leven(a,b)<<endl;
    for(int i=0;i<=n;++i)
    {
    for(int j=0;j<=m;++j)cout<<d[i][j]<<" ";
    cout<<endl;
}
}
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Zeeshan\Desktop\CS204\leven.exe
- Output Size: 1.83385276794434 MiB
- Compilation Time: 1.91s
```

- Backtracking to find the edited characters and modify the results accordingly :

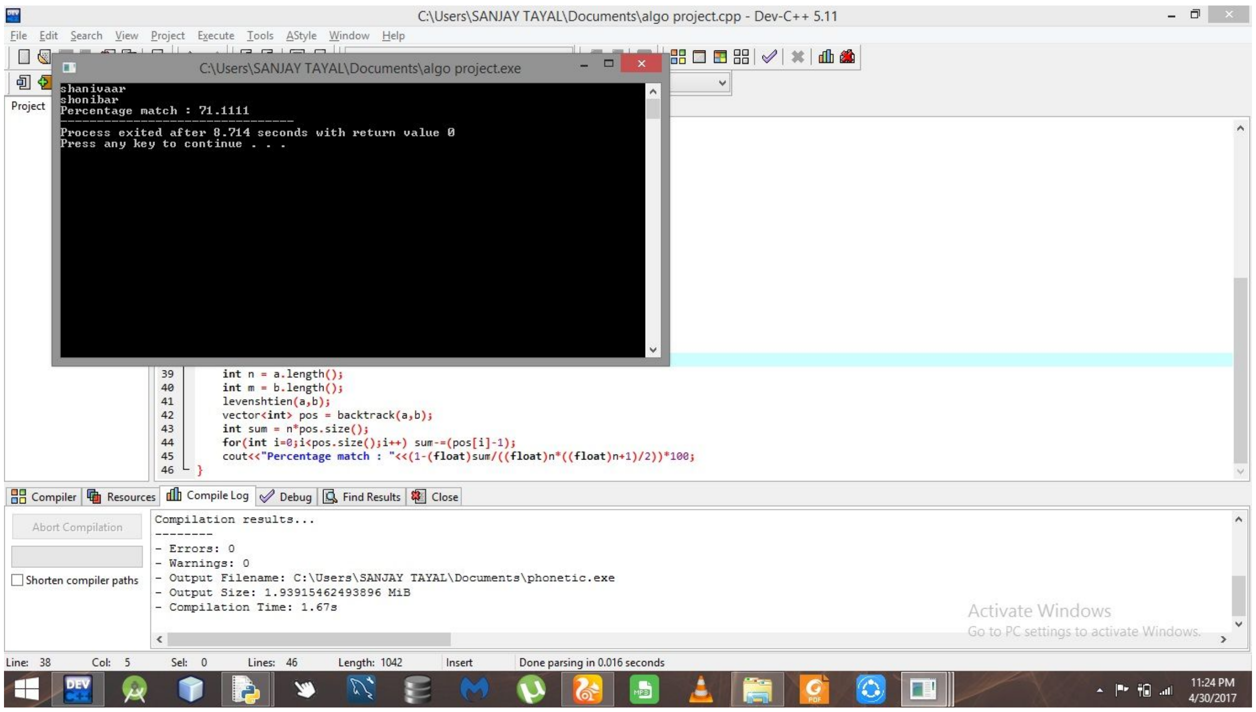
  We backtrack to find the characters we have to make edits. For example, for the Words “gambol” and “gumbo”, the edits are required at the second and sixth positions. In the final result, we are giving priority to the initial positions (for Instance, first letter has more priority than the second letter and so on). We Incorporate this idea in our algorithm to have more precise output.

The equivalent C++ implementation is as follows :-

```
#include <bits/stdc++.h>
using namespace std;
int d[100][100];
void levenshtien(string a, string b){
      int n = a.length(),m = b.length();
      for(int i=1;i<=n;i++) d[i][0] = i;
      for(int j=1;j<=m;j++) d[0][j] = j;
      for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                  d[i][j] =
min(d[i-1][j-1]+((a[i-1]==b[j-1])?0:1),min(d[i-1][j],d[i][j-1])+1);
            }
      }
}

vector<int> backtrack(string a,string b){
      vector<int>pos;
      int n = a.length(),m = b.length();
      for(int i=n;i>0;){
            for(int j=m;j>0;){
                  if(d[i-1][j]+1==d[i][j]){
                        pos.push_back(i);
                        i--;
                  }
                  else if(d[i-1][j-1]+((a[i-1]==b[j-1])?0:1)==d[i][j]){
                        if(a[i-1]!=b[j-1])pos.push_back(i);
                        i--; j--;
                  }
                  else{
                        j--;
                  }
            }
      }
      return pos;
}

int main(){
      string a,b;
      cin>>a>>b;
      int n = a.length();
      int m = b.length();
      levenshtien(a,b);
      vector<int> pos = backtrack(a,b);
      int sum = n*pos.size();
```

```
        for(int i=0;i<pos.size();i++) sum-=(pos[i]-1);
        cout<<"Percentage match :
"<<(1-(float)sum/((float)n*((float)n+1)/2))*100;
}
```

Here is a sample output-


C:\Users\SANJAY TAYAL\Documents\algo project.cpp - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
Project
C:\Users\SANJAY TAYAL\Documents\algo project.exe
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
Shorten compiler paths
Activate Windows
Go to PC settings to activate Windows.
Line: 38 Col: 5 Sel: 0 Lines: 46 Length: 1042 Insert Done parsing in 0.016 seconds
11:24 PM
4/30/2017

```
shanivaar
shonibar
Percentage match : 71.1111
--------------------------------
Process exited after 8.714 seconds with return value 0
Press any key to continue . . .
```

```
    int n = a.length();
    int m = b.length();
    levenshtien(a,b);
    vector<int> pos = backtrack(a,b);
    int sum = n*pos.size();
    for(int i=0;i<pos.size();i++) sum-=(pos[i]-1);
    cout<<"Percentage match : "<<(1-(float)sum/((float)n*((float)n+1)/2))*100;
}
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\SANJAY TAYAL\Documents\phonetic.exe
- Output Size: 1.93915462493896 MiB
- Compilation Time: 1.67s
```

For the words “Shanivaar” and “Shonibar”, it’s showing a percentage similarity of 71.11 %.

Algorithm Analysis

The modified algorithm runs in **O(nm)** where **n** and **m** are the lengths of string **a** and **b** respectively because backtracking takes the same runtime as levenshtein algorithm.

Conclusion

Hence, we modified the Levenshtein Algorithm to achieve more accurate results by giving priority to the positions as well. We will incorporate this algorithm for the search portal of our website. As a part of the future work, we wish to extend it to “weighted edit distance” that takes into account the statistical data as well.

(2) Understanding Linguistic differences using Levenshtein Algorithm

Introduction
Different languages have often evolved from a common root. Although, over the course of time, languages have undergone massive change, similarities can still be traced in their classic vocabulary . In this paper, we wish to exploit these similarities using the Levenshtein Algorithm. For this project, we are using the days of the week and numerals (1-10) as our sample space to trace the similarity. We are taking Hindi to be the base language and checking the similarities for Marathi, Bengali, Punjabi, Bhojpuri, Marwari and Bhojpuri.

Procedure

Since we are assuming Hindi to be the base language, we are taking edit distances of the words in various languages as compared to their Hindi equivalent. And using that, we are calculating the fractional / percentage similarity of that language with Hindi.

Inference

(a) Numerals (1-10) :

Hindi	**Marathi**	**Bengali**	**Punjabi**	**Marwadi**	**Bhojpuri**
Ek	Ek (0)	Ek (0)	Ikk (2)	Ek (0)	Ek (0)
Do	Don (1)	Dui (2)	Do (0)	Doy (1)	Dui (2)
Teen	Tin (2)	Tin (2)	Tinn (2)	Tin (2)	Tin (2)
Chaar	Char (1)	Char (1)	Cha (2)	Chyar (1)	Cari (3)
Paanch	Pach (2)	Pach (2)	Punj (4)	Pach (2)	Pac (3)
Cheh	Saha (3)	Chchoy (3)	Che (1)	Chaw(2)	Chae (2)
Saat	Sat (1)	Shat (1)	Satt (1)	Sat (1)	Sat (1)
Aath	Ath (1)	At (2)	Ath (1)	At (2)	Ath (1)
Nao	Nau (1)	Noy (2)	Naum (2)	Naw (1)	Nao (0)
Das	Daha (2)	Dosh (2)	Dass (1)	Das (0)	Das (0)

*Numbers in the brackets represent the edit distance w.r.t the Hindi equivalent

Hence, on the basis of the numerals, following is the percentage similarity of various languages with Hindi :-

- Marathi - **62.16 %**
- Bengali - **59.4 %**
- Punjabi- **56.7 %**
- Marwari- **67.6 %**
- Bhojpuri - **62.16 %**

(b) Days of the week :

Hindi	**Marathi**	**Bengali**	**Punjabi**	**Bhojpuri**
Somvaar	Somavar (2)	Shombar (3)	Somvaar (0)	Somaar (1)
Mangalvaar	Mangalavaar (2)	Mongolbar (4)	Mangalvaar (0)	Mangar (4)
Budhvaar	Budhavar (2)	Budhbar (2)	Budhvaar (0)	Budh (4)
Shukravaar	Sukravar (1)	Shukrobar (3)	Shukarvaar (2)	Shukkar (4)
Shanivaar	Shanivar (1)	Shonibar (3)	Shanivaar (0)	Sanichar (3)
Ravivaar	Ravivar (1)	Robibar (4)	Aitvaar (3)	Etvaar (4)

*Numbers in brackets indicate the values of Levenshtein Distance corresponding to Hindi equivalent

Hence, on the basis of the days of week, following is the percentage similarity of various languages with Hindi :-

- Marathi - **82.4 %**
- Bengali- **62.7 %**
- Punjabi - **90.2 %**
- Bhojpuri - **60.8%**

Combining the two results, overall similarity are as follows -

(1) Marathi - 73. 9 %
(2) Bengali- 61. 4 %
(3) Punjabi- 76.13 %

(4) **Bhojpuri - 61.4 %**

Limitations

- This approach is not taking care of the accents or phonetic similarity.
- In some cases, synonyms are causing non-linearity. E.g. There are two words for 'Thursday' in Hindi - ' Brihashpativaar' and ' Guruvaar'. In Marathi, the equivalent word is 'Guruvaar' but in Bengali, it's ' Brihoshpotibaar'. However, in Punjabi, it's 'Veervaar'.
- The results would be more accurate if we have a large sample space.

Advanced Analysis and Modifications

- Phonetic Correction :- In order to fix the phonetic misspellings in proper nouns, we can use the ' Phonetic Hash ' technique so that similar sounding terms correspond to the same value. Using this, we can create a soundex map. Following are the basic principles -

(1) We turn every term to be indexed into a 4-character reduced form. And then we build an inverted index from these reduced forms to the original terms; and call this the soundex index.

(2) We do the same with both the terms.

(3) We compare the terms on the basis of their soundex index.

Algorithm :

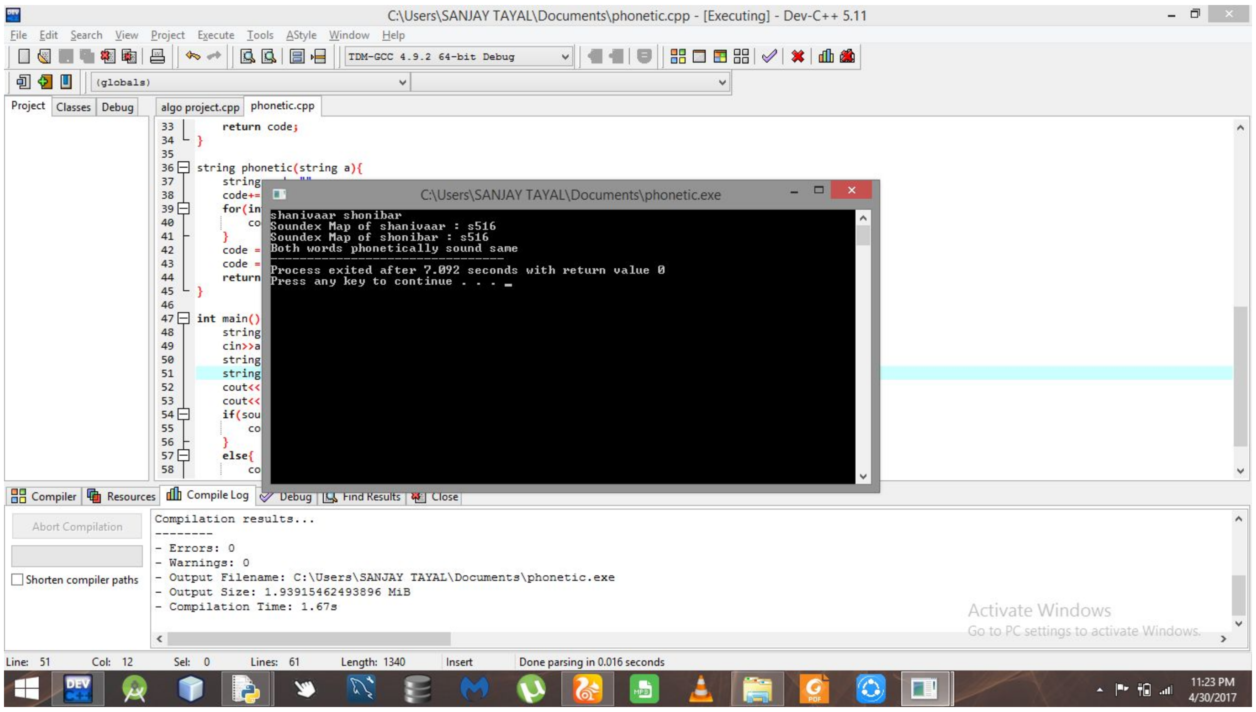
1. Retain the first letter of the term.
2. Change all occurrences of the following letters to '0' (zero): 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows: B, F, P, V to 1. C, G, J, K, Q, S, X, Z to 2. D,T to 3. L to 4. M, N to 5. R to 6.
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all zeros from the resulting string. Pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

This is the C++ code to generate the soundex map of a string :-

```
#include <bits/stdc++.h>
using namespace std;

int Map(char c){
      if(c=='a'||c=='e'||c=='i'||c=='o'||c=='u'||c=='h'||c=='w'||c=='y')
return '0';
      if(c=='b'||c=='f'||c=='p'||c=='v') return '1';
      if(c=='c'||c=='g'||c=='j'||c=='q'||c=='s'||c=='x'||c=='z') return
'2';
      if(c=='d'||c=='t') return '3';
      if(c=='l') return '4';
      if(c=='m'||c=='n') return '5';
      if(c=='r') return '6';
}

string reducecode(string a){
      string code = "";
      for(int i=0;i<a.length();i++){
            int j=i;
            while(a[i]==a[j]){
                  j++;
            }
            code+=a[i];
            i=j-1;
      }
      if(a==code) return code;
      else return reducecode(code);
}

string removezeroes(string a){
      string code = "";
      for(int i=0;i<a.length();i++){
            if(a[i]!='0') code+=a[i];
      }
      return code;
}

string phonetic(string a){
      string code="";
      code+=a[0];
      for(int i=1;i<a.length();i++){
            code+=Map(a[i]);
      }
```

```
        code = reducecode(code);
        code = removezeroes(code);
        return code;
}

int main(){
        string a;
        cin>>a;
        cout<<phonetic(a);
        return 0;
}
```

Here’s a sample output-


C:\Users\SANJAY TAYAL\Documents\phonetic.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Debug
(globals)
Project Classes Debug
algo project.cpp phonetic.cpp
C:\Users\SANJAY TAYAL\Documents\phonetic.exe
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
Shorten compiler paths
Activate Windows
Go to PC settings to activate Windows.
11:23 PM
4/30/2017

```
    return code;
}

string phonetic(string a){
    string
    code+=
    for(in
        co
    }
    code =
    code =
    return
}

int main()
    string
    cin>>a
    string
    string
    cout<<
    cout<<
    if(sou
        co
    }
    else{
        co
```

```
shanivaar shonibar
Soundex Map of shanivaar : s516
Soundex Map of shonibar : s516
Both words phonetically sound same
--------------------------------
Process exited after 7.092 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\SANJAY TAYAL\Documents\phonetic.exe
- Output Size: 1.93915462493896 MiB
- Compilation Time: 1.67s
```

```
Line: 51    Col: 12    Sel: 0    Lines: 61    Length: 1340    Insert    Done parsing in 0.016 seconds
```

The soundex map can be used to compare two words phonetically.
E.g Saturday in Hindi means ‘ Shanivaar’, while in Bengali, it’s ‘ Shonibar’.
Soundex map for the words is S516, that suggests both are phonetically similar.
This concept can be very useful.

Future Work

- In future, we can extend the sample space to have more accurate results.
- We can plot a 3-d relation between the geographical distances and the Levenshtein similarities between various languages. More often, places are close to each other geographically possess similar language patterns. Our algorithm can be used to verify this proposition.

Appendix

This was a learning project for us where we worked rigorously upon the applications and various implementations of the Levenshtein Algorithm. Although our initial objective was to design an efficient searching algorithm, we invested more time in modifying the Levenshtein Algorithm which will give us the precise results. Applying Levenshtein Distance to find the linguistic similarities has not yet been explored and this was our wholly novel concept. As far as the project is concerned, we exploited this aspect on a small scale but it can be easily furthered to find the evolutionary linguistic patterns and its correlation with geographical separations.

References

- Levenshtein Distance- https://en.wikipedia.org/wiki/Levenshtein_distance
- http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm
- Phonetic correction- https://nlp.stanford.edu/IR-book/html/htmledition/phonetic-correction-1.html