澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

**Faculty of Science and Technology**

**CISC3025 – Natural Language Processing**

# Project Task 1: Implementation and Usage of a Sequence Comparison Algorithm

**Group Members:**

Huang Yanzhen DC126732

# Table of Contents

## Introduction

In numerous areas such as auto-correction programs and biological research, we encounter the challenge of quantifying the resemblance between two words. Sequence comparison algorithms based on Levenshtein Algorithm provide a useful method for achieving this goal. It follows a series of steps that show how, and how complicatedly one word is transformed into another. This project implements a sequence comparison algorithm based on the Levenshtein distance using Python and gives an example of its usage. It also extends this to sentence comparison by extensively tokenizing the sentences.

## Background

### 2.1 Minimum Edit Distance

The Minimum Edit Distance between two strings stands for the minimum cost of insertion, deletion and substitution needed to be performed in order to transform one string to another. The Minimum Edit Distance between two strings provides a way of quantifying the similarity between two strings.

### 2.2 String Tokenization

String tokenization is the process of parsing a string (including spaces) into token segments in accordance to a specific rule defined by regular expression and delimeters. String tokenization allows us to extract common features from various strings.

## Approach & Challenges

### 3.1 Introduction of Table Class

Encapsulation and simplification are the most essential ways of resoluting hard problems. The first challenge to be faced is that there isn't any sufficient libraries to support the requirement of using a table for dynamic programming. Using 2D arrays indeed works, but it takes a lot of time to consider the correct structure of it. For example, to access a cell (x,y), the correct method using a 2D array is arr[y][x], which is counterintuitive and problematic. Therefore, it is necessary to introduce a *Table* class that encapsulates the 2D array to prevent redundant works.

Other than the 2D array itself, the *Table* class also encapsulated some essential methods:

| Functions (Partial) | Description |
|---|---|
| `read(x,y)` | Read the content stored in (x,y). |
| `write(x,y,val)` | Write an intended value into cell (x,y). |
| `fill(coord1, coord2, val)` | Fill all cells within the range defined by the two coordinates an intended value. |
| `levenshtein_init()` | Initialize the table using Levenshtein distance. |
| `print_table()` | Print the data stored in the 2D array inside the class in a neat way. |

### 3.2 Basic Algorithm Construction

Given two different words, the goal of this algorithm is to find the uniquely quantified similarity of them. It defines this quantified similarity by first quantifying the cost of editing operations, which includes:

| Operations | Cost | Description |
|---|---|---|
| `Insertion` | 1 | Insert a letter before or after another. |
| `Deletion` | 1 | Delete an existing letter. |
| `Substitution` | 2 | Replace a letter with another. |

Roughly, it defines the Minimum Edit Distance (MED) of two words as the minimum cost to transform from one to another using the three operations defined above. The inductive definition of Minimum Edit Distance is as follows:

***Definition. Minimum Edit Distance***

For two strings: X of length n, Y of length m;

Define the minimum edit distance between the prefixes X[1:i] and Y[1:i] for i ∈[1, n], j∈[1,m] as D(i, j).

Then, D(n,m) is the Minimum Edit Distance of X and Y.

To tradeoff time and space complexity of calculation, dynamic programming is applied to calculate the MED of two strings. From the ground up, compute D(i, j) for all i ∈[1, n] and j∈[1,m] by introducing a newer value based on the older ones.

**Algorithm 1.1:** Leveshtein sequence MED calculation

**Procedure** Proc(x, y);
Let D(i, 0) = D(i-1, 0) + 1 for all i∈[1, n];
Let D(0, j) = D(0, j-1) + 1 for all j∈[1, m];
**for** i = 1 … N **do**
  **for** j = 1 … M **do**

$$D(i,j) \leftarrow \min \begin{cases} D(i-1,j-1) + \begin{cases} 2 \; if \; x_i \neq y_j \\ 0 \; if \; x_i = y_j \end{cases} \\ D(i-1,j) + 1 \\ D(i,j-1) + 1 \end{cases}.$$

**Output** D(n,m).

It is essential to distinguish the two input strings as Template String and Operand String. Template String is the anchor of comparison, on which all altering of Operand string is based. Operand String is the string being changed, whose goal of transforming is the Template String.

In the following table, Template String lies on the x-axis while the Operand string lies on the y-axis. The first row and the first column is initialized trivially: Comparing any prefixes of a string to an empty string, the operation cost is always length of the prefix itself. The dynamic programming table for calculating the MED between *execution* and *intention* is shown as follows:

**Table 1:** Value table of MED calculation

| | # | E | X | E | C | U | T | I | O | N |
|---|---|---|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| I | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 6 | 7 | 8 |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 7 | 8 | 7 |
| T | 3 | 4 | 5 | 6 | 7 | 8 | 7 | 8 | 9 | 8 |
| E | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 |
| N | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 10 |
| T | 6 | 5 | 6 | 7 | 8 | 9 | 8 | 9 | 10 | 11 |
| I | 7 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 9 | 10 |
| O | 8 | 7 | 8 | 9 | 10 | 11 | 10 | 9 | 8 | 9 |
| N | 9 | 8 | 9 | 10 | 11 | 12 | 11 | 10 | 9 | 8 |

Here, 8 is the MED between the word *execution* and *intention*.

Another functionality of this algorithm is to remember the operation on each cell, such that one can trace back to how the MED is calculated. At each cell above, the alrogithm face the tradeoff of three possible selections correspondingly: *Insertion, Deletion,* and *Substitution.* The algorithm performs a selection based on their operation costs demonstrated in Algorithm 1.1. Moreover, to reduce calculation time, a prioritized selection is made: If any two of them are equal, the selection sequence is *Substitution* > *Insertion* > *Deletion.* The modified version of algorithm 1.1 is shown below:

**Algorithm 1.2:** Leveshtein sequence MED calculation

1 **Procedure** Proc(x, y);

2 Let D(i, 0) ← D(i-1, 0) + 1, P(i, 0) ← Insertion for all i∈[1, n];

3 Let D(0, j) ← D(0, j-1) + 1, P(0, j) ←Deletetion for all j∈[1, m];

4 **for** i = 1 … N **do**

5 **for** j = 1 … M **do**

$$D(i,j) \leftarrow \min \begin{cases} D(i-1,j-1) + \begin{cases} 2 \; if \; x_i \neq y_j \\ 0 \; if \; x_i = y_j \end{cases} \\ D(i-1,j) + 1 \\ D(i,j-1) + 1 \end{cases}.$$

$$P(i,j) \leftarrow \text{Prioritized} \begin{cases} Substitution \; if \; D(i,j) = D(i-1,j-1) + 2 \\ Insertion \; if \; D(i,j) = D(i-1,j) + 1 \\ Deletion \; if \; D(i,j) = D(i,j-1) + 1 \end{cases}$$

6 **Output** D(n,m), **BackTrack**(P).

Again, the Template String is on the x-axis while the Operand String is on the y-axis. When the prefixes of Template String is compared with the empty string (first row of the Operation Table), it indicates an insertion should be implemented on the Operand String, and vise versa. This explains the initialization process of the Operation Table in line 2 and 3. Correspondingly, the operation table is formed using the algorithms.

**Table 2:** Operation table of MED calculation

| | # | E | X | E | C | U | T | I | O | N |
|---|---|---|---|---|---|---|---|---|---|---|
| # | - | i | i | i | i | i | i | i | i | i |
| I | d | s | s | s | s | s | s | - | i | i |
| N | d | s | s | s | s | s | s | d | s | - |
| T | d | d | s | s | s | s | - | i | s | d |
| E | d | - | i | - | i | i | i | s | s | d |
| N | d | d | d | s | s | s | s | s | s | - |
| T | d | d | s | s | s | s | - | i | i | i |
| I | d | d | s | s | s | s | d | - | i | i |
| O | d | d | s | s | s | s | d | d | - | i |
| N | d | d | s | s | s | s | d | d | d | - |

In Table 2, the hyphen sign "-" indicates that the data has never been modified since the initialization of the table. This means a match of the two letter. Backtracking can be done through this mapping from operations to movements on the operation table:

**Table 3:** Map from operations to movements

| Operation | Movements | Geometric Move |
|---|---|---|
| Substitution \|\| Match | Pointers of both template strings decreases by 1. | Move diagnally. |
| Insertion | Pointer of template string decreases by 1, pointer of operation string remain static. | Move left. |
| Deletion | Pointer of template string remains static, pointer of operation string decreases by 1. | Move up. |

By implementing this idea, the backtracking algorithm can be defined as:

**Algorithm 2:** Backtrack Operation Table

```
Procedure BackTrack(P);
Let cur_x ← P.length(x), cur_y ← P.length(y);
Let op_track ← {};
while cur_x >= 0 and cur_y >= 0 do
    cur_op ← P.read(cur_x, cur_y);
    op_track.append(cur_op)
    if cur_op = Substitution or cur_op = Match do
        cur_x --;   cur_y--;
    if cur_op = Insertion do
        cur_x--;
    if cur_op = Deletion do
        cur_y--;
Output op_track.
```

## 3.3 Array Restoration and String Alignment

### *3.3.1 Algorithm Ideas*

To fully define what operation is finally made, it is required to align the two strings. The finally alignment of the given example is:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Template String:** | - | E | X | **E** | C | U | **T** | **I** | **O** | **N** |
| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| |
| **Operand String:** | I | N | T | **E** | - | N | **T** | **I** | **O** | **N** |
| **Operations:** | d | s | s | **-** | i | s | **-** | **-** | **-** | **-** |

Note that the Template String shouldn't be altered, while all the operations are performed on the Operand String. The hyphen at the Template String indicates a *Deletion* in the corresponding position of Operand String, while the hyphen at the operand string denotes an *Insertion* in the Operand String. The algorithm restores these two strings into arrays using the following idea.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | (1) | | | (2) | | | | | |
| **Reversesd Template String:** | **E** | X | E | **C** | U | T | I | O | N |
| | (1) | | | | (2) | | | | |
| **Reversed Operand String:** | **I** | N | T | E | **N** | T | I | N | I |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | (1) | | | | (2) | | | | | |
| **Backtracked Operations:** | **d** | s | s | - | **i** | i | - | s | s | d |

(1) At the beginning position of letter "E" in the Template String, a *Deletion* is told to be performed. In this case, the corresponding letter in the Operand String "I" should be deleted, so it should be matched to a hyphen. Therefore the algorithm need to insert a hyphen before the letter "E" in the Template String. After that, since it still don't know which letter letter "E" should match to, the pointer at the Operand String proceeds to the letter "N" while the pointer at the Template String doesn't proceed.
(2) The opposite works the same: At the position C in the Template string, where the corresponding letter in the Operand String is the second "N", an *Insertion* is instructed, meaning that the letter "C" in Template String matches to a hyphen. Therefore a hyphen before "N" in the operand string should be inserted. Here, the pointer at the Operand String needs to remain while the pointer at the Template String should proceed from "C" to "U", facing another choice of operations.

One common case is that the algorithm needs to wait for a long time until it see a *Substitution* or a *Match*, meaning that one pointer, either at the Template String or the Operand String, should not proceed for a long time. That is to say, each letter in either strings can work as a storage of hyphens, whose number represents how many *Insertions* (for hyphens inserted in Operand String) or *Deletions* (for hyphens inserted in Templated String) have been occurred. To fully restore the aligned string, the algorithm should print all the hyphens one letter stored before printing the letter itself. This is the basic idea of the restoration algorithm.

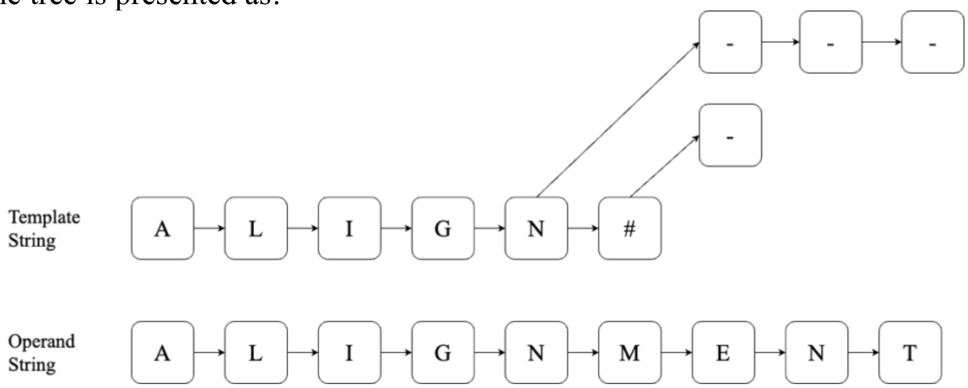
### *3.3.2 Algorithm Implementation*

Since directly inserting hyphens into array will mess around with the pointer indexes, the alignment array generating algorithm is implemented using a Tree structure for better clarity. I defined a *Node* class to maintain the pointers. Pre-order traverse is conducted to ensure that all the hyphens one letter stores comes before the letter.

The main chain of the tree is the string array itself, whereas the left node of any letter is a list storing arbitrary number of hyphens. Take the alignment between the word *align* and *alignment* as an example:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Template String:** | A | L | I | G | - | - | - | N | - |
| | \| | \| | \| | \| | \| | \| | \| | \| | \| |
| **Operand String:** | A | L | I | G | N | M | E | N | T |
| **Operations:** | - | - | - | - | d | d | d | - | d |

Here, the letter N in the Template String "ALIGN" stores three hyphens in its left child of the tree. The tree is presented as:




Pre-order traverse is then performed when iterating along the tree. This ensures all stored hyphens by a letter are printed before the letter who stores it is printed. This consolidates the core of the algorithm. There are some other essentials, like I need to add an extra buffer to the strings since there may be *Insertions* or *Deletions* at the very end and a placeholder is needed to store the extra hyphens. For instance, the buffer "#" here in the Template String stores the extra hyphen for matching the last letter T in the Operand String.

Having these ideas, the algorithms for restoring alignment is shown below:

**Algorithm 3.1:** Edit tree for restoration

```
Procedure Restore(x_root, y_root, op_track);
Let track_ptr ← 0; x_node_ptr ← x_root, y_node_ptr ← y_root;
while x_node_ptr != NULL and y_node_ptr != NULL do
    if op_track[track_ptr] = Insertion do
        y_node_ptr.insertToLeft("-");
        x_node_ptr ← x_node_ptr.next();
    else if op_track[track_ptr] = Deletion do
        x_node_ptr.insertToLeft("-");
        y_node_ptr ← y_node_ptr.next();
    else if op_track[track_ptr] = Substitution or Match do
        x_node_ptr ← x_node_ptr.next();
        y_node_ptr ← y_node_ptr.next();
    track_ptr ++;
Output x_root, y_root.
```

**Algorithm 3.2:** Traverse tree to restore alignment array

**Procedure** Traverse(root, result);
**if** result = NULL **do**
    Let result = {};
**if** root != NULL **do**
    **Traverse**(root.getLeft(), result);
    result ← result + root.getVal();
    **Traverse**(root.getNext(), result);
**Output** result.

Having algorithm 3.1 and 3.2 defined, given any two strings and their alignment operation track, it is able to fully restore their alignment as defined before.

### 3.4 String Tokenization

To extend the computation of word edit distances to the computation of sentence distances, the only extra thing I should do is to tokenize sentences into individual word tokens, then compare the sequence of word tokens just like how it compares two words.

To perform this task, I used the regular expression library of python to define the delimiter for tokenizing. This delimiter includes spaces, tabs, hyphens and other punctuation marks and symbols. After tokenizing, a raw array of the sentence is generated, which may contain some empty string members. The function then further remove those elements. The python code for word tokenization is shown as below:

**Code 1:** Python code for word tokenization

```
def sentence_preprocess(sentence):
    # Define the splitting delimiters using regular expression.
    rule = r'[\s\~\`\!\@\#\$\%\^\&\*\(\)\-\_\+\=\{\}\[\]\;\:\'\"\,\<\.\>\/\?\\|]+'
    re.compile(rule)

    # Store distinct tokens into array.
    # This may contain empty member '' (empty string).
    tokens_ = []
    # Since we consider it case-sensitive, no need to convert to lowercase here.
    tokens_ = tokens_ + re.split(rule, sentence)

    # Remove the potential empty member ''
    tokens = []
    for term in tokens_:
        if term != '':
            tokens.append(term)
    return tokens
```

### 3.5 Batch Word & Batch Sentence

There are two essential subtask to batch calculate word similarities compared to reference. First, line wise read the input .txt file. Then, identify and separate the head code (H or R). Lastly, perform comparison, get the edit distance, and write it back to the word file.

The python code for implementing this task is shown below:

**Code 2:** Python code for batch word

```
def batch_word(input_file, output_file=None):
   # Open files, store lines into array.
    with open(input_file, "r") as file:
        data = file.readlines()

    # Define a rule to split the line into code and words.
    rule= r'[\s]+'
    re.compile(rule)

    # Start to process.
    cur_anchor = ""          # Code-R words
    code_and_words = []      # Stored instances of code, words and edit dist.
    for token in data:
        code_and_word = re.split(rule,token)
        if code_and_word[0]=="R":
            cur_anchor = code_and_word[1]
            code_and_words.append(code_and_word)
        elif code_and_word[0]=="H":
            [edit_distance,_] = word_edit_distance(cur_anchor,code_and_word[1])
            code_and_word[2] = str(edit_distance)
            code_and_words.append(code_and_word)
        else:
            raise Exception("Invalid header code!")

    # Initialize output
    output = ""
    for code_and_word in code_and_words:
        item = code_and_word[0] + " " + code_and_word[1] + " " + code_and_word[2] + "\n"
        output = output + item
    print(output)

    # Write output to external file.
    if output_file is not None:
        with open(output_file,"w") as o_file:
            o_file.write(output)
```

First, the file was read line-wisely into an array, whose item contains the H/R code, the word itself, and an empty member ‘’. I define a regex rule to split them into a tuple. A sample tuple is:

`[‘R’, ‘raining’, ‘’]`

Then, traverse the array of this kind of tuples. As long as a tuple with code “R” is encountered, change the reference to the word in that tuple. Once a reference is established, the function compare each word with code “H” using the pre-defined word_edit_distance function. Lastly, it replaces the empty member with the edit distance, and re-construct each tuple back to a string. Batch sentence and batch word are essentially the same.

## Results

## 4.1 Requirement 1: Word Edit Distance (Case Sensitive)

### *4.1.1 Standard Showcases:*

Intention & Execution (Course Example)

```
Terminal   Local
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % python edit_distance.py -w I
NTENTION EXECUTION
The cost is: 8
An possible alignment is:
I N T E - N T I O N
| | | | | | | | | |
- E X E C U T I O N
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

AACGCA & GAGCTA (Project Requirement Example)

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task
_1 % python edit_distance.py -w AACGCA GAGCTA
The cost is: 4
An possible alignment is:
A A C G C - A
| | | | | | |
G A - G C T A
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task
_1 %
```

### *4.1.2 Exreme Showcases*

1. Two empty strings

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % pyth
on edit_distance.py -w "" ""
The cost is: 0
An possible alignment is:



(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

2. Exact same word

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % pyth
on edit_distance.py -w HAPPY HAPPY
The cost is: 0
An possible alignment is:
H A P P Y
| | | | |
H A P P Y
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

3. One word is the prefix of another

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % pyth
on edit_distance.py -w ALIGNMENT ALIGN
The cost is: 4
An possible alignment is:
A L I G N M E N T
| | | | | | | | |
A L I G - - - N -
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % pyth
on edit_distance.py -w ALIGNMEST ALIGN
The cost is: 4
An possible alignment is:
A L I G N M E S T
| | | | | | | | |
A L I G N - - - -
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

4. Very long words

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % pyth
on edit_distance.py -w AGGCTATCACCTGACCTCCAGGCCGATGCCC TAGCTATCACGACCGCGGTCGATTTGCCCGAC
The cost is: 15
An possible alignment is:
- A G G C T A T C A C C T G A C C T C C A G G C C G A - - T G - C C - - C
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
T A - G C T A T C A - C - G A C C - G C - G G T C G A T T T G C C C G A C
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

One interesting phenomenon is that this algorithm seems to prioritize further matches over nearer ones. One important factor that results in this difference I found is the prioiritized selection facing the same cost of *Insertion, Deletion* and *Substitution* during each step of the computation. The tables below shows the results of the different prioritization methods.

**Prioritize "Substitution/Match"**

```
Operation Table:
   #  A  L  I  G  N

#  -  i  i  i  i  i

A  d  -  i  i  i  i

L  d  d  -  i  i  i

I  d  d  d  -  i  i

G  d  d  d  d  -  i

N  d  d  d  d  d  -

M  d  d  d  d  d  d

E  d  d  d  d  d  d

N  d  d  d  d  d  -

T  d  d  d  d  d  d

The cost is: 4
An possible alignment is:
A L I G - - - N -
| | | | | | | | |
A L I G N M E N T
(base) huangyanzhen@HYZ-2 CISC3025_Proje
ct_Task_1 %
```

**Prioritize "Deletion"**

```
Operation Table:
   #  A  L  I  G  N

#  -  i  i  i  i  i

A  d  -  i  i  i  i

L  d  d  -  i  i  i

I  d  d  d  -  i  i

G  d  d  d  d  -  i

N  d  d  d  d  d  -

M  d  d  d  d  d  d

E  d  d  d  d  d  d

N  d  d  d  d  d  d

T  d  d  d  d  d  d

The cost is: 4
An possible alignment is:
A L I G N - - - -
| | | | | | | | |
A L I G N M E N T
(base) huangyanzhen@HYZ-2 CISC3025_Proje
ct_Task_1 %
```

It is obvious that at the crucial part is the alignment of the first "N" in *align* and the second "N" in *alignment*. Here, the *Substitution*-prioritize algorithm selects *Match* while the *Deletion*-prioritize algorithm selects *Deletion*, having both costs the same. The tiny change in the tailer part of the table results in a different track of operations, thus a different way of aligning.

### 4.2 Requirement 2: Sentence Edit Distance (Case Sensitive)

**Project Requirement Example:**

“I love natural language processing.” & “I really like natural language processing course.”

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % python edit_distance.py -s "I love natural language processing.
 "I really like natural language processing course."
The cost is: 4
An possible alignment is:
        I          -       love    natural   language processing          -
        |          |          |          |          |          |          |
        I     really       like    natural   language processing     course
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

**Contains one to two common words:**

“Cake is good” & “The cake is a lie.”

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % python edit_distance.py -s "Cake is good." "The cake is a lie."
The cost is: 6
An possible alignment is:
   - Cake   is    - good
   |    |    |    |    |
 The cake   is    a  lie
```

**Contains a lot of common words:**

“How many cookies could a good cook cook if a good cook could cook cookies?” & "A good cook could cook as much cookies as a good cook who could cook cookies."

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % python edit_distance.py -s "How many cookies could a good cook c
ook if a good cook could cook cookies?" "A good cook could cook as much cookies as a good cook who could cook cookies."
The cost is: 13
An possible alignment is:
    How    many cookies   could       a    good    cook       -    cook       -       -       -      if       a    good    cook       -   could    cook cookies
      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
      -       -       -       -       A    good    cook   could    cook      as    much cookies      as       a    good    cook     who   could    cook cookies
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

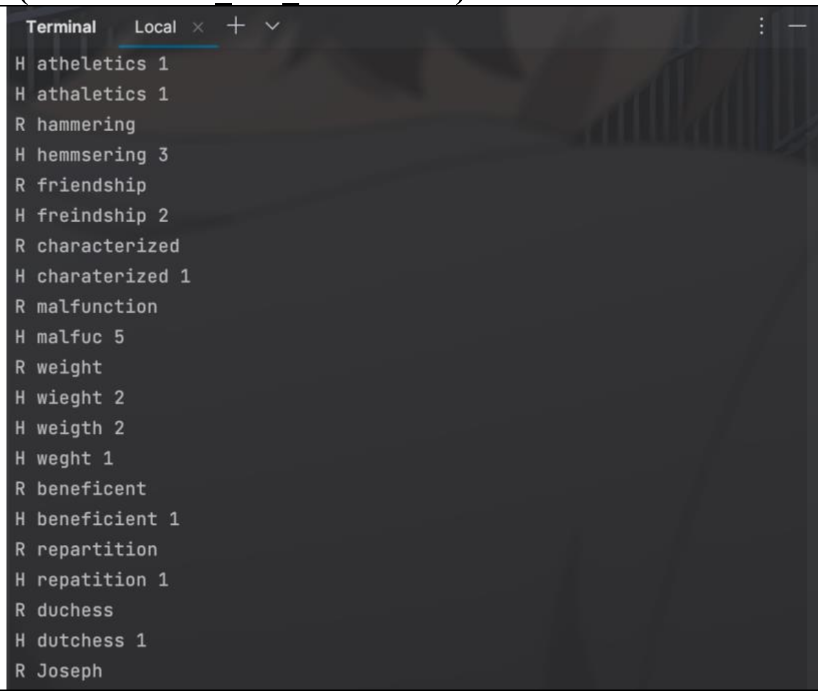
**Revert order of two words:**

“Be happy and cheerful.” & “Be cheerful and happy.”

```
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 % python edit_distance.py -s "Be happy and cheerful." "Be cheerful and happy."
The cost is: 4
An possible alignment is:
      Be    happy      and cheerful
       |        |        |        |
      Be cheerful      and    happy
(base) huangyanzhen@HYZ-2 CISC3025_Project_Task_1 %
```

### 4.3 Requirement 3: Word Corpus

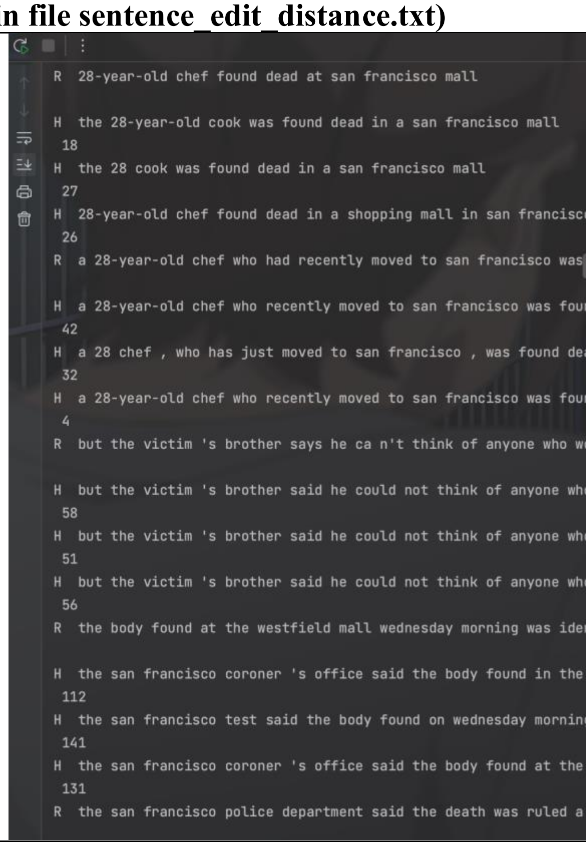
| Partial Output (More in word_edit_distance.txt) |
|---|



```
Terminal   Local
H atheletics 1
H athaletics 1
R hammering
H hemmsering 3
R friendship
H freindship 2
R characterized
H charaterized 1
R malfunction
H malfuc 5
R weight
H wieght 2
H weigth 2
H weght 1
R beneficent
H beneficient 1
R repartition
H repatition 1
R duchess
H dutchess 1
R Joseph
```

### 4.4 Requirement 4: Sentance Corpus

| Partial Output (More in file sentence_edit_distance.txt) |
|---|



```
R  28-year-old chef found dead at san francisco mall

H  the 28-year-old cook was found dead in a san francisco mall
 18
H  the 28 cook was found dead in a san francisco mall
 27
H  28-year-old chef found dead in a shopping mall in san francisco
 26
R  a 28-year-old chef who had recently moved to san francisco was

H  a 28-year-old chef who recently moved to san francisco was foun
 42
H  a 28 chef , who has just moved to san francisco , was found dea
 32
H  a 28-year-old chef who recently moved to san francisco was foun
 4
R  but the victim 's brother says he ca n't think of anyone who wo

H  but the victim 's brother said he could not think of anyone who
 58
H  but the victim 's brother said he could not think of anyone who
 51
H  but the victim 's brother said he could not think of anyone who
 56
R  the body found at the westfield mall wednesday morning was iden

H  the san francisco coroner 's office said the body found in the
 112
H  the san francisco test said the body found on wednesday morning
 141
H  the san francisco coroner 's office said the body found at the
 131
R  the san francisco police department said the death was ruled a
```

## Conclusion

Through implementing this project, I’ve had a deeper view of the dynamic programming process of calculating the Minimum Edit Distance, as well as backtracking the operations performed to transform one word string to another. I condcluded the initialization steps of the dynamic programming table and a prioritized selection method facing the dilemma of same costs. I also found an interesting feature of this algorithm, which is that different priority selection sequence performed during the process may lead to different aligning alternatives.

Moreover, with string tokenization, I’m able to easily extend the usage of this algorithm to computing sentence edit distance. Having the two basic processing functions, I have got the ability to batch process multiple words and sentences.