|  |  |  |
| --- | --- | --- |
| A logo on a black background  Description automatically generated | | |
| **Faculty of Science and Technology**  **CISC3025 – Natural Language Processing** | | |
| **Project Task 1: Implementation and Usage of a Sequence Comparison Algorithm using Levenshtein Distance** | | |
| **Group Members:** | | |
| Huang Yanzhen | | DC126732 |
|  |  |  |

# 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 one word is transformed into another. This project implements a sequence comparison algorithm based on the Levenshtein distance using Python. It also extends this to sentence comparison by extensively tokenizing the sentences.

# Background

# Approach & Challenges

## 3.1 Introduction of Table Class

Encapsulation and simplification are the most essential way 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 an encapsulated *Table* class to prevent redundant works.

The *Table* class 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. |
| levenshtien\_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.s

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, we 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 | |
| 1 | **Procedure** Proc(x, y); |
| 2 | Let D(i, 0) = D(i-1, 0) + 1; |
| 3 | Let D(0, j) = D(0, j-1) + 1; |
| 4 | **for** i = 1 … N **do** |
| 5 | **for** j = 1 … M **do** |
|  | D(i, j) ← min |
| 6 | **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 sequence of the letter 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, we face the tradeoff of three possible selections: *Insertion, Deletion,* and *Substitution.* The algorithm performs a selection based on their operation costs emonstrated 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 algorithm 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; |
| 3 | Let D(0, j) ← D(0, j-1) + 1, P(0, j) ←Deletetion; |
| 4 | **for** i = 1 … N **do** |
| 5 | **for** j = 1 … M **do** |
|  | D(i, j) ← min  P(i, j) ← Prioritized |
| 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 | |
| 1 | **Procedure** BackTrack(P); |
| 2 | Let cur\_x ← P.length(x), cur\_y ← P.length(y); |
| 3 | Let op\_track ← {} |
| 4 | **while** cur\_x >= 0 and cur\_y >= 0 **do** |
| 5 | cur\_op ← P.read(cur\_x, cur\_y); |
| 6 | op\_track.append(cur\_op) |
| 7 | **if** cur\_op = Substitution or cur\_op = Match **do** |
| 8 | cur\_x --; cur\_y--; |
| 9 | **if** cur\_op = Insertion **do** |
| 10 | cur\_x--; |
| 11 | **if** cur\_op = Deletion **do** |
| 12 | cur\_y--; |
| 13 | **Output** op\_track. |

## 3.3 Array Restoration and String Alignment

To fully define what operation is finally made, we need 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** |  |
|  | d | s | s | **-** | i | s | **-** | **-** | **-** | **-** | ← **Backtracked Operations** |

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. We need to restore these two strings in arrays.

Due to the reverse nature of backtracking, the two strings are implemented also in a reversed manner. The graph below demonstrates the idea of the restoration:

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Backtracked Operations:** | - | - | - | - | s | **i** | - | s | s | **d** |

At the position C in the reversed Template String, an *Insertion* needs to be performed. We need to insert a hyphen between N and E in the operand string. At the highlighted position I in the reversed Operand String, a *Deletion* needs to be performed. We need to insert a hyphen

# Results

# Conclusion