Levenshtein Distance: Implementation, Optimization, and Applications

Turan Abbasli

March 6, 2025

Abstract

The Levenshtein distance, also known as Edit Distance, measures the minimum number of single-character operations (insertion, deletion, substitution) required to transform one string into another. This report explores its mathematical formulation, various implementations, and optimizations. We discuss the recursive, dynamic programming, and optimized dynamic programming approaches, highlighting their time and space complexities. A performance analysis is conducted to compare these methods, finding out computation time is least in dynamic programming approach, with a focus on improving efficiency for practical applications such as spell checking. In addition to standart Levenshtein distance, we have also discussed other string comparison metric, such as Damerau-Levenshtein, Jaro similariy, Jaro-Winkler similarity, and Hamming distance. This report also mentions main obstacles for these algorithms and their drawbacks.

Contents

1 Introduction

The Levenshtein distance, also known as **Edit Distance**, was introduced in 1965 by Vladimir Levenshtein. The algorithm measures the minimum number of single-character operations, including deletion, insertion, and substitution, required to transform one string from another [1]. The resulting count of edits represents how similar the given two strings are. Although there are wide range of its applications, such as plagiarism checking and DNA mutation detecting, this report mainly focuses on use of Levenshtein distance in spell checking.

2 Basics of Levenshtein Distance

The Levenshtein distance is an algorithm used to calculate how different two given words are by using three different operations - deletion, insertion, and substitution (1) [2].

$$lev(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min \begin{cases} lev(i-1,j)+1 & \text{(deletion)} \\ lev(i,j-1)+1 & \text{(insertion)} \\ lev(i-1,j-1)+1(s_i \neq t_j) & \text{(substitution)} \end{cases} & \text{otherwise} \end{cases} \quad (1)$$

Let s_1 be the first word, and s_2 be the second word. The formula $lev(i-1,j)+1$, in the case of a **deletion** operation, means that the i-th letter of s_1 has been removed, the +1 edit count has been added. The algorithm will then check again the number of edits required to convert the first i-1 letters of s_1 into the first j letters of s_2. Similarly, $lev(i,j-1)+1$ in the case of an **insertion** operation indicates that after inserting the j-th letter of s_2 into s_1 and adding the +1 edit count. The algorithm then checks the number of edits required to convert the first i letters of s_1 into the first j-1 letters of s_2.

On the other hand, when **substitution** is required, we increment the edit count by one, only if two letters are different, and move back one letter in both words, $lev(i-1,j-1)$. Notably, setting the edit distance equal to $max(i,j)$ when $min(i,j) = 0$ simply means that if one of the words is empty, we should either add up or remove the number of letters from the word that

has letters in it, end of recursion.

3 Implementations of Levenshtein Distance

In this section we are going to explore three different python implementations of the Levenshtein distance by introducing their time and space complexities and solutions to optimize them.

3.1 Recursive Implementation

This method directly follows the mathematical formula for the Levenshtein distance (1). Since in the worst-case scenario there are three possible paths, the time complexity of this implementation is $O(3^n)$, where n is the length of the longest between the given two (1). An exponential complexity is observed due to the overlapping number of sub-problems.

```
def levenshtein_recursive(s1: str, s2: str) -> int:
    # Condition return max(i, j) if min(i, j) = 0
    if not s1:
        return len(s2)
    if not s2:
        return len(s1)

    # otherwise
    edit_substitution = 0 if s1[0] == s2[0] else 1
    return min(
        levenshtein_recursive(s1[1:], s2) + 1,  # Deletion
        levenshtein_recursive(s1, s2[1:]) + 1,  # Insertion
        levenshtein_recursive(s1[1:], s2[1:]) + edit_substitution  # Substitution
    )
```

Figure 1: Recursive implementation of the Levenshtein distance algorithm.

3.2 Dynamic Programming Implementation

Dynamic programming approach, on the other hand, significantly reduces the time complexity of the algorithm by memorizing past calculations in a matrix. Both the time complexity and the space complexity of this implementation are equal to $O(nm)$, where n and m are the lengths of the given words.

In this approach, we fill an $(n + 1) \times (m + 1)$ matrix, where each cell stores the Levenshtein distance between two words, labeled s_1 and s_2. The value in matrix$[i][j]$ represents the Levenshtein distance between the first i characters of s_1 and the first j characters of s_2 (2).

```
def levenshtein_dp(s1: str, s2: str) -> tuple[int, list[list[int]]]:
    n = len(s1)
    m = len(s2)

    # (n + 1, m + 1) matrix to store values
    matrix = [[0] * (m + 1) for _ in range(n + 1)]

    # filling matrix
    for i in range(n + 1):
        for j in range(m + 1):
            # if min(i, j) = 0 return max(i, j)
            if i == 0:
                matrix[i][j] = j

            elif j == 0:
                matrix[i][j] = i

            # otherwise
            else:
                # Extra edit count for substitution depending whether two letters are same
                edit_substitution = 0 if s1[i - 1] == s2[j - 1] else 1

                matrix[i][j] = min(
                    matrix[i - 1][j] + 1,                       # Deletion
                    matrix[i][j - 1] + 1,                       # Insertion
                    matrix[i - 1][j - 1] + edit_substitution    # Substitution
                )

    return matrix[-1][-1], matrix
```

Figure 2: Dynamic programming implementation of the Levenshtein distance algorithm.

3.3 Optimized Dynamic Programming Implementation

This method focuses on optimizing space complexity rather than time complexity and is quite similar to the dynamic programming approach 3.2. While

filling a cell (i, j) of the matrix 2 consisting of Levenshtein values, we only refer to the values stored in cells $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$, which are the values from the previous row and the previous column. When calculating the values of i-th row, the values after the row i are not used.

Thus, instead of keeping the entire $(n+1) \times (m+1)$ matrix for Levenshtein distances, we can reduce the space complexity by storing only the current and previous rows in a $2 \times (m+1)$ matrix.

As only two rows are required to calculate any Levenshtain distance in between first i letters of s_1 and s_2, we can process the matrix without losing necessary information. This optimization reduces the space complexity from $O(nm)$ to $O(m)$, where m is the length of the second string, significantly reducing memory usage for large strings.

3.4 Performance Analysis of the Implementations

In order to evaluate the efficiency of different Levenshtein distance implementations, execution time analysis has been conducted using Python's `timeit` module. We compared three versions:

- **Recursive implementation:** The naive approach, which has exponential time complexity $O(3^n)$, see 3.1.
- **Dynamic Programming (DP) implementation:** An optimized version with a time complexity and a space complexity of $O(mn)$, using a 2D table, see 3.2.
- **Optimized DP implementation:** A further improved version that reduces space complexity to $O(m)$, see 3.3.

The experiment was conducted using the input strings `"sitting"` and `"kitten"`. Each function was executed **1000 times** to obtain a stable average runtime.

The performance test was conducted on a system equipped with an Intel i5-11400H mobile processor (6 cores, 12 threads) running at 2.70 GHz, with 16 GB of DDR4 RAM. The experiment was executed using Python 3.10.4.

The measured execution times were as follows:

```
def levenshtein_dp_optimized(s1: str, s2: str) -> int:
    n = len(s1)
    m = len(s2)

    # (2 x (m + 1)) matrix to store values
    matrix = [[0] * (m + 1) for _ in range(2)]

    # filling the matrix
    for i in range(n + 1):
        for j in range(m + 1):
            # if min(i, j) = 0 return max(i, j)
            if i == 0:
                matrix[i % 2][j] = j
            elif j == 0:
                matrix[i % 2][j] = i

            # otherwise
            else:
                # Extra edit count for substitution depending whether two letters are same
                edit_substitution = 0 if s1[i - 1] == s2[j - 1] else 1

                matrix[i % 2][j] = min(
                    matrix[(i - 1) % 2][j] + 1,                     # Deletion
                    matrix[i % 2][j - 1] + 1,                       # Insertion
                    matrix[(i - 1) % 2][j - 1] + edit_substitution  # Substitution
                )

    return matrix[-1][-1]
```

Figure 3: Optimized dynamic programming implementation of the Levenshtein distance algorithm.

Due to exponential time complexity of recursive approach, an expected massive difference has been observed between recursive implementation and dynamic programming ones.

Implementation	Execution Time (1000 runs)
Recursive	7.351031 sec
DP	0.035177 sec
Optimized DP	0.031729 sec

Table 1: Execution Time Comparison of Different Levenshtein Implementations

4 Related String Similarity Metrics

4.1 Damerau-Levenshtein Distance

By Damerau-Levenshtein distance in addition to deletion, insertion, and substitution a new string operation has been introduced - **Transposition**.

$$\min \begin{cases} dl(i-1,j)+1 & \text{(deletion)} \\ dl(i,j-1)+1 & \text{(insertion)} \\ dl(i-1,j-1)+1(s_i \neq t_j) & \text{(substitution)} \\ dl(i-2,j-2)+1, & \text{if } i,j>1 \text{ and } s_i = t_{j-1} \text{ and } s_{i-1} = t_j \text{ (transposition)} \end{cases}$$

(2)

As row index 0 and column index 0 represents minimum number of edit operations needed to create other word, and we can only check possible transposition operation starting from the second letters of both words, condition $i, j > 1$ has been set. If transposition can be done, edit count is incremented by 1, and coordinates are decreased by 2.

In order to reduce space complexity, instead of 2 3.3, this time we keep three rows to store values which is necessary to return back 2 steps in case of transposition. However, the space complexity will still be $O(m)$, and time complexity will be $O(nm)$.

4.2 Jaro Similarity

Jaro similarity is a measurement method to calculate how two given words are similar, particularly useful when finding duplicates [2]. In case of complete

similar the output of the algorithm is equal to 1, while in case of complete difference it is equal to 0.

$$Jaro\ similarity\ (s_1, s_2) = \begin{cases} 0, & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right), & otherwise \end{cases} \tag{3}$$

In the given formula 3, m represents the count of matching letters in s_1 *and* s_2, t is half of letters which can be transposed, and $|s_1|, |s_2|$ are total number of letters in s_1 and s_2, respectively.

In order to decide whether a pair of strings are matching or not matching distance is used.

$$maximum\ matching\ distance = \frac{max(|s_1|, |s_2|)}{2} - 1 \tag{4}$$

If a pair of letter is positioned within maximum matching distance, they are considered matching. In this way, we avoid significant reduction in similarity score, if a small shift has been happened while typing. On the other hand, the penalizing factor $\frac{m-t}{m}$ ensures that in case of strings where matches appear in different orders reduction of the similarity score appropriately.

Since in the worst case scenario each string of the first word must be compared to all string of the second word, the time complexity of Jaro similarity is $O(nm)$, where n and m are length of the given words. Space complexity, on the other hand, will be $O(n + m)$.

4.3 Jaro-Winkler Similarity

Similar to Jaro 4.2, Jaro-Winkler also shows how similar two given words are by returning a score between 0 and 1 [2]. The formula of Jaro-Winkler uses new terms (l and p) in addition to Jaro similarity value (5).

$$Jaro\ Winkler\ (s_1, s_2) = Jaro\ (s_1, s_2) + lp(1 - Jaro\ (s_1, s_2)) \tag{5}$$

Where,

- **l**: Measures length of matching prefix length at the beginning of both strings. In order to avoid bias in small strings maximum length of prefix has been set as 4.

- **p**: A scaling factor which controls influence of prefix in similarity score (default 0.1)
- **Jaro**: Jaro similarity of two given strings 4.2

Compared to Jaro similarity, Jaro-Winkler similarity is much more sensitive to matching prefixes. Thus, it is more prefered to use Jaro-Winkler algorithm when dealing matching names and misspelled words problems. The drawback of this algorithm is its biased scores for short strings.

Since Jaro-Winkler inherits Jaro similarity, and prefix check is limited to length of 4, both time complexity and space complexity will be same as Jaro similarity, $O(nm)$ and $O(n+m)$, respectively.

4.4 Hamming Distance

The Hamming distance metric measures the count of positions where letters are different (6). It is used for comparing two equal length words.

$$H(s_1, s_2) = \sum_{i=1}^{n} \delta(s_{1i}, s_{2i}) \tag{6}$$

Where,

- if a $\neq b$, $\delta(a, b) = 1$, else $= 0$.
- $\text{len}(s_1) = \text{len}(s_2)$

While Levenshtein distance considers deletion, insertion, and substitution (also transposition in Damerau-Levenshtein), Hamming distance only counts substitution.

As the comparison of strings are done in parallel, the time complexity of the Hamming distance will be $O(n)$, while the space complexity is $O(1)$, because we only need a single integer variable to keep count of differences.

5 Practical Application - Spell Check

This section covers application of Damerau-Levenshtein distance in spell checking. Since the algorithm corrects the most 4 common types (insertion, deletion, substitution, and transposition) of typos and tells us the number of

edits required to correct the given text, it is considered more appropriate to use in spell checking compared to other metrics we have talked.

For instance, while standard Levenshtein distance takes into account only 3 kinds of typos, Damerau-Levenshtein considers 4 kinds of typos, and knowing least amount of required edits is more valuable in spell checking rather than just similarity between words, Damerau-Levenshtein distance has been chosen in this example.

The dictionary we had was ['apple', 'pear', 'grape', 'google']. By comparing each word given in the dictionary with the input string 'gappeel' we found following result:

- apple: 3
- pear: 5
- grape: 4
- google: 5

Since apple requires less number of edits chosen corrected word from the dictionary was the 'apple'.

6 Challenges and Limitations

Even though mentioned metrics works quite well, there are several challenges of using this algorithms. First of all, depending on the context the algorithm can chose wrong correction for the word. For example, if user has written 'appll' it can be either corrected as 'apple' or 'apply'. Thus, current spell checking softwares also uses AI to handle this.

Furthermore, dictionaries for spell checking consists of plethora of words, and each time iterating through all of them results in extra time complexity of $O(n)$, where n is size of the dictionary. In order to handle this issue, **trie** data structure is prefered. A trie is a tree-like structure where each node stores a letter. Using a trie, we can efficiently find words by traversing through child nodes, adding up letters along the way.

7 Conclusion

The Levenshtein distance is a fundamental string similarity metric widely used in text processing. While the recursive implementation provides a direct understanding of the algorithm, it is computationally inefficient. The dynamic programming approach significantly improves performance by reducing redundant calculations, while the optimized version further minimizes space complexity. These optimizations make Levenshtein distance practical for large-scale applications such as spell checking and natural language processing.

Bibliography

[1] Claire Lee. Levenshtein distance. *Medium*, October 2022. Accessed: 2025-03-04.

[2] Srinivas Kulkarni. Jaro winkler vs levenshtein distance. *Medium*, March 2021. Accessed: 2025-03-06.

[3] Wilbert Jan Heeringa. Measuring dialect pronunciation differences using levenshtein distance. 2004.

[4] GeeksforGeeks. Damerau–Levenshtein Distance, 2023. Last Updated: 07 Mar, 2023.

[5] Jake Cutter. Computer science — hamming distance. *Medium*, March 2022. Accessed: 2025-03-06.

[6] Timur Akra. Hamming distance. *Medium*, January 2023. Accessed: 2025-03-06.