Analysis of Levenshtein Distance for DNA Sequences

Shivek

November 12, 2025

The **Levenshtein distance** is a "string metric" that measures the difference between two sequences. It is formally defined as the **minimum number of single-character edits** (insertions, deletions, or substitutions) required to change one string into the other.

This algorithm is a foundational concept in bioinformatics, forming the basis for more complex sequence alignment algorithms.

---

1 Basic Algorithm

The Levenshtein distance is calculated using a **dynamic programming** approach. The core idea is to build a 2D matrix (or grid) that stores the distances between all prefixes of the two input strings.

Let's say we have two strings, A of length m and B of length n. We create a matrix D of size $(m+1) \times (n+1)$.

A cell $D[i][j]$ in this matrix represents the Levenshtein distance between the first i characters of string A (i.e., $A[1...i]$) and the first j characters of string B (i.e., $B[1...j]$).

The algorithm fills this matrix using two steps:

Step 1: Initialization

The first row and first column are the "base cases."

- $D[i][0] = i$: The distance from a string of length i to an empty string is i deletions.
- $D[0][j] = j$: The distance from an empty string to a string of length j is j insertions.

Step 2: Recurrence Relation

We fill the rest of the matrix, cell by cell. For any cell $D[i][j]$, we have three choices to get to this state:

1. **Deletion:** We had the solution for $D[i-1][j]$ (aligning $A[1...i-1]$ with $B[1...j]$) and we **delete** $A[i]$. Cost: $D[i-1][j]+1$.
2. **Insertion:** We had the solution for $D[i][j-1]$ (aligning $A[1...i]$ with $B[1...j-1]$) and we **insert** $B[j]$. Cost: $D[i][j-1]+1$.
3. **Substitution (or Match):** We had the solution for $D[i-1][j-1]$ (aligning $A[1...i-1]$ with $B[1...j-1]$). We now look at $A[i]$ and $B[j]$.

- If $A[i] == B[j]$ (it's a **match**), no cost is added. Cost: $D[i-1][j-1] + 0$.
- If $A[i] \neq B[j]$ (it's a **substitution**), we add 1. Cost: $D[i-1][j-1] + 1$.

The algorithm is "greedy" and always takes the minimum of these three possibilities. The complete recurrence relation is:

$$D[i][j] = \min \begin{cases} D[i-1][j] + 1 & \text{(Deletion from A)} \\ D[i][j-1] + 1 & \text{(Insertion into A)} \\ D[i-1][j-1] + \text{cost} & \text{(Match/Substitution)} \end{cases}$$

Where cost is 0 if $A[i] == B[j]$ and 1 otherwise.

The final answer (the Levenshtein distance between all of A and all of B) is the value in the bottom-right cell: $D[m][n]$.

---

2 Time and Space Complexity Analysis

2.1 Time Complexity: $O(m \times n)$

- The algorithm must fill every cell in the $(m+1) \times (n+1)$ matrix.
- The calculation for each cell, $D[i][j]$, involves three lookups, two additions, and one min operation, all of which are constant time, $O(1)$.
- Therefore, the total time complexity is the number of cells multiplied by the constant work per cell, which is $O(m \times n)$.

2.2 Space Complexity: $O(m \times n)$

- The default algorithm requires storing the entire $(m+1) \times (n+1)$ matrix in memory to compute the final value.
- This results in a space complexity of $O(m \times n)$.

Optimization: This can be optimized to $O(\min(m, n))$ space. To compute the *current* row i, we only need the values from the *previous* row $i-1$. We never need to look back further. By storing only two rows (the "previous" and "current") and alternating between them, we can reduce the space complexity significantly.

---

3 Proof of Correctness (Why it Works)

The algorithm's correctness is proven by induction.

1. **Base Case:** The initialization (Step 1) is correct. The edit distance from an empty string to a string of length j is j insertions (e.g., `""` $\to$ `"ATG"` is 3 insertions). $D[0][j] = j$ is correct. The same logic applies to $D[i][0]$.
2. **Inductive Hypothesis:** Assume that for all $i' < i$ and $j' < j$, the sub-problems $D[i'][j']$ have been solved correctly and store the true Levenshtein distance.

3. **Inductive Step:** We must prove that $D[i][j]$ is calculated correctly. Any optimal sequence of edits that transforms $A[1...i]$ to $B[1...j]$ *must* end in one of three operations:

   - **Case 1 (Deletion):** The last operation is deleting $A[i]$. This means we first optimally transformed $A[1...i-1]$ to $B[1...j]$. The total cost is (cost of that optimal transform) + 1. By our hypothesis, this is $D[i-1][j]+1$.
   - **Case 2 (Insertion):** The last operation is inserting $B[j]$. This means we first optimally transformed $A[1...i]$ to $B[1...j-1]$. The total cost is $D[i][j-1]+1$.
   - **Case 3 (Substitution/Match):** The last operation involved $A[i]$ and $B[j]$. This means we first optimally transformed $A[1...i-1]$ to $B[1...j-1]$.
     - If $A[i] == B[j]$, no new cost is added. Total cost: $D[i-1][j-1]+0$.
     - If $A[i] \neq B[j]$, the last operation was a substitution. Total cost: $D[i-1][j-1]+1$.

   Since the Levenshtein distance is the *minimum* number of edits, $D[i][j]$ must be the minimum of these three possible cases. This matches the recurrence relation. By induction, the entire matrix is filled correctly, and $D[m][n]$ holds the final answer.

---

4 Pseudo Code

Note: This pseudo-code assumes 0-based indexing for strings (e.g., $A[0]$ is the first character) and 1-based indexing for the DP matrix loops.

```
function LevenshteinDistance(string A, string B):
  // m = length of A, n = length of B
  m = A.length
  n = B.length

  // 1. Initialize the (m+1) x (n+1) matrix D
  D = new int[m + 1][n + 1]

  // 2. Fill the base cases (first row and column)
  for i from 0 to m:
    D[i][0] = i  // Cost of deleting i chars from A to get ""

  for j from 0 to n:
    D[0][j] = j  // Cost of inserting j chars to "" to get B

  // 3. Fill the rest of the matrix
  for i from 1 to m:
    for j from 1 to n:

      // Check if characters at A[i-1] and B[j-1] are the same
      // (We use i-1 and j-1 because strings are 0-indexed)
      if A[i-1] == B[j-1]:
        cost = 0
      else:
        cost = 1

      // Calculate costs for each operation
      deletion = D[i-1][j] + 1
      insertion = D[i][j-1] + 1
      substitution = D[i-1][j-1] + cost

      // The cell D[i][j] gets the minimum of these three
      D[i][j] = min(deletion, insertion, substitution)

  // 4. Return the final answer
  return D[m][n]
```

---

5 Dry Run (Biological Sequences)

Let's find the Levenshtein distance between two short DNA sequences:

- A = "CAT" (m = 3)
- B = "TAG" (n = 3)

We create a $(3 + 1) \times (3 + 1) = 4 \times 4$ matrix.

Table 1: Dry Run Matrix for "CAT" vs "TAG"

			$B \rightarrow$		
		(empty)	**T**	**A**	**G**
$A \downarrow$	**D**	**j = 0**	**j = 1**	**j = 2**	**j = 3**
(empty)	**i = 0**	**0**	**1**	**2**	**3**
C	**i = 1**	**1**	**1**	**2**	**3**
A	**i = 2**	**2**	**2**	**1**	**2**
T	**i = 3**	**3**	**2**	**2**	**2**

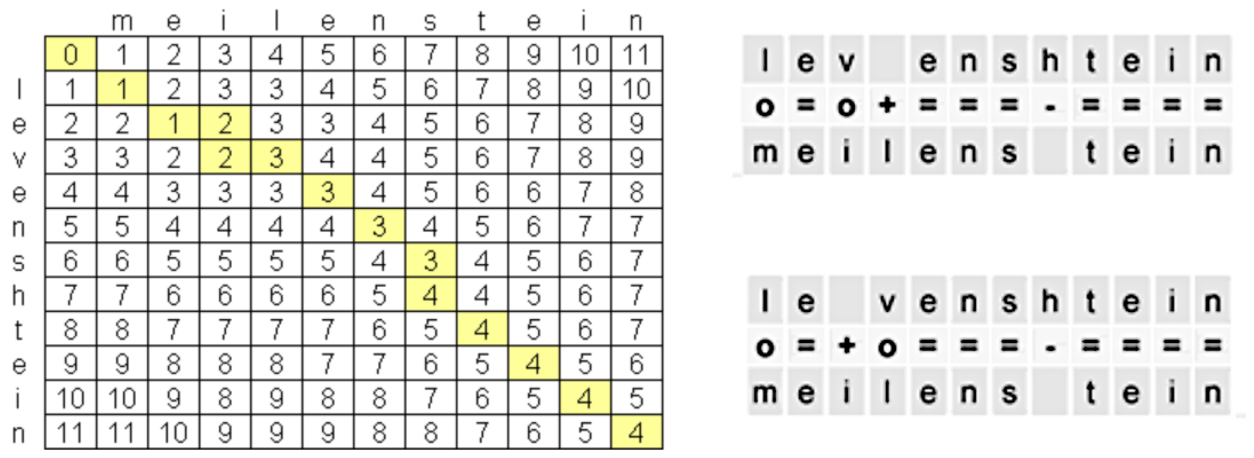
Final Answer: The Levenshtein distance is **2**.

This makes sense: $C \rightarrow T$ (substitution), $A \rightarrow A$ (match), $T \rightarrow G$ (substitution). Total: 2 substitutions.

---

6 Graphical Visualization

The completed dry run table *is* the graphical visualization. We can also visualize the **edit path** by backtracking from the final cell $D[3][3]$ to $D[0][0]$, always moving to the cell that generated the minimum value.



		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

```
l e v   e n s h t e i n
o = o + = = = - = = = =
m e i l e n s   t e i n

l e   v e n s h t e i n
o = + o = = = - = = = =
m e i l e n s   t e i n
```

Figure 1: Visualization of the DP matrix

Table 2: Matrix with Backtrace Path (←)

			$B \rightarrow$		
		(empty)	**T**	**A**	**G**
$A \downarrow$	**D**	**j = 0**	**j = 1**	**j = 2**	**j = 3**
(empty)	**i = 0**	**0** ←	1 ←	2	3
C	**i = 1**	1	**1** ←	2	3
A	**i = 2**	2	2	**1** ←	2
T	**i = 3**	3	2	2	**2**

Backtrace Path:

1. **Start at D[3][3] (value 2):** Came from $D[2][2]$ (a substitution, since $T \neq G$).
2. **At D[2][2] (value 1):** Came from $D[1][1]$ (a match, since $A == A$).
3. **At D[1][1] (value 1):** Came from $D[0][0]$ (a substitution, since $C \neq T$).

This traceback gives us the full alignment:

```
C A T
T A G
```

(Sub, Match, Sub) → 2 edits.

7 Situational Performance (On DNA Sequences)

This is the most critical part for your evaluation. Levenshtein distance is a *good* algorithm, but it's often *not* the best tool for biological sequences.

7.1 Strengths

1. **Simplicity and Accuracy:** It's a simple, easy-to-implement algorithm that gives a mathematically correct *edit distance.*
2. **Good for Short, Similar Sequences:** It works very well for comparing two short reads, or checking for SNPs (Single Nucleotide Polymorphisms) between two highly similar sequences.
3. **Foundation for Other Algorithms:** It is the direct parent of the algorithms actually used in bioinformatics. **Smith-Waterman** (for local alignment) and **Needleman-Wunsch** (for global alignment) are just Levenshtein with two key changes:
    - A more complex scoring system (e.g., scoring matrices like BLOSUM).
    - Different rules for initialization and traceback.

7.2 Weaknesses (Major Performance Issues for DNA)

1. **Biologically Naive:** This is the biggest problem. Levenshtein treats all operations as equal (cost=1). In biology, this is wrong:
    - **Substitutions:** A substitution of $A \to G$ (a **transition**) is *far* more common and biologically likely than $A \to T$ (a **transversion**). Levenshtein can't do this.
    - **Gaps:** An insertion or deletion (an "indel" or "gap") is often a single biological event, but it might delete 10 bases at once. Levenshtein would count this as 10 separate edits. Proper alignment algorithms use an "affine gap penalty".
2. **It's a *Global* Alignment:** The standard Levenshtein algorithm forces the *entire* string A to match the *entire* string B. This is almost *never* what you want in bioinformatics.
    - **Problem:** You usually want to find a *small* gene (e.g., 1,000 bases) inside a *huge* chromosome (e.g., 250,000,000 bases).
    - **Solution:** You need **local alignment** (like the **Smith-Waterman** algorithm or **BLAST**).
3. **Poor Time/Space Complexity for Genomes:** The $O(m \times n)$ complexity is completely unusable for large-scale genomics.
    - Comparing two human chromosomes ($m \approx 250M, n \approx 250M$) would require a matrix with 6.25×10^{16} cells. This is not computationally feasible.
    - This is why genomics relies on **heuristic** algorithms like **BLAST** or index-based aligners like **BWA** and **Bowtie**.

Conclusion: You should use Levenshtein distance to understand the *core concept* of dynamic programming for sequence comparison. For *actual performance on DNA sequences*, it is a poor choice. You should immediately move on to its biological-aware children: **Needleman-Wunsch** (for global alignment) and **Smith-Waterman** (for local alignment), which use scoring matrices and affine gap penalties.

8 Adaptation for Fuzzy Pattern Matching in DNA

The standard algorithm (Section 1) is a **global alignment**, comparing the *entire* string A to the *entire* string B. This is useless for finding a small pattern (e.g., a 6-base-pair motif) inside a large genome (e.g., 4.6 million base pairs).

To find all substrings in a large Text T that "fuzzily" match a Pattern P with at most k edits, we must adapt the algorithm.

Let P be the pattern of length m (on the vertical axis) and T be the genome text of length n (on the horizontal axis).

8.1 The "Slight Modification"

The algorithm is modified in two ways: the initialization and the final result-finding.

8.1.1 Modified Initialization

The key is to allow a match to **start at any point** in the text T without penalty. We achieve this by setting the first row to all zeros.

- $D[0][j] = 0$ for all j from 0 to n (text length): This means the cost of matching an empty pattern to *any* prefix of the text is 0. This "zeros" the cost for starting a new match.
- $D[i][0] = i$ for all i from 0 to m (pattern length): This remains the same. The cost to match a pattern of length i to an empty string is i deletions.

8.1.2 Modified Result Finding

The recurrence relation (Section 1.2) is filled exactly the same way. However, the answer is no longer in the single cell $D[m][n]$.

Instead, we must scan the **entire last row** (row m).

- A value $D[m][j]$ in the last row represents the minimum Levenshtein distance required to match the *entire* pattern P to a substring of the text T that *ends* at position j.
- **The Result:** We iterate j from 1 to n. If $D[m][j] \leq k$ (where k is our error threshold), we record j as the *end position* of a fuzzy match.

8.2 Complexity of the Adaptation

The time and space complexity remain the same as the standard algorithm.

- **Time:** $O(m \times n)$. This is why it is linear $O(n)$ for a fixed pattern m, but slow $O(m)$ for a fixed text n and growing pattern m.
- **Space:** $O(n)$ with the standard row-optimization.
