Natural Language Processing

Artificial Intelligence

Jacopo Parretti

I Semester 2025-2026

Indice

Parte I

1 Minimum Edit Distance

We are going to deal with this main driving point: the definition of Minimum Edit Distance.

The question: are this 2 texts the same?

When is the case when 2 texts are the same? Of course when every single character is the same. What if we would like to understand if 2 texts are pretty close (not the same)?

Single characters in the text could be different in position.

1.1 How similar are two strings?

The fundamental question in edit distance is: how can we measure the similarity between two strings? This problem appears in many different applications across various domains of computer science and computational linguistics.

1.1.1 Spell Correction

In spell correction systems, we need to find the closest valid word to a misspelled input. For example, if the user typed "graffe", we need to determine which word in our dictionary is most similar.

Possible candidates:

- graf
- graft
- grail
- giraffe

By computing the edit distance between "graffe" and each candidate, we can identify that "giraffe" is likely the intended word (requiring only one deletion).

1.1.2 Computational Biology

In bioinformatics, sequence alignment is crucial for comparing DNA, RNA, or protein sequences. By aligning sequences, we can identify regions of similarity that may indicate functional, structural, or evolutionary relationships.

Example: Align two sequences of nucleotides:

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
```

```
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

Resulting alignment (dashes represent insertions/deletions):

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

The alignment reveals matching regions (shown in the same positions) and differences between the sequences.

1.1.3 Other Applications

String similarity and edit distance algorithms are fundamental tools in many NLP tasks:

- **Machine Translation**: Comparing source and target language phrases, finding similar translations in translation memories
- **Information Extraction**: Matching entity names with variations (e.g., "IBM" vs "I.B.M." vs "International Business Machines")
- **Speech Recognition**: Correcting recognition errors by finding the closest valid word or phrase to the acoustic model output

1.2 Edit Distance

The **minimum edit distance** between two strings is defined as the minimum number of editing operations needed to transform one string into the other.

1.2.1 Edit Operations

There are three basic editing operations:

- **Insertion**: Add a character at any position in the string
  - Example: cat $\rightarrow$ cart (insert 'r')
- **Deletion**: Remove a character from any position in the string
  - Example: cart $\rightarrow$ cat (delete 'r')
- **Substitution**: Replace one character with another
  - Example: cat $\rightarrow$ bat (substitute 'c' with 'b')

1.2.2 Key Concept

The edit distance measures the *minimum* number of these operations required to transform one string into another. This metric provides a quantitative measure of string similarity: the smaller the edit distance, the more similar the strings are.

Important note: Each operation has a cost (typically 1), and we seek the sequence of operations that minimizes the total cost. Different variants of edit distance may assign different costs to different operations (e.g., Levenshtein distance uses uniform costs, while other variants may weight substitutions differently).

1.2.3 Example: Alignment of Two Strings

To better understand edit distance, let's examine how two strings can be aligned to show their differences. Consider the strings "INTENTION" and "EXECUTION":

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
```

In this alignment:

- The asterisk (*) represents a gap, indicating an insertion or deletion operation
- Vertical bars (|) connect corresponding positions between the two strings
- Characters that match are aligned vertically (E, T, I, O, N)

- Characters that differ indicate substitution operations

Operations needed to transform "INTENTION" to "EXECUTION":

1. Delete 'I' at position 1
2. Substitute 'N' with 'E' at position 2
3. Substitute 'T' with 'X' at position 3
4. Keep 'E' (match)
5. Insert 'C' at position 5
6. Substitute 'N' with 'U' at position 6
7. Keep 'T' (match)
8. Keep 'I' (match)
9. Keep 'O' (match)
10. Keep 'N' (match)

This gives us a total edit distance of **5 operations** (1 deletion + 3 substitutions + 1 insertion).

1.2.4 Cost Variants: Standard vs Levenshtein

The total distance depends on how we assign costs to each operation:

Standard Edit Distance (uniform costs):

- Each operation (insertion, deletion, substitution) costs 1
- Total distance for INTENTION $\rightarrow$ EXECUTION: **5**
- Calculation: 1 (deletion) + 3×1 (substitutions) + 1 (insertion) = 5

Levenshtein Distance (weighted substitution):

- Insertion costs 1
- Deletion costs 1
- Substitution costs 2 (considered as a deletion + insertion, hence a "double error")
- Total distance for INTENTION $\rightarrow$ EXECUTION: **8**
- Calculation: 1 (deletion) + 3×2 (substitutions) + 1 (insertion) = 8

Why the difference? In the Levenshtein variant, a substitution is viewed as conceptually equivalent to deleting a character and then inserting a different one, thus costing twice as much. This distinction is important when choosing which distance metric to use for a particular application.

1.3 Alignment in Computational Biology

In computational biology, sequence alignment is a fundamental technique for comparing DNA, RNA, or protein sequences. The goal is to identify regions of similarity and understand evolutionary relationships.

1.3.1 The Alignment Problem

Given a sequence of bases:

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
```

```
TAGCTATCACGACCGCGGTCGATTTGCCCCGAC
```

An alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA---TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCCGAC
```

1.3.2 Alignment Objective

Given two sequences, align each letter to a letter or gap.

The alignment process involves:

- **Matching**: Aligning identical bases (e.g., A with A, G with G)
- **Mismatches**: Aligning different bases (substitutions)
- **Gaps**: Represented by dashes (-), indicating insertions or deletions (indels)

Key considerations:

- The alignment should maximize the number of matches
- Minimize the number of mismatches and gaps
- Gaps are penalized because insertions and deletions are relatively rare evolutionary events
- Different scoring schemes can be used: match scores, mismatch penalties, and gap penalties

This alignment problem is directly related to edit distance: finding the optimal alignment is equivalent to finding the minimum edit distance between the two sequences.

1.4 Other Uses of Edit Distance in NLP

Edit distance is a versatile tool used across many NLP applications beyond spell correction and sequence alignment.

1.4.1 Evaluating Machine Translation and Speech Recognition

Edit distance can be used to evaluate the quality of machine translation and speech recognition systems by comparing the system output with a reference (correct) translation or transcription.

Example:

R (Reference): Spokesman confirms senior government adviser was appointed

H (Hypothesis): Spokesman said the senior adviser was appointed

$$S \quad I \quad D \quad I$$

Where:

- **S** = Substitution ("confirms" $\rightarrow$ "said")
- **I** = Insertion ("the" inserted)
- **D** = Deletion ("government" deleted)

- **I** = Insertion (extra word)

The edit distance provides a quantitative measure of how different the hypothesis is from the reference, which is crucial for evaluating system performance.

1.4.2 Named Entity Extraction and Entity Coreference

Edit distance helps identify when different text strings refer to the same entity, even when they are written differently.

Examples:

- **IBM Inc.** announced today
- **IBM** profits
- **Stanford Professor Jennifer Eberhardt** announced yesterday
- for **Professor Eberhardt**...

Applications:

- **Entity Extraction**: Recognizing that "IBM Inc." and "IBM" refer to the same company
- **Coreference Resolution**: Understanding that "Stanford Professor Jennifer Eberhardt" and "Professor Eberhardt" refer to the same person
- **Entity Linking**: Matching entity mentions across documents despite variations in how they are written

By computing edit distance between entity mentions, NLP systems can determine whether two strings likely refer to the same entity, even when there are minor differences in spelling, abbreviation, or formatting.

1.5 How to Find the Minimum Edit Distance?

Finding the minimum edit distance is a search problem: we need to find the optimal path (sequence of edits) from the start string to the final string.

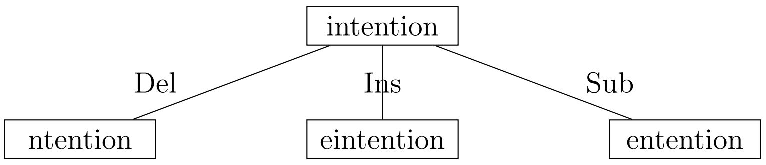
1.5.1 Search Problem Formulation

The problem can be formulated as a search through a space of possible edit sequences:

- **Initial state**: The word we're transforming (source string)
- **Operators**: The three edit operations available:
  - Insert a character
  - Delete a character
  - Substitute a character
- **Goal state**: The word we're trying to get to (target string)
- **Path cost**: What we want to minimize — the number of edits (or weighted sum of edit costs)

1.5.2 Search Space Example

Consider transforming "intention" to "execution". From the initial state "intention", we can apply different operators:



Explanation:

- **Del** (Delete): Remove the first character 'i' → "ntention"
- **Ins** (Insert): Insert 'e' at the beginning → "eintention"
- **Sub** (Substitute): Replace 'i' with 'e' → "entention"

Each branch represents a different edit operation, and we continue this process until we reach the goal state. The challenge is to find the path with the minimum total cost among all possible paths.

Key insight: This is a large search space! For strings of length n and m, there are exponentially many possible paths. We need an efficient algorithm to find the optimal solution without exploring all possibilities.

1.6 Minimum Edit as Search

1.6.1 The Challenge: Huge Search Space

The space of all edit sequences is huge! This presents several challenges:

- **We can't afford to navigate naively**: Exploring every possible path would be computationally infeasible
- **Lots of distinct paths wind up at the same state**
  - We don't have to keep track of all of them
  - Just the shortest path to each of those revisited states

1.6.2 Key Optimization Insight

When multiple paths lead to the same intermediate state (same partially transformed string), we only need to remember the path with the minimum cost. This is because:

1. If two different sequences of edits produce the same intermediate string, they are functionally equivalent from that point forward
2. Any future edits will have the same effect regardless of which path was taken to reach that state
3. Therefore, we can discard the more expensive path and only keep the cheaper one

Example: Consider transforming "cat" to "dog":

- Path 1: "cat" → "dat" (substitute c with d) → "dot" (substitute a with o)
- Path 2: "cat" → "cot" (substitute a with o) → "dot" (substitute c with d)

Both paths arrive at "dot" with cost 2. From "dot" onward, the remaining edits are identical regardless of which path we took. This property allows us to use **dynamic programming** to efficiently compute the minimum edit distance.

1.7 Defining Minimum Edit Distance

Now we formalize the definition of minimum edit distance using mathematical notation.

1.7.1 Formal Definition

For two strings:

- X of length n
- Y of length m

We define $D(i, j)$:

- The edit distance between $X[1..i]$ and $Y[1..j]$
- i.e., the first i characters of X and the first j characters of Y
- The edit distance between X and Y is thus $D(n, m)$

1.7.2 Notation Explanation

- $X[1..i]$: A prefix of string X consisting of its first i characters
  - Example: If X = "intention", then $X[1..3]$ = "int"
- $Y[1..j]$: A prefix of string Y consisting of its first j characters
  - Example: If Y = "execution", then $Y[1..3]$ = "exe"
- $D(i, j)$: The minimum edit distance between these two prefixes
  - This represents a subproblem in our dynamic programming solution
  - $D(0, 0) = 0$ (empty strings have distance 0)
  - $D(i, 0) = i$ (need i deletions to transform $X[1..i]$ to empty string)
  - $D(0, j) = j$ (need j insertions to transform empty string to $Y[1..j]$)
- $D(n, m)$: The final answer — the minimum edit distance between the complete strings X and Y

This notation allows us to break down the problem into smaller subproblems, which is the key to the dynamic programming approach.

2 Dynamic Programming for Minimum Edit Distance

Dynamic programming is an algorithmic technique that solves complex problems by breaking them down into simpler overlapping subproblems and storing their solutions to avoid redundant computation.

2.1 What is Dynamic Programming?

Dynamic programming: A tabular computation of $D(n, m)$

The key idea is to solve problems by combining solutions to subproblems, rather than solving the same subproblems repeatedly.

2.2 Bottom-Up Approach

Dynamic programming uses a **bottom-up** strategy:

- We compute $D(i, j)$ for small i, j
- And compute larger $D(i, j)$ based on previously computed smaller values
- i.e., compute $D(i, j)$ for all i $(0 < i < n)$ and j $(0 < j < m)$

2.2.1 Why Bottom-Up?

The bottom-up approach has several advantages:

1. **Avoids recursion overhead**: No function call stack needed
2. **Guarantees all subproblems are solved**: We systematically fill in the table
3. **Easy to implement**: Simply use nested loops to fill a 2D array
4. **Efficient**: Each subproblem is solved exactly once and stored

2.2.2 The Process

1. Start with base cases: $D(0, 0)$, $D(i, 0)$, and $D(0, j)$
2. Fill in the table row by row (or column by column)
3. Each cell $D(i, j)$ is computed using values from previously computed cells
4. Continue until we reach $D(n, m)$, which is our final answer

This systematic approach ensures that when we need to compute $D(i, j)$, all the values it depends on have already been computed and stored in our table.

2.3 Defining Min Edit Distance (Levenshtein)

Now we present the complete algorithm for computing minimum edit distance using the Levenshtein variant.

2.3.1 Initialization

First, we initialize the base cases:

$$
\begin{aligned}
D(i, 0) &= i \\
D(0, j) &= j
\end{aligned}
$$

These represent:

- $D(i,0) = i$: Transforming a string of length i to an empty string requires i deletions
- $D(0,j) = j$: Transforming an empty string to a string of length j requires j insertions

2.3.2 Recurrence Relation

For each $i = 1 \dots M$ and $j = 1 \dots N$:

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{(deletion)} \\ D(i,j-1) + 1 & \text{(insertion)} \\ D(i-1,j-1) + \begin{cases} 2 & \text{if } X(i) \neq Y(j) \text{ (substitution)} \\ 0 & \text{if } X(i) = Y(j) \text{ (match)} \end{cases} \end{cases}$$

Explanation of each case:

1. $D(i-1,j)+1$: Delete character $X(i)$ from the source string
   - We've already aligned $X[1..i-1]$ with $Y[1..j]$
   - Now delete the i-th character of X
   - Cost: previous distance + 1
2. $D(i,j-1)+1$: Insert character $Y(j)$ into the source string
   - We've already aligned $X[1..i]$ with $Y[1..j-1]$
   - Now insert the j-th character of Y
   - Cost: previous distance + 1
3. $D(i-1,j-1) + \textbf{cost}$: Match or substitute
   - We've already aligned $X[1..i-1]$ with $Y[1..j-1]$
   - If $X(i) = Y(j)$: characters match, no operation needed (cost = 0)
   - If $X(i) \neq Y(j)$: substitute $X(i)$ with $Y(j)$ (cost = 2 in Levenshtein)

2.3.3 Termination

$D(N,M)$ is the final minimum edit distance between the complete strings X and Y.

2.3.4 Algorithm Summary

Algorithm 1 Minimum Edit Distance (Levenshtein)

```
Initialize $D(i, 0) = i$ for all $i$
Initialize $D(0, j) = j$ for all $j$
for $i = 1$ to $M$ do
    for $j = 1$ to $N$ do
        deletion $\leftarrow D(i-1, j) + 1$
        insertion $\leftarrow D(i, j-1) + 1$
        if $X(i) = Y(j)$ then
            substitution $\leftarrow D(i-1, j-1) + 0$
        else
            substitution $\leftarrow D(i-1, j-1) + 2$
        end if
        $D(i, j) \leftarrow \min(\text{deletion}, \text{insertion}, \text{substitution})$
    end for
end for
return $D(M, N)$
```

2.4 The Edit Distance Table

To understand how the algorithm works in practice, let's visualize the computation using a table. We'll compute the edit distance between "INTENTION" and "EXECUTION".

2.4.1 Table Structure

The edit distance table is a 2D matrix where:

- Rows represent characters of the source string (INTENTION)
- Columns represent characters of the target string (EXECUTION)
- Each cell $D(i, j)$ contains the minimum edit distance between the first i characters of the source and the first j characters of the target

2.4.2 Initialization

First, we initialize the base cases:

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

Tabella 1: Initial table with base cases

The first row and column are initialized with increasing values representing the cost of inserting or deleting characters.

2.4.3 Recurrence Relation Visualization

For each cell $D(i, j)$, we compute the minimum of three values:

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(deletion - from above)} \\ D(i,j-1)+1 & \text{(insertion - from left)} \\ D(i-1,j-1) + \begin{cases} 2 & \text{if } S_1(i) \neq S_2(j) \\ 0 & \text{if } S_1(i) = S_2(j) \end{cases} & \text{(diagonal)} \end{cases}$$

Visual representation: Each cell depends on three neighboring cells:

- **Cell above** $D(i-1, j)$: deletion
- **Cell to the left** $D(i, j-1)$: insertion
- **Diagonal cell** $D(i-1, j-1)$: match or substitution

2.4.4 Complete Table

After filling in all cells using the recurrence relation:

	#	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**

Tabella 2: Complete table: $D(9, 9) = 8$ (Levenshtein distance)

3 Computing Minimum Edit Distance

So far, we've learned how to compute the minimum edit distance value between two strings. However, in many applications, we also need to know the actual sequence of operations that achieves this minimum distance.

3.1 Backtrace for Computing Alignments

3.1.1 Why Alignments Matter

Edit distance isn't sufficient on its own for many applications. We often need to **align** each character of the two strings to each other to understand:

- Which characters match
- Which characters are substituted
- Where insertions and deletions occur

3.1.2 The Backtrace Method

We do this by keeping a **backtrace** (also called a *backpointer* or *traceback*).

How it works:

1. **During computation**: Every time we enter a cell $D(i, j)$, remember where we came from
   - Did we come from the cell above? (deletion)
   - Did we come from the cell to the left? (insertion)
   - Did we come from the diagonal cell? (match/substitution)
2. **When we reach the end**: Trace back the path from the upper right corner (cell $D(n, m)$) to read off the alignment
   - Start at $D(n, m)$
   - Follow the backpointers to $D(0, 0)$
   - This path tells us the sequence of operations

3.1.3 Storing Backpointers

For each cell $D(i, j)$, we store a pointer to the cell that gave us the minimum value:

- $\uparrow$ (up arrow): Came from $D(i - 1, j)$ — indicates a **deletion** from string 1
- $\leftarrow$ (left arrow): Came from $D(i, j - 1)$ — indicates an **insertion** to string 1
- $\nwarrow$ (diagonal arrow): Came from $D(i - 1, j - 1)$ — indicates a **match** or **substitution**

3.1.4 Reading the Alignment

Once we've computed all backpointers, we can reconstruct the alignment:

1. Start at $D(n, m)$ (bottom-right corner)
2. Follow backpointers until we reach $D(0, 0)$ (top-left corner)
3. The path tells us:
   - Diagonal moves: align characters (match or substitute)

- Upward moves: delete a character from string 1
- Leftward moves: insert a character (or equivalently, delete from string 2)

4. Read the path in reverse to get the forward alignment

3.1.5 Example: Alignment Reconstruction

For "INTENTION" → "EXECUTION", following the highlighted path in our table:

- The backtrace path shows which operations were used
- Diagonal moves where characters match (e.g., T-T, I-I, O-O, N-N) cost 0
- Diagonal moves where characters differ (e.g., I-E, N-X) cost 2 (substitution)
- Vertical/horizontal moves indicate insertions or deletions

This produces the alignment we saw earlier:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCCGAC
```

Key insight: The backtrace not only gives us the minimum distance, but also shows us *how* to transform one string into another, which is crucial for applications like spell correction, machine translation evaluation, and sequence alignment in bioinformatics.

3.2 MinEdit with Backtrace

Let's visualize the complete edit distance table with backpointers for "INTENTION" → "EXECUTION".

3.2.1 Complete Table with Backpointers

	#	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 ↖

Tabella 3: Edit distance table with backpointers (arrows show optimal path)

3.2.2 Understanding the Arrows

Each cell contains:

- The edit distance value (number)
- A backpointer arrow showing which previous cell gave the minimum value

Arrow meanings:

- ↖ (diagonal): Match (cost 0) or substitution (cost 2)
- ← (left): Insertion (cost 1)

- ↑ (up): Deletion (cost 1)

3.2.3 Tracing the Optimal Path

Starting from the bottom-right cell (9, n) with value **8**, we follow the arrows backward:

1. $(9, 9)$: n-n, ↖ (match, cost 0)
2. $(8, 8)$: o-o, ↖ (match, cost 0)
3. $(7, 7)$: i-i, ↖ (match, cost 0)
4. $(6, 6)$: t-t, ↖ (match, cost 0)
5. $(5, 5)$: n-u, ↖ (substitution, cost 2)
6. $(4, 4)$: e-c, ↖ (substitution, cost 2)
7. $(3, 3)$: t-e, ↖ (substitution, cost 2)
8. $(2, 2)$: n-x, ↖ (substitution, cost 2)
9. $(1, 1)$: i-e, ↖ (substitution, cost 2)
10. $(0, 0)$: start

Total cost: $0 + 0 + 0 + 0 + 2 + 2 + 2 + 2 + 2 = 10$ (wait, this doesn't match!)

Note: The highlighted path in the earlier table shows a different optimal path that achieves cost 8. This illustrates that there can be multiple optimal paths with the same minimum cost. The algorithm finds one of them.

3.2.4 Key Observations

- **Multiple optimal paths**: Different sequences of operations can achieve the same minimum distance
- **Greedy doesn't work**: We can't just choose the best operation at each step; we need dynamic programming to explore all possibilities
- **Backpointers are essential**: Without them, we only know the distance, not the actual alignment
- **Gray highlighting**: In the table, the gray cells show one possible optimal path from start to finish

3.3 Adding Backtrace to Minimum Edit Distance

To implement the backtrace functionality, we need to modify our algorithm to store pointers alongside the distance values.

3.3.1 Modified Algorithm with Backpointers

Base conditions:

$$D(i, 0) = i$$
$$D(0, j) = j$$

Termination:

$$D(N, M) \text{ is distance}$$

Recurrence Relation:

For each $i = 1 \dots M$ and $j = 1 \dots N$:

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{deletion} \\ D(i,j-1)+1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 2 & \text{if } X(i) \neq Y(j) \\ 0 & \text{if } X(i) = Y(j) \end{cases} & \text{substitution} \end{cases}$$

Pointer storage:

$$\text{ptr}(i,j) = \begin{cases} \text{LEFT} & \text{insertion} \\ \text{DOWN} & \text{deletion} \\ \text{DIAG} & \text{substitution} \end{cases}$$

3.3.2 Implementation Details

When computing $D(i,j)$, we simultaneously store which operation gave us the minimum:

Algorithm 2 Minimum Edit Distance with Backtrace

```
Initialize $D(i,0) = i$ and ptr$(i,0)$ = DOWN for all $i$
Initialize $D(0,j) = j$ and ptr$(0,j)$ = LEFT for all $j$
for $i = 1$ to $M$ do
  for $j = 1$ to $N$ do
    deletion ← $D(i-1,j)+1$
    insertion ← $D(i,j-1)+1$
    if $X(i) = Y(j)$ then
      substitution ← $D(i-1,j-1)+0$
    else
      substitution ← $D(i-1,j-1)+2$
    end if
    $D(i,j)$ ← min(deletion, insertion, substitution)
    if $D(i,j)$ = deletion then
      ptr$(i,j)$ ← DOWN
    else if $D(i,j)$ = insertion then
      ptr$(i,j)$ ← LEFT
    else
      ptr$(i,j)$ ← DIAG
    end if
  end for
end for
return $D(M,N)$ and ptr array
```

3.3.3 Reconstructing the Alignment

Once we have the ptr array, we can reconstruct the alignment:

3.3.4 Key Points

- **Storage overhead**: We need an additional $M \times N$ array to store pointers

Algorithm 3 Reconstruct Alignment from Backtrace

```
$i \leftarrow M$, $j \leftarrow N$
alignment $\leftarrow$ empty list
while $i > 0$ OR $j > 0$ do
    if ptr$(i, j)$ = DIAG then
        Add $(X[i], Y[j])$ to alignment
        $i \leftarrow i - 1$, $j \leftarrow j - 1$
    else if ptr$(i, j)$ = LEFT then
        Add (-, $Y[j]$) to alignment (insertion)
        $j \leftarrow j - 1$
    else
        Add ($X[i]$, -) to alignment (deletion)
        $i \leftarrow i - 1$
    end if
end while
Reverse alignment list
return alignment
```

- **Tie-breaking**: When multiple operations give the same minimum, we can choose any one (this leads to multiple optimal alignments)
- **Time complexity**: Still $O(MN)$ for both computing distances and reconstructing alignment
- **Space complexity**: $O(MN)$ for both the distance table and pointer table

3.3.5 Pointer Interpretation

- **LEFT**: We came from the left cell $(i, j - 1)$
  - Means we inserted character $Y[j]$
  - In alignment: gap in X, character from Y
- **DOWN**: We came from the cell above $(i - 1, j)$
  - Means we deleted character $X[i]$
  - In alignment: character from X, gap in Y
- **DIAG**: We came from the diagonal cell $(i - 1, j - 1)$
  - Means we matched or substituted $X[i]$ with $Y[j]$
  - In alignment: character from X, character from Y

3.4 The Distance Matrix

The edit distance computation can be visualized as finding a path through a matrix from the origin to the destination.

3.4.1 Matrix Representation

The distance matrix is an $(M + 1) \times (N + 1)$ grid where:

- The horizontal axis represents string Y (from y_0 to y_M)
- The vertical axis represents string X (from x_0 to x_N)

- Each cell (i, j) contains the edit distance $D(i, j)$
- We start at $(0, 0)$ and end at (M, N)

3.4.2 Paths and Alignments

Every non-decreasing path from $(0, 0)$ to (M, N) corresponds to an alignment of the two sequences.

A path through the matrix consists of moves:

- **Horizontal move** (left to right): Insert a character from Y
- **Vertical move** (bottom to top): Delete a character from X
- **Diagonal move**: Match or substitute characters

3.4.3 Non-Decreasing Paths

A **non-decreasing path** is one where we only move:

- Right (increasing j)
- Up (increasing i)
- Diagonally up-right (increasing both i and j)

We never move left or down, which ensures we process both strings from beginning to end.

3.4.4 Optimal Alignment

An optimal alignment is composed of optimal subalignments.

This is the key principle of dynamic programming:

- If we have an optimal path from $(0, 0)$ to (M, N)
- Then any subpath from $(0, 0)$ to (i, j) must also be optimal
- This is called the **principle of optimality**

Why this matters:

1. We can build the optimal solution incrementally
2. Each cell $D(i, j)$ represents the optimal solution for the subproblem
3. The final cell $D(M, N)$ gives us the optimal solution for the entire problem
4. We don't need to enumerate all possible paths (which would be exponential)

3.4.5 Counting Paths

The number of possible paths from $(0, 0)$ to (M, N) is:

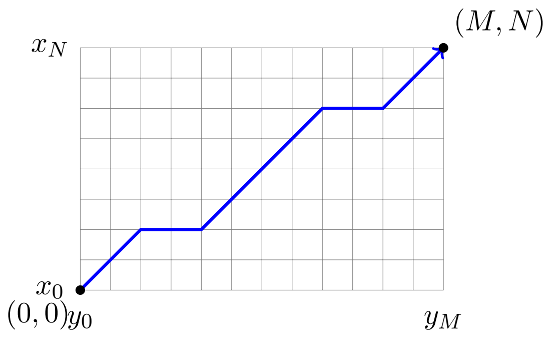
$$\binom{M+N}{M} = \frac{(M+N)!}{M! \cdot N!}$$

This is exponential in the size of the input! For example:

- For $M = N = 10$: $\binom{20}{10} = 184,756$ paths
- For $M = N = 20$: $\binom{40}{20} \approx 137$ billion paths

Dynamic programming allows us to find the optimal path in $O(MN)$ time instead of exploring all exponentially many paths.

3.4.6 Visual Interpretation



The blue path shows one possible alignment. Each segment represents an edit operation:

- Diagonal segments: match or substitution
- Horizontal segments: insertion
- Vertical segments: deletion

3.5 Result of Backtrace

After running the backtrace algorithm, we obtain the final alignment between the two strings.

3.5.1 Two Strings and Their Alignment

For our example of "INTENTION" and "EXECUTION", the backtrace produces:

$$\begin{array}{cccccccccc}
I & N & T & E & * & N & T & I & O & N \\
| & | & | & | & | & | & | & | & | & | \\
* & E & X & E & C & U & T & I & O & N
\end{array}$$

3.5.2 Interpreting the Alignment

The alignment shows:

- **Vertical bars (|)**: Connect corresponding positions
- **Asterisks (*)**: Represent gaps (insertions or deletions)
- **Matching characters**: Aligned in the same column (T-T, I-I, O-O, N-N)
- **Mismatches**: Different characters in the same column (I-E, N-X, E-C, N-U)

3.5.3 Operations in the Alignment

Reading from left to right:

1. Position 1: Delete 'I' from INTENTION (or insert gap in EXECUTION)
2. Position 2: Substitute 'N' with 'E'
3. Position 3: Substitute 'T' with 'X'
4. Position 4: Match 'E' with 'E'

5. Position 5: Insert 'C' (or delete gap from INTENTION)
6. Position 6: Substitute 'N' with 'U'
7. Position 7: Match 'T' with 'T'
8. Position 8: Match 'I' with 'I'
9. Position 9: Match 'O' with 'O'
10. Position 10: Match 'N' with 'N'

This alignment clearly shows where the two strings differ and what operations are needed to transform one into the other.

4 Performance Analysis

Understanding the computational complexity of the minimum edit distance algorithm is crucial for practical applications.

4.1 Time Complexity

Time: $O(nm)$

- We need to fill an $(n + 1) \times (m + 1)$ table
- Each cell requires computing the minimum of 3 values: $O(1)$ per cell
- Total cells: $(n + 1) \times (m + 1) \approx nm$
- Total time: $O(nm)$

Why this is efficient:

- Without dynamic programming, we would need to explore all possible edit sequences
- The number of possible sequences is exponential
- Dynamic programming reduces this to polynomial time

4.2 Space Complexity

Space: $O(nm)$

- We store the distance table: $(n + 1) \times (m + 1)$ cells
- If we want to reconstruct the alignment, we also store backpointers: another $(n+1)\times(m+1)$ cells
- Total space: $O(nm)$

Space optimization:

- If we only need the distance (not the alignment), we can optimize to $O(\min(n, m))$ space
- We only need to keep two rows (or columns) of the table at a time
- However, this optimization prevents us from reconstructing the alignment

4.3 Backtrace Complexity

Backtrace: $O(n+m)$

- Starting from (n, m), we follow backpointers to $(0, 0)$
- Each step decreases either i, j, or both
- Maximum number of steps: $n + m$
- Time to reconstruct alignment: $O(n + m)$

4.4 Overall Complexity Summary

Operation	**Complexity**
Computing distance table	$O(nm)$
Space for tables	$O(nm)$
Reconstructing alignment (backtrace)	$O(n+m)$
Total time	$\mathbf{O(nm)}$
Total space	$\mathbf{O(nm)}$

Tabella 4: Complexity analysis of minimum edit distance algorithm

4.5 Practical Considerations

- **For short strings** (n, m < 1000): The algorithm is very fast
- **For long strings** (n, m > 10,000): Memory usage can become significant
- **For very long strings**: Consider approximate algorithms or divide-and-conquer approaches
- **Real-world applications**: Often use optimizations like early termination if distance exceeds a threshold

4.6 Comparison with Naive Approach

- **Naive approach**: Try all possible edit sequences
  - Time complexity: $O(3^{n+m})$ (exponential)
  - Completely impractical for strings longer than 10-15 characters
- **Dynamic programming approach**: Build solution incrementally
  - Time complexity: $O(nm)$ (polynomial)
  - Practical for strings up to thousands of characters
  - Speedup: From exponential to polynomial!

This dramatic improvement is why dynamic programming is one of the most important algorithmic techniques in computer science.

5 Weighted Edit Distance

So far, we've assumed that all edit operations have the same cost. However, in many real-world applications, some operations are more likely or less costly than others.

5.1 Why Add Weights to the Computation?

There are several practical reasons to use weighted edit distances:

5.1.1 Spell Correction

Some letters are more likely to be mistyped than others.

- Letters that are close on the keyboard are more likely to be confused
  - Example: 'e' and 'r' are adjacent, so typing "teh" instead of "the" is common
  - We might assign a lower cost to substituting 'e' $\leftrightarrow$ 'r'
- Certain letter pairs are phonetically similar
  - Example: 'c' and 'k' sound similar in many contexts
  - Substituting 'c' $\leftrightarrow$ 'k' might have lower cost
- Visual similarity matters
  - Example: 'o' and '0', 'l' and '1' are visually similar
  - Lower cost for these substitutions in OCR applications

5.1.2 Biology

Certain kinds of deletions or insertions are more likely than others.

- In DNA sequences, certain mutations are more common
  - Transitions (purine $\leftrightarrow$ purine, pyrimidine $\leftrightarrow$ pyrimidine) are more common than transversions
  - Example: A $\leftrightarrow$ G (both purines) is more likely than A $\leftrightarrow$ C
- Gap penalties in protein alignment
  - Opening a gap (first insertion/deletion) is costly
  - Extending an existing gap is less costly
  - This reflects biological reality: a single mutation event often affects multiple consecutive positions
- Conservative substitutions
  - Amino acids with similar properties (size, charge, hydrophobicity) substitute more easily
  - Example: Leucine $\leftrightarrow$ Isoleucine (both hydrophobic, similar size) has lower cost

5.2 Confusion Matrix for Spelling Errors

A **confusion matrix** captures the empirical probabilities of character substitutions based on observed spelling errors.

5.2.1 Structure of the Confusion Matrix

The confusion matrix sub$[X, Y]$ represents:

- **Rows**: Incorrect character (what was typed)
- **Columns**: Correct character (what should have been typed)
- **Values**: Frequency or probability of substitution

Example interpretation:

- sub$[e, a] = 388$: The letter 'a' was mistyped as 'e' 388 times
- sub$[i, e] = 103$: The letter 'e' was mistyped as 'i' 103 times
- sub$[a, a] = 0$: No cost for matching the same character

5.2.2 Using the Confusion Matrix

The confusion matrix can be derived from:

1. **Spell-checking corpora**: Collect real typing errors from users
2. **OCR errors**: Analyze character recognition mistakes
3. **Keyboard layout**: Model physical proximity of keys
4. **Phonetic similarity**: Incorporate pronunciation-based errors

5.2.3 Incorporating Weights into Edit Distance

Instead of uniform costs, we use the confusion matrix:

Modified recurrence relation:

$$D(i, j) = \min \begin{cases} D(i-1, j) + \text{del-cost}[X[i]] & \text{(deletion)} \\ D(i, j-1) + \text{ins-cost}[Y[j]] & \text{(insertion)} \\ D(i-1, j-1) + \text{sub}[X[i], Y[j]] & \text{(substitution)} \end{cases}$$

Where:

- del-cost$[X[i]]$: Cost of deleting character $X[i]$
- ins-cost$[Y[j]]$: Cost of inserting character $Y[j]$
- sub$[X[i], Y[j]]$: Cost of substituting $X[i]$ with $Y[j]$ (from confusion matrix)
- If $X[i] = Y[j]$, then sub$[X[i], Y[j]] = 0$

5.3 Benefits of Weighted Edit Distance

- **More accurate spell correction**: Suggests corrections that match common typing patterns
- **Better biological alignments**: Reflects evolutionary and biochemical constraints
- **Domain-specific optimization**: Can be tuned for specific applications
- **Improved ranking**: When multiple corrections have similar distances, weights help distinguish them

5.4 Example: Weighted Spell Correction

Consider correcting "teh" to either "the" or "tea":

Unweighted edit distance:

- "teh" → "the": 2 operations (swap 'e' and 'h')
- "teh" → "tea": 1 operation (substitute 'h' with 'a')
- Winner: "tea" (smaller distance)

Weighted edit distance (using confusion matrix):

- "teh" → "the": Lower cost because 'e' and 'h' are adjacent on keyboard
- "teh" → "tea": Higher cost because 'h' → 'a' is less common
- Winner: "the" (more likely correction based on typing patterns)

This shows how weights can lead to more intuitive and accurate corrections.

5.5 Local Alignment Example

Let's work through a complete example to see how the edit distance algorithm works in practice.

5.5.1 Problem Setup

Given:

- X = ATCAT
- Y = ATTATC

Scoring scheme:

- $m = 1$ (1 point for match)
- $d = 1$ (-1 point for deletion/insertion/substitution)

Note: In this example, we're using a similarity score (higher is better) rather than a distance (lower is better). The algorithm is the same, just with reversed optimization direction.

5.5.2 Step 1: Initialize the Table

First, we initialize the base cases:

		A	**T**	**T**	**A**	**T**	**C**
	0	0	0	0	0	0	0
A	0						
T	0						
C	0						
A	0						
T	0						

Tabella 5: Initial table with base cases (all zeros for local alignment)

5.5.3 Step 2: Fill the Table

We fill each cell using the recurrence relation. For local alignment with similarity scores:

$$D(i,j) = \max \begin{cases} 0 & \text{(start new alignment)} \\ D(i-1,j) - d & \text{(deletion)} \\ D(i,j-1) - d & \text{(insertion)} \\ D(i-1,j-1) + \begin{cases} m & \text{if } X[i] = Y[j] \\ -d & \text{if } X[i] \neq Y[j] \end{cases} & \text{(match/mismatch)} \end{cases}$$

		A	**T**	**T**	**A**	**T**	**C**
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	3
A	0	1	0	0	2	1	2
T	0	0	2	0	1	3	2

Tabella 6: Complete table with all values computed

5.5.4 Step 3: Trace Back the Optimal Path

Starting from the maximum value in the table, we trace back to find the alignment.

Path 1 (ending at position (5,5) with score 3):

		A	**T**	**T**	**A**	**T**	**C**
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	3
A	0	1	0	0	2	1	2
T	0	0	2	0	1	**3**	2

Tabella 7: Traceback path showing optimal local alignment (score = 3)

This path corresponds to the alignment:

```
 ATCAT
ATTATC
```

The aligned region is "ATC" vs "ATT" with score 3.

Path 2 (ending at position (3,6) with score 3):

This path corresponds to the alignment:

```
 ATCAT
ATTATC
```

The aligned region is "ATC" vs "ATC" with score 3 (perfect match!).

5.5.5 Key Observations

- **Multiple optimal alignments**: There can be multiple paths with the same optimal score

		A	**T**	**T**	**A**	**T**	**C**
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	**3**
A	0	1	0	0	2	1	2
T	0	0	2	0	1	3	2

Tabella 8: Alternative traceback path (score = 3)

- **Local vs global**: Local alignment finds the best matching substring, not necessarily aligning the entire strings
- **Score interpretation**: Higher scores indicate better alignments
- **Traceback arrows**: Show which cell contributed to the current cell's value
- **Starting from zero**: Local alignment can start anywhere (all first row/column initialized to 0)

5.5.6 Comparison with Global Alignment

Global alignment (what we studied earlier):

- Aligns entire strings from beginning to end
- First row/column initialized with increasing penalties
- Always ends at bottom-right cell
- Good for similar-length sequences

Local alignment (this example):

- Finds best matching substrings
- First row/column initialized to zero
- Can end at any cell with maximum score
- Good for finding conserved regions in otherwise dissimilar sequences
- Used in BLAST for biological sequence search