Algorithms, Design & Analysis

Lecture 21: Edit Distance Problem And Longest Common Subsequence Problem

Muhammad Hassan & Muhammad Fahad

Information Technology University

April 15, 2025

About Your Fellows

- Hi there! We are **Muhammad Hassan** and **Muhammad Fahad**.
- We are Associate Students at ITU.

Recap

Dynamic Programming (DP)

- A method for solving complex problems by breaking them down into simpler subproblems.
- Solves each subproblem only once and stores the result (memoization).
- Used when the problem has optimal substructure and overlapping subproblems.

Floyd-Warshall Algorithm

- Finds shortest paths between all pairs of vertices.
- Handles negative edge weights (not negative cycles).
- Update rule: `d[i][j] = min(d[i][j], d[i][k] + d[k][j])`
- Works step-by-step by allowing increasing number of intermediate vertices.

Key Concepts

- Dynamic Programming optimizes overlapping subproblems.
- Absence of ∞ does not always mean shortest path is found.
- Negative cycle exists if $d[i][i] < 0$ after Floyd-Warshall.

Problem Statement

Given: Two strings, s_1 and s_2.

Goal: Transform s_1 into s_2 using the minimum number of operations.

Allowed Operations:

- **Insert:** Add a character at any position.
- **Delete:** Remove any character.
- **Switch:** Replace one character with another.

Example:

- $s_1 = \texttt{GTGTACC}$ (length $n = 7$)
- $s_2 = \texttt{CCGAT}$ (length $m = 5$)

Objective: Determine the minimum number of operations required to make $s_1 = s_2$.

Manual Solution - Part 1/2

Step	Alignment	Operation
Initial	s1: G T G T A C C s2: C C G A T	Start with original strings
1	s1: G T G T A C C s2: C C G A T C T G T A C C ⇒ C C G A T	Compare G (s1) vs C (s2) Substitute G→C (+1)
2	s1: C T G T A C C s2: C C G A T C C G T A C C ⇒ C C G A T	Compare T (s1) vs C (s2) Substitute T→C (+1)
3	s1: C C G T A C C s2: C C G A T	Match G (no cost)
4	s1: C C G T A C C s2: C C G A T C C G A A C C ⇒ C C G A T	Compare T vs A Substitute T→A (+1)

Progress after Step 4

- Current s1: C C G A A C C
- Target s2: C C G A T
- Operations: 3 substitutions
- Total cost: 3

Manual Solution - Part 2/2

Step	Alignment	Operation
5	s1: C C G A A C C s2: C C G A T C C G A T C C ⇒ C C G A T	Compare A vs T Substitute A→T (+1)
6	s1: C C G A T C C s2: C C G A T ϵ C C G A T C ⇒ C C G A T	Extra C in s1 Delete C (+1)
7	s1: C C G A T C s2: C C G A T ϵ C C G A T ⇒ C C G A T	Extra C in s1 Delete C (+1)

Final Cost Summary

- Substitutions: 4 (Steps 1,2,4,5)
- Deletions: 2 (Steps 6,7)
- **Total Cost: 6**

Recursive solution Step 1: Base Case Identification

Handling Strings of Length ≤ 1

Core Principle:

- Solve smallest possible subproblems first
- Only consider strings where $|s_1| \leq 1$ and $|s_2| \leq 1$

Base Cases:

- **Case 1: Both Empty**
  - $s_1 = "", s_2 = ""$
  - Operations: None needed
  - $\boxed{\text{Distance} = 0}$
- **Case 2: Empty to Single Character**
  - $s_1 = "", s_2 = "G" \ (|s_2| = 1)$
  - Operation: Insert "G"
  - $\boxed{\text{Distance} = 1}$
- **Case 3: Single Character to Empty**
  - $s_1 = "C" \ (|s_1| = 1), s_2 = ""$
  - Operation: Delete "C"
  - $\boxed{\text{Distance} = 1}$

Recursive Solution

Case 4: Both Strings Non-Empty (Length $\leq$ 1)

Case 4: Single Character to Single Character

- $s_1 = "T"$, $s_2 = "G"$ (both length 1)
- Possible operations:
  - Substitute "T" $\rightarrow$ "G"
  - Or delete "T" and insert "G"
- $\boxed{\text{Minimum Distance} = 1}$

Recursive Solution – Step 2

Comparing First Characters

Step 2: Comparing First Characters of s_1 and s_2

- If the first characters match (i.e., $s_1[0] = s_2[0]$):
  - Return:

$$0 + \text{edit\_distance}(s_1[1 \ldots n-1],\ s_2[1 \ldots m-1])$$

- If the first characters do not match (i.e., $s_1[0] \neq s_2[0]$), consider three operations:
  - **Insertion:**

$$1 + \text{edit\_distance}(s_1,\ s_2[1 \ldots m-1])$$

  - **Deletion:**

$$1 + \text{edit\_distance}(s_1[1 \ldots n-1],\ s_2)$$

  - **Substitution (Swap):**

$$1 + \text{edit\_distance}(s_1[1 \ldots n-1],\ s_2[1 \ldots m-1])$$

- Finally, return the minimum of these three values.

Example of Recursive Solution (Part 1)

Given:

- $s_1 = \texttt{GTGTACC}$ (length $= 7$)
- $s_2 = \texttt{CCGAT}$ (length $= 5$)

Step 1: Compare first characters

- $s_1[0] = \text{G}$ and $s_2[0] = \text{C}$.
- They do not match, so we need to consider the three operations:

Step 2: Consider Three Possibilities

- **Insertion:**
  - Insert a character into s_1. The problem reduces to finding the edit distance between s_1 and the string $s_2[1 \dots 4] = \texttt{CGAT}$.
  - We calculate: $1 + \text{edit distance}(s_1, s_2[1 \dots 4])$.

Example of Recursive Solution (Part 2)

Step 2 (continued): Consider Three Possibilities

- **Deletion:**
  - Remove the first character from s_1. The problem reduces to finding the edit distance between $s_1[1\dots6] = \texttt{TGTACC}$ and s_2.
  - We calculate: $1 + \text{edit distance}(s_1[1\dots6], s_2)$.

- **Switch:**
  - Consider swapping the first characters of both strings: $s_1[0] = \text{G}$ and $s_2[0] = \text{C}$.
  - They are not equal, so we perform a swap operation.
  - We calculate: $1 + \text{edit distance}(s_1[1\dots6], s_2[1\dots4])$.

Dynamic Programming Solution

Memoization Approach (Part 1)

Key Idea:

Dynamic programming stores the results of sub-problems in a table (`memo`) and reuses them to avoid redundant calculations.

What does `memo[i][j]` represent?

- Since we have two strings s_1 and s_2, the memo should be a 2D table.
- `memo[i][j]` represents the edit distance between the suffixes:

$$s_1[i:] \text{ and } s_2[j:]$$

- That is, how much it costs to convert the substring starting at index i in s_1 to the substring starting at j in s_2.

Dynamic Programming Solution

Memoization Approach (Part 2)

Base Case Definitions:

- `memo[n][m]` $= 0$, meaning both substrings are empty — no operations required.
- If $i = n$, then `memo[i][j]` $= m - j$ (insert all remaining characters of s_2).
- If $j = m$, then `memo[i][j]` $= n - i$ (delete all remaining characters of s_1).

Visual Interpretation:

- The last row and last column of the DP table are filled directly based on the number of characters left in one string when the other is exhausted.

Dynamic Programming Solution: Recursive Relation

Recursive Relation:

- For `memo[1][1]`, if $s_1[0] = s_2[0]$, then no operation is needed:

$$\texttt{memo[1][1]} = \texttt{memo[0][0]} \quad \text{(no operation)}$$

- If $s_1[0] \neq s_2[0]$, then `memo[1][1]` is calculated as the minimum of three possible operations:
  - `1 + memo[1][0]` (insert)
  - `1 + memo[0][1]` (delete)
  - `memo[0][0] + 1` (substitute)
- Combined relation:

$$\texttt{memo[1][1]} = \min \begin{cases} 1 + \texttt{memo[1][0]} & \text{(insert)} \\ 1 + \texttt{memo[0][1]} & \text{(delete)} \\ \texttt{memo[0][0]} + \begin{cases} 0 & \text{if } s_1[0] = s_2[0] \\ 1 & \text{if } s_1[0] \neq s_2[0] \end{cases} & \text{(switch)} \end{cases}$$

Dynamic Programming Solution:

- **Time Complexity:** The time complexity of the Edit Distance algorithm is $O(n \times m)$, where n is the length of string 1 and m is the length of string 2. This is because we need to fill a DP table of size $n \times m$.

Pseudocode

```
• for i = 1 to n
    for j = 1 to m
      if (s1[i-1] == s2[j-1])
        memo[i][j] = memo[i-1][j-1]
      else
        memo[i][j] =
          min(
            1 + memo[i-1][j]     (insert),
            1 + memo[i][j-1]     (delete),
            1 + memo[i-1][j-1]    (switch)
          )
```

Edit Distance: Dynamic Programming Table

Key Idea:

- Build a 2D table where cell (i, j) contains the edit distance between `String1[0..i-1]` and `String2[0..j-1]`.
- Initialize boundaries (comparisons with empty strings).
- Fill the table using the recurrence relation:

$$dp[i][j] = \begin{cases} \min \begin{cases} dp[i-1][j] + 1 & \text{(deletion)} \\ dp[i][j-1] + 1 & \text{(insertion)} \\ dp[i-1][j-1] + \begin{cases} 0 & \text{if } s_1[i-1] = s_2[j-1] \\ 1 & \text{if } s_1[i-1] \neq s_2[j-1] \end{cases} & \text{(switch)} \end{cases} \end{cases}$$

Next: We'll visualize the complete DP table for:

- String1: `GTGTACC` (length 7)
- String2: `CCGAT` (length 5)

Edit Distance DP Table Structure

Strings:

- String1 (row): `G T G T A C C empty` (length 7)
- String2 (column): `C C G A T empty` (length 5)

	G	T	G	T	A	C	C	empty
C								
C								
G								
A								
T								
empty								

- Rows represent String2 + empty string (vertical)
- Columns represent String1 + empty string (horizontal)
- Next: We'll fill this table step-by-step

How We Fill the DP Table

Key Rules:

- **Base Cases**:
  - Last row: `memo[i][m] = n - i` (cost to delete remaining in String1)
  - Last column: `memo[n][j] = m - j` (cost to insert remaining from String2)
- **For Other Cells**:
  - If characters match: `memo[i][j] = memo[i+1][j+1]` (diagonal value)
  - If mismatch: Take minimum of these 3 operations:
    - `1 + memo[i][j+1]` (delete from String1)
    - `1 + memo[i+1][j]` (insert from String2)
    - `1 + memo[i+1][j+1]` (switch)

Filling Direction:

- Start from `memo[n-1][m-1]` (bottom-right of non-base cells)
- Fill **backwards** to `memo[0][0]` (top-left)

Boundary Conditions: Last Column

Converting String2 to Empty String

Initialization Logic (Last Column):

- We are converting the remaining part of s_2 to an empty string.
- This requires deleting all characters from $s_2[i]$ to $s_2[m-1]$.

Logic:

-
  ```
  for i from m to 0:
      dp[i][n] = m - i
  ```
  (Delete all remaining characters)

	G	T	G	T	A	C	C	empty
C								**5**
C								**4**
G								**3**
A								**2**
T								**1**
empty								**0**

- Each cell shows how many deletions are needed to reach an empty string.
- **Why corner value is 0:**
  If both strings are empty (bottom-right), no operations are needed.

Boundary Conditions: Last Row

Converting Empty String to String1

Initialization Logic (Last Row):

- We are converting an empty string to the remaining part of s_1.
- This requires inserting all characters from $s_1[j]$ to $s_1[n-1]$.

Logic:

-
  ```
  for j from n to 0:
    dp[m][j] = n - j
  ```
  (Insert all remaining characters)

	G	T	G	T	A	C	C	empty
C								**5**
C								**4**
G								**3**
A								**2**
T								**1**
empty	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Explanation:

- `dp[m][j]` = length of `s1[j..n-1]` = `n - j`
- Each cell shows how many insertions are needed to construct string1 from an empty string.

Filling the DP Table

Computing dp[4][6] (T vs C)

Cell Being Filled: Matching 'T' (from s2) with 'C' (from s1)

	G	T	G	T	A	C	C	empty
C								5
C								4
G								3
A								2
T							X	1
empty	7	6	5	4	3	2	1	0

Next: We'll compute the cost for insertion, deletion, and switch to fill cell X.

Filling the DP Table

Computing dp[4][6] (T vs C) – Cost Explanation

Cost Calculation:

- Insertion Cost: $dp[4][7] + 1 = 1 + 1 = 2$
  - Insert 'C' then convert 'T' to empty
- Deletion Cost: $dp[5][6] + 1 = 1 + 1 = 2$
  - Delete 'T' then convert empty to 'C'
- Switch Cost: $dp[5][7] + \begin{cases} 0 & \text{if } \texttt{T} = \texttt{C} \\ 1 & \text{if } \texttt{T} \neq \texttt{C} \end{cases} = 0 + 1 = 1$
  - Switch 'T' to 'C' (cost 1 since characters differ)

Result:

- Minimum cost is 1 (switch operation)
- $X = 1$ because switching 'T' to 'C' is optimal

Filling the Column: Matching C vs C

For Matching Characters

Target Cell: dp[1][6] (Second 'C' vs last 'C')

Column Progress So Far:

	G	T	G	T	A	C	C	empty
C								5
C							X	4
G							3	3
A							2	2
T							1	1
empty	7	6	5	4	3	2	1	0

Next: We'll calculate insertion, deletion, and switch cost for cell X.

Filling the Column: Matching C vs C

Cost Calculation for dp[1][6]

Calculation for dp[1][6] (C vs C):

- Insertion: $\text{dp}[1][7] + 1 = 4 + 1 = 5$
- Deletion: $\text{dp}[2][6] + 1 = 3 + 1 = 4$
- Switch: $\text{dp}[2][7] + \begin{cases} 0 & (\text{since } C = C) \\ 1 & (\text{if unequal}) \end{cases} = 3 + 0 = 3$

Result:

- Minimum cost is 3
- So, $X = 3$ using switch operation (no cost as characters match)

When Substitution Isn't Optimal

Example: Cell dp[4][2] (A vs T)

Neighbor Values in Backwards DP Table

	G	T	G	T	A	C	C	empty
C				5	4	3	4	5
C				4	4	3	3	4
G				3	3	3	3	3
A				3	2	2	2	2
T			X	3	3	2	1	1
empty	7	6	5	4	3	2	1	0

Next: We'll compute Insertion, Deletion, and Substitution costs to find the optimal operation.

When Substitution Isn't Optimal

Cost Calculation for dp[4][2] (A vs T)

Cost Calculations

- Deletion: $dp[i+1][j] + 1 = dp[5][2] + 1 = 5 + 1 = \mathbf{6}$
- Insertion: $dp[i][j+1] + 1 = dp[4][3] + 1 = 3 + 1 = \mathbf{4}$
- Substitution: $dp[i+1][j+1] + 1 \quad (\text{since A} \neq \text{T}) = dp[5][3] + 1 = 4 + 1 = \mathbf{5}$

Optimal Choice

$$\min(6, 4, 5) = \boxed{4} \quad (\text{Insertion})$$

- Insertion is better than substitution $4 < 5$
- Edit sequence: Insert 'G' first (cost 1), then match A (total cost 4)

Edit Distance DP Table

- **This is the final table after filling each box using the same recursive rule**

Strings:

- String1 : `G T G T A C C empty` (length 7)
- String2 : `C C G A T empty` (length 5)

	G	T	G	T	A	C	C	empty
C	5	5	5	5	4	3	4	5
C	5	4	4	4	4	3	3	4
G	4	4	3	3	3	3	3	3
A	6	5	4	3	2	2	2	2
T	6	5	4	3	3	2	1	1
empty	7	6	5	4	3	2	1	0

- Numbers represent minimum operations needed.
- `empty` means zero-length string.
- `Memo[0][0]` contains the final edit distance

Clarification: Table Orientation

Question :

"In most tables, the top-left is considered as [0][0], but in our DP table, it seems like the bottom-right is being treated as [0][0]. Why is that?"

Answer:

- In many textbooks and implementations, the top-left corner of a table is indexed as `[0][0]`.
- However, in some dynamic programming visualizations (especially when dealing with string operations), the table is drawn such that the bottom-right corner represents `[0][0]` for easier tracking of suffixes.
- Both approaches are valid — it just depends on how the algorithm is structured and visualized.

Edit Distance DP Algorithm

Top-Left Corner as [0] [0]

Function Name: `edistance(S1, S2)`

Pseudocode

```
Function edistance(S1, S2):
    For i = n-1 to 0:
        For j = m-1 to 0:
            Memo[i][j] = min(
                1 + Memo[i][j+1],         # Insertion
                1 + Memo[i+1][j],         # Deletion
                Memo[i+1][j+1] +
                    (1 if S1[i] != S2[j] else 0)  # Switch
            )
```

Understanding the Algorithm

Edit Distance DP Table Filling Strategy

- This algorithm fills the DP table **from bottom-right to top-left**.
- It assumes the **top-left cell is** `[0][0]`, following standard array notation.
- Each cell `Memo[i][j]` represents the edit distance from suffix `S1[i:]` to `S2[j:]`.
- The cost of **insertion**, **deletion**, and **substitution (switch)** is calculated at each step.
- Substitution has zero cost when characters match, and 1 when they don't.

Note: You can trace back from `[0][0]` to reconstruct the optimal edit sequence.

Longest Common Subsequence (LCS)

Definitions:

- **String**: A sequence of characters (e.g., `CTGA`)
- **Substring**: Contiguous part of a string (e.g., {`TG`}, {`GA`})
- **Subsequence**: Characters in order, but not necessarily contiguous (e.g., {`T`, `G`, `T`})
- **Subset**: Any selection of characters without regard to order or position (e.g., {`A`, `C`, `G`})

Examples:

- For string `CTGAAGT`:
  - Substring: {`GA`}, {`AAG`}
  - Subsequence: {`C`, `G`, `T`}, {`T`, `A`, `T`}
  - Subset: {`A`, `C`, `G`} (unordered selection)

Longest Common Subsequence (LCS) Problem

Problem Statement:

- Given two strings `S1` and `S2`, find the **length of the longest subsequence** common to both strings.

Inputs:

- Two strings `S1` and `S2`.

Output:

- The length of the longest subsequence that is common between the two strings.

LCS Recursive Solution - Part 1

Recursive Approach:

Base Case

```
If either string is empty:
    Return 0
```

Recursive Case

```
If S1[0] == S2[0]:
    Return 1 + LCS(S1[1..n], S2[1..m])
Else:
    Return max(LCS(S1[1..n], S2),
               LCS(S1, S2[1..m]))
```

LCS Recursive Solution - Part 2

Explanation:

- **Base Case:** If either of the strings is empty, return 0 because an empty string has no subsequence.
- **Recursive Case:** If the first characters of both strings match, we include that character and move to the next characters of both strings. If they do not match, we compute the LCS for two cases: excluding the character from either string.

LCS Dynamic Programming

Function: LCS

Initialization

```
LCS(n, m):
    memo[n][m] = 0
    memo[n][i] = 0
    memo[j][m] = 0
```

DP Fill

```
For i = n-1 to 0:
    For j = m-1 to 0:
        memo[i][j] = max(
            (1 + memo[i+1][j+1]) -> if S1[i]==S2[j],
            memo[i+1][j],
            memo[i][j+1]
        )
```

LCS Dynamic Programming - Explanation

Explanation:

- **Initialization:** We start by setting the boundary conditions. The last row and column are initialized to 0, as the LCS with an empty string has a length of 0.
- **DP Fill:** We fill the DP table by iterating backwards through the strings. For each pair of characters, we check:
  - If the characters match ($S1[i] == S2[j]$), we add 1 to the result from the diagonally next cell (memo[i+1][j+1]).
  - Otherwise, we take the maximum of either excluding the character from string1 or string2 (memo[i+1][j] or memo[i][j+1]).
- **Result:** The value at memo[0][0] contains the length of the longest common subsequence.