CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Dynamic Programming: Edit Distance

Instructor: Xiao Liang

Department of Computer Science and Engineering
The Chinese University of Hong Kong

The Edit Distance Problem

Goal: Given two strings

$$A = a_1 a_2 \cdots a_m \quad \text{and} \quad B = b_1 b_2 \cdots b_n,$$

compute the minimum number of **operations** to convert A into B.

Allowed **operations:**

- Insert a character
- Delete a character
- Substitute one character for another

Edit Distance: The minimum number of such operations.

Example

Input: $A = \texttt{kitten}$, $B = \texttt{sitting}$

1. $\texttt{kitten} \rightarrow \texttt{sitten}$ (substitute 'k' with 's')
2. $\texttt{sitten} \rightarrow \texttt{sittin}$ (substitute 'e' with 'i')
3. $\texttt{sittin} \rightarrow \texttt{sitting}$ (insert 'g')

Edit Distance = 3

Why Study Edit Distance?

Motivation:

- Measures similarity between two strings.
- Fundamental in understanding string algorithms.
- Real-world relevance:
  - Spell checking and autocorrect
  - Code versioning systems (e.g., Git)
  - DNA sequence comparison
  - File comparison and diff tools
  - Plagiarism detection
- Theoretically interesting: combines recursion, optimal substructure, and overlapping subproblems.

A Dynamic Programming Algorithm for the Edit Distance Problem

Subproblems! Subproblems! Subproblems!

The core trick of dynamic programming is to break the problem into subproblems that can be solved in a nice order that later subproblems can reuse the solutions of earlier subproblems.

With that in mind, when solving a problem by dynamic programming, the most crucial question is:

> What are the proper subproblems?

Unfortunately, this step requires both experience and creativity. There is no universal method that guarantees you will always find the appropriate subproblems.

For our problem in hand: what are the proper subproblems for **Edit Distance**?

Subproblems for Edit Distance?

Given

$$A = a_1 a_2 \cdots a_{m-1} a_m \quad \text{and} \quad B = b_1 b_2 \cdots b_{n-1} b_n.$$

The difficulty of this problem seems to be that we don't know what is the best way to "align" these two strings to maximize their overlap (so that it minimize the number of edit operations).

Recall our example $A = kitten$ and $B = sitting$. Here are some possible alignments (there could be more)

A =		k	i	t	t	e	n	
B =	s	i	t	t		i	n	g

Table: another alignment (bad)

A =	k	i	t	t	e	n	
B =	s	i	t	t	i	n	g

Table: one alignment (good)

It's inefficient (i.e., exponential-time) to try all possible alignments to find the best one.

The previous discussion motivates us to examine the endpoints of the alignment in both strings. This is arguably the simplest part of the alignment and, hopefully, it can reveal important structural insights about the problem.

Let us isolate the last char as $A = \alpha a_m$ and $B = \beta b_n$, where

$$A = \underbrace{a_1 a_2 \cdots a_{m-1}}_{\text{prefix } \alpha} a_m \quad \text{and} \quad B = \underbrace{b_1 b_2 \cdots b_{n-1}}_{\text{prefix } \beta} b_n.$$

If we classify them using the last char, there are only three possible alignments:

A =	$\alpha\ a_m$	
B =	β	b_n

Table: case 1

A =	α	a_m
B =	$\beta\ b_n$	

Table: case 2

A =	α	a_m
B =	β	b_n

Table: case 3

If we classify them using the last char, there are only three possible alignments:

A =	$\alpha\ a_m$	
B =	β	b_n

Table: case 1

A =	α	a_m
B =	$\beta\ b_n$	

Table: case 2

A =	α	a_m
B =	β	b_n

Table: case 3

If we use $D(str_1,\ str_2)$ to denote the edit distance between strings str_1 and str_2, then it is not hard to see that:

$$D(A,\ B) = \begin{cases} D(\alpha a_m,\ \beta) + 1 & \text{if case 1 happens} \\ D(\alpha,\ \beta b_n) + 1 & \text{if case 2 happens} \\ D(\alpha,\ \beta) + 1 & \text{if case 3 happens, and } a_m \neq b_n \\ D(\alpha,\ \beta) & \text{if case 3 happens, and } a_m = b_n \end{cases}$$

Good news: We seems to break the problem into subproblems of **smaller size**!

Issues: However, there are 4 possible cases, and we don't know which one will actually happen.

Solution:

- By the definition of editing distance, we know that our goal is to **minimize** the number of operations.
- Moreover, there are only 4 possible cases using our classification method. This allows us to brute-force all possibilities *efficiently*.
- Thus, we know for sure that

$$D(A,B) = \min\left\{D(\alpha a_m,\beta)+1,\ D(\alpha,\beta b_n)+1,\ D(\alpha,\beta)+[a_m == b_n]\right\},$$

where $[am == b_n] := \begin{cases} 1 & a_m = b_n \\ 0 & a_m \neq b_n \end{cases}.$

Summary of our observations so far

We classify the alignments using the last char:

A =	α a_m	
B =	β	b_n

Table: case 1

A =	α	a_m
B =	β b_n	

Table: case 2

A =	α	a_m
B =	β	b_n

Table: case 3

Then, it must hold that

$$D(A, B) = \min\left\{D(\alpha a_m, \beta) + 1,\ D(\alpha, \beta b_n) + 1,\ D(\alpha, \beta) + [a_m == b_n]\right\}.$$

Punchline: this structure is generally true, not only specific to the last char! We will be done once we generalize this **recursively** to all subproblems.

Recursive Thinking

Some Convenient Notations (for $1 \leq i \leq m$ and $1 \leq j \leq n$):

Let $A[1 \ldots i]$ and $B[1 \ldots j]$ be the prefixes of the strings A and B. I.e.,

$$A[1 \ldots i] = a_1 a_2 \ldots a_i \quad \text{and} \quad B[1 \ldots j] = b_1 b_2 \ldots b_j.$$

Let $D(i, j)$ be the edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$.

Goal: Compute $D(m, n)$

Recursive Relation

Let us generalize the earlier ideas about “alignments” to any i and j:

$A[1\ldots i] =$	$a_1 \ldots a_{i-1}\ a_i$	
$B[1\ldots j] =$	$b_1 \ldots b_{j-1}$	b_j

Table: case 1

$A[1\ldots i] =$	$a_1 \ldots a_{i-1}$	a_i
$B[1\ldots j] =$	$b_1 \ldots b_{j-1}\ b_j$	

Table: case 2

$A[1\ldots i] =$	$a_1 \ldots a_{i-1}$	a_i
$B[1\ldots j] =$	$b_1 \ldots b_{j-1}$	b_j

Table: case 3

Then, it must hold that

$$D(i,j) = \min\left\{D(i,j-1)+1,\ D(i-1,j)+1,\ D(i-1,j-1)+[a_i == b_j]\right\}.$$

Boundary Cases

Don't forget about the easy-yet-crucial boundary conditions.

$$\begin{aligned} D(i,0) &= i \quad \text{(delete all characters from } A[1\ldots i]) \\ D(0,j) &= j \quad \text{(insert all characters of } B[1\ldots j]) \end{aligned}$$

Recursive Formula: Summary

In summary, we have found the complete recursive formula for the **Edit Distance Problem**:

For all $0 \leq i \leq m$ and $0 \leq j \leq n$,

$$D(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\left\{ D(i,j-1)+1,\ D(i-1,j)+1,\ D(i-1,j-1)+[a_i == b_j] \right\} & i \neq 0 \text{ and } j \neq 0 \end{cases}$$

The edit distance between A and B is given by the value $D(m,n)$.

Are we done?

- Don't forget the final step of Dynamic Programming—finding the **proper order** in which to solve the subproblems.

Solving subproblems in order

Key observation:

- To solve the subproblem $D(i, j)$, we only need to know the solutions to three subproblems: $D(i-1, j)$, $D(i, j-1)$, and $D(i-1, j-1)$.
- (we also need to know a_i and b_j. But this is trivial—simply read them from A and B.)

This reveals the proper order to solve the subproblems. This order is best illustrated by a 2-dimensional **dynamic programming table** (shown on the next slide).

Dynamic Programming Table

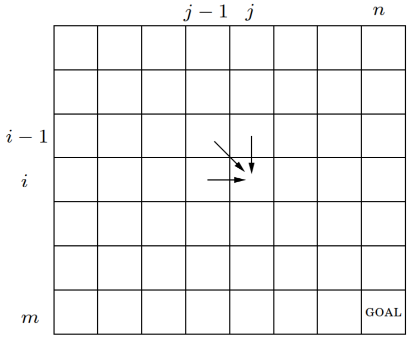

Figure: DP table for Edit Distance Problem (from Section 6.3 of [DPV])

$$A = a_1 a_2 \ldots a_m, \quad B = b_1 \ldots b_n$$

$$D(i,j) = \min\left\{D(i,j-1)+1,\ D(i-1,j)+1,\ D(i-1,j-1)+[a_i == b_j]\right\}$$

Fill the table following the suggested order. We can solve the problem **in time** $O(mn)$.

An Example of DP Table

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Figure: An example with $A = exponential$ and $B = polynomial$ (from Section 6.3 of [DPV])

Pseudo-code

Input: Strings $A[1\dots m]$, $B[1\dots n]$
Initialize:

$$D[0][j] \leftarrow j \quad \text{for } j = 0\dots n$$
$$D[i][0] \leftarrow i \quad \text{for } i = 0\dots m$$

Fill table:

```
• For $i = 1$ to $m$:
    • For $j = 1$ to $n$:
        • If $A[i] = B[j]$, set $D[i][j] \leftarrow D[i-1][j-1]$
        • Else,
```

$$D[i][j] \leftarrow 1 + \min \begin{cases} D[i-1][j] & \text{/* delete */} \\ D[i][j-1] & \text{/* insert */} \\ D[i-1][j-1] & \text{/* substitute */} \end{cases}$$

Reconstructing the Edits

Piggybacking:

- Store direction of choice at each cell:
    - Diagonal: match or substitution
    - Up: deletion
    - Left: insertion
- Backtrack from $D[m][n]$ to $D[0][0]$ to find the edit sequence

Closing Remarks

In this problem, we once again see how the idea of dynamic programming guides us to break the problem down into:

subproblems that can be solved in a nice order.

As before, this process often involves steps that may feel a bit magical. Unfortunately, there is no universal rule that always works—we have to rely on our experience and creativity.

That said, Page 178 of our textbook [DPV] provides some widely used templates for designing subproblems.