Algorithms and Data Structures
Module 3. Dynamic programming

Lecture 16
Edit distance.
The Longest Common Subsequence.

Edit distance

The notion of ‘distance’ in math is the generalization of a ‘physical distance’ (= Euclidian distance). In general, ’distance’ (or ‘metric’) is the measure of difference between two objects (the more is the distance, the more different the two objects are).

Edit distance

Definition. ***Distance*** (metric) is a numerical function $d: X \times X \longrightarrow R_+$ which satisfies 'metric axioms' for all $x, y, z \in X$:

1. $d(x, y) = 0 \Leftrightarrow x = y$;
2. $d(x, y) = d(y, x)$;
3. $d(x, y) \leq d(x, z) + d(z, y)$; (*triangle inequality*)

Edit distance

Examples of distances are:

- *Euclidian distance* in R^n: $d(x, y) = \sqrt[2]{\sum_{i=1}^{n}(x_i - y_i)^2}$
- *Graph distance*: $d_G(x, y)$ is the length (weight) of the shortest path between vertices x and y.
- *Hamming distance*: if x and y are strings of equal length, $d_H(x, y)$ is the number of positions in which x and y differ.
- *Edit distance*.

Edit distance

<u>Definitions</u>.

- An ***alphabet*** is a finite set of distinct elements, called ***symbols*** or ***letters***.

Examples: $\{0,1\}, \{0,1,2,3,4,5,6,7,8,9\}, \{a, b, \dots, z\}, \{A, C, G, T\}$

- A ***word*** in alphabet A is a finite ***sequence*** (*string*)of symbols of A. The symbols in a word may coincide. The order of symbols in a word <u>does matter</u>.

Examples: 'AACTAC' is a word of length 6.

Edit distance

Let P,Q and R be sequences (words, strings) in the same alphabet.

P = ‘HONEY’

Q = ‘FOOD’

R = ‘MONEY’

Is ‘HONEY’ closer to ‘FOOD’ then to ‘MONEY’?

Edit distance

Let P,Q and R be sequences (words, strings) in the same alphabet.

P = ‘HONEY’

Q = ‘FOOD’

R = ‘MONEY’

H O N E Y
M O N E Y

(1 difference)

H O N E Y
F O   O D

(4 differences)

Edit distance

<u>Definition</u>

Let P and Q be two sequences (words, strings).

The ***edit distance*** between P and Q is the minimum number of operations required to transform P into Q (or vice versa).

There are several versions of edit distance, differing in the set of operations considered.

Edit distance

<u>Definition</u>

The ***Levenshtein distance*** is the minimum number of *insertions/deletions (indels)* or *substitutions* required to transform P into Q (or vice versa).

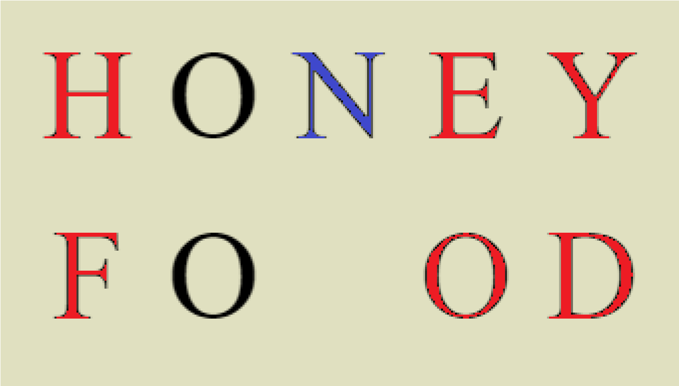
FOOD → MOOD → MON∧D → MONED → MONEY

http://jeffe.cs.illinois.edu/teaching/algorithms/

Edit distance

<u>Definition</u>

The ***Levenshtein distance*** is the minimum number of *insertions/deletions (indels)* or *substitutions* required to transform P into Q (or vice versa).



Edit distance

Other possible operations:

- Transpositions: CFOFEE -> COFFEE
- Inversions: AACGATTTA -> AATTAGCTA

Edit distance

Let us design a DP algorithm for calculating Levenshtein edit distance.

The 1st step: we need a recurrence for the optimal solution (= the minimum number of operations).

To build a recurrence we need to formulate *the principle of optimality* for the given problem.

Edit distance

Generic form of the principle of optimality: a part of an optimal solution is an optimal solution of a subproblem.

Example (http://jeffe.cs.illinois.edu/teaching/algorithms/):

P = ‘ALGORITHM’

Q = ‘ALTRUISTIC’

Edit distance

Let us consider an optimal alignment of these strings.

P = 'ALGORITHM'	A	L	G	O	R		I		T	H	M
Q = 'ALTRUISTIC'	A	L		T	R	U	I	S	T	I	C

We can formulate the principle of optimality: for all *k*, the leftmost *k* columns of an optimal alignment represent an optimal alignment for the corresponding prefixes of the strings.

Edit distance

Let us consider an optimal alignment of these strings.

P = ‘ALGORITHM’

Q = ‘ALTRUISTIC’

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

We can formulate the principle of optimality: for all *k*, the leftmost *k* columns of an optimal alignment represent an optimal alignment for the corresponding prefixes of the strings.

Edit distance

The principle of optimality:

For all *k*, the leftmost *k* columns of an optimal alignment represent an optimal alignment for the corresponding prefixes of the strings.

Let $\delta(i, j)$ be the edit distance between $P[1..i]$ and $Q[1..j]$. We need to calculate $\delta(m, n)$, for $m = |P|, n = |Q|$.

Edit distance

Let $\delta(i, j)$ be the edit distance between $P[1..i]$ and $Q[1..j]$.

The last column in the optimal alignment of P and Q can represent one of the 3 situations:

1) Insertion: $\delta(i,j)$=$\delta(i, j-1)+1$

Edit distance

2) Deletion: $\delta(i,j)$=$\delta(i-1,j)+1$

3) Substitution:

$a)\ p[i] \neq q[j]$: $\delta(i,j)$=$\delta(i-1,j-1)+1$

$b)\ p[i] = q[j]$: $\delta(i,j)$=$\delta(i-1,j-1)$

Edit distance

Base cases: $i = 0$ or $j = 0$. => one of the prefixes, or both, are empty.

- $i = 0$: to transform an empty string to a string of length j, we need j insertions => $\delta(0, j) = j$.
- $j = 0$: => $\delta(i, 0) = i$.

Edit distance

Recurrence:

$$\delta(i,j) = \begin{cases} j, & if\ i = 0 \\ i, & if\ j = 0 \\ \min\left\{\begin{matrix} \delta(i,j-1) \\ \delta(i-1,j) \\ \delta(i-1,j-1) + \Delta(p[i],q[j]) \end{matrix}\right\}, & otherwise \end{cases},$$

where $\Delta(x,y) = \begin{cases} 0, & if\ x = y \\ 1, & if\ x \neq y \end{cases}$

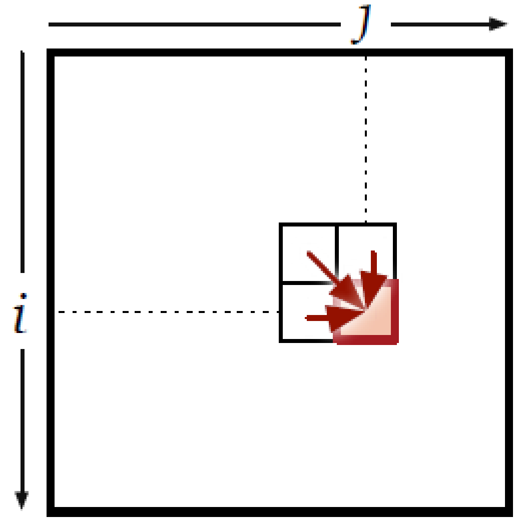
Edit distance

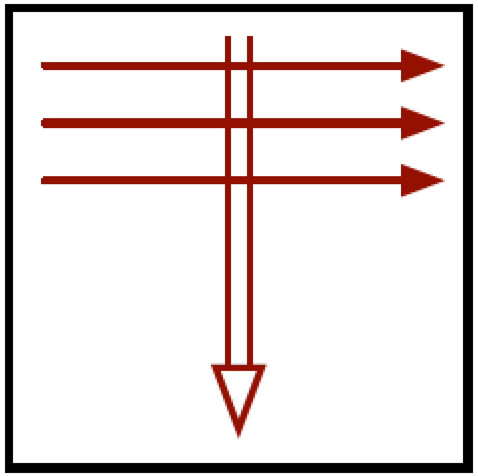
Let us implement this recurrence in (pseudo)code.

- The recurrent function $\delta(i, j)$ has 2 arguments => we need a 2D table (matrix) to store the results for the subproblems.
- D[0..m, 0..n]

Edit distance

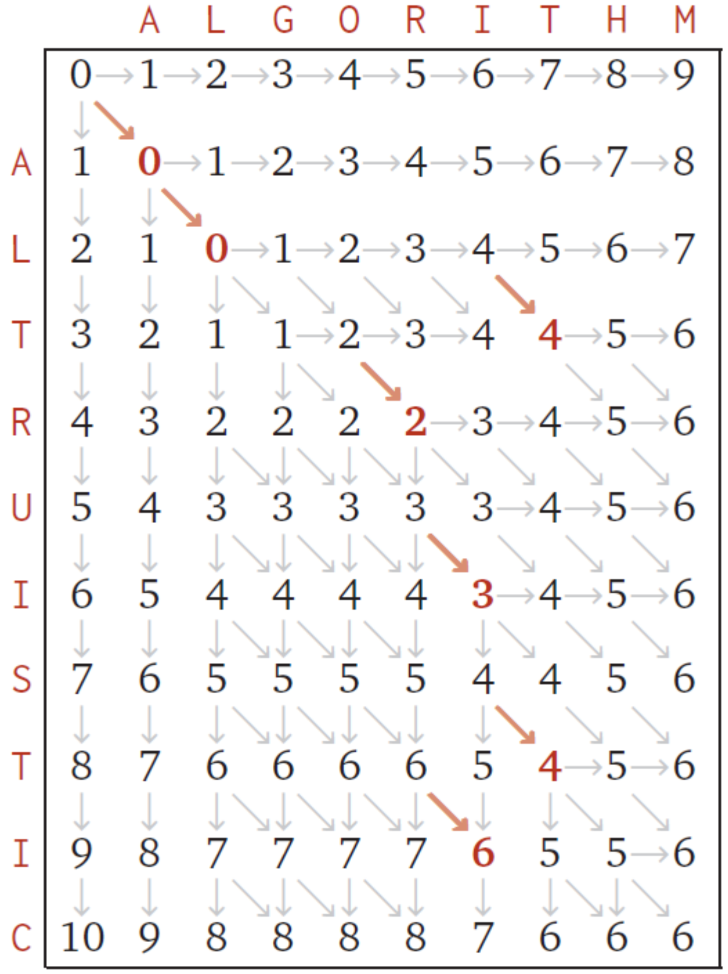
- A possible order we fill in the table D depends on the data dependencies in the recurrence.
- To calculate d$[i, j]$, we need only values of d$[i - 1, j], d[i, j - 1]$ and $d[i - 1, j - 1]$.





Edit distance

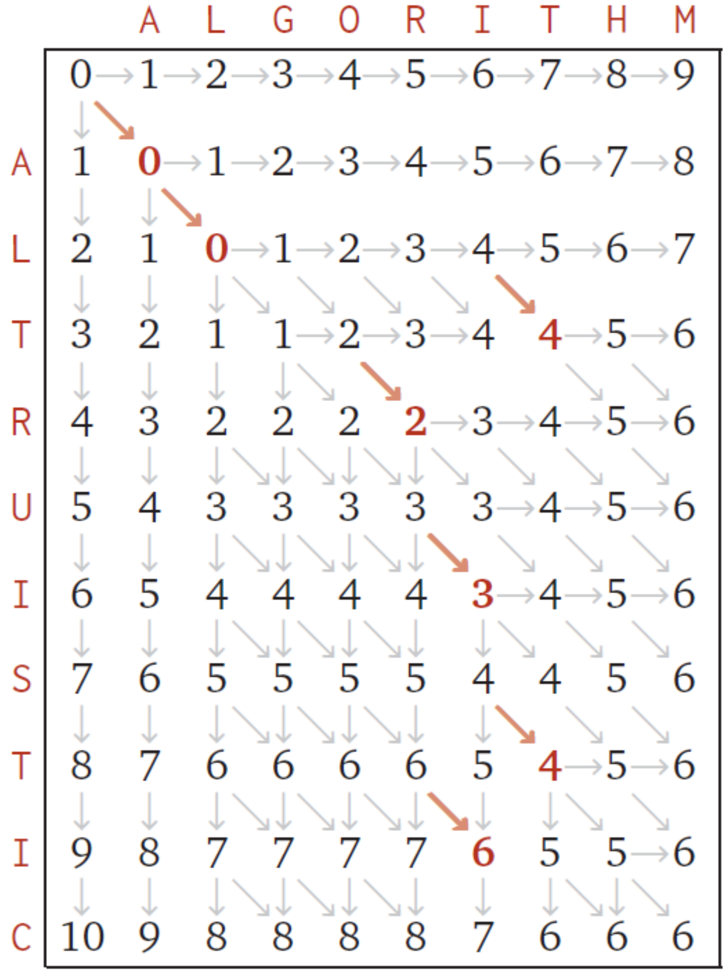
```
// Initialization (the base cases)
for i=0 to m: d[i,0] = i;
for j=0 to n: d[0,j] = j;
// Filling the table
for i=1 to m:
        for j=1 to n:
                ins = d[i,j-1]+1
                del = d[i-1,j]+1
                if p[i]=q[j] then sub = d[i-1,j-1]
                        else sub = d[i-1,j-1]+1
                d[i,j] = min(ins,del,sub)
```



Edit distance

Building an optimal alignment:

- start from the [*m*,*n*] entry (bottom-right corner);
- move backwards to the [0,0] (top-left corner);
- at the current entry [i,j]: compare d[i,j-1]+1, d[i-1,j]+1, d[i-1,j-1](+1) and move to the entry corresponding to the minimum expression + make appropriate operations in the alinment.



Edit distance

```
A L G O R I   T H M
A L T R U I S T I C
```

```
A L G O R   I   T H M
A L   T R U I S T I C
```

```
A L G O R   I   T H M
A L T   R U I S T I C
```

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Edit distance

The space and time complexities are: $O(m \cdot n)$.

Can we reduce the space complexity?

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Edit distance

Q: Can we reduce the space complexity?

A: Yes, we can– if we need the distance only. We can keep 2 rows instead of *n* rows. Thus, we reduce the space complexity from $O(m \cdot n)$ to $O(m)$.

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Edit distance

Some illustrative online calculators:

https://phiresky.github.io/levenshtein-demo/

http://www.let.rug.nl/~kleiweg/lev/

Edit distance

Generalization of the edit distance: weights for operations (indels, substitutions).

Special cases:

- $w(indel) = +\infty$; $w(sub)$=1 => we get Hamming distance.
- $w(indel)$ =0; $w(sub) = +\infty$ => we get the Longest Common Subsequence (LCS) problem.

Longest Common Subsequence

Definitions

- Let *P* be a word (sequence). A word/sequence *Q* is a *subsequence* of *P* iff *Q* contains some letters of *P* in the same order, with possible gaps.

A formal definition. Let $P = p_1 p_2 \ldots p_n$ and $Q = q_1 q_2 \ldots q_m$, $m \leq n$. *Q* is a subsequence of *P* iff there exists an increasing sequence of indices $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ such that $q_k = p_{i_k}$ for all $k = 1, \ldots, m$.

Example: 'LOT' is a subsequence of 'ALGORITHM'.

Longest Common Subsequence

<u>Definitions</u>

- *S* is a *common subsequence* of *P* and *Q* if *S* is a subsequence of *P* and a subsequence of *Q*.

Example: ‘LOT’ is a common subsequence of ‘A<u>L</u>G<u>O</u>RI<u>T</u>HM’ and ‘S<u>LO</u>WES<u>T</u>’.

- S is the *longest common subsequence* (LCS) of *P* and *Q* if *S* is a common subsequence of *P* and *Q* of the maximum length.

Longest Common Subsequence

Idea of a recurrence for the LCS problem.

Let $P = p_1 p_2 \dots p_n$ and $Q = q_1 q_2 \dots q_m$.

The LCS of *P* and *Q* is the longest of the 3 subsequences:

$1)\, LCS(p_1 p_2 \dots p_{n-1}, q_1 q_2 \dots q_{m-1}) + p_n$, if $p_n = q_m$;

$2)\, LCS(p_1 p_2 \dots p_{n-1}, q_1 q_2 \dots q_m)$

$3)\, LCS(p_1 p_2 \dots p_n, q_1 q_2 \dots q_{m-1})$

Longest Common Subsequence

Base cases: if either of P and Q is empty, then $LCS(P, Q)$ is an empty string.

The computational scheme is very similar to that of the algorithm for edit distance.

Weighted edit distance

Generalization of the edit distance: weights for operations (indels, substitutions).

In the general case, the weights for substitutions may differ for different pairs of letters.

Weighted edit distance

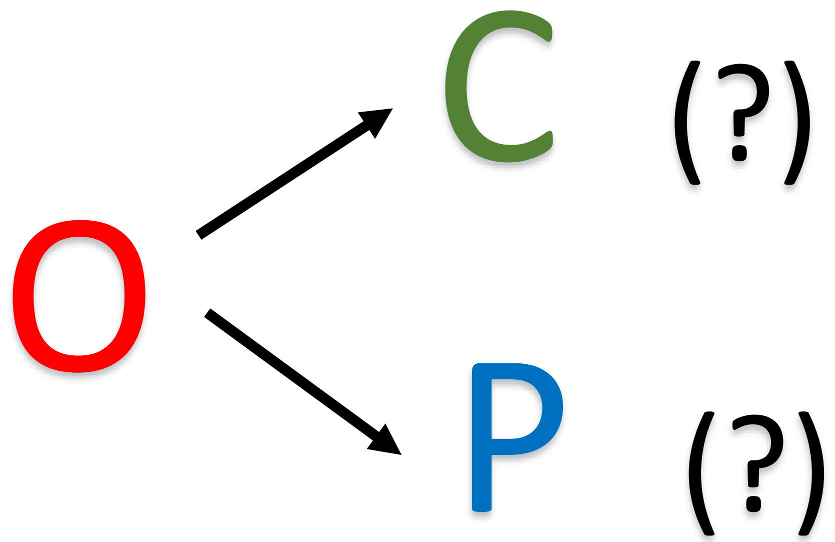
Application: protein structures comparison

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Weighted edit distance

Application: error correction

- misprints (typos) of users
- errors of the optical character recognition (OCR) software



Weighted edit distance

DP algorithm: modification **Needleman-Wunsch** algorithm.

Why modification? The original Needleman-Wunsch algorithm *maximizes similarity* instead of minimizing distance.

Weighted edit distance

```
// Initialization (the base cases)
for i=0 to m: d[i,0] = i*w_indel;
for j=0 to n: d[0,j] = j*w_indel;
// Filling the table
for i=1 to m:
      for j=1 to n:
            ins = d[i,j-1]+w_indel
            del = d[i-1,j]+w_indel
            sub = d[i-1,j-1]+w_sub[p[i],q[j]]
            d[i,j] = min(ins,del,sub)
```