COMP251: Dynamic programming (2)

Jérôme Waldispühl

School of Computer Science

McGill University

Based on (Kleinberg & Tardos, 2005)

Bellman’s principle of optimality

``An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.’’

PAIRWISE SEQUENCE ALIGNMENT

How similar two strings are?

S_1 = `tentation` S_2 = `temptation`

```
ten-tation
temptation
```

Pairwise sequence alignment

Definition (Pairwise sequence alignment)
Let $a=a_1 \ldots a_m$ and $b=b_1 \ldots b_n$ be two sequences over an alphabet Σ (i.e. $a, b \in \Sigma^*$). A pairwise alignment is a mapping *f* of the letters of *a* to *b*, such that if $f(a_i,b_j)$ and $f(a_k,b_l)$ *then* $i<k$ & $j<l$ *or* $k<i$ & $l<j$.

Example: *a*=`ABBCEE`, *b*=`BBCCDE`

```
ABBC--EE   ABB-C-EE   ABBCEE  A-B-B-C-E-E-
 |||   |     || |  |  :|:|:|
-BBCCDE-   -BBCCD-E   BBCCDE  -B-B-C-C-D-E
```

Note: letters can be mapped to an empty character.

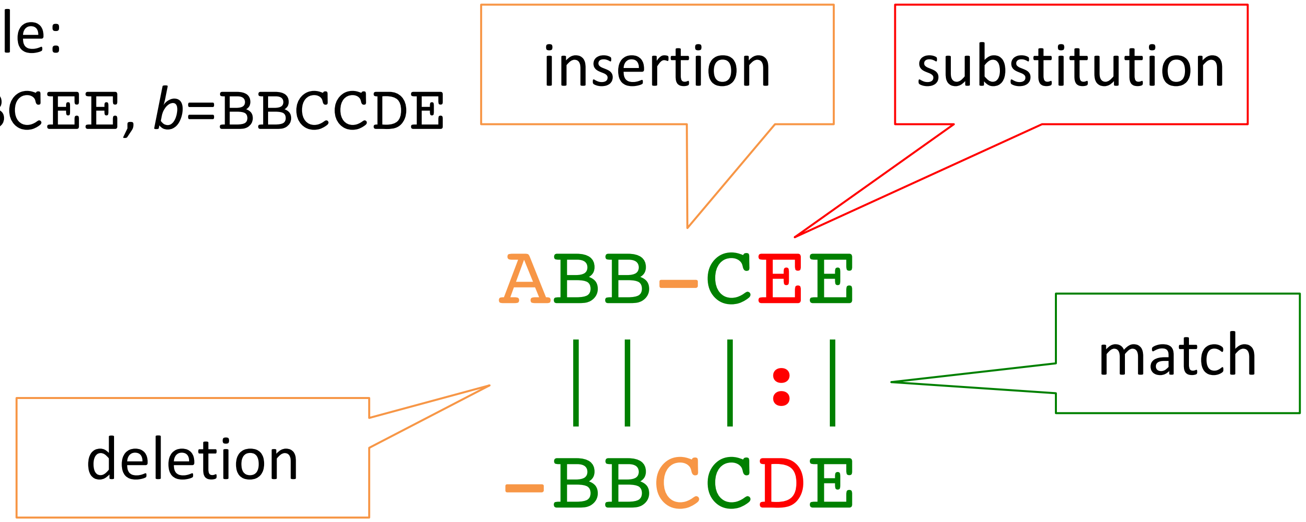
Question: All are valid alignments, but which one is best?

Vocabulary

Match: letters are identical
Substitution: letters are different
Insertion: a letter of *b* is mapped to the empty character
Deletion: a letter of *a* is mapped to the empty character

Insertions & deletions are also call **indels**.

Example:
a=ABBCEE, *b*=BBCCDE



Notations

The empty character/letter is noted -

An alignment can be decomposed in column (or vectors).

$$\begin{matrix} a_1 & a_2 & - \\ - & b_1 & b_2 \end{matrix} = \begin{pmatrix} a_1 \\ - \end{pmatrix} \begin{pmatrix} a_2 \\ b_1 \end{pmatrix} \begin{pmatrix} - \\ b_2 \end{pmatrix}$$

Bioinformatics

Let *a* and *b* be two homologous (same function) biological sequences (DNA, RNA, Protein).

A sequence alignment allows us to estimate the similarity between the 2 sequences in order to:

- model evolution
- reveal functional motifs

```
ACCAGTAGCGGGGGACA---GACCTCGCAT
ATC--TAGGGGGGGACATTTGACGACGC--
```

Counting alignments (1)

Let $a=a_1 \ldots a_m$ and $b=b_1 \ldots b_n$ be two sequences over an alphabet Σ.

Let *c(m,n)* be the the number of alignments that can be formed between them.

First, we note that an alignment of *a* and *b must end by:*

$$\begin{pmatrix} a_m \\ - \end{pmatrix} \qquad \begin{pmatrix} a_m \\ b_n \end{pmatrix} \qquad \begin{pmatrix} - \\ b_n \end{pmatrix}$$

Thus, $c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)$

Counting alignments (2)

We have a recursion:

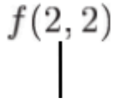
$c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)$

Initialization?

$f(0,n)=f(m,0)=f(0,0)=1$

Recursive evaluation (top-down):

5+3+5=13



1+1+3=5

3+1+1=5

1+1+1=3

1+1+1=3

1+1+1=3

Counting alignments (2)

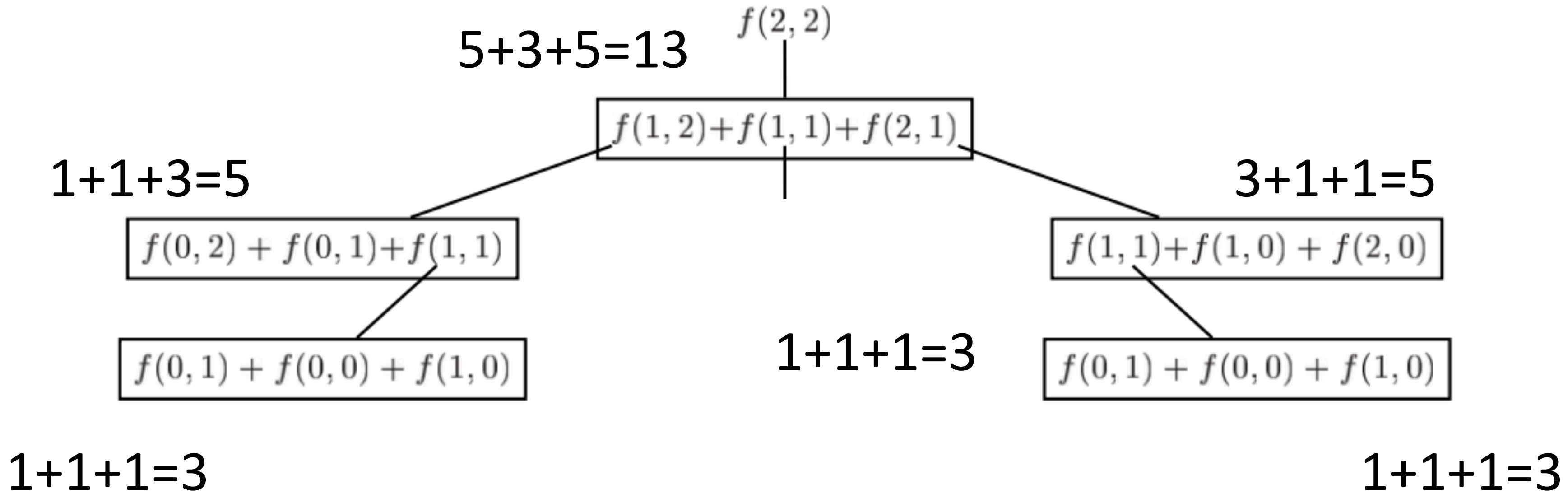
We have a recursion:

c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)

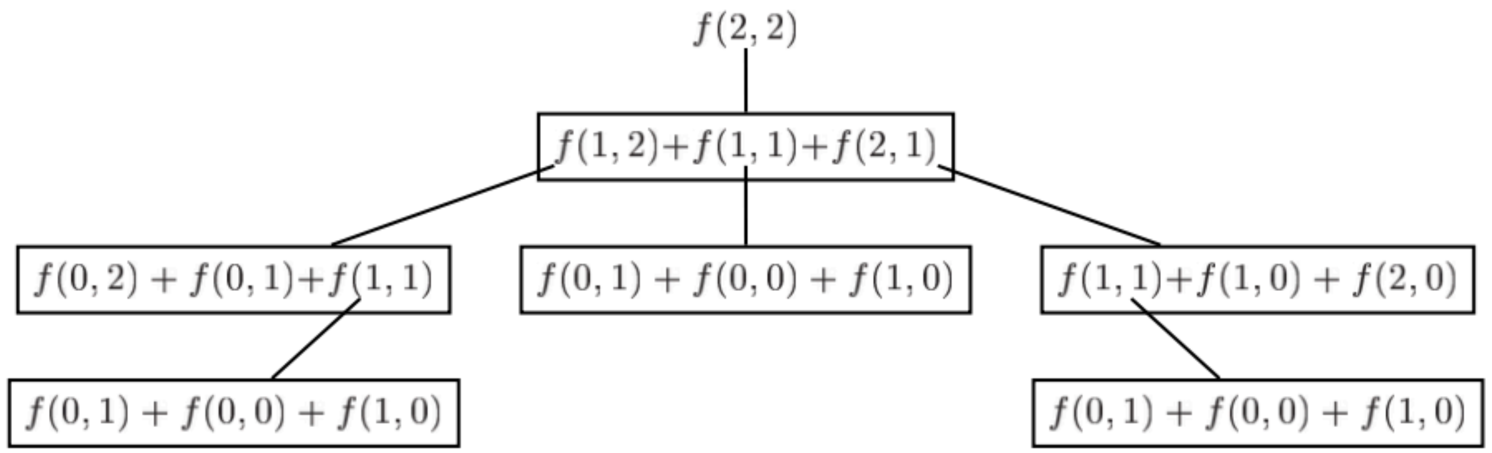
Initialization?

f(0,n)=f(m,0)=f(0,0)=1

Recursive evaluation (top-down):



Counting alignments (3)



Note: f(1,1) appears 3 times, and evaluated 3 times…

How to speed up this calculation?

- Memoization
- Dynamic programming

Counting alignments (4)

$$c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)$$

- Indices of c() are strictly decreasing during the recursion
- We can compute c() for smaller indices first (bottom-up)
- Define a partial order on the c() such that c(i,j)<c(i',j') iff i<i' or j<j'
- Compute c() using this partial order:

```
for i=0 to m do
   for j=0 to n do
      c(i,j) = c(i-1,j)+c(i-1,j-1)+c(i,j-1)
```

- Complexity: O(mn)

Optimal pairwise alignment

a=ABBCEE, *b*=BBCCDE

```
ABBC--EE  ABB-C-EE  ABBCEE  A-B-B-C-E-E-
 |||  |    || |  |  :|:|:|
-BBCCDE-  -BBCCD-E  BBCCDE  -B-B-C-C-D-E
```

Among all alignments, which one is the best?

Levenshtein distance

Definition (Levenshtein Distance)
The Levenshtein Distance between two words/sequences is the minimal number of substitutions, insertions and deletions to transform one into the other.

Example:

```
ABB-CEE
-BBCCDE
```

1 deletion + 1 insertion + 1 substitution $\Longrightarrow$ d=3

Edit cost/distance

Definition (edit cost)
Let δ(x,y) be a cost function for each edit operation (match, substitution, insertion, deletion). The edit cost of two words/sequences is the sum of the cost of each edit operation used transform one into the other.

Definition (edit distance)
The edit distance between two words/sequences is the minimal cost (sometimes max) to transform one into the other.

Example:

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

```
ABB-CEE       4 match+1 deletion+1 insertion+1 substitution
-BBCCDE       ⟹ d = 4 * (0) + 1 * (+1) + 1 * (+1) + 1 * (+1)  =  3
```

Edit distance

If

- Every edit operation has positive cost
- for every operation, there is an inverse operation with equal cost

Then, the edit distance is a metric:

• $d(x,y) \geq 0$	(separate axiom)
• $d(x,y) = 0$ iff $x=y$	(coincidence axiom)
• $d(x,y) = d(y,x)$	(symmetry)
• $d(x,y) \leq d(x,z) + d(z,y)$	(triangle inequality)

Optimal sub-structure

``A sub-alignment of an optimal alignment w.r.t. the edit cost is also optimal''

Proof: cut-and-paste argument & contradiction

- Let A be an optimal alignment
- Let $A = A_1A_2A_3$ be a decomposition of A such that A_2 is not optimal.
- Let A'_2 be an optimal alignment of the substrings in A_2
- Substitute A_2 by A'_2 to build a new alignment A'
- $\delta(A') = \delta(A_1A'_2A_3) = \delta(A_1)+\delta(A'_2)+\delta(A_3)$

  $< \delta(A_1)+\delta(A_2)+\delta(A_3) = \delta(A_1A_2A_3) = \delta(A)$
- contradiction with A optimal

Problem Structure

Definition (dynamic array):
$d(i,j)$ = minimal cost of aligning prefix strings $a_1 \ldots a_i$ and $b_1 \ldots b_j$.

Case 1 (a_i matches b_j)
cost of matching a_i with b_j + min cost of aligning $a_1 \ldots a_{i-1}$ and $b_1 \ldots b_{j-1}$.

Case 2a (a_i deleted)
cost of deletion of a_i + min cost of aligning $a_1 \ldots a_{i-1}$ and $b_1 \ldots b_j$.

Case 2b (b_j inserted)
cost of insertion of b_j + min cost of aligning $a_1 \ldots a_i$ and $b_1 \ldots b\text{-}1_j$.

Recursion

Insert string of length *j*

Match, delete, or insert rightmost character(s)

$$d(i,j) = \begin{cases} j \cdot \delta(-,*) & if\ i = 0 \\ min \begin{cases} \delta(a_i, b_j) + d(i-1, j-1) \\ \delta(a_i, -) + d(i-1, j) \\ \delta(-, b_j) + d(i, j-1) \end{cases} & otherwise \\ i \cdot \delta(*, -) & if\ j = 0 \end{cases}$$

Delete string of length *i*

Needleman-Wunch Algorithm

```
for i=0 to m do
   $d(i,0)=i*\delta(-,-)$
for j=0 to n do
   $d(0,j)=j*\delta(-,-)$

for i=1 to m do
      for j=1 to n do
         $d(i,j) = \min(d(i-1,j)+\delta(a_i,-),$
                      $d(i-1,j-1)+\delta(a_i,b_j),$
                      $d(i,j-1)+\delta(-,b_j))$
return d(m,n)
```

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1				
T	2				

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	?			
T	2				

- match/substitution: d(0,0) + δ(A,C)=0+(+1)=+1

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	↘ ? ↓			
T	2				

- match/substitution: d(0,0) + δ(A,C)=0+(+1)=+1
- insertion: d(1,0)+δ(-,C) = 1+(+1)=+2

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	↘ ↓ → ?			
T	2				

- match/substitution: d(0,0) + δ(A,C)=0+(+1)=+1
- insertion: d(1,0)+δ(-,C) = +1+(+1)=+2
- deletion: d(0,1)+δ(A,-) = +1+(+1)=+2

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	$\searrow\ \rightarrow\ \downarrow$ 1			
T	2				

- match/substitution: d(0,0) + δ(A,C)=0+(+1)=+1
- insertion: d(1,0)+δ(-,C) = +1+(+1)=+2
- deletion: d(0,1)+δ(A,-) = +1+(+1)=+2

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2		
T	2				

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	
T	2				

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2				

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2			

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1		

Example

a=ATTG b=CT

$$\delta(x, y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	2	

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	2	3

Backtracking

How to retrieve the optimal alignment?

- Each move is associated to one edit operation
  - Vertical = insertion
  - Diagonal = match/substitution
  - Horizontal = deletion
- We use one of these 3 move to fill a cell of the array
- From the bottom-right corner (i.e. d(m,n)), find the move that has been used to determine the value of this cell.
- Apply this principle recursively.

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	2	3

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	2 $\leftarrow$	3

G

–

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	↖ 2	← 3

```
TG
T-
```

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

	-	A	T	T	G
-	0	1	2	3	4
C	1	1 ←	2	3	4
T	2	2	1	↖ 2 ←	3

```
TTG
-T-
```

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$

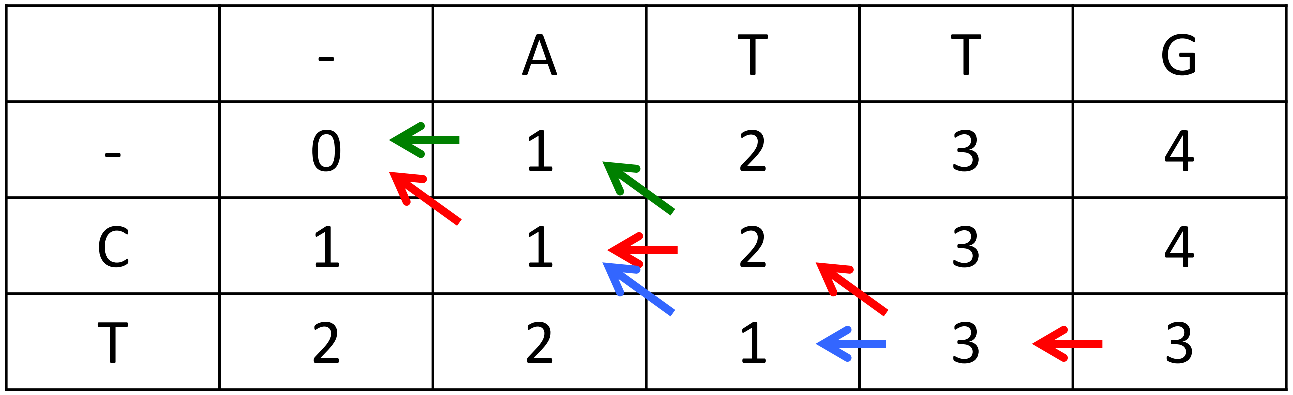
	-	A	T	T	G
-	0	1	2	3	4
C	1	↖ 1	← 2	3	4
T	2	2	1	↖ 2	← 3

```
ATTG
C-T-
```

Example

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 0 & if\ x = y \\ 1 & otherwise \end{cases}$$



	-	A	T	T	G
-	0	1	2	3	4
C	1	1	2	3	4
T	2	2	1	3	3

```
ATTG   ATTG   ATTG
C-T-   CT--   -CT-
```

Analysis

Theorem: The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.

Proof:

- Algorithm computes edits distance.
- Can trace back to extract an optimal alignment.

Q. Can we avoid using quadratic space?
A. Easy to compute optimal value in $\Theta(mn)$ time and $\Theta(m+n)$ space.

- Compute OPT(i,•) from OPT(i-1,•).
- But, no longer easy to recover optimal alignment itself.

Bioinformatics

- Different cost functions, For instance:

$$\delta(x,y) = \begin{cases} 1 & if\ x = y \\ -1 & otherwise \end{cases}$$

  Cost of alignment is being maximized.

- Variants of optimal pairwise alignment algorithm:
  - Ignore trailing gaps (Smith & Waterman, 1981)

- Optimal alignment not practical for multiple sequences.