

# Bioinformatics Algorithms

## (Fundamental Algorithms, module 2)

**Zsuzsanna Lipták**

Masters in Medical Bioinformatics  
academic year 2018/19, II. semester

Pairwise Alignment 3

# Optimal pairwise alignment in linear space

Given two sequences  $s, t$  of length  $n$ :

- DP algorithm for global alignment:  $O(n^2)$  time and space
- if we only want the **score of an optimal alignment**  $\text{sim}(s, t)$  (problem variant 1), then we can do this in  $O(n^2)$  time and  $O(n)$  space (space-saving variant)
- But that algo does not give us the optimal alignment itself (problem variant 2)
- **Now:** algorithm for computing an optimal alignment itself in time  $O(n^2)$  but **space  $O(n)$**

There are several algorithms achieving this, e.g. Hirschberg (1975) a.k.a. Myers-Miller (1988). Here we present the divide-and-conquer algorithm from the book by Durbin, Eddy, Krogh, Mitchison: *Biological Sequence Analysis*, 1998 (ch. 2.6).

$s = \text{GAAGA}$ ,  $t = \text{CACA}$

match: 2, mismatch: -1, gap: -1

$D(i, j)$			C	A	C	A
		0	1	2	3	4
	0	0	-1	-2	-3	-4
G	1	-1	-1	-2	-3	-4
A	2	-2	-2	1	0	-1
A	3	-3	-3	0	0	2
G	4	-4	-4	-1	-1	1
A	5	-5	-5	-2	-2	1

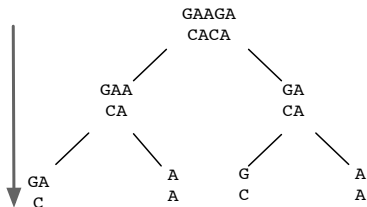
The optimal alignments are:

1.  $\begin{pmatrix} \text{GAAGA} \\ -\text{CACA} \end{pmatrix}$
2.  $\begin{pmatrix} \text{GAAGA} \\ \text{CA}-\text{CA} \end{pmatrix}$
3.  $\begin{pmatrix} \text{GAAGA} \\ \text{C}-\text{ACA} \end{pmatrix}$
4.  $\begin{pmatrix} \text{GAAGA} \\ \text{CAC}-\text{A} \end{pmatrix}$

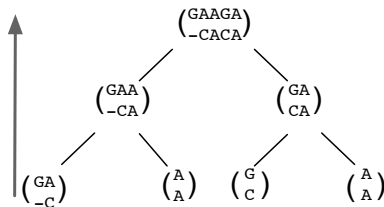
Consider the first optimal alignment  $\begin{pmatrix} \text{GAAGA} \\ -\text{CACA} \end{pmatrix}$ :

### Idea: Divide-and-conquer

We divide the two sequences  $s, t$  in two parts, left and right, align left with left, right with right, and then concatenate the two alignments:



*top-down: split sequences into two*

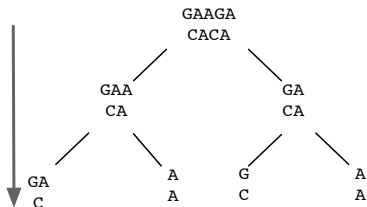


*bottom-up: concatenate alignments*

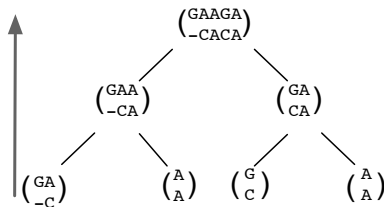
Consider the first optimal alignment  $\begin{pmatrix} \text{GAAGA} \\ -\text{CACA} \end{pmatrix}$ :

### Idea: Divide-and-conquer

We divide the two sequences  $s, t$  in two parts, left and right, align left with left, right with right, and then concatenate the two alignments:



*top-down: split sequences into two*



*bottom-up: concatenate alignments*

Why does this work?

Generalization of the theorem on which the DP recursion for pairwise alignment is based (see p. 18 of "Pairwise alignment 1"):

### Theorem

Let alignment  $\mathcal{A}$  be the concatenation of two alignments  $\mathcal{B}$  and  $\mathcal{C}$ , i.e.  $\mathcal{A} = \mathcal{B} \cdot \mathcal{C}$ . If  $\mathcal{A}$  is optimal, then so are  $\mathcal{B}$  and  $\mathcal{C}$ .

Generalization of the theorem on which the DP recursion for pairwise alignment is based (see p. 18 of "Pairwise alignment 1"):

### Theorem

Let alignment  $\mathcal{A}$  be the concatenation of two alignments  $\mathcal{B}$  and  $\mathcal{C}$ , i.e.  $\mathcal{A} = \mathcal{B} \cdot \mathcal{C}$ . If  $\mathcal{A}$  is optimal, then so are  $\mathcal{B}$  and  $\mathcal{C}$ .

### Proof

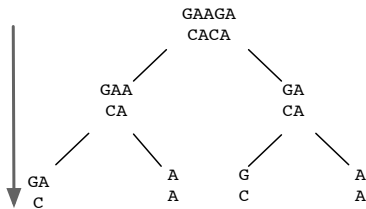
Again, we prove the claim by contradiction. Let  $\mathcal{A}$  be an alignment of  $s$  and  $t$ ,  $\mathcal{B}$  one of  $s'$  and  $t'$ , and  $\mathcal{C}$  one of  $s''$  and  $t''$ . (Thus  $s = s's''$  and  $t = t't''$ .) Assume that  $\mathcal{B}$  is not optimal, then  $\mathcal{B}$  can be replaced by some alignment  $\mathcal{B}'$  of the same strings  $s', t'$  with higher score than  $\mathcal{B}$ . Define  $\mathcal{A}' = \mathcal{B}' \cdot \mathcal{C}$ . Then  $\mathcal{A}'$  is also an alignment of  $s, t$ , and

$$\text{score}(\mathcal{A}') = \text{score}(\mathcal{B}') + \text{score}(\mathcal{C}) > \text{score}(\mathcal{B}) + \text{score}(\mathcal{C}) = \text{score}(\mathcal{A}),$$

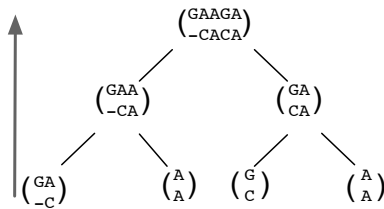
a contradiction to the optimality of  $\mathcal{A}$ .—The case where  $\mathcal{C}$  is not optimal is analogous. □



So it's okay to align optimally the left and the right parts, and then to concatenate them:

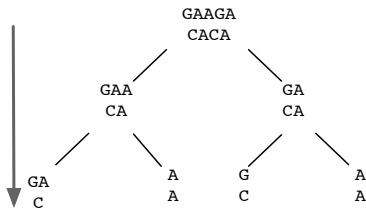


*top-down: split sequences into two*

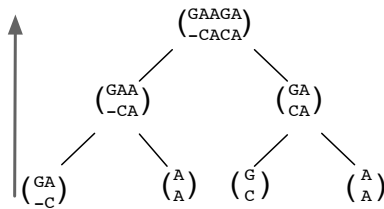


*bottom-up: concatenate alignments*

So it's okay to align optimally the left and the right parts, and then to concatenate them:



*top-down: split sequences into two*



*bottom-up: concatenate alignments*

## Question

But how do we know **where** to divide them?

**The problem is:** The reverse of the theorem is not true!  
Concatenating two optimal al's does not always yield an optimal al.:  
e.g.  $\begin{pmatrix} GA \\ G- \end{pmatrix} \cdot \begin{pmatrix} -C \\ AC \end{pmatrix}$  yields  $\begin{pmatrix} GA-C \\ G-AC \end{pmatrix}$ , which is not optimal.

**The problem is:** The reverse of the theorem is not true!

Concatenating two optimal al's does not always yield an optimal al.:

e.g.  $\begin{pmatrix} GA \\ G- \end{pmatrix} \cdot \begin{pmatrix} -C \\ AC \end{pmatrix}$  yields  $\begin{pmatrix} GA-C \\ G-AC \end{pmatrix}$ , which is not optimal.

### Definition

A **cut** is a pair of positions  $(n', m')$ , where  $1 \leq n' \leq n$ , and  $1 \leq m' \leq m$  (with  $|s| = n, |t| = m$ ).

**The problem is:** The reverse of the theorem is not true!

Concatenating two optimal al's does not always yield an optimal al.:

e.g.  $\begin{pmatrix} \text{GA} \\ \text{G-} \end{pmatrix} \cdot \begin{pmatrix} \text{-C} \\ \text{AC} \end{pmatrix}$  yields  $\begin{pmatrix} \text{GA-C} \\ \text{G-AC} \end{pmatrix}$ , which is not optimal.

## Definition

A **cut** is a pair of positions  $(n', m')$ , where  $1 \leq n' \leq n$ , and  $1 \leq m' \leq m$  (with  $|s| = n, |t| = m$ ).

We are looking for a **good** cut, i.e. one for which there is an optimal alignment passing through it.

- $(3, 2)$  is a good cut: the optimal alignments  $\begin{pmatrix} \text{GAAGA} \\ \text{-CACA} \end{pmatrix}$ ,  $\begin{pmatrix} \text{GAAGA} \\ \text{CA-CA} \end{pmatrix}$ ,  $\begin{pmatrix} \text{GAAGA} \\ \text{C-ACA} \end{pmatrix}$  all pass through the cell  $(3, 2)$ , aligning GAA with CA.

**The problem is:** The reverse of the theorem is not true!

Concatenating two optimal al's does not always yield an optimal al.:

e.g.  $\begin{pmatrix} \text{GA} \\ \text{G-} \end{pmatrix} \cdot \begin{pmatrix} \text{-C} \\ \text{AC} \end{pmatrix}$  yields  $\begin{pmatrix} \text{GA-C} \\ \text{G-AC} \end{pmatrix}$ , which is not optimal.

## Definition

A **cut** is a pair of positions  $(n', m')$ , where  $1 \leq n' \leq n$ , and  $1 \leq m' \leq m$  (with  $|s| = n, |t| = m$ ).

We are looking for a **good** cut, i.e. one for which there is an optimal alignment passing through it.

- $(3, 2)$  is a good cut: the optimal alignments  $\begin{pmatrix} \text{GAAGA} \\ \text{-CACA} \end{pmatrix}$ ,  $\begin{pmatrix} \text{GAAGA} \\ \text{CA-CA} \end{pmatrix}$ ,  $\begin{pmatrix} \text{GAAGA} \\ \text{C-ACA} \end{pmatrix}$  all pass through the cell  $(3, 2)$ , aligning GAA with CA.
- $(3, 3)$  is a good cut: the optimal alignment  $\begin{pmatrix} \text{GAAGA} \\ \text{CAC-A} \end{pmatrix}$  passes through the cell  $(3, 3)$ , aligning GAA with CAC.

**The problem is:** The reverse of the theorem is not true!

Concatenating two optimal al's does not always yield an optimal al.:

e.g.  $\begin{pmatrix} GA \\ G- \end{pmatrix} \cdot \begin{pmatrix} -C \\ AC \end{pmatrix}$  yields  $\begin{pmatrix} GA-C \\ G-AC \end{pmatrix}$ , which is not optimal.

## Definition

A **cut** is a pair of positions  $(n', m')$ , where  $1 \leq n' \leq n$ , and  $1 \leq m' \leq m$  (with  $|s| = n, |t| = m$ ).

We are looking for a **good** cut, i.e. one for which there is an optimal alignment passing through it.

- $(3, 2)$  is a good cut: the optimal alignments  $\begin{pmatrix} GAAGA \\ -CACA \end{pmatrix}$ ,  $\begin{pmatrix} GAAGA \\ CA-CA \end{pmatrix}$ ,  $\begin{pmatrix} GAAGA \\ C-ACA \end{pmatrix}$  all pass through the cell  $(3, 2)$ , aligning GAA with CA.
- $(3, 3)$  is a good cut: the optimal alignment  $\begin{pmatrix} GAAGA \\ CAC-A \end{pmatrix}$  passes through the cell  $(3, 3)$ , aligning GAA with CAC.
- $(3, 1)$  is not a good cut, since no optimal alignment passes through cell  $(3, 1)$ , i.e. no optimal alignment aligns GAA with C.

## Computing a good cut

1. In sequence 1, we will always take the middle cut position  $n' = \lceil n/2 \rceil$ .
2. In sequence 2, we will remember where the middle row  $n' = \lceil n/2 \rceil$  was crossed.
3. For this, we will need to compute another matrix  $M$  (again, in space-saving manner!).



## Matrix $M$

- **Definition:** For  $i \geq n'$ , cell  $M(i, j)$  contains an index  $r$  s.t. there exists an optimal alignment with score  $D(i, j)$  passing through cell  $(n', r)$ .

## Matrix $M$

- **Definition:** For  $i \geq n'$ , cell  $M(i, j)$  contains an index  $r$  s.t. there exists an optimal alignment with score  $D(i, j)$  passing through cell  $(n', r)$ .
- **Computation of  $M(i, j)$ :**
  - for  $i = n'$  and  $j = 1, \dots, m$ :  $M(n', j) = j$ ;
  - for  $i > n', 0 \leq j \leq m$ :  
 $M(i, j) = M(i', j')$ , where  $D(i, j)$  derives from cell  $(i', j')$   
—if there is more than one, choose acc. to priority (e.g. *left-diag-top*)

## Matrix $M$

- **Definition:** For  $i \geq n'$ , cell  $M(i, j)$  contains an index  $r$  s.t. there exists an optimal alignment with score  $D(i, j)$  passing through cell  $(n', r)$ .
- **Computation of  $M(i, j)$ :**
  - for  $i = n'$  and  $j = 1, \dots, m$ :  $M(n', j) = j$ ;
  - for  $i > n', 0 \leq j \leq m$ :  
 $M(i, j) = M(i', j')$ , where  $D(i, j)$  derives from cell  $(i', j')$   
—if there is more than one, choose acc. to priority (e.g. *left-diag-top*)
- Note that by definition  $(i', j') \in \{(i-1, j), (i-1, j-1), (i, j-1)\}$ .

## Matrix $M$

- **Definition:** For  $i \geq n'$ , cell  $M(i, j)$  contains an index  $r$  s.t. there exists an optimal alignment with score  $D(i, j)$  passing through cell  $(n', r)$ .
- **Computation of  $M(i, j)$ :**
  - for  $i = n'$  and  $j = 1, \dots, m$ :  $M(n', j) = j$ ;
  - for  $i > n', 0 \leq j \leq m$ :  
 $M(i, j) = M(i', j')$ , where  $D(i, j)$  derives from cell  $(i', j')$   
—if there is more than one, choose acc. to priority (e.g. *left-diag-top*)
- Note that by definition  $(i', j') \in \{(i-1, j), (i-1, j-1), (i, j-1)\}$ .
- Then  $M(n, m) = r$  s.t. there is an optimal alignment of  $s$  and  $t$  which passes through cell  $(\lceil n/2 \rceil, r)$ .

## Matrix $M$

- **Definition:** For  $i \geq n'$ , cell  $M(i, j)$  contains an index  $r$  s.t. there exists an optimal alignment with score  $D(i, j)$  passing through cell  $(n', r)$ .
- **Computation of  $M(i, j)$ :**
  - for  $i = n'$  and  $j = 1, \dots, m$ :  $M(n', j) = j$ ;
  - for  $i > n', 0 \leq j \leq m$ :  
 $M(i, j) = M(i', j')$ , where  $D(i, j)$  derives from cell  $(i', j')$   
—if there is more than one, choose acc. to priority (e.g. *left-diag-top*)
- Note that by definition  $(i', j') \in \{(i-1, j), (i-1, j-1), (i, j-1)\}$ .
- Then  $M(n, m) = r$  s.t. there is an optimal alignment of  $s$  and  $t$  which passes through cell  $(\lceil n/2 \rceil, r)$ .
- Thus, we can use the cut  $(n', r) = (\lceil n/2 \rceil, M(n, m))$  in the divide-step and recurse with  $s_1 \dots s_{n'}$  and  $t_1 \dots t_r$  on the left, and  $s_{n'+1} \dots s_n$  and  $t_{r+1} \dots t_m$  on the right.

Back to the example (p. 4): Here  $n = 5$ , thus  $n' = \lceil n/2 \rceil = 3$ . We compute the matrix  $M$  according to the priority *left-diag-top*:

$D(i,j)$			C	A	C	A						
		0	1	2	3	4						
	0	0	-1	-2	-3	-4						
G	1	-1	-1	-2	-3	-4						
A	2	-2	-2	1	0	-1	$M(i,j)$	0	1	2	3	4
A	3	-3	-3	0	0	2	3	0	1	2	3	4
G	4	-4	-4	-1	-1	1	4	0	0	2	2	4
A	5	-5	-5	-2	-2	1	5	0	0	0	2	2

So we have to recurse with  $r = 2$ , i.e. GAA,CA (left) and GA,CA (right).

For the left part GAA,CA, we have  $n' = \lceil n/2 \rceil = 2$ , and we get

$D(i,j)$			C	A					
		0	1	2					
	0	0	-1	-2					
G	1	-1	-1	-2	$M(i,j)$	0	1	2	
A	2	-2	-2	1	2	0	1	2	
A	3	-3	-3	0	3	0	0	1	

Thus,  $r = 1$  and we have to divide these at cut  $(2, 1)$ , yielding GA,C and A,A.

## Algorithm PWA(s,t)

1. if  $\max(|s|, |t|) \leq 2$ , then return an optimal alignment computed with N-W-algorithm
2. else
3.     for  $i = 0$  to  $n' - 1$  compute  $i$ 'th row of  $D$   
      (space-saving manner, row-wise)
4.     for  $i = \lceil n/2 \rceil$  to  $n$ , compute  $i$ 'th row of  $D$  and  $i$ 'th row of  $M$   
      (space-saving manner, row-wise)
5.      $r \leftarrow M(n, m)$
6.     return  $PWA(s_1 \dots s_{\lceil n/2 \rceil}, t_1 \dots t_r)$  concatenated  
      with  $PWA(s_{\lceil n/2 \rceil + 1} \dots s_n, t_{r+1} \dots t_m)$ .



# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$

# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$
- at any given time, there are the two matrices  $D$  and  $M$  to be computed

# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$
- at any given time, there are the two matrices  $D$  and  $M$  to be computed
- nothing needs to be stored for later, once we have computed  $r = M(n, m)$

# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$
- at any given time, there are the two matrices  $D$  and  $M$  to be computed
- nothing needs to be stored for later, once we have computed  $r = M(n, m)$
- thus for the matrix computations we need space  $O(m)$ ;

# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$
- at any given time, there are the two matrices  $D$  and  $M$  to be computed
- nothing needs to be stored for later, once we have computed  $r = M(n, m)$
- thus for the matrix computations we need space  $O(m)$ ;
- we need to store the partial alignments, whose total length is the length of the final alignment, thus  $O(n + m)$

# Analysis (1)

## Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always  $\leq m$
- at any given time, there are the two matrices  $D$  and  $M$  to be computed
- nothing needs to be stored for later, once we have computed  $r = M(n, m)$
- thus for the matrix computations we need space  $O(m)$ ;
- we need to store the partial alignments, whose total length is the length of the final alignment, thus  $O(n + m)$
- altogether space  $O(n + m)$

## Analysis (2)

### Time

- in the first iteration, we compute the entries of the two matrices  $D$  and  $M$ , each in constant time:  $\underbrace{(n+1)(m+1)}_D + \underbrace{\lceil n/2 \rceil(m+1)}_M$   
entries, so  $O(nm)$  time

## Analysis (2)

### Time

- in the first iteration, we compute the entries of the two matrices  $D$  and  $M$ , each in constant time:  $\underbrace{(n+1)(m+1)}_D + \underbrace{\lceil n/2 \rceil(m+1)}_M$  entries, so  $O(nm)$  time
- In each iteration, we are exactly halving the problem size (wherever we cut  $t$ , string  $s$  is always cut in the middle), thus we get:

$$nm + \frac{1}{2}nm + \frac{1}{4}nm + \dots \leq nm \sum_{k=0}^{\infty} \frac{1}{2^k} = 2nm \in O(nm).$$



## Analysis (2)

### Time

- in the first iteration, we compute the entries of the two matrices  $D$  and  $M$ , each in constant time:  $\underbrace{(n+1)(m+1)}_D + \underbrace{\lceil n/2 \rceil(m+1)}_M$  entries, so  $O(nm)$  time
- In each iteration, we are exactly halving the problem size (wherever we cut  $t$ , string  $s$  is always cut in the middle), thus we get:

$$nm + \frac{1}{2}nm + \frac{1}{4}nm + \dots \leq nm \sum_{k=0}^{\infty} \frac{1}{2^k} = 2nm \in O(nm).$$

Thus we doubled the time (asymptotically the same:  $O(nm)$ ), but reduced the space from quadratic to linear.