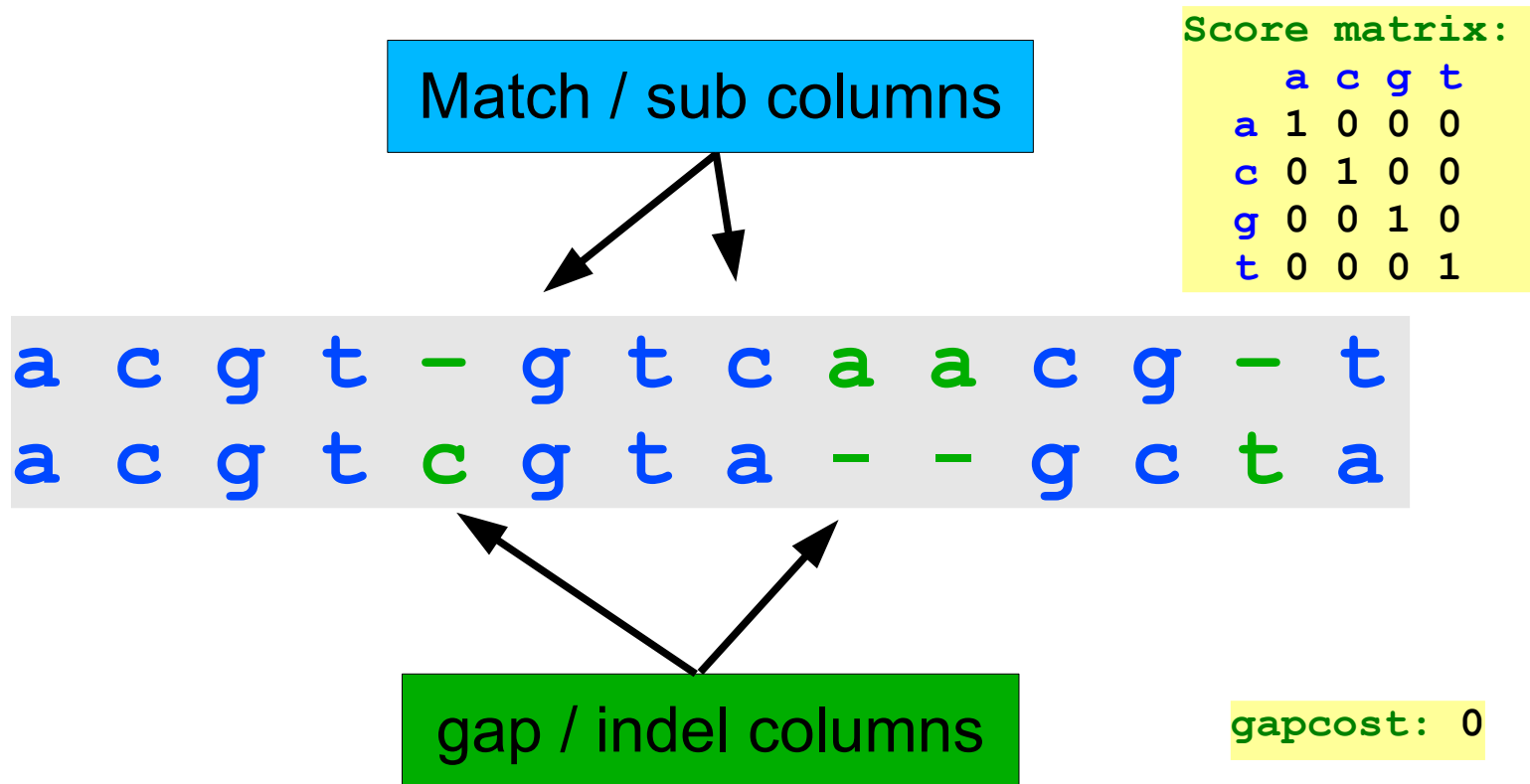


Alignment graph

Global alignment



Cost of alignment = “sum of the cost of each column”

Global alignment

Match / sub columns

Score matrix:

Score	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

a c g t - g t c a a c g - t
a c g t c g t a - - g c t a

Note about gap cost

In general: cost of "gap block" = $w(k)$, where k is the gap length

Our examples: $w(k) = 0 \cdot k$ "zero gap cost"

$w(k) = -1 \cdot k$ "linear gap cost"

Many programs: $w(k) = a + b \cdot k$ "affine gap cost"

gapcost: -1
gapcost: 0

Constructing an algorithm

Problem: What is the cost, $\text{Cost}(i, j)$, of an optimal alignment of

$A[1..i] = \text{acgtgtcaacgt}$ and $B[1..j] = \text{acgtcgtagcta}$

Solution: Consider the three possible rightmost columns, pick the best ...

acgtgtcaacg
acgtcgtagct

t
a

$\text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j])$

acgtgtcaacg
acgtcgtagcta

t
-

$\text{Cost}(i-1, j) + \text{gapcost}$

acgtgtcaacgt
acgtcgtagct

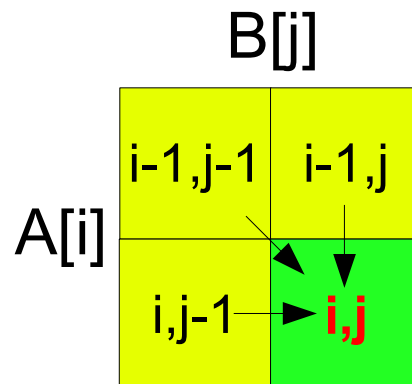
-
a

$\text{Cost}(i, j-1) + \text{gapcost}$

Immediately yields a recursive solution

Global alignment recursion

$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \text{ if } i=0 \text{ and } j=0 \end{cases}$$



Basic idea: Filling out a table

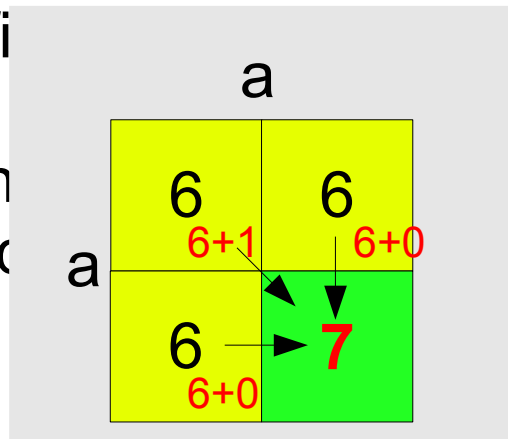
	a	c	g	t	c	g	t	a	g	c	t	a
a												
c												
g												
t												
g												
t												
c												
a												
a												
c												
g												
t												

	a	c	g	t	c	g	t	a	g	c	t	a
a	0	0	0	0	0	0	0	0	0			
c	0	1	1	1	1	1	1	1	1			
g	0	1	2	2	2	2	2	2	2			
t	0	1	2	3	3	3	3	3	3			
g	0	1	2	3	4	4	4	4	4			
t	0	1	2	3	4	4	5	5	5			
c	0	1	2	3	4	4	5	6	6			
a	0	1	2	3	4	5	6	6	6			
a	0	1	2	3	4	5	6	6	6			
a												
c												
g												
t												

	a	c	g	t	c	g	t	a	g	c	t	a
a	0	0	0	0	0	0	0	0	0	0	0	0
c	0	1	1	1	1	1	1	1	1	1	1	1
g	0	1	2	2	2	2	2	2	2	2	2	2
t	0	1	2	3	3	3	3	3	3	3	3	3
g	0	1	2	3	4	4	4	4	4	4	4	4
t	0	1	2	3	4	4	5	5	5	5	5	5
c	0	1	2	3	4	4	5	6	6	6	6	6
a	0	1	2	3	4	5	6	6	6	7	7	7
a	0	1	2	3	4	5	6	7	7	7	7	8
c	0	1	2	3	4	5	5	6	7	7	8	8
g	0	1	2	3	4	5	6	6	7	8	8	8
t	0	1	2	3	4	5	6	7	7	8	8	9

The algorithm essentially fills the table (using dynamic programming)

Observation: The value in a cell is the maximum of its three neighbor cells (left, upper-left, upper)



dynamic programming)

three of its neighbor cells (left, upper-left, upper) ...

The table can also be filled out row by row

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Exercise

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a
a	0			
g				
g				
t				

Exercise

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a
a	0	-1		
g				
g				
t				

Exercise

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2		
g					
g					
t					

Exercise

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	2

The optimal score, but what about an optimal alignment?

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	2

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	2

-
a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	-2

t -
t a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
g	-3	0	1	2	1
t	-4	-1	0	3	2

g t -
- t a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
g	-3	0	1	2	1
t	-4	-1	0	3	2

g g t -
c - t a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	2

a g g t -
a c - t a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **agg**t and **act**a using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Also optimal:

a g g t -
a - c t a

a g g t
a c t a

	a	c	t	a	
a	0	-1	-2	-3	-4
g	-1	2	1	0	-1
g	-2	1	2	1	0
t	-3	0	1	2	1
t	-4	-1	0	3	2

a g g t -
a c - t a

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

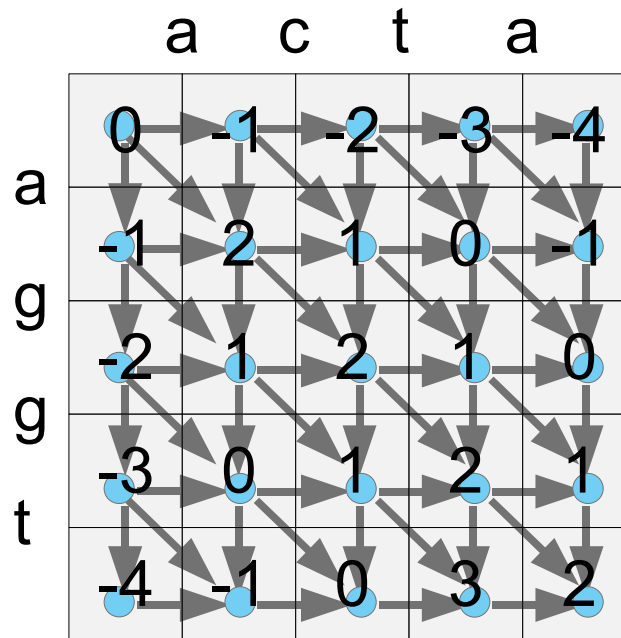
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



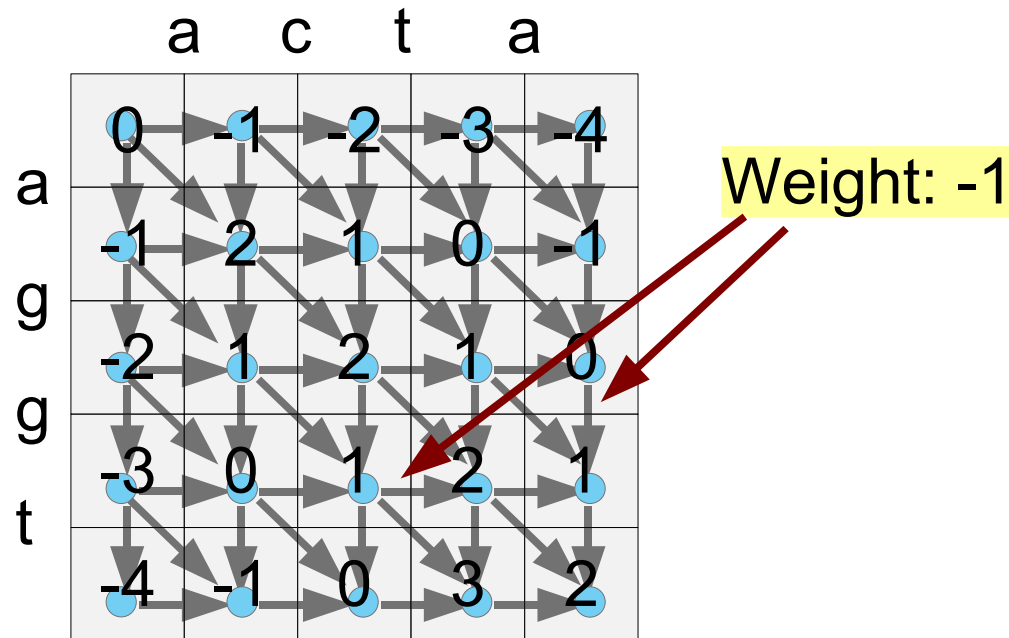
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



Alignment graph

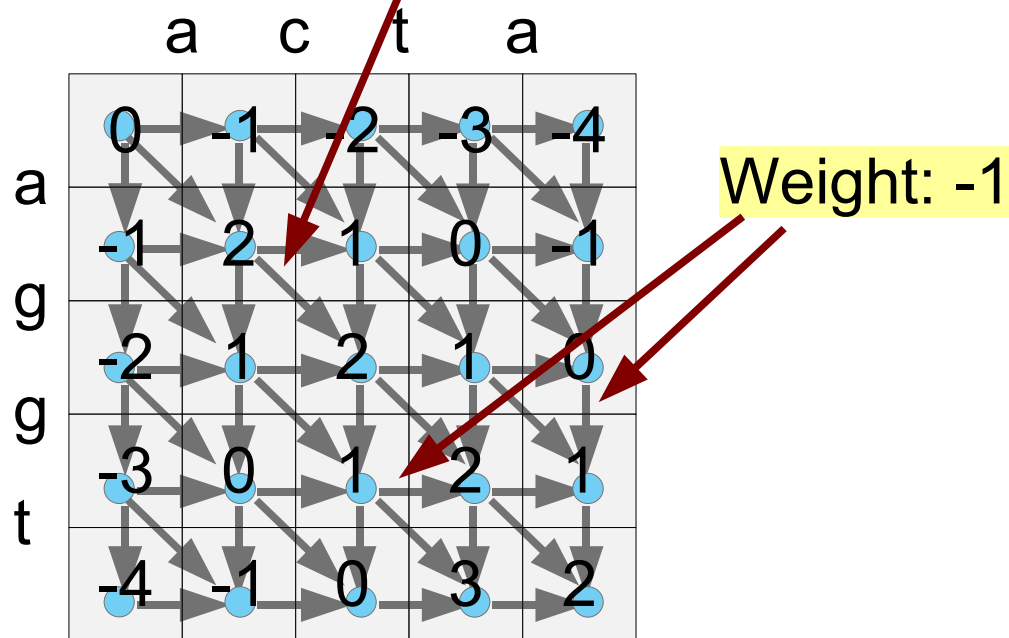
An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Weight: subcost(g,c)=0



Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

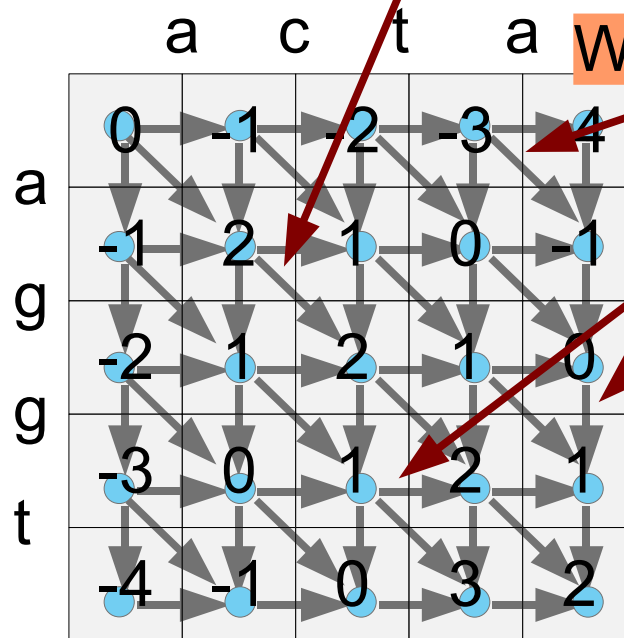
	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Weight: subcost(g,c)=0

Weight: subcost(a,a)=2

Weight: -1



Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

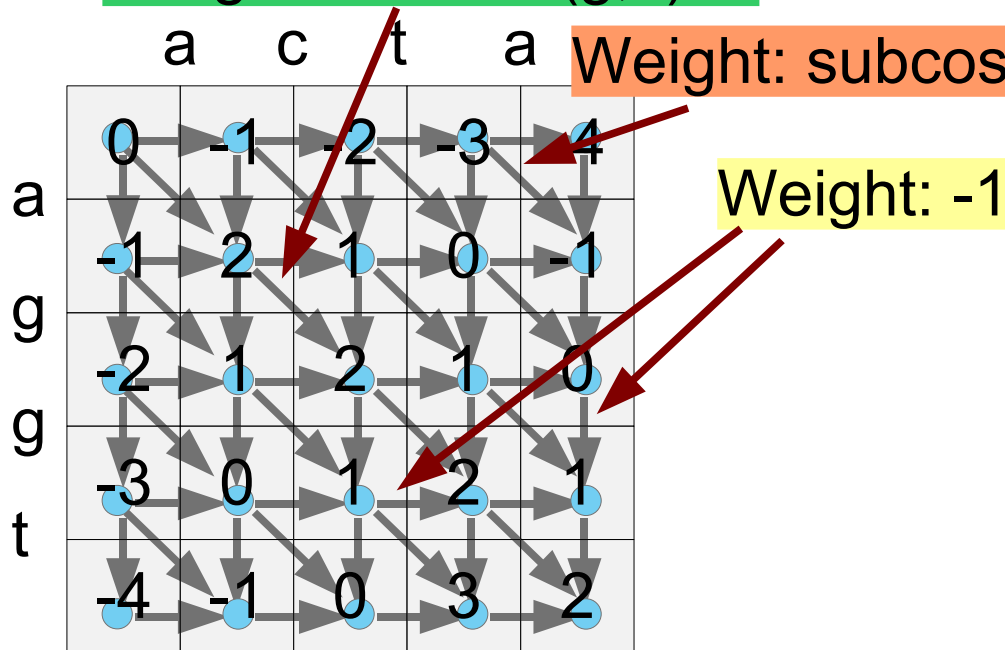
Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Weight: subcost(g,c)=0

Weight: subcost(a,a)=2



The cost of an optimal alignment is the length of a longest (shortest) path from (0,0) to (n,m), and the path yields the alignment ...

Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

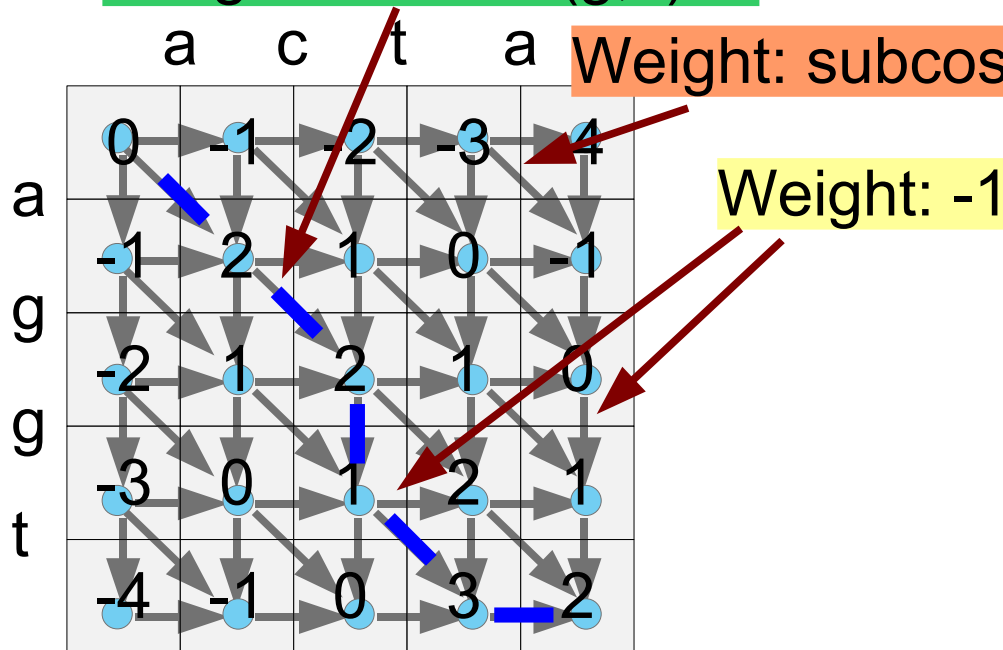
Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Weight: subcost(g,c)=0

Weight: subcost(a,a)=2

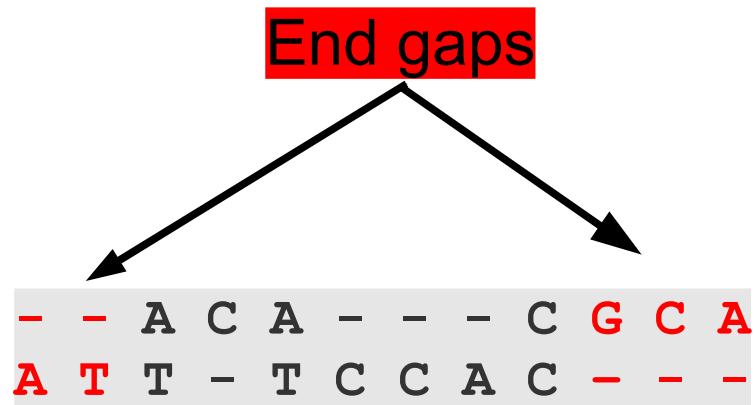


The cost of an optimal alignment is the length of a longest (shortest) path from (0,0) to (n,m), and the path yields the alignment ...

Exercises

Implement the simple algorithm for global alignment with backtracking, try to extend backtracking to print *all* optimal alignments.

Think about how to compute an optimal alignment when “end gaps” are free, i.e.



Now ...

What about general gapcost? affine gapcost?

Local alignment

Extension – Modeling gapcost

Biological observation: longer insertions and deletions (indels) are more common than shorter indels, i.e. a “good” alignment tends to have few long indels rather than many short indels ...

Can the simple algorithm for pairwise alignment be adapted to reflect this additional biological insight, i.e. a better model of biology?

Yes, we introduce the concept of a gapcost-function $g(k)$ which gives the cost/penalty for a block of k consecutive insertions or deletions ...

Example

A	T	A	C	A	-	-	-	C	G	C	A
A	T	-	C	T	C	C	A	C	-	C	T

$$s(A,A) + s(T,T) + g(1) + s(C,C) + s(A,T) + g(3) + s(C,C) + g(1) + s(C,C) + s(A,T)$$

Global alignment

Match / sub columns

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2



Note about gap cost

In general: cost of "gap block" = $g(k)$, where k is the gap length

Our examples: $g(k) = 0 \cdot k$ "zero gap cost"

$g(k) = -1 \cdot k$ "linear gap cost"

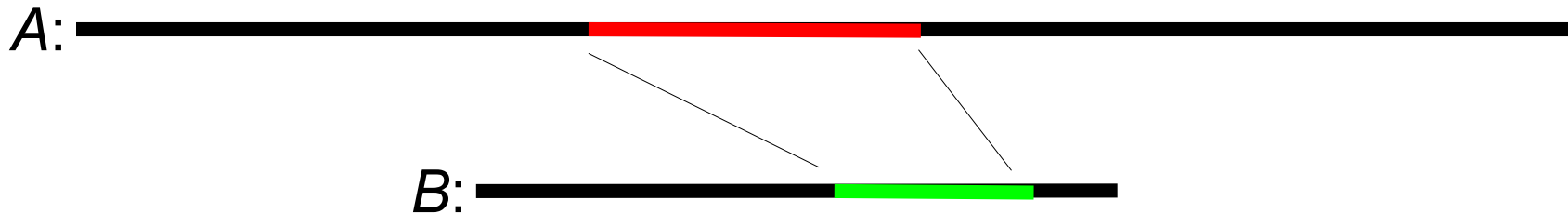
Many programs: $g(k) = a + b \cdot k$ "affine gap cost"

gapcost: -1
gapcost: 0

Local alignment

Objective:

Given two sequences A and B , a score matrix and a gap cost, find the pair of segments of A and B which has maximum alignment cost, i.e. maximum similarity.



Motivation: “... one must be alert to regions of similarity even when they occur embedded in an overall background of dissimilarity.” (Doolittle)

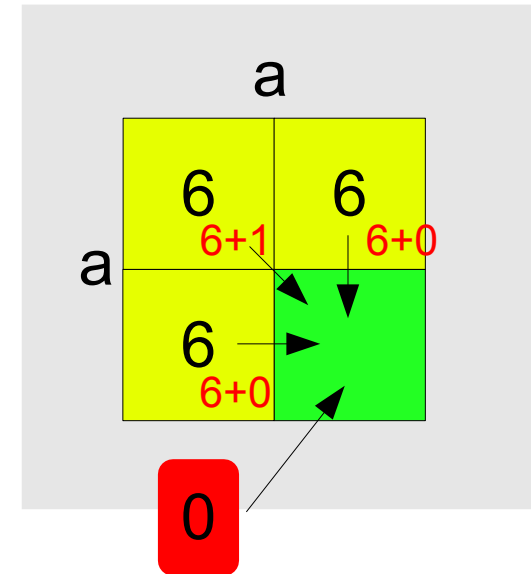
Seems more difficult than global alignment.

We have to align all pairs of segments?

Finding an optimal local alignment

Local alignment

$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \end{cases}$$



Fill out table (e.g. row by row), return maximum entry

Global alignment

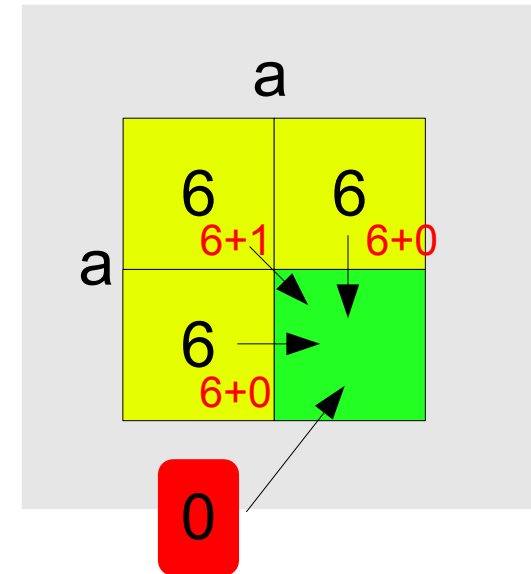
$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \text{ if } i=0 \text{ and } j=0 \end{cases}$$

Fill out table (e.g. row by row), return entry (n, m)

Finding an optimal local alignment

Local alignment

$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \end{cases}$$



Fill out table (e.g. row by row), return maximum entry

Note: If the score function is such that the cost of an alignment cannot be negative, then an local alignment is the same as a global alignment

Global alignment

$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i], B[j]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \text{ if } i=0 \text{ and } j=0 \end{cases}$$

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Fill out table (e.g. row by row), return entry (n,m)....