(Aspekte der Thermodynamik in der Strukturbiologie)

Einführung in die Bioinformatik

Wintersemester 2012/13
16:00-16:45 Hörsaal N100 B3

Peter Güntert

Alignment algorithms

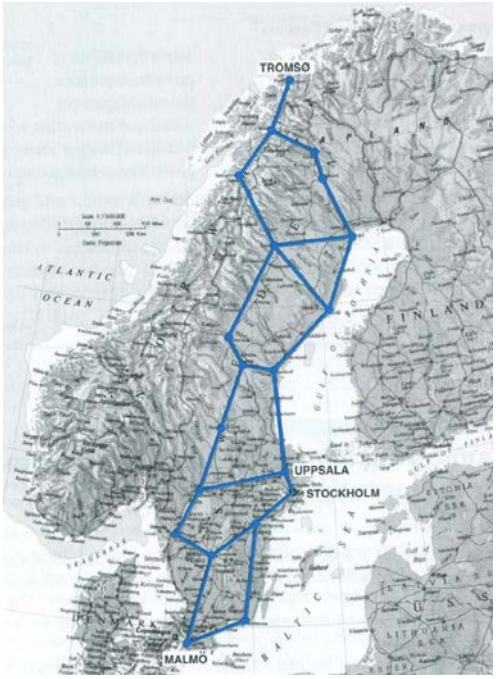
Outline

- Dynamic programming
- Global alignment algorithm of Needleman and Wunsch (1970)
- Local alignment algorithm of Smith and Waterman (1981)

Alignment algorithms

Possibles routes from Malmö to Tromsø.

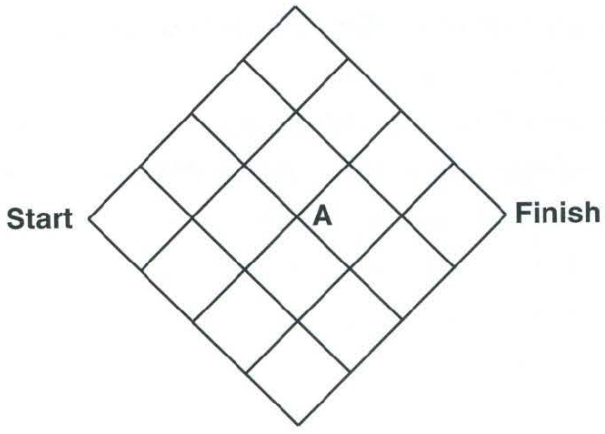
How can you determine an optimal route?



Blosum62 scoring matrix = cost function

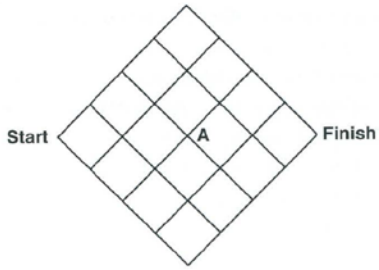
	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4																			
R	-1	5																		
N	-2	0	6																	
D	-2	-2	1	6																
C	0	-3	-3	-3	9															
Q	-1	1	0	0	-3	5														
E	-1	0	0	2	-4	2	5													
G	0	-2	0	-1	-3	-2	-2	6												
H	-2	0	1	-1	-3	0	0	-2	8									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

Dynamic programming



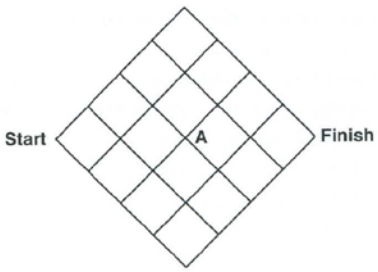
Consider possible paths from **Start** to **Finish** that pass through **A**.

Dynamic programming



- Consider first: how many paths from Start to Finish pass through A?
- There are six paths from Start to A.
- By symmetry, there are also six paths from A to Finish.
- Therefore, there are a total of 36 paths from Start to Finish through A.
- Assuming that we have assigned costs to the individual steps, do we have to check all 36 paths to find the path of minimum cost that goes from Start to Finish passing through A?

Dynamic programming



- No: the choice of the best path from A to Finish is independent of the choice of path from the Start to A.
- If we determine the best of the six paths from Start to A, and the best of the six paths from A to Finish, the best path from Start to Finish passing through A is: The best path from Start to A, followed by the best path from A to Finish.
  → No more than 12 of the 36 paths through A need to be considered.
- Even greater simplification is possible by systematically resubdividing the problem.
- The dynamic programming method for finding the optimal path through the matrix is based on this idea.

Global alignment with the algorithm of Needleman and Wunsch

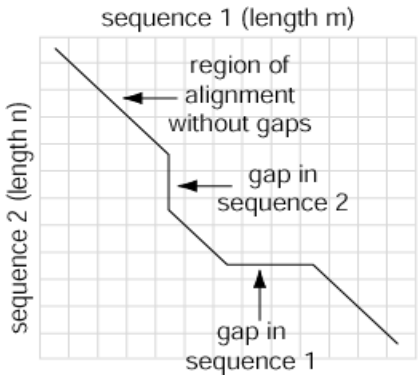
- Two sequences can be compared in a matrix along x- and y-axes.
- If they are identical, a path along a diagonal can be drawn.
- Find the optimal subpaths, and add them up to achieve the best score. This involves
  - adding gaps when needed
  - allowing for conservative substitutions
  - choosing a scoring system (simple or complicated)
- The Needleman-Wunsch algorithm is guaranteed to find optimal alignment(s).

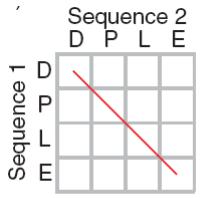
Three steps to global alignment with the Needleman-Wunsch algorithm

1. set up a matrix
2. score the matrix
3. identify the optimal alignment(s)

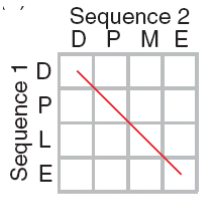
Four possible outcomes in aligning two sequences

1. identity
   (stay along a diagonal)
2. mismatch
   (stay along a diagonal)
3. gap in one sequence
   (move vertically!)
4. gap in the other
   sequence
   (move horizontally!)

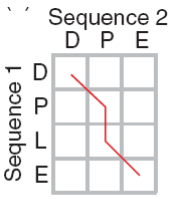




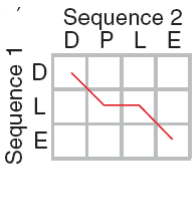
1 DPLE
2 DPLE



1 DPLE
2 DPME



1 DPLE
2 DP-E



1 D-LE
2 DPLE

Dynamic programming algorithm I

A statement of the optimal alignment problem and the dynamic programming solution are as follows: given two character strings, possibly of unequal length: $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$, where each a_i and b_j is a member of an alphabet set $\mathcal{A}$, consider sequences of edit operations that convert A and B to a common sequence. Individual edit operations include:

Substitution of b_j for a_i—represented (a_i, b_j).

Deletion of a_i from sequence A—represented (a_i, ϕ).

Deletion of b_j from sequence B—represented as (ϕ, b_j).

If we extend the alphabet set to include the null character ϕ: $\mathcal{A}^+ = \mathcal{A} \cup \{\phi\}$, a sequence of edit operations is a set of ordered pairs (x, y), with $x, y \in \mathcal{A}^+$.

A *cost function*, d, is defined on edit operations:

$d(a_i, b_j)$ = cost of a mutation in an alignment in which position i of sequence A corresponds to position j of sequence B, and the mutation substitutes $a_i \leftrightarrow b_j$.

$d(a_i, \phi)$ or $d(\phi, b_j)$ = cost of a deletion or insertion.

Define the minimum weighted distance between sequences A and B as

$$D(A, B) = \min_{A \to B} \Sigma d(x, y)$$

where $x, y \in \mathcal{A}^+$ and the minimum is taken over all sequences of edit operations that convert A and B into a common sequence.

Dynamic programming algorithm II

The problem is to find $D(A, B)$ and one or more of the alignments that correspond to it.

An algorithm that solves this problem, requiring execution time proportional to the product of the lengths of the two sequences, creates a matrix $\mathcal{D}(i, j), i = 0, \cdots n; j = 0, \cdots m$, such that $\mathcal{D}(i, j)$ is the minimal distance between the strings that consist of the first i characters of A and the first j characters of B. Then $\mathcal{D}(n, m)$ will be the required minimal distance $D(A, B)$.

The algorithm computes $\mathcal{D}(i, j)$ by recursion. The value of $\mathcal{D}(i, j)$ corresponds to the conversion of the initial subsequences $A_i = a_1 a_2 \cdots a_i$ and $B_j = b_1 b_2 \cdots b_j$ into a common sequence by L edit operations $S_k, k = 1, \cdots L$, which can be considered to be applied in increasing order of position in the strings. Consider *undoing* the last of these edit operations. The resulting truncated sequence of edit operations, $S_k, k = 1, \cdots L - 1$, is a sequence of edit operations for converting a substring of A_i and a substring of B_j into a common result. What is more, it must be an *optimal* sequence of edit operations for these substrings, for if some other sequence S'_k were a lower-cost sequence of operations for these substrings, then S'_k followed by S_L would be a lower-cost sequence of operations than S_k for converting A_i to B_j. Therefore, there should be a recursive method for calculating the $\mathcal{D}(i, j)$.

Dynamic programming algorithm III

Recognize the correspondence of steps between adjacent squares in the matrix, and individual edit operations (see Fig. 5.1):

$(i-1, j-1) \rightarrow (i, j)$	corresponds to the substitution $a_i \rightarrow b_j$.
$(i-1, j) \rightarrow (i, j)$	corresponds to the deletion of a_i from A.
$(i, j-1) \rightarrow (i, j)$	corresponds to the insertion of b_j into A at position i.

Sequences of edit operations correspond to stepwise paths through the matrix

$$(i_0, j_0) = (0, 0) \rightarrow (i_1, j_1) \rightarrow \cdots (n, m)$$

where $0 \leq i_{k+1} - i_k \leq 1$ (for $0 \leq k \leq n-1$), $0 \leq j_{k+1} - j_k \leq 1$ (for $0 \leq k \leq m-1$). Considering the possible sequences of edit operations and the corresponding paths through the matrix, the predecessor of an optimal string of edit operations leading from $(0,0)$ to (i,j), where $i, j > 0$, must be an optimal sequence of edit operations leading to one of the cells $(i-1, j), (i-1, j-1)$ or $(i, j-1)$; and, correspondingly, $\mathcal{D}(i,j)$ must depend only on the values of $\mathcal{D}(i-1, j), \mathcal{D}(i-1, j-1)$ and $\mathcal{D}(i, j-1)$, (together of course with the parameterization specified by the cost function d).

Dynamic programming algorithm IV

The algorithm is then as follows:
Compute the $(m+1) \times (n+1)$ matrix $\mathcal{D}$ by applying:

1. the initialization conditions on the top row and left column:

$$\mathcal{D}(i, 0) = \sum_{k=0}^{i} d(a_k, \phi)$$

$$\mathcal{D}(0, j) = \sum_{k=0}^{j} d(\phi, b_k)$$

These values impose the gap penalty on unmatched residues at the beginning of either sequence
and then

2. the recurrence relationships:

$$\mathcal{D}(i, j) = \min\{\mathcal{D}(i-1, j) + d(a_i, \phi), \mathcal{D}(i-1, j-1) + d(a_i, b_j), \mathcal{D}(i, j-1) + d(\phi, b_j)\}$$

Dynamic programming algorithm V

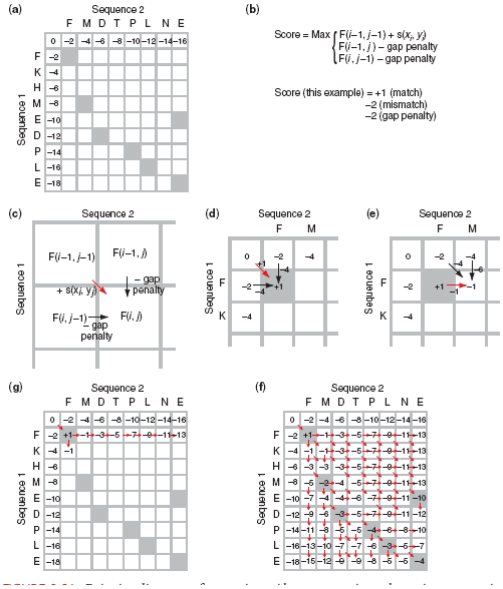
for $i = 1, \cdots n; j = 1, \cdots m$. This means: consider all three possible steps to $\mathcal{D}(i,j)$:

Operation	Cumulative cost
Insert a gap in sequence A	$\mathcal{D}(i-1,j) + d(a_i, \phi)$
Substitute $a_i \leftrightarrow b_j$	$\mathcal{D}(i-1,j-1) + d(a_i, b_j)$
Insert a gap in sequence B	$\mathcal{D}(i,j-1) + d(\phi, b_j)$

From these, choose the minimal value of the cumulative cost. For each cell, record not only the value $\mathcal{D}(i,j)$ but a pointer back to (one or more of) the cell(s) $(i-1,j)$, $(i-1,j-1)$ or $(i,j-1)$ selected by the minimization operation. Note that more than one predecessor may give the same value.

When the calculations are complete, $\mathcal{D}(n,m)$ is the optimal distance $D(A,B)$. An alignment corresponding to the sequence of edit operations recorded by the pointers can be recovered by tracing a path back through the matrix from (n,m) to (0,0). This alignment corresponding to the minimal distance $D(A,B) = \mathcal{D}(n,m)$ may well not be unique.

Fill in the matrix using "dynamic programming"



Fill in the matrix using "dynamic programming"

(**b**)

$$\text{Score} = \text{Max}\begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - \text{gap penalty} \\ F(i, j-1) - \text{gap penalty} \end{cases}$$

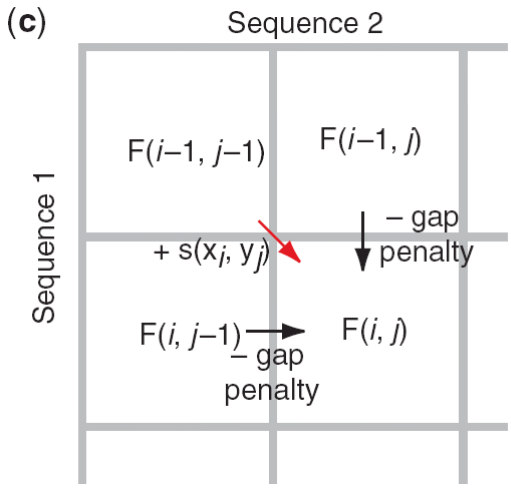
Score (this example) = +1 (match)
–2 (mismatch)
–2 (gap penalty)

Initialize with gap penalties

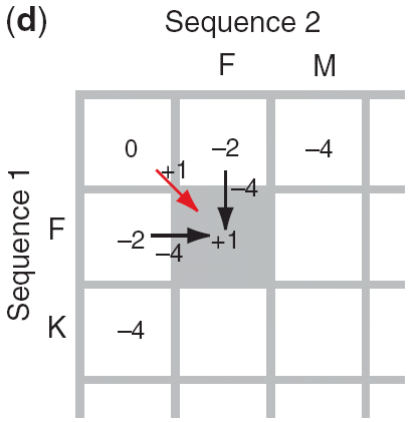
(**a**)

		Sequence 2								
			F	M	D	T	P	L	N	E
		0	–2	–4	–6	–8	–10	–12	–14	–16
Sequence 1	F	–2								
	K	–4								
	H	–6								
	M	–8								
	E	–10								
	D	–12								
	P	–14								
	L	–16								
	E	–18								

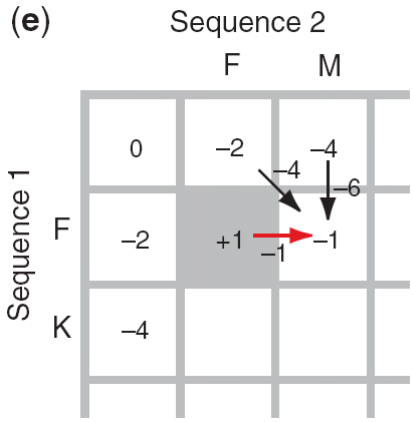
Fill in the matrix using "dynamic programming"



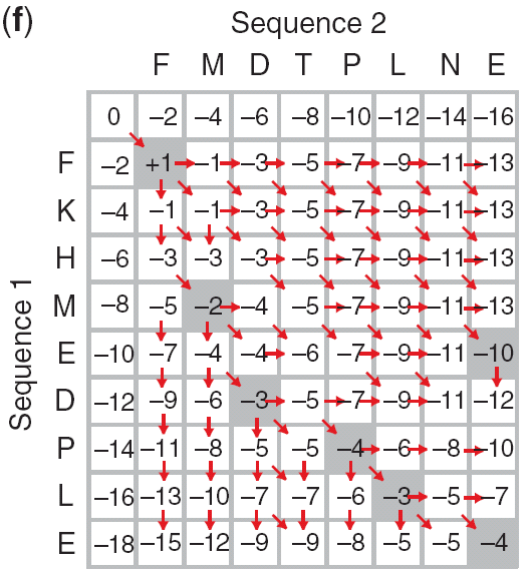
Fill in the matrix using "dynamic programming"



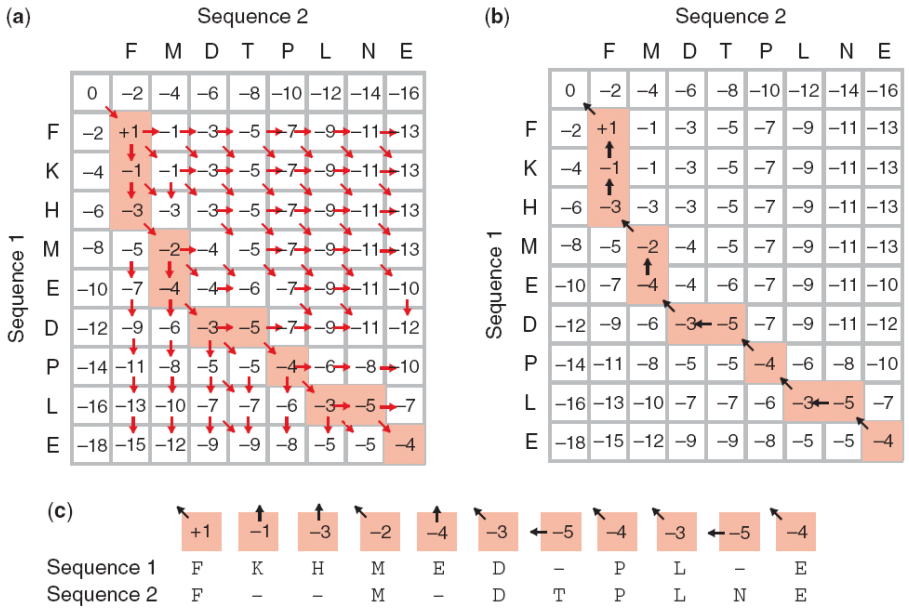
Fill in the matrix using "dynamic programming"



Fill in the matrix using "dynamic programming"



Traceback to find the optimal (best) pairwise alignment



Dynamic programming: Example II

Align the strings $A = \texttt{ggaatgg}$ and $B = \texttt{atg}$, according to the simple scoring scheme: match = 0, mismatch = 20, insertion or deletion = 25.

Here is the state of play after the top row and leftmost column have been initialized (italic), and the element in the second row and second column has been entered as **20** (boldface):

	ϕ	a	t	g
ϕ	*0*	*25*	*50*	*75*
g	*25*	**20**		
g	*50*			
a	*75*			
a	*100*			
t	*125*			
g	*150*			
g	*175*			

The value of **20** was chosen as the minimum of 25 + 25 (horizontal move, or insert gap into string atg), 0 + 20 (substitution a ↔ g) and 25 + 25 (vertical move, or insert gap into string ggaatgg). Because the substitution (the diagonal move) provided the minimal value, the cell containing 0 in the upper left-hand corner of the matrix is the predecessor of the cell in which we have just entered the 20. For traceback purposes, we would also draw an arrow from the value of 20 just entered, back to the 0 at the upper left. (If two or even three of the possible moves produce the same value, the resulting cell has multiple predecessors.)

Dynamic programming: Example II

Here is matrix after completion of the calculation:

	ϕ	a	t	g
ϕ	0	←25	←50	←75
	↑	↖	↖	↖
g	25	20	←45	50
	↑	↖↑	↖	↖
g	50	45	40	45
	↑	↖	↖↑	↖
a	75	50	65	60
	↑	↖↑	↖	↖↑
a	100	75	70	85
	↑	↑	↖	↖
t	125	100	75	90
	↑	↑	↑	↖
g	150	125	100	75
	↑	↑	↑	↖↑
g	175	150	125	100

It includes the traceback information in the form of arrows pointing from each cell to its predecessor(s). For some applications we may need only the value of $D(A, B)$ but not an alignment; if so, it is unnecessary to save the pointers. Boldface arrows delineate the paths of optimal alignment retracing a trail of predecessors from lower right, back to upper left. In some cases, one cell may show two predecessors. These correspond to alternative alignments with the same score.

There are two cells at which the traceback path branches. This gives a total of four optimal alignments with equal score:

```
ggaatgg    ggaatgg    ggaatgg    ggaatgg
---atg-    ---at-g    --a-tg-    --a-t-g
```

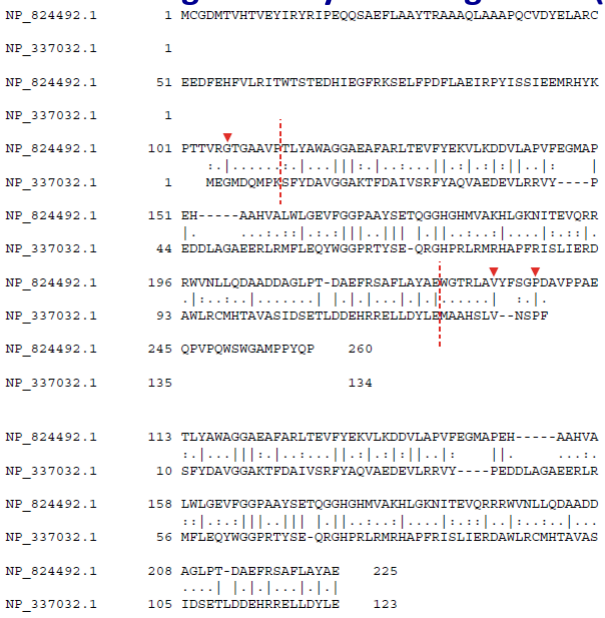
Needleman-Wunsch: dynamic programming

- N-W is guaranteed to find optimal alignments, although the algorithm does not search all possible alignments.
- It is an example of a dynamic programming algorithm: an optimal path (alignment) is identified by incrementally extending optimal subpaths.
- Thus, a series of decisions is made at each step of the alignment to find the pair of residues with the best score.

Global alignment versus local alignment

- Global alignment (Needleman-Wunsch) extends from one end of each sequence to the other.
- Local alignment finds optimally matching regions within two sequences ("subsequences").
- Local alignment is almost always used for database searches such as BLAST. It is useful to find domains (or limited regions of homology) within sequences.
- Smith and Waterman (1981) solved the problem of performing optimal local sequence alignment. Other methods (BLAST, FASTA) are faster but less thorough.

Global alignment (top) includes matches ignored by local alignment (bottom)



Global alignment: 15% identity

Local alignment: 30% identity

Local alignment algorithm of Smith-Waterman

- The Needleman-Wunsch algorithm determines the optimal global alignment of two sequences.
- It is inappropriate for detection of local regions of high similarity within two sequences, or for probing a long sequence with a short fragment, because it imposes gap penalties outside the similar regions.
- The method of T. Smith and M. Waterman solves this problem.
- Their modifications of the basic dynamic programming algorithm find optimal local alignments; that is, they select the substrings from both sequences that are most similar to each other. Their changes affect three parts of the algorithm.

Local alignment algorithm of Smith-Waterman

1. Initialization of the matrix—setting the values of the top row and left column. In the Smith–Waterman method, the top row and left column are set to 0. As a result, either sequence can slide along the other before alignment starts, without incurring any gap penalty against the residues it passes by.
2. Filling in the matrix In global alignment, at each step a choice is forced among match, insertion or deletion, even if none of these choices is attractive and even if a succession of unattractive choices degrades the score along a path containing a well-fitting local region. The Smith–Waterman method adds the fourth option: end the region being aligned.
3. Scoring and traceback The score of a global alignment is the number in the matrix element at the lower right. In the Smith–Waterman method it is the optimal value encountered, wherever in the matrix it appears. For global alignment, traceback to determine the actual alignment starts at the lower-right cell. In the Smith–Waterman method it starts at the cell containing the optimal value and continues back only as far as the region of local similarity continues.

The Smith–Waterman method would report a unique global optimum for our example:

```
ggaatgg
  atg
```

Note that no gaps appear outside the region matched.

How the Smith-Waterman algorithm works

Set up a matrix between two proteins (size m+1, n+1)

No values in the scoring matrix can be negative! S $\geq$ 0

The score in each cell is the maximum of four values:

[1] s(i-1, j-1) + the new score at [i,j] (a match or mismatch)
[2] s(i,j-1) – gap penalty
[3] s(i-1,j) – gap penalty
[4] zero

Smith-Waterman algorithm allows the alignment of subsets of sequences

Sequence 1 (length m)

Sequence 2 (length n)

		C	A	G	C	C	U	C	G	C	U	U	A	G
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
A	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
A	0.0	0.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.7
U	0.0	0.0	0.0	0.7	0.3	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.7
G	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.7	1.0	0.0	0.0	0.7	0.7	1.0
C	0.0	1.0	0.0	0.0	2.0	1.3	0.3	1.0	0.3	2.0	0.7	0.3	0.3	0.3
C	0.0	1.0	0.7	0.0	1.0	3.0	1.7	1.3	1.0	1.3	1.7	0.3	0.0	0.0
A	0.0	0.0	2.0	0.7	0.3	1.7	2.7	1.3	1.0	0.7	1.0	1.3	1.3	0.0
U	0.0	0.0	0.7	1.7	0.3	1.3	2.7	2.3	1.0	0.7	1.7	2.0	1.0	1.0
U	0.0	0.0	0.3	0.3	1.3	1.0	2.3	2.3	2.0	0.7	1.7	2.7	1.7	1.0
G	0.0	0.0	0.0	1.3	0.0	1.0	1.0	2.0	3.3	2.0	1.7	1.3	2.3	2.7
A	0.0	0.0	1.0	0.0	1.0	0.3	0.7	0.7	2.0	3.0	1.7	1.3	2.3	2.0
C	0.0	1.0	0.0	0.7	1.0	2.0	0.7	1.7	1.7	3.0	2.7	1.3	1.0	2.0
G	0.0	0.0	0.7	1.0	0.3	0.7	1.7	0.3	2.7	1.7	2.7	2.3	1.0	2.0
G	0.0	0.0	0.0	1.7	0.7	0.3	0.3	1.3	1.3	2.3	1.3	2.3	2.0	2.0

Performing global and local pairwise alignment

http://www.ebi.ac.uk/Tools/psa/

Rapid, heuristic versions of Smith-Waterman: FASTA and BLAST

- Smith-Waterman is very rigorous and it is guaranteed to find an optimal alignment.
- But Smith-Waterman is slow. It requires computer space and time proportional to the product of the two sequences being aligned (or the product of a query against an entire database).
- Gotoh (1982) and Myers and Miller (1988) improved the algorithms so both global and local alignment require less time and space.
- FASTA and BLAST provide rapid alternatives to S-W.

Unterlagen zur Vorlesung

http://www.bpc.uni-frankfurt.de/guentert/wiki/index.php/Teaching