

# Introduction to Computational Biology

## Lecture # 3: Sequences Alignment, part A

Nadav Rappoport

17/11/2008

### 1 The problem

There is a variety of sequences that we would like to align like DNA, RNA, proteins, sequence of angles in a proteins ect. Given 2 sequences, Can we tell how much are they similar? Can we show the similarity?

For example given these two sequences:

1. GCGCATGGATT
2. TGCGCCATTGATG

Are these sequences similar?

### 2 Sequence Alignment

We have to define a similarity measurement. The measurement can change depending on the situation. For example, if we look for a specific sequence in the same organism, we should use another measurement than if we look for a similarity in another organism. Figure 1 shows an example of two DNA sequences we would like to align and two possible alignments.

(a)

s:	A	A	T	C	G	G	A	T
t:	A	T	T	C	G	A	G	T

(b)

s:	A	A	T	C	G	-	G	A	T
t:	A	T	T	C	G	A	G	-	T

(c)

s:	A	A	-	-	T	C	G	G	A	T
t:	-	-	A	T	T	C	G	A	G	T

Figure 1: (a)Two DNA sequences we would like to align. (b)One option to align these sequences. (c)Another option to align the sequences.

There are three kind of alignment in each place:

1. *Match* which is an identity.

2. *Indel* is the alignment of a letter to the sign '-' (gap). Indel stands for insertion/deletion, meaning a base was inserted or deleted in one of the sequences in comparison to the other. Note that it is possible to insert gaps also at the beginning or at the end of the sequences.
3. *Mismatch* is just two characters that differ from each other, but are aligned.

Now we want a score function that gives bonuses and penalties. For example:

- Match: +1
- Mismatch: -1
- Indel : -2

The score of an alignment will be the sum of positional scores. Notice that different score functions can give different optimal alignments.

Let's describe the score function formally.  $\Sigma$  stand for the alphabet. For example, if we align DNA then  $\Sigma = \{A, C, G, T\}$

Then we define  $\Sigma^+ = \Sigma \cup \{-\}$ .

Finally the function looks like:

$$\sigma : \Sigma^+ \times \Sigma^+ \longrightarrow \mathbb{R}$$

A question in computer science:

Given two sequences  $s$  and  $t$ , find alignment  $A$  s.t.  $Score(A) = \text{Max}_{\text{all possible alignments } A'} \{Score(A')\}$ .

Is it a complicate problem or not?

Intuitively we would say it's a complicate problem, because there is an exponential possibilities for alignments. But we will give a solution using dynamic programming that won't take exponential time.

## 2.1 The Algorithm

Given two sequences:  $\vec{s} = s_1 \dots s_n$  and  $\vec{t} = t_1 \dots t_m$

Our matrix will look like this: Each cell  $V(i, j) = \text{The cost of the best alignment of } s_1 \dots s_n \text{ and } t_1 \dots t_m$

And formally:

$$V(i, j) = \max \begin{cases} \sigma(s_i, t_j) + V(i-1, j-1) \\ \sigma(s_i, -) + V(i-1, j) \\ \sigma(-, t_j) + V(i, j-1) \end{cases}$$

And the base cases are:

$$\begin{aligned} V(0, 0) &= 0 \\ V(i, 0) &= \sum_{k=1}^i \sigma(s_k, -) \\ V(0, j) &= \sum_{k=1}^j \sigma(-, t_k) \end{aligned}$$

When an algorithm like this is proposed we have to:

1. Prove or convince correctness. In our case to proof that the best alignment is  $V(i, j)$  (Here the proof is trivial, because the construction is the proof).
2. To show that what we get in the end is answering to the question. In our case, that  $Score(A) = V(|s|, |t|)$
3. To stand on the time and space complexity. This algorithm's time and space complexity will be compute later.

For example, let's look on the alignment of AGC against AAAC with the scoring alignment function we saw before (Match +1, Mismatch -1, Indel -2). The matrix will be:

		$\vec{S} = (s_1 \dots s_n)$			
			A	G	C
		0	1	2	3
$\vec{t} = (t_1 \dots t_m)$	0	0	-2	-4	-6
	A 1	-2	1	-1	-3
	A 2	-4	-1	0	-2
	A 3	-6	-3	-2	-1
	C 4	-8	-5	-4	-1

MATCH ↖

GAP<sub>s</sub> ↑

GAP<sub>t</sub> ←

Figure 2: An example of a scoring matrix for the alignment of AGC against AAAC which it's best score is -1.

The best score is the value in  $V(|s|, |t|)$ . Now we know the best score, but we don't know which alignment gave that score. In order to find the best alignment we record which case in the recursive rule maximized the score and trace back the path which corresponds to the best alignment. In other words, if we look at the last point of the path  $(m, n)$ , for every path that arrives to it, the last step in the path is one of three:

1. Diagonal, which represent a match.
2. Vertical, which represent a gap in sequence  $s$ .
3. Horizontal, which represents a gap in sequence  $t$ .

The path's score is the score of that last step plus the score of the path to get to the point where the last step begins. Figure 3 shows the reconstruction of the best alignment of the two sequences  $s$  and  $t$ .

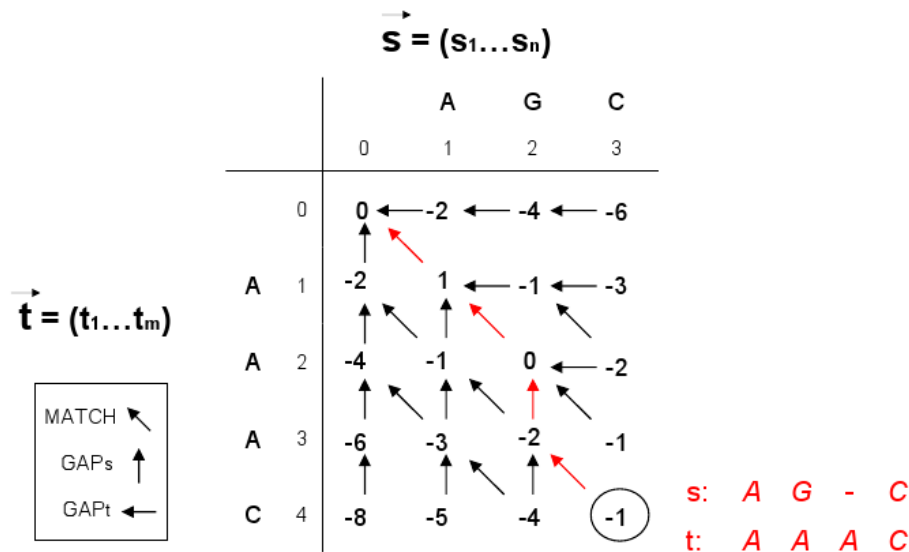


Figure 3: The red arrows represents one option of the best alignments. Note that sometimes, more than one alignment has the best score, like here there are 3 best alignments.

## 2.2 Space and Time Complexity

Filling the whole matrix, and finding the alignment take  $3 \cdot m \cdot n = O(m \cdot n)$  (for each cell we check three options). The space used is  $2 \cdot m \cdot n = O(m \cdot n)$  (we have to visit each cell and save both the choice we made and the cost). This could be acceptable in the case of two short sequences, but when comparing two whole genomes we will run out of space.

To reduce the amount of memory used we notice that in order to fill one column in the matrix all we need is the column that was before it. That way, at each point of the algorithm all we need is to remember  $2 \cdot m$  cells. Now when we'll reach the last column we'll have the correct score in the last cell. But how can we discover the alignment? If in each cell we will "remember" the best alignment till here, it won't help us saving place, because we need to "remember"  $n$  paths each one is  $n + m$  long.

### 2.2.1 Divide and Conquer

**Observation:** We aligned the sequences from left to right, but an alignment from right to left will give exactly the same alignment.

Let's assume that we know at which row (let  $l$  be that row) the best alignment passes through the middle column of the matrix. If so, we can split the dynamic programming problem into two parts: from  $(0, 0)$  to  $(l, \frac{n}{2})$ , and from  $(l+1, \frac{n}{2}+1)$  to  $(m, n)$ . The optimal alignment for the whole matrix will be the concatenation of the optimal alignments for these two separate sub matrices. Note that two sub-sequences alignments will take only half the space of the matrix. Continuing the process this way for each of the sub matrices will result in  $\log(n)$  steps and we'll have all the points through which the best alignment goes.

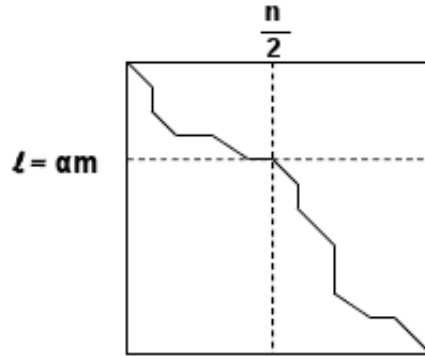


Figure 4: Divide and conquer. It is possible to see the 2 new matrices we'll continue out with after we have found the point where the best alignment passes through the middle column.

Now we have another problem: Finding the index of  $t$  that give the middle of  $s$  in the best alignment. We can write the score of the best alignment that goes through  $l$  as:

$$V\left(s\left[1 \dots \frac{n}{2}\right], t\left[1 \dots l\right]\right) + V\left(s\left[\frac{n}{2} + 1 \dots n\right], t\left[l + 1 \dots m\right]\right)$$

Now we need to compute these quantities for all values of  $l$  and take the value which gives us the maximal alignment and that will be the index.

We'll define another matrix:

$$W(i, j) = \max \begin{cases} \sigma(s_i, t_j) + W(i + 1, j + 1) \\ \sigma(s_i, -) + W(i + 1, j) \\ \sigma(-, t_j) + W(i, j + 1) \end{cases}$$

$W(i, j)$  is computed in the same manner as  $V(i, j)$ , going backward from  $W(m, n)$ .

Now, for a certain point  $(i, j)$ ,  $V(i, j) + W(i, j)$  is the score of the best alignment that goes through that point.

Before we'll describe the algorithm, we'll define  $A \odot B$  as a diagonal concatenating of the two matrices  $A$  and  $B$ .

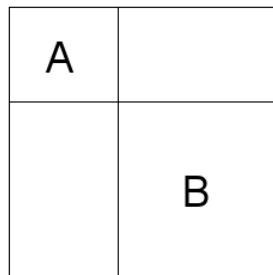


Figure 5: Concatenating two matrices. The top left corner of B is attached to the bottom right corner of A.

Now we can describe the Space-Efficient-Algorithm (SEA):

$SEA(\vec{s}, \vec{t})$

1. Compute  $V(*, \frac{n}{2})$
2. Compute  $W(*, \frac{n}{2})$
3.  $l^* = \operatorname{argmax}_l \{V[l, \frac{n}{2}] + W[l, \frac{n}{2}]\}$   
 $\vec{s}_1 = s_1 \dots s_{\frac{n}{2}}, \vec{t}_1 = t_1 \dots t_{l^*}$  and  $\vec{s}_2 = s_{\frac{n}{2}+1} \dots s_n, \vec{t}_2 = t_{l^*+1} \dots t_m$

4. return  $SEA(\vec{s}_1, \vec{t}_1) \odot SEA(\vec{s}_2, \vec{t}_2)$

### 2.2.2 Space and Time Complexity

Spaced complexity: Every column of  $V(i, j)$  can be calculated if its prior column is known, and same goes for  $W(i, j)$ . Therefore the amount of space needed in each of the first 2 steps is  $2 \cdot m$ . The third step doesn't use additional space and the fourth step uses the previous space. So the total space adds up to  $O(m)$ .

Time complexity: Each of the two first steps takes  $O(\frac{n}{2} \cdot m)$  and the third step takes  $O(m)$ . In order to calculate the fourth step we'll first define  $c(n, m)$  as the time needed for  $SEA(s, t)$ . We'll also define  $l = \alpha m$  while  $0 \leq \alpha \leq 1$ . Now, the time needed in the fourth step is:  $c(\frac{n}{2}, \alpha \cdot m) + c(\frac{n}{2}, (1 - \alpha) \cdot m)$ . The time needed in the first three steps of  $c(\frac{n}{2}, \alpha \cdot m) + c(\frac{n}{2}, (1 - \alpha) \cdot m)$  is  $\frac{n}{2} \cdot m + m$ . Because  $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{\log_2 n}} \leq 2$  we get that the total running time is:  $c(n, m) = n \cdot m + m + c(\frac{n}{2}, \alpha \cdot m) + c(\frac{n}{2}, (1 - \alpha) \cdot m) \leq 2 \cdot m \cdot n = O(m \cdot n)$ . Therefore, the running time is still  $O(m \cdot n)$  but the space used is reduced to  $O(m)$ .

## 3 Local Alignment

Sometimes we would like to find similarity of subsequences. For example, if we have a protein known as having ATP binding domain, and we want to verify if another protein has the same domain. We want to find:

$$\max_{i_1, i_2, j_1, j_2} \text{Score}(s_{i_1} \dots s_{i_2}, t_{j_1} \dots t_{j_2})$$

### 3.1 Smith-Waterman algorithm

This problem seems to be harder because there is more flexibility. But we can do three changes in the Global alignment algorithm and use that. The changes are:

1.  $V(i, j) =$  The cost of the best **local alignment from  $(s_1, t_1)$  till  $(s_i, t_j)$**

$$2. V(i, j) = \max \begin{cases} \sigma(s_i, t_j) + V(i-1, j-1) \\ \sigma(s_i, -) + V(i-1, j) \\ \sigma(-, t_j) + V(i, j-1) \\ 0 \end{cases}$$

Taking the option 0 corresponds to starting a new alignment. If the best alignment up to some point has a negative score, it is better to start a new one rather than extend the old one.

3. And the base cases are:

$$\begin{aligned} V(0, 0) &= 0 \\ V(i, 0) &= \max_i \{ \sigma(s_i, -) + V(i-1, 0), 0 \} \\ V(0, j) &= \max_j \{ \sigma(-, t_j) + V(0, j-1), 0 \} \end{aligned}$$

Note that because of this option, the top row and the left column will be filled with zeros.

In the *Local alignment*, the alignment can end anywhere in the matrix, so instead of taking the value in the bottom right corner  $V(m, n)$ , for the best score, we look for the highest value over the whole matrix. Then we start the trace back from there. The trace back ends when we meet a cell with value 0.

An example of the way the algorithm works can be seen in Figure 6. It iterates over all the local alignment in the matrix and finds the best one.

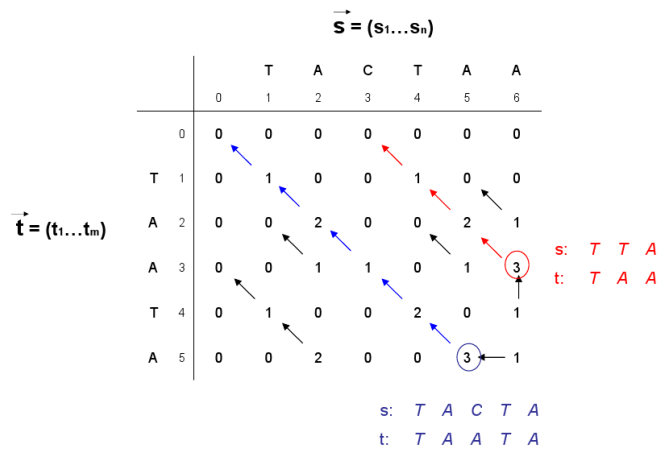


Figure 6: An example of a scoring matrix which is a result of the Smith-Waterman algorithm. We can see that the best score is 3 and the arrows show two paths that led to the two high scores. We used the same scoring function (Match +1, Mismatch -1, Indel -2).

### 3.2 Complexity

In This case we cannot save space, because the alignment doesn't have to pass through the middle, so we have to save the whole matrix. So the time and space complexity are the same as the global alignment before we saved space. That means that time and space complexity are  $O(m \cdot n)$ .

## 4 What remain

- Where do we take the scoring function from? How do we calculate the scores?
- $O(m \cdot n)$  is not useful for long genomes. We want an algorithm more efficient. We will look for a heuristic algorithm, which maybe doesn't find the best alignment, but will work fast. There will be algorithms that by making some assumptions will run faster.
- When we find a maximum score, we have to verify if it's interesting, related to a random sequence.