Computational Biology
Lecture 4: Overlap detection, Local Alignment, Space Efficient Needleman-Wunsch

Saad Mneimneh

Overlap detection: Semi-Global Alignment

An overlap of two sequences is considered an alignment where start and end gaps are ignored. This is also called semi-global alignment because we are globally aligning the two sequences but ignoring trailing gaps at both extremities. First we need to make clear what we mean by start and end gaps. Start gaps are gaps that occur before the first character in a sequence. Similarly, end gaps are gaps that occur after the last character of a sequence. Therefore, if the two sequences are x and y, start and end gaps can be either in x or in y. If the optimal overlap (or semi-global alignment) of x and y is to be computed, the basic Needleman-Wunsch algorithm will not work. Consider the following example of aligning $x = CAGCACTTGGATTCTCGG$ and $y = CAGCGTGG$.

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
```

The above alignment is not an optimal global alignment. The score, as computed by the basic algorithm, is $1(6) - 1(1) - 2(12) = -19$. The optimal global alignment:

```
CAGCACTTGGATTCTCGG
CAGC----G--T----GG
```

gives a score of -12. However, the first alignment is an optimal semi-global alignment. If start and end gaps are ignored, the score becomes 3. When detecting the optimal overlap of x and y, the possibilities are:

- a suffix of x aligns with a prefix of y;
- a suffix of y aligns with a prefix of x;
- y aligns with a substring of x.
- x aligns with a substring of y.

The figure below shows some examples.

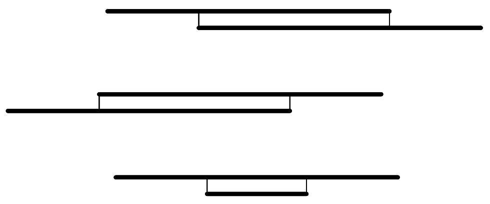

Figure 1: Different overlaps

We would like to modify the basic Needleman-Wunsch algorithm to compute the optimal semi-global alignment where start and end gaps are ignored. Such an alignment is useful for potential overlap detection. If we use Needleman-Wunsch, the negative score resulting from gaps at the extremities will not reflect the fact that the two sequences have a considerable overlap.

How should we modify our basic algorithm. Well, let us first look at the start gaps. Since we would like not to penalize start gaps, this can be accounted for by initializing the first row and first column of the dynamic programming table to zeros. This is to say that he part of the alignment that starts with gaps in x or gaps in y is given a score of zero. Figure 2 illustrates the modification to the basic Needleman-Wunsch initial step.

Also, gaps at the end of the alignment should be ignored when computing the optimal score. For this, no more modifications are needed in the dynamic programming table. $A(i, j)$ represents the score of the optimal alignment of $x_1...x_i$ and $y_1...y_j$ (now with start gaps ignored). Hence $A(m, j)$ is the score corresponding to optimally aligning x with $y_{1..j}$. In other words, it is the score of optimally aligning x with a length j prefix of y.

Since the optimal semi-global alignment can - theoretically - be obtained using any prefix alignment, the optimal alignment score is now detected as the maximum value on the last row (when x is aligned with any prefix of y) or column (when y is aligned with any prefix of x).

	-	y_1	y_2	$\cdots$	y_n
-	0	$-g$	$-2g$	$\cdots$	$-ng$
x_1	$-g$				
x_2	$-2g$				
$\vdots$	$\vdots$				
x_m	$-mg$				

=>

	-	y_1	y_2	$\cdots$	y_n
-	0	0	0	$\cdots$	0
x_1	0				
x_2	0				
$\vdots$	$\vdots$				
x_m	0				

Figure 2: Modified initialization for semi-global alignment

	-	y_1	y_2	$\cdots$	y_n
-					$\uparrow$
x_1					.
x_2					.
$\vdots$					.
x_n	$\leftarrow$	.	.	.	.

Figure 3: Taking maximum of last row and last column

Therefore, the best score is now in $A(i,j)$ such that $A(i,j) = \max_{k,l}(A(k,n), A(m,l))$ and the alignment itself can be obtained by tracing back from $A(i,j)$ to $A(0,0)$ as before.

Variations of this optimal alignment algorithm can accommodate situations where gaps are ignored only for x and/or y or only at the start or end of the sequence. The following table summarizes the changes needed to the semi-global alignment algorithm.

Places where gaps are not penalized	Action
Start of x	Initialize first row to zeroes
End of x	Look for max in last row
Start of y	Initialize first column to zeroes
End of y	Look for max in last column

Table 1: General semi-global alignment

Local Alignment

Definition: A local alignment between two strings x and y is an alignment between a substring of x and a substring of y.

We would like to compute the optimal local alignment between x and y. Why is such an alignment useful? Genes are shuffled in different genomes, so similar genes are not going to be detected by globally aligning the two sequences. Moreover, protein sequences tend to have preserved repetitive patterns that we might want to identify; therefore, an optimal local alignment will produce the highest scoring repetitive pattern.

There is a simple way of obtaining the highest score local alignment. Enumerate all pairs of substrings of x and y and compute a global alignment for each pair, then choose the highest scoring one. Since a string of length l contains $O(l^2)$ substrings, this algorithm will have an $O(m^2n^2).O(mn) = O(m^3n^3)$ time. This will be very inefficient. We would like to again modify our basic algorithm to compute the optimal local alignment.

Since we are interested in local alignments (alignment of substrings), we need to change the definition of the matrix A to reflect that notion. Recall that in the case of the basic algorithm the entry $A(i,j)$ of the matrix A represents the highest score for aligning $x_1...x_i$ with $y_1...y_j$. Now $A(i,j)$ will be defined as the highest score of aligning some suffix of $x_1...x_i$ with some suffix of $y_1...y_j$. This takes care of the locality characteristic. Now we need to look as the update rules.

Let us identify some properties of a local alignment. A local alignment cannot be negative, since aligning an empty substring of x with an empty substring of y gives a score of zero. Therefore, the score of the optimal local alignment has to be at least zero (practically speaking, it is always positive unless x and y do not have any character in common, in which case we are not interested in a local alignment).

In the optimal local alignment, the two substrings have to match at both ends. Otherwise, we can just ignore the start and/or the end, and obtain a better local alignment.

More generally, combining the two observations above, we can state the following: The optimal local alignment cannot start or end with an alignment with a negative score (same reasoning). The statement of course assumes that the additive property holds. In particular:

Local alignment property: the optimal alignment has no prefix alignment with negative score.

Therefore, negative entries in our matrix A are now meaningless. The update rules are still valid, except that a negative entry should not be used. If $A(i,j)$ is negative, then by definition of $A(i,j)$, a local alignment between any suffix of $x_1...x_i$ and any suffix of $y_1...y_j$ has a negative score. But this cannot be optimal according to our property above. This is because it is better to align the empty suffix of $x_1...x_i$ with the empty suffix of $y_1...y_j$ resulting in a score of 0.

Therefore, $A[i,j]$'s score would be (exhaustively) computed now as one of four (as opposed to three before) cases:

- align x_i with y_j and add the score to $A(i-1,j-i)$
- align x_i with a gap and add the score to $A(i-1,j)$;
- align a gap with y_j and add the score to $A(i,j-1)$;
- align a gap with a gap (empty suffixes for both $x_1...x_i$ and $y_1...y_j$ are now considered, a score of zero).

Since $A(i,j)$ represents the best local alignment for any suffixes of $x_1...x_i$ and $y_1...y_j$, and also empty suffixes can be considered, it follows that $A(i,j)$ cannot be negative.

Another intuitive way of looking at this is the following: since a local optimal alignment of x and y cannot have a negative prefix alignment, then a negative $A(i,j)$ cannot contribute to $A(i+1,j+1)$, $A(i+1,j)$, and $A(i,j+1)$. The algorithm will force $A(i,j)$ to be zero as if $A(i+1,j+1)$, $A(i+1,j)$, and $A(i,j+1)$ are starting from the beginning ignoring the previously negative part.

The resulting update rule for $A(i,j)$ is:

$$A(i,j) = \max \begin{cases} A(i-1,j-1)+s(i,j), & Ptr(i,j) = diag \\ A(i,j-1)-d, & Ptr(i,j) = left \\ A(i-1,j)-d, & Ptr(i,j) = up \\ \mathbf{0}, & Ptr(i,j) = null \ (no\ pointer) \end{cases}$$

Hence one change to the basic algorithm is the fact that a 0 is selected in case the other three cases generate a negative score.

Another difference is that an optimal local alignment does not necessarily include the last symbols of x and y. To reflect this, the result is a maximum value entry for the whole matrix A.

In order to find a maximal local alignment, pointers are followed starting with a maximum value entry until it is reached a position where there are no pointers out. An alternative is to stop as soon as a 0 is reached.

Here's an example:

		T	A	C	G
	0	**0**	**0**	**0**	**0**
A	**0**	**0**	↖1	**0**	**0**
A	**0**	**0**	↖1	↖0	**0**
C	**0**	**0**	**0**	↖2	**0**

Figure 4: Local alignment example

In the example above the best score (2) is found at $A(3,3)$ which corresponds to aligning AC with AC.

Saving Space for Needleman-Wunsch Algorithm

As previously stated, the time and space complexities of the basic optimal alignment Needleman-Wunsch algorithm are both $O(mn)$. While there are not known algorithms to perform faster than $O(mn)$ while keeping the same generality, space complexity can be improved. This is especially helpful when a large amount of data is to be processed.

We will present a variation on the basic algorithm that improves the space complexity to $O(m+n)$ at the expense of only roughly doubling the time.

The values in one row depend only on the values in that row and those in the previous row. Therefore, by keeping only two rows, the same computation can be carried out leading to $A(m,n)$. The time is the same as before, but space is now reduced to $O(n)$ for the two rows. If we consider the space needed to store x and y, the total space requirement is $O(m+n)$.

The main difference is that pointers to previous cannot be kept anymore. The question is now: can the optimal alignment itself be determined while still using linear space? This can be achieved using a divide and conquer strategy.

First of all, note that for a given prefix $x_1...x_i$, we can compute in linear space the scores of optimally aligning $x_1...x_i$ with any prefix of y. These correspond to row i of the matrix A. Similarly, for a given suffix $x_i....x_m$, we can compute in linear space the scores of optimally aligning $x_i....x_m$ with any suffix of y. This can be done by reversing the sequences x and y and computing row $(m-i+1)$ of the matrix A^R, where R stands for reverse sequences.

The crucial step is the following. For a given x_i we can determine in linear space where x_i should go in an optimal alignment. There are $2n+1$ possibilities:

- x_i aligns with some y_j for $j = 1...n$
- x_i aligns with a gap between y_j and y_{j+1} for $j = 0...n$ (here we abuse the notation a little bit for j to signify a gap at the beginning when $j = 0$ and a gap at the end when $j = n+1$).

The following figure illustrate the two cases:

x_1	x_2	$\cdots$	x_{i-1}	x_i	x_{i+1}	$\cdots$	x_m
y_1	y_2	$\cdots$	y_{j-1}	y_j	y_{j+1}	$\cdots$	y_n

x_1	x_2	$\cdots$	x_{i-1}	x_i	x_{i+1}	$\cdots$	x_m
y_1	y_2	$\cdots$	y_j	-	y_{j+1}	$\cdots$	y_n

In the first case, for a given j, the score of the optimal alignment is

$$OPT(x_1...x_{i-1}, y_1...y_{j-1}) + s(i,j) + OPT(x_{i+1}...x_m, y_{j+1}...y_n)$$

and in the second case, for a given j, the score of the optimal alignment is

$$OPT(x_1...x_{i-1}, y_1...y_j) - d + OPT(x_{i+1}...x_m, y_{j+1}...y_n)$$

where OPT denotes the score of the optimal alignment between two sequences.

The optimal alignment is the one with the highest score among the $2n+1$ possibilities above. The good thing is that all of the $2n+1$ scores can be computed in linear space using row $(i-1)$ of A and row $(m-i+1)$ of A^R.

For a given j,

$$OPT(x_1...x_{i-1}, y_1...y_{j-1}) + s(i,j) + OPT(x_{i+1}...x_m, y_{j+1}...y_n) = A(i-1, j-1) + s(x_i, y_j) + A^R(m-i, n-j)$$

and

$$OPT(x_1...x_{i-1}, y_1...y_j) - d + OPT(x_{i+1}...x_m, y_{j+1}...y_n) = A(i-1, j) - d + A^R(m-i, n-j)$$

Determining a maximum among the above $2n+1$ possibilities reveals a possible position for x_i in the alignment. Therefore, for given x and y and i, we can obtain the position of x_i in an optimal alignment for x and y in $O(mn)$ time and $O(m+n)$ space.

This suggest the following divide and conquer algorithm:

- compute A and A^R
- pick $x_{\lceil \frac{m}{2} \rceil}$
- find max $\begin{cases} A(\lceil \frac{m}{2} \rceil - 1, j-1) + s(x_{\lceil \frac{m}{2} \rceil}, y_j) + A^R(m - \lceil \frac{m}{2} \rceil, n-j), & j = 1...n \\ A(\lceil \frac{m}{2} \rceil - 1, j) - d + A^R(m - \lceil \frac{m}{2} \rceil, n-j), j = 0...n \end{cases}$
- figure out the position of $x_{\lceil \frac{m}{2} \rceil}$ in the optimal alignment according to the max above
- if $x_{\lceil \frac{m}{2} \rceil}$ is aligned with y_j for some j, recursively solve for the two optimal alignments of $x_1...x_{\lceil \frac{m}{2} \rceil - 1}$ with $y_1...y_{j-1}$ and $x_{\lceil \frac{m}{2} \rceil + 1}...x_m$ with $y_{j+1}...y_n$
- if $x_{\lceil \frac{m}{2} \rceil}$ is aligned with a gap between y_j and y_{j+1} for some j, recursively solve for the two optimal alignments of $x_1...x_{\lceil \frac{m}{2} \rceil - 1}$ with $y_1...y_j$, and $x_{\lceil \frac{m}{2} \rceil + 1}...x_m$ with $y_{j+1}...y_n$

The space requirement for the algorithm above is clearly the space needed to compute one row of A and one row of A^R for each of the subproblems. This however can be done using the same $O(m+n)$ space over and over. Therefore, we have a linear space algorithm. What about the time? An $O(mn)$ time is spent on for the first iteration only for figuring out the position of $x_{\lceil \frac{m}{2} \rceil}$. Could that time bound be exceeded asymtotically? The answer is no. Let $T(m,n)$ be the time we spend on one subproblem involving sequences of length m and n. Then we can define $T(m,n)$ recursively as $T(m,n) = cmn + T(\frac{m}{2}, k) + T(\frac{m}{2}, n-k)$. This is because we need $O(mn)$ time to compute A and A^R and $O(n)$ time to compute the max above, and we end up with two subproblems of size (m,k) and $(m,n-k)$ for some k depending on where $x_{\lceil \frac{m}{2} \rceil}$ aligns. This recurrence can be solved by the substitution method. Guess that $T(m,n) \leq 2cmn$. Then $T(m,n) \leq cmn + 2c\frac{m}{2}k + 2c\frac{m}{2}(n-k) = cmn + 2c\frac{m}{2}n = 2cmn$, verified.

References

Setubal J., Meidanis, J., Introduction to Molecular Biology, Chapter 3.