2. Homology and Multiple Sequence Alignment

EEOB563

Spring 2025

1 Homology

Definition: Homology is the relationship of two characters that have descended, usually with divergence, from a common ancestral character.

If interested, consult Walter Fitch's article for some of the common problems with the usage of homology and its associated terms.

2 Sequence alignment

The generation of alignment is a common task in bioinformatics and is used for structure prediction, sequence comparison, database searching, and phylogenetic analysis. A sequence alignment is a hypothesis about homology (as is the choice of sequences for it). Note, that it is impossible (in most cases) to know the true alignment and it is difficult to generate an optimal alignment. To carry out an automatic alignment, it is necessary to quantify biochemical similarity. We can do it by considering genetic code, chemical properties, or empirical datasets.

The pairwise alignment is usually accomplished by dynamic programming. If all sequences would be of the same length (no indels) an alignment would be a trivial task. However, the sequences are usually of different lengths, because of the insertion and deletion mutations (indels). Accommodating such mutation requires placing gaps in some sequences, which is the most difficult part of the alignment process. We do it by introducing a gap penalty $GP = g + e(l - 1)$, where g=gap opening cost, e=gap extension cost, l=length of the gap. Note, that there is no mathematical, statistical, or biological justification for this formula. Just a common sense ;) Also note that there are 10179 possible alignments for two sequences of length 300!

2.1 Dynamic programming

Dynamic programming was formalized in 1950's by mathematician Richard Bellman, who was working at the time on optimal decision processes at the RAND corporation supervised by the secretary of defense. It is based on the Bellman's principle of optimality: every subsolution of an optimal solution is itself and optimal solution.

The first example we discussed in class comes from : "*Imagine a climber trying to climb on top of a wall. A wall is constructed out of square blocks of equal size, each of which provides one handhold. Some handholds are more dangerous or complicated than others. From each block the climber can reach three blocks of the row righ above: one right on top, one to the right and one to the left* *(unless right or left are no available because that is the end of the wall). The goal is to find the least dangerous path from the bottom of the wall to the top, where danger rating (cost) of a path is the sum of danger ratings (costs) of blocks used on that path."*

We represent this problem as follows. The input is an nxm grid, in which each cell has a positive cost $C(i, j)$ associated with it. The bottom row is row 1, the top row is row n. From a cell (i, j) in one step you can reach up to three cells: $i+1, j-1$ if $j > 1$; $i+1, j$; and $i+1, j+1$ if $j < m$. Here is an example of an input grid:

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

j=1————————>m

The first step is to define an array to hold intermediate values. In our case we define an array, where A(i,j) is the cost of the least dangerous (cheapest) path from any given cell to the top. The first row is easy (it's the same as in the original grid):

2	8	9	5	8

j=1————————>m

For the second row we have to consider several solutions and chose the one that is the cheapest. We also have to account for the fact that some cells are on the leftmost or on the rightmost sides of the grid. For example, for the two cells below we need to consider the sums: 4+2, 4+8, 4+9 for the first, but only 3+5 and 3+8 for the second:

2	8	9	5	8
	[4]			[3]

j=1————————>m

Mathematically, we can write it down as:

$$x = \begin{cases} C_{(i,j)} + min\{A_{(i-1,j-1)}, A_{(i-1,j)}\} & \text{if } j = m \\ C_{(i,j)} + min\{A_{(i-1,j)}, A_{(i-1,j+1)}\} & \text{if } j = 1 \\ C_{(i,j)} + min\{A_{(i-1,j-1)}, A_{(i-1,j)}, A_{(i-1,j+1)}\} & \text{if } j \neq 1 \text{ and } j \neq m \end{cases}$$

So we get:

2	8	9	5	8
4+2	4+2	6+5	2+5	3+5

j=1——————————->m

And we repeat the same procedure with the next row, *etc.*, keeping track of where our solution comes from:

2	8	9	**5**	8
6	6	11	**7**	8
11	13	11	13	**8**
14	13	16	**12**	16

j=1——————>m

The best path with the total cost of 12 is marked in bold. Note that a greedy approach – choosing the lowest cost cell at every step – would not yield an optimal solution: if we start from cell (1, 2) with cost 2, and choose a cell with minimum cost at every step, we can at the very best get a path with total cost 13.

The beauty of this algorithm is that it allows us to find the optimal solution without the need of considering all possibilities.

Now, consider an example with two amino-acid sequences:

Sequence 1

Sequence 2

	*	G	R	Q	T	A	G	L
*	0	-8	-8	-8	-8	-8	-8	-8
G	-8	6	-2	-2	-2	0	6	-4
T	-8	-2	-1	-1	5	0	-2	-1
A	-8	0	5	-1	0	4	0	-1
Y	-8	-3	-2	-1	-2	-2	-3	-1
D	-8	-1	-2	0	-1	-2	-1	-4
L	-8	-4	-2	-2	-1	-1	-4	4

We have two sequences arranged at the margins of a table and substitution scores for different amino acids based on the BLOSUM62 matrix. We also use a simple linear gap penalty of -8. To allow for end gaps, we use a complete gap column and a complete gap row marked by *.

In the first step of the algorithm, we need to find optimal alignments for smaller subsequences (individual cells in our table). It's easy for the first row and column of the table, we just adding gap penalty as we are sliding one sequence relative to the other:

		Sequence 1							
		*	G	R	Q	T	A	G	L
Sequence 2	*	0	-8	-16	-24	-32	-40	-48	-56
	G	-8							
	T	-16							
	A	-24							
	Y	-32							
	D	-40							
	L	-48							

For the rest of the cells, we consider three possibilities and chose the highest score:

- we align an amino acid in seq1 to an amino acid in seq2 and inherit the cost of the previously aligned sequence found in the cell on the left one column and up one row;
- we introduce a gap in sequence #1 at cost -8 and inherit the cost of the previously aligned sequence in the cell on the left;
- we introduce a gap in sequence #2 at cost -8 and inherit the cost of the previously aligned sequence in the cell on the top;

Mathematically, we can write it like this:

$$S_{(i,j)} = max \begin{cases} S_{(i-1,j-1)} + d(A_i, B_i) \\ S_{(i-1,j)} - g \\ S_{(i,j-1)} - g \end{cases}$$

where $d(A_i, B_i)$ is the score defined by the matrix. Notice, that we only use the substitution score when we move diagonally, because we are aligning to a gap in other cases.

After we complete the table, we find the best score in the last row and trace back the optimal alignment (**Do this on your own!**): (shown in bold):

Sequence 1

Sequence 2	*	G	R	Q	T	A	G	L
*	**0**	-8	-16	-24	-32	-40	-48	-56
G	-8	**6**	**-2**	-10	-18	-26	-34	-42
T	-16	-2	**5**	**-3**	-5	-13	-21	-29
A	-24	-10	-3	4	**-3**	-1	-9	-17
Y	-32	-18	-11	-4	2	**-5**	-4	-10
D	-40	-26	-19	-11	-3	0	**-6**	-8
L	-48	-34	-27	-19	-11	-4	-4	**-2**

Notice, that by introducing a strong end-gap penalty, we very much guarantee that the ends of the sequences will be aligned.

2.2 Global vs. local alignments:

A couple of fancy names will be introduced here, which you can use to impress your friends. The ideas behind them, however, are simple and reflect the historical development of the algorithm discussed above.

The alignment algorithm described above (and on pages pages 76-79 of Lemey, Salemi and Vandamme) was introduced by Needleman & Wunsch in 1970 and is known as the Needleman-Wunsch algorithm.

“Needle”, a program in the EMBOSS package, will produce such optimal “global” alignment for you and can be used to measure similarity between two sequences.

NW algorithm creates a global alignment with the max score. But short and highly similar subsequences may be missed because they can be overweight by the rest of the sequence. A different algorithm was introduced by Smith and Waterman (1981) that finds an alignment that determines the longest/best subsequence pair that gives the maximum degree of similarity between the two original sequences. “Water”, another program in the EMBOSS package will calculate the optimal local alignment.

The **Smith-Waterman algorithm** says:

$$S_{(i,j)} = max \begin{cases} S_{(i-1,j-1)} + d(A_i, B_j) \\ S_{(i-1,j)} - g \\ F_{(i,j-1)} - g \\ 0 \end{cases}$$

and in this case SM,N is the optimal local alignment score. Notice that the only difference between the two methods is a “0” in the SW algorithm. This 0 will stop the alignment process if the score becomes negative.

A heuristic approach based on the Smith-Waterman algorithm is used in the BLAST (Basic Local Alignment Search Tool) program.

Since both algorithms require a scan of an MxN matrix, they are both $O(MN)$ or **of complexity** $O(N^2)$ for sequences of equal length.

2.3 Multiple sequence alignment:

To align more than two sequences, we could use dynamic programming. For three sequences our condition would look something like this:

$$S_{(i,j,k)} = max \begin{cases} S_{(i-1,j-1,k-1)} + d(A_i, B_j, C_k) \\ S_{(i,j-1,k-1)} + d(-, B_j, C_k) - g \\ S_{(i-1,j,k-1)} + d(A_i, -, C_k) - g \\ S_{(i-1,j-1,k)} + d(A_i, B_j, -) - g \\ S_{(i-1,j,k)} + d(A_i, -, -) - g \\ S_{(i,j-1,k)} + d(-, B_j, -) - g \\ S_{(i,j,k-1)} + d(-, -, C_k) - g \end{cases}$$

But this algorithm is $O(N^3)$. In general, for k sequences, the dynamic programming algorithms are $O(N^k)$ exponential in k. This makes dynamic programming too complex for any reasonable number of sequences.

Instead we use other multiple alignment strategies:

Progressive multiple sequence alignment steps:

- construct a guide tree using Neighbor-joining or UPGMA based on pairwise similarity scores
- use dynamic programming to align the two most similar sequences
- once two sequences have been aligned to each other, combine them into a single profile
- align the next most similar sequence to that profile using the sum of pairs score and dynamic programming
- continue until all sequences have been aligned

Clustalw/x is an example of a purely progressive multiple alignment program.

Progressive alignment is much faster than dynamic programming, but it is not guaranteed to find the optimal alignment. The quality of the alignment depends on the quality of the first pairwise alignment. Error can be propagated through the progression and the resulting alignments are prone to be local optima.

To improve the quality of progressive alignment, we can use iterative refinement methods. In this approach,

- the sequences are divided into two sets which are each aligned using progressive alignment;
- The resulting alignments are then aligned and scored using the sum of pairs score or some other objective function;
- The resulting alignment is used to reconstruct the guide tree, and the process is repeated until the score no longer improves.

Muscle and **MAFFT** are programs that use iterative strategies.

T-Coffee and **MAFFT** also use consistency-based scoring to perform a multiple alignment. In this approach a heuristic search is employed to find the multiple alignment that directly maximizes the overall sum of pairs score or some other objective function.