1 Scoring gaps in alignments more accurately

Last week we talked about algorithms for global and local alignments. In the scoring system we discussed there was a constant penalty for inserting/deleting a base. In this lesson we will discuss models that give different penalties for starting a gap and for enlarging it. Let’s start with some definitions to clear the discussion:

Definition 1.1 Space - an insertion/deletion of one base.

Definition 1.2 Gap - a maximal sequence of spaces

Our target is to define a scoring system that looks at gaps and not spaces. In some biological situations we can expect large gaps. For example, alternative splicing may result in large gaps. When comparing cDNA we would like to penalize the excision of an exon without giving the length of the exon a big impact on the score.

1.1 Models for scoring gaps

1.1.1 The Constant model

In the constant model every gap receives a constant score regardless of its length. Lets denote Ngaps - number of gaps, Wg - penalty for each gap. The score function is:

$$\sum_{i,j} \sigma(s[i],t[j]) + Wg \cdot N_{gaps}$$

The time complexity is $O(m \cdot n)$ if we calculate the score according to the Needleman-Wunsch algorithm. (see last week’s lesson for details)

1.1.2 The Affine model

The affine model is one of the common methods. In this model we give a certain penalty (denoted Wg) for starting a gap, and a different, usually smaller, penalty for enlarging the gap by one base (denoted Ws). The score function is:

$$\sum_{i,j} \sigma(s[i],t[j]) + Wg \cdot N_{gaps} + Ws \cdot N_{spaces}$$

To calculate the best alignment for this model we will use an algorithm similar to that discussed last week, with the difference that now we have to find a way to differentiate between opening a gap and enlarging it. To do so the algorithm will work with 3 matrices defined as following:

- $A(i, j)$ - optimal score for an alignment of $S_1 \cdots S_i$ and $T_1 \cdots T_j$ that ends with no gaps.
Figure 1: An automaton describing the legal transitions between the matrices. We assume that the penalty for mismatches is less than the penalty for opening two gaps and expanding them. Therefore we will never open a gap in one sequence immediately following a gap in the other. This explains why there are no legal transitions between B and C.

- $B(i, j)$ - optimal score for an alignment of $S_1 \cdots S_i$ and $T_1 \cdots T_j$ that ends with a gap at S.

- $C(i, j)$ - optimal score for an alignment of $S_1 \cdots S_i$ and $T_1 \cdots T_j$ that ends with a gap at T.

So now our recursion rule becomes:

$$
A(i, j) = \max \left\{ \begin{array}{l}
A(i - 1, j - 1) + \sigma(i, j) \\
B(i - 1, j - 1) + \sigma(i, j) \\
C(i - 1, j - 1) + \sigma(i, j)
\end{array} \right.
$$

$$
B(i, j) = \max \left\{ \begin{array}{l}
A(i - 1, j) + W_g + W_s \\
B(i - 1, j) + W_s
\end{array} \right.
$$

$$
C(i, j) = \max \left\{ \begin{array}{l}
A(i, j - 1) + W_g + W_s \\
C(i, j - 1) + W_s
\end{array} \right.
$$

the initiation is

$$
A(i, 0) = B(i, 0) = W_g + i \cdot W_s
$$

$$
A(0, j) = C(0, j) = W_g + j \cdot W_s
$$

$$
B(0, j) = C(i, 0) = -\infty
$$

$$
A(0, 0) = 0
$$

and the best score is

$$\max (A(n, m), B(n, m), C(n, m))$$

In order to trace back the alignment, instead of having each cell hold an arrow to the cell it came from, each cell will point to the matrix it came from.

The time complexity is $O(m \cdot n)$ (again following Needleman-Wunsch).

### 1.1.3 The Convex model

Spaces tend to occur in alignments in groups. This is due to the fact that insertions/deletions are usually block events. Therefore if we have a space in the sequence, the probability of the next character to be also a space is higher. To
model this phenomenon, in the convex model each additional space in a gap contributes less to the gap weight than the preceding space. A common score function for this model is:

$$\sum_{i,j} \sigma(s[i], t[j]) + Wg \cdot N_gaps + \sum_l \log SizeOfGap(l)$$

We can write in each cell how many spaces led to it. Thus, when calculating the score for each cell we don’t need to go back and count spaces, but simply calculate $$\log(x)$$. The time complexity of calculating $$\log(x)$$ is $$\log(x)$$ (divide x in 2 $$\log(x)$$ times). Under the assumption of $$m < n$$ and since $$x \leq n$$ then the time complexity is $$O(m \cdot n \log(n))$$

### 2 Heuristics

A heuristic is a “rule of thumb,” a guideline that wasn’t proven mathematically but our intuition /experience tells us is correct. When working under heuristic assumptions we can’t guarantee that we will get the best answer, but we will get a correct answer, and we believe that in most cases it will be a good answer. Heuristics are usually used to improve run time.

#### 2.1 Heuristics for global alignments

When searching for global alignments we can assume that a good alignment won’t have many large, consecutive gaps. Such an alignment will stay close to the diagonal in our matrix.

##### 2.1.1 Banded DP

In the banded DP algorithm, instead of calculating the whole matrix we will define an area around the diagonal, and perform all calculations only in this area. If we denote the width of the diagonal - k, then the time complexity is $$O(k \cdot n)$$. Notice that we have succeeded in reducing the time complexity from quadratic to linear. k can be picked empirically - after a few runs on the full matrix we can calculate the width of the band the best alignments fell in and pick k.

#### 2.2 Heuristics for local alignments

When working with local alignments it is not so simple to follow the diagonal heuristic because we don’t know in advance where in the matrix the best alignment falls.

##### 2.2.1 FASTA

FASTA works under the heuristic that good local matches contain patches of perfect matches. Overview of FASTA:

1. Find all perfect match (PM) diagonals of size k. When aligning DNA sequences typical k values are $$4 \leq k \leq 6$$ and for proteins $$2 \leq k \leq 4$$.
2. Find best band.
3. run Banded DP algorithm on best band.

- step 1: Finding PM diagonals in $$O(n)$$
  Formally we are looking for all pairs of (i,j) such that $$S[i,i+k] = T[j,j+k]$$. Since k is relatively small, we can hold a hash of all possible combinations of the alphabet of size k. Then we can iterate through the longer sequence and insert into the hash a list of indexes where such a sequence starts. This takes $$O(n)$$. If we are comparing a sequence to a database, we can do this procedure once and save the hash. Now all we have to do is iterate through the second sequence, and for every k characters find the all matching indexes in the hash.
step 2: Finding the best band.
After we have a list of all PM pairs we can try to combine PM diagonals. In order to locate PM’s on the same diagonal we can sort the pairs according to the difference: $i - j$. Then we can find diagonals which contain the most PM’s, greedily try to expand them, and proceed to step 3 with the best scoring diagonals.

2.2.2 BLAST
BLAST is NCBI's tool for finding local alignments. It is based on similar ideas to FASTA, with the main difference that it does not require PM diagonals, but will take any diagonal match that passes a certain predefined threshold. BLAST works with 2 parameters:
k - the length of the diagonal. The default value is $k=11$ for DNA and $k=3$ for proteins.
t - the threshold value.
k-diagonals that pass the threshold are named high scoring pairs (HSP) and are the seeds for the next step of searching for best bands. In order to compute HSP we iterate through the long sequence in the same way described in FASTA, with the difference that we index not only the exact k-sequences that are in the original sequence, but all k-sequences that are similar enough to the original one in order to pass the threshold. For example - if the original sequence is FSG... then we will index also YSG, FTG etc. Like in FASTA we can index the database in advance, and for every triplet save a list of triplets that are similar enough to pass the threshold.