COMPUTATIONAL BIOLOGY - ex1

Eliezer Shapira, Gil Adam

November 18, 2019

Q1.

Let s,t be two sequences of length n∈ℕ denoted by (s_1, ..., s_n), (t_1, ..., t_n) such that s_i, t_i∈ {A,C,G,T}

we show the number of possible alignments by a global algorithm with gaps allowed to be at least 3^n, under the assumption that the length of the alignment is no more than 2n (i.e. the alignment does not contain gaps in both sequences at the same index).

Alignment (s',t') is denoted by s'=(s'$_1$, ..., s'$_m$), t'=(t'$_1$, ..., t'$_m$) where s'$_i$, t'$_i$∈ {A,C,G,T,-}, n ≤ m ≤ 2n and the concatenations of s', t' without the symbol '-' are s and t respectively.

The shortest alignment is the trivial (s,t) without any gaps, and it is of length n. We can create 3^nalignment prefixes of length n in the following iterative manner: we start with s',t' as empty sequences and for each 1 ≤ i ≤ n we append one letter to each sequence in one of three ways: 1) append the first available letter from s to s' and the first available letter from t to t'. 2) append the first available letter from s to s' and '-' to t'. 3) append the first available letter from t to t' and '-' to s'. (where the first available letter from s is the lowest index letter which we have not yet appended to s', and likewise for sequence t).

Thus we have created 3^nunique alignment prefixes (s',t') of length n, each of which can be completed to a full and valid alignment in the following manner: denote j ≤ n to be the number of '-' in the sequence s' and k ≤ n to be the number of '-' in the sequence t', we assume w.l.o.g that k ≤ j. Thus the concatenation of s' without '-' is the sequence (s_1, ..., s_{n-j}) and the concatenation of t' without '-' is the sequence (t_1, ..., t_{n-k}).

We append to s' the sequence (s_{n-j+1}, ..., s_n) and to t' the sequence (t_{n-k+1}, ..., t_n) followed by j-k times the symbol '-', resulting in the alignment (s', t') of length n+j+k.

These 3^nalignments all have different prefixes, thus the overall number of possible alignments is no less than 3^n.

Q2.

As we saw in class (and in the book), the general function to update the cell [i, j] is

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

.

For the independent penalty we saw in class, $\sigma\left(s_i,\_\right),\sigma\left(\_,t_j\right)$ are constant d, so we get

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

.

Here $\sigma\left(s_i,\_\right),\sigma\left(\_,t_j\right)$ are no longer constant, and actually are function not only of the gap and the input, but also the length till here of the gap.

So, if we save also the length of the gap along the other things that we already save - the max score and how to get it - we can calculate it as before, taking the gap length plus one as the third argument of the σ, we can calculate the maximum score as before, and backtracking also as before - since we got the maximum and the path to get it we no longer needs to now how we calculate it.

So, the pseudo code is as follow (next page):

Algorithm 1 Linear Gap Penalty - Global Alignment

```
score(seq1_ltr, seq2_ltr, gap_length):
    if seq1_ltr is _ or seq2_ltr is _:
        return d + e * (gap_length - 1), gap_length
    else:
        return score_matrix[seq1_ltr, seq2_ltr], 0

backtrack(seq1, seq2, best_path):
    i, j = len(seq1), len(seq2)
    align1, align2 = [], []
    while i + j > 0:
        if best_path[i, j] is ←:
            align1.add_first(seq1[i])
            align2.add_first(_)
            i--
        if best_path[i, j] is ↑:
            align1.add_first(_)
            align2.add_first(seq2[j])
            j--
        if best_path[i, j] is ↖:
            align1.add_first(seq1[i])
            align2.add_first(seq2[j])
            i--
            j--
    return align1, align2

calculate_alignments(seq1, seq2):
    # initialize
    alignments = Matrix(len(seq1) + 1, len(seq2) + 1).fill(0)
    best_path = Matrix(len(seq1) + 1, len(seq2) + 1).fill(0)
    gap_length = Matrix(len(seq1) + 1, len(seq2) + 1).fill(0)
    # initialize 2 - first row, first column
    for i in len(seq1):
        gap_length[i, 0] = i
        alignment[i, 0] = score(seq1[i], _, i)
        best_path[i, 0] = ←
    for j in len(seq2):
        gap_length[0, j] = j
        alignment[0, j] = score(_, seq2[j], j)
        best_path[0, j] = ↑
    # recursion formula
    for i in len(seq1):
        for j in len(seq2):
            possible_alignments = [score(seq1[i], _, gap_length[i-1,j]+1),
                                   score(_, seq2[j], gap_length[i,j-1]+1),
                                   score(seq1[i], seq2[j])]
            possible_scores = first_from_any_element(possible_alignments)
            possible_gap_lens = last_from_any_element(possible_alignments)
            index_of_max = argmax(possible_scores)
            gap_length[i, j] = possible_gap_lens[index_of_max]
            alignment[i, j] = possible_scores[index_of_max]
            best_path[i, j] = [←, ↑, ↖][index_of_max]
    # get the best alignment
    return backtrack(seq1, seq2, best_path)
```