Sequence Alignment

Francesco Moraglio

April 15, 2020

1 Introduction

The main goal of computational sequence analysis in bioinformatics is to predict the function and structure of genes and proteins from their sequence. This is made possible since organisms evolve by mutation, replication and selection of their genes. Thus, sequence similarity often indicates functional and structural similarity.
We wish to identify what regions are most similar to each other in two sequences. Sequences are shifted one by the other and gaps introduced, to cover all possible alignments. The shifts and gaps provide the steps by which one sequence can be converted into the other one. In an alignment of two DNA sequences (pair-wise alignment), the two strings are written one under the other, with the purpose of collecting identical or similar pair of bases in the same columns. The introduction of gaps is allowed in order to pursue the construction of as many identical or similar pairs as possible. When we have two identical basis in the same column, we have a match; otherwise we have a mismatch (when different basis are in the same column) or a gap, when one of the two sequences has a deletion. We fix scores for matches and mismatches; we also have to decide a negative score to add to the total score of the alignment, for each gap.
This way we should be able to compute a score for each alignment and find the best one.

2 The Needleman-Wunsch algorithm

Dynamic programming algorithms find the best solution by breaking the original problem into smaller sub-problems and then solving. The Needleman-Wunsch algorithm is a dynamic programming algorithm for optimal sequence alignment, which was first proposed in the 1970s. Basically, the concept behind the algorithm stems from the observation that any partial sub-path that tends at a point along the true optimal path must itself be the optimal path leading up to that point. Therefore the optimal path can be determined by incremental extension of the optimal sub-paths. In a Needleman-Wunsch alignment, the optimal path must stretch from beginning to end in both sequences; for this reason this is considered a "global alignment" algorithm.

2.1 Description of the algorithm

First, let us establish some notation. We will be considering a pair of sequences, x and y, of lengths m and n, respectively. Let x_i be the i-th symbol in x and y_j be the j-th symbol

of y. These symbols, in a DNA sequence will come from the alphabet consisting in the four bases, $A = \{A, G, C, T\}$. We will denote symbols from this alphabet with lowercase letters. The additional gap symbol in alignments will be '−'.
We now have to fix a scoring system. In our case, for each couple of symbols (a, b), we define

$$s(a,b) = \begin{cases} m_A & a = b, \quad a, b \neq " - " \\ m_P & a \neq b, \quad a, b \neq " - " \\ g_P & a = " - " \quad \text{or} \quad b = " - " \end{cases} \tag{1}$$

where m_A is a fixed positive constant that represents the match award and m_P another constant, negative, for the mismatch penalty. g_P, the gap penalty, another negative fixed constant, is returned whenever a gap is found. Given an alignment, the total score will be

$$T = \sum_i s(a_i, b_i).$$

First step of the algorithm consists in creating a score matrix S of size $(m + 1) \times (n + 1)$, which allows us to compare the two sequences. S is indexed by $i = 0, \ldots, m$ and $j = 0, \ldots, n$, one index for each sequence, where the value $S_{i,j}$ is the score of the best alignment between the initial segment $(x_1, \ldots, x_i)$ of x and the initial segment $(y_1, \ldots, y_j)$ of y. We can build this matrix recursively, setting $S_{0,0} = 0$. We then proceed to fill the matrix from top left to bottom right. If $S_{i-1,j-1}$, $S_{i-1,j}$ and $S_{i,j-1}$ are known, it is possible to calculate $S_{i,j}$. There are three possible ways that the best score $S_{i,j}$ of an alignment up to x_i , y_j could be obtained: x_i could be aligned to y_j, in which case $S_{i,j} = S_{i-1,j-1} + s(x_i, y_j)$; or x_i is aligned to a gap, in which case $S_{i,j} = S_{i-1,j} + g_P$; or y_j is aligned to a gap, in which case $S_{i,j} = S_{i,j-1} + g_P$. The best score up to (i, j) will be the largest of these three options. Therefore, we have

$$S_{i,j} = max\{S_{i-1,j-1} + s(x_i, y_j), \quad S_{i-1,j} + g_P, \quad S_{i,j-1} + g_P\} \tag{2}$$

This equation is applied repeatedly to fill in the matrix of values $S_{i,j}$, calculating the value in the bottom right-hand corner of each square of four cells from one of the other three values (above-left, left, or above).
As we fill in the values $S_{i,j}$, for each cell we can also keep trace of the cell from which its $S_{i,j}$ was derived; we'll explain later the traceback procedure for determining the best alignment from the scores.
To complete our specification of the algorithm, we must deal with some boundary conditions. Along the top row, where $j = 0$, the values $S_{i,j-1}$ and $S_{i-1,j-1}$ are not defined so the values $S_{i,0}$ must be handled specially. The values $S_{i,0}$ represent alignments of a prefix of x to all gaps in y, so we can define $S_{i,0} = ig_P$. Likewise down the left column $F(0, j) = jg_P$.
The value in the final cell of the matrix, $S_{m,n}$, is by definition the best score for an alignment of $(x_1, \ldots, x_m)$ to $(y_1, \ldots, y_m)$, which is what we want: the score of the best global alignment of x to y.
To find the alignment itself, we must find the path of choices from (2) that led to this final value. The procedure for doing this is known as a traceback. It works by building the alignment in reverse, starting from the final cell, and following the pointers that we stored when building the matrix. At each step in the traceback process we move back from the current cell (i, j) to the one of the cells $(i - 1, j - 1)$, $(i - 1, j)$ or $(i, j - 1)$ from which the value $S_{i,j}$ was derived. At the same time, we add a pair of symbols onto the front of the

current alignment: x_i and y_j if the step was to $(i-1, j-1)$, x_i and the gap character '$-$' if the step was to $(i-1, j)$, or '$-$' and y_j if the step was to $(i, j-1)$. At the end we will reach the start of the matrix, $i = j = 0$.
Note that in fact the traceback procedure described here finds just one alignment with the optimal score; if at any point two of the derivations are equal, an arbitrary choice is made between equal options. The reason that the algorithm works is that the score is made of a sum of independent pieces, so the best score up to some point in the alignment is the best score up to the point one step before, plus the incremental score of the new step.

2.2 Complexity

It is useful to know how this algorithm's performance in CPU time and required memory storage will vary the size of the problem. From the algorithm above, we see that we are storing $(m+1)\times(n+1)$ numbers, and each number costs us a constant number of calculations to compute (three sums and a max). Hence the algorithm takes $O(mn)$ time and $O(mn)$ memory. Since m and n are usually comparable, the algorithm is usually said to be $O(n^2)$.

3 Smith–Waterman algorithm

So far we have assumed that we know which sequences we want to align, and that we are looking for the best match between them from one end to the other. A much more common situation is where we are looking for the best alignment between subsequences of x and y. This arises for example when comparing extended sections of genomic DNA sequence. It is also usually the most sensitive way to detect similarity when comparing two very highly diverged sequences.
The highest scoring alignment of subsequences of x and y is called the best local alignment.

3.1 The algorithm

The algorithm for finding optimal local alignments is closely related to the previous one, but there are two differences. First, in each cell in the table, an extra possibility is added to (2), allowing $S_{i,j}$ to take the value 0 if all other options have value less than 0:

$$S_{i,j} = max\{0, S_{i-1,j-1} + s(x_i, y_j), \quad S_{i-1,j} + g_P, \quad S_{i,j-1} + g_P\}. \tag{3}$$

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. Note that a consequence of this value is that the top row and left column will now be filled with zeroes, not ig_P and jg_P as for global alignment.
The second change is that now an alignment can end anywhere in the matrix, so instead of taking the value in the bottom right corner, $S_{m,n}$, for the best score, we look for the highest value of $S_{i,j}$ over the whole matrix, and start the traceback from there. The traceback ends when we meet a cell with value 0, which corresponds to the start of the alignment.
The local version of the dynamic programming sequence alignment algorithm was developed in the early 1980s. It is frequently known as the Smith–Waterman algorithm.

3.2 Complexity

As for the Needleman-Wunsch algorithm, to align two sequences of lengths m and n, the algorithm takes $O(mn)$ time and $O(mn)$ memory. There exists an optimized version of this algorithm, due to Myers and Miller, which reduces the space complexity to linear, that is $O(n)$.

4 Description of the implementation

Here a few technical details about the code.
Implementation of both algorithms is contained in the file `seqalign.py`, while files `nwrun` and `swrun` are python scripts for testing Needleman-Wunsch and Smith-Waterman algorithms, respectively. These scripts take as arguments two sequences in plain text format (no parsing is implemented) and run the algorithms. Everything has been implemented using Python 3.
I decided to write a procedural program since I saw no need for using objects.
In `seqalign.py`, after fixing values for awards and penalties, two utility functions are declared and implemented: `matchscore` and `finalize`. The former corresponds to the function $s(a, b)$ of Sec. 2, while the latter finalizes the results of the algorithm. This procedure takes as input the two aligned sequences named `align1` and `align2`, reverses them (algorithms output strings in reverse order since it starts the traceback from the last cell of the alignment) and computes `score` and `identity`. The first quantity is the total score of the alignment, while the second is the ratio between the number of matches of `align1` and `align2` and the length of `align1`. Moreover this procedure prints these values and the two aligned sequences, highlighting the presence of common bases making use of the `symbol` list.
In function `needle`, Needleman-Wunsch algorithm is implemented. The scores are computed iteratively (same for the other algorithm); a recursive version is also possible. Follows the traceback section, where, in this case, the alignment is built directly from the score values.
In `smith`, instead, the traceback is attained making use of a pointer matrix; solutions are equivalent. A part from this and the obvious difference in scoring, the codes are quite similar.
Several comments have been added to the code for explanatory purposes.