Multiple sequence alignment using Needleman-Wunsch and Smith-Waterman

Ken Bauwens

1 Introduction

In this report i will go over my implementation of the Smith-Waterman and Needleman-Wunsch algorithms extended to more than 2 sequences.
The biggest challenge in this assignment was to implement the algorithms for an arbitrary amount of sequences. Most of what is explained in this report will explain how i did this.

2 Requirements

The code is written in Python. The only required library is BioPython which is used to read fasta files.

3 Running the code

The algorithms can be ran as follows:
Smith-Waterman: python sw.py *[list of sequences]*
Needleman-Wunsch: python sw.py *[list of sequences]*
[list of sequences] is a list of fasta files containing one sequence each. The code requires at least 2 sequences.

4 Algorithm

A counter is maintained which indicates the current coordinate that has to be filled in. This counter will contain an element for each sequence. Each iteration the counter is increased as follows:

- Increase the first element of the counter
- If the first element of the counter reaches the length of the first sequence, set this element to 0 and increase the second element of the counter
- If the second element of the counter reaches the length of the second sequence, set this element to 0 and increase the third element of the counter
- Keep following this method until all elements of the counter are at the length of their respective sequence.

By following this method a counter can be initialized of arbitrary size depending on the amount of sequences to be aligned. Once the final counter has been reached all elements of the matrix will be filled.
Each iteration a single cell in the matrix is calculated. I will explain how this happens with an example.
Lets say we try to align three sequences S_1 = ABCD, S_2 = BBEF, S_3 = GHIJ and we want to calculate cell (1,1,1) where counting begins at 0. The algorithm goes through the following steps:

- generatebasepair((1,1,1)): Retrieve the nucleotides corresponding with the counter. This results in "BBH".
- generatemoves("BBH"): Generate a list of all possible moves leading to this cell. This results in ["BBH","_BH","B_H","BB_","__H","B__","_B_"] where _ is a gap.
- For each move:

  generatecoordinates(move): For a move calculate the coordinates of the cell which leads to the current cell. For "B_H" and the current counter this gives (0,1,0) as the "origin".

  retrievematrixelement(coordinate).score : Using this "origin" coordinate, retrieve the score for the sequence up to that "origin".

  getallpairs(move): For the current move. eg "B_H" get all pairs. This gives ["B_", "BH", "_H"]. These pairs are used to calculate the score according to the algorithm given in the assignment.

  scorePair(u) for u in pairs: For each pair the score is calculated based on a match, mismatch, gap or 2 gaps.

  Calculate the new score: The complete score for the current move can be calculated by adding all scores for the pairs to the score for the "origin".
- We now have the scores for all moves that can lead to the current cell. The score for the current cell is now the largest score, and the best "origin" is used during the backtracking to find the optimal alignment.
- Increase the counter

The backtracking can easily be done because the "origin" for each cell is saved. Backtracking happens according to the Smith-Waterman or Needleman-Wunsch algorithms.

5 Complexity

The upperbound for time-complexity is $O(M^N)$ where M is the largest sequence and N is the number of sequences. The upperbound for space-complexity is also $O(M^N)$ where M is the largest sequence and N is the number of sequences.