Bioinformatics Project Report: Sequence Alignment

Elias Dams s0201945

May 2024

Contents

1 Introduction

This report describes the implementation and application of sequence alignment algorithms for bioinformatics. It specifically tackles both local and global alignment for an arbitrary number of DNA/protein sequences using dynamic programming.

2 Algorithm Description

2.1 My Dynamic Programming Approach

The algorithm starts with the initialization of scoring parameters in the constructor (`__init__`), where match, mismatch, indel, and two-gap scores are set. These parameters dictate how sequence alignments are scored and are crucial for tuning the algorithm's sensitivity and specificity.

The method `_read_file` is used to read sequences from a FASTA file. It processes the file line by line, distinguishing between sequence identifiers and the sequences themselves. Each sequence is stored in a dictionary with its identifier as the key. This structured input makes it easier to handle multiple sequences.

The `_get_alignment` method initializes a multi-dimensional scoring matrix based on the number of sequences and their lengths. This matrix is central to the dynamic programming approach, where each cell represents an alignment score of sub-sequences up to that point. Using the initialized scoring matrix, the algorithm iteratively calculates scores for each cell: For each cell, the "previous" positions (neighbors) are generated using `_get_previous_neighbours`, which considers moving back one step in each dimension or not moving. The `_calculate_score_and_direction` method calculates possible scores from each neighbor by considering pairwise alignments of the sequences at each position. It then selects the optimal score based on the alignment method (global or local), updating the matrix accordingly.

Once the matrix is filled the traceback starts from the bottom-right corner for global alignment or the highest score position for local alignment, determined by `_get_current_position`.The `_update_sequences` method reconstructs the aligned sequences by moving backwards from the starting point to the origin, guided by the directions stored during the matrix filling. Each step potentially adds gaps or aligns residues based on how the current position relates to its optimal previous position. The traceback process continues until a stopping condition is met, defined in `_should_break`. For global alignment, it stops at the matrix's top-left corner. For local alignment, it stops when a cell with a score of zero is reached.

The aligned sequences and their corresponding alignment score are then output. If specified by the user, this output is not only printed but also written to a file. The `align_sequence` method manages these output operations.

2.2 Time and Memory Complexity

The scoring matrix's size is determined by the number of sequences (m) and the length of the longest sequence (n). For m sequences, the matrix becomes an m-dimensional hypercube, where each dimension is sized based on the length of a sequence (n). Each sequence adds a dimension, and each dimension can have up to n positions. This means the total number of cells in the matrix is n^m, representing all possible combinations of sequence positions. For each cell, the algorithm must compute scores based on all possible moves from previous cells (`_get_previous_neighbours`). For each cell, this involves checking potentially $2^m - 1$ neighbours (excluding the case where no movement occurs in any dimension). Of all these neighbours we need to calculate the pairwise score of (m^2) cells. This will result in a final time complexity of $O(n^m \cdot 2^m \cdot m^2)$.

As I previously described, the scoring matrix is an m-dimensional hypercube with n positions along each dimension, resulting in a total of n^m cells. Each cell must store an integer or floating-point number representing the score, which consumes a constant amount of space per cell. However, the exponential growth in the number of cells (n^m) leads to a significant increase in memory usage as the number or length of

sequences increases. To reconstruct the alignment from the scoring matrix, the algorithm also needs to store the traceback directions. I do thus in a dictionary with as key the position and as value the previous position. Storing these directions doubles the memory requirement because each cell in the matrix now also stores a reference to another cell. The factor 2 does not change the total space complexity which is $O(n^m)$.

3 Application to Biological Problem

3.1 Global Multiple Sequence Alignment

The sequences of the T cell receptor alpha and beta chains were analyzed using global multiple sequence alignment (MSA) to differentiate between TCB and TCA J protein sequences. The alignment results were as follows:

```
unknown_J_region_1: GYSSASKIIFGSGTRLSIRP
unknown_J_region_2: NTE-AF---FGQGTRLTVV-
unknown_J_region_3: NYG-YT---FGSGTRLTVV-
```

This analysis revealed two TCB J protein sequences and one TCA J sequence. Because regions 2 and 3 are greater similarity to each other,I conclude that these are the TCB sequences. Region 1 is identified as the TCA J sequence due to its distinct pattern.

3.2 Local Multiple Sequence Alignment

For the local alignment, the mismatch penalty was adjusted to -4. I now want to identify conserved regions among the sequences. The local MSA results were:

```
unknown_J_region_1: FGSGTRL
unknown_J_region_2: FGQGTRL
unknown_J_region_3: FGSGTRL
```

The conserved region identified in all three sequences is FG[S—Q]GTRL, highlighting a region of similarity across the T cell receptor sequences.

4 Appendix

4.1 Source Code

The source code is attached as part of the submission. It includes comments and is structured for easy readability and modification. (`multi_sequence_aligner.py`).

4.2 Output Files

The outputs of the script for given input sequences are included as separate files in the submission zip (`cs_assignment_output.txt`).