Parallel implementation of the Needleman-Wunsch algorithm for Global Sequence Alignment

This program computes in parallel the optimal global alignment of two sequences (DNA, RNA or protein sequences).

Required libraries

- pandas (version 2.2.1)*
- numpy (version 1.26.4)*
- multiprocessing**
- sys**
- argparse (version 1.1)*
- random**
- blosum (version 2.0.3)*
- tqdm (version 4.66.4)*

** you can install these individually or all together by using the requirements.txt file in this folder: 'pip install -r requirements.txt'*

*** same version of Python used: 3.12.0*

Usage

GlobalAlignment_Needleman_Wunsch.py [-h] (-s SEQUENCE SEQUENCE | -i INPUT INPUT) [-t {d,r,p}] [-g GAP] [-mm MISMATCH] [-m MATCH] [-b BLOSUM] [-o OUTPUT] [-q] [-c CORES] [-v {1,2}]

Options

-h, --help	show this help message and exit
-s SEQUENCE SEQUENCE, --sequence SEQUENCE SEQUENCE	the two sequences to be aligned
-f FASTA FASTA, --fasta FASTA FASTA	the two FASTA files containing the sequences to be aligned
-t {d,r,p}, --type {d,r,p}	type of sequences to align: 'd' for DNA sequences, 'r' for RNA sequences and 'p' for protein sequences. Default is 'd'
-g GAP, --gap GAP	negative GAP penalty. Default is -4
-mm MISMATCH, --mismatch MISMATCH	negative MISMATCH penalty. Default is -5
-m MATCH, --match MATCH	positive MATCH score. Default is 5
-b BLOSUM, --blosum BLOSUM	use the specified BLOSUM matrix for MATCHES/MISMATCHES (compatible only with protein sequences)
-o OUTPUT, --output OUTPUT	save the alignment(s) in the specified output file
-q, --quiet	don't display the output
-c CORES, --cores CORES	number of cores to use. Default is 3 (if available)
-v {1,2}, --verbose {1,2}	increase verbosity. Default is 1
-a, --approximation	don't show all possible alignments but a reduced number of them

Examples

- py GlobalAlignment_Needleman_Wunsch.py -s AUCAUCA ACUCAU --type r -o alignments.txt
- py GlobalAlignment_Needleman_Wunsch.py -f fasta1.fa fasta2.fa -t p --blosum 62
- py GlobalAlignment_Needleman_Wunsch.py -s MKTYS MKFFS -t p --verbose 2 -g -1 -mm -4 -m 3

Try to align 'ATCCCCATCAAAA' and 'GACTGGCTAAA' both with and without the option '--approximation' (keep match/mismatch/gap set to the default values).

Implementation

The approach presented here is based on dynamic programming and it consists of three main steps:

1) Initialization
2) Matrix fill (scoring)
3) Traceback (alignment)

1) Initialization

The first step is to create a matrix with N+1 rows and M+1 columns, where N and M are the lengths of the sequences to be aligned. After that, a '0' is put in the first upper left most cell.
Considering the two DNA sequences ATCG and CGGG, the starting matrix would be:

	-	A	T	C	G
-	0				
C					
G					
G					
G					

2) Matrix fill (scoring)

At this point the maximum score is assigned to each cell, and to do this two things are needed:

- A scoring system, i.e. the values to use for mismatch, match and gap; in this program they can be specified individually or, just for the protein sequences, matches/mismatches can be handled by a BLOSUM matrix (for which the user has to specify the number, e.g. 62).
  Quick recall: for evolutionarily distant sequences, a low number is advised, instead for evolutionarily close sequences, a high number is advised.
- The scores of the adjacent cells: more specifically, in order to compute the score of the cell $M_{i,j}$, the scores of the cells $M_{i-1,j}$, $M_{i,j-1}$ and $M_{i-1,j-1}$ are needed.

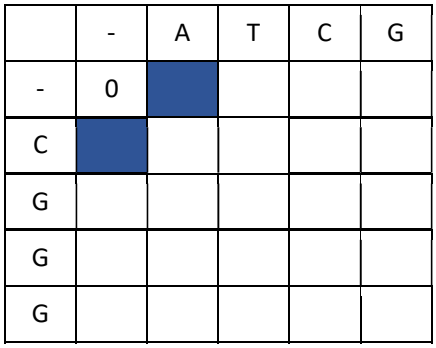
This is resumed by the following expression:

For each position, $M_{i,j}$ is defined to be the maximum score at position i,j; i.e.

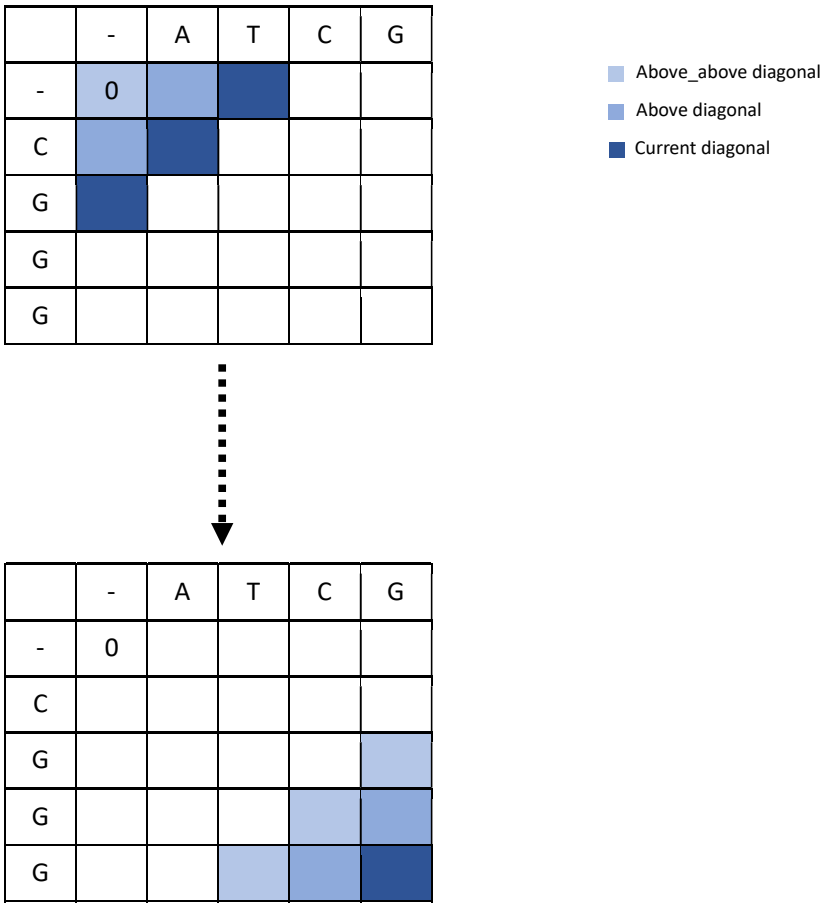
$$M_{i,j} = MAXIMUM[$$
$$M_{i-1,j-1} + S_{i,j}\ (match\ or\ mismatch\ in\ the\ diagonal)$$
$$M_{i,j-1} + w\ (gap\ in\ sequence1)$$
$$M_{i-1,j} + w\ (gap\ in\ sequence2)]$$

*Where **w** is the gap penalty and **S** is either the match score or the mismatch penalty (depending on the equality/inequality of the two correspondent characters).*

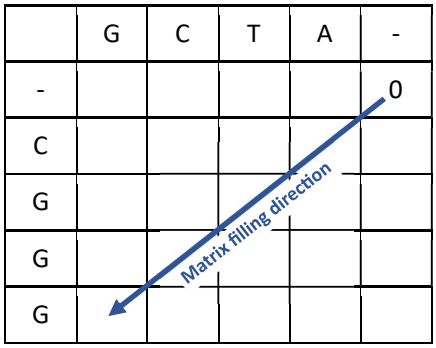
Starting from the first iteration, it's already possible to see that all the information to compute both $M_{2,1}$ and $M_{1,2}$ are present, because their scores only depends on the score of $M_{1,1}$ (and the scoring system of course). So they can be computed in **parallel**!



In the next iteration, as in the previous case, it's possible to compute in parallel the scores of the cells belonging to the next diagonal.
Following this pattern, it's now clear that all the cells belonging to a diagonal can be computed in parallel and all the information needed are present in the two diagonals above *('above' and 'above_above' diagonals, as they are called in the script)*.



Note: the real implementation of the script is slightly different: because of how the function to compute the indices of the diagonal elements was originally thought, the sequence placed above (the longer one) is actually reversed, so the initialization is made by a 0 in the upper **right** most cell and so the last cell to be computed will be the lower left most cell, but the concept behind is exactly the same.



So the program is basically iterating over the diagonals of the matrix and at each iteration, the only information needed are the scores of the cells of the two diagonals above the current one (and the scoring system of course).

In principle the maximum number of cells that could be computed in parallel, at each iteration, is the length of the current diagonal, which of course depends on the lengths of the two sequences to align, but in practice this number also depends on the number of cores available: e.g. if the computer has 12 cores and the current diagonal has 20 cells, only 12 cells will be actually computed in parallel. So, in the end, the number of cells computed in parallel depends both on the machine specifications and sequences length.

At the end of this phase, a 3D array is filled: the first layer is the matrix of scores, while the second one is the matrix of moves, which are represented by a number:
0 = horizontal move (insertion of gap in the sequence placed above)
1 = vertical move (insertion of gap in the sequence placed laterally)
2 = diagonal move (alignment of the two respective characters)
3 = horizontal and diagonal moves
4 = horizontal and vertical moves
5 = vertical and diagonal moves
6 = all three alternative moves

3) Traceback (alignment)

This is the final phase and the focus is on the second layer of the previously mentioned 3D array.
The best alignment score is found in the lower right most cell of the **first** layer, so following the path represented by the numbers of the **second** layer is possible to reconstruct the alignments with the same best score.
Considering moves 3, 4, 5 and 6, it's clear that there may be more than one alignment: for this reason, a recursive traceback was implemented, in order to report every equivalent alignment found.

At each iteration of the recursion, there is a growing string in which all the information regarding each distinct alignment are stored and separated by a ':' symbol.
In this way, in the end, there is a string like 'alignment1:alignment2: … : alignmentN', from which the graphical representation of the alignment(s) is built.

Considering the example sequences, the output string is:
'A-T-CC-G-GGG:A-T-CC-GGG-G:A-T-CCGG-G-G', which corresponds to:

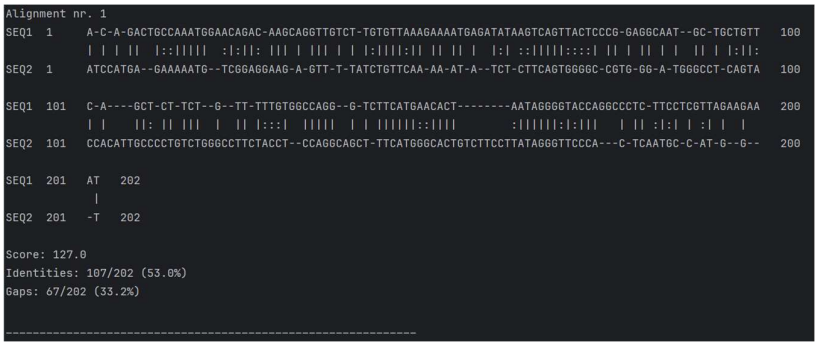
Alignment 1
SEQ1: A T C _ _ G
SEQ2: _ _ C G G G

Alignment 2
SEQ1: A T C _ G _
SEQ2: _ _ C G G G

Alignment 3
SEQ1: A T C G _ _
SEQ2: _ _ C G G G

Given the fact that the method is based on recursion, very long sequences, particular choices of match/mismatch/gap values or very different sequences, could cause very long waiting time (the alignment of 'ATCAAAAAAAAAAAAAGAAAAAAAGAG' and 'CGCCCCCCCCCCGGGCCGG', with default values, leads to 39879 alignments) or even RecursionError.
For this reason, the option '-a or --approximation' was added. As already mentioned above, the moves 3, 4, 5 and 6 are composed by more than one move: when the user uses the '--approximation' option, an heuristic is applied, which consists in transforming each of the encountered complex moves (either 3, 4, 5 or 6) into a simplified version of them.
For example, move 3 represents both horizontal and diagonal movements, so only one of those two will be chosen as new move. The same applies to the other complex moves.
The choice of the new move is made at **random** so to not introduce bias in the shown alignments.
But, this solution doesn't prevent RecursionError in **all** cases: so, if even after applying heuristic, a RecursionError is raised, then only one alignment will be shown, which is the one **maximizing the characters aligned (or minimizing the introduction of gaps)**, i.e. choosing 2 (which corresponds to the diagonal movement) whenever is possible. Move 4 is the only one not including the diagonal move, so in this case the new move is chosen at random between moves 0 and 1).

Output example:



‘|’ means the two characters aligned are equal, ‘:’ means the two characters are aligned but are different and ‘ ’ means there is a gap insertion in either the first or the second sequence. Basic statistics regarding each alignment found are also reported.

Additional information

In the same folder of the script there are also the requirements.txt file (already mentioned in the **Required libraries** paragraph) and two trimmed versions of real FASTA files to try the --fasta option (the usage of the --approximation option is highly recommended).

Marco Cominelli, Università degli Studi di Milano / Politecnico di Milano