COMPSCI 260, Introduction to Computational Genomics **14 October 2022**

Problem Set 4

Due: 5pm on 28 October 2022

Problem 1: Filling in a small DP table by hand (8 points)

List of files to submit:

1. `README.problem1.[txt/pdf]`
2. `(DP table if it is not part of your README)`

This problem will help you understand how dynamic programming (DP) for sequence alignment really works. You must solve this problem by hand, in preparation for the following problem in which you will be writing code to implement the solution for a sequence alignment problem using dynamic programming. For most students, it is much easier to implement an alignment algorithm in code *after* they have solved a simple example by hand. As a further bonus, you will be able to use the table you construct here to help verify the correctness of your implementation in the following problems.

The table can be drawn by hand or computer and can be embedded in your README.problem1.[txt/pdf] or uploaded as a separate file at time of submission.

Consider the following DNA sequences: `GAATCGGA` and `TAGTA`. Use the following simple scoring system:

- A single match scores $+2$
- A single mismatch scores -1
- A single gap character scores -2 (a.k.a. linear gap penalty with $g = 2$)

a) Compute and provide the dynamic programming table that would be filled in by a global alignment algorithm given these two sequences. Please place the first (longer) sequence along the rows of the table and the second (shorter) sequence along the columns, so your table should end up being taller than it is wide.

b) Use your table to determine the score of an optimal alignment, the number of alignments that achieve this score, and the full set of all optimal alignments.

c) Finally, if an affine gap score were used instead of a linear gap score, which alignment(s) would most likely no longer be optimal?

Problem 2: Global alignment with linear gap score (22 points)

List of files to submit:

1. `README.problem2.[txt/pdf]`
2. `GlobalAligner.py`

3. `GlobalAlignerPlus.py`
4. `atpa_Hs.fasta` (no longer empty)
5. `atpa_Ec.fasta` (no longer empty)

⋆ **Step 1:** Implementing a global aligner

Plan

Examine the Python program entitled `GlobalAligner.py`, provided as part of the code for this problem set. It is designed to use a dynamic programming algorithm to compute the score of the optimal global alignment between two sequences in the context of a linear gap score (as discussed in class). It accepts as input a pair of sequences, and those may be either a pair of protein sequences or a pair of DNA sequences. To score correctly, the program needs to know whether the two sequences are protein sequences or DNA sequences. So, it scans the two input sequences and looks for any characters different from `A`, `C`, `G`, or `T`. If no character other than these four are found, it is assumed that the sequence is DNA. If the input sequences are DNA, the match and mismatch values used to score the alignments are used to build the substitution matrix that will be used as an input parameter to the aligner function. If instead the sequences are proteins (to validate its input, the program makes sure that only the standard 20 amino acid codes are present), the program loads the BLOSUM62 substitution matrix from a file `BLOSUM62.txt`. In either case, the program later uses the linear gap penalty passed as an input parameter (corresponding to the value g), which should be entered as a positive number according to the requirements of this particular program. Study this code carefully and make sure you understand how it works because you will be extending it in different ways throughout this problem set.

Note: We provide a number of helper functions in `aligner_helpers.py`, some of which are already being called and others not. You may need or want to use some of them later when you complete `GlobalAlignerPlus.py`, and others may just be useful to you while developing your code.

Develop

Although `GlobalAligner.py` is designed to compute the score of the optimal global alignment(s) with a linear gap score, it does not quite work yet, nor does it output the actual optimal alignment(s). Let's fix these things, one at a time.

a) In `GlobalAligner.py`, examine the provided code and be sure you understand how the dynamic programming table is being created and laid out in `solve_global_aligner`(). Then, add the necessary code in `global_aligner`() to fill in all the values of the table, which will allow you to compute and return the optimal alignment score. When working with a DNA sequence, your program should report the score of the optimal alignment using the same match, mismatch, and gap penalty values from problem 1; in fact, you can test if you've done this right by comparing the answer you get with the results of problem 1.

b) Now, write a separate program called `GlobalAlignerPlus.py` (based on `GlobalAligner.py`) that computes and then **reports** the following:

1. the optimal alignment score
2. the top-most alignment achieving this score
3. the bottom-most alignment achieving this score

To ensure the computed alignments are readable, use the provided `print_alignment()` function when **reporting** them. This function will print a sequence alignment by breaking up the aligned sequences using a fixed number of alignment columns per line.

When using this function or extensions thereof in future problems, don't forget to include all of these reported alignments in your `README`.

Note: "top-most" and "bottom-most" are relative terms based on the order in which sequences are entered: If you swap the sequences, you will swap the labels on these alignments, though the two alignments are otherwise unchanged.

Extra Challenge: If you're looking for an extra challenge, have your program output not just the top-most and bottom-most optimal alignments, but rather *all* optimal alignments. This can be done using either breadth-first search (BFS) or depth-first search (DFS) on the traceback pointers. If your memory of these is hazy, a simple overview of the BFS and DFS algorithms can be found at `http://www.cs.duke.edu/courses/spring22/compsci260/resources/graphSearch.pdf`. If you want an even greater challenge, don't bother to store the traceback pointers when you first fill in the dynamic programming table, but instead, compute them *on the fly* while using DFS to output all the alignments.

Apply

c) Next, you will apply your global aligner to actual protein sequences. Four fasta files are provided to you in this problem set. However, two of them (`atpa_Hs.fasta` and `atpa_Ec.fasta`) are empty.

Visit the UniProt protein sequence database. This can be located at `http://www.uniprot.org/`. Browse around a little to see what kinds of things this database is useful for. Now, search for the protein with accession number P25705, which should return a page for a human ATP synthase (ATPA_HUMAN), an important enzyme that produces ATP from ADP in your mitochondria. *Examine some of the fields in this page and see what you can understand, and what seems foreign.* Click on the "Sequences" tab on the left and then click on the "FASTA" button near the top of the section (make sure you are looking at the first isoform) to get the sequence of this protein in FASTA format. **Save this FASTA-formatted sequence in the file `atpa_Hs.fasta`.**

Now search for the protein with accession number P0ABB0, which should be an ATP synthase from the bacterium *Escherichia coli* (ATPA_ECOLI). Again, navigate to the "Sequence" tab and use the "FASTA" button to download the sequence of this protein in FASTA format. **Save this FASTA-formatted sequence in the file `atpa_Ec.fasta`.**

Use your `GlobalAlignerPlus.py` to generate and **report** the top-most and bottom-most optimal global alignments (with gap penalty 8, now that we're working with protein sequences) for the sequence in `atpa_Hs.fasta` with the sequence in `atpa_Ec.fasta`. Since these are protein sequences, the code will use an amino acid substitution matrix (in this case, BLOSUM62) to compute the various match and mismatch scores.

Reflect

d) *Given what you know about these proteins, what do you notice about the alignments and the degree of similarity?*

⋆ **Step 2:** Using your global aligner

Plan

We have provided you with the sequences for the ATP synthase corresponding to the mouse *Mus musculus* and the bacterium *Bacillus subtilis* in the files `atpa_Mm.fasta` and `atpa_Bs.fasta`, respectively.

e) *Based on what you know already, how do you expect these sequences to align with the sequences you found for human and E. coli?*

Apply

f) Use your `GlobalAlignerPlus.py` to generate and **report** the optimal global alignments (with the BLOSUM62 substitution matrix and gap penalty 8) of the sequences in `atpa_Mm.fasta` and `atpa_Bs.fasta` with the sequences in both `atpa_Hs.fasta` and `atpa_Ec.fasta`. Also generate and **report** the optimal global alignments between `atpa_Mm.fasta` and `atpa_Bs.fasta` (which means that between subproblems **c** and **f**, you will have now completed alignments for all 6 possible pairs among these 4 sequences).

Note: You should produce top-most and bottom-most alignments if there are multiple, but you may report just one alignment if the optimal alignment is unique.

Reflect

g) *What do you observe about your results from part f?*

h) For two sequences X and Y (of size m and n, respectively), the algorithm we originally gave you in `GlobalAligner.py` takes $O(mn)$ time and $O(mn)$ space complexity. We know that space is precious, especially when we have to handle extremely large sequences.

How could you modify the original global alignment algorithm to improve its space complexity? In particular, can you come up with a way to compute the score (not the actual alignment) of the best global alignment still in $O(mn)$ time, but using only $O(m)$ or $O(n)$ space? Or better yet, using $O(\min(m, n))$ space? You do not need to implement this: simply describe clearly how it would work.

Extra Challenge: Implement your solution in the function `compute_global_aligner_score_linspace` provided in `GlobalAligner.py`.

Problem 3: From linear to affine gap score (20 points)

List of files to submit:

1. `README.problem3.[txt/pdf]`
2. `AffineGlobalAlignerPlus.py`

Plan

Now suppose we wish to extend `GlobalAlignerPlus.py` so as to replace the linear gap score with an affine gap score. As a reminder, this means we wish that each position in a gap block will decrease the score by $g > 0$, with the initial gap position decreasing the score by an additional value $h > 0$ (in other words, the first position of a gap scores a total of $-(g + h)$ and subsequent positions only $-g$).

Develop

a) Write a separate program called `AffineGlobalAlignerPlus.py` (based on `GlobalAlignerPlus.py`) that is capable of using an affine gap score to compute the top-most and bottom-most optimal global alignments for two sequences, along with those alignments' score. The score you compute for an optimal

alignment should depend on g and h. These values should be obtained from the input parameters of the function. Your function will also accept as input a pair of sequences, which may be either a pair of protein sequences or a pair of DNA sequences. Additionally, your function will return the same set of values as your `GlobalAlignerPlus.py` function did: the optimal alignment score, the top-most optimal sequence alignment, and the bottom-most optimal sequence alignment.

Apply

b) Retrieve the two ATP synthase fasta files for human and *E. coli* you obtained in the previous problem set (i.e., `atpa_Hs.fasta` and `atpa_Ec.fasta`). Using your `AffineGlobalAlignerPlus.py`, globally align these sequences using affine parameters $g = 1$ and $h = 11$. Then, **report** the optimal alignment score, the top-most global alignment, and the bottom-most global alignment.

Reflect

c) *How does this alignment compare with those you generated for the same sequences using `GlobalAlignerPlus.py`? Do your observations line up with your expectations given the difference between these two algorithms? What do your observations suggest about the sequences you are comparing?*

d) Imagine setting $h = 0$ and $g = 8$ in `AffineGlobalAlignerPlus.py`. Will this always return the same score for an optimal alignment as `GlobalAlignerPlus.py` with a linear gap penalty of 8? If yes, be sure to explain why; if not, be sure to explain why not or give a counterexample.

e) How do the running times of `GlobalAlignerPlus.py` and `AffineGlobalAlignerPlus.py` compare, both in terms of asymptotic time complexity and real time elapsed? How do you predict the space will compare, in terms of asymptotic space complexity and actual space used?

Problem 4: From global to local alignment (10 points)

List of files to submit:

1. `README.problem4.[txt/pdf]`
2. `LocalAlignerPlus.py`
3. `FLAG.txt`

Plan

In this problem, you will be extending the global alignment program developed thus far to handle local alignments. Recall from class that even though $\Theta(m^2n^2)$ different pairs of substrings are selectable from X and Y, it is possible to compute the score of the optimal local alignment in time that is still $\Theta(mn)$. The DP subproblems in global alignment are based on prefixes of X and Y, and key idea behind the local alignment algorithm is that *every substring is the suffix of some prefix.*

For example, the substring AGTT of the string CAAGTTCAT is a suffix of the prefix CAAGTT. With this insight in mind, in the local alignment problem, the (i, j) element of the local alignment dynamic programming table V' will be defined to be the score of the optimal alignment of *any* suffix of $x_1 \ldots x_i$ to *any* suffix of $y_1 \ldots y_j$.

Develop

a) Write a separate program called `LocalAlignerPlus.py` that performs local alignment. You should base your program on `GlobalAlignerPlus.py`, because we will only use a linear gap penalty, rather than an affine gap penalty (we're keeping things simpler for you here, but an affine gap penalty is certainly possible in local alignment: you simply combine the two ideas; that would be an interesting *extra challenge* if you want to test yourself). As above, your function should accept as input a pair of sequences, and those may be either a pair of protein sequences or a pair of DNA sequences.

In the case of a tie where stopping your traceback or continuing on both score 0, we are interested in the longer option. So don't stop your traceback until stopping is the only option.

In local alignment, the dynamic programming table might have multiple locations where the maximum score appears.

Your function will return, and you should use what it returns to **report**, the following:

- The optimal local alignment score
- The total number of locations in the table achieving this optimal score
- The locations in the table of each of these optima
- Any **one** optimal alignment. In reporting your one optimal alignment, you will report the two aligned sequences and for each sequence, the nucleotide range of the characters being used in the alignment (inclusive). So if your reported optimal local alignment aligns nucleotides 27–42 of the first sequence with nucleotides 89–118 of the second, you would report those four numbers.

Apply

b) Use your `LocalAlignerPlus.py` to **report** the optimal local alignment of the two protein sequences with UniProt accession numbers P63015 and O18381 (as before, since these are proteins, use a gap penalty of 8). These two FASTA files are provided to you.

Reflect

c) What proteins are these? How are they similar to one another? What common bit of functionality do they share? *How would you interpret the local alignment results in this context?*

d) What does the value of $V'(m, n)$ mean? How does this value compare to the score of the best local alignment?