COMPSCI 260, Introduction to Computational Genomics **11 October 2019**

Problem Set 4

Due: 5pm on 25 October 2019

Please comment your code clearly, especially when you're changing code that has been provided for you, but please don't change function signatures (the expected parameters and output, which we try to be clear about in each function's comments). Do your best to submit code that runs without errors, even if you don't completely finish the problem.

Problem 1: Filling in a small DP table by hand (10 points)

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 problems in which you will be writing code to implement the solution for a sequence alignment problem using DP. 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 DP table you construct here to help verify the correctness of your implementation in the following problems.

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

List of files to submit

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

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 score; gap penalty: 2)

a) Compute and provide the DP table that would be filled in by a global alignment algorithm given these two sequences.

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 optimal alignments.

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

Problem 2: Global alignment with linear gap score (35 points)

List of files to submit

1. `README.problem2.[txt/pdf]`
2. `GlobalAligner.py`
3. `GlobalAlignerPlus.py`
4. `atpa_Hs.fasta`
5. `atpa_Ec.fasta`

⋆ **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 either a pair of protein sequences or a pair of DNA sequences. To score correctly, the program needs to know whether the two sequences being aligned 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 obtained from the input parameter (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: You may need to use helper functions we have provided in `GlobalAligner.py` for `GlobalAlignerPlus.py` as well as for your implementations of the extensions of global alignment in problems 3 and 4. Additionally, in the case where you have written additional helper functions in `GlobalAligner.py`, you will need to submit `GlobalAligner.py` for problems 3 and 4.

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, and it does not output the actual optimal alignment(s). Let's fix these things.

a) In `GlobalAligner.py`, we provided code to initialize the dynamic programming table. However, the code that fills the rest of the table is missing. First, fill in the missing bit of code in `GlobalAligner.py` so that it reports the score of the optimal alignment using the same match, mismatch, and gap penalty values from problem 1 (2, -1, 2). You can test if you've done this right by comparing the answer you get with the results of the previous problem.

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 up-most alignment achieving this score
3. the down-most alignment achieving this score

When using this function or extensions thereof in future problems, be sure to include all of these reported values in your `README`.

Note: "Up-most" and "down-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 up-most and down-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/fall19/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.

Check

c) 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), one that produces ATP from ADP. *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 a file called `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 a file called `atpa_Ec.fasta`.**

Use your `GlobalAlignerPlus.py` to generate and **report** the optimal global alignment (with gap penalty 8) for the sequence in `atpa_Hs.fasta` with the sequence in `atpa_Ec.fasta`. Use the same match and mismatch values provided in the previous alignment (2 and -1).

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?*

Check

f) Use your `GlobalAlignerPlus.py` to generate and **report** the optimal global alignments (with match 2, mismatch -1, 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`.

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 (30 points)

List of files to submit

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

Plan

Suppose we wish to extend `GlobalAlignerPlus.py` to replace the linear gap score with an affine gap score. As a reminder, this means we wish that each position in the gap decreases the score by $g > 0$, with the initial gap position decreasing the score by an additional value $h > 0$ (so 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 up-most and down-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. Additionally, have your function **report** the same set of output as was reported in your `GlobalAlignerPlus.py` function.

Check

b) Use your `AffineGlobalAlignerPlus.py` to generate and **report** the optimal up-most and down-most global alignment (with $g = 1$ and $h = 11$) for the two ATP synthase sequences you found (i.e., for human and *E. coli*).

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 you predict the running times of the two programs will compare, in terms of real time elapsed, not asymptotic time complexity?

Problem 4: From global to local alignment (25 points)

List of files to submit

1. `README.problem4.[txt/pdf]`
2. `GlobalAligner.py`
3. `LocalAlignerPlus.py`
4. `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 there are m^2n^2 different pairs of substrings selectable from X and Y, it is possible to compute the score of the optimal local alignment in time that is still $O(mn)$. 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 $x_1 \dots x_i$ to any suffix of $y_1 \dots y_j$.

Develop

a) Write a separate program called `LocalAlignerPlus.py` (based on `GlobalAlignerPlus.py`) that computes the up-most and down-most optimal local alignments and **reports** the same set of output as was reported in `GlobalAlignerPlus.py`.

Check

b) Use `LocalAlignerPlus.py` to **report** the up-most and down-most optimal local alignments (with gap penalty 8) of the two protein sequences with UniProt accession numbers P63015 and O18381.

Reflect

c) What do the values of $V'(m, n)$ mean? How do these values compare to the score of the best local alignment?

d) 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?*