COMPSCI 260, Introduction to Computational Genomics **25 November 2022**

Problem Set 7

Due: 5pm on 9 December 2022

Problem 1: Viterbi decoding a genome (20 points)

List of files to submit:

1. `README.problem1.[txt/pdf]`
2. `viterbi.py`

Plan

In this problem, you will implement a Viterbi decoder for HMMs and apply it to the genome sequence of an ocean-dwelling archaeon. The decoder has three major steps: initialization (already implemented in `viterbi.py`), matrix fill, and traceback. Before you begin writing your code, you **must** read the information in `durbin.pdf`, included in the zipfile you downloaded for this problem set. It is important that you perform the decoding using **log probabilities**; the PDF file describes exactly why and how this works.

Develop

a) Implement the matrix fill step of your Viterbi decoder in `viterbi.py`. What is required to do this is pretty straightforward for the Viterbi case (but a bit more complicated when you do posterior decoding; more on that later). Also implement the traceback step so that your code computes the optimal Viterbi decoding path, $\Pi^\star$. Upon computing this path, your program should compress it into a list of contiguous segments in which the state remained the same. Each such segment will be represented by a dictionary. The returned list of dictionaries should therefore look something like this:

```
[
    {'start': 1, 'end': 12, 'state': 'S'},
    {'start': 13, 'end': 20, 'state': 'W'},
    ...
]
```

Note: The start and end coordinates must be 1-indexed and inclusive.

b) Write a function called `count_segments` in `viterbi.py` to count the number of segments that exist in each state. You will use the returned dictionary from `count_segments` in the next subproblem. It could be useful to **report** other information along the way, including perhaps statistics about the length distribution of the segments for each state.

Note: For debugging and testing purposes, you may find it useful to apply your program to the short artificial sequence contained in the file `artificial.genome.fasta`. In `artificial.genome.fasta`, the lowercase

nucleotides were generated by W; uppercase nucleotides by S. The artificial sequence was generated using the HMM parameters provided in `HMM.methanococcus.txt`. The HMM parameters in `HMM.methanococcus.txt` were initialized to values that would help detect tRNA segments in the genome.

Apply

We will now use the Viterbi decoder to understand one aspect of the structure of a real genome.

A standard Markov model is not likely to be a great model for a genome sequence because the base composition of most genome sequences is not homogeneous. In particular, nucleotide composition can vary regionally throughout the genome.

An interesting example is found in bacterial genomes. It turns out that the overall GC composition of bacterial genomes appears to be fairly consistent for most of the genome, but that the GC composition of the parts of the genome encoding structural RNA genes is strongly correlated with the normal growth temperature of the organism (we think this is because structural RNAs have to maintain a base-paired secondary structure; thus, if the organism lives at higher temperatures, these sequences are driven towards higher GC composition because base-paired RNA structures with more Gs and Cs have higher thermostability).

As one instance of this, consider the genome of *Methanocaldococcus jannaschii* (its genus was recently changed from *Methanococcus*, which is what it was known as when this genome was first sequenced, so sometimes you will see that used instead). It is a hyperthermophilic (meaning it survives at extremely high temperatures), methanogenic (meaning it produces methane) archaeon (meaning it is a member of the domain Archaea) that lives in deep ocean vents, where pressures and temperatures are exceedingly intense. Overall, the base composition of the genome is strongly AT-biased, but the regions of its genome that encode structural RNA genes are strongly GC-biased. We therefore might want to segment the genome sequence of *M. jannaschii* into regions of different nucleotide composition in order to identify the locations where structural RNA genes are found. We will try to accomplish this using an HMM.

Let's assume the *M. jannaschii* genome can be modeled as a two-state HMM, with the two states being: W (AT-rich; this will model the bulk genome) and S (GC-rich; this will model potential locations of structural RNA genes). The parameters of this model are specified in the file `HMM.methanococcus.txt`, whose format is similar to what you saw in the last problem set.

c) Once it's ready to go, use your Viterbi decoder to decode the state regions within the actual genome sequence of *M. jannaschii*. A copy of the complete *M. jannaschii* genome is available in FASTA format in the file `bacterial.genome.fasta`. Run `viterbi_decoding` on the *M. jannaschii* genome and **report** the first ten and last ten segments of your decoding. Then, call `count_segments` and **report** the number of segments in which each state appears.

Reflect

d) How many structural RNA regions (segments corresponding to state S) do you detect with your Viterbi decoder? A file containing the positions of known transfer RNA genes is available in `tRNA.locations.txt`. *What is the relationship between these positions and those learned by your Viterbi decoder? Discuss.*

Problem 2: Posterior decoding a genome (20 points)

List of files to submit:

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

Plan

This problem is similar to the previous Viterbi decoding problem, except that here we will implement a posterior decoder instead of a Viterbi decoder. To do so requires that we compute the probabilities that the sequence was generated from W versus S for every position in the genome: If we denote the complete genome sequence as X, we will be computing $P(\pi_i = W|X)$ and $P(\pi_i = S|X)$ for all i. To accomplish this, we will need to implement three routines:

1. Calculation of forward probabilities: we need to compute $f_k(i)$ for $k \in \{W, S\}$ and all i.
2. Calculation of backward probabilities: we need to compute $b_k(i)$ for $k \in \{W, S\}$ and all i.
3. Calculation of posterior probabilities: using the forward and backward probabilities along with $P(X)$, we need to compute $P(\pi_i = k|X)$ for $k \in \{W, S\}$ and all i. This step is already implemented in the Python file `posterior.py`.

As with Viterbi, you must be sure to do all this using log probabilities or your computer is likely to crash. Again, descriptions for how to do this are in the excerpt from the Durbin book that we provided; as the excerpt makes clear, it is slightly more complicated in the posterior decoding setting because you need to *add* log probabilities rather than simply take the maximum of log probabilities. You'll probably want to create a helper function that knows how to efficiently compute the log of the sum of a set of probabilities when it is passed a list of the logs of those probabilities (the direct way to do this leads to underflow, which is what we are trying to avoid; therefore, your helper function should make use of the strategy outlined in the Durbin excerpt).

Develop

a) Implement the three routines, `posterior_decoding`, `run_forward`, and `run_backward`, found in the `posterior.py` skeleton file.

Apply

b) Once the three routines are implemented in `posterior.py`, you can use the computed posterior probabilities to produce the posterior decoding $\hat{\Pi}$ by selecting the state that is most likely at each position.[1] As before, you may wish to begin by testing your program on the sequence in `artificial.genome.fasta` before running your posterior decoder on the real *M. jannaschii* genome.

Report the first ten and last ten segments in your posterior decoding $\hat{\Pi}$ of the *M. jannaschii* genome.

[1] If you wanted to take this further and display a figure plotting the posterior probabilities as a function of the position, feel free—it may help your analysis and a Python module like `matplotlib` could be used to produce it; but in this assignment, we only require you to segment the genome (determine the contiguous segments in which the most likely state is the same for the segment) and doing that only requires $\hat{\Pi}$.

Reflect

c) How many structural RNA regions do you detect now? *What is the relationship between their positions and the positions of the transfer RNA genes? Discuss.*

d) Finally, compare the results of both decoders. *Which decoder seems to work more effectively in detecting structural RNA regions of the genome and why?*

Problem 3: Gene prediction and identification (20 points)

List of files to submit:

1. `README.problem3.[txt/pdf]`
2. `assembly_tester.py`
3. `genscan.contigX.txt` where `X` is each contig submitted to GENSCAN (this will make sense below)
4. `peptides.fasta`
5. `FLAG.txt`

Plan

One of your research colleagues, Dr. Klum C. Mann, has taken a human chromosome and cloned a small region of it into a bacterial artificial chromosome (BAC). This cloned region was then repeatedly "shredded" into even smaller random fragments whose lengths are all known to be 2000 ± 10. Those random fragments were then sequenced an average of 500 base pairs from both ends by a DNA sequencer, resulting in 391 mated pairs of reads (782 reads in total). His task is to assemble these reads into the original BAC-cloned chromosomal region, and then analyze the genes within it.

Because the DNA sequencer can produce reads from both ends of a fragment, and thus generates reads in opposite orientations, one of the reads in each pair has already been reverse-complemented so that the two reads in each pair are both being reported in the same orientation. Moreover, Dr. Mann has somehow managed to ensure that *all* the pairs of reads are reported in the *same* orientation with respect to the original genome. In other words, we can be confident that the reported reads are all taken from the same strand of the human chromosome. This represents quite a simplification over the situation that typically prevails in a sequencing project because when our genome assembly algorithm checks for overlaps, it need not check every read with the reverse complements of the other reads. It also means that all contigs (long stretches of contiguously assembled nucleotide sequences) and supercontigs (sets of contigs with gaps between them but those gaps are 'spanned' by a mated pair of reads, with one read in one contig and the other read in the other; these supercontigs are sometimes also called 'scaffolds') can be correctly oriented (but not ordered) with respect to one another. For reference, you may study the cartoon provided in Fig. 1 to understand the distinction between contigs and supercontigs.

Dr. Mann had the 782 sequences written down on note cards, along with their IDs. Unfortunately, on the way to your office, he slipped and dropped the whole container of note cards! To make things worse, when he bent down to pick them up, a stack of note cards with irrelevant reads from a separate sequencing project slipped out of his shirt pocket and into the pile. Not one to be easily deterred, Dr. Mann gathered up all the cards and placed them back in the container in a random order. Arriving to your office, he placed the container on your desk and promptly discovered that he was late for a meeting and just disappeared, leaving the container of note cards behind.

Luckily, you had a scanner with powerful OCR (optical character recognition) that you used to digitally convert the sequences and their names into one gigantic FASTA file called `paired.reads.fasta`. This file

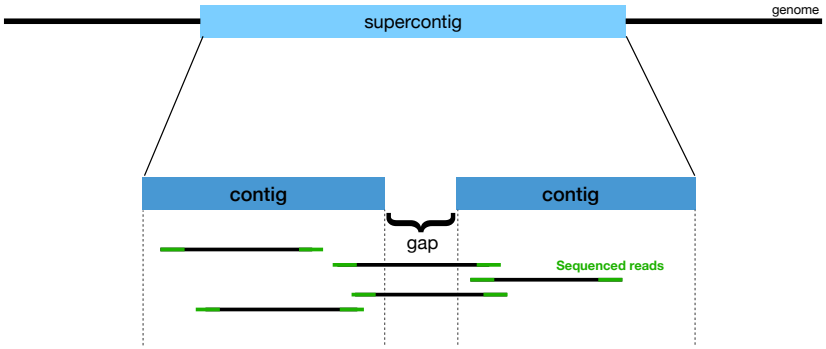

Figure 1: A supercontig formed from two contigs. The two ends of each fragment, highlighted in green, are sequenced to form pairs of reads (the black nucleotides between them are not sequenced). During assembly, these green reads can be stitched together into contigs when they overlap sufficiently (in this picture, there would need to be many more green reads covering the full length of the contigs that are shown; we are streamlining the picture so it isn't too crowded). The two reads in a given pair will often be located in the same contig, but sometimes they will instead 'span' across two contigs because none of the other reads happen to fall in the gap between the contigs. These spanning read pairs allow us to collecting contigs into supercontigs, and they also give us some information about the length of the gap between consecutive contigs in a supercontig.

now contains 842 reads (421 pairs of reads), which means that 60 reads (30 pairs of reads) from the separate sequencing project must've been mistakenly mixed in.

You next ran a shotgun sequence assembly algorithm to assemble the reads, and this algorithm was able to find an assembly that properly incorporated most of the reads (the reads from the separate sequencing project didn't have evidence of overlap with other reads in the set, so they did not make it into the final assembly, appropriately). The output of the assembler is a list of contigs. In this case, you got four contigs, each provided as one of the reads in `contigs.fasta`.

a) Figure 1 illustrates some of the ideas we have discussed about the arrangement of reads, contigs, and supercontigs. In a situation such as that in Figure 1, where a gap of unsequenced nucleotides exists between two consecutive contigs within a supercontig, how could one most accurately estimate (give bounds for) the length of the gap between two contigs?

Develop

b) Write a small Python program called `assembly_tester.py`. The purpose of this program will be to assess how a collection of paired reads fits into an assembly (in the form of a collection of contigs). It should check two things for every pair of reads:

- First, whether both reads appear in the correct orientation somewhere in the set of contigs.
- Second, whenever both reads appears within the same contig, whether the distance from the beginning of the first read to the end of the second read is indeed 2000 ± 10.

In short, your program should verify that the assembled sequence is correctly incorporating the information in the reads. If either of the above conditions are violated for some pair of reads, your program should report the read and which check failed. The remaining pairs of reads will necessarily pass both of the above checks, and your program can simply output the number of those read pairs (ones that are consistent with the assembly; you don't need to list them all because the majority will pass the checks). You may assume that whenever a mated pair of reads appears in different contigs, the first read in the pair (labeled `a`) appears in the genome before the second read in the pair (labeled `b`).

Your program should check these conditions using the output of a function called `find_contig_reads()`. This function exists to pre-process all the reads and return info about where in the assembly each is found. It should take as input the collections of reads and contigs as dictionaries, each of which will be returned by `get_fasta_dict()`. It should return a dictionary mapping each read pair's name (with the end character distinguishing `a`/`b` removed) to: the contig, start location, and end location within that contig to which the `a` and `b` reads mapped. As always, locations should be 1-indexed, and if the read cannot be mapped, indicate this with the special Python value `None`. The returned dictionary should look something like this:

```
{
    'seq1': {
        'contig_a': 'contig1',
        'start_a': 301,
        'end_a': 800,
        'contig_b': None
        'start_b': None,
        'end_b': None,
    },
    'seq2': {
        'contig_a': 'contig2',
        'start_a': 1101,
        'end_a': 1600,
        'contig_b': 'contig1'
        'start_b': 201,
        'end_b': 700,
    },
    'seq3' : {
        'contig_a': None,
        'start_a': None,
        'end_a': None,
        'contig_b': None
        'start_b': None,
        'end_b': None,
    },
    ...
}
```

Apply

In the context of an actual sequencing project, because of sequencing errors, repeats, and chimeric cloned inserts, the assembly may not always agree with the information present in the reads. Therefore, one should not expect in general for the assembled sequence to be consistent with the entire set of reads (even when reads from a different sequencing project were not accidentally intermingled). In such settings, an assembly testing program like the one you just wrote can be used to see how well the assembler is doing at incorporating the read data.

c) In the context of this problem, we know that some read pairs don't belong to the assembly, and your program `assembly_tester.py` should be able to identify which pairs they are. Process the output of `find_contig_reads()` to **report** the reads that don't belong to the assembly. Also, identify and **report** any other exceptions to the checks you performed.

d) Using the output of `find_contig_reads()`, identify which pairs of contigs have read pairs spanning them. Use this knowledge to determine how many supercontigs there are (you note that this should be less than or equal to the total number of contigs), which supercontigs contain which contigs, and the order of the contigs within their respective supercontigs. Identify the supercontig that contains contig1; we will call this "supercontig1", and **analysis in the remainder of the problem set will revolve around supercontig1**.

e) Further process the output of `find_contig_reads()` to **report** the number of mated pairs of reads where the first read (the `a` read) maps to one contig and the second read (the `b` read) maps to another contig (both in supercontig1): these are the properly-oriented read pairs that span the gap(s) between contigs in supercontig1. For each pair of reads that maps this way, **report** the possible length of the gap between the two contigs (lower and upper bound), using the knowledge that these two reads originally came from the two ends of a single genomic fragment of constrained length (it can help to draw yourself a picture here).

Your report should look something like this:

Read name	Gap size range	Lower bound	Upper bound
seq42	300±50	250	350
...			

f) Based on the various values reported in the previous problem, calculate the size of any gap(s) that exist between the contigs of supercontig1. Show your work.

g) Dr. Mann returns to your office precisely as you are putting the finishing touches on your `assembly_tester.py` program. "Well, that's fine timing," you mutter. He hears you and smiles weakly, but goes on to say that he's a little unclear about how to find all the genes contained in this sequence. "We can't simply apply an ORF-finder here, because of that whole exon-intron-splicing thingee. I'm really at a loss for how to proceed." With that, he turns and leaves your office, mumbling something about having to teach a class.

Fortunately, you've already taken COMPSCI 260 (wait: is this the movie *Inception*?) and you know that a hidden semi-Markov model can be used to decode a genomic region to look for genes, even those with exons and introns. Unearth the ancient MIT web server that implements the GENSCAN algorithm for doing exactly this (`http://hollywood.mit.edu/GENSCAN.html`) and use it for this problem (many of the links there are broken, but the tool still works; the paper describing GENSCAN is available from the Resources tab on the COMPSCI 260 web page).

Note that the GENSCAN HMM is *not* constrained to start in the state corresponding to background nucleotide distribution (state N) but can start in any state (i.e., the initial probability for every one of the states is nonzero). *Explain why this fact is relevant when you go to submit your specific assembled sequence.*

h) Submit your assembled sequence from supercontig1 to GENSCAN. If supercontig1 has more than one contig, you will need to submit each contig separately (GENSCAN cannot leverage the information that contigs are bridged into supercontigs, so if there are multiple contigs, just submit them one at a time and ignore the fact that they are organized into one supercontig). Save the results for each contig into a text file (call each of them `genscan.contigX.txt`, where `X` is the corresponding contig number). Submit these files.

Reflect

i) Just as you are finishing up with the results from GENSCAN, Dr. Mann pops in his head through your door and asks, "How's it going?" You try to show him the output on your screen, but he seems to be in a great hurry as usual. "Could you just write up an explanation for me and include it with the results?" *Please write up an explanation for Dr. Mann sufficient for him to be able to interpret the GENSCAN output* (the goal here is to ensure you understand it yourself sufficiently well to explain it to someone else).

j) Now, across all the contigs you submitted, let k be the total number of protein-coding genes predicted by GENSCAN. What is k? For each of those k predicted genes, how many exons are predicted? Save all the predicted peptide sequences in one FASTA file `peptides.fasta`.

k) Are the predicted number of exons typical of genes in the human genome or atypical?

l) Once again, Klum's timing is impeccable: As soon as you have prepared the `peptides.fasta` file, he's knocking at your door. It's almost as if he's got some preternatural sense for keeping you maximally busy (this is what grad school is like, by the way). Anyway, he tells you he thinks your peptides may be related to one another. But he's also worried about the fact that each may not be a full protein. So he asks you to study them in relation to one another, initially as peptide fragments, and then to figure out the corresponding full-length proteins. He starts to sing a ditty with valines and tryptophans as the main characters so you tune him out as you get to work.

Submit the k peptide sequences to BLAST to determine their identities, using `blastp` and the UniProtKB/Swiss-Prot protein sequence database. Given that we know these genes are from the human genome, are the top hits returned by BLAST believable? Which of the peptide sequences represent full-length proteins, and which represent fragments? How does this relate to their locations in the two contigs?

m) You should observe that three of these peptides return BLAST matches that are similar to each other in some way. In what sense are the three proteins similar to one another? What do these particular types of proteins do in the cell?