PR2: Genome Assembly: Contamination & Correction

CS 182/282 Spring 2026

Released: Wednesday, February 25th, 2026

Due: 11:59pm on Tuesday, March 17th, 2026

Overview

In this PR and the next you will develop your own genome assembler. At the end of both PRs, you will have implemented a program which can take in a set of reads from a text file and output a De Bruijn graph that will correspond to your given reads after decontamination and error correction, along with the most likely target sequence associated with those reads. Note that the assembler implemented in these PRs is a simplified version of an actual genome assembler, but involves many of the important concepts underlying genome assembly. This PR will focus on quality control of sequencing data; namely, the implementation of algorithms to remove contamination from reads and correct errors due to missequencing of bases.

This assignment is worth a total of 100 points.

Setup

To access test case files and data for P2, download the zip files from the assignments page of the course website.

README

Your README should contain the following information:

- the command calls for any scripts required to run your program
- a general overview of your program's methodology
- a description of any auxiliary files you created or employed
- any known bugs
- anything else you want the TAs to know about your handin

Do not include any identifying information in your README.

Gradescope

To hand in your project, submit all of your project files to Gradescope. Make sure that any required shell scripts are in the root directory of your submission (or uploaded as individual files). We have made a small number of test cases visible to you after you submit so that you can ensure your program is working properly.

Our Gradescope autograder is currently configured to accept solutions written in Python 3, Java, and R. If you plan to use a different programming language that is installed on the department machines, you must let the TA staff know as soon as possible so we can update our autograder settings.

P1: Contamination (30 points)

Your first task will be to implement a contamination removal algorithm. The NCBI defines a contaminated sequence as "one that does not faithfully represent the genetic information from the biological source organism/organelle because it contains one or more sequence segments of foreign origin."

There are several ways by which a DNA preparation/sample may become contaminated:

- A culture of human DNA or protein in the lab may become infested by microbes (unbeknownst to researchers). The microbes may then contaminate the nucleic acid or protein preparation, and this contamination may propagate to the sequencing phase, complicating accurate alignment and assembly.
- Human DNA floating around in a lab may contaminate cultures or DNA/protein preparations if the target DNA and contaminant DNA originated from different regions of interest in an experiment.
- Vectors (e.g., plasmids) are often employed to generate many copies of specific DNA or protein sequences. In these cases, vector DNA may very easily contaminate the extracted product sequence.

Contamination in sequenced reads should be removed before alignment and assembly begin, as it can introduce complications during the crucial final stages of the process. Typically, a BLAST-like search is conducted to remove regions of reads that are highly similar to a known vector's DNA or the DNA of an organism other than the one from which the target sequence originated. You can read more about vector contamination on the NCBI site.

Detecting Contamination

This PR assumes a very specific definition of contamination; namely, that only the ends of reads may have originated from some foreign source. In other words, there will not be contamination in the middle of reads.

For the purposes of this PR, contamination is defined as a contiguous subsequence of the contamination source of **length greater than or equal to *k*** located at the beginning **or** end of a read. All such segments of reads should be removed (note that contamination may be present at one, neither or both ends of a given read). You may assume that only one contaminated segment is present on either side of a read.

You should use a BLAST-like approach to detect contamination at either end of each read, and remove the **longest** region of contamination possible whenever DNA from the vector source is detected. Note that an initial contaminated "seed" of length k may occur multiple times within the contamination source!

Implementation

You should submit a shell script called `contamination.sh` which can be run as follows:

```
$ sh contamination.sh <reads.txt> <vector.txt> <k>
```

where `reads.txt` is a text file containing a list of DNA reads of constant length (separated by newlines), `vector.txt` is a text file containing a single DNA sequence originating from the source of contamination, and k is a positive integer which represents the minimum length of contamination to be removed.

When called as above, the script should print out the following on successive lines:

- All indices (zero-indexes) in the input read file corresponding to reads that were contaminated with the vector DNA, each separated by a comma (no spaces), and sorted in ascending order
- 20 dashes (-)
- Each read in the **same order** given in the input read file, one read per line, with any contaminated segments removed (note that every read should be left-justified, regardless of length)

Test Cases

```
$ sh contamination.sh contam_reads1.txt vector1.txt 3
0,1,2,3
--------------------
TGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCTG
TGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
$ sh contamination.sh contam_reads1.txt vector1.txt 4
3
--------------------
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCTGAAA
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

You are encouraged to develop additional test cases to confirm that your program works as intended on more complicated inputs. Consider the range of possible edge cases and complexities that may complicate the decontaminated output!

You may find it helpful to break down your code into functions that accomplish the following tasks:

- generating all k-mers from the vector sequence
- carrying out a BLAST-like "seeding" phase at the ends of each read
- carrying out a BLAST-like "extension" phase from all seeds
- removing all maximal contiguous contaminated subsequences from the ends of each read

P2: Correction (30 points)

Your second task will be to implement an error correction algorithm. Multiple types of errors can arise in the context of genome assembly, and many methods have been developed to correct for these errors during the alignment and assembly process. The method described below is a commonly-used approach to correcting errors which may occur in the sequencing process.

Detecting Errors

Consider the set of all k-mers present in a set of reads, such that each k-mer appears as many times as it occurs across all of the reads. It may be observed that certain k-mers will only appear a few times in this set, while most k-mers will appear a relatively high number of times. The latter observation can be explained by the concept of sequencing depth (coverage), while the former indicates that errors may have occurred during the sequencing process.

Assume that "infrequent" k-mers (those which appear **less than** some threshold t times) are the result of sequencing errors, and correct these k-mers (and the reads from which they originated) by replacing them with "frequent" k-mers which differ from them at **at most** d positions. If there are multiple such "frequent" k-mers, choose the most frequent one which differs from the "infrequent" k-mer at the **fewest** positions (choosing randomly in the case of a tie). If there are no such "frequent" k-mers, do not replace the "infrequent" k-mer.

Note one subtlety that may arise during this process: when replacing "infrequent" k-mers, you may observe that the sequences of the neighboring/overlapping k-mers in a given read will be altered. This may introduce difficulties if other "infrequent" k-mers are "overwritten", preventing their being replaced later. There are multiple ways to resolve this issue; you may choose how to address this phenomenon, bearing in mind the biological and statistical principles behind sequencing technologies. **You should detail your chosen process and a brief rationale for its methodology in a new file named** `application.txt`.

Implementation

You should submit a shell script called `correction.sh` which can be run as follows:

```
$ sh correction.sh <reads.txt>
```

where `reads.txt` is a text file containing a list of DNA reads of constant length (separated by newlines). As opposed to being inputs to your program, you should tune the values of k and t yourself (both should be positive integers). We recommend setting $d = 2$.

When called as above, the script should print out each read in the **same order** given in the input read file, one read per line, with any corrections made

For this portion of the project, you will be graded on your program's ability to correct sequencing reads generated from the HIV genome with a 1% error rate. If your program improves the number of sequencing errors, you will receive full points for this section. Specifically, this means that the log comparison score between the corrected reads outputted by your program and the mutated reads generated from the HIV genome must reach a particular threshold, which you should think about before submitting. Please read the application section for a description of the log comparison score.

Test Cases

```
$ sh correction.sh error_reads1.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

While we have provided you this test case in the project files, do not get too stuck on it (you won't be graded on it). The correction process changes significantly when we increase k and t (think about why!). You might find that, for low k, your chosen method of dealing with replacement will fail to produce as clean an output as the above. However, for higher k (> 15), you might find that your program does better on this particular case (but maybe worse, on the application dataset). There are a lot of complexities to be considered here, but keep in mind that we will only be grading the performance of your correction script on the faulty reads generated using the process described in the application section. Completing the application section will be very useful in helping you to determine the optimal values of k and t for your program.

You may find it helpful to break down your code into functions that accomplish the following tasks (and which can be compartmentalized for easy access in the application):

- tallying all unique k-mers in the set of reads
- determining "infrequent" and "frequent" k-mers (based on the value of t)
- selecting "frequent" k-mers to replace each "infrequent" k-mer (based on the value of d)
- replacing all "infrequent" k-mers according to your chosen method

Supplementary Material

If you're interested, this seminal paper by Pevzner et al. introducing an Eulerian path assembly algorithm describes how error correction is used during genome assembly (pp. 3-4).

P3: Application (40 points)

Your final task will be to apply your correction algorithm to a real-world example and experiment with the parameters involved in error correction to identify their optimal values. The following problem will provide experience in a fundamental process in the field of computational biology: integrating multiple workflows with different inputs and outputs into a cohesive scientific exploration.

All genome sequencing technologies—from Sanger sequencing to whole-genome shotgun sequencing to next-generation methods—have non-trivial error rates, ranging from 0.1% to 2% of all bases sequenced. Thus, error correction is a critical part of any genome assembly.

Because you were given freedom in choosing how to handle the subtleties of error correction in P2, optimal parameters may differ across implementations. In this problem, you will determine optimal values for the parameters k and t (set $d = 2$) specific to your implementation by simulating a genome assembly experiment using real data. There are many possible results for this investigation, but note that insightful trends should become apparent with any reasonable implementation.

You will receive as inputs the following files:

- `hiv-1_genome.txt`: the 9,181 bp RNA genome of HIV-1
- `unitary.m`: a DNA scoring matrix (with all letters in uppercase, all rows separated by newlines and all columns separated by spaces, and an `X` in the top-left cell)

You should submit one or more code files which carry out the following:

- Simulate extraction of 50-mer reads from the HIV genome
- Mutate these reads randomly based on a 1% error rate
- Correct these reads using your scripts from P2 for various values of k and t (you may find it useful to implement helper functions and some method of storing values across multiple trials)

You should also submit the following output files:

- `true_reads.txt`: This file should contain 50-mer reads (one per line) randomly drawn from the HIV genome. You should sample enough reads to ensure that at least 99.9% of the target sequence is expected to be covered by contigs. Use Lander-Waterman statistics (see CH2 notes for review) to compute the minimum number of reads required (N_{min}) and **report this number** in `application.txt`.
- `mutated_reads.txt`: This file should contain the same 50-mer reads from `true_reads.txt` (in the same order, one per line), with a global error rate of 1% introduced. Errors should be introduced randomly with respect to both location and identity of the mutated base. You may choose how to implement this process. Compute the average ungapped global alignment score of each true/mutated read pair (S_m) using the provided scoring matrix, and **report this number** in `application.txt`.

To determine optimal parameters for maximizing correction efficiency, run your error correction code on `mutated_reads.txt` over a range of values for k and t, chosen at your discretion to balance performance with runtime (you may find that some choices of k and t greatly impact the feasibility of your correction and lead to very long runtimes). Choose at least 3 values for k and t. For each of these trials, compute the average ungapped global alignment score of each true/corrected read pair (S_k) using the provided scoring matrix, and record the quantity

$$-\log\left(\frac{50 - S_k}{50 - S_m}\right)$$

Feel free to include a plot of these values in your final application to support your answers to the qualitative questions at the end of this document. Choose your optimal parameters from this search, and report $-\log\left(\frac{50-S_k}{50-S_m}\right)$ for this pair of parameters in `application.txt`. We require that this score be **above the threshold that indicates your correction reads are an IMPROVEMENT upon the mutated**

reads. Think about what this number is, and report your score below. Passing this threshold is a significant part of your grade for P2 and P3.

Finally, you should submit a file called `application.txt`, containing the following:

- a description of why you implemented correction in the way that you did
- the values of N_{min}, S_m, your optimal pair of parameters k^* and t^*, and a value $-\log\left(\frac{50-S_k}{50-S_m}\right)$ for these optimal parameters.
- answers to the following questions:
    1. What general trends do you observe in varying k and t? Justify your claims using the data you generated. Give an explanation (either mathematical or intuitive) for why modulating each variable produced these results.
    2. How is the choice of t dependent on k? Are there any values of k for which varying t does not produce a significant difference in correction efficiency? Why might this be the case?
    3. What feature(s) of the data are the parameters k and t dependent on? Give an explanation (either mathematical or intuitive).
    4. Discuss the impact of varying k and t on the running time of your correction script on the HIV-1 genome. For what values of k and t is the running time untenable, with your implementation? What statistical reason might cause this?