Note: This is part one of a two part assignment. Much of the code you write for this assignment will be used in part two. This assignment is due by **11:55pm on Mon, April 6**. This part of the assignment is worth **17** points. If you choose, you may work with a partner on this assignment. You must submit your code and any other requested files as a zip file to Moodle.

1 Project Overview

In this project we will explore the process of genome assembly. Genome assembly of real DNA sequencing data is still a challenging area where many researchers are currently working. Thus, this assignment will focus on a simplified version of the problem using simulated data. As an overview, in part one of this assignment you will do the following:

- You will create a simulator that emulates the process of sampling reads from a longer piece of DNA (similar to the output of a DNA sequencing experiment).
- You will construct a De Bruijn graph from a set of reads. (Note, in part 2 of the assignment, you will modify and use this graph to complete assembly tasks).

You must use Python for this assignment. If you'd like to use another programming language (C, C++, Java, etc.) you must talk to me first. Any data files mentioned below can be downloaded from the link on Moodle (HW 5 Files). This directory also contains examples of any file formats mentioned below. Lastly, **please read this whole document carefully before beginning.**

2 Project Specifications

Note: I do NOT expect you to be able to assemble "full size" genomes as that would require more RAM than is available on most personal machines. You should be able to handle data created for the `sample.fasta` file provided.

2.1 Simulating Data

You will need to provide the following Python script: `simulate.py` that simulates DNA sequencing reads. I want to be able to run your program from the command line by typing something like the following:

```
$ python simulate.py example.fa 30 50 0.01
```

I have provided starter code in the HW5 Files folder that shows you how to do this (`simulate_starter.py`). This file also includes code to read a fasta file; you have written this function before, but I wanted to make sure everyone used the same `read_fasta` function.
`simulate.py` will take in the following parameters (in this order):

- FASTA sequence file (string) - File containing a DNA sequence.
- Coverage (integer) - The coverage of the simulated sequencing experiment.
- Read length (integer) - The length of reads to simulate.
- Error rate (float) - The sequencing error rate (between 0 and 1).

`simulate.py` will simulate N reads from the DNA sequence in the supplied FASTA. You will use something called Lander-Waterman statistics to compute the value of N. In particular, if you have a genome of length G, coverage of C and read length of L, then you will compute N as follows: $N = \frac{C \cdot G}{L}$. Ex: $G = 100, C = 5, L = 50$, then $N = 10$, meaning you should create 10 reads of length 50, randomly sampled from the original genome. Reads should start at random indices in the genome (that is, they are uniformly distributed across the genome) and will all have the same orientation (that is, we are ignoring the existence of the reverse complement in DNA). Make sure that reads that start near the end of the genome still contain exactly L characters.

An error (incorrect nucleotide) should be introduced at every position in every read with probability equal to the specified error rate. The allele of the incorrect nucleotide is chosen from the other three alleles with equal probability. For example, suppose your error rate is 0.01. For each position in each read you will emulate "flipping a coin" with a 99% chance of landing on tails and a 1% chance of landing on heads. If the coin lands as heads (let's say on a character `A`), you will swap the character to one of the other characters with equal probability (so, here it will change to either `G`, `T`, or `C`).

You will need to utilize the `random` library in Python (`import random`). Take a look at the Python documentation: `https://docs.python.org/3.8/library/random.html`.

Your program will output each simulated read as a separate line in a file called `reads.txt` in the current directory. For example, running:

```
   $ python simulate.py example.fa 30 50 0.01
```

will create a file called `reads.txt` that may look like the following:

```
TAGCACCACTTCTGCGACCCAAATGCACCCTTTCCACGAACACAGGGTTG
TCCGATCCTATATTACGACTTCGGGAAGGGGTTCGCAAGTCCCACCCTAA
ACGATGTTGAAGGCTCAGGTTACACAGGCACAAGTACTATATATACGTGT
```

2.2 Constructing a De Bruijn Graph

You will need to provide the following Python script: `assemble.py` that constructs a De Bruijn Graph from a set DNA sequencing reads. (You will add more to this part of the assignment in part 2).

`assemble.py` will take in the following parameters (in this order):

- Reads file (string) - A file of reads, as output by `simulate.py`.
- k (int) - the size of k-mer to use when building a De Bruijn graph.

`assemble.py` will construct a de Bruijn graph for the specified k-mer size from the set of reads contained in the supplied `reads.txt` file. Once the graph is constructed, your program should print the following information to the screen: (1) The number of vertices in the de Bruijn graph; and (2) The number of unique edges in the de Bruijn graph. So, if you have an edge from vertex v to w with multiplicity 2, this edge will only count once towards the total number of edges.

For example, running your Python script might produce the following output (for a very small input).

```
$ python assemble.py reads.txt 3

Number of vertices in graph: 5
Number of edges in graph: 5
```

As a reminder, a de Bruijn graph for a specified k-mer will have vertices that represent sequences of length $k - 1$.

2.2.1 Optional: Visualizing the graph

This section is **optional**; adding code to your program that outputs a .dot file will earn you up to 4 additional points on this assignment.

It is often very useful to create something called a DOT file that allows you to visualize the de Bruijn graph (see below for instructions on how to do this). However, for large and complicated assemblies this file may not be very interpretable, but it should be useful for debugging with smaller examples. **Your program should automatically create a .dot file (in the same directory as your code) for all de Bruijn graphs it constructs if they contain less than or equal to 30 vertices.**

The DOT language provides an easy way to make visualizations of graphs (the wikipedia page also gives a nice overview). Suppose, your code produced a file called `example.dot` in the current directory that looks like the following:

```
digraph mygraph{
    "ATG"->"TGC"
    "ATG"->"TGG"
    "GTG"->"TGG"
    "TGG"->"GGC"
}
```

In the DOT format, a directed edge between two vertices is indicated by `->` (we can also represent undirected edges with `--`. The name of the graph (mygraph in the above example) does not matter. In fact, the dot files you produce will look very much like the above example, but with different sequences. This makes it very easy for you to visualize your output. For example, when your program outputs a file called `example.dot` you can easily visualize this tree using dot program, by either opening the .dot file with the GraphViz software on your computer, or typing the following command into the terminal (depending on your OS and how you installed the program):

```
   $ dot -Tpng example.dot -o example.png
```

This will produce a graphic image in PNG format, and save it to a file called `example.png`. For instance, the above command, when run on the example DOT file above, will produce the following PNG (Figure 1). You can get the dot program by installing graphviz.

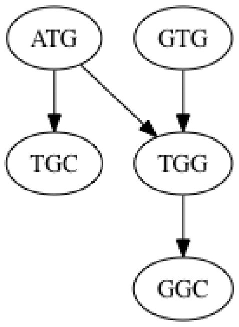

Figure 1: The graph created from the DOT file above.

3 Data Analysis

Complete the following steps and report your answers to any of the questions asked in a PDF you submit with your assignment.

1. Using the FASTA file called `sample.fasta` in the data directory for this assignment, create a set of reads using the following parameters: coverage = 12, read length = 50, error rate = 0.0. Save this as a file called `sample_c12_r_50_e0.00.txt` and include the file in your submission.
2. Construct a de Bruijn graph for the dataset created in the previous setp using k = 13 and report the number of vertices and edges in your graph.
3. Using the FASTA file called `sample.fasta` in the data directory for this assignment, create a set of reads using the following parameters: coverage = 12, read length = 50, error rate = 0.01. Save this as a file called `sample_c12_r_50_e0.01.txt` and include the file in your submission.
4. Construct a de Bruijn graph for the dataset created in the previous step using k = 13 and report the number of vertices and edges in your graph.
5. Give a short description (a few sentences) for why the number of edges/vertices are either similar or different between these two simulations.

4 Suggestions

- I suggest doing lots of testing of your code on smaller input files first before trying to run on the FASTA file I provided you. It is much easier to debug on smaller input files. You might even want to think about how to create a "perfect" set of reads to help with debugging.
- Carefully commenting your code and thinking about how you design your code will be helpful for the second part of this assignment. In this part of the assignment you will likely be extending your existing code. For example, you may need to perform some graph simplifications or do traversal through your graph. Keeping this in mind now, will help you later.
- I will only test your code with valid input (e.g., the provided error rate with be between 0 and 1, inclusive), but it is always good practice to think about how you handle invalid input with your code.

5 What to hand in

Please hand in to Moodle a zip file containing all of the following:

1. The two Python scripts as described above.
2. A README that explains the following:
    - How to compile your code (if necessary) and any dependencies (i.e. Biopython).
    - A description of any known bugs.
    - Any major design decisions that may affect the output of your programs.
    - If you attempted the .dot file output for extra credit and if you got it working.
    - The names of anyone you discussed the project with and any outside resources you consulted when writing the code.
3. A PDF that includes the information requested in the Data Analysis section.

6 Grading

This project will be worth 17 points. These points will be assigned according to the following five categories.

- **Data Simulation Program (6 points)** - I will analyze your code to ensure that it adheres to the specifications outlined in this document. This is where clear commenting will be useful to you as this will be graded only by looking at your code.
- **De Bruijn Graph Constructions (6 points)** - I will analyze and run your code to ensure that you are correctly building the de Bruijn graph and adhering to the specifications outlined in this document.
- **Data analysis (3 points)** - Did you include the requested files and answers in your submission?
- **Comments, style and design (2 points)** - Make sure that your code is well commented and any important design decisions are clearly documented in your README file.
- **Graph visualization extra credit (up to 4 additional points)** - I will test your output file. (4 if it outputs correct file for all inputs, 3 if it outputs correct files for some inputs, 2 if it outputs a mostly correct file, 1 if you output a file with a .dot extension but it is not correct).