1 Introduction

The most widely used DNA sequencing machines generate billions of sequences reads that are approximately 100 nucleotides long. The problem of assembling these reads into the DNA sequence that generated them is called the Sequence Assembly problem.

In this assignment, you will build a simple DNA assembler based on De Bruijn graphs. Your program will construct a De Bruijn graph from a collection of $k$-mers, and assemble the DNA sequence by finding an Eulerian path in the graph. In practice, the sequencing assembly problem is much more difficult than you will encounter in this project, because of complicating factors including: (a) Errors in the reads; (b) Repeats in the DNA sequence; and (c) Missing data. We will avoid these complications by making the following simplifying assumptions about our data.

1. The given reads are all $k$-mers of the original string; i.e. there are no missing or erroneous reads.

2. Reads were produced using a sliding window of size $k$, and were not randomly sampled.

Because of these simplifying assumptions, the assembler that you write will likely not produce good results on real data. However, your program has the first ingredients for a bona-fide assembler.

2 Task

You will implement a program that takes in a file containing $k$-mer reads. Your task is to:

1. Parse the reads from the file.

2. Construct the de Bruijn graph corresponding to those reads.

3. Attempt to reconstruct the original sequence by finding an Eulerian Path through the de Bruijn graph. This project asks you to resolve the graph in a specific way, so it is very important that you read this document carefully before starting your implementation.
3 Project Specifications

Important: make sure your code works on department machines, even if you developed code on your own computer.

In addition to the general specifications given in the first project’s handout, please adhere to the following:

3.1 Shell script

You should provide a shell script, assembly.sh, that takes in a parameter as follows:

```
./assembly.sh [reads file]
```

and prints the assembled genome to the terminal window (stdout). For example, one might see something like this:

```
$> ./assembly.sh reads.txt
AAAAGCCGTA
```

If you are not using Julia, you will need to write your own shell script to provide this interface and describe in your README how to compile your code if necessary.

3.2 Input type

A typical reads file will consist of multiple lines of strings in some random order, where each line is a single read. For example, a typical file might look like:

```
CCGTAT
AAACTG
...
AACTGT
GTATAA
```

Some sample reads files are provided for you in /course/cs181/data/assembly.

3.3 Graph Resolution

Since each read has length exactly equal to $k$, you should generate $(k - 1)$-mers to construct the de Bruijn graph.
The assembled string is represented by an eulerian path through the de Bruijn graph. You should use the algorithm shown in lecture 21 to accomplish this, with a few minor modifications. For clarifications on this part, check the Piazza post that’s been posted and pinned on top.

- Steps 1) and 2) of the algorithm involves particular choices of vertices. In both of these steps, if there is no clear source node, select the one that comes first lexicographically.
- Step 1a) of this algorithm tells you to “select an unused edge e”. For every branch in a path, choose the edge that takes you to the node that comes first lexicographically.

These modifications should remove any randomness/arbitrarity of the execution of your algorithm.

3.4 README

As usual, your handin must include a README. It should explain the following:

- If applicable, how to compile your code.
- Your answers for section 4. Tell us the names of any files that we might be interested in looking at.
- A description of any known bugs.
- Any major design decisions that might affect the output.

4 Analysis

All files that you will need for this part are located in /course/cs181/data/assembly.

In the file sample_gene.txt, you will find a sample nucleotide sequence. Your task here consists of three major parts:

1. Write a function that outputs all $k$-mers of the string, for any arbitrary $k$. This means that you should generate reads using a sliding window of length $k$. For example, if the string is ABCDE and $k = 3$, your reads are: ABC, BCD, CDE.

2. (You don’t have to turn this part in.) Try running your assembler on the multi-set of 3-mers from your string. The string you generate will probably be different than the one you started with. There are a few reasons for this, but the biggest culprit is our dumb way of resolving ambiguous paths from section 3.3.
3. So, how many assemblies are there? If this number is high, we can’t really hope to get a correct assembly without doing something much smarter. Let’s find out how the number of assemblies \( N \) depends on the read length \( k \) for this particular string.

To do this, we can apply the BEST theorem to compute \( N \), given an eulerian de Bruijn graph.

(Note: This means that you will want to Eulerize the graph before you apply the formula stated in the BEST theorem. In theory, there should be a unique node with net +1 indegree, and a unique node with net +1 outdegree; join these using an edge.)

As a reminder, the theorem states that the number of eulerian circuits in a connected, Eulerian graph \( G \) is equal to

\[
t(G) \cdot \prod_{v \in V} (\deg(v) - 1)!
\]

where \( t(G) \) is the number of arborescences rooted at any arbitrary vertex; they are all equal for an eulerian graph.

In HW5, you computed \( t(G) \) using brute force for a very simple example. The challenging part is to compute \( t(G) \) efficiently for any arbitrary graph \( G \). If you’re using the support code, you can simply call `num_arborescence` to compute this quantity (refer to section 5 for the method header). For those of you who want to implement this yourselves, it is easiest to compute \( t(G) \) as the determinant of a matrix. We’ll post something on Piazza about this.

Using the string from `sample_gene.txt`, plot \( N \) as a function of \( k \), and output the plot as an image called `debruijn_plot.png`. Please use a plotting tool/library to do this. In Julia, you can use the PyPlot or Gadfly package. (If there is enough demand, we will put up examples of how to use these libraries.) For Python users, the `matplotlib.pyplot` module would be a suitable choice.

4. Does the plot match your expectations? What is the general relationship between \( k \) and \( N \)?

5  Support Code

5.1 Stencil Code

To obtain the Julia stencil code, run:

/course/cs181/bin/cs181_setup assembly

This will create a project directory, `assembly`, for you to work in. Your goal is to fill in the empty function bodies in `assembly.jl`. 

4
5.2 The MultiGraph Package

We’ve provided for you a new graph package to use for this project: MultiGraph. Functionally, this is the same as LabeledGraph from the last assignment, but we’ve modified it to allow edge multiplicities (so that we can simulate having many copies of the same edge).

Note that this implementation does NOT provide a distinct edge instance for each copy - it merely stores a number to remember how many copies of it there are. We’ve provided the interface for having edge properties for MultiGraph just like LabeledGraph. One thing you may want to make use of is graph_adjacency, but as with everything else, there are many ways to approach this problem.

Here is a summary of the functions that we believe you’ll need to complete this project. If there are any basic features (aside from computing an eulerian path) that you think would come in handy but is not listed, post a request on piazza and we’ll see what we can do.

1. MultiGraph{VERTEX_TYPE}(is_directed::Bool) – Initializes an empty graph, where the vertices are of type VERTEX_TYPE.

2. add_vertex!(g::MultiGraph, v) – Adds the vertex v to the graph.

3. add_edge!(g::MultiGraph, v, w) – Adds a new copy of the edge v → w to the graph, adding the vertices v and w to the vertex set if necessary.

4. num_arborescence(g::MultiGraph, v) – Returns the number of arborescences (directed spanning trees) in the graph, rooted at vertex v. This function ignores self-loops in the graph.

5. vertex_index(v, g::MultiGraph) – Returns a number that represents v’s index in the graph’s adjacency matrix.

6. graph_adjacency(g::MultiGraph) – Returns a copy of the graph’s adjacency matrix M, with multiplicities. For example, if the graph has 3 copies of the edge A → B, then the matrix entry $M_{AB}$ will be 3.

7. edge_multiplicity(u,v,g::MultiGraph) – Returns the multiplicity of the edge $u \rightarrow v$.

8. vertices(g::MultiGraph) – Returns a list of all vertices in the graph.

9. out_neighbors(v, g::MultiGraph) – Returns a list of vertices u such that $v \rightarrow u$ is an edge in the graph. To get the number, it is preferred to use out_degree instead.

10. in_neighbors(v, g::MultiGraph) – Returns a list of vertices u such that $u \rightarrow v$ is an edge in the graph. To get the number, it is preferred to use in_degree instead.
6 Handin

You should hand in your code, shell script, and README file from a single directory. To hand in, navigate to your project directory, and type `cs181_handin assembly`. This will recursively hand in everything in your current directory.

7 Grading

TAs will grade your handin using pre-generated test cases.

You will be graded solely on the correctness of your implementation and adherence to the project specifications. You will not be graded on code performance, as long as it is within reasonable bounds.

Here are some common pitfalls to watch out for:

- The code does not terminate; it enters an infinite loop when traversing the graph due to cycles.
- Your assembled string does not produce the original $k$-mers, because some edges didn’t get used.
- Your assembled string is missing a few characters at the beginning or at the end.