CSE185-LAB2-PART2

July 17, 2023

1 Lab 2: So where do all those reference genomes come from anyway? (Sequence assembly)

1.1 Part 2 (40 pts)

Notes: * there should be "high memory" instances available to complete this lab, which should make the commands run faster. * Multiple commands in this lab might take a while to run. Strongly consider using "nohup" to run commands in the background so you can keep working through the rest of the lab while you wait for commands to finish. * To make sure you stay on the right track, many of the autograder tests are fully visible.

In this lab, we will assemble the genome of E. coli using two different sequencing technologies: short reads from Illumina, and long reads from PacBio, and compare the results.

1.2 1. E. coli sequencing datasets

We will be working with two E. coli sequencing datasets in this lab:

1. **Illumina paired-end short reads**. We used this dataset for kmer analysis in part 1. Recall you can find the two fasta files at:

```
~/public/lab2/shortfrag_1.fq
~/public/lab2/shortfrag_2.fq
```

2. **Pacbio "hifi" reads**. We will additionally be working with a long-read dataset generated using the PacBio hifi technology. You can find the fasta file in the same directory:

```
~/public/lab2/pacbio.fastq
```

We'll perform assembly using these two different datasets, and then use an online tool called QUAST to evaluate our results.

But let's first just get some basic info about the datasets we're working with.

You might find some of the following UNIX commands helpful in answering some of the questions below:

- `awk` is a general purpose tool for all sorts of data wrangling. Awk excels at parsing delimited data, where you have a lot of fields. The simplest form of an awk command is (all one line):

```
awk '/search_pattern/ {action to take on matches; another_action;}' [path to file]
```

For example, the following command prints out every other line of our fasta file (i.e., only the lines with sequences on them).

```
cat ~/public/lab2/pacbio.fastq | awk 'NR%4==2 {print length}' | head
```

Here `NR` means "row number" (or record number) and prints out rows where row number mod 4 is 2. `length` prints the length of the line. For files with many columns, you can also do `length($i)` to print the length of column `i`.

- `datamash` is a really helpful tool for cmputing simple operations on columns of data. Below is an example of how to use the command. Type `datamash --help` for more usage info.

```
cat ~/public/lab2/pacbio.fastq | awk 'NR%4==2 {print length}' | datamash mean 1 # print mean o
```

Question 1 (6 pts): How many read pairs are in the Illumina dataset? What is their length (in bp)? Based on a genome size of 4.6 million bp for the E. coli genome, what average sequencing coverage do you expect? Fill in the python variables with your answers below.

[5]:
```
shortfrag_numpairs = 14214324 # Fill this in with the number of read pairs
shortfrag_readlen = 100 # Fill this in with the read length in bp
shortfrag_coverage = 618.014 # Fill this in with the coverage

# your code here
```

[6]:
```
"""Basic checks on shortfrag_numpairs"""
assert(shortfrag_numpairs > 14000000 and shortfrag_numpairs < 20000000)
assert(type(shortfrag_numpairs)==int)

# Hidden tests check actual values
```

[7]:
```
"""Basic checks on shortfrag_readlen"""
assert(shortfrag_readlen > 50 and shortfrag_readlen < 500)
assert(type(shortfrag_readlen)==int)

# Hidden tests check actual values
```

[8]:
```
"""Basic checks on shortfrag_coverage"""
assert(shortfrag_coverage > 100 and shortfrag_coverage < 1000)
assert(type(shortfrag_coverage)==float)

# Hidden tests check actual values
```

Question 2 (6 pts): How many reads (single end) are in the Pacbio dataset? What is their *average* length (in bp)? What average coverage across the E. coli genome do you expect (again, assuming a genome size of 4.6 million)? Fill in the python variables with your answers below.

[11]:
```
pacbio_numreads = 95514 # Fill in with the number of reads
pacbio_mean_readlen = 14547.609 # Fill this in with the read length
pacbio_coverage = 302.065288 # Fill this in with the coverage

# your code here
```

[12]:
```
"""Basic checks on pacbio_numreads"""
assert(pacbio_numreads > 80000 and pacbio_numreads < 100000)
assert(type(pacbio_numreads)==int)

# Hidden tests check actual values
```

[13]:
```
"""Basic checks on pacbio_mean_readlen"""
assert(pacbio_mean_readlen > 10000 and pacbio_mean_readlen < 20000)
assert(type(pacbio_mean_readlen)==float)
# Hidden tests check actual values
```

[14]:
```
"""Basic checks on pacbio_coverage"""
assert(pacbio_coverage > 100 and pacbio_coverage < 1000)
assert(type(pacbio_coverage)==float)

# Hidden tests check actual values
```

1.3 2. Assemble reads with minia

First, we'll use a lightweight program called `minia`, which is based on de Bruijn graphs, to assemble the Illumina reads. Even though the questions above asked about the original sequencing datasets, for the assembly we'll be using the trimmed reads you generated in part 1.

To get started, type `minia` at the command line to see how to use it.

```
minia
```

`minia` needs a list of our files as input, so let's make a list of files to include. We'll use this opportunity to learn about some cool Jupyter "magics". Cell magics start with `%%`. The cell below uses the `%%file` magic, which writes the contents of that cell to the specified file. If necessary you can edit the file below to reflect actual paths to your trimmed fastq files if you named them something else. For assembling *contigs* with minia, we will use only the shortgrag library, which is what we used to get our kmer distributions using jellyfish last time.

[23]:
```
%%file ~/lab2/minia_input_files.txt
shortfrag_trimmed_1.fq
shortfrag_trimmed_2.fq
```

```
Overwriting /home/g1cole/lab2/minia_input_files.txt
```

Test that the above cell wrote your file correctly on the terminal:

```
cat ~/lab2/minia_input_files.txt
```

By the way, you can also run bash commands inside of your notebook. Either using a the `%%bash` magic or starting a line with an exclamation point `!` which we saw last week.

[24]:
```
%%bash
# Example bash magic cell
cat ~/lab2/minia_input_files.txt
```

```
shortfrag_trimmed_1.fq
shortfrag_trimmed_2.fq
```

[25]:
```
# Example running command using "!"
! cat ~/lab2/minia_input_files.txt
```

```
shortfrag_trimmed_1.fq
shortfrag_trimmed_2.fq
```

Now let's get back to running `minia`.

The most important parameter is the `kmer-size`. From our jellyfish histogram of the corrected data, we know that a kmer of 18 produces a defined 'true-reads' peak, and we know where the valley is. For the minia command, you only want to use k-mers with abundance higher than this valley point (`-abundance-min`).

Set the output prefix to `minia_assembly_18`.

Question 3 (3 pts): Run `minia` using a kmer size of 18 and min abundance 23. Set the value of `q3_cmd` to your command. How long did it take (in seconds)? Set the variable `q3_time` to your answer below.

[26]:
```
q3_cmd = """
minia -in ~/lab2/minia_input_files.txt -kmer-size 18 -abundance-min 23 -out␣
 ↪minia_assembly_18
"""

q3_time = 129 # set to the run time of your minia command (in seconds)

# your code here
```

[27]:
```
"""Check minia command and time"""
assert("minia" in q3_cmd)
assert("18" in q3_cmd)
assert("23" in q3_cmd)
assert("-kmer-size" in q3_cmd)
assert("-abundance-min" in q3_cmd)
assert(q3_time>60) # lots of variability, but should take more than 1 minute
```

Examine your assembly. The most important file minia created is the `minia_assembly_18.contigs.fa` file, which is a list of each assembled contigs in fasta format. Use `head` to look at the first few contigs.

Question 4 (5 pts): Gather some basic statistics about the contigs. Write one-line UNIX commands to answer the following:

- How many contigs are there?
- What is the longest contig length?
- What is the shortest contig length?
- What is the median contig length?

Set the variable `q4_cmds` to the commands you used. Set the variables `num_contigs`, `longest_contig_length`, `shortest_contig_length`, and `median_contig_length` to your answers to the questions above.

[34]:
```
q4_cmds = """
cat minia_assembly_18.contigs.fa | grep -c ">"
awk '/>/ {if (seqlen){print seqlen};seqlen=0;next}␣
 ↪{seqlen+=length($0)}END{print seqlen}' minia_assembly_18.contigs.fa | sort␣
 ↪-rn | head -n 1
awk '/>/ {if (seqlen){print seqlen};seqlen=0;next}␣
 ↪{seqlen+=length($0)}END{print seqlen}' minia_assembly_18.contigs.fa | sort␣
 ↪-n | head -n 1
awk '/>/ {next;} {print length}' minia_assembly_18.contigs.fa | sort -n | awk␣
 ↪'{a[NR]=$1;} END {print (NR%2==0)?(a[NR/2]+a[NR/2+1])/2:a[(NR+1)/2];}'
"""

# Set the variables to your answers from the above commands
num_contigs = 13540
longest_contig_length = 5374
shortest_contig_length = 18
median_contig_length = 177

# your code here
```

[35]:
```
"""Basic check on commands"""
assert(q4_cmds.strip() != "")
assert("minia_assembly_18.contigs.fa" in q4_cmds)
```

[36]:
```
"""Check num_contigs"""
# Note allow some wiggle room. Sometimes there are small differences
assert(abs(num_contigs - 13578)<100)
```

[37]:
```
"""Check contig lengths"""
# Note allow some wiggle room. Sometimes there are small differences
assert(abs(longest_contig_length - 5374)<100)
```

[38]:
```
"""Check contig lengths"""
# Note allow some wiggle room. Sometimes there are small differences
assert(abs(shortest_contig_length - 18)<10)
```

[39]:
```
"""Check contig lengths"""
# Note allow some wiggle room. Sometimes there are small differences
assert(abs(median_contig_length - 176)<50)
```

Question 5 (5 pts): Compute the N50 value of your short read assembly. You can use the approximate size of the E. coli genome that you computed in the last lab section. You may use whatever method you like to compute this based on your contigs fasta file. For example, you can

write your own script or UNIX command, or you may search for a tool online that can compute this for you.

Paste the commands you used in `q5_cmds` below. Report the estimated N50 in `q5_N50`. You should use *all* contigs to compute the N50.

[40]:
```
q5_cmds = """
 cat minia_assembly_18.contigs.fa | grep -v "^$" | awk '{if($0 ~ /^>/) {if (len␣
 ↪!= 0) {print len}} len=0} {if($0 !~ /^>/) {len += length($0)}} END {print␣
 ↪len}' | sort -rn | awk -v genome_size=4600000 '{sum+=$1}{if(sum>=genome_size/
 ↪2) {print $1; exit}}
"""

q5_N50 = 797 # Set to the estimated N50

# your code here
```

[41]:
```
"""Basic check on q5_commands"""
# Lots of ways to do this. Just check this was set to something.
assert(q5_cmds.strip() != "")
assert(q5_cmds.strip() != "Change to commands you used to get the N50")
```

[42]:
```
"""Basic check on q5_N50"""
assert(type(q5_N50)==int)
assert(q5_N50>700 and q5_N50<1000)
```

Question 6 (4 pts): We chose a specific value for k (18) for our `minia` assembly.

6.1: What do you think would happen if we chose a larger value of k?

6.2: What do you think would happen if we chose a smaller value of k?

Choose from:

- A: The sequencing coverage will go down, resulting in a lower quality assembly.
- B: We will be less likely to observe the same exact kmer multiple times since errors in reads will result in mismatches between kmers.
- C: Individual kmers will be less likely to be unique, resulting in a lot of unresolved loops in the assembly.

Set `q6_1` and `q6_2` to your answers below.

If you choose too short of a kmer, individual kmers will be less likely to be unique, resulting in a lot of unresolved loops in the assembly. If you choose too long of a kmer, if there are errors in the reads it will be unlikely to observe the same kmer multiple times, making it difficult to find reads to stitch together.

[43]:
```
q6_1 = 'B' # Set to answer to 6.1
q6_2 = 'C' # Set to answer to 6.2

# your code here
```

[44]:
```
""" Basic check on answer to 6.1"""
assert(q6_1 in ["A","B","C"])
# Hidden tests check actual answer
```

[45]:
```
""" Basic check on answer to 6.2"""
assert(q6_2 in ["A","B","C"])
# Hidden tests check actual answer
```

In a perfect world, we'd end up with an assembly of E. coli with only a single contig, since E. coli has only a single chromosome. You'll notice that our results with short reads still have a long way to go! (We ended up with thousands of contigs, and don't know how they should all be stitched together). In the next section, we'll see how much better we can do when performing assembly with long reads instead.

1.4 3. Installing Flye

Now, we will assemble the E. coli genome using long reads from the Pacbio Hifi platform. As we'll discuss in class, assembly methods that work for short reads (like from Illumina) will not work as well for long read data. For this section, we'll be using a different assembly tool, called Flye (see https://www.nature.com/articles/s41587-019-0072-8, developed right here at UCSD!).

To prepare you for your final project (and bioinformatics in the real world), we have not installed Flye, so you will need to install it yourself. Note, there are generally two ways that command line tools can be installed:

- "Global": tools that are installed globally are available to all users in the system. Usually you need to have root (sudo) permissions to install tools globally. We have installed many tools, such as `minia` and `bwa` so that you didn't have to install those on your own.
- "Local": tools that are installed locally are only available to the user that installed them. While you do not have permissions to install tools globally, you can usually install a tool locally just for yourself.

Head to the github page for Flye: https://github.com/fenderglass/Flye, and follow the link for "Installation instructions". Follow the instructions under "Installing from source" to install Flye. You should see the following instructions:

```
git clone https://github.com/fenderglass/Flye # make a copy of the repo in your local directory
cd Flye # change to the Flye directory
make # compile
```

To see if your install worked, type:

```
python bin/flye # this assumes you are in the Flye directory. If you are not, you'll need to cl
```

This should bring up a help message.

1.5 4. Assemble the PacBio reads with Flye

Now, we're ready to run Flye! Read through the Flye website, or the help message when you type `python bin/flye`, to see how to run it on our pacbio reads. Save the output to `~/lab2`. You can also use multiple threads (I used 4) on the high memory instances.

Note, this might take a little while. With 4 threads, my run took less than an hour. Recall, to run a process in the background you can do:

`nohup [command] &`

This will write everything that would have been printed to the terminal to `nohup.out` and will keep running even after you close your computer.

Question 7 (5 pts): Set `q7_cmd` to the command you used to run flye.

[1]:
```
q7_cmd = """
nohup python bin/flye --pacbio-hifi ~/public/lab2/pacbio.fastq --out-dir ~/lab2␣
  ↪&
"""
# your code here
```

[2]:
```
"""Basic checks on flye command"""
assert("flye" in q7_cmd)
```

[3]:
```
"""Basic checks on flye command"""
assert("--pacbio-hifi" in q7_cmd)
# Hidden tests
```

[4]:
```
"""Basic checks on flye command"""
assert("-o" in q7_cmd or "--out-dir" in q7_cmd)
```

Examine the Flye output. The most important files are:

- `assembly.fasta`: fasta file with the final assembly.
- `assembly_graph.gfa`: the repeat graph. See a description here: https://github.com/GFA-spec/GFA-spec/blob/master/GFA2.md of GFA format.
- `assembly_info.txt`: contains additional information about the assembly.

Question 8 (3 pts) How many contigs did Flye output? What is the N50? Set the variables `q8_numcontigs` and `q8_N50` to your answers below.

[5]:
```
q8_numcontigs = 1 # set to your answer
q8_N50 = 4653400 # set to your answer

# your code here
```

[6]:
```
"""Basic check on numcontigs"""
assert(type(q8_numcontigs)==int)
assert(q8_numcontigs > 0 and q8_numcontigs < 10)
# Hidden tests check actual answer
```

[7]:
```
"""Basic checks on N50"""
assert(type(q8_N50)==int)
assert(q8_N50>4600000 and q8_N50<4700000)
```

1.6 5. Evaluating assemblies with QUAST

Since we are using raw data from a genome that is actually already solved, we can align our contigs to that reference to evaluate our de novo assembly performance. Remember, if we were trying to solve the genome of a new or unknown organism, we wouldn't be able to do this, but we could try using a close relative.

The NCBI id of the E. coli strain used here is NC_000913.3. The reference genome fasta file and gene annotations (GFF file) are available here: https://www.ncbi.nlm.nih.gov/nuccore/NC_000913.3. Download the reference fasta to a file on your local computer.

Also download your final assembly files (`minia_assembly_18.contigs.fa` and `assembly.fasta`). You can download files to your local computer by navigating to your `lab2` folder in Jupyter lab, right clicking the files, and selecting "Download". Note you could rename these files to something like `illumina_assembly.fasta` and `pacbio_assembly.fasta` to make sure you remember what they are.

Now, go to QUAST: http://cab.cc.spbu.ru/quast/

First, make sure that you will get an email report. On the right hand side, type in your email and click 'get personal page'. Close the QUAST tab, then wait for the email to arrive. Click on the "Personal Page" link, and perform your analysis there in order to get the results emailed.

Use quast to align your assemblies to the actual reference sequence of the bacterial strain used to generate the original data.

- Use "Add Files" to upload both of your assemblies for comparison.
- Check the "scaffolds", "find genes" boxes, and the "Prokaryotic" button.
- Under genome indicate you want to choose your own genome, and upload the fasta file we got from NCBI.
- Type in a title for this analysis under caption, then click evaluate.

You can check your answers for N50 values above based on those computed in QUAST. Do they match?

You may also use the "Icarus" browser to explore the differences between assemblies.

Question 9 (3 pts) Which of the following is true? Set the variable `q9_answer` to your answer.

- A: The NGA50 of the Illumina assembly is higher.
- B: The Pacbio assembly has fewer contigs than the Illumina assembly.
- C: The Illumina assembly has fewer contigs than the Pacbio assembly.
- D: The Illumina assembly has more indels.

[8]:
```
q9_answer = "B" # Set to your answer

# your code here
```

[9]:
```
""" Basic check on q9 answer"""
assert(q9_answer in ["A","B","C","D"])
# Hidden tests check actual answer
```