Module

3

Genome Assembly

Goal: 1) Learn how to use the command line program **Velvet** to generate genome assemblies
2) Assemble the genome of the bacterium *Bacillus cereus*
3) Visualize the assembly layout

Input(s): R1 and R2 reads of *B. cereus*:
`Bcereus_S1_L001_R1_001.fastq.gz`
`Bcereus_S1_L001_R2_001.fastq.gz`

Output:	`Bcereus_velvet1`	directory of Velvet assembly results
	`Bcereus_velvet_optimal`	directory for optimized Velvet assembly results

Most genome assembly programs are run in a similar manner to one another, with the main differences usually being the types of sequence reads for which they are optimized. For this particular exercise, we will use the **Velvet** assembler because it is a good all-round genome assembler for Illumina sequence data.

Velvet 1.2.10

Velvet is a short read aligner and easily handles the assembly of large datasets comprising reads ranging from 50 to 300 bp in length.

Zerbino DR, Birney E. (2008) Velvet: algorithms for *de novo* short read assembly using de Bruijn graphs. Genome Res. 18:821–829.

https://github.com/dzerbino/velvet/tree/master

The **Velvet** package contains two main programs that perform separate functions. **Velveth** creates a table that lists all of the subsequences of a pre-chosen length, k (“k-mers” *sensu* monomer, dimer, etc.), that are found within the sequence reads along with the number of times each k-mer occurs. **Velvetg** then builds and traverses the de Bruijn graph using the k-mer table as its guide. Before we start running either program, however, we must first get the sequence reads into a suitable format.

3.1 Analyze the quality of the sequence reads

- □ First, make sure you are in the home directory of your virtual machine.
- □ Change into the *assembly* directory.
- □ List the directory contents. You will see that it contains two sequence files, *Bcereus_S1_L001_R1_001.fastq.gz* and *Bcereus_S1_L001_R2_001.fastq.gz*. The sequence filenames contain useful information about the sequences. "S1" means that this was the first barcode on the flow cell. "L001" indicates the data came from flow cell lane 1. "R1" means forward read (while "R2" means reverse). The "001" means this is dataset 1 from this lane of the flow cell. "fastq" indicates the sequence format, and "gz" tells us that the data were compressed with gzip.
- □ Before we start the assembly, let's examine the data quality using **FastQC**. Refer to Lab 2 for a refresher on using the program. If you have any difficulty launching **FastQC**, make sure you have followed the instructions for connecting to your VM with X11 forwarding.
- □ There are possible problems with the input data. What are these problems and how would you solve them?

1. Problem: ____________________; Solution: ____
2. Problem: ____________________; Solution: ____
3. Problem: ____________________; Solution: ____
4. Problem: ____________________; Solution: ____

3.2 OPTIONAL: Trim/filter the reads to remove poor quality sequences

Only perform the following trimming steps if you are super proficient with the command line.

- □ Use **Trimmomatic** to trim/filter the data to remove poor quality sequence. Refer to the instructions in Lab 2 for a reminder of required arguments and their values (modify the values as needed for this dataset).
- □ Record the **Trimmomatic** arguments that you used so that we can compare assemblies generated with trimmed/non-trimmed data:

____

____

3.3 Identify a suitable k-mer length

Because **Velvet** is a de Bruijn graph assembler, it breaks the sequencing reads down into subsequences known as k-mers. The choice of k-mer length is very important and will affect the quality of the assembly. A k-mer that is too long will produce an incomplete assembly, while one that is too short will yield a fairly complete assembly but will have a large number of very short contigs. To assist us in the choice of a suitable k-mer length, we will use an online tool called Velvet Advisor:

http://dna.med.monash.edu.au/~torsten/velvet_advisor/

- □ Visit the URL in the link. (**Important: Note that there is an underscore _ between the words velvet and advisor**).
- □ Fill in the relevant fields with information on the sequencing reads and estimated genome size (5.5 Mb). **When selecting the "paired-end reads" option, you should enter the number of reads in either the R1 or R2 datasets—not both.** Use a target k-mer coverage of 20. (Note that assemblies typically improve as k-mer coverage increases to 20 and then reach a plateau beyond this value.) Inspect the highlighted "Answer" field to find the recommended k-mer length. Note, this is only a guideline, and the best length is determined empirically by running **Velvet** iteratively, using a range of k-mer values around the one suggested by **Velvet Advisor**.
- □ What is the suggested k-mer length for the *B. cereus* genome assembly? ____________________

Before we move on, let's get a sense of how the choice of k-mer length is affected by the size of the genome under investigation:

- □ Fungal sequencing project: 30 million reads; 150 bp PE; genome size 45 Mbp

  Suggested k-mer length: ____________

- □ Human sequencing project: 600 million reads; 150 bp PE; genome size 3.1 Gbp

  Suggested k-mer length: ____________

3.4 Using screen

In this lab and others, we will be running programs that take a lot of time to run. While these programs are running, you may need to leave, or your SSH connection may drop unexpectedly. This is problematic because closing an SSH connection generally terminates your shell process and any processes you are running from that shell. We do not want the program to stop running whenever we disconnect; fortunately, there are ways to allow our programs to persist even when we are not connected.

Fortunately, various programs exist to solve this issue without a desktop environment. We will be using **screen**, a popular terminal multiplexer. As its name suggests, **screen** allows us to create virtual "screens" on which to view terminals. Like actual screens we might access with remote desktop software, those created by **screen** persist even if we close our connection.

We can create a **screen** session by simply running **screen** with no arguments. However, it is common to give names to new **screen** session, as it makes it easier to access them later. We will name the session for this class "velvet" by providing **screen** with a the -S argument.

□ Create a new **screen** session named “velvet”.

```
•   screen –S velvet
```

You should see that the screen immediately seems to clear. We have created a new **screen** session named “velvet” and are now attached to that screen. The new **screen** session starts a new **bash** process, which runs the *.bashrc* file upon startup and inherits any exported variables (such as PATH) from the shell you used to create the session.

□ Run a few commands, such as **ls**, in the new screen.

□ After running a few commands, detach from the current screen by typing Ctrl-a, releasing those keys and then hitting d. (Ctrl-a → d)

You will see that you are in the same shell you were using before you created the new screen. We can now easily reattach the screen we just created to resume working in that session.

□ Reattach the named screen session created earlier:

```
•   screen –r velvet
```

Unless you cleared the screen before detaching, you should see the output from the commands you ran in the **screen** earlier.

Although here we knew the name of the **screen** session we wanted to reattach, this may not always be the case (especially if we create a **screen** without specifying a name). Fortunately, we can list existing screen sessions by running:

```
       screen –ls
```

The output of this command looks something like this:

```
There is a screen on:
       5868.velvet (02/02/2019 12:42:22 AM) (Detached)
1 Socket in /var/run/screen/S–acta225
```

The first column starting on the second line lists the process ID followed by a dot and the name of the screen. You can reattach the screen using either.

We will mainly be using **screen** for its ability to continue running programs after SSH connections are closed. We can test this by disconnecting from the VM without terminating the screen session, reconnecting, and reattaching the screen.

□ Close the SSH connection (e.g., by closing the PuTTY/terminal emulator window or by detaching from the screen again and executing “**exit**”). Do not use the **exit** command within the screen. **exit** will terminate the **screen** session, but we only want to detach the screen from the current terminal.

□ Reconnect to the VM via SSH.

□ Reattach to the screen we created. (Note: If you closed the terminal window before detaching an active screen, that screen may still be attached. Add the -d argument to the command we used above to detach the screen before attaching it again)

□ Check that the output from the commands you ran earlier is still present in the screen.

Although we can terminate a **screen** session by running the **exit** command with the screen attached, it may sometimes be useful to terminate a screen without attaching it first. We can do this by running

```
    screen -X -S <screen_name> quit
```

It is also possible to terminate an attached **screen** session without typing the **exit** command in the shell. The command Ctrl-C will terminate the attached **screen** session

3.5 Run velveth using the suggested k-mer value

Before we begin, let's recap how **Velvet's** implementation of de Bruijn Graph assembly works: i) **velveth** breaks reads up into sets of overlapping k-mers and builds a temporary library that records the sequences and positions of k-mers within the reads; ii) it then identifies overlapping reads by comparing the k-mer profile for each read against the library to find the reads with the largest overlap; iii) these overlaps are represented as a graph that serves as "roadmaps" for guiding the de Bruijn process; iv) **velvetg** breaks the reads into k-mers again and implements a traditional Eulerian path traversal to form a PreGraph v) this preliminary graph is then threaded through the roadmaps, so as to take advantage of the additional information contained in the reads (i.e. links between k-mers that are lost when you generate a k-mer list); vi) finally, **velvetg** tries to resolve repeats and removes bubbles caused by sequencing errors.

We are now ready to run our genome assembly. First, we will use velveth to build the Roadmaps (a PreGraph) that describe connections between reads.

Usage:

```
velveth <output-directory> <kmer-length> <read-type> <read-format> <r1> <r2>
```

□ Run **velveth** on the *B. cereus* sequence reads using the suggested k-mer value obtained from **velvet advisor**:

```
•  velveth Bcereus_velvet1 <kmer-value> -shortPaired -fastq.gz
   -separate Bcereus_S1_L001_R1_001.fastq.gz
   Bcereus_S1_L001_R2_001.fastq.gz
```

□ Assembly should take about a minute and will produce a new directory (*Bcereus_velvet1*) that contains the following files:

Log	describes the compilation settings for the program
Sequences	the uncompressed sequences used for assembly
Roadmaps	a graph file summarizing overlaps between reads based on shared k-mers

You could look at the *Roadmaps* file, but you will find that the information contained therein is not very useful to us (it just helps **velvetg** to perform its tasks).

3.6 Run velvetg to perform the assembly

Velvetg performs the graph assembly, works to remove "bubbles" caused by sequencing errors, and resolves repeated sequences.

Usage:

```
velvetg [options] <velvet-output-directory>
```

We'll run **velvetg** using the default parameters to generate a "baseline" assembly.

☐ Run **velvetg** to generate our first assembly.

```
• velvetg Bcereus_velvet1
```

☐ When the run has completed, list the output directory. There should be seven output files:

Log describes the compilation settings for the program and summarizes the final assembly

stats.txt summarizes information about each contig

Sequences contains the input reads

PreGraph Initial graph created before any simplification or error correction

Graph Created after removal of tips, bubbles, and erroneous branches in the PreGraph

LastGraph The *final graph* after all simplification, repeat resolution, and scaffolding steps have been completed

contigs.fa fasta file of contigs longer than 2k (where k is the k-mer length used in **velveth**)

☐ Interrogate the end of the **velvetg** runtime message or the resulting *Log* file to determine the following metrics for the assembly:

Genome size (total): ______________________________

Number of scaffolds (nodes)*: ____________________________________

Longest scaffold (max): ____________________________________

N50 value: ____________________________________

*Note: Scaffolds are referred to as "nodes" in the *Log* file but they are actually scaffolds because we used paired-end reads to stitch adjoining "nodes" together.

3.7 Run velvet using a range of k-mer values

To obtain an optimal genome assembly, we could run **Velvet** multiple times using different k-mer lengths surrounding our suggested value. Fortunately, the nice folks at the Victorian Bioinformatics Consortium have written a script that automates this process. All we have to do is provide it a range of k-mer values to test and the step size between each value.

Usage:

```
velvetoptimiser [options] –f ‘<velveth input line>’
```

☐ **Make sure you are in the velvet screen**.

□ Run **velvetoptimiser** using the k-mer range 121 to 201. (Be patient. **velvetoptimiser** takes around 20 minutes to run to completion. Also note that there is no tab completion on the arguments to **velveth**, so be careful not to make typos!)

```
• velvetoptimiser -s 121 -e 201 -x 10 -d Bcereus_velvet_optimal
  -f '-shortPaired -fastq.gz -separate
  Bcereus_S1_L001_R1_001.fastq.gz
  Bcereus_S1_L001_R2_001.fastq.gz' -t 1
```

`-s` The starting (lower) k-mer value (default '19')

`-e` The end (higher) k-mer value (default '201')

`-x` The step in k-mer search, min 2, no odd numbers (default '2')

`-d` The name of the directory where final output files are saved

`-f` The section of the **velveth** command line that contains information about the input files and the types of sequences being assembled.

`-shortPaired` Type of reads in assembly

`-fastq.gz` Sequence read format

`-separate` The sequence reads are paired-end with the pairs being provided in separate files (Important: the R2 file must contain paired-ends for all the reads in the R1 file and vice versa, and corresponding pairs must be in the same order in the two files)

`-t` The number of parallel threads to run with. (default: number of processors)

□ When the run has completed, examine the *Bcereus_velvet_optimal* output directory. Now there should be eight output files: the seven that were produced in the first run, and an eighth that starts with a date and time and ends with "*Logfile.txt*"(***XX-XX-XXXX-XX-XX-XX_Logfile.txt***) —the actual name depends on the date and time of your run. This last file is the main log created by **velvetopimiser**, and it summarizes all the different assemblies and optimization steps that were performed. Metrics for the optimized assembly are reported at the bottom.

□ Find the information on the final optimized assembly from the end of the **velvetoptimiser** runtime message, or from the aforementioned log file and summarize the following metrics:

Genome size (total bases in contigs): ______________________________________
Number of scaffolds (contigs)*: ______________________________________
Longest scaffold: ________________________________
N50: ______________________________________

Note: Scaffolds are referred to as "contigs" in the **velvetoptimiser log file but they are actually scaffolds because we used paired-end reads to stitch adjoining "contigs" together.

What characterizes an optimal assembly? It depends on how you plan to utilize your genome. For example, if you wish to identify and characterize specific genomic regions, you might want to have maximal contig lengths (which means a high N50) to increase the chance of having the target sequence(s) on a single contig. On the other hand, if you plan to use the whole genome in subsequent analyses, then having a small number of total contigs might be desirable. (Note that the two metrics, though often correlated, don't always fully correspond.) In **velvetoptimiser**, the optimization function can be tuned to maximize different variables:

LNbp	The total number of nucleotides in large contigs
Lbp	The total number of base pairs in large contigs
Lcon	The number of large contigs
max	The length of the longest contig
n50	The n50 value
ncon	The total number of contigs
tbp	The total number of base pairs in contigs

Examples of optimization functions include:

'Lbp'	Just the total base pairs in contigs longer than 1kb
'n50*Lcon'	n50 times the number of long contigs
'n50*Lcon/tbp+log(Lbp)'	n50 times the number of long contigs divided by the total bases in all contigs, plus the log of the number of bases

3.8 Further optimize the assembly (optional)

- □ Use the optimized parameters established in the above **velvetoptimiser** run as a guide to try and further improve the *Bacillus cereus* genome assembly. Specifically, select new start and end k-mer sizes that bracket the optimal value obtained from the first run. **Remember to use odd starting k values (to avoid palindromes) and an even step size to maintain k "oddness" throughout the run. (2 is a good step size.)**
- □ Interrogate the end of the runtime message, or the new *XX-XX-XXXX-XX-XX-XX_Logfile.txt* file, to see how the final metrics compare with the first round of optimization:

Genome size (total bases in contigs): round 2:____________________ round 1:____________________

Number of scaffolds (contigs): round 2:____________________ round 1:____________________

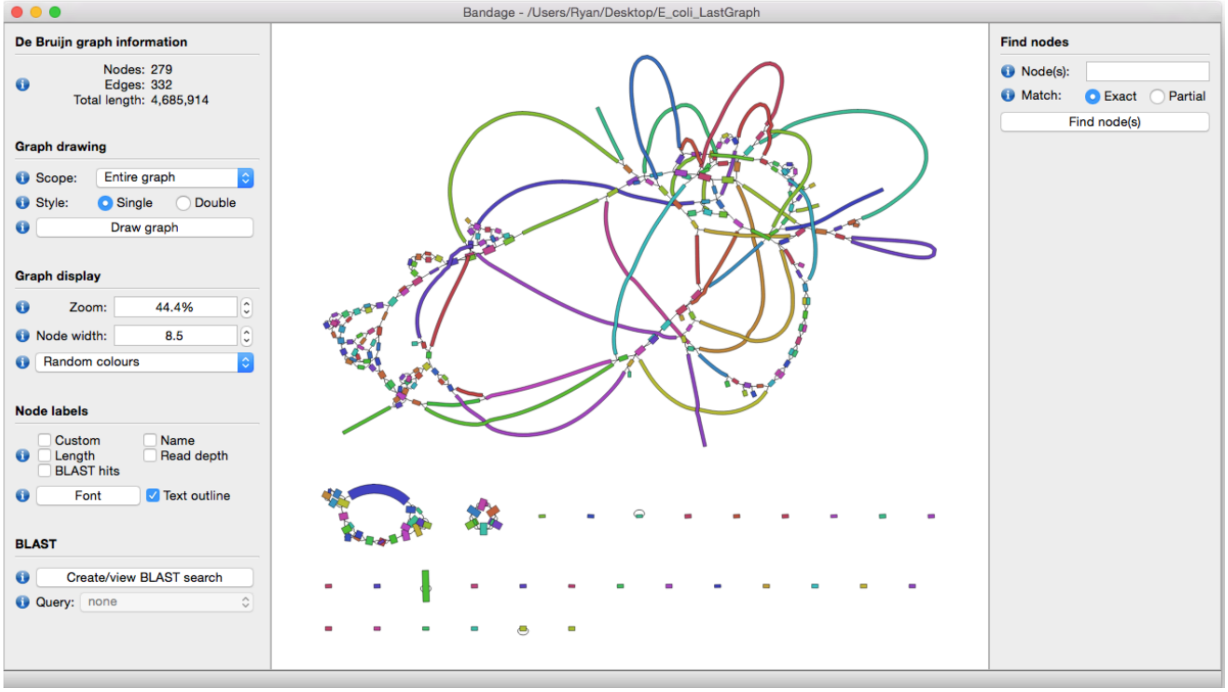
Longest scaffold: round 2:____________________ round 1:____________________

N50: round 2:____________________ round 1:____________________

Optional exercise: Visualizing assemblies using Bandage

Bandage is a graphical tool that allows one to interrogate an assembly graph to examine contiguity and resolve ambiguities. It also incorporates a **blast** search engine for homing in on sequences of interest. One can view entire assemblies or select individual nodes and their neighbors.

https://rrwick.github.io/Bandage/



Bandage - /Users/Ryan/Desktop/E_coli_LastGraph

De Bruijn graph information

Nodes: 279
Edges: 332
Total length: 4,685,914

Graph drawing

Scope: Entire graph
Style: ● Single ○ Double

Graph display

Zoom: 44.4%
Node width: 8.5
Random colours

Node labels

□ Custom □ Name
□ Length □ Read depth
□ BLAST hits
☑ Text outline

BLAST

Query: none

Find nodes

Node(s):
Match: ● Exact ○ Partial

- ☐ Visit the above link and download and install the Bandage executable on your local machine (Mac or PC)
- ☐ Use **scp** to download a *LastGraph* or *Graph2* file generated by one of your **velvet** assemblies to your local machine.
- ☐ Open Bandage and under the **File** menus, select "**Load graph**".
- ☐ In the file selection dialog, navigate to the downloaded *Graph2* file.
- ☐ Select the *Graph2* file and click "**Open**".
- ☐ Click the "**Draw graph**" button on the left-hand panel.

Bandage will now display the entire assembly graph, consisting of one major cluster of interconnected nodes (i.e., contigs) in a highly reticulated network, a number of minimally branched graphs, and mostly single linear nodes. Overall, this assembly looks quite good because there aren't too many "snarled" networks, and the number of nodes is fairly small. **Note that this graph has many more nodes than the final optimized assembly. This is because the VelvetOptimiser wrapper that invokes the velvetg processes appears to prevent the last run from writing the LastGraph file.**

- □ Click on any line (single, or within a branched network). Observe that the identity of the selected node is displayed in the right-hand panel, along with its length and average depth of coverage.

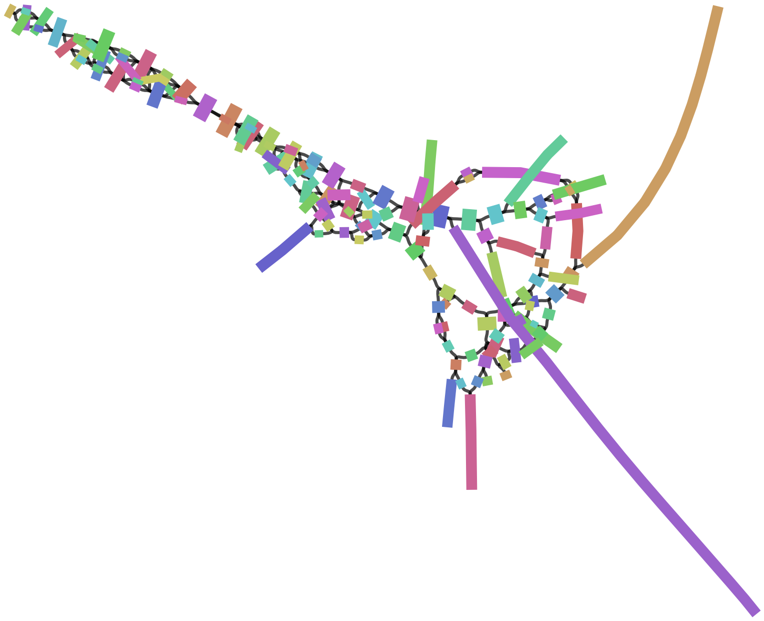


Figure 1. Example of a complex layout in a part of the *B. cereus* assembly

Now let's find out how to zoom in on a specific node in the layout. First let's find a suitable node to work on.

- □ Click on any node in a region with a complex layout. An example of such a region is shown in Figure 1.
- □ Under **Find nodes** in the right-hand panel, type the number of the node about which you'd like information, then click "**Find node(s)**"

The viewer will zoom in on the node(s) you selected within the context of the entire layout graph. As you can see, the layout is quite complex and hard to interpret. We will need to zoom in more closely on regions for more clarity.

If we wish to view the environs of our selected node with more clarity, we can change the **Scope** under "**Graph drawing**" (left-hand panel).

- □ Select "**Around node**(s)" using the **Scope** pull-down menu. Type your node number into the "**Node(s)**" box that appears, increase **Distance** to 4, and then click "**Draw Graph**". This will show your favorite node at the center of the graph and linked nodes extending up to 4 nodes away.

This ability to "see" complex layouts is a great benefit when trying to resolve genomic regions that are poorly assembled. Plus, it can tell you if your sequence of interest is in a poorly assembled region. **Bandage can even perform BLAST searches to find out in which node(s) any given query sequence resides.**