Assembly using velvet

Where is what

For this practical, in addition to the files we trimmed yesterday, we will be using the following two other files in `/data/assembly/illumina`

`jump_reads_42866.A_RC.fastq`	2x93	3kb MP read1	362200 reads, 34 Mbp
`jump_reads_42866.B_RC.fastq`	2x93	3kb MP read2	362200 reads, 34 Mbp

These files are from a 3 kb mate pair ('jumping') library.

***De novo* assembly of Illumina reads using velvet**

Learning points

- understand k-mers and how to estimate genome size from short-read data
- understand how to run Velvet
- understand the concept of k-mer coverage
- understand the effect of k-mer settings on assemblies
- understand how to use paired-end information in Velvet
- understand the significance of setting expected coverage values

Assembling short-reads with Velvet

We will now use Velvet to assemble Illumina reads on their own, a program that uses the *de Bruijn graph* approach.

We will assemble the *E. coli K12* strain MG1655 which was sequenced on an Illumina MiSeq. The sequencing centre read 150 bases from each direction. We know from the sequencing centre that the fragment size they chose was approximately 450bp.

You should have two files from yesterday's filtering practical in your `qc_filter folder`, named

```
MiSeq_50x_R1_trimmed.fastq
MiSeq_50x_R2_trimmed.fastq
```

When we filtered the data, some of the reads were removed. This means that for some pairs, either read 1 or read 2 was lost. Most programs, including velvet, require the paired reads to match up between the fastq files. We will therefore first create new fastq files with properly matched up pairs only. “Orphan” reads which have no pair will be sent to a separate file which we will not use in this practical.

We will use the `pair_up_reads script`:

```
cd /home/yourusername/qc_filter
```

OR (remember, the ‘~’ symbol stands for your home

```
cd ~/qc_filter

pair_up_reads.py MiSeq_50x_R1_trimmed.fastq \
     MiSeq_50x_R2_trimmed.fastq \
     MiSeq_50x_R1_good.fastq \
     MiSeq_50x_R2_good.fastq \
     MiSeq_50x_orphans.fastq
```

Examine the resulting files.

Are there the same number of sequences in `MiSeq_50x_R1_trimmed.fastq` as `MiSeq_50x_R2_trimmed.fastq`? Why? (Hint: `wc -l filename` will tell you the number of lines in a file)	
Are there the same number of sequences in `MiSeq_50x_R1_good.fastq` and `MiSeq_50x_R2_good.fastq`? Why?	
How many orphan sequences were produced?	

Building the Velvet Index File

Velvet requires an index file to be built before the assembly takes place. We must choose a *k*-mer value. Longer *k*-mers result in a more stringent assembly, at the expense of coverage. There is no definitive value of *k* for any given project. However, there are several absolute rules - *k* must be less than the read length and it should be an odd number.

Firstly we are going to run Velvet in single-end mode, *ignoring the pairing information*. Later on we will incorporate this information.

Create a new folder for the output

```
cd /home/yourusername
```

or simply type

```
cd
```

Create the assembly folder if it doesn't already exist:

```
mkdir assembly

cd assembly
mkdir velvet
cd velvet
```

Find a value of *k* (between 21 and 99) to start with, and record your choice in this google spreadsheet: **`bit.ly/INFBIO1`**. Run `velveth` to build the hash index (see below).

Program	Options	Explanation
velveth		Build the Velvet index file
	foldername	use this name for the results folder
	value_of_k	use k-mers of this size
	-short	short reads (as opposed to long, Sanger-like reads)
	-separate	read1 and read2 are in separate files
	-fastq	read type is fastq

```
velveth asm_name value_of_k \
     -short -separate -fastq \
     ~/qc_filter/MiSeq_50x_R1_good.fastq \
     ~/qc_filter/MiSeq_50x_R2_good.fastq
```

After it has finished, look in **`asm_name`**. You should see the following files:

```
Log
Roadmaps
Sequences
```

`Log` is a useful file, this is a useful reminder of what commands you typed to get this assembly result, useful for reproducing results later on. Sequences contains the sequences we put in, and `Roadmaps` contains the index you just created.

Now we will run the assembly with default parameters:

```
velvetg asm_name
```

Velvet will end with a text like this:

```
Final graph has ... nodes and n50 of ..., max ..., total ...,
using .../... reads
```

The number of nodes represents the number of nodes in the graph, which (more or less) is the number of contigs.
Look again at asm_name, you should see the following extra files;

```
contigs.fa
Graph
LastGraph
PreGraph
stats.txt
```

The important files are:

`contigs.fa` - the assembly itself
`Graph` - a textual representation of the contig graph
`stats.txt` - a file containing statistics on each contig

What *k*-mer did you use?	
What is the N50 of your assembly?	
What is the N50 of an assembly with 7 contigs of sizes: 20, 9, 9, 6, 3, 2 and 1 long?	
What is the size of the largest contig?	
How many contigs are there in the contigs.fa file? use `grep -c NODE contigs.fa`	

Log your results in this google spreadsheet: **`bit.ly/INFBIO1`**
We will discuss the results together.

Advanced tip: You can also use VelvetOptimiser to automate this process of selecting appropriate *k*-mer values. VelvetOptimizer is included with the Velvet installation.
Now run velveth and velvetg for the kmer size determined by the whole class. Use this kmer from now on!

Estimating and setting exp_cov

Much better assemblies are produced if Velvet understands the expected coverage for unique regions of your genome. This allows it to try and resolve repeats. The command velvet-estimate-exp-cov.pl is supplied with Velvet and will plot a histogram of k-mer frequency for each node in the graph, listing k-mer frequency, and the number of count of nodes with that frequency

```
velvet-estimate-exp_cov.pl asm_name/stats.txt
```

The peak value in this histogram can be used as a guide to the best value for *exp_cov*.

What k-mer frequency is the most frequent? Why?	
What do you think is the approximate k-mer coverage for your assembly?	
Convert this value from k-mer coverage into *genome* coverage. The formula is: $\frac{Ck * L}{(L - k + 1)} = C$ Where Ck = k-mer coverage, L = read length, k = k-mer size for your assembly	

Now run velvet again, supplying the value for exp_cov (k-mer coverage, *not* genome coverage) corresponding to your answer:

```
velvetg asm_name -exp_cov peak_k_mer_coverage
```

What improvements do you see in the assembly by setting a value for exp_cov?	

Setting cov_cutoff

You can also clean up the graph by removing low-frequency nodes from the *de Bruijn* graph using the cov_cutoff parameter. This will often result in better assemblies, but setting the cut-off too high will also result in losing bases. Using the histogram from previously, estimate a good value for cov_cutoff.

```
velvetg asm_name -exp_cov your_value \
    -cov_cutoff your_value
```

Try some different values for cov_cutoff, keeping exp_cov the same and record your assembly results:

Exp_cov	Cov_cutoff	N50	Total assembly size

Now ask Velvet to predict the values for you:

```
velvetg asm_name -exp_cov auto -cov_cutoff auto
```

What values of exp_cov and cov_cutoff did Velvet chose? Check the output to the screen	
Is this assembly better than your best one?	

Summarize your assemblies so far by picking a few and recording the metrics below:

Exp_cov	Cov_cutoff	N50	Total assembly size
auto	auto		

Incorporating paired-end information

Paired end information contributes additional information to the assembly, allowing contigs to be scaffolded. Velvet needs to be told to use paired-end information as follows:

Now, re-index your reads telling Velvet to use paired-end information (using `-shortPaired` instead of `-short` for `velveth`) and re-run Velvet using the best value of *k, exp_cov and cov_cutoff* from the previous step.

********** IMPORTANT Pick a new name for your assembly ***********

```
velveth asm_name2 value_of_k \
     -shortPaired -fastq -separate \
     ~/qc_filter/MiSeq_50x_R1_good.fastq \
     ~/qc_filter/MiSeq_50x_R2_good.fastq

velvetg asm_name2 -exp_cov value_of_exp_cov \
     -cov_cutoff value_of_cov_cutoff
```

How does doing this affect the assembly?

The contigs are actually scaffolds. Use the assemblathon_stats script (see the file 'Assembly: general tools') to generate metrics for this, and all following

assemblies. Use the the total length reported by velvet as estimate for the genome size.

Looking for repeats

Have a look for contigs which are long and have a much higher coverage than the average for your genome:

E.g.

```
awk '($2>=1000 && $6>=60)' stats.txt
```

Will find contigs larger than 1kb with k-mer coverage greater than 60.

awk is an amazing program for tabular data. In this case, we ask it to check that column 2 ($2, the length) is at least 1000 and column 6 ($6, coverage) at least 60. if this is the case, awk will print the entire line. See http://bit.ly/QjbWr7 for more information on awk

Find the contig with the highest coverage. perform a BLASTX search using NCBI. What is it? Is this surprising?

Generate assembly output

When you have produced the best assembly you can, you can re-run Velvet and produce output files for viewing:

```
velvetg asm_name options_you_used \
    -read_trkg yes -amos_file yes
```

This will produce an AMOS-compatible .afg file which can be read in the viewer program Tablet (see the file 'Assembly: general tools'). Start Tablet and load the afg file. Look at some of the alignments. What kind of variations (where reads diverge from the consensus) do you see most often?

The effect of mate-pair libraries

Long-range "mate-pair" libraries can also dramatically improve an assembly by scaffolding contigs. Typical sizes for Illumina are 2kb and 6kb, although any size is theoretically possible. You can supply a second library to Velvet. However, it is important that files are reverse-complemented first as Velvet expects a specific orientation. We have supplied a 3kb mate-pair library in the correct orientation.

********** IMPORTANT Pick a new name for your assembly ***********

We will use `-shortPaired` for the paired end library reads, and `-shortPaired2` for the mate pairs:

```
velveth asm_name3 value_of_k \
 -shortPaired -separate -fastq \
 ~/qc_filter/MiSeq_50x_R1_good.fastq \
 ~/qc_filter/MiSeq_50x_R2_good.fastq \
 -shortPaired2 -separate -fastq \
 /data/assembly/illumina/jump_reads_42866.A_RC.fastq \
 /data/assembly/illumina/jump_reads_42866.B_RC.fastq
```

We use auto values for velvetg because the addition of new reads will change the genome coverage:

```
velvetg asm_name3 -cov_cutoff auto -exp_cov auto
```

What is the N50 of this assembly?	
How many scaffolds?	
How many bases are in gaps?	
What did velvet estimate for the insert length of the paired-end reads, and for the standard deviation? Use the last mention of this in the velvet output.	
And for the mate-pair library?	

Make a copy of the contigs file and call it '`MP.fa`'

Mate-pair data can contain significant paired-end contamination which generates misassemblies. To account for this, let Velvet know about the problem with the following flag:

```
velvetg asm_name3 -cov_cutoff auto \
     -exp_cov auto -shortMatePaired2 yes
```

What is the N50 of this assembly?	
How many scaffolds?	
How many bases are in gaps?	

Make a copy of the contigs file and call it '`MP_nocont.fa`'

Evaluation of the assemblies

We will evaluate today's assemblies against the reference using the program 'Mauve'.

Make a new folder called

```
~/assembly/mauve
```

Make a copy of the all the contig/scaffolds files into this folder, change the name of the file if needed but keep the names as short (yet informative) as possible. For those with more unix experience, another way is to generate symlinks in this folder (with the target a short name).

Now, use the descriptions in the document ***Assembly evaluation using Mauve*** to investigate your assemblies:

- reorder contigs from the first assembly against the reference, visually inspect the results
- run mauveAssemblyMetrics.sh on all assemblies