De Novo Genome Assembly using Short Reads

Introduction

Genome assembly is a multi step process involving many complexities and problems that must be dealt with to produce high-quality genome sequences. Some of these problems are the result of the underlying genome that is being assembled and some of these problems are intrinsic to the data we use to assemble the underlying genome. Throughout this demo, you will explore the effects of some of these problems and employ commonly used strategies and techniques for handling such problems. The three problems you will tackle are 1) sequencing read errors, 2) non-uniform coverage, and 3) repetitive genome elements.

Assessing genome assemblies

Because assembly algorithms are imperfect, we rarely uncover the target genome in the first attempt at assembling it. Often times we rely on a trial and error process to get a "best" assembly. Furthermore, to get the "best" assembly, we must use some criteria for determining what that may be. Methods have been developed for assessing the quality of a draft genome assembly, ranging from the application of statistical models for calculating the likelihood of an assembly, to whole genome alignment against a closely related organism for which a high quality draft or finished genome exists. For this laboratory, you will use simple descriptive statistics for all three exercises, and for the third you will also use whole genome alignment against a reference sequence.

When a reference is not available, we must use simple descriptive statistics to assess the quality of a draft genome assembly. Although imperfect, these statistics are extremely useful, straightforward, and easy to calculate. These statistics include, but are not limited to: total number of contigs, contig L50, total number of bases in the assembly, the largest contig in the assembly, and the number of gaps.

> **Contig L50**: Given a set of varying length contigs, the L50 length is defined as the length of L for which 50% of all bases in the assembly are in a sequence of length $l < L$.

Target Genome

The genome you will be assembling is that of the human immunodeficiency virus I (HIV I). An overview of this genome can be found at http://www.ncbi.nlm.nih.gov/genome/10319.

HIV is a retrovirus, meaning that its genomic material is stored in RNA rather than DNA. The virus operates by infecting the host cell and is replicated using reverse transcriptase to create DNA from the RNA genome. This strand of DNA is then integrated into the host genome and replicated when the host cell replicates.

The HIV genome consists of a single single-stranded linear RNA chromosome. The single chromosome contains 9 protein-coding genes and 1 signaling peptide. At each end of the linear chromosome lie 96bp repetitive elements that initiate transcription of the chromosome.

Software

For this laboratory, you will use multiple programs to carry out analyses. For assembling genomes, you will use the *de Bruijn* graph assembler, Velvet [3]. Velvet was one of the first *de Bruijn* graph assemblers developed when high-throughput short-read genomic data first became available.

Velvet is run using two commands. The first command is `velveth`. This command will count all *k*-mers in a data set and build a *de Bruijn* graph. The second command is `velvetg`, which traverses the *de Bruijn* graph created using `velveth` and creates contigs. You can obtain the usage for these two commands by simply executing them on the command line with no arguments.

In addition to Velvet, you will be using the program BLAST [1] to compare a DNA sequence against a genome assembly done in the third exercise. BLAST, a **B**asic **L**ocal **A**lignment **S**earch **T**ool, is one of the most widely used bioinformatics tools. It can be used for comparing DNA sequences and amino acid sequences. BLAST is invoked with the command `blastall`. Like Velvet, you can see the usage for it by simply executing it on the command line with no arguments. Because BLAST can be run on a variety of different ways, there are multiple programs within the single command. You will be using the `blastn` program for this laboratory. This is invoked with the following command:

```
$ blastall -p blastn.
```

The output mode you will be using BLAST will print twelve columns. The columns are as follows:

1. Query
2. Subject
3. Percent identity
4. Alignment length
5. Number of mismatches
6. Gap openings
7. Query start
8. Query end
9. Subject start
10. Subject end
11. E-value
12. Bit score

Columns 3, 4, 5, 6, 11, and 12 provide information about the quality and significance of the alignments (referred to as 'hits' or 'matches'). Columns 1 and 2 provide information about which sequences are matching, and columns 7, 8, 9, and 10 provide the coordinates within these sequences that match. Columns 9 and 10 will be of most interest to you in this lab.

Summary statistics on assemblies can be obtained using the command `sumfasta`. This program simply works by providing the assemblies as arguments to the command. For example, the command:

```
$ sumfasta assembly1.fasta assembly2.fasta
```

will provide summary statistics for assemblies stored in the files `assembly1.fasta` and `assembly2.fasta`

For the third exercise, you will evaluate the effect of repetitive elements on assemblies. To do so, you will align assemblies against a reference genome using the aligner `progressiveMauve` [2] and look over the alignments using the Mauve alignment viewer.

Platform

Exercises for this demo will be carried out on a virtual machine running Ubuntu. Ubuntu is an open source distribution of the Linux operating system.

To run the virtual machine, you will use the program VirtualBox, which can be downloaded at the following website: https://www.virtualbox.org/wiki/Downloads

The virtual machine image has been prebuilt with the necessary software. The virtual machine image can be downloaded from the following link:
https://drive.google.com/file/d/0BxoXt7fkRSWvVGo3eUdQRXNyb0U

Once you have downloaded VirtualBox and the virtual machine image, you will need to set up the virtual machine within VirtualBox. To do so, click on "New" at the top of the VirtualBox Manager, and follow the walk through instructions to set up the virtual machine. When you get to the VM Name and OS Type step, set the Operating System to be "Linux," and the Version to be "Ubuntu" (do not select "Ubuntu 64 bit"). Name the image "engr230-lec4". The next step will be for Memory. I suggest setting this to 1024 MB. Next you select the Virtual Hard Disk. Make sure the the box for "Start-up Disk" is checked, and select the option "Use existing hard disk". The dropdown box should be empty. To add a disk to this, click on the folder icon (with a green arrow) to the left of the dropdown box, and locate the virtual machine image you just downloaded and extracted.

Now that you have your virtual machine setup, you can boot it up. From the VirtualBox Manager, double-click on the machine "engr230-lec4" to boot the machine. Once it has completed booting, you will need to login. The username is "ubuntu" and the password is "reverse".

When you are signed in, you will need to open Terminal, the command-line interface. This is where all your work will be carried out. You can open Terminal be clicking on the icon  on the left panel. It is highly recommended that you maximize the screen once you have it opened. This will make it much easier to read the output of programs you will run.

Although it is not necessary, installing the Guest Additions is recommended. This will allow you to display the virtual machine full screen. To do so, click the "Devices" tab at the top of the virtual machine screen, and select the option "Install Guest Additions". The installer will launch automatically and prompt you. Select "Run" and when prompted for a password, enter "reverse" and select "Authenticate". A Terminal screen will pop and print the progress of the installer to the screen. When the installation is complete, hit "Return" to close the window. For the Guest Additions to take effect, you will need to restart the virtual machine. To do so, click on the power-button icon on the upper right corner of the screen and select "Shut Down", and select "Restart". Once the machine restarts, log back in and you will now be able to enter full screen mode (with Ctrl-F or ⌘-F), and increase the window size to your preference.

Exercises

All of the exercises you carry out will be done in the directory `GenomeAssembly`. You can change into this directory using the `cd` command. Once you are in that directory, change into the `exercises` directory. Within this directory there are three subdirectories, E1, E2, and E3; one for each exercise. Change into the `E1` directory to begin with Exercise 1.

```
$ cd GenomeAssembly
$ cd exercises
$ cd E1
```

1. Handling errors in data sets.

   Sequencing platforms are imperfect, and often generate erroneous sequencing reads. These sequencing errors pose problems with assembling a genome. However, there are simple methods for handling these errors.

   As discussed in the lecture, a *de Bruijn* graph is created by counting *k*-mers in the data set. *K*-mers that represent the true genome sequence will have counts that are

proportional to the coverage generated by the sequencing run. Due to sequencing error, erroneous *k*-mers can be present in the data set in low frequency. These low-frequency *k*-mers create "bubbles" in the *de Bruijn* graph and cause problems when extracting contigs. By setting a minimum frequency, erroneous *k*-mers can be removed to obtain more complete assemblies.

In this exercise, you will experiment with setting a minimum frequency to obtain an optimal assembly.

1.1. Naïve Assembly

First you will do an assembly with no minimum *k*-mer frequency. To do so, run the `velveth`, using *k*=27. You should output the results to the directory `velvet.k27.mink0`.

```
$ velveth velvet.k27.mink0 27 -fastq -short reads_shuf.fastq
```

This will create the *de Bruijn* graph. Next create contigs using `velvetg`.

```
$ velvetg velvet.k27.mink0 -cov_cutoff 0 -exp_cov 80
```

This will create a file, `contigs.fa` in the `velvet.k27.mink0` directory. Now run `sumfasta` to get stats on the assembly.

```
$ sumfasta velvet.k27.mink0/contigs.fa
```

Record some summary statistics from `sumfasta`'s output:

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

1.2. Filtered assemblies

Now try some assemblies with a minimum *k*-mer frequency. You will try a range of values to experiment with the effects of setting the minimum frequency too low and too high.

First try setting the minimum frequency to 3 and record the summary statistics

```
$ velveth velvet.k27.mink3 27 -fastq -short reads_shuf.fastq
$ velvetg velvet.k27.mink3 -cov_cutoff 3
$ sumfasta velvet.k27.mink3/contigs.fa
```

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

Now try setting the minimum frequency to 10 and record summary statistics

```
$ velveth velvet.k27.mink10 27 -fastq -short reads_shuf.fastq
$ velvetg velvet.k27.mink10 -cov_cutoff 10 -exp_cov 80
$ sumfasta velvet.k27.mink10/contigs.fa
```

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

Finally, try setting the minimum frequency very high

```
$ velveth velvet.k27.mink40 27 -fastq -short reads_shuf.fastq
$ velvetg velvet.k27.mink40 -cov_cutoff 40 -exp_cov 80
$ sumfasta velvet.k27.mink40/contigs.fa
```

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

What happens when you set a minimum *k*-mer frequency?

_____

_____

What do you think the optimal minimum *k*-mer frequency is and why?

_____

_____

2. Leveraging paired-end reads and compensating for non-uniform coverage

   Due to the stochastic nature of DNA sequencing, coverage is not uniform across the target genome. Furthermore, sequencing coverage varies drastically across the genome, leaving some areas very poorly covered. In the regions where there is very little to no coverage, the assembly breaks, resulting in the end of a contig. Although the sequence in a low-coverage region cannot be recovered, if the region is short enough, the surrounding regions can connected using read pairing information. This is referred to as "scaffolding". If two reads are sequenced from the same molecule (one from each end), and one of these reads maps to the end of one contig, and the other read maps to the end of separate contig, we can use the fact that the reads came from the same molecule to connect the two contigs. At the base level, this would look something like this:

Contig 1 **Contig 2**

...ATGCTANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNGCTAGCTA...

This sequence of N's is referred as a gap, and the N characters are referred to as unknown characters.

In this exercise, you will experiment with utilizing paired-end information to improve an assembly. Before beginning the exercise, change into Exercise 2 directory (`E2`):

```
$ cd ../E2
```

2.1. Naïve Assembly

First run an assembly that does not use any of the read pairing information. If you have not been doing so, record "`Num Nucs`", "`Num Unks`", and "`Num Gaps`"

```
$ velveth velvet.k27 27 -fastq -short reads_shuf.fastq
$ velvetg velvet.k27 -cov_cutoff 3 -scaffolding no -exp_cov 80
  -min_contig_lgth 100
$ sumfasta velvet.k27/contigs.fa
```

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

2.2. Paired-end assembly

Now run an assembly that utilizes read pairing information and record summary statistics.

```
$ velveth velvet.k27.Paired 27 -fastq -shortPaired
  reads_shuf.fastq
$ velvetg velvet.k27.Paired -cov_cutoff 3 -scaffolding yes
  -exp_cov 80 -ins_length 450 -ins_length_sd 50
  -min_contig_lgth 100
$ sumfasta velvet.k27.Paired/contigs.fa
```

Contigs:	N50:	AvgContigLen:
MaxContigLen:	MinContigLen:	TotalBases:
NumNucs:	NumUnks:	NumGaps:

What happens to the assembly when you utilize paired read information? (Hint: look at the summary statistics)

_____

_____

3. Repetitive sequence in the target genome

For many reasons, genome sequences contain repetitive sequence. It is repetitive sequence that poses the largest problem for genome assembly. If reads are not long enough to provide enough context for the assembler, or if the repetitive elements are too similar, the multiple copies will collapsed and assembled into a single sequence. In some cases, the collapsed sequence will be assembled into the middle of contig. Often times the sequence upstream of the collapsed sequence represents one context in which the repetitive element truly exists, and the sequence downstream represents another context.

Although repetitive elements can cause problems, we can use longer sequences in the assembly to help resolve these repeats. In some cases, the repeats are so long that no type of short read data can resolve the repeats, and longer reads (e.g. Sanger) need to be used. In cases where the repeat is short enough, using longer *k*-mers to exploit the full length of a read can help to resolve repeats. In this exercise

In this exercise, you will experiment with using a longer *k*-mer to resolve repetitive sequence. Before beginning the exercise, change into Exercise 3 directory (`E3`):

```
$ cd ../E3
```

3.1. Naïve assembly

First run an assembly with the standard *k*-mer size we have been using.

```
$ velveth velvet.k27 27 -fastq -shortPaired reads_shuf.fastq
$ velvetg velvet.k27 -cov_cutoff 3 -scaffolding yes
  -ins_length 450 -ins_length_sd 50 -exp_cov 80
```

3.2. Longer *k*-mer assembly

Now use a longer *k*-mer to assemble the genome.

```
$ velveth velvet.k103 103 -fastq -shortPaired reads_shuf.fastq
$ velvetg velvet.k103 -scaffolding yes -ins_length 450
  -ins_length_sd 50 -exp_cov 80
```

3.3. Evaluate the effect of the longer choice of *k*.

Using BLAST, search for the repetitive element in the assembly. There will be one line of output for each BLAST hit to the genome assembly.

```
$ formatdb -p F -i velvet.k27/contigs.fa
$ blastall -p blastn -m 8 -d velvet.k27/contigs.fa
  -i repeat_element.fasta > velvet.k27/blast.repeat_element.out
$ formatdb -p F -i velvet.k103/contigs.fa
$ blastall -p blastn -m 8 -d velvet.k103/contigs.fa
  -i repeat_element.fasta > velvet.k103/blast.repeat_element.out
```

Executing the above commands will store the output from BLAST in the files following the right carrot (>). You can display their contents with the command `cat`:

```
$ cat velvet.k27/blast.repeat_element.out
$ cat velvet.k103/blast.repeat_element.out
```

Record the number of hits and the locations for each BLAST hit for the two assemblies.

_____

_____

_____

3.4. Visualize the effect of repetitive elements on the assembly.
Using the whole genome aligner, progressiveMauve, align both draft assemblies to the true reference, and visualize the alignments with the Mauve genome alignment viewer.

To align the two genomes, use the following commands:

```
$ progressiveMauve --output=align.velvet.k103 HIV.gbk
  velvet.k103/contigs.fa
$ progressiveMauve --output=align.velvet.k27 HIV.gbk
  velvet.k27/contigs.fa
```

Now open up the alignments with the viewer:

```
$ mauve align.velvet.k27 &
$ mauve align.velvet.k103 &
```

In the alignment viewer, there will be two tracks: one for each genome. The top track represents the true genome, and the bottom track represents the draft assembly. The colored blocks in these tracks, referred to as locally collinear blocks (LCBs), correspond to different segments of similar sequence between the two genomes. Clicking on these blocks will align the similar sequence within the alignment viewer. You can navigate through the alignments with the left and right arrow buttons, and zoom in and out with the magnifying glass buttons at the top of the window. You can also use Ctrl-left arrow, Ctrl-right arrow, Ctrl-up arrow, and Ctrl-down arrow, respectively, for the same operations. To go back to the original view, click on the Reset display icon at the top (house icon).

Within the top track, there are sub-tracks below the main track. The blocks in these sub-tracks represent annotations of genomic elements (aka 'features'). Clicking these blocks will prompt you to view more information about the element you have clicked on. Notice there are two different colors of blocks in these sub-tracks: red and white.

Can you identify the elements discussed in the background information in these tracks? More specifically, what do you the red blocks represent?

_____

_____

Notice how alignment `align.velvet.k27` has two LCBs (a red and a green block) and alignment `align.velvet.k103` has a single block. In alignment `align.velvet.k27`, click to the left of the junction of the green and red LCBs in the draft assembly. The viewer should move the corresponding true genome sequence into view. What feature is present in the true genome sequence here?

_____

Now click to the right of the junction of the green and red LCBs in the draft assembly. What feature is present in the true genome sequence here?

_____

Knowing the identity of these elements, what do you think happened with the assembly done with *k*=27?

_____

_____

_____

_____

Why does the assembly done with *k*=103 only have a single LCB?

_____

_____

Can you reconcile what you see here in the alignment viewer with the results of the BLAST analysis?

_____

_____

_____

_____

References

1. Altschul S, Gish W, Miller W, Myers E, Lipman D (1990). Basic local alignment search tool. *Journal of Molecular Biology* 215(3): 403–410
2. Darling AE, Mau B, Perna NT (2010). progressiveMauve: Multiple Genome Alignment with Gene Gain, Loss and Rearrangement. *PLoS ONE* 5(6): e11147
3. Zerbino DR, Birney E (2008). Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research* 18(5): 821–829