A Primer to *de novo* Genome Assembly

Adam Karami

Introduction

Perhaps one of the most commonly performed activities in Bioinformatics - aside from overloading NCBI's servers with BLASTing your sequence of choice - is dealing with data that comes out of NGS. With an overwhelming number of different -Seqs today, you are sure to find yourself wrangling with these data sets in the near future if you stick to Bioinformatics. It's likely beneficial for you to introduce yourself to one of the first steps you have to take in the analytics involved with biological data from NGS: Assembling the short reads into discernible scaffolds. This operation will be done mainly on a Linux or Mac platform, and what you need is a nifty tool called Velvet.

Velvet is a short read assembler that, well, assembles scaffolds from short reads. Velvet deals in particular with *de novo* assembly, essentially meaning that the reads are assembled together without a guiding model reference. The capability to assemble reads without a reference is important because you won't always find yourself with a reliable reference genome. When you expect that your organism of choice has large-scale variations that differ greatly from the reference model, or when you just plain don't know what the hell it is that you're sequencing, *de novo* is there for you. You can obtain Velvet through the link https://www.ebi.ac.uk/~zerbino/velvet/ and following its installation manual on Linux or Mac.

What does Velvet do?

Velvet implements the de Bruijn graph algorithm to assemble the short reads into scaffolds. The de Bruijn graph forms assemblies based on the overlaps between k-mers, which are subsequences of biological information (nucleotide or amino acid). Velvet's algorithm hashes the input sequence into a series of k-mers, turns them into nodes, and forms edges between nodes that have an overlap of k-1. Below is an example of 3 10-mers overlapping to reveal the original sequence.

A T C G G A C T C A	10-mer 1
T C G G A C T C A G	10-mer 2
C G G A C T C A G A	10-mer 3
A T C G G A C T C A G A	**Original Sequence**

Velvet turns the thousands of reads from an NGS run into millions of k-mers by "hashing" them (a fancy word for chopping up reads and turning them into equal length "words"). It then assembles these millions of k-mers in a process like shown above until "scaffolds" are formed. Scaffolds essentially are linked up k-mers that no longer have k-1 overlaps with other k-mers, representing the largest pieces of DNA that Velvet can discern with the given parameters.

Running Velvet: A Soft Tutorial

Below is a short tutorial on how to process raw NGS data - in fastq format - into assembled scaffolds using Velvet.

Scenario:
A local hospital found a contaminating bacteria in their emergency rooms. After swabbing, streaking, and other detection measures that we don't need to care about in genome assembly, they were able to perform NGS on a single colony of these contaminants. The nice technicians at the sequencing core handed a pair of paired read fastq files to you. The expected coverage is 30X, and the read length is 200bp. Armed with a laptop, a Linux/Mac OS, and an acceptable internet connection, it is now up to you to find out what the contaminant is.

You will need:

1. Velvet
2. Download one part of a paired read: https://drive.google.com/open?id=1YjVqsGd_apGeG5XafGoGExnImPQHDewN
3. Download the other part: https://drive.google.com/open?id=19zTiDr0mfzzPF9-kWvinyeLhKBsag1xc

Now let's dive right into it.

1. Create a working directory for your project. It usually is a good idea to make your directory name indicative of the project as well as dating it. For example:

   mkdir -p path/to/hospital_contaminant_082019

2. Copy the two fastq files to this directory for ease of use.

   **cp path/of/{SRR396636.sra_1.fastq,SRR396636.sra_2.fastq} **
   path/to/hospital_contaminant_082019

3. Now we can work! First, let's use Velvet to hash your fastq files. Hashing as explained above turns your reads into millions of k-length "words"
   Below is the structure of the velveth command line:

   velveth [output directory] [k-mer/hash length] [various options…] [input file(s)]

   Output directory: This directs velveth to pick the directory to which it will dump its output (there will be several files). If the directory doesn't exist, velveth will make a new directory with the given name and dump its output in it.
   k-mer/hash length: The length of k-mers that velveth's hashing will produce. Shorter k-mers will produce results faster, but longer k-mers will give the best results with the longest scaffolds - meaning that more reads will be connected. You should try out different hash lengths to see for yourself! 50 is a good starting number.

Various options: There are various options that you can specify. For a full list, visit the documentation linked above. For now, all we care about are -fastq and shortPaired, as we are working with fastq files and short, paired-end data.
Input files: Your input files will be the fastq files you downloaded. So, for this case, SRR396636.sra_1.fastq,SRR396636.sra_2.fastq.

Let's put it together: Try hashing the reads with a k of 50.

velveth k50 50 -fastq -shortPaired SRR396636.sra_1.fastq,SRR396636.sra_2.fastq

This will result in the k50 folder being created and populated with the output of velveth. The output files will be used by the subsequent program: velvetg.

4. Finally, the core of the tool. Velvetg constructs a de Bruijn graph using the hashed k-mers to create scaffolds as large as possible given the parameters with the specified directory a la velveth. For paired-end reads, the expected coverage and read lengths have to be specified. Thus, the following command will suffice for velvetg. We know that for this scenario, the read length is 200 bp and the coverage is 30X. So..

   velvetg k50 -ins_length 200 -exp_cov 30

5. Note that it **will** take a while. But once it's done, congratulations! You now have the assembled scaffolds in the output directory. Take note of the output on the terminal, because it will tell you a very important stat: N50. It is a very common stat to use in genomics to describe assembly quality, and it can be described as "Contig or scaffold N50 is a weighted median statistic such that 50% of the entire assembly is contained in contigs or scaffolds equal to or larger than this value". Low N50s tell you that there are a large number of small scaffolds, meaning the assembly is likely to be poor. So, the bigger the better!

6. Now check out the output files. There are two important files that you should look at:
   a. **contigs.fa** : What you came here for. This is a list of all scaffolds that velvet was able to generate. Using the scaffolds, you can try to identify what organism it is by BLAST or other means.
   b. **LastGraph** : The final iteration of the de Bruijn graph that velvet made. It describes to you the connection of all the scaffolds as nodes and you can visualize it with software like Gepard.

7. Good job! You are finished. What you can do now is to try different hash lengths (k-mers) and repeat this experiment to see if you can achieve higher N50. Since you started with 50, try going to 75 to see if you get better results.