Ben Siranosian
CSCI1820 hw5

GENOME ASSEMBLY – PROJECT REPORT

OVERVIEW

My genome assembler has two main components. A "read simulator" simulates the process of DNA sequencing by reading a template genome and producing a set of DNA fragments similar to the output of a sequencing machine, given parameters and assumptions. A genome assembler takes as input a set of DNA reads (either produced by the generator or from an external source), constructs a De Bruijn graph for a given *k,* corrects for errors, and returns a list of contigs representing the assembly of the genome. The genome assembler uses the velvet algorithm with some adaptations to do assembly. In this report I will describe each component in detail.

READ SIMULATOR

Code for the read simulator can be found in "simulate_reads.py". Simulation of DNA sequencing reads is done according to 5 parameters: the input *genome*, *coverage, length, error rate,* and *reverse complement*. The input genome is the name of a fasta file that will be used to simulate the reads. *Coverage*, *length*, and *error rate* are all interpreted according to Lander-Waterman statistics. *Reverse complement* determines if the reverse complement of all reads should be included in the output.

Coverage is the desired number of times a given base from the genome will appear in the read set. Along with *length* it is used to determine the number of reads.

Length is the desired mean length of reads in the output. Actual read lengths are drawn from a normal distribution with mean *length* and standard deviation 3. The number of reads in the output is determined by $N = \frac{C}{G*L}$ where C is *coverage*, G is the length of the genome, and L is *length*.

Error rate is a number in [0,1] to model the error rate in DNA sequencing. We assume all errors are substitutions and add them randomly by drawing a random number for each base in each read.

Reverse complement is either 0 for false or 1 for true. If true, the reverse complement of every read is included in the output. It's important to note that coverage does not take into account the number of reads coming from the reverse complement of a given base.

GENOME ASSEMBLER

The genome assembler has many functions that come together to produce a finalized output. Code for each function is found in a separate python file in the "code" folder. These functions shouldn't be called individually, they work as part of a larger script that calls them in order.

The first function is make_kmers. This takes in a list of reads (produced by the simulator or any other source) and constructs a list of kmers for the given value *k.* If a read is shorter than *k,* it is discarded.

The second function is construct_de_bruijn_velvet. This takes in a list of kmers (produced by make_kmers) of a consistent length and constructs the basic De Bruijn graph for them. I implement the De Bruijn graph

where nodes are *k-1*mers and edges are *k*mers. I use a networkx directed graph for my internal implementation. Coverage data is maintained on each individual node.

The third function is simplify. This takes an input networkx directed graph and does the simplifying operations as specified in the velvet algorithm. Specifically, if a node *A* has one outgoing edge that points to a node *B* that has one incoming edge, the two nodes are merged. The resulting sequence is the union of the two nodes, and the resulting coverage is the sum of the coverage of each node. Simplification is the longest part of the assembly algorithm.

The fourth function is remove_tips. This takes an input networkx directed graph and removes tips (from errors at the edge of reads) as described in the velvet algorithm. Tips are identified as having an in or out degree of 0, a sequence length of less than *2k* and a less frequent alternative to another path. If a tip has an incoming edge, less frequent meads a different neighbor of the predecessor of the tip has higher coverage. If a tip has an outgoing edge, this means a different predecessor of the neighbor of the tip has higher coverage. Positively identified tips are simply removed from the network. After removing tips, the graph should be simplified again.

The fifth function is resolve_repeats. This function recognizes cycles in the graphs and maps them to single strings. The number of times a string should be repeated is calculated by the average coverage of the nodes in the cycle divided by the average coverage of the nodes incoming and exiting the cycle.

The sixth function is tour_bus. This is the main error correction function I use, an implementation of the tour bus algorithm in velvet. This algorithm starts an exploration from every node. Nodes are visited in order of increasing distance from the start node. When a node that has been previously visited is discovered, it backtracks to the first node before the split and extracts the sequences from the path. The sequences are then aligned, and if similar enough (a threshold is set based on the alignment parameters) the two paths are merged. The consensus path is the one that the algorithm discovered first (shorter path). At each position along the path, nodes are compared and merged if similar enough. The non-consensus nodes has all of its neighbors and predecessors mapped to the consensus node and merging continues down the path.
I noticed two problems with tour bus that I correct for later in the algorithm. First, chains of nodes with slight mismatches between them can be created. These chains can't be simplified directly but are dealt with in a later step. Second, structures where a node points to two neighbors, one of which points to the other neighbor, remained after tour bus. The longer path seemed to be extraneous so it is removed in a later step.

The seventh function is assemble_paths. I found that after tour bus completed, three typed of structures were left in the graph: isolated nodes, chains of nodes that weren't error free and couldn't be simplified, and structures where a node pointed to two neighbors, one of which pointed to the other neighbor. Assemble_paths corrects for these errors. Chains of nodes with mismatches are simplified if they have enough of an overlap with a maximum of 2 mismatches. The third structure left over from tour bus is

simplified by removing the longest path because I found the longest path was an error-prone extension of the shortest path.

The last function is remove_low_coverage. Much like velvet, I remove nodes after all the corrections with a simple coverage cutoff. By default, nodes with length less than 2k or coverage less than 10% of the maximum value are removed.

Error correction is the sum of the resolve repeats, tour bus, assemble paths, and remove love coverage algorithms. Not much changed from my first week plan – I implemented the algorithms I had planned to and didn't do any major restructuring of my code from the first week.

Output files

The simulated reads from the SigmaKappa genome can be found in "sigma_readout.txt"
The assembled contigs from the Sigma Kappa genome can be found in "sigma_contigs.txt"
The assembled contigs from the provided reads can be found in "provided_read_contigs.txt"

Playing with parameters

Changing the parameters of the read generator had a large effect on the quality of the assembly. Producing a read set at higher coverage improved the assembly up to a certain point, when it became more of a burden than a benefit. Additionally, more reads with errors are generated at a higher coverage that con potentially complicate the assembly (although the error reads will still be low frequency compared to the rest).
The error rate also had a large effect on the quality of the assembly. Without errors, I was able to assemble perfect genomes at low coverage if I was lucky. Even a small error rate (like 0.01) drastically complicated the assembly, decreased the lengths of contigs, and increased computational time.
Read length obviously correlated with assembly quality. Theoretically, a higher read length allows for resolution of longer repeats, but I didn't observe an effect like that with the provided genomes.
I found k to also have a large effect on assembly quality. I initially assembled graphs at k=20, but noticed a large jump in quality by moving to k=25. Setting k=30 sometimes improved the assembly, but not as much as the first jump. The optimal k for these genomes is probably in the 25-35 range.

Second week's genome

I actually didn't find the second week's genome more difficult to assemble than the sigma genome. My assembler produced a similar number of contigs and average contig size for both genomes. Repeats didn't seem to be a problem in the provided read set (after simplifying the de bruijn graph at k=25, there were no simple cycles).

Percent coverage

My assembly produced 89 contigs covering about 11300bp total. To find the percent coverage, I have to determine how significant the overlaps between contigs are. I computed the mappings of all contigs to the genome (with a small number of errors allowed) and reported the result in

sigma_contig_genome_map.png. Although I didn't get an exact number, the percent coverage looks fairly high!

CONTIG MAP

A contig map for the Sigma genome assembly can be found in sigma_coverage.png.