David Galambos
Blake Johnson

CS362 De Bruijn Assembly Report

Here we present our *de novo* assembly algorithm ViralGreed (VIRtual AssembLy, GREEDy). Our assembler is best suited to assembly of short genomes <5 kbp (such as a viral genome) with high error rates and coverage. In addition to being submitted on Moodle, all of our files, code and README can be found at this Git repository.

Simulator

Design and Implementation

First, our simulator uses values for read length *L* and coverage *c* (from user input) to calculate the number of reads *N* to generate.

To generate a single random read, we use a random number generator to pick an index *i* between -*L* and the end of the input genome. This becomes the starting index of our read, and the end index is *i* + *L*. We then attempt to create the read by sequentially picking individual characters between these indices from our input genome. If any positions in the read occur before the beginning or after the end of the genome, we simply ignore them, leading to a read shorter than *L* that covers the first or the last character of our genome. The assembler deals with these extremely short reads by getting rid of them if they are smaller than the k-mer size. Each time we pick a character from our genome, there is an *e* chance that it will convert to one of the other three DNA bases. This step for creating a single read is done *N* times; at the end, we write our reads to `reads.txt`.

This method of allowing reads to start and end ‘outside’ of our input genome ensures that we get relatively uniform coverage *c* across our whole input genome. While real sequencing technologies do not always produce the same coverage at the end of a genome as in the middle, uniform coverage is useful in our case because our algorithm uses the degree of the nodes in our De Bruijn graph (which corresponds to coverage) to find the path that produces our output. We chose our standard read length (150 bp) based on the read lengths usually produced by Illumina next-generation sequencing. We chose an error rate of 0.05 because we wanted to showcase the error-correcting abilities of our algorithm while using an error rate equal to or higher than the Illumina next-generation error rate (0.01).

Creating Randomized Genomes

We created a file `make_genome.py` which will create a randomized “genome” of the argument size, with nucleotides added completely randomly. We also created a brilliant little shell script, `test.sh`, which takes 2 arguments: a genome size and a number of trials. For each trial, it will generate a random genome of that given size, simulate reads of that genome, assemble, and then output a true or false value for if it's 100% identical to the original genome. While testing if the genome is perfectly identical is definitely not the best technique, it proved extremely useful and gave us tons of data, which we have kindly shared below. By default, test.sh runs with parameters of coverage 25, read length 150, error rate 5%, and k-mer length of 13.

Easy and Hard Datasets

Our algorithm finds a path in the De Bruijn graph by greedily following a path with the maximum number of edges from each node. When it encounters a loop (ie. a branch where all direction add the same number of edges to the path), it chooses randomly. Therefore, it depends on coverage (which corresponds roughly to edge multiplicity) to find correct paths that don't contain repeats, and the inclusion of loops is random. If k-mer length is treated as constant, this implies that longer sequences will likely include more repeats. Overall, an easy dataset for ViralGreed is short (<5000 bp), has high coverage (>25x) and few repeats (a randomly generated dataset). An easy dataset for ViralGreed does not necessarily require a low error rate, thanks to high coverage and greed eliminating virtually all error. Our easy dataset contains reads with coverage 25, length 150bp, and error rate 0.05 from a genome of 2kpb. This yielded an output of a single contig with a 100% match to our original input genome.

A hard dataset for our algorithm is anything with lots of repeats (or >5 kbp), especially if it has a repeated beginning section. This will completely throw off ViralGreed as its searching for the beginning, causing it to start off on a bad foot and probably miss a good section of the dataset. Errors are easy for ViralGreed, but bad coverage hurts our algorithm as well, given that it returns just one greedy contig. For our hard dataset, we created a set of reads with coverage 10, length 150bp, error rate 0.05 from a genome of 9kpb. This yielded an output of a single contig of 3350 bp, that, upon inspection, contains at least the first ~10 nucleotides of our original genome.

Assembler

Design and Implementation

Our algorithm, nicknamed ViralGreed, is an extremely fast greedy algorithm best suited for viral genomes (3-5 kbp) with high coverage (about 25) and read length designed for next-generation and third-generation sequencing (>150bp and 1-3kbp, respectively). Our decision to create a greedy algorithm lead us to not just list out all the possible contigs, but instead greedily choose the ideal longest contig. **Because assemble.py always outputs exactly one contig, we do not output an N50 value.** It is very efficient at dealing with errors (.05% error rate tested) and it is practically a linear time algorithm, with just 1 N^2 time complexity step (finding the greediest starting position). ViralGreed's greedy nature means it's fast, which means it's possible to run several iterations of ViralGreed to insure success.

First, our algorithm breaks down the reads. As it does so, it remembers the first k-1mer for each read so that it is able to look at possible beginning positions quickly. Our algorithm then creates a De Bruijn graph with all possible connections between nodes. Because it uses python dictionaries, this step is O(N log N) time (placement and lookup are both log operations for dictionaries). Then, it scans through its list of possible starting positions for the greediest possible start (most connections to ANY other node). This works because usually the node with nothing pointing to it that has the most connections away from it is the starting node without any errors, while the others are often erroneous reads. After determining a likely starting position, our algorithm is very simple and follows the path with the most connections. With k-mer length as *k*, ViralGreed begins to keep a running average of the node multiplicity after it passes the *k*th node. It is able to predict when it might be hitting a repeat (if the number of connections suddenly doubles or jumps relative to the average), and it is able to assure that it doesn't infinitely loop on repeats (by subtracting the average number of edges as it goes by). Importantly, ViralGreed only modifies the running average when the multiplicity of the current node suggests that it is outside a loop. However, our program is still very far from ideal for dealing with repeats, due to its greedy nature of taking the densest edge path and breaking ties at random. This is why a high read length and high coverage help our program so much-- its error-handling abilities are much much better than its repetition abilities.

Testing

We read in the k-mers and construct a de Bruijn graph successfully, as can be seen with the dot file for a small string. The dot file construction automatically skips when it's predicted to be very large. We created a python file `test_assembly.py` which will return True if the sequence in the 2 files provided is identical. Given more time, we would have implemented a global-alignment style algorithm that checks the differences between the original and assembled genomes to give a better sense of how well we did. However, ViralGreed's incredible speed makes running repeats incredibly easy: running a genome of 2 kpb takes about .6 seconds on a simple laptop computer (coverage 25, read length 150, error rate 5%, k-mer length 13).

We also performed testing on the optimal k-mer length for an easy dataset of 2000bp with high coverage (Fig. 1). We found that a k-mer length of 13 or 14 was optimal with this genome length; it returned our original genome as a single contig with 100% accuracy 90% of the time. Due to the limitations of `test_assembly.py` described above, we cannot currently evaluate the quality of the remaining 10% of the assemblies. We believe that k-mer lengths of less than 13 cause more loops in our De Bruijn graph (shorter k-mers are more likely to be repeated). Since our assembler handles loops at random, this allows more chances to miss repeats. In contrast, the drop off in accuracy with k-mers larger than 15 is likely due to a decrease in repeats and thus the multiplicity of the nodes in the De Bruijn graph. If the multiplicity of the correct path in the graph drops too much and becomes closer to the multiplicity of the erroneous paths, our greedy algorithm is more prone to choosing those erroneous paths. A k-mer size of 13 strikes a balance between too many loops and too low multiplicity for the correct path.

David Galambos
Blake Johnson

Following our determination of the best k-mer size, we tested the assembly of genomes of various sizes using k-mers of length 13 (Fig. 2). We assembled genomes of length <7000 with 100% accuracy more than half the time. Again, the limitations of `test_assembly.py` don't allow us to evaluate any imperfect assemblies.

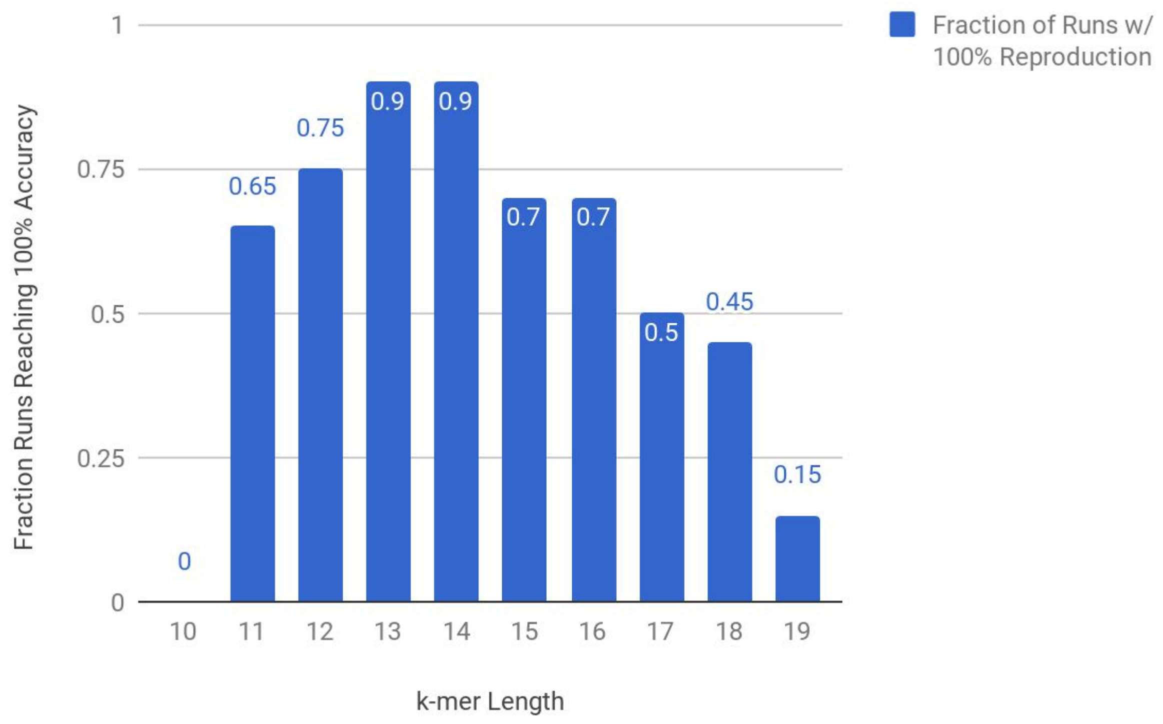

Fig. 1. Assembly Accuracy vs. k-mer length for a 2000bp genome. We ran 20 runs of `assemble.py` with varying k-mer length on reads from randomly generated genomes. For each run, we generated a new random genome, created a new dataset of simulated reads using `simulator.py`, and ran `assemble.py`. We report the fraction of the runs that produced an output with 100% match to our original genome as reported by `test_assembly.py`. The simulated read datasets were created with constant parameters of coverage 20, read length 150 bp, and error rate 0.05.

David Galambos
Blake Johnson

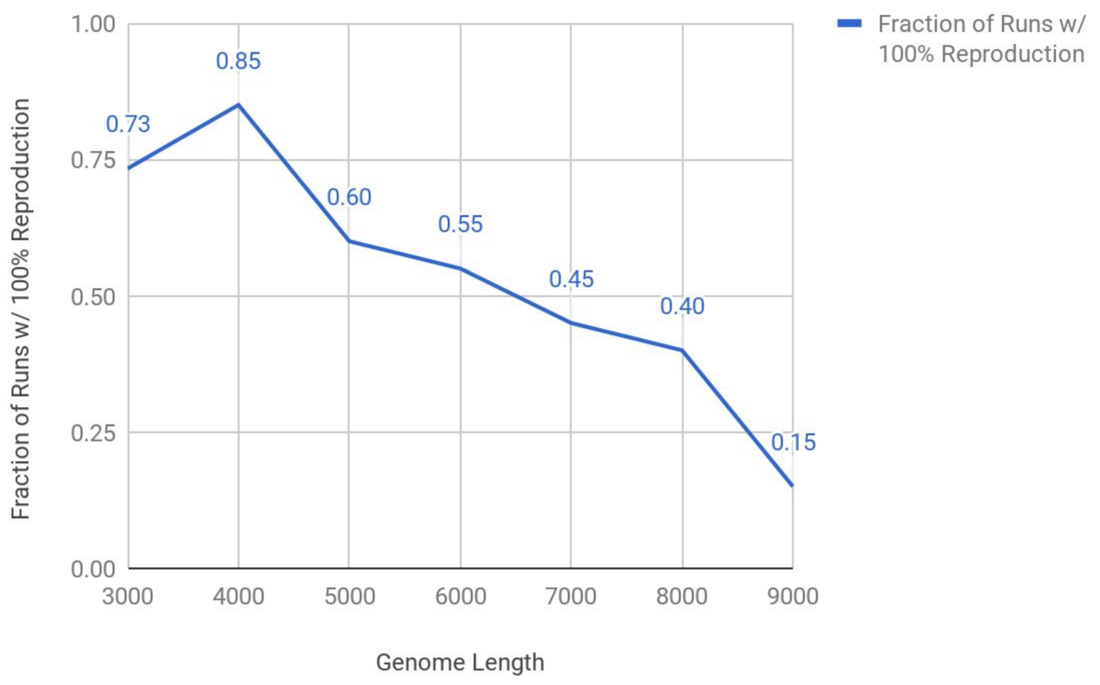

Fig. 2. Assembly Accuracy vs. genome length with k-mer length 13. We ran 20 runs of `assemble.py` with varying genome length on reads from randomly generated genomes. As an exception, we had 30 runs of 3000 bp. For each run, we generated a new random genome with `make_genome.py`, created a new dataset of simulated reads using `simulator.py`, and ran `assemble.py`. We report the fraction of the runs that produced an output with 100% match to our original genome as reported by `test_assembly.py`. The simulated read datasets were created with constant parameters of coverage 20, read length 150 bp, and error rate 0.05.

Assembly of `sample.fasta`

Assembling the sample.fasta file was extremely difficult, because our algorithm, ViralGreed, is great at long read lengths and high error rates, and awful at lower read lengths and lower coverage. We used a k-mer length of 13 based on our previous results showing that this is optimal for short sequences. In addition, preliminary testing (not shown here) suggested that much higher (>15) or much lower (<10) k-mer values produced very suboptimal results for `sample.fasta`. Thus, we did not attempt different k-mer values specifically for this assembly. In addition, because our algorithm is greed-based, when it doesn't get the 100% correct sequence it will often have gone off-track, sometimes with spectacular results. However, while our algorithm might not get all the repeats, it will usually find the start and end with success, as seen by our `sample_contigs.txt,` which is our output file for `sample.fasta`. Our code probably got most of the file (we got exactly 10kbp), because it got the start and end, which are key points for ViralGreed, as it indicates not getting caught in a wrong path and finding the correct start. It probably missed several repeats, and quite possibly got many more repeats out of order. This is common for our algorithm, given how it has a really hard time following repeats. However, I’m satisfied with how it performs, especially with the speed.