**Benchmark Comparison of Velvet and Greedy Overlap *de novo* Assemblers**

Andrew Gordon; Karthik Handady; Lee Korshoj; Alex Okeson

*University of Colorado Boulder; CSCI 4314/5314, MCDB 5314; Final Report; 5/3/2016*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Several programs exist for genomic assembly from shotgun sequencing data. These programs will need to become more efficient and utilize new algorithmic approaches as next-generation sequencing technologies continue to produce shorter reads and larger coverages in comparison to older Sanger sequencing methods. In this paper, we present a benchmark comparison of two programs that employ different assembly algorithms. The first program, Velvet, is currently one of the most widely-used assemblers. It operates with a de Bruijn graph assembly and was developed by Zerbino and Birney in 2008. The second program is our assembler, which operates with a greedy overlap approach. The performance of both the Velvet assembler and our greedy overlap assembler was benchmarked on the resulting N50 in relation to four variables: read length, repeats, sequencing errors, and coverage. We found that*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Introduction**

Progress in genomics is dependent upon sequencing and the ability to process large quantities of sequencing data to determine complete genomes. Although scientists have been successful in reading the human, mouse, and several other genomes, the process remains limited by cost and throughput of current technologies [1]. As researchers seek to further understand complex diseases, pathogenicity, and evolution, improvements to the bioinformatics tools necessary for genome construction will remain in high demand. Next-generation sequencing technologies, such as 454 Pyrosequencing and Illumina-Solexa, produce millions of short “shotgun” reads (often 30-100 bp) from a longer unknown sequence [2]. Assembly programs must use these millions of shotgun reads to piece together one or more contiguous sequences (contigs). Several programs exist to execute this whole-genome shotgun (WGS) assembly process, and most employ one of three main assembly algorithms: greedy, overlap-layout-consensus, and de Bruijn graphs. The greedy algorithm sequentially pieces reads together in a pairwise fashion from those with most overlap to least overlap. The overlap-layout-consensus approach represents each read as a node on a graph and each detected overlap as an arc between the appropriate nodes. Contigs are generated by navigating through the graph node-to-node. Assemblers using the de Bruijn graphs method represent words from the sequencing reads as nodes on a graph and the resulting overlap and an arc between the nodes. Due to the nature of next-generation sequencing, the sheer number of reads makes the overlap graph, with one node per read, extremely large and lengthy to compute. The de Bruijn graphs approach alleviates this by organizing its graphs around words instead of reads. This is one of the main reasons why most emerging *de novo* assembly programs employ a de Bruijn graphs solution. In this study, we benchmarked two algorithms associated with this *de novo* genome assembly. The first was a de Bruijn graphs algorithm used by the Velvet program, and the second was a greedy overlap algorithm implemented by our group.

**Velvet de Bruijn Graph Assembly**

One of the most prominent *de novo* assembler is Velvet, which uses de Bruijn graphs to assemble contiguous sequences from the shotgun reads of next-generation sequencing platforms. The developers of Velvet claim that the structure of a de Bruijn graph makes Velvet cleaner and simpler than the assembly methods employed by other programs. The main reasoning behind this claim is the aforementioned fact that since graphs are not organized around reads, but around words, high redundancy is easily handled and errors are quickly resolved [3].

In a de Bruijn graph, each node, , is a series of overlapping k-mers, where adjacent k-mers overlap by k-1 nucleotides. The series of last nucleotides in each k-mer is stored as the sequence of the node. For example, if a node consists of the 5-mers ACTGA, CTGAT, TGATT, and GATTG, then the sequence of the node is ATTG. Each node has a twin, , which represents the last nucleotides of all reverse complement k-mers of the node, . Each combination of node and twin is a block. The nodes are connected by arcs such that the last k-mer of an arc’s origin overlaps with the first of its destination. There is also symmetry between nodes and twins, so if node arcs to node , then node arcs to node .

Construction of the de Bruijn graph begins by hashing all sequences into predetermined k-mers. The length, k, is set based on size of the reads. If an average read is 25 bp, k would be set to 21. Next, two data sets or “roadmaps” for the graph are created. The first roadmap rewrites each read as a set of original k-mers combined with overlaps. Every time a k-mer is observed, a hash-table records its read number and position within the read. The second roadmap records which of original k-mers are overlapped by subsequent reads. Uninterrupted sequences of original k-mers become the nodes. Velvet then uses both roadmaps to proceed from one node to the next, creating new directed arcs and incrementing existing arcs.

After the de Bruijn graph is constructed, it is simplified and corrected for errors. The simplification process results in no loss of information. It looks to condense the graph to reduce memory usage and calculation times. Nodes are created each time reads start and end, which can lead to the formation of “chains” of nodes linearly connected as subgraphs. To de-fragment these chains, the program looks for nodes that have only one outgoing arc that points to another node with only one ingoing arc. Whenever this is seen, the two nodes are merged. Long chains of nodes can therefore be condensed into a single node.

Following simplification, three types of error are removed by the Velvet program: “tips” result from errors at the edges of reads, “bulges” result from internal read errors, and erroneous connections result from cloning errors or merging tips. The first type of error, tips, are chains of nodes that are disconnected at one end, and they are removed based on their length and connectivity. The second type of error, bulges, are redundant paths that start and end at the same point. They are resolved with a “Tour Bus” algorithm. The algorithm begins at an arbitrary node and progresses along the graph until a previously-visited node is encountered. Backtracking allows the program to identify the closest common ancestor between the node and starting location. The sequences from the two retraced paths are extracted and aligned. If similar, the sequences (and nodes) are merged. The final type of errors are erroneous connections. Velvet uses a coverage cutoff value to rid of extra connections.

In the absence of ideal data sets, some problems remain unresolved. The de Bruijn graph is broken if there is a repeat longer than k-mer length. This could force the program to use only short k-mers in certain situations and require additional paired read information for complete resolution. Also, resolution of long repeats leads to incorrect assembly, specifically regions being omitted. Graph construction appears to be the main bottleneck of Velvet in terms of speed and memory. To download and find more detailed information on the layout and performance of Velvet, visit the official website: https://www.ebi.ac.uk/~zerbino/velvet/.

**Greedy Overlap Assembly**

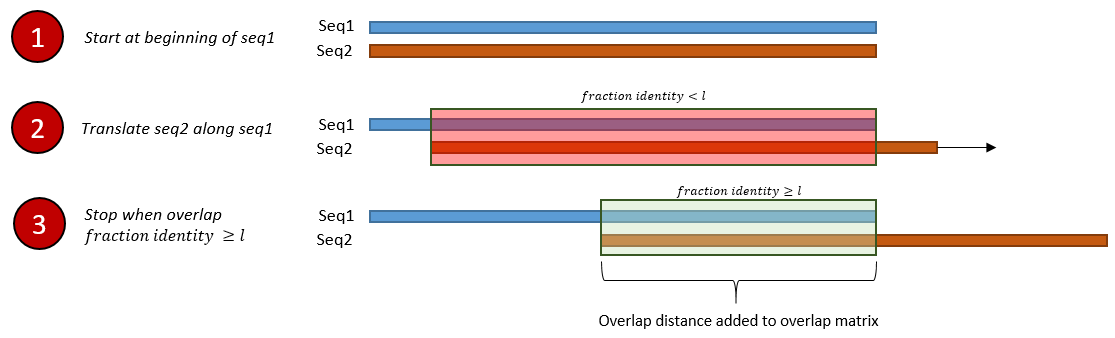
We employed a greedy overlap approach for *de novo* assembly. The first assembly programs that were created used greedy algorithms where the most similar reads are joined in a stepwise manner. This algorithm has proven simple and effective at finding the shortest common supersequence (SCS), or the shortest sequence that contains all sub-sequences. In comparison to Velvet, we expect to see larger, more complex data structures since our algorithm processes each read, instead of breaking the reads into words. This difference will become more prominent when processing larger sequencing datasets [4].

The greedy overlap algorithm consists of two main steps:

1. Calculating overlap
2. Greedy combination

In the first step, the overlap between each pair of reads must be quantified. This is accomplished based on a user-specified minimum fraction identity, . To achieve more accurate assembly, this fraction should be kept close to unity. For each read (seq1), the program looks at every other read (seq2) and calculates the length of the overlapping region containing at least the minimum fraction identity, . To do this, we employ a translational comparison algorithm. First, seq2 is aligned at the starting position of seq1. It is then translated one nucleotide at a time until the fraction identity, , is reached. At that time, the number of overlapping nucleotides is recorded in an overlap matrix holding overlap scores for all combinations of reads. If the fraction identity, , is never achieved, the two reads will receive an overlap score of zero. In general, the overlap score denotes the overlap length between the suffix of seq1 and the prefix of seq2. The translational comparison process is shown in Fig. 1.

In the second step, the reads are greedily merged together in a stepwise fashion from pairs containing the most overlap to least overlap. To identify two sequences to merge, the program takes the maximum value from the overlap matrix generated in the first step. At each combination, new overlaps must be calculated between the newly-merged sequence and all other reads using the same translational comparison method previously described in the first step. The greedy combination process, and subsequent calculation of overlaps, is repeated until there is only one sequence remaining or the largest value in the overlap matrix is zero. If one sequence remains, the assembler was able to combine all reads into one large contig representing the genomic sequence of interest. If the greedy combination process ceases due to a maximum overlap matrix score of zero, then the final assembly consists of multiple contigs representing a fragmented illustration of the sequence of interest.



**Fig. 1** – Calculating overlap between reads for the greedy overlap assembly algorithm.

Previously, problems with greedy assembly have been seen when the read lengths are too short, there are long repeats, or sequencing errors. For each of these cases, the greedy algorithm has tendencies to omit regions and fragment contigs. To combat these known issues, we borrowed several aspects from assemblers that use an overlap-layout-consensus strategy, such as the Celera Assembler [5]. Thus, we call our algorithm greedy overlap. Filtering and the translational comparison overlap determination are two such aspects. Reads that are too short, or show a high error rate, can be filtered out prior to the overlap analysis. Then, instead of using a specified minimum overlap distance, we use a specified minimum fraction identity, . Allowing the overlap between two sequences to have less than 100% identity helps handle single nucleotide sequencing errors and allows smaller reads to be merged with larger reads instead of leading to fragmentation.

**Benchmarking Procedures**

For a benchmark comparison between Velvet and our greedy overlap assembly, we explored the effects of four independent variables on N50 and accuracy:

1. Read length
2. Repeat regions
3. Error-prone reads
4. Minimal coverage

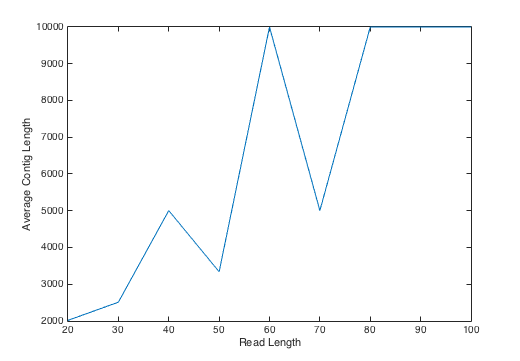
Code was written to generate simulated next-generation sequencing data where one independent variable (read length, repeat regions, errors, and coverage) could be altered for each test case. A long, mock-genomic sequence comprised of random sets of nucleotides *A*, *T*, *G*, and *C* was first generated. Then, the sequence was split into a set number of total reads with a desired overlap length and coverage. The independent variable of interest was altered in each case.

In the first test case, the effect of read length on the average length of the output contigs was explored. Read length was increased while keeping repeats, error rate, and coverage constant. The second test case varied the length of a repetitive region to test how the assemblers resolved this important genetic aspect. Read length, error rate, and coverage were kept constant. The third test case altered the coverage while maintaining constant read length, repeats, and error rate. The fourth test case introduced errors into the reads at various levels to see how assembler performance was effected. Read length, repeats, and coverage were held constant. Detailed results are shown in the following two sections.

**Velvet Results**

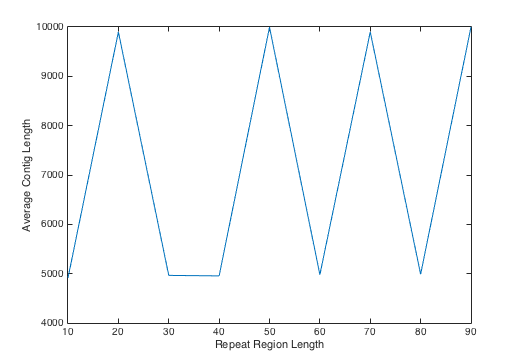
**Greedy Overlap Results**

As expected, when there was a large read length, our Greedy Overlap program correctly assembled the sequence and returned one contig that was the same as the original sequence. As the read lengths got smaller, the program was unable to assemble the entire sequence and returned several smaller contigs that were subsequences of the original sequence.



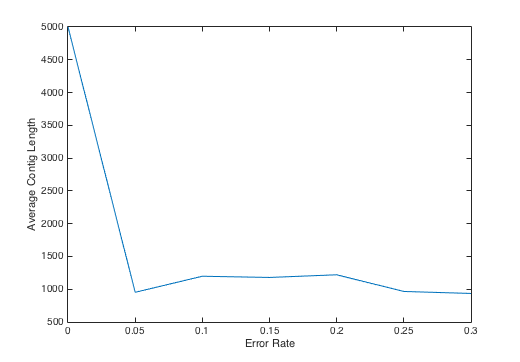
**Fig. 2** – Results with varied read lengths. The original sequence length was 10,000 nucleotides, error rate was 0%, overlap was 10 nucleotides, and no repeated regions. In the Greedy Overlap algorithm the smallest allowed overlap region matching percentage was 90%.

Our algorithm did very well with varied numbers of repeated regions. In each of the test runs, the algorithm returned either one or two contigs. If one contig was returned, it was either the same as the original sequence, or very similar but missing one or two of the repeated regions. If two contigs were returned, they were two halves of the original subsequence that the program could not figure out how to connect to get the original sequence.



**Fig. 3** – Results with varied number of repeated regions. The original sequence length was 10,000 nucleotides, the read length was 100 nucleotides, error rate was 0%, and overlap was 10 nucleotides. In the Greedy Overlap algorithm the smallest allowed overlap region matching percentage was 90%.

Our implementation did not do very well with error prone reads. The program was very slow and did not produce very good results, even when there were few errors. We did implement a scoring system to try to allow for errors in the overlap, but it was very basic and therefore did not perform well. In future iterations of the program, we hope to improve this feature.



**Fig. 4** – Results with error prone subsequences. The original sequence length was 5,000 nucleotides, the read length was 100 nucleotides, overlap was 10 nucleotides, and no repeated regions. In the Greedy Overlap algorithm the smallest allowed overlap region matching percentage was 90%.

When we attempted to change the amount of coverage of the original sequence with multiple reads, our program was not able to run. For each of the test cases ran, our program ran for a long time before crashing with a “maximum recursion depth exceeded” error. We think this is because our algorithm does not check whether or not two reads are the same, so if reads are very similar, the part of our algorithms that scores the matching of suffixes and prefixes will run for many times. This results in a recursion depth error. We also hope to experiment more with this to find the exact source of the problem and improve it in future implimentations.

**Discussion and Conclusions**

**References**

[1] International Human Genome Sequencing Consortium. 2001. Initial sequencing and analysis of the human genome. *Nature* 409: 860–921.

[2] Bentley, D.R. 2006. Whole-genome re-sequencing. *Curr. Opin. Genet. Dev.* 16: 545–552.

[3] Zerbino, D.R. and Birney, E. 2008. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research.* 18:821–829.

[4] Langmead, Ben. 2013. Assembly & shortest common superstring. http://www.langmead-lab.org/teaching-materials/

[5] Myers, Eugene, et al. 2000. A Whole-Genome Assembly of Drosophila. *Science*. 287: 2194-2204.