Final Report

Team: HiA+
Group members: Lilin Huang(lilinh), Tianyue Zhang(tianyue3), Wenduo Cheng(wenduoc)
Date: 12/16/2022

Introduction

The de novo assembly of shotgun reads to form a set of contiguous sequences (contigs) is a critical step in genome sequencing (Zerbinoet al., 2008). De novo genome assemblies are essential because they assume no prior knowledge of the source DNA sequence length, layout, or composition, which might be particularly useful for mapping unknown organism genomes or completing known organism genomes.

The De Bruijn graph is commonly used for assembling short-read data, where reads are broken down into kmers (substrings of length k) that form nodes and are linked when sharing a kmer (Ekblom & Wolf, 2014). The value of k is essential for constructing de Bruijn graphs. A large value of k could remove some short repetitive regions, thus reducing the number of nodes in the de Bruijn graph; however, this could cause problems that increase the number of unconnected subgraphs and the number of gap regions. On the contrary, a small value of k could reduce some gap regions, thus increasing the connectivity of de Bruijn graphs, but could result in adding more nodes and increasing short repetitive regions (Liao, X. et al., 2019). As shown in Figure 1, a smaller k results in a more complicated de Brujin graph (1A), while a larger k makes the graph disconnected (1B). Therefore, the choice of kmer size involves a tradeoff between different things.

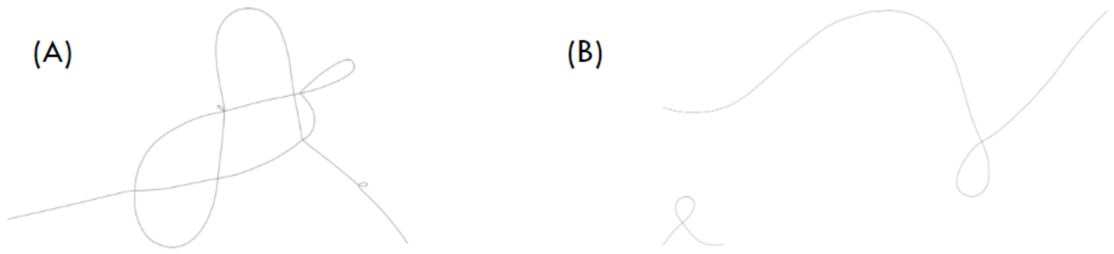

Figure 1: The de Bruijn graph constructed using kmer length of 15 and 25.

A few popular de novo assembly packages are implemented based on the de Bruijn graphs, such as SOAPdenovo2(Luo et al., 2012) and Velvet (Zerbino & Birney, 2008). To use these packages, users are asked to provide a value of kmer size. Inexperienced users might find it challenging to choose an appropriate value of kmer length. To address this problem, we desired to create an assembler that advises choosing kmer length.

In this project, we developed a short read genome assembler named DAH based on the de Bruijn Graph data structure using Golang, which incorporated kmer length advice and graph visualizations. The source codes are available on GitHub (https://github.com/wenduocheng/DAH/). Currently, DAH supports as input the fasta format, which is commonly used to represent nucleotide or amino acid sequences of nucleic acids and proteins. To test the performance of our assembler, we generated a fasta file containing pseudo reads from a real genome. DAH successfully produced a list of contigs. The assembly result showed a relatively high N50, which is a statistics often used to evaluate the quality of a set of contigs in a genome assembly.

Methods

Hash the Kmers

A kmer is a k-length substring of a read. DAH breaks up each read into kmers and stores the kmers in a hash table. The key of the hash table represents the kmer, and the value represents the frequency of the kmer. Figure 2 demonstrates how DAH breaks the read “ATGCGTGGCA” into kmers.

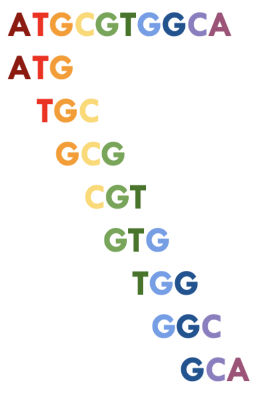

Figure 2: DAH breaks up the read “ATGCGTGGCA” into kmers.

Construct the de Bruijn Graph

The de Bruijn Graph is built based on the overlaps between the short reads. DAH assigns each kmer to an edge. It then breaks up the kmer into two overlapping (k-1)-mers and assigns the two (k-1)-mers to two nodes. A node has five attributes, shown in Table 1. The two nodes are connected by an edge that corresponds to an overlap between two k-1 mers. In other words, it corresponds to a k-mer.

Table 1: Attributes of the Node object

Attributes	Data type	Comments
label	string	K-1 mer
inDegree	int	The number of incoming edges
outDegree	int	The number of outgoing edges
children	[]*Node	Children nodes
parents	[]*Node	Parental nodes

Input Data
Currently, DAH supports the fasta format only. Below we provide an example fasta file containing three reads. Each read is represented by a single string of characters on a separate line, preceded by a label that begins with a ">" symbol. In this example, the three reads are labeled "read1," "read2," and "read3."

reads.fasta
```
>read1
GATCAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
>read2
TCGATCGAATCGATCGAATCGATCGAATCGATCGA
>read3
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
```

To test our assembler, we created a fasta file containing pseudo reads derived from a real genome. The reference genome we chose is a COVID-19 viral genome (Genebank accession: OP890706, https://www.ncbi.nlm.nih.gov/nuccore/OP890706.1). We randomly selected a position on the reference genome and obtained a fragment of length 100 because Illumina reads can range from around 50 bp to 400 bp in length. We produced 3000 reads to achieve 10X coverage and wrote them into a file named readsWithoutNoise.fasta.

Parameters we used to generate pseudo reads:

Genome length: 29706

Read length: 100

Read counts: 3000

Number of mutations: 300

Number of deletions: 30

To mimic the read reads, we introduced noise by randomly mutating, shuffling and deleting the pseudo reads and produced another fasta file named readsWithNoise.fasta.

Simplify of the Graph

Chain-merging: Once the Bruijn Graph is constructed, it is common for there to be many chains of nodes that are connected to each other. These chains can take up a lot of space but may not contain much useful information about the overall structure of the de Bruijn Graph. This can be a waste of resources, as these chains can consume a large amount of memory and processing power without providing significant benefits. In order to improve the efficiency and effectiveness of the assembly process, it may be necessary to reduce the number of these chains by merging them together. When there are two connected nodes in the de Bruijn graph without a divergence (if the indegree and outdegree of each node are both 1, we merge the nodes together into a larger node. This process, known as chain merging, can help to reduce the size of the de Bruijn graph and make it more manageable while also improving the accuracy and time efficiency of the resulting contigs.

Tip Clipping: A "tip" is a series of unconnected nodes at one end of a de Bruijn Graph, which might be an indicator of mistakes or insufficient information in the original data. While removing tips from the graph may not have a major impact, it may be necessary in order to produce more accurate contigs. In general, excluding tips from the graph can help to improve the reliability of the assembly process. In DAH, a tip will be removed if its length is less than 2k, and there are no minor effect to the de Bruijn Graph after tip clipping.

Find the Eulerian Path

Reconstructing the genome from kmers is reducible to finding a path in the de Bruijn graph that visits every edge exactly once, which is an Eulerian path. Due to noises, it might not be possible to output a single string from the reads. Instead, DAH outputs a list of contigs. A contig is a contiguous length of genomic sequence. To find the eulerian path, DAH first decided the starting nodes of different contigs according to the difference between indegree and outdegree (the outDegree of starting nodes should be greater than the inDegree). It then traversed the de Bruijn graph from the selected start points. Here is the pseudocode for the transversal:

```
procedure EulerianTraverse(Graph dbgraph, Node startnodes, Path eulerianpath)
  currentNode = startnodes
  while currentNode has children
    nextNode = pop a child node of currentNode
    add currentNode to eulerianpath
    currentNode = nextNode
  end while
  if there are more start nodes
    EulerianTraverse (G, next start node, eulerianpath)
  else
    end EulerianTraverse
  end if
end procedure
```

Kmer length selection

The kmer selection strategies can be based on the number of unique kmers. The distinct kmers are kmers that counted only once. The rationale is that if a k-mer only appears once in a genome (unique kmer), it can be used as a distinctive maker to align reads since it could serve the function of checking whether or not a certain part is present, which decreases the ambiguity and error rate (Clavijo, 2022). Therefore, by assessing the counts of unique kmers, we can determine the optimal value of kmer length. However, due to the noise and repeats, the number of unique kmers may be difficult to decide. To address this problem, we utilize distinct kmers, which are kmers that count only once. To clarify the distinction between the two terms, we provide an example below:

Read: *AGTCCTCC*

3-mers: *[AGT, GTC, TCC, CCT, CTC, TCC]*

Distinct 3-mers: *[AGT, GTC, TCC, CCT, CTC]*

Unique 3-mers: *[AGT, GTC, CCT, CTC]*

To explore the relationship between the unique kmers and distinct kmers, we plotted their counts in the same plot with different values of kmer length. According to Figure 3, the counts of unique kmers and distinct kmers are largely overlapped, consistent with a previous study (Clavijo, 2022). Therefore, we choose the kmer length mainly based on the number of distinct kmers to simplify the computations.

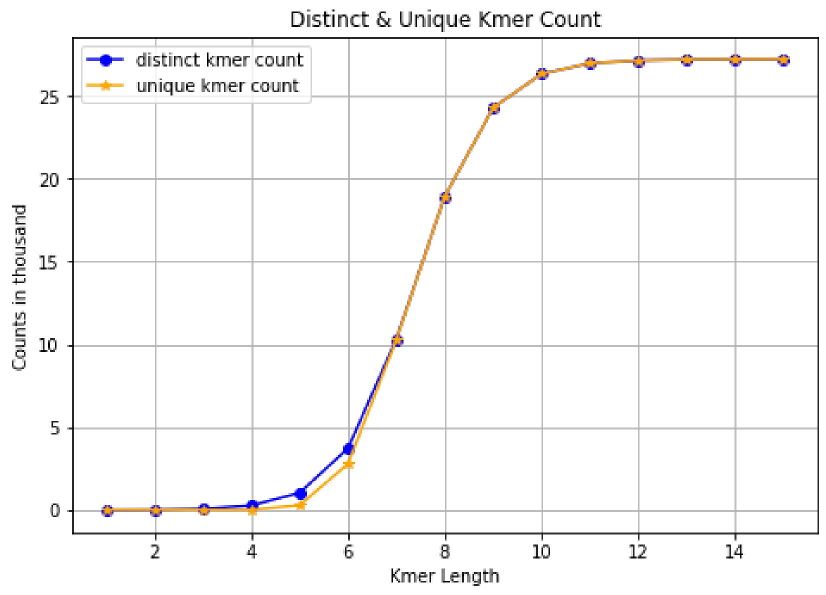

Figure 3: Unique, distinct and total kmer counts of a viral genome with different values of kmer length.

Visualization

To visualize the de Bruijn graph, we utilized a GUI program named "Bandage" (https://github.com/rrwick/Bandage), which allows users to interact with the assembly graphs made by de novo assemblers (Wick et al., 2015). The nodes in the graph, representing contigs, can be labeled with their ID, length, or depth. We can also manipulate the graph, including moving, labeling, and coloring nodes. "Bandage" supports a specific file format: GFA, which is a tab-delimited text format for describing a set of sequences and their overlap. GFA has been a standard format for the exchange of genome assemblies and sequence graphs (Dawson & Durbin, 2019). Thus, we wrote scripts in Golang to read through all the nodes and edges in our graph and output a GFA format file. The resulting GFA file can then be loaded into "Bandage" to generate an interactive assembly graph.

To further explore the distribution of kmer frequencies, we created histograms using the "NewHist" function from the "plotter" GO package (https://pkg.go.dev/gonum.org/v1/plot/plotter). We also produced a coverage count plot using the "NewBarChart" function from the "plotter."

Results

Pseudo Reads Generation

After generating a list of pseudo reads, we produced a coverage count plot to show the coverage of each genomic position. As shown in Figure 4, the coverage is about 10X across the genome.

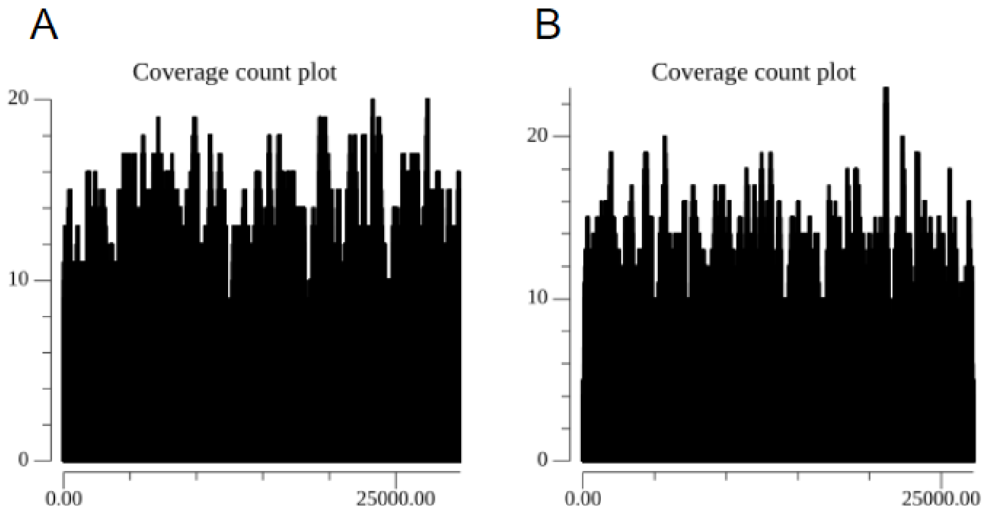

Figure 4: Coverage count plot. (A) Coverage count plot of reads without noise. (B) Coverage count plot of reads with noise.

Kmer Length Selection

DAH allows users to choose from two kmer selection methods, “default” and “range.” For the “default,” DAH first generates a set of prime numbers smaller than the read length to represent different values of k length. It then iterates this set to find the k value with the most distinct kmers. Finally, it adjusts the k length according to the formula (Cha, 2022):

kmer size = 1 + read length - (k-mer coverage * read length)/(Genome coverage)

For the “range” method, DAH requires users to provide two values: min and max. It then iterates over this range to find the optimal value of kmer length, which gives the greatest distinct kmer counts.

The results of the kmer selection are shown below:

Without noise
1)Method: *“range”*
Min: *15*
Max: *25*
Kmer length: *18*

2)Method*: “default”*
Kmer length: *17*

With noise
1)Method: *“range”*
Min: *15*
Max: *30*
Kmer length: *30*

2)Method*: “default”*
Kmer length: *25*

The two approaches have a reasonably high degree of agreement using reads without noise. The ideal kmer length will rise when noise is added. This makes sense, given that noise may make the graph more complicated with more breakpoints and branches, and a longer kmer length can counteract the complexity increase.

Kmer Counts

Using the suggested kmer length, DAH can hash the reads into kmers and produce a kmer frequency distribution plot. According to Figure 5, the majority of kmers have a frequency of 6 to 12.

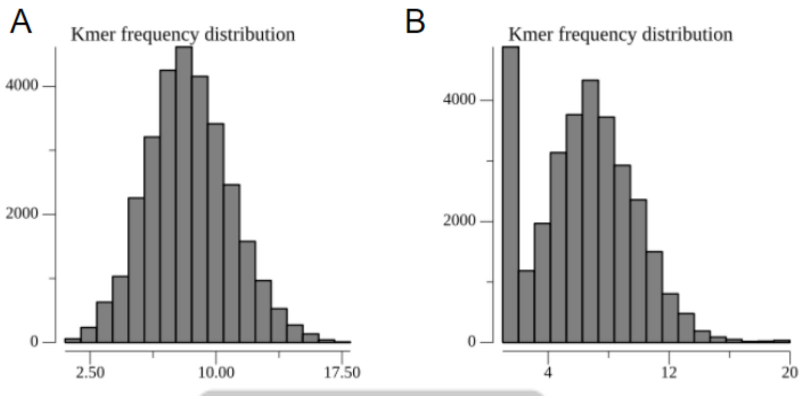

Figure 5: Histograms show the distribution of kmer frequency. (A) Kmer frequency distribution for reads without noise. (B) Kmer frequency distribution for reads with noise.

De Bruijn Graph

DAH constructed a de Bruijn graph based on the kmers and produced a GFA file that can be loaded to “Bandage” to create an assembly graph.

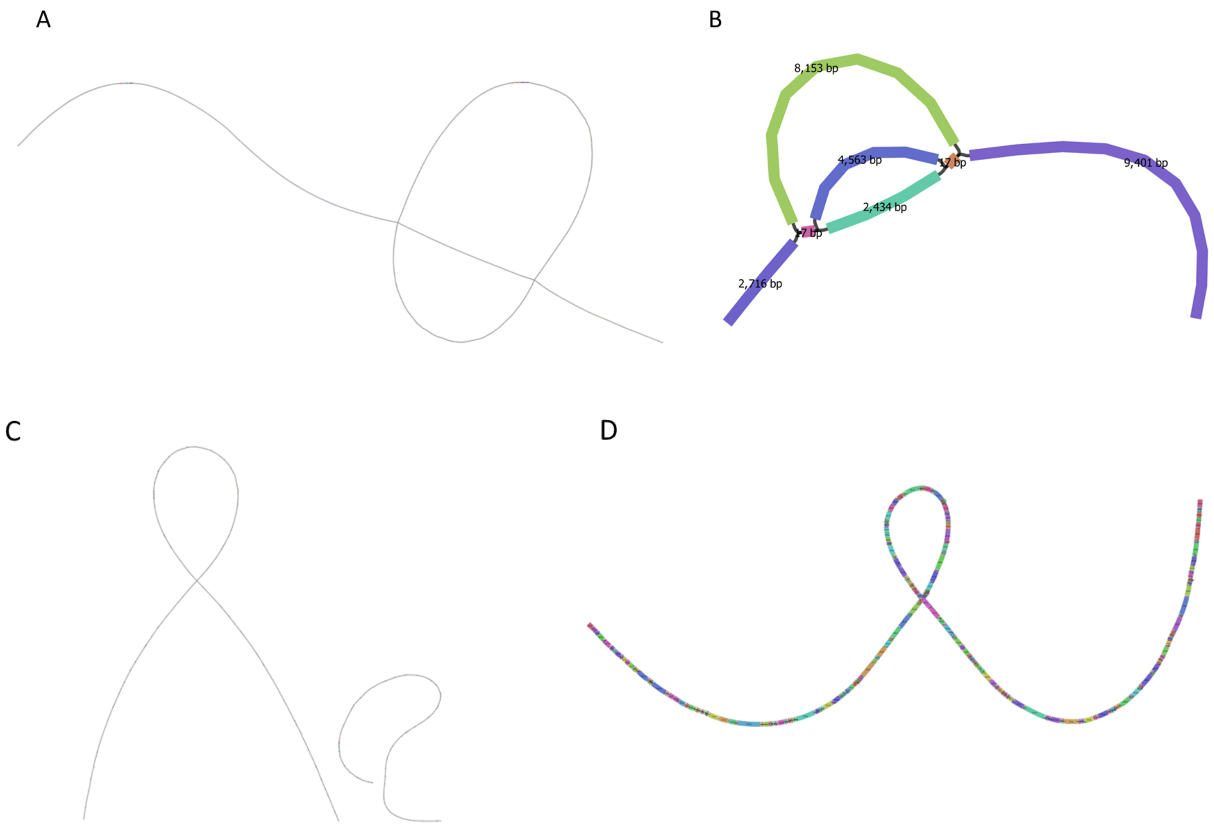

Figure 6: The de Bruijn graph produced by “Bandage” before and after chain-merging. (A) The de Bruijn graph before chain-merging for reads without noise. (B) The de Bruijn graph after chain-merging for reads without noise. (C)The de Bruijn graph before chain-merging for reads with noise. (D)The de Bruijn graph after chain-merging for reads with noise.

Figure 6 demonstrates how successfully chain-merging reduces the number of nodes and edges in the de Bruijn graph. To investigate how chain-merging impacts the assembly process, we compared the amount of time it took to find the eulerian path with and without chain-merging.

Runtime of Finding the Eulerian Path

Without noise:
Without chain-merging: *5.142294ms*
With chain-merging: *6.512μs*

With noise:
Without chain-merging: *12.763693ms*
With chain-merging: *11.891μs*

In both situations, chain-merging can hasten the process of finding the Eulerian path. This technique is particularly effective at speeding up the assembly process when the de Bruijn graph is complex.

Eulerian Path

Finally, DAH outputted a list of contigs and outputted them into a fasta file. Here is an example of a FASTA file containing three contigs:

---

contigs.fasta

```
>contig1 length: 48
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
>contig2 length: 60
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
>contig3 length: 40
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
```

It then calculated the N50 of the assembly results. N50 is defined as the length of the smallest contig for which longer and equal length contigs cover at least 50% of the assembly (Alhakami et al., 2017), a measure commonly used to evaluate an assembly's contiguity. Below we showed the assembly results for reads with or without noise.

Without noise:
Number of Contigs: *3*
N50: *16650*

With noise:
Number of Contigs: *112*
N50: *1369*

When noise was introduced, DAH produced a larger number of contigs and the N50 value decreased significantly. This outcome is expected, as the presence of noise in the data leads to a more complex de Bruijn graph with more branches and breakpoints, resulting in a decrease in contiguity and an increase in the number of contigs.

Discussion

In general, the assembly algorithms have three basic frameworks: overlap layout consensus graph, de Bruijn graph, and string graph (Liao, X., et al., 2019). The de Bruijn graph is further classified into Hamiltonian and Eulerian de Bruijn graphs (Ekblom, R., & Wolf, J. B., 2014). In this project, we built a short read genome assembler named DAH, based on the Eulerian de Bruijn graph, which has been proven to work well in assembling short reads.

DAH is an innovative genome assembler for three reasons:

1. It provides guidance on selecting the optimal k-mer length, a feature that is not offered by many other assemblers.
2. It allows users to visualize the de Bruijn graph during the assembly process.
3. It is the first short read genome assembler written in the Go programming language, as far as we know.

The pipeline of DAH is as follows: determining the optimal kmer length, hashing the reads, constructing the de Bruijn graph, simplifying the graph, and reading off the contigs. Given an input fasta file containing the reads, DAH can output a fasta file containing the contigs, a coverage count plot, a kmer frequency histogram, and a GFA file that can be loaded to “Bandage”. It can also write the N50 and runtime to standard out.

In this paper, we showed the assembly results of pseudo reads produced from a real viral genome, with or without noise. When noise was added, DAH advised using a longer kmer length to counteract the de Bruijn graph’s growing complexity. As anticipated, DAH produced more contigs, an the N50 was much lower.

There is, however, still a great deal of opportunity for improvement. For example, DAH estimates the ideal kmer length using a naive method. Additional algorithms could be added later. Furthermore, more methods for graph simplification and mistake correction might be used to enhance the outcomes of the assembly.

Reference

Cha, S., & Bird, D. M. (2016). Optimizing k-mer size using a variant grid search to enhance de novo genome assembly. Bioinformation, 12(2), 36.

Dawson, E. T., & Durbin, R. (2019). GFAKluge: A C++ library and command line utilities for the Graphical Fragment Assembly formats. Journal of open source software, 4(33).

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., ... & Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. Genome research, 20(2), 265-272.

Clavijo, B. J. (n.d.). *K-Mer counting, part I: Introduction*. BioInfoLogics. Retrieved December 16, 2022, from https://bioinfologics.github.io/post/2018/09/17/k-mer-counting-part-i-introduction/

Wick, R. R., Schultz, M. B., Zobel, J., & Holt, K. E. (2015). Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*, *31*(20), 3350-3352.

Zerbino, D. R., & Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research, 18(5), 821-829.

A comparative evaluation of genome assembly reconciliation tools. https://genomebiology.biomedcentral.com/articles/10.1186/s13059-017-1213-3