Day3 Assignment

Introduction:

A contig is a contiguous sequence assembled from a set of sequence fragments. The goal of contig assembly is to accurately reconstruct the original genomic sequence from fragmented data obtained through sequencing technologies. For example, if we consider a study investigating the microbial diversity in a soil sample, the sample collected undergoes DNA extraction, followed by shotgun sequencing to fragment the DNA into short reads and sequencing using high-throughput sequencing technologies. The short reads obtained from the soil sample are processed using genome assembly algorithms that analyze the overlaps between the reads and assemble them into longer contiguous sequences (contigs) representing genomic fragments from individual microorganisms present in the sample. Contig assembly is typically done as an early step in genome sequencing projects, especially in de novo assembly where no reference genome is available. The resulting contigs further undergo scaffolding and gap filling, to reconstruct the complete genome sequence. Figure 1 [1] gives an overview of this process.

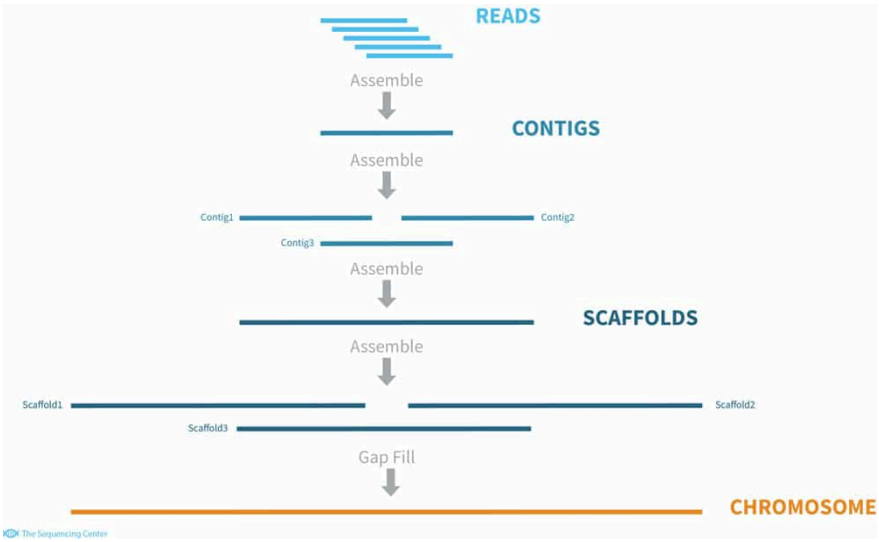


Figure 1: shows an overview of de novo assembly process

Computational Problem:

Given sequencer reads obtained from Next Generation Sequencing platforms and a query sequence, both provided in the FASTA format, the computational task is to identify the longest contig containing the query sequence.

Approach:

The current approach uses the de Bruijn graph and graph traversal via the Eulerian path to assemble the contigs. Figure 2 provides an overview of this process.

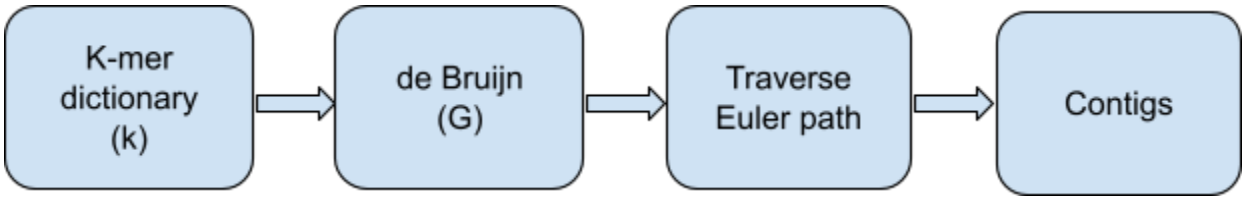


Figure 2: Workflow for contig assembly

The de Bruijn graph serves as a convenient tool for reconstructing the original sequence. Initially, the reads are divided into overlapping k-mers of length “k”. For instance, in the sequence ATAACTGCA, with k=4, the resulting k-mers would be ATAA, TAAC, AACT, ACTG, CTGC, and TGCA. Each k-mer is then split into a (k-1)mer prefix and suffix. For example, the k-mer ATAA would yield ATA as the prefix and TAA as the suffix. This step further involves constructing a graph with nodes representing (k-1) mer prefixes and suffixes from the previous step, and connecting edges between the prefix and suffix pairs derived from the same k-mer, resulting in a de Bruijn graph.

The next step involves graph traversal to obtain an Eulerian path. In a connected graph, an Eulerina path exists if and only if it contains two semi-balanced nodes, with all others being balanced, and each edge is traversed only once. The two semi-balanced nodes are allowed for start and end nodes of this travel. A node is balanced if the number of incoming edges equals the number of outgoing edges.

A start node would have one more outgoing edge than incoming edge, while an end node would have one more incoming edge than outgoing edge. Figure 3 [2] illustrates de Bruijn graph and Eulerian path for contig assembly. Algorithms such as Depth-First search (DFS), can be employed to locate Eulerian paths. DFS starts with a root node and explores the graph in-depth. Upon reaching the end node, it starts backtracking, eventually completing the search.

In this context, two approaches were explored to identify the Eulerian path, which will be discussed below.

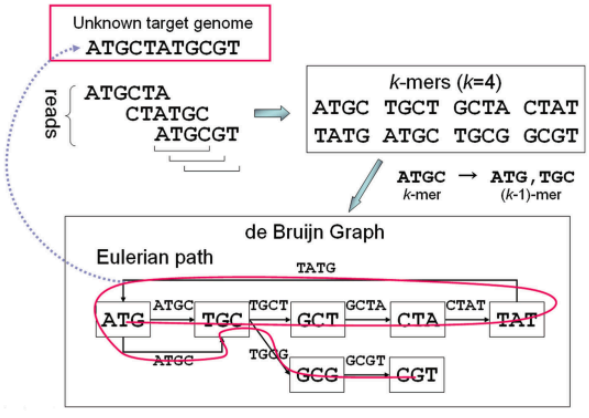


Figure 3: Illustration of de Bruijn graph and Eulerian path

Computational workflow:

The contig assembly tool in python (https://github.com/am794/CPBS7712/) generates a k-mer dictionary and a prefix/suffix dictionary, which are used in constructing the de Bruijn graph. The de Bruijn graph is represented as an adjacency list dictionary, where each key represents a node in the graph, and its corresponding value is a list containing its neighboring nodes. This format effectively captures the relationships between nodes in the graph. Additionally, for each node, the in-degree and out-degree are calculated. Nodes with one more out-degree than in-degree are potential starting does, while nodes with one more in-degree than out-degree are potential end nodes.

Approach 1 for Euler path: This approach identifies a single Euler path for each starting node. Traversal of the graph begins at the start node, and at each step, the neighboring node is removed and added to the path list. This deletion ensures that the path is traversed only once. The final path list contains the Euler path associated with the starting node.

Approach 2 - using DFS: Overcoming the limitations of Approach 1, this identifies all possible Eulerian paths associated with a start node. Similar to the previous approach, traversal begins at the start node and continues until reaching the end node. Upon reaching the end node, backtracking occurs to traverse through the unvisited nodes. Here, a stack dictionary is used, following the Last-in-first-out (LIFO) principle. The stack dictionary maintains the current node as its key and the traversed path as its value. At each step of traversal, we retrieve the key-value pair and pop it to update the neighbor node and the new path. Upon reaching the end, the nodes in the path are concatenated to assemble the contig. Figure 4 illustrates an example of the DFS algorithm [3].

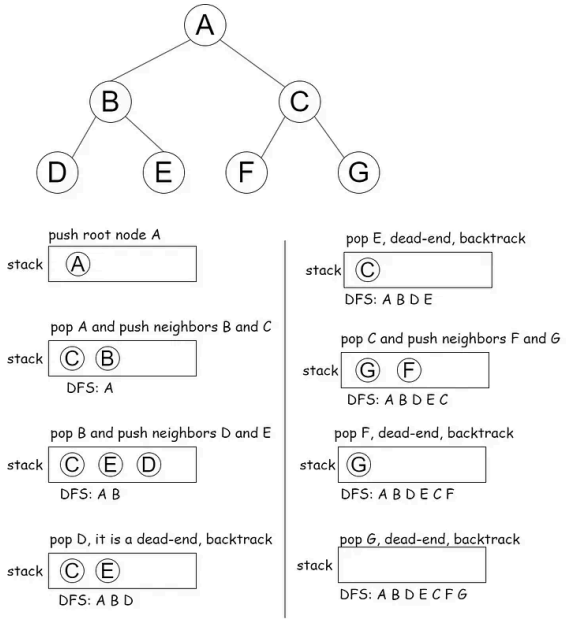


Figure 4: Shows DFS approach that has been adapted for the eulerian path traversal

Once the list of contigs is obtained, the next step involves identifying the longest contig containing the query sequence. This involves aligning the query sequence to the contigs and determining the optimal alignment among them. I am planning to use Hamming distance for this purpose, which quantifies the number of positions where characters differ between two strings, representing the number of mismatches or substitutions. Using this, the longest contig with the smallest Hamming distance would be selected.

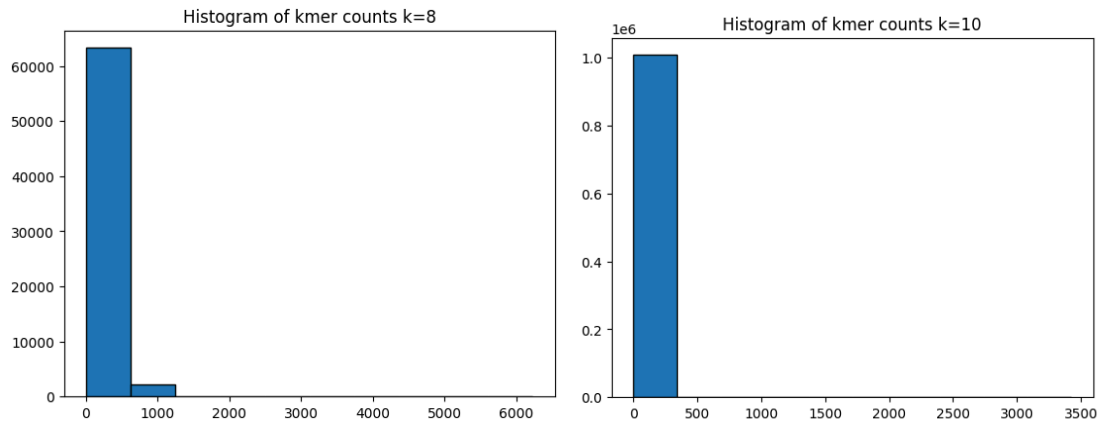
Results and Discussion:

The current input reads file comprises 124,520 reads, with the longest read spanning 346 bases and the shortest read spanning 30 bases. Theoretically, to build the graph, we need to opt for a k-mer length not exceeding 30. For a read of length ‘n’ and a k-mer size ‘k’, the number of k-mers generated is calculated as (n-k+1).

K-mer length:

The choice of K-mer length is an important parameter in the assembly process. Smaller k introduces ambiguities in the graph, particularly between similar regions, resulting in multiple branches, bubbles (repeats and paralogs). Repeated k-mers introduce complexities that can lead to assembly errors and mis-assemblies. Conversely, larger k might avoid ambiguities, but require longer stretches of correct nucleotides for successful assembly. Additionally, low coverage and sequencing errors can create gaps, resulting in a disconnected de Bruijn graph, and potential mis-assemblies.

Figure 5 shows the distribution of k-mers, illustrating a decrease in frequency as k increases. For instance, when k=8, there are approximately 60,000 k-mers with frequencies ranging from 0 to 600. This frequency diminishes as k increases, with k=20 yielding approximately $1X10^7$ k-mers and frequencies ranging from 0 to 50. Subsequently, for k=25, the frequency further declines. Table 1 shows two most frequent and least frequent k-mers across each k-length. For instance, when k=8, the two most frequent k-mers are TTTTTTTT and AAAAAAAA with frequencies 6222 and 5889 respectively. As k increases, the frequencies of the most frequent k-mers decrease.



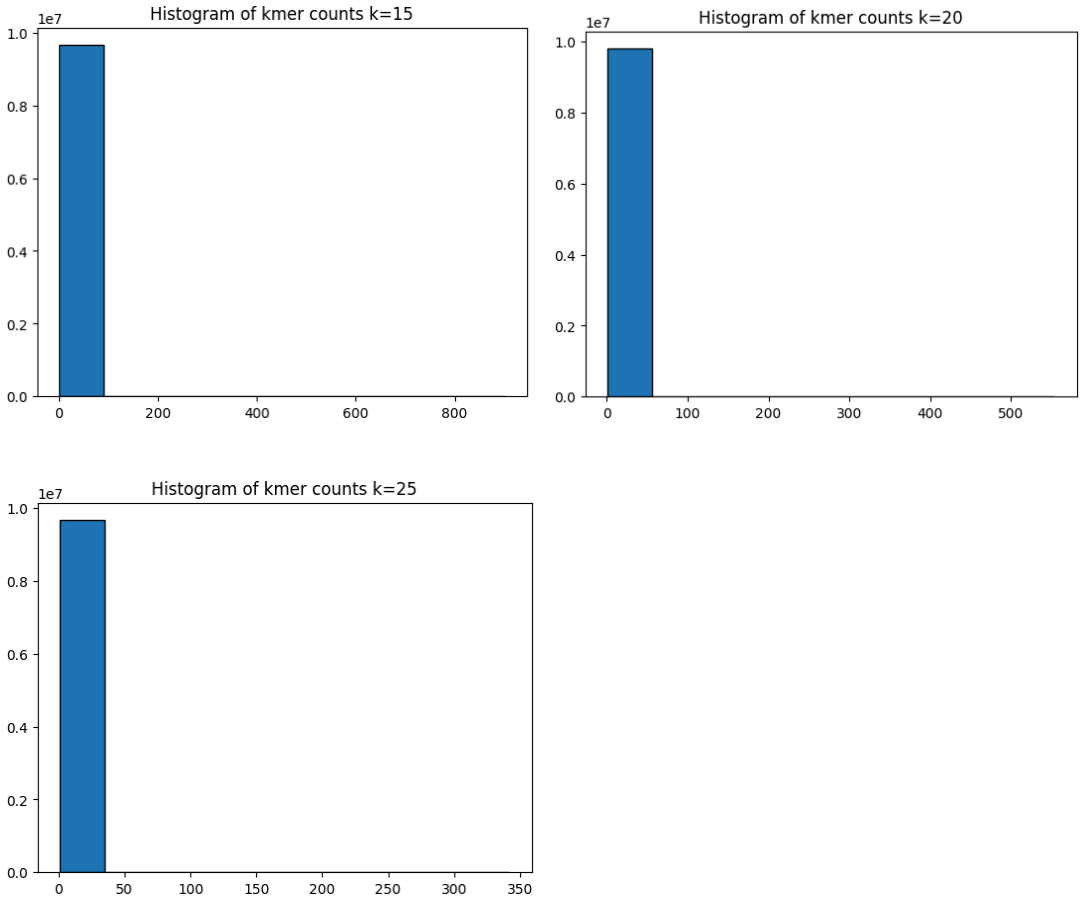


Figure 5: Histograms showing the distribution of k-mers across different k lengths. X-axis is the k-mer frequency at which k-mer is repeated and the y-axis is the frequency of occurrences for each frequency value. The histograms are organized from left to right and top to bottom, corresponding to k=8, k=10, k=15, k=20, and k=25.

K mer length	Most frequent kmer and frequency	Second most frequent kmer and frequency	Least frequent kmer and frequency	Second least frequent kmer and frequency
8	TTTTTTTT: 6222	AAAAAAAA: 5889	TACGCGCG: 2	TTTCGCGA: 4
10	TTTTTTTTTT: 3424	AAAAAAAAAA: 2403	CGGCATCCTA: 1	TATGTAGCGA: 1
15	TTTTTTTTTTTTTTT: 901	ACACACACACACACA: 862	TTCAGGCTCTGGCAT: 1	TCAGGCTCTGGCATG: 1
20	CACACACACACACACACACA: 554	ACACACACACACACACACAC: 534	TTCAGGCTCTGGCATGCATT: 1	TCAGGCTCTGGCATGCATTA: 1
25	CACACACACACACACACACACACAC: 342	ACACACACACACACACACACACA: 342	TTCAGGCTCTGGCATGCATTAGAAA:1	TCAGGCTCTGGCATGCATTAGAAAT: 1

Table 1: shows two most frequent and least frequent k-mers and their frequencies.

de Bruijn graph and Eulerian path:

Figure 6 shows a de Bruijn graph constructed using a k-mer length of 8 and the first two sequencing reads in the FASTA file

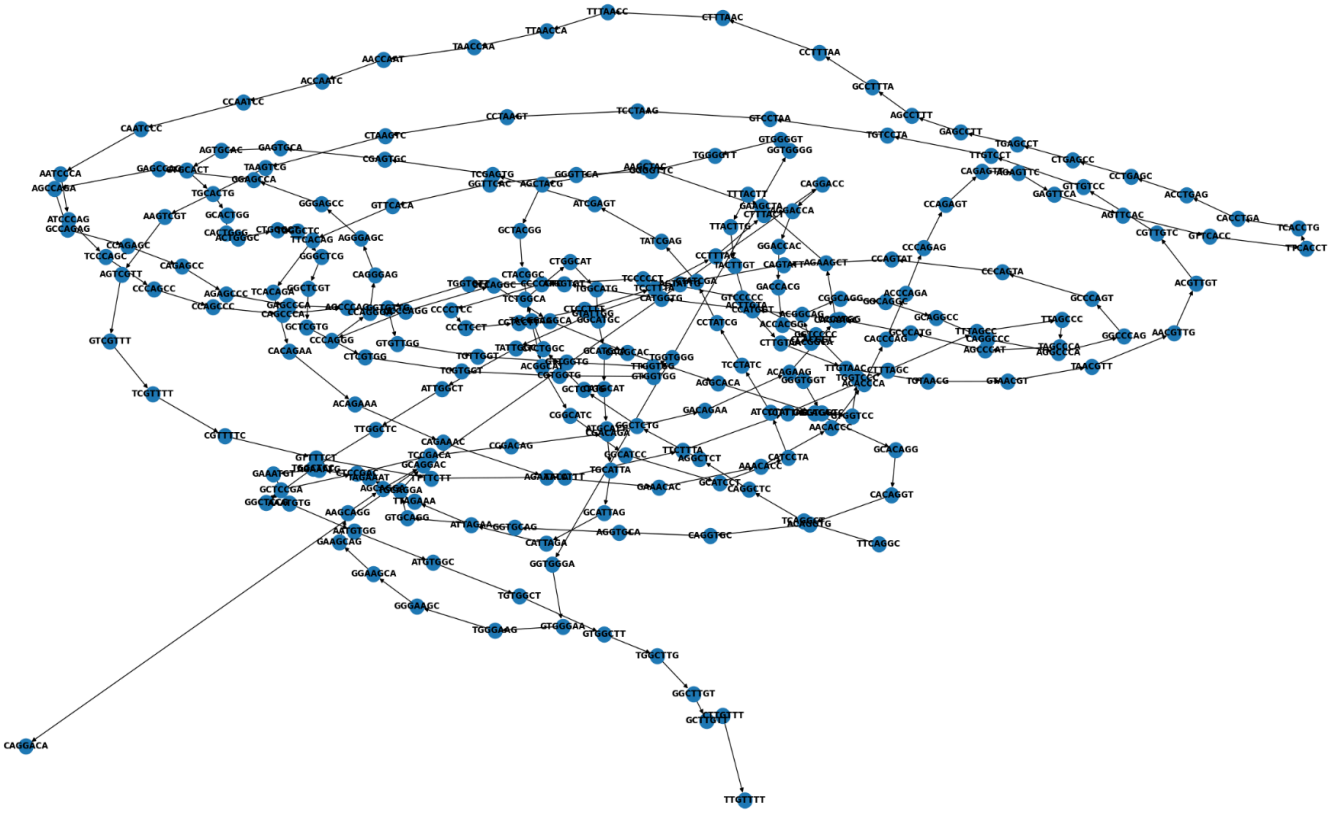


Figure 6: De Bruijn graph for the first two sequencing reads in the FASTA file (k=8)

Challenges:

In the current approach, the length of “k” is not optimized, instead relying on an arbitrary value. Additionally, approach 1 employed does not comprehensively identify all possible Eulerian paths for each starting node, but captures only a single path. Consequently, this overlooks alternative paths that could potentially lead to a more complete reconstruction of the contigs. Furthermore, the current approaches fail to address complex graph structures such as bubbles, branches, and disconnected graphs. These structures can arise due to genomic repeats, sequencing errors, and structural variations. This may result in inaccurate contig assembly.

Additionally, while Hamming distance may serve as a straightforward metric for measuring the sequence dissimilarity, it does not account for insertions and deletions (indels). This can impact the alignment and the resulting output contig containing the query sequence.

Current Status:

At the time of submission, the tool outputs the longest contig identified during contig assembly, while the generation of required outputs is still a work in progress.

Moving forward, my focus would be on identifying the longest contig containing the query sequence. To achieve this, I intend to use Hamming distance, which will help in selecting the longest contig with the smallest Hamming distance from the query sequence.

I haven't conducted unit testing and this will be my focus in the next step.

References:

1. (https://thesequencingcenter.com/knowledge-base/de-novo-assembly/)
2. Namiki T, Hachiya T, Tanaka H, Sakakibara Y. MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads. Nucleic Acids Res. 2012 Nov 1;40(20):e155. doi: 10.1093/nar/gks678. Epub 2012 Jul 19. PMID: 22821567; PMCID: PMC3488206.
3. https://medium.com/geekculture/depth-first-search-dfs-algorithm-with-python-2809866cb358
4. https://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf