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.

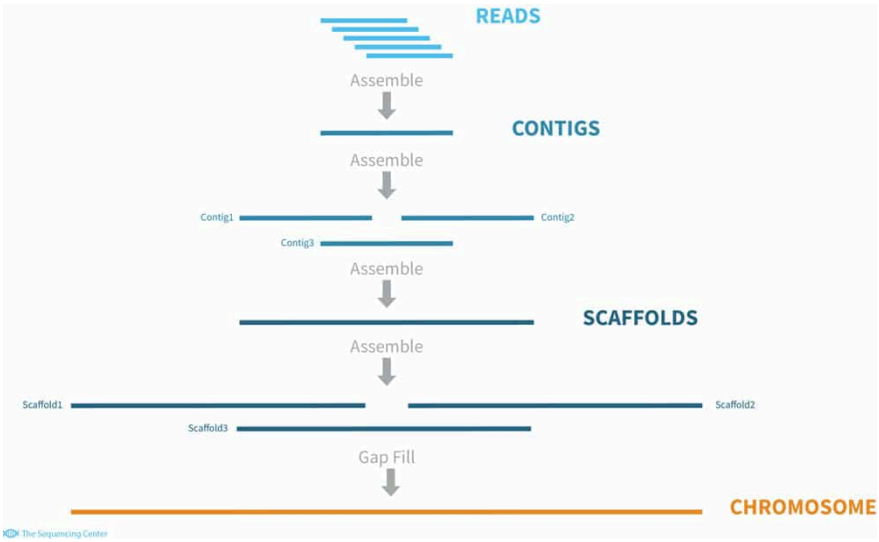


Figure 1: shows an overview of de novo assembly process

Furthermore, the reference-based alignment involves aligning sequences obtained from experiments against known reference sequences. The alignment algorithms search for potential mapping locations in the reference sequence where the query sequences could align. This can be achieved by exact matching or by evaluating sequence similarity using various scoring schemes. The alignment algorithm assigns scores to matches, mismatches, gaps, and other alignment features based on a scoring scheme, facilitating the assessment of alignment quality and the identification of optimal mapping locations [2]. Short-read sequence aligners like Burrows-Wheeler Aligner [3] and Bowtie

[4] (unspliced read aligners), as well as STAR [5] (splice aware aligner), are commonly employed to align short reads produced by Next Generation Sequencing (NGS) technologies to the reference genome. Additionally, Database search tools such as Basic Local Alignment Tool (BLAST) [6], enable the comparison of a query sequence against an extensive database of sequences, for instance, to determine if a novel gene sequence shares similarity with any known sequences in public databases or to infer functional annotations and evolutionary relationships of a gene of interest.

Computational Problem:

Given sequencer reads obtained from NGS platforms and a query sequence, both provided in the FASTA format, the computational task is to identify the longest contiguous sequence, also known as contig, containing the given 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.

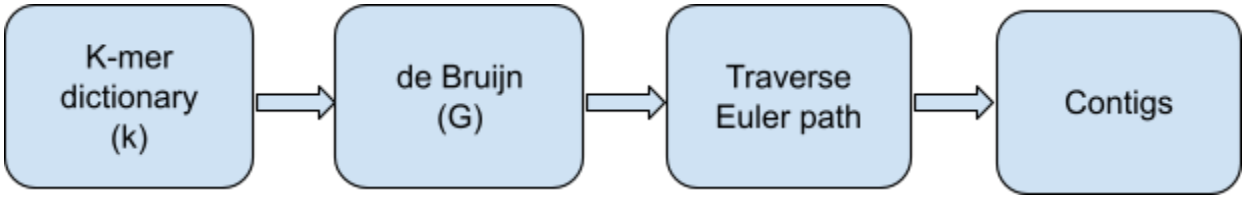


Figure 2: Workflow for contig assembly

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 [7] 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.

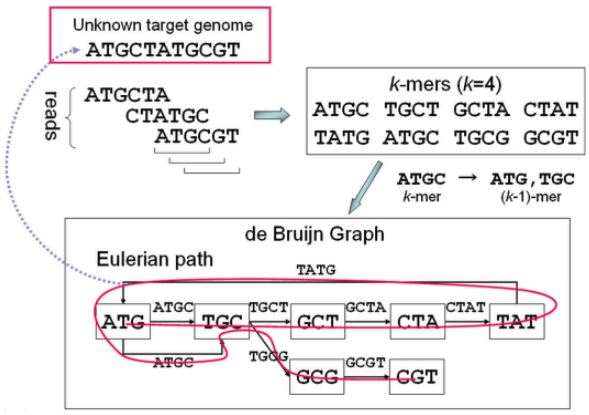


Figure 3: Illustration of de Bruijn graph and Eulerian path

Computational workflow:

a) Contig assembly:

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 [8].

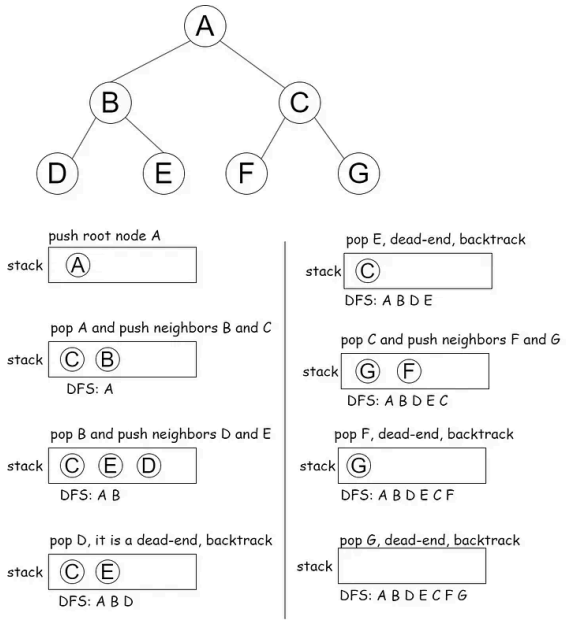


Figure 4: Shows DFS approach that has been adapted for the eulerian path traversal

Once the list of contigs is obtained, the next step involves identifying the longest contig containing the query sequence.

b) Sequence alignment:

To find an optimal alignment between the query sequence and the contigs, I implemented a seed-and-extend-based approach inspired from [10]. Initially, the query sequence is fragmented into non-overlapping seed/k-mers of size “k” (note that this “k” does not necessarily match the k used in contig assembly). Subsequently, a hash index is constructed by recording the occurrences of these seeds within the query sequence. Next, each contig is scanned to identify occurrences of these seeds by querying the hash index database, and whenever a match is found, its position from the database is recorded for the contig. Finally, contigs with consecutive or ascending indices are selected, indicating a potential pre-alignment with the query sequence. Followed by this filtering, a dynamic programming (DP) approach called edit distance is used to calculate the number of mismatches and insertions/deletions as a measure of dissimilarity between the filtered contigs and query sequence [11]. The adjacent seeds are linked together to form a longer chain of seeds by examining the mismatches between the gaps.

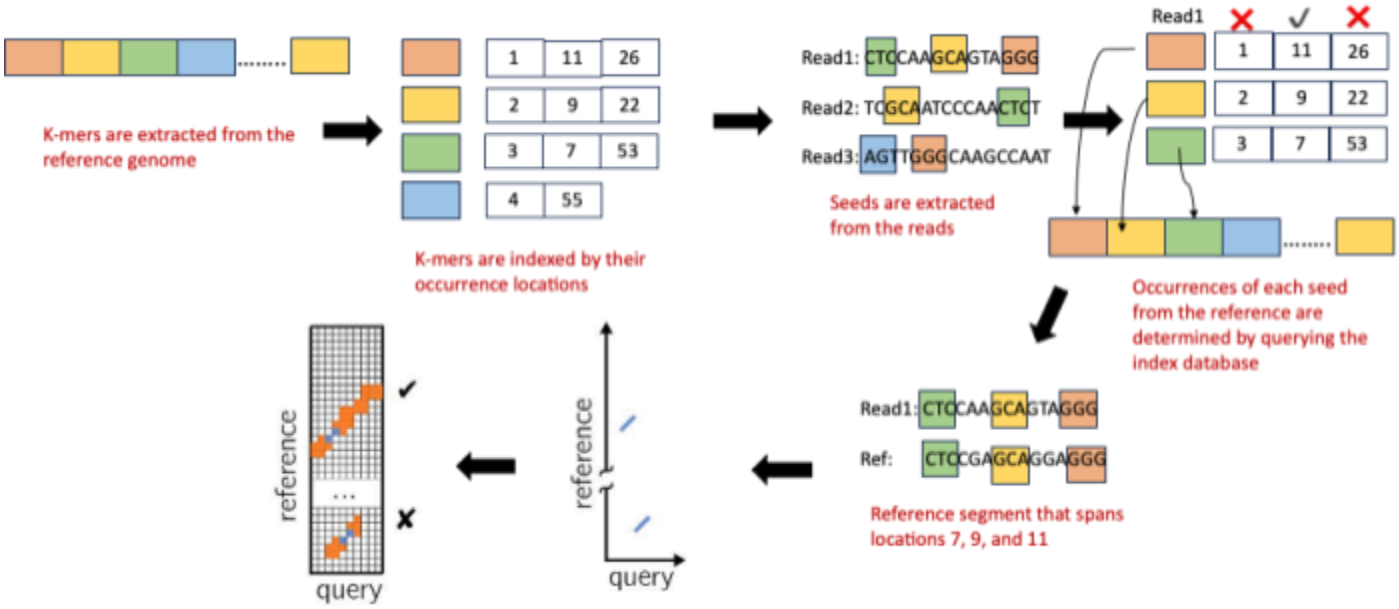


Figure 5: Shows seed and extend approach for alignment (adapted from [10])

Edit distance:

Edit distance refers to the minimum number of editing operations (such as insertions, deletions, and substitutions) required to transform one string into another [11]. In the context of sequence alignment, edit distance could serve as a metric for ranking alignments and identifying the optimal alignment for the longest contig, aiming to minimize the total edit distance.

For instance, let's consider two reads X and Y below. If 'α' represents the prefix of X and 'β' denotes the prefix of Y, the edit distance of a character pair in the 15th position, A and C, can be calculated as:

$$edist(\alpha A, \beta C) = min\{edist(\alpha, \beta) + 1;\ edist(\alpha C, \beta) + 1;\ edist(\alpha, \beta A) + 1\}$$

And here's the representation of reads X and Y:

X: G C G T A T G C G G C T A - A C G C

Y: G C - T A T G C G G C T A T C C G C

I employed edit distance to evaluate the alignment quality between the query sequence and the pre-filtered contigs. For each alignment, three metrics were recorded: a) the edit distance, b) the length of the contig, and c) the edit distance normalized by the contig length. Followed by this, contigs were ranked in ascending order of their normalized edit distance. However, the normalized edit distance alone may not be sufficient to determine the optimal alignment. So, I opted for a combined metric of edit distance and contig length, such that, contig with least edit distance and greater contig length is prioritized.

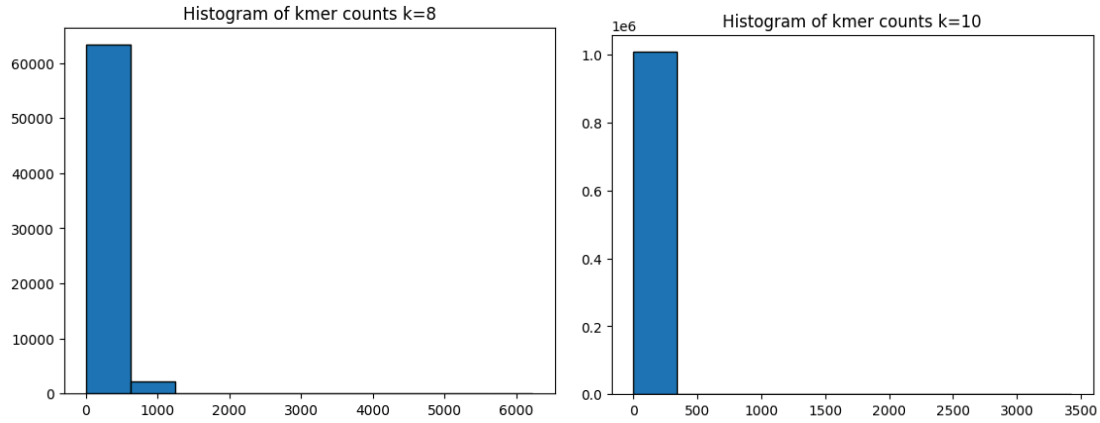
Results and Discussion:

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 6 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.



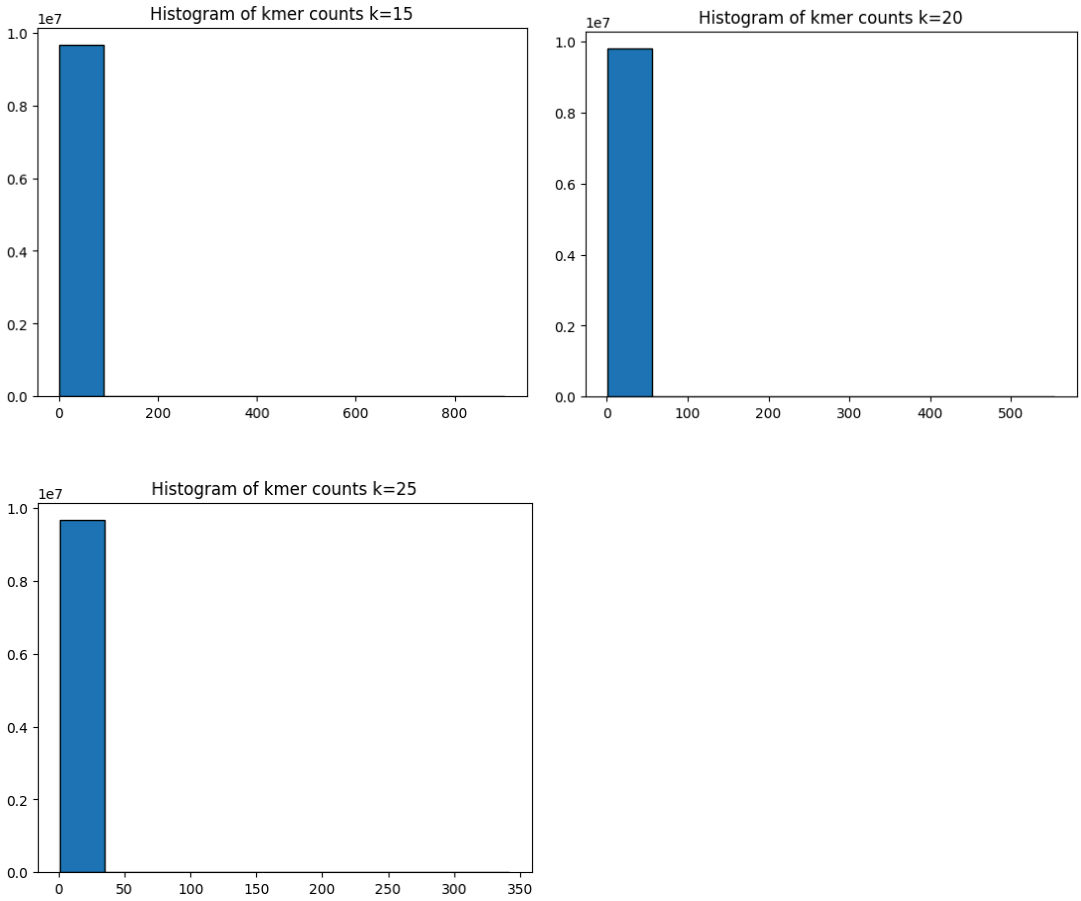


Figure 6: Histograms showing the distribution of k-mers across different k lengths. X-axis is the k-mer frequency at which k-mer is repeated and the y-axis is the frequency of occurrences for each frequency value. The histograms are organized from left to right and top to bottom, corresponding to k=8, 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:

The de Bruijn graph (DBG) and depth first search (DFS) for Eulerian path generates all contigs by selecting start nodes based on their in-degree and out-degree. Each DBG generated will all have two semi-balanced nodes representing the start and end nodes. For k=15, there are 105,639 potential start nodes and 114,163 end nodes.

However, the DBGs in the current implementation do not account for bubble or loop formations, which will also be discussed in the following section. Figure 7 shows a de Bruijn graph constructed using a 8-mer length and the first two sequencing reads from the FASTA file.

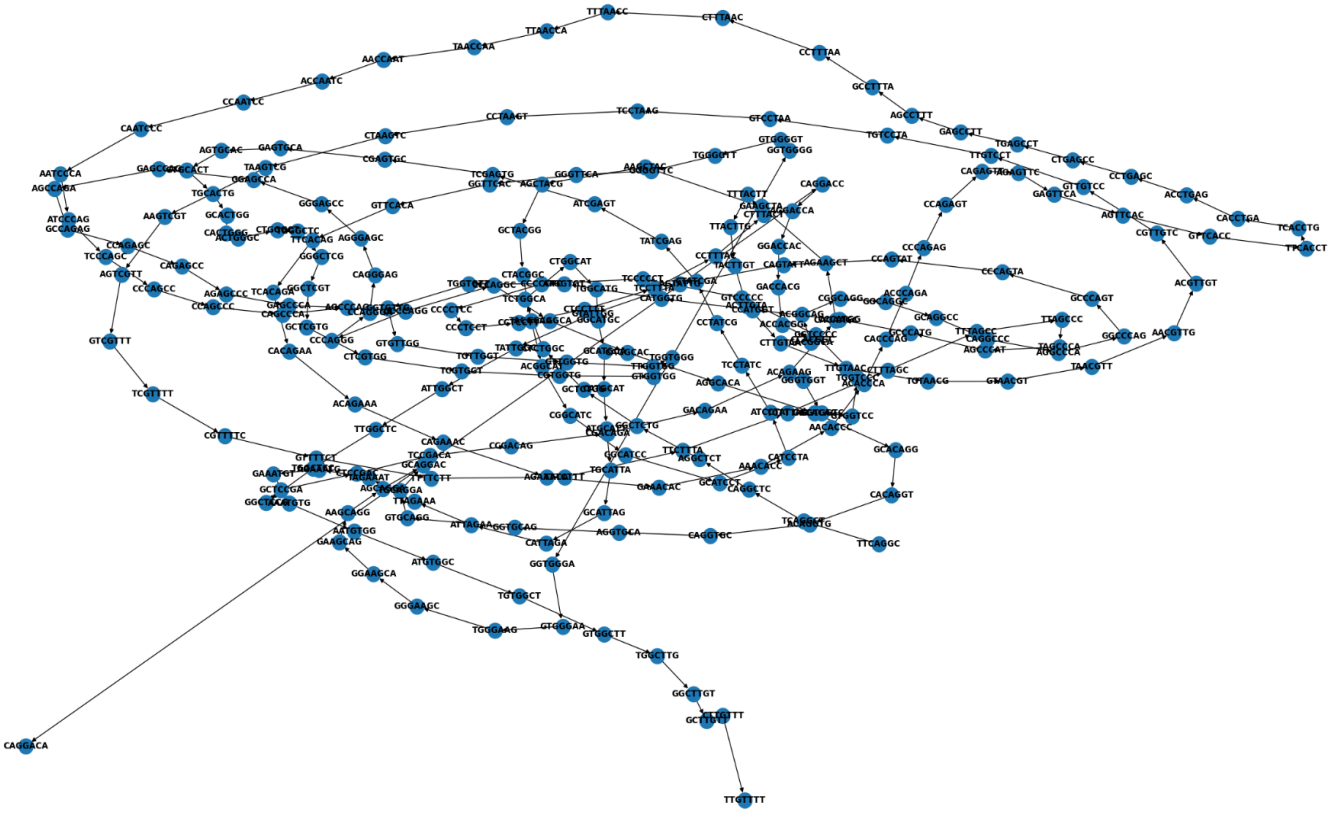


Figure 7: De Bruijn graph for the first two sequencing reads in the FASTA file (k=8)

The primary constraint at present is the runtime of the workflow. Due to the exhaustive nature of the contig generation algorithm, I could not execute the workflow for all 124,520 sequencing reads. So, a subset of the data, comprising the first 1000 reads from the READS.fasta file was utilized.

From this subset, the longest contig containing the query sequence spans 168 bases, notably shorter than the full query sequence length of 649 bases. However, this contig fails to identify overlapping coordinates from the sequence reads, suggesting potential gaps or inconsistencies in the alignment process. The edit distance of this contig is calculated at 82, with a normalized edit distance of 0.488. This further suggests that the

longest contig generated from the first 1000 reads using the current workflow is not an optimal alignment to the query sequence, highlighting the need for refinement and improvement of the workflow.

Challenges and Limitations:

In the current approach, the length of "k" is not optimized, instead relying on an arbitrary selection. This could significantly affect the accuracy and completeness of the contig assembly, particularly when dealing with complex genomic regions such as repeat elements. 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. Neglecting to address these complexities may result in incomplete or erroneous contigs.

The alignment approach relies on seeds to generate hash index, however, the seed length has not been optimized, and this might affect the accuracy and efficiency of the alignment process. The seeds generated from the query sequence are non-overlapping, in contrast to the k-mers used in assembly and the overlapping vs non-overlapping k-mer comparison has not been explored. Moreover, contigs with less than two index overlaps are filtered out, and this may not be an accurate strategy. Additionally, the choice of metric for selecting the optimal alignment is unexplored and the edit distance is not compared with other dynamic programming approaches for pairwise sequence alignment.

The most significant limitation lies in the run time and computational resources required to run this tool. The computational demands pose constraints on scalability and efficiency, especially for larger numbers of reads that are very common to high-throughput data from NGS technologies. To conclude, this workflow needs refinement to address the limitations discussed above and to consider alternative use cases to enhance its utility.

References:

1. (https://thesequencingcenter.com/knowledge-base/de-novo-assembly/)
2. Benjamin, A.M., Nichols, M., Burke, T.W. *et al.* Comparing reference-based RNA-Seq mapping methods for non-human primate data. *BMC Genomics* 15, 570 (2014). https://doi.org/10.1186/1471-2164-15-570
3. Li H. and Durbin R. (2009) Fast and accurate short read alignment with Burrows-Wheeler Transform. Bioinformatics, 25:1754-60. [PMID: 19451168]

4. Langmead, B., Trapnell, C., Pop, M. *et al.* Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* 10, R25 (2009). https://doi.org/10.1186/gb-2009-10-3-r25]
5. Dobin A, Davis CA, Schlesinger F, Drenkow J, Zaleski C, Jha S, Batut P, Chaisson M, Gingeras TR. STAR: ultrafast universal RNA-seq aligner. Bioinformatics. 2013 Jan 1;29(1):15-21. doi: 10.1093/bioinformatics/bts635. Epub 2012 Oct 25. PMID: 23104886; PMCID: PMC3530905.
6. Camacho, C., Coulouris, G., Avagyan, V., Ma, N., Papadopoulos, J., Bealer, K., & Madden, T.L., 2009. BLAST+: architecture and applications. *BMC Bioinformatics*, 10, 421.
7. 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.
8. https://medium.com/geekculture/depth-first-search-dfs-algorithm-with-python-2809866cb358
9. https://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf
10. Alser et al. Genome Biology (2021) 22:249 https://doi.org/10.1186/s13059-021-02443-7
11. https://web.stanford.edu/class/cs124/lec/med.pdf