CS 238 – Assignment 2

Name – Lohith Kumar Bhambore

Student ID – 862393065

Net ID – Lbham003

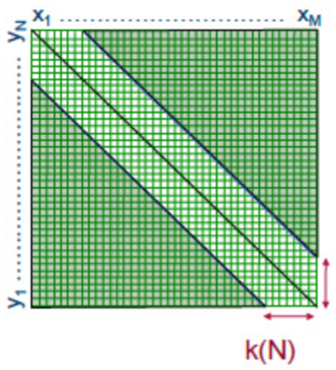
<u>Question 1</u>

Solution

The idea is to use a dp approach only if k is greater than zero. After every insert or delete, decrease the value of k. And once k is zero, all we are left with is just the match or mismatch scores for each element. The dp recurrence when the number of insertions and deletions are bounded by k is given as follows –

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + w(a_i, -) & , k > 0 \\ S_{i,j-1} + w(-, b_j) & , k > 0 \\ S_{i-1,j-1} + w(a_i, b_j) & , k > 0 \end{cases}$$

This leads to a dp table as shown below, we only compute the values which are a distance k away from the diagonal –



Thus, the runtime complexity becomes O((m+n)*k) where m is the length of A, n is the length of B and k is the bound on the number of insertions and deletions.

Question 2

See Colab for code and algorithm – https://colab.research.google.com/drive/1xoUMHcTVFxX-X8cu40CDFnqDyjsoH8QW?usp=sharing

I have also shared it below –

Algorithm

1. Read the shotgun reads from the fasta file.
2. Create a graph representation of the overlaps between reads.
3. Find the Hamiltonian path in the graph
4. Stitch the reads together based on the path to obtain the reconstructed genome sequence.

Code

```python
def parse_fasta(file_path):
    reads = []
    with open(file_path, "r") as file:
        lines = file.readlines()
        for line in lines:
            if line.startswith(">"):
                reads.append("")
            else:
                reads[-1] += line.strip()
    return reads


def build_graph(reads):
    graph = {}
    for i, read in enumerate(reads):
        graph[i] = []
        for j, other_read in enumerate(reads):
            if i != j:
                overlap = find_overlap(read, other_read)
                if overlap:
                    graph[i].append(j)
    return graph


def find_overlap(read1, read2):
    min_overlap = min(len(read1), len(read2)) // 2
    for i in range(min_overlap, 0, -1):
        if read1.endswith(read2[:i]):
            return i
    return 0
```

```python
def find_hamiltonian_path(graph):
    stack = [0]
    path = []

    while stack:
        node = stack[-1]
        if node not in path:
            path.append(node)
        if len(path) == len(graph):
            return path
        found_neighbor = False
        for neighbor in graph[node]:
            if neighbor not in path:
                stack.append(neighbor)
                found_neighbor = True
                break
        if not found_neighbor:
            stack.pop()

    return None


def reconstruct_genome(reads, path):
    genome = reads[path[0]]
    for i in range(1, len(path)):
        overlap = find_overlap(reads[path[i - 1]], reads[path[i]])
        genome += reads[path[i]][overlap:]
    return genome[:1000]  # Take the first 1000 basepairs


# Parse the fasta file and extract the reads
reads = parse_fasta("reads.fasta")

# Build the graph representation
graph = build_graph(reads)

# Find the Hamiltonian path
hamiltonian_path = find_hamiltonian_path(graph)

if hamiltonian_path is not None:
    # Reconstruct the original genome sequence
    genome = reconstruct_genome(reads, hamiltonian_path)
    print("Reconstructed Genome Sequence:")
    print(genome)
    print("Length of Reconstructed Genome Sequence:")
    print(len(genome))
else:
    print("No Hamiltonian path found.")
```

Output –

```
Reconstructed Genome Sequence:
AGCCAATAGCAGATATGCCCATACCGCTGTATTCATAGCTTTCTCTACACGGCCTAAAAGCGGTCGACTGCACG
GTCGACTGCACGCAGTCTTCCGGAGCCAATAGCAGATATGCCCATACTTATCACGTACGTAGATTCGAATACAA
ATTCGACAGATGTGGTTTAATGATTCCGCCTCCTATTACAACAGCCCCGAGGATCCTGCACTGAGTCTGAGGAG
CTGGGGTGCAAATTAGACGGCCTAAAAGCGGTCGACTGCAAGGTAGGGGTACGTACATGTTTCCCATAGGCAGC
GCCTGCCGCTGTTACAACAGCCGACATTGCGACACAATACCAGTTTTTATTGTGTCCATGTACCGCCTAACACT
TATCACGTACGTACATGTTTCGGGAGAGAAAGGGGTGATGTTCTGTTATTAGACCGACGCCCCTAATTGGATCA
ATCAGGGTAGGTCATGGGAGGGGTGATGTTCGAATAAATGGCATATAAGCCCGGATCCGTCCTGTCTGCGACAC
TGACATGGATCCGTCCTGTCTGCGACGTTTCGGGTCGATAAAGCGTTGTCCGACGCCCCTAATTGGATCAATCC
TGATGGTACTCCCCCTTCATTGCGCCCGTTTCCCATGTACCGCCTAGTACAAATTCGACAGATGTGGTTTATTC
GATGGGTAGGTCATGGGAGTAGAGTCGGTGAGGAGCTGGGGTGAGTCTTCCTGATGGTACTCCCCCTGCACCAT
GAACGCGATTGCTAAACATGGATCCGTCCTGTCTGCGACATGGGAGGGGTGATGTTCTGTTATTGGTACTCCCC
CTTCATTGGTCGGTAGAGTCGGTGAGGAGCTGGGGTGAGGATTAGCGCCCGTTTCCCATGTACCGCCTAACACT
CGTGGTGAGCAGGAAATTATTCGCTTTACTAGTCACGTGCTCTAAAATAGCTTTCTCTACACGATCGAGTTGGG
AGGATTAGCGCCCGTTTCCTCCTGGGGTCCGCGATGTCATAGCGAATCCGGAGTGGGTATAGAGGCTCTGTTGT
GACTGTCCGTACTCTCCAATAGCAGATATGCCCATACCCGATTCCGCCTCCTATTACAACAGCCGCTGTATTCA
TAGCTTTCTCTACAAT
Length of Reconstructed Genome Sequence:
1126
```

Question 3

See Colab for code and algorithm – https://colab.research.google.com/drive/1xoUMHcTVFxX-X8cu40CDFnqDyjsoH8QW?usp=sharing

I have also shared it below –

Algorithm

1. Parse the fasta file and extract the shotgun reads.

2. Build a de Bruijn graph using the shotgun reads.

3. Find the Eulerian path in the de Bruijn graph.

4. Concatenate the nodes of the Eulerian path to obtain the reconstructed genome sequence.

The key part is choosing the appropriate value for k in the de Bruijn graph approach can be a challenging task, as it depends on various factors such as read length, sequencing errors, and genome complexity.

Code

```
def parse_fasta_file(file_path):
    sequences = []
    with open(file_path, 'r') as file:
        lines = file.readlines()
        for line in lines:
            line = line.strip()
            if line.startswith('>'):
                sequences.append('')
            else:
                sequences[-1] += line
    return sequences

def build_de_bruijn_graph(sequences, k):
    graph = {}
    for sequence in sequences:
        for i in range(len(sequence) - k + 1):
            kmer = sequence[i:i+k]
            prefix = kmer[:-1]
            suffix = kmer[1:]
            if prefix in graph:
                graph[prefix].append(suffix)
            else:
                graph[prefix] = [suffix]
    return graph

def find_eulerian_path(graph):
    start_node = list(graph.keys())[0]
    current_node = start_node
    path = [current_node]

    while True:
        if current_node not in graph:
            break
        next_node = graph[current_node].pop()
        if len(graph[current_node]) == 0:
            del graph[current_node]
        current_node = next_node
        path.append(current_node)

    return path

def reconstruct_genome(sequences, k):
    graph = build_de_bruijn_graph(sequences, k)
    eulerian_path = find_eulerian_path(graph)
    reconstructed_genome = eulerian_path[0]
    for node in eulerian_path[1:]:
        reconstructed_genome += node[-1]
    return reconstructed_genome[:1000]  # Take the first 1000 basepairs

# Provide the path to the fasta file
fasta_file = 'reads.fasta'

# Parse the fasta file and extract the shotgun reads
sequences = parse_fasta_file(fasta_file)

# Reconstruct the original genome sequence
reconstructed_sequence = reconstruct_genome(sequences, k=5)  # Adjust the value of k as needed
print("Reconstructed Genome Sequence using Eulerian Path approach:")
print(reconstructed_sequence)
print("Length of Reconstructed Genome Sequence:")
print(len(reconstructed_sequence))
```

Output

```
Reconstructed Genome Sequence using Eulerian Path approach:
AGCCCATTACGTGATTCCGTGCTCTACCAGTTTCGGGAGAGATGGTCGGTGAAGTTTCATTGGTACTCCCCTTC
ATCTAAAACCGATCTGTTATTAGACCCCCGGCCCGTACATGTTCTGCCGTCCTGTCTGCCGCCGCTGCGAATGG
ATCCGGAGATGGTTGGGAGTGGAGAAAGGCATATAAGCCCGGAATACCCATAGGCCAATCCGGAGTGGAGGATT
ACTAGTCACGTACGAAAGCCAATCCGGCCAATAAGCGCCTGCGAATCCGCCTCCTAGACATGGCAGCGCAAGGC
TTTACAACAGTGGGAGTGGGTACATGGTTGGGTGATTCGGGAGGGGTAGAGCCGACAAAACAGCTTCAATCCGG
AGAGATATAAAACAGTTATTGTGTCCCATGTTTTATCTGCGACACGTGCTCTAAATGGCGCCCGTTTCCGGAGT
GGAGTGATGTATAAGCCCGTGCTCTAAATGGTTATCTGCGAATCCGGAGATTAGCGTTTCGAATAGCAGATAAA
GGTAGGTCGGTGAATCCGGAGTTAATGGGGTATTATGCCGTCCTGTACGTGCTCTACACAAGGTACTAGTCCGG
CCCATTGGTCACGTACCGATATGATTTGCAAGGCATATAGAGATGTTTCTCTCGTTGGTTATCTGCGACATTGG
GTAGGGGTATAGCTTTACTAGTCATTGCGACACTGAATAAAAACCGCCCGTAGAGGCCCGGAGCTTCAGCTGCG
AATGGTCGGGTCGATCCGTTTCGGGTCGGTGCAAGGCATATAGACATAGGATCATGTGGTTTCGGTGATCGATT
AGAGGCTTTACTCCCCTTCAGCTTTACTAGTCACGTGAGCTGGGAGTGGGTGATGGCAGAGGATCTACCAGTCA
TAGACAACAGCCGCCGACGTGTCTGTCCCATTAGCGCCTATTCATTGCGACATAGGAGGCGACACAAATTACGA
CATTGCAAAAACCGACACTGGATTATCATCGACATTCGATGGTTAATGTTCCCATAAGCCGATCACGATGTACT
CTCTAAGTCGGTGAGGGGTATAAGTGTCTAACAGCTGGATCGATCTGCCCAGCCCATGGGAGGATTAGAGTCCG
TACTAGACCCCCTTCATGTTCTGTCTTCATCGAATCCGGCAGAGTTTAATGAGGATTTGCGACGTGCTCCTCTA
CAACACAGATGAAGTGTTGTCCGGC
Length of Reconstructed Genome Sequence:
1209
```

The performance of the Eulerian path approach and the Hamiltonian path approach can differ significantly depending on the characteristics of the problem.

In the Eulerian path approach, the goal is to find a path that visits every edge in the graph exactly once. The complexity of finding an Eulerian path in a graph is typically O(V + E), where V is the number of vertices and E is the number of edges in the graph. In the context of genome assembly, the graph represents the overlaps between the shotgun reads. If the de Bruijn graph constructed from the shotgun reads has a large number of edges, the Eulerian path approach can be computationally efficient. However, it is important to note that the Eulerian path approach may lead to the reconstruction of repetitive regions in the genome, which can introduce challenges in resolving ambiguities.

Another significant difference between the 2 approaches is choosing the right k-mer value in the eulerian path approach. I had to try multiple values of k to finally observe a trend in the size of the reconstructed genome and get it to approach 1000. This is not a problem we face when using Hamiltonian path approach.

On the other hand, the Hamiltonian path approach aims to find a path that visits every vertex in the graph exactly once. Finding a Hamiltonian path is a well-known NP-complete problem, which means there is no known polynomial-time algorithm to solve it in the general case. The time complexity of finding a Hamiltonian path in a graph can be exponential, particularly as the graph size increases. In the context of genome assembly, constructing a de Bruijn graph with a large number of shotgun reads can result in a complex graph structure, making it computationally challenging to find a Hamiltonian path.

Given the problem size of reconstructing a 1000-basepair genome from 1200 shotgun reads, the performance difference between the Eulerian path approach and the Hamiltonian path approach may not be significant.

The Eulerian path approach is generally more efficient and can handle larger graphs compared to the Hamiltonian path approach. However, it's always a good practice to consider the specific characteristics of the data, such as read coverage and graph complexity, to choose the most suitable approach for genome reconstruction.