DNA Sequence Reconstruction from CSV Files

Author: [Mohamadreza Hooshmandzadeh]
Studentnumber: [2366713]

Overview

In this project, we process raw CSV files containing DNA sequences and, if a valid DNA sequence can be constructed, we save it in a text file. The process involves several steps, including reading the file, cleaning the data, generating sequences, constructing a graph, and finding an Eulerian path if it exists. Below are the detailed steps and examples for each stage:

1. Reading the File

We read the given CSV file and extract the data, file name, and K factor.

Input file: `dna_sequences.csv`

Output:

- DataFrame containing the DNA segments.
- `k` : An integer representing the length of the K-mers. `filename`
- : The base name of the file without the extension.

2. Cleaning the Data

The data is cleaned based on several scenarios, including:

- Omitting duplicate records.
- Removing segments with missing positions.
- Filtering out segments with incorrect data (more or less than a single nucleotide). Eliminating
- segments with redundant positions containing different nucleotides.

3. Generating Sequences

The cleaned DataFrame is transformed into a JSON structure where each segment is represented by its complete DNA sequence.

Example:

Input:

SegmentNr	Position	A	C	G	T
1	1	1	0	0	0
1	2	0	0	0	1
2	1	1	0	0	0
2	2	0	1	0	0

Output:

```
[
  {"SegmentNr": 1, "segment": "AT"},
  {"SegmentNr": 2, "segment": "AC"}
]
```

4. Constructing the De Bruijn Graph

A De Bruijn graph is created using the segments and the given K factor to generate K-mers.

5. Plotting the graph:

The De Bruijn graph is plotted and saved as a PNG file.

6. Checking for a valid Eulerian path

The graph is checked to see if it contains a valid Eulerian path. Output would be a True or flase.

7. Finding the Eulerian Path and Constructing the DNA Sequence

If a valid Eulerian path is found, the path is printed, and the complete DNA sequence is generated.
Example: input:

```
[
  {"SegmentNr": 1, "segment": "ATTACTCG"}
]
```

K-mers: ATTA, TTAC, TACT, ACTC, CTCG
Edges: (ATT, TTA), (TTA, TAC), (TAC, ACT), (ACT, CTC), (CTC, TCG)
Path: [('ATT', 'TTA'), ('TTA', 'TAC'), ('TAC', 'ACT'), ('ACT', 'CTC'), ('CTC', 'TCG')]
Sequence: ATTACTCG

8. Saving the DNA sequence

The valid DNA sequence is saved in a text file.

Implemetation Details

---

1. `read_csv(name)`

Description:
Reads a CSV file and converts it to a Pandas DataFrame.

Possible Approaches:

- Pandas read_csv: Using the pd.read_csv function to read the CSV file directly into a DataFrame.
- Manual Parsing: Reading the file line by line and manually parsing the content into a DataFrame.

Chosen Approach:

Pandas read_csv

Rationale:

- Efficiency: pd.read_csv is highly optimized for reading CSV files.
- Simplicity: Using pd.read_csv simplifies the code and reduces potential errors associated with manual parsing.

2. `clean_data(data_frame)`

Description:
Cleans the DataFrame by removing duplicates, handling missing positions, and filtering out incorrect data.

Possible Approaches:

- Sequential Cleaning: Applying cleaning steps in sequence, with intermediate DataFrame modifications.
- Batch Cleaning: Combining all cleaning criteria into a single filtering operation.

Chosen Approach: Sequential

Cleaning

Rationale:

- Clarity: Each cleaning step is clearly defined and can be independently verified.
- Flexibility: Easier to modify individual cleaning steps without affecting the entire cleaning process.

3. `generate_sequences(df)`

Description:

Generates JSON objects representing DNA segments from the cleaned DataFrame.

Possible Approaches:

- Lambda Functions: Using lambda functions and groupby to concatenate nucleotide sequences. Iterative
- Concatenation: Iterating through rows and manually concatenating sequences.

Chosen Approach: Lambda

Functions

Rationale:

- Conciseness: Lambda functions provide a concise way to perform group-wise operations.
- Performance: Leveraging groupby and lambda functions is generally faster and more efficient.

4. `construct_graph(df, k)`

Description:

Constructs a De Bruijn graph from the JSON objects using the given K factor.

Possible Approaches:

- NetworkX MultiDiGraph: Using NetworkX's MultiDiGraph to handle multiple edges between nodes.
- Adjacency List: Manually maintaining an adjacency list for the graph.

Chosen Approach: NetworkX

MultiDiGraph Rationale:

- Convenience: NetworkX provides built-in functions for graph manipulation and visualization. Robustness:
- MultiDiGraph easily handles multiple edges, simplifying graph construction.

5. `orthogonal_layout(G)`

Description:

Creates a layout for the graph nodes.

Possible Approaches:

- Predefined Layout: Using a fixed grid layout based on node count.
- Personalized Layout: Dynamically adjusting node positions based on graph structure.

Chosen Approach: Personalized

Layout

Rationale:

- Simplicity: A fixed layout is simpler to implement and understand.
- Sufficiency: For this project's scope, considering the number of nodes and recurring edges between nodes a customized layout in orthogonal shape is looking better.

6. `plot_graph(G, filename)`

Description:

Plots and saves the De Bruijn graph as a PNG file.

Possible Approaches:

- NetworkX Drawing Functions: Using NetworkX's drawing functions for visualization. Custom
- Drawing: Manually plotting the graph using Matplotlib.

Chosen Approach:

NetworkX Drawing Functions

Rationale:

- Ease of Use: NetworkX integrates well with Matplotlib for graph visualization. Effectiveness:
- Provides sufficient customization for the project's needs.

7. `is_connect(graph)`

Description:

Checks if the graph is connected.

Possible Approaches:

- Depth-First Search (DFS): Using DFS to check connectivity by traversing and counting reachable nodes with number of existing nodes in the graph.
- `nx.is_weakly_connected` : Using predefined networkx algorithm.

Chosen Approach:

DFS

Rationale:

Considering limitations and terms of the project we can not use prepared algorithms.

8. `is_valid_graph(graph)`

Description:

Checks if the graph is a valid Eulerian graph.

Possible Approaches:

- Degree Condition Checks: Checking in-degrees and out-degrees of nodes and their connectivities. Networkx
- predefined algorithms.

Chosen Approach:

Degree Condition Checks

Rationale:

Considering limitations and terms of the project we can not use prepared algorithms.

9. `Eulerian_path_gen(graph)`

Description:

Generates an Eulerian path from the graph.

Possible Approaches:

- Hierholzer's Algorithm: Using Hierholzer's algorithm for finding Eulerian paths and by customizing a DFS and traversing the graph we can find the Eulerian path. `nx.eulerian_circuit`
-

Chosen Approach: Hierholzer's

Algorithm

Rationale:
Considering limitations and terms of the project we can not use prepared algorithms.

10. `construct_dna_sequence(graph)`

Description:

Constructs the DNA sequence from the Eulerian path.

Possible Approaches:

- Concatenation: Concatenating nucleotide sequences along the path.
- Recursive Assembly: Recursively assembling the sequence from path segments.

Chosen Approach:

Concatenation

Rationale:

Simplicity: Direct concatenation is simple and efficient.

11. `main()`

Description:
Main function to execute the entire pipeline.

Possible Approaches:

- Modular Execution: Breaking down tasks into modular functions.
- Monolithic Execution: Implementing all tasks in a single function.

Chosen Approach: Modular

Execution

Rationale:

- Maintainability: Modular functions enhance readability and maintainability. Debugging
- Ease: Easier to debug and test individual components.

Evaluation

I have implemented all of the compulsory questions perfectly as required. All of the test cases in test files implemented and tested as required as weel. Furthemore I have tested my code manually and run it from terminal with all of the csv files and outputs were as requested in the project. I did not implement any bonus task. I also implement my code pep8 friendly and with function documentation and neccessary comments in code.

Example Run

Although I have implemented and tested all of the csv files but considering abstraction in this report I will explain only outputs and steps for DNA_3_3 :

1. Read file and save to a data frame (DNA_3_3)

- input: `DNA_3_3.csv`
- output:

```
k = 3
```

```
filename = DNA_3_3
```

data frame as below:

SegmentNr	Position	A	C	G	T
1	1	0	0	1	0
1	2	0	0	1	0
1	3	0	0	1	0
1	4	0	0	0	1
2	1	0	0	0	1
2	2	0	0	0	1
2	3	0	0	1	0
2	4	0	0	1	0
3	1	0	0	1	0
3	2	0	0	0	1
3	3	0	0	0	1
3	4	0	0	0	1
4	1	0	0	1	0
4	2	1	0	0	1
4	3	0	0	0	1
4	4	0	0	0	1

2. clean data

-
- input: last step dataframe output: cleaned dataframe

SegmentNr	Position	A	C	G	T
1	1	0	0	1	0
1	2	0	0	1	0
1	3	0	0	1	0
1	4	0	0	0	1
2	1	0	0	0	1
2	2	0	0	0	1
2	3	0	0	1	0
2	4	0	0	1	0
3	1	0	0	1	0
3	2	0	0	0	1
3	3	0	0	0	1
3	4	0	0	0	1

3. Generate sequence

- input: cleaned dataframe from step 2
- output: json format of each sequence

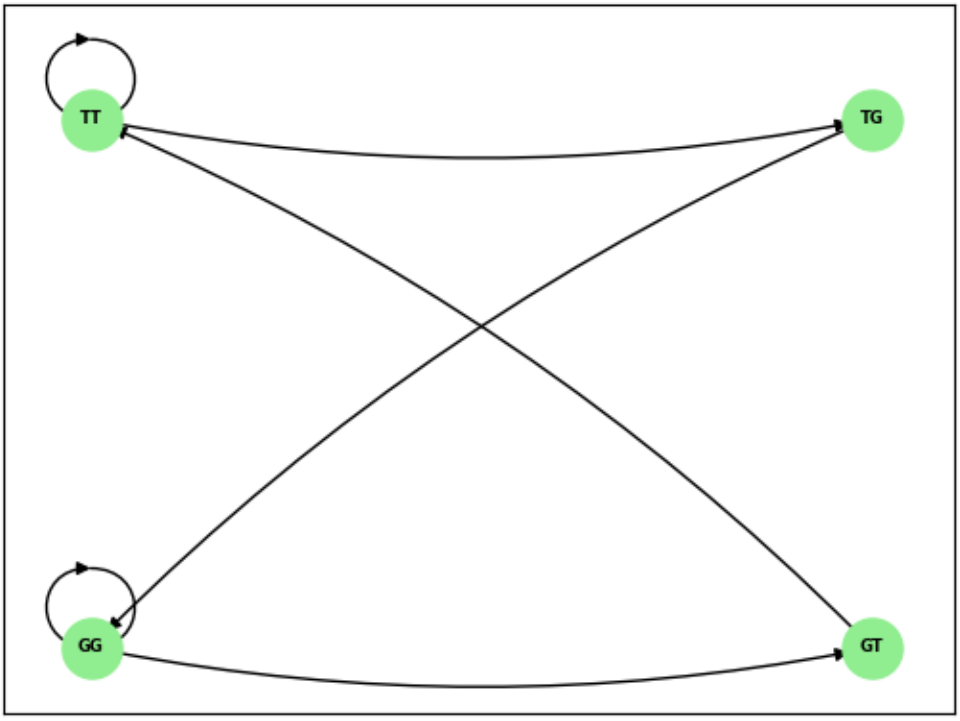
```
[{'SegmentNr': 1, 'segment': 'GGGT'},
 {'SegmentNr': 2, 'segment': 'TTGG'},
 {'SegmentNr': 3, 'segment': 'GTTT'}]
```

4. construct De Bruijn graph

- input: json object from last step, k from step 1
- output: a Multi directed graoh object

5. plot the graph

- input: graph object from last step
- output: saved png file



6. Check validity of graph (existance of Eulerian path)

- input: graph object from step 4
- output: `True` if graph has an Eulerian path, `False` otherwise

7. Finding the Eulerian Path and Constructing the DNA Sequence

- input: graph object from step 4
- if output of step 6 would be `True` :
  - output: print eulerian path on terminal output as below
    [('GG', 'GG'), ('GG', 'GT'), ('GT', 'TT'), ('TT', 'TT'), ('TT', 'TG'), ('TG', 'GG')]
  - save the DNA sequence constructed from above path in a txt file named DNA_3_3 as below: "GGGTTTGG"
- if output of step 6 would be False, save `The DNA sequence can not be constructed!` in the txt file named DNA_3_3

Challenges

Most challenging part for me was finding the Eulerian path using a DFS, I changed my code several times and test and compare its output every time with nx prepared algorithm for eulerian path generation. Furthermore, there was another challenging part in plotting graph to show the edegs perfectly since if there are more than 1 edge between 2 nodes, its tricky to show them separatly, so I used an option in nx.draw called `connectionstyle` and using a dummy index generator pass different radios to each edge and finally was successful to discreminate between them.

Tests Description

Test cases

Cleaning test based on missing value conditions in the project description

- Remove the redundant records
- Remove perfectly duplicate segment even if the segment number is different Remove
- records wich has both conditions above at the same time
  Geberate sequence test
- Generate json output including each segment sequence for 2 different segments consisting 2 neuclodes.
- Generate json output including each segment sequence for 2 different segments consisting neuclodes with different lengths.
- Generate json output including each segment sequence for 3 different segments consisting neuclodes with different lengths.
  test_construct_graph
  Constructing De Bruijn graph and checking its edges with what is expected. in these cases I tested 3 graphs with different number of segments and 3 and 5 mers as k factor and checked the edges with what is expected. test_is_valid_graph testing different graphs existance an eulerian path also there is a case which does not have a eulerian path. and they should be behave as we expected. test_construct_dna_sequence
  testing construction of a DNA from a given list of pathes. so the expected values are generated based on eulerian cycle algoritm. I checked cases with self edges on a given node, multiple edges between 2 nodes and longer pathes to check any possible bug for instance if we dont test multiple nodes it may be the case that there is an error when dfs wants to choose a path between multipe outgoing edges from a single node.