Advanced Programming in Python
Project: DNA Sequence Assembly

Due date is 27/05/2022 at 23:59

The assignment is strictly individual. Plagiarism and cooperation will be thoroughly checked by hand, and by specialised software. Not only copying other peoples code, but also giving your source code to other students will be considered as fraud. Using solutions from the internet, even very small portions are considered as plagiarism.

When we detect fraud, conform with the exam regulations (OER), we will inform the exam committee, which will start a fraud process (see the OER). Unfortunately, every academic year students get caught; we hope that this will not be the case this year.

Objective & General Information

A nucleotide is the basic building block of DNA sequences. The four nucleobases that can appear in a DNA sequence are `A`,`C`,`G` and `T`. Thus, a DNA sequence is a long chain (string) of nucleotides such as (the following sequence will be referred to it as `SQ1`)

```
TTAATTACTCACTGGCTAATTACTCACTGGGTCACTACGCACTG
```

In order to construct a DNA sequence of a given DNA, typically the sequence is multiplied and segmented into different lengths. These smaller segments of the DNA undergo a chemical process in order to know the order of nucleotides in each segment. The following sequences are segments of `SQ1`:

```
TTAATTA             ATTACTC             ACTCAC             TCACTGGCTAA
CTAATTACTCACTGG         CTGGGT          GGGTCACT          CACTACGCACTG
```

These DNA sequences of the smaller segments typically overlap. Thus, they have to be aligned or connected together in order to know the correct sequence. For example, the correct alignment of the previous set of sequences relative to the original DNA sequence is as follows

```
TTAATTA_____________________________________
___ATTACTC__________________________________
______ACTCAC________________________________
________TCACTGGCTAA_________________________
_______________CTAATTACTCACTGG______________
__________________________CTGGGT____________
____________________________GGGTCACT________
________________________________CACTACGCACTG
TTAATTACTCACTGGCTAATTACTCACTGGGTCACTACGCACTG
```

Without a reference DNA sequence, there is no algorithm that will get the correct sequence for all the possible cases of segments. However, we are going to add some assumptions under which we guarantee to find the correct DNA sequence from the given segments.

Main Objective In this project, you will implement a program that given an overlapping sequences of small DNA segments it assembles these segments and returns the whole DNA sequence.

De Bruijn Graph

One possible way to generate the original DNA sequence from the sequences of the smaller segments is by constructing the de Bruijn graph of these segments. Thus, constructing the original DNA sequence using the de Bruijn graph without a reference DNA sequence is one type of de novo sequence assemblers.

In our case, the de Bruijn graph is constructed relative to a given number `k`. This number determines the length of the subsequences that we will consider. Thus, the given DNA segments will be divided into smaller sequences of length `k` if they are longer than `k` (see below for more details). Each one of these subsequences of length `k` is called a `k`-mer.

Example
Let `s` be the sequence `ATTACTC`. The following 3 subsequences are all the `5`-mers of `s`:

- `ATTAC` is the first `5`-mer of `s`
- `TTACT` is the second `5`-mer of `s`
- `TACTC` is the last `5`-mer of `s`

Constructing the de Bruijn Graph Given a number `k` and a set of sequences `S`, the de Bruijn graph is constucted as follows:

1. For each sequence `s` in `S`, you get all the `k`-mers of `s`.

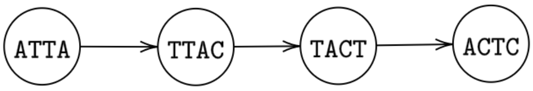
   Example
   Let `s` be the sequence `ATTACTC`. The `5`-mers of `s` are {`ATTAC`, `TTACT`, `TACTC`}.

2. For each `k`-mer of `s`, you get its two `(k-1)`-mers. We will refer to them as `L` and `R` segments of the `k`-mer.

   Example
   Let `ATTAC` be one of the `k`-mers of `s`. Then its `L` segment is `ATTA`, while its `R` segment is `TTAC`.

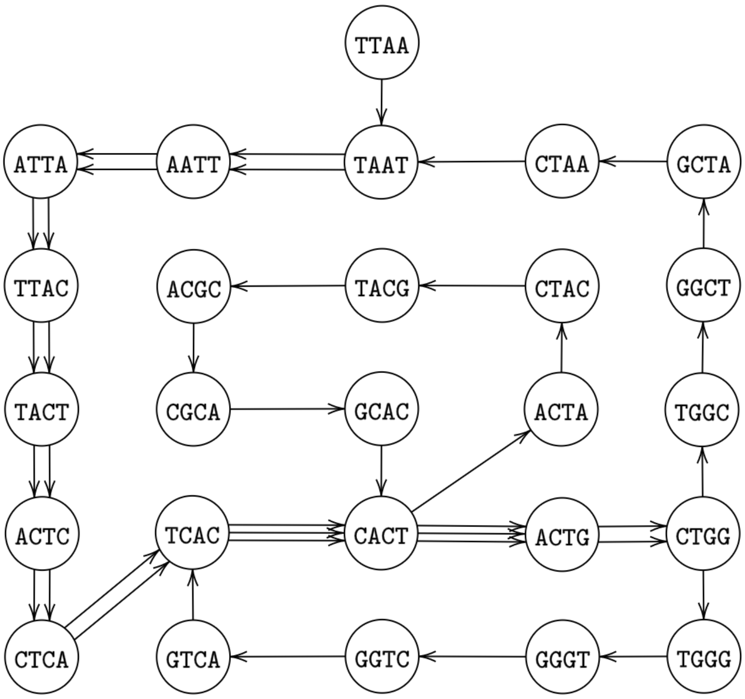
3. For each `L` and `R` segment, if the segment is not in the graph as a node, then the node is added to the graph. Next, we add an edge from the node representing the `L`-segment to the one representing the `R`-segment. (Note that we add an edge even if there was one already existing before. This type of graphs is called a *multigraph* such that there can be multiple edges between any two nodes in the graph.)

   Example
   The de Bruijn graph of `ATTACTC` that is built from its three `5`-mers is



This should be repeated for all the sequences in S.

The following figure is the de Bruijn graph of the 8 segments of SQ1 given in the beginning of the description with k = 5.



Limitations of de Bruijn Graphs Constructing a DNA sequence from the given segments through using the de Bruijn graph with some k does not always work. In order to make sure that we can get the DNA sequence from the graph, it must be the case that each k-mer of the original DNA sequence is present in exactly one of the given segments. For the scope of this project, we assume that this will always be the case. Under that assumption, we guarantee that each k-mer of the original DNA sequence is represented by a unique edge in the graph.

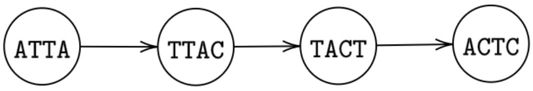
Another Limitation of de Bruijn graphs is the presence of repeated patterns. These patterns may result in generating a sequence different than the original one. But this problem is unsolvable in general (so we won't require that you must generate the original DNA sequence.)

Eurler's Path

In order to get the DNA sequence from the de Bruijn graph, we must visit every edge exactly once. This type of path is called an *Euler's Path or Eurler's Walk.*

Generating DNA Sequence from an Euler's Path Assume we have an Euler's path, $\mathtt{s}_1 \to \mathtt{s}_2 \ldots \to \mathtt{s}_\mathtt{n}$, of a given de Bruijn graph. Then the DNA sequence represented by that path is read as follows $\mathtt{s}_1\mathtt{l}_2 \ldots \mathtt{l}_\mathtt{n}$, where each $\mathtt{l}_\mathtt{i}$ is the last nucleotide of $\mathtt{s}_\mathtt{i}$.

Example
Assume we have the following de Bruijn graph



Then the only Euler's path of the following de Bruijn graph is

$$\texttt{ATTA} \rightarrow \texttt{TTAC} \rightarrow \texttt{TACT} \rightarrow \texttt{ACTC}$$

Hence, the DNA sequence is `ATTACTC`.

Existence Conditions of an Euler's Path Given a de Bruijn graph that follows our assumption, we guarantee to find at least one Euler's path. This assumption can be reflected on the graph as follows. There is at least one Euler's path in a de Bruijn graph iff the following conditions hold:

- The graph must be connected.
- The number of nodes in the graph that have its `InDegree` different from its `OutDegree` is either `0` or `2`.

  In case there are none of these nodes, then the whole graph is an *Euler cycle* and the graph is called *Eulerian*. Thus, the Euler's path in this case begins and ends at the same node.
- Moreover, in case the number of these nodes is two, then one of them must have its `InDegree - OutDegree = 1`, while the other must have its `OutDegree - InDegree = 1`.

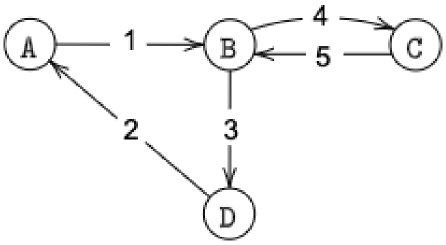
  Clearly, the one that has one extra outgoing edge will be the first edge to be traversed, while the one that the extra incoming edge will be the last edge.

Finding an Euler's Path An efficient algorithm to find Euler's paths is Hierholzer's algorithm. However, this algorithm only works for Eulerian graphs. Hence, it finds a path that represents a (Euler's) cycle.

The algorithm works as follows (given a start node `s`):

1. Starting from the current node (which is `s` in the beginning), follow a path of unique unvisited edges till you reach the current node again.
2. In case all the edges of the graph are visited, then you have the cycle.
3. Otherwise, find a node that is on the previously generated path and which has an unvisited outgoing edge to, say, node `v`. Then, starting from `v`, find a path of unvisited edges until you reach `v` again. This is similar to step 1 but for node `v` instead of `s`. Update the cycle you got from step 1 by replacing any occurrence of node `v` by the cycle you got from this step.
4. Repeat the above steps until there are no unvisited edges.

Example
Consider the following de Bruijn graph.



We want to get an Euler's cycle of this graph starting from node `A`. Assume the order of edges we visited in step 1 is the following: `1, 3, 2`.

Thus, we have the following cycle so far `A` → `B` → `D` → `A`.

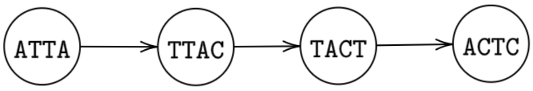
Now, you find that `B` is one of the nodes of the cycle and it has an outgoing edge that is not visited, so we apply the algorithm for this node. In this case, there is only one possible path to follow which is: `B` → `C` → `B`.

Finally, we update the original cycle to incorporate the new path of `B`. Hence, we get the following Euler's cycle of the given graph:

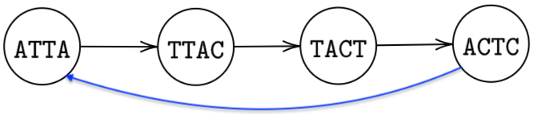
`A` → `B` → `C` → `B` → `D` → `A`

Make Sure that the Graph is Eulerian In case the graph is not Eulerian, but it satisfies the conditions of existence of an Euler's path, then we can make the graph Eulerian by adding an edge from the node that has extra incoming edge to the one that has the extra outgoing edge.

Example
Given the following non-Eulerian graph:



We can add an edge from `ACTC` to `ATTA` to make it Eulerian.



Clearly, in this case we are only interested in cycles that begins with `ATTA`. Moreover, we have to drop the last node in the cycle returned from the algorithm.

So when the algorithm returns the cycle

`ATTA` → `TTAC` → `TACT` → `ACTC` → `ATTA`

The correct one that represents the Euler's path of our original graph is

`ATTA` → `TTAC` → `TACT` → `ACTC`

Program Details & Requirements

Program Input The program requires a csv file that contains different sequences of segments of the DNA (described below). The name of the csv file is of the form `DNA_[x]_[k].csv`, where `x` is a number referring to which DNA this csv file belongs, while `k` is the number that you need to construct the de Bruijn graph with.

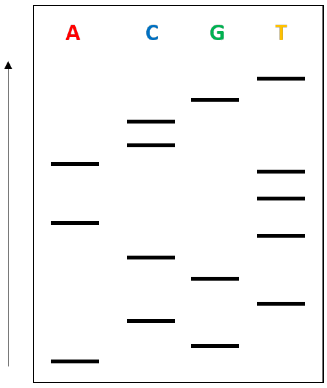
Program Output When the program has executed, two files must have been created:

- A plot of the de Bruijn graph. The name of the file must be `DNA_[x].png` where `x` is the one given as input.
- A txt file with the name `DNA_[x].txt` that contains:
  - the DNA sequence of the given data in case the de Bruijn graph satisfies the conditions of having an Euler's path; or
  - the message `"DNA sequence can not be constructed."`, otherwise.

Format of the Input Files The data in the csv files represents the readings of the 4 tubes of Sanger sequencing of DNA (which is a way of sequencing the segmented DNA).

Example

This picture[1]represents one segment that is of length 15. The order of reading the result is from bottom to top. Thus, the nucleotide at position 1 is `A`, the one at position 2 is `G`, while the last one (the one at position 15) is `T`.
Moreover, the sequence that corresponds to this graph is `AGCTGCTATTACCGT`



Each csv file contains the information of multiple segments of the same DNA. The file contains the following 6 columns in that order:

`SegmentNr`, `Position`, `A`, `C`, `G`, `T`

- `SegmentNr` is a number that represents the segment of the DNA that this record belongs to.
- `Position` is a number that represents the position of this reading in the Sanger sequence.
- Each of `A`, `C`, `G`, and `T` can have a binary value `0` or `1` that mentions which one of these 4 tubes is read at the given position. Clearly, exactly one of these 4 values should be `1` at the same position.

Example

Consider the following two sequences of DNA segments `ATTACTC` and `ACTCAC`. The following four records (among others) should be in the csv file of the DNA:

```
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
```

Note that the records in the csv file do not have to be in a specific order. Thus, you may have data about some segment between data of another segment.

[1] Source of picture https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Errors in the Data The data provided in the csv can contain errors that you need to resolve first. The following are the errors that you can expect to find in the data and instructions on how you should clean it:

1. Missing position in a segment. If the maximum position of some segment is `m`, but not all the records of the positions from `1 ... m` exist. In this case, you have to ignore the whole segment.

2. Duplicated position in a segment. If there are multiple records of the same position with the same `A`, `C`, `G`, and `T` values, then remove the duplicates and keep only one copy of the record. However, if not all of the records agree on these values, then it is impossible to know which of them is the correct one. Hence, in this case, ignore the whole segment.

3. Wrong position. If at some position all the values of `A`, `C`, `G`, and `T` are `0`s or more than one of them have the value `1`. Clearly, this is a wrong record. In this case, you have to ignore the whole segment.

4. Duplicated segments. If the sequences of two segments are identical (even if the segments have different numbers), then you need to remove one of them.

Project Tasks The program must be divided into subtasks (functions) as follows:

1. Read the csv file given its name.

   This should be done with the function `read_csv(name)` that returns a dataframe with the data.

2. Clean the data of the dataframe.

   This should be done with the function `clean_data(df)` where `df` is the dataframe with the data. This function returns the dataframe after cleaning all its errors.

3. Generate JSON sequences from the dataframe.

   This should be done with the function `generate_sequences(df)` that given the cleaned dataframe it returns a JSON with the sequences of the segments.

4. Construct de Bruijn graph.

   This should be done with the function `construct_graph(json_data, k)` where `json_data` is the one constructed from the previous step, and `k` is the length of `k`-mers. You should get the `k` from the name of the input csv file as mentioned before.

5. Plot the de Bruijn graph.

   This should be done with function `plot_graph(graph, filename)` where `graph` is the one constructed in the previous step, while `filename` is the name of the output file. You should get the name of the file from the name of the input file as mentioned before.

6. Check whether the de Bruijn graph can be sequenced.

   This should be done with the function `is_valid_graph(graph)` that returns a boolean value representing whether it satisfies the conditions of the Euler's path existence or not.

7. Construct DNA sequence.

   This should be done with the function `construct_dna_sequence(graph)` that returns a string representing the DNA sequence. Of course, this step should only be applied to valid graphs. In other words, the function should only be applied to a graph if the previous function returns `True` on that graph.

Moreover, this function should print to the console the Euler's path found to get the DNA sequence.

8. Save DNA sequence or write the error message.

   This should be done with the function `save_output(s, filename)` where `s` is the string of the DNA sequence or the error message that the DNA sequence can not be constructed, while `filename` is the name of the output file. You should get the name of the file from the name of the input file as mentioned before.

You have to name the functions as given above. You may (and are encouraged to) subdivide the functions into further tasks when necessary.

Bonus Tasks The following are three extra (optional) features that will reward you with extra points if you add them.

9. Construct all possible DNA sequences.

   This should be done with the function `construct_all_dna(graph)` that returns a set of all the possible DNA sequences of the graph.

   You should write all the returned DNA sequences in a file with the name `DNA_[x]_All.txt`. Each one of the possibilities is written in a separate line.

10. Align segments with DNA sequence.

    This should be done with the function `align_segments(json_data,dna,filename)`. This function should write in a txt file with the name `filename` the alignment of the input segments. The `dna` is the DNA sequence that the input segments are being aligned with, while `json_data` is JSON of the input segments returned from task 3.

    An alignment is valid only if every nucleotide is covered by at least one segment. The name of file must be `DNA_[x]_alignment_[z].txt` where `z` is the value of `dna`.

    Note that it is possible that you can not find an alignment. This is possible if there are more than one Euler's path in the de Bruijn graph.

    In case there is a possible alignment, then the contents of the file is in the following form:

    ```
    CTAATTACTCACTGG______________
    ___________CTGGGT____________
    _____________GGGTCACT________
    _________________CACTACGCACTG
    CTAATTACTCACTGGGTCACTACGCACTG
    ```

    On the other hand, when there is no possible alignment then your output file should have the following form:

    ```
    For this DNA sequence
    TGGGTTTG
    there is no possible alignment of the following segments
    GGGT
    GTTT
    TTGG
    ```

11. Generate all alignments with all the possible DNA sequences.

    This is basically running the function `align_segments(json_data, dna, filename)` for all the possible values of `dna` which are returned from task 9.

How to Run Your full program must be runnable with the following command (`x` and `k` are as previously defined):

```
python project.py DNA_[x]_[k].csv
```

should output a plot of the de Bruijn graph as in task 5 and write one of the possible DNA sequences out to a file as mentioned in task 8.

Moreover, if you have added any of the bonus tasks, then also the results of these functions should be generated after running the command.

Unit Tests You will be given a few csv files to test your program with. However, you will not be given any unit tests. In this project, one of the requirements is that you have to write your own unit tests. Your unit tests should reside in a file named `project_tests.py`. For each of the functions implemented in the above tasks, you should write at least three meaningful test cases.

You do not have to write test cases for the tasks: 1, 5, 8, 10, and 11.

Restriction on the Implementation Some remarks on the data structures that you shall/ shall not use:

- Use NetworkX for your graph data structure. You are only allowed to use the methods provided by the graph classes, i.e., only the functionality mentioned under the 'Graph types' section of the networkx documentation (mentioned here). You may not use any functionality from the Algorithms, Functions or other subpackages). When in doubt, do not hesitate to contact the teaching assistants.
- You are allowed to use data structures such as lists, sets, dictionaries, etc.
- You must use dataframes in the tasks where this is mentioned.
- You must use JSON where mentioned.

Project Report

In addition to the code and the test file, you should write a report that contains the sections described below. The grade of the report is part of your final exam.

Overview In this section, you should give an overview of your code structure and the general approach you followed.

Implementation Details In this section, you should describe the details of your implementation. For each of the main functions, you should

- describe the different possible approaches to implement it (if any),
- mention which approach you chose, and
- argue why your choice is a good one.

Evaluation In this section, you should describe to what degree you completed the project. You need to mention:

- what is implemented, and whether it is correctly implemented or partially implemented. For example, it could be the case that your implementation may not work as expected for some specific input.
- what is not implemented, and
- the bonus questions that you implemented

Example Run In this section, you should provide a full run for one of the csv files given to you. You should mention which csv file you use, and give all the intermediate results that your program computes for every implemented function. Describe the input and output at every step.

Challenges In this section, you have to mention what the challenges/difficulties you faced during the implementation (if any).

Tests Description In this section, you should explain the different test cases you have designed for each function. Describe for each test case *why* it is relevant.

Report Length The report should not be longer than 6 pages, *excluding* the example run. You are graded on the contents of the report, not its length.

Submission and Grading Criteria

You may have multiple files for the project. However, your main solution file must have the name `project.py`. Make sure that every file begins with

```
"""
author: [firstname lastname]
studentnumber: [your studentnumber]
"""
```

Before the deadline make sure to submit all your files, including the report in pdf format, **in a zip file** through Blackboard.

Grading In grading we do not only look at functionality and completeness, but we also take code quality and the report into account. Take care when constructing your solution, some things that will cost you points:

- pytests are too narrow. For example, you have a code that passes the pytests but the tests do not check for different possible scenarios
- your code does not run from the command line as expected
- your code does not adhere to the specified naming of functions
- generally, when not following the requirements described in this document

When in doubt, do not hesitate to contact the teaching assistants.