Eulerian Path Report

Xinyuan Min 950829573070

1. Is the graph from Fig 8.22 Eulerian (in other words: what's the result of your function is_eulerian, given this graph as input)?

   The output of my is_eulerian *function determines graph_822 as Eulerian. Because all the vertices in graph_822 are balanced.*

2. Does the graph from Fig 8.22 contain a Eulerian path?

   Graph_822 contains a Eulerian path.
   The condition of containing a Eulerian path is the number of semi-balanced vertices does not exceed 2, and all other vertices should be balanced. All the vertices in graph_822 are balanced, so graph_822 does not only contain a Eulerian path, but also is Eulerian.

3. Print the Eulerian path that your code can find in graph_822.

```
eulerian cycle for graph_822:
 [('A', 'B'), ('B', 'C'), ('C', 'J'), ('J', 'D'), ('D', 'C'), ('C', 'I'), ('I',
'H'), ('H', 'F'), ('F', 'E'), ('E', 'J'), ('J', 'F'), ('F', 'G'), ('G', 'A')]
```

   p.s. In my algorithm, each edge is defined as a tuple (start vertex, pointed vertex)

4. If you run it 3 times (print your results below), do you always find the same path? Why or why not?

   1st time:

```
[[('A', 'B'), ('B', 'C'), ('C', 'J'), ('J', 'D'), ('D', 'C'), ('C', 'I'), ('I',
'H'), ('H', 'F'), ('F', 'E'), ('E', 'J'), ('J', 'F'), ('F', 'G'), ('G', 'A')]]
```

   2nd time:

```
[[('A', 'B'), ('B', 'C'), ('C', 'J'), ('J', 'D'), ('D', 'C'), ('C', 'I'), ('I',
'H'), ('H', 'F'), ('F', 'E'), ('E', 'J'), ('J', 'F'), ('F', 'G'), ('G', 'A')]]
```

   3rd time:

```
[[('A', 'B'), ('B', 'C'), ('C', 'J'), ('J', 'D'), ('D', 'C'), ('C', 'I'), ('I',
'H'), ('H', 'F'), ('F', 'E'), ('E', 'J'), ('J', 'F'), ('F', 'G'), ('G', 'A')]]
```

   Given the same input, my algorithm would always find the same path. Because the algorithm always takes the first key in the graph(dictionary) as the start node, and then start tracing. In terms of facing multiple outgoing nodes, the first value in the value list would be selected.

5. Print the graph that you constructed from the spectrum s (Fig 8.20). Use: for k, v in graph.items(): print(k, v)

```
graph from fig.8.20:
GC ['CA', 'CG']
GT ['TG']
CG ['GT']
AT ['TG']
TG ['GG', 'GC']
GG ['GC']
CA []
```

6. Is this graph Eulerian? Why or why not? And does it contain a Eulerian path? Why or why not?

Fig.8.20 is not Eulerian, because a directed graph is called Eulerian is and only if each of its vertices is balanced. For fig.8.20, the vertices AT and CA is not balanced (indegree ≠ outdegree)

But fig.8.20 does contain a Eulerian path. According to theorem 8.2, a connected graph has a Eulerian path if and only if it contains at most two semi-balanced vertices, and all other vertices are balanced. In fig.8.20, all vertices are balanced except for vertices AT and CA, AT and CA are both semi-balanced.

7. Print the Eulerian path that your algorithm finds. To which DNA sequence does this path correspond? Print the sequence.

```
path of s:
 [('AT', 'TG'), ('TG', 'GC'), ('GC', 'CG'), ('CG', 'GT'), ('GT', 'TG'),
('TG', 'GG'), ('GG', 'GC'), ('GC', 'CA')]

DNA sequence: ATGCGTGGCA
```

8. Which Eulerian cycle or path (if any) do you find in the bigger_graph (provided in the skeleton)?

```
Eulerian cycle of bigger graph:
 [[(1, 2), (2, 3), (3, 7), (7, 8), (8, 4), (4, 10), (10, 9), (9, 4), (4, 5),
(5, 6), (6, 7), (7, 11), (11, 12), (12, 7), (7, 9), (9, 3), (3, 1)]]
```

9. Is the algorithm exact or approximate?

This is an exact algorithm, because given a Eulerian graph, the algorithm would always give a Eulerian path that travels through all edges only once (optimal instead of approximate output).

10. Is the running time of Euler's algorithm proportional to the number of nodes or the number of edges in the graph? Can you explain why?

The running time of Euler's algorithm is proportional to the number of nodes/ edges in the graph. i.e. O(n)

The algorithm starts from an arbitrary vertex and start tracing by referring to the graph (dictionary) to get the connected vertex / vertices. In terms of facing multiple connected vertices, a random path will be picked. All the traversed edge will be deleted from the graph. The traversing will stop when going back to the starting vertex. The time complexity for this part is proportional to the number of traversed edges.

And then a if statement will be employed to check if the graph dictionary is empty. If not, the above procedure will be repeated for the remaining graph. Similarly, the time complexity is also linear.

And finally, all the sub-cycles will be combined into an integrated Eulerian cycle. In my algorithm, the combining procedure is to combine the first and second sub-cycles in the sub-cycle list into one cycle, then put the combined cycle at the beginning the sub-cycle list. This procedure is repeated until all sub-cycles have been integrated into one big Eulerian cycle. The time complexity of this part is also linear.

As stated above, the overall time complexity of this algorithm is proportional to the edges in the graph.

Discussion

In my code I implemented 3 recursive function, though they made the code a bit more compact, it comes with a cost. Recursive calls generally take up more memory and time compared with loops. For this assignment, this code works well given the small graphs, but when facing much bigger graphs, the algorithm would be inefficient.