SER 501: Adv Data Struct and Algorithms

Name : **Sai Swaroop Reddy Vennapusa**

Assignment 3

Instructor : **Dr. Ajay Bansal**

Due Date : 16^{th} Nov 2023, 11:59PM

- Run flake8 in addition to testing your code; I expect professional and clear code with minimal flake8 warnings (<5) and having McCabe complexity (<10) from all of you.

```
⊗ swaroop@swaroop:~/Downloads/Assignments/SER501/Assign3$ flake8 assignment_3.py
  assignment_3.py:122:80: E501 line too long (124 > 79 characters)
⊗ swaroop@swaroop:~/Downloads/Assignments/SER501/Assign3$ flake8 --max-complexity 10 assignment_3.py
  assignment_3.py:122:80: E501 line too long (124 > 79 characters)
```

Problem 1:
Part A In your program.
Given an adjacency-matrix representation of a directed graph, implement the 'in_degree', and 'out_degree' method in the scaffolding code. You must invoke the 'print_degree' to output the degree in a certain format, so the grading script can judge the correctness of your answer.

Code :

```
def in_degree(self):
    # Method 1
    print("Method 1 for in degree")
    print("========================")
    for edge in self.edges:
        if edge[1] in self.indegree:
            self.indegree[edge[1]] += 1

    print("In degree of the graph:")
    for key, value in self.indegree.items():
        print(f"Vertex : {key} Degree : {value}")

    # Method 2:
    print("Method 2 for in degree")
    print("========================")
    in_degrees = [0] * len(self.vertices)
    for i in range(len(self.vertices)):
        for j in range(len(self.vertices)):
            if self.matrix[j][i] != 0:
                in_degrees[i] += 1

    print("In degree of the graph:")
    for i, degree in enumerate(in_degrees):
        print(f"Vertex: {self.vertices[i]} Degree: {degree}")

def out_degree(self):
    # Method 1:
    print("Method 1 for out degree")
    print("========================")
    for edge in self.edges:
        if edge[0] in self.outdegree:
            self.outdegree[edge[0]] += 1
```

```
        print("Out degree of the graph:")
        for key, value in self.outdegree.items():
            print(f"Vertex : {key} Degree : {value}")

        # Method 2:
        print("Method 2 for out degree")
        print("========================")
        out_degrees = [sum(row) for row in self.matrix]

        print("Out degree of the graph:")
        for i, degree in enumerate(out_degrees):
            print(f"Vertex: {self.vertices[i]} Degree: {degree}")
```

Output:

```
Method 1 for in degree
========================
In degree of the graph:
Vertex : 1 Degree : 0
Vertex : 2 Degree : 1
Method 2 for in degree
========================
In degree of the graph:
Vertex: 1 Degree: 0
Vertex: 2 Degree: 1
Method 1 for out degree
========================
Out degree of the graph:
Vertex : 1 Degree : 1
Vertex : 2 Degree : 0
Method 2 for out degree
========================
Out degree of the graph:
Vertex: 1 Degree: 1
Vertex: 2 Degree: 0
```

Part B In your write-up.
How long does it take (in big-O notation) to compute the out-degree of every vertex? How long does it take (in big-O notation) to compute the in-degrees? Justify your answer with proper reasoning.

Answer:

Out-Degree Calculation:

Method 1:
The first method for computing out-degrees involves iterating through each edge in the graph's edge list. For each edge, we increment the count of the out-degree for the source vertex (the first element in the edge tuple). This method has a time complexity of O(E), where E is the number of edges in the graph. This linear relationship arises because each edge is examined exactly once.

Method 2:
The second method for out-degree computation sums the entries in each row of the graph's adjacency matrix, where each row corresponds to the out-going edges from a vertex. Since the adjacency matrix is a V x V matrix (V being the number of vertices), we must inspect each of the V^2 entries, resulting in a time complexity of O(V^2).

In-Degree Calculation:

Method 1:
Similarly, the first method for in-degree calculation iterates over the edge list. Here, we increase the in-degree count for the destination vertex (the second element of the edge tuple). Like the out-degree, this method is also O(E) due to the one-to-one examination of each edge.

Method 2:
For the second method, we sum the entries in each column of the adjacency matrix to find the in-degrees, which requires us to visit all V^2 entries as well, yielding a time complexity of O(V^2).

Overall Analysis:

- For out-degree calculations:
  - Method 1: O(E)
  - Method 2: O(V^2)

- For in-degree calculations:
  - Method 1: O(E)
  - Method 2: O(V^2)

The choice between these methods depends on the properties of the graph. If the graph is sparse, with the number of edges (E) significantly less than the square of the number of vertices (V^2), the first method is more efficient. Conversely, if the graph is dense and the number of edges approaches V^2, both methods will have comparable efficiency. The implementation of these methods will also consider the specific data structures chosen for representing the graph, as this can affect the constants hidden in the big-O notation, although it does not change the overall complexity.

Problem 2:

The transpose of a directed graph G = (V, E) is the graph G T = (V, E T), where E T ={(v,u) | (u,v) □ E}. Thus, G T is G with all its edges reversed. Implement an efficient algorithm for the 'transpose' method in the scaffolding code for computing G T from G for the adjacency-matrix representation of G. Provide the running time of your algorithm in big-O notation.

Code:

```
# Method 1:
self.matrix = [list(row) for row in zip(*self.matrix)]

# Method 2:
size = len(self.matrix)
for i in range(size):
    for j in range(size):
        self.matrixT[j][i] = self.matrix[i][j]
self.matrix = [row[:] for row in self.matrixT]
self.T = True
```

Output:

```
['1', '2']
1 [0, 1]
2 [0, 0]
['1', '2']
1 [0, 0]
2 [1, 0]
['1', '2']
1 [0, 1]
2 [0, 0]
```

The running time of the code is O(V^2) where V is the number of vertices. This is because the matrix is V×V in size, and the code iterates over each element once to create the transposed matrix.

Problem 3:

Part A In your code
Show how depth-first search works on the graph below by implementing the 'dfs_on_graph' method in the scaffolding code. Assume that the vertices are explored in alphabetical order (at any point, if it must choose between q and r, then q will be explored before r), and that each vertex in the adjacency matrix is ordered alphabetically. Print the discovery and finishing times for each vertex using 'print_discover_and_finish_time' method.

Code:

```
def dfs_visit(self, u):
    Graph.dfs_timer += 1
    self.discover[u] = Graph.dfs_timer
    # Graph.user_defined_vertices[u] = True
    for v in range(len(self.vertices)):
        # if self.matrix[u][v] != 0 and not Graph.user_defined_vertices[v]:
        if self.matrix[u][v] != 0 and self.discover[v] == 0:
            self.dfs_visit(v)
    Graph.dfs_timer += 1
    self.finish[u] = Graph.dfs_timer

def dfs_on_graph(self):
    for u in range(len(self.vertices)):
        # if not Graph.user_defined_vertices[u]:
        if self.discover[u] == 0:
            self.dfs_visit(u)
    self.print_discover_and_finish_time(self.discover, self.finish)
```

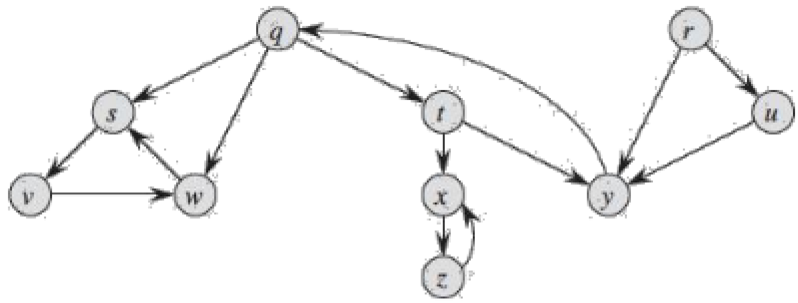
Output:

```
['q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
q [0, 0, 1, 1, 0, 0, 1, 0, 0, 0]
r [0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
s [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
t [0, 0, 0, 0, 0, 0, 0, 1, 1, 0]
u [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
v [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
w [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
x [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
y [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
z [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

Vertex: q     Discovered: 1    Finished: 16
Vertex: r     Discovered: 17   Finished: 20
Vertex: s     Discovered: 2    Finished: 7
Vertex: t     Discovered: 8    Finished: 15
Vertex: u     Discovered: 18   Finished: 19
```

Vertex: v Discovered: 3 Finished: 6
Vertex: w Discovered: 4 Finished: 5
Vertex: x Discovered: 9 Finished: 12
Vertex: y Discovered: 13 Finished: 14
Vertex: z Discovered: 10 Finished: 11

Part B In your write-up
Show the classification of each edge in your write-up
(Tree/Forward/Back/Cross).



Tree Edges:

- (q, s)
- (s, v)
- (v, w)
- (q, t)
- (t, x)
- (x, z)
- (t, y)
- (r, u)

Back Edges:

- (z, x) - z connects back to its ancestor x.
- (y, q) - y connects back to its ancestor q.

Forward Edges:

Grey to black node

- (q,w)
- (r,y)
- (u,y)

Cross Edges:

- (w,s)

Problem 4:

Suppose that we represent the graph G = (V, E) as an adjacency matrix. Give a simple (using linear search) implementation of Prim's algorithm (O(V^2) time) by implementing the 'prim' method in the scaffolding code.

Code:

```
def prim(self, root):
    n = len(self.vertices)
    root_index = self.vertices.index(root)
    self.weight[root_index] = 0

    self.print_d_and_pi("Initial", self.weight, self.parent)

    for iteration in range(n):
        min_index = -1
        min_value = sys.maxsize
        for v in range(n):
            if self.weight[v] < min_value and self.in_mst[v] is False:
                min_value = self.weight[v]
                min_index = v

        self.in_mst[min_index] = True
        for v in range(n):
            if self.matrix[min_index][v] > 0 and self.in_mst[v] is False and self.weight[v] >
self.matrix[min_index][v]:
                self.weight[v] = self.matrix[min_index][v]
                self.parent[v] = self.vertices[min_index]

        self.print_d_and_pi(iteration, self.weight, self.parent)
```

Output:

```
Iteration: Initial
Vertex: A    d: inf  pi: None
Vertex: B    d: inf  pi: None
Vertex: C    d: inf  pi: None
Vertex: D    d: inf  pi: None
Vertex: E    d: inf  pi: None
Vertex: F    d: inf  pi: None
Vertex: G    d: 0    pi: None
Vertex: H    d: inf  pi: None
Iteration: 0
Vertex: A    d: inf  pi: None
Vertex: B    d: inf  pi: None
Vertex: C    d: inf  pi: None
Vertex: D    d: inf  pi: None
```

Vertex: E d: inf pi: None
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 14 pi: G
Iteration: 1
Vertex: A d: inf pi: None
Vertex: B d: inf pi: None
Vertex: C d: inf pi: None
Vertex: D d: inf pi: None
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 10 pi: F
Iteration: 2
Vertex: A d: inf pi: None
Vertex: B d: 2 pi: E
Vertex: C d: inf pi: None
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 10 pi: F
Iteration: 3
Vertex: A d: 4 pi: B
Vertex: B d: 2 pi: E
Vertex: C d: 9 pi: B
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 5 pi: B
Iteration: 4
Vertex: A d: 4 pi: B
Vertex: B d: 2 pi: E
Vertex: C d: 9 pi: B
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 5 pi: B
Iteration: 5
Vertex: A d: 4 pi: B
Vertex: B d: 2 pi: E
Vertex: C d: 9 pi: B
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 5 pi: B
Iteration: 6

Vertex: A d: 4 pi: B
Vertex: B d: 2 pi: E
Vertex: C d: 9 pi: B
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 5 pi: B
Iteration: 7
Vertex: A d: 4 pi: B
Vertex: B d: 2 pi: E
Vertex: C d: 9 pi: B
Vertex: D d: 15 pi: E
Vertex: E d: 8 pi: F
Vertex: F d: 3 pi: G
Vertex: G d: 0 pi: None
Vertex: H d: 5 pi: B

Problem 5:

Implement 'bellman_ford' method in the scaffolding code and run the Bellman-Ford algorithm on the directed graph given below, using vertex z as the source. In each pass, relax edges in the order (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y) and invoke the 'print_d_and_pi' method to print out the d and π values after each pass. (For reference: if (u,v) is an edge which is relaxed in the pass, then v. = u)
Now, change the weight of edge (z,x) to 4 and run the algorithm again, using s as the source.

Code:

```
def bellman_ford(self, source):
    d = [sys.maxsize] * len(self.vertices)
    pi = [None] * len(self.vertices)
    source_index = self.vertices.index(source)
    d[source_index] = 0

    self.print_d_and_pi("Initial", d, pi)

    for iteration in range(len(self.vertices) - 1):
        for u, v, w in self.edges:
            u_index = self.vertices.index(u)
            v_index = self.vertices.index(v)
            if d[u_index] != sys.maxsize and d[u_index] + w < d[v_index]:
                d[v_index] = d[u_index] + w
                pi[v_index] = u

        self.print_d_and_pi(iteration, d, pi)
```

```
    for u, v, w in self.edges:
        u_index = self.vertices.index(u)
        v_index = self.vertices.index(v)
        if d[u_index] != sys.maxsize and d[u_index] + w < d[v_index]:
            print("No Solution")
            return
```

Output:

```
Iteration: Initial
Vertex: s    d: inf  pi: None
Vertex: t    d: inf  pi: None
Vertex: x    d: inf  pi: None
Vertex: y    d: inf  pi: None
Vertex: z    d: 0    pi: None
Iteration: 0
Vertex: s    d: 2    pi: z
Vertex: t    d: 8    pi: s
Vertex: x    d: 7    pi: z
Vertex: y    d: 9    pi: s
Vertex: z    d: 0    pi: None
Iteration: 1
Vertex: s    d: 2    pi: z
Vertex: t    d: 5    pi: x
Vertex: x    d: 6    pi: y
Vertex: y    d: 9    pi: s
Vertex: z    d: 0    pi: None
Iteration: 2
Vertex: s    d: 2    pi: z
Vertex: t    d: 4    pi: x
Vertex: x    d: 6    pi: y
Vertex: y    d: 9    pi: s
Vertex: z    d: 0    pi: None
Iteration: 3
Vertex: s    d: 2    pi: z
Vertex: t    d: 4    pi: x
Vertex: x    d: 6    pi: y
Vertex: y    d: 9    pi: s
Vertex: z    d: 0    pi: None
```

After changing the edge (z,x) to 4:

Iteration: Initial
Vertex: s d: 0 pi: None
Vertex: t d: inf pi: None
Vertex: x d: inf pi: None
Vertex: y d: inf pi: None
Vertex: z d: inf pi: None
Iteration: 0
Vertex: s d: 0 pi: None
Vertex: t d: 6 pi: s
Vertex: x d: inf pi: None
Vertex: y d: 7 pi: s
Vertex: z d: inf pi: None
Iteration: 1
Vertex: s d: 0 pi: None
Vertex: t d: 6 pi: s
Vertex: x d: 4 pi: y
Vertex: y d: 7 pi: s
Vertex: z d: 2 pi: t
Iteration: 2
Vertex: s d: 0 pi: None
Vertex: t d: 2 pi: x
Vertex: x d: 4 pi: y
Vertex: y d: 7 pi: s
Vertex: z d: 2 pi: t
Iteration: 3
Vertex: s d: 0 pi: None
Vertex: t d: 2 pi: x
Vertex: x d: 2 pi: z
Vertex: y d: 7 pi: s
Vertex: z d: -2 pi: t
No Solution

Problem 6:

Implement the 'dijkstra' method in the scaffolding code and run Dijkstra's algorithm on the directed graph given below, using vertex s as the source. Invoke 'print_d_and_pi' method to print out the d and π values and the vertices in set S after each iteration of the while loop in the algorithm given above.

Code:

```
    def dijkstra(self, source):
```

```
        n = len(self.vertices)
        d = [sys.maxsize] * n
        pi = [None] * n
        source_index = self.vertices.index(source)
        d[source_index] = 0
        queue = [(0, source_index)]
        visited = set()

        self.print_d_and_pi("Initial", d, pi)

        while queue:
            (dist, u_index) = heapq.heappop(queue)
            if u_index in visited:
                continue

            visited.add(u_index)
            u = self.vertices[u_index]

            for v_index, v in enumerate(self.vertices):
                if self.matrix[u_index][v_index] != 0:
                    if d[u_index] + self.matrix[u_index][v_index] < d[v_index]:
                        d[v_index] = d[u_index] + self.matrix[u_index][v_index]
                        pi[v_index] = u
                        heapq.heappush(queue, (d[v_index], v_index))

            self.print_d_and_pi(len(visited) - 1, d, pi)
```

Output:

```
Iteration: Initial
Vertex: s	d: 0	pi: None
Vertex: t	d: inf	pi: None
Vertex: x	d: inf	pi: None
Vertex: y	d: inf	pi: None
Vertex: z	d: inf	pi: None
Iteration: 0
Vertex: s	d: 0	pi: None
Vertex: t	d: 3	pi: s
Vertex: x	d: inf	pi: None
Vertex: y	d: 5	pi: s
Vertex: z	d: inf	pi: None
Iteration: 1
Vertex: s	d: 0	pi: None
Vertex: t	d: 3	pi: s
Vertex: x	d: 9	pi: t
Vertex: y	d: 5	pi: s
Vertex: z	d: inf	pi: None
Iteration: 2
```

Vertex: s d: 0 pi: None
Vertex: t d: 3 pi: s
Vertex: x d: 9 pi: t
Vertex: y d: 5 pi: s
Vertex: z d: 11 pi: y
Iteration: 3
Vertex: s d: 0 pi: None
Vertex: t d: 3 pi: s
Vertex: x d: 9 pi: t
Vertex: y d: 5 pi: s
Vertex: z d: 11 pi: y
Iteration: 4
Vertex: s d: 0 pi: None
Vertex: t d: 3 pi: s
Vertex: x d: 9 pi: t
Vertex: y d: 5 pi: s
Vertex: z d: 11 pi: y