Assignment-8 (30 points)
Graph Data Structure

Due date: 11/18/2022

Note: graph_ds word file is uploaded in Canvas which contains the basic methods of graphs.

Part-1: Write a class for graph data structure containing the following functions:

1. Function to generate the list of all edges
2. Function to calculate isolated nodes of a given graph
3. Function to find a path from a start vertex to an end vertex
4. Function to find all the paths between a start vertex to an end vertex
5. Function to check if a graph is a connected graph.
6. Function to perform BFS
7. Function to perform DFS

Part-2:

8. Write the Kruskal's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.
9. Write the Prim's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.
10. Write the Dijkstra's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.

Note: each function should take a graph as a parameter along with other required parameters. Check each function with an example. You might require more examples (test examples (minimum: 2)) for the completion of this assignment. Exercise:1-5 is already given in the graph_ds file. Some functions in the file might not take graph as an input parameter. Explain each function in 2-3 lines. Exercise: 1-5 carries 2 points each and exercise: 6-10 carries 4 points each.

Attach the code and screenshots of your results here.

```python
def generate_edges(graph: Graph):
    for _ in range(0, 10):
        x = y = 0
        while x == y:
            x = randint(0,5)
            y = randint(0,5)
        graph.add_vertex(x, y)

    graph.add_vertex(randint(6, 10), None)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
➜ Assignment-8 python main.py
0 - 5 , 4
5 - 0 , 3 , 2
1 - 2
2 - 1 , 4 , 5 , 3
3 - 5 , 2 , 4
4 - 2 , 3 , 0
9

2 - 0 , 1 , 3
0 - 2 , 3 , 4 , 1
3 - 4 , 0 , 2
4 - 3 , 0
1 - 5 , 2 , 0
5 - 1
6
```

```python
g1 = Graph()
g2 = Graph()
generate_edges(g1)
generate_edges(g2)
print(g1)
print(g2)
```

```python
print("\nisolation")

isolation1 = find_isolated_nodes(g1)
isolation2 = find_isolated_nodes(g2)
print(isolation1)
print(isolation2)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
isolation
[9]
[6]
```

```python
def find_isolated_nodes(graph: Graph):
    isolated = []
    verticies = graph.get_all_vertex()
    for i in verticies:
        if not graph.get(i):
            isolated.append(i)

    return isolated
```

```python
def find_path(start: int, end: int, path:list, graph: Graph):
    path.append(start)
    re = None
    if start == end:
        return 0

    for i in graph.get(start):
        if i in path:
            continue
        re = find_path(i, end, path, graph)
        if re == 0:
            return 0

    if re == None:
        path.pop()
```

```python
print("\nfind single path")
p1 = []
p2 = []
find_path(5, 0, p1, g1)
find_path(5, 0, p2, g2)
print(p1)
print(p2)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
find single path
[5, 0]
[5, 1, 2, 0]
```

```python
def find_all_paths(start: int, end: int, path:list, graph: Graph, walked = None):
    if walked is None:
        walked = []
    walked.append(start)

    if start == end:
        path.append(walked.copy())
        walked.pop()
        return

    for i in graph.get(start):
        if i in walked:
            continue
        find_all_paths(i, end, path, graph, walked)
    walked.pop()
```

```python
print("\nfind all paths")
p1all = []
p2all = []
find_all_paths(0, 3, p1all, g1)
find_all_paths(0, 3, p2all, g2)
print(p1all)
print(p2all)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
find all paths
[[0, 5, 3], [0, 5, 2, 4, 3], [0, 5, 2, 3], [0, 4, 2, 5, 3], [0, 4, 2, 3], [0, 4, 3]]
[[0, 2, 3], [0, 3], [0, 4, 3], [0, 1, 2, 3]]
```

```python
def is_connected(vert_encountered: list = None, start:int = None, graph: Graph = None):
    if start in vert_encountered:
        return

    vert_encountered.append(start)
    for i in graph.get(start):
        is_connected(vert_encountered, i, graph)
```

```python
print("\nfind connected")
pconnect1 = []
pconnect2 = []
is_connected(pconnect1, 0, g1)
is_connected(pconnect2, 0, g2)
print(pconnect1)
print(pconnect2)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
find connected
[0, 5, 3, 2, 1, 4]
[0, 2, 1, 5, 3, 4]
```

BFS:

```python
def BFS(start, graph: Graph,path: list = None):
    q = [start]
    visited = [start]
    while q:
        i = q.pop()
        path.append(i)

        for j in graph.get(i):
            if j not in visited:
                q.append(j)
                visited.append(j)

def DFS(start, graph: Graph, path:list = None):
    s = [start]
    while s:
        i = s.pop()
        if i not in path:
            path.append(i)
            s.extend(graph.get(i).keys() - path)
```

```python
pbfs = []
print('\nBFS')
BFS(1, g1, pbfs)
print(pbfs)

pdfs = []
print("\nDFS")
DFS(1, g1, pdfs)
print(pdfs)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
BFS
[1, 2, 3, 5, 0, 4]

DFS
[1, 2, 5, 3, 4, 0]
```

BFS:

Write the Dijkstra's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.

```python
def dijkstra(start, graph: Graph):
    distance = {}
    visitnt = []
    for i in graph.get_all_vertex_flatten():
        visitnt.append(i)
        if i == start:
            distance[i] = {"path": [start], "cost": 0}
        else:
            distance[i] = {"path": [start], "cost": float("inf")}
    walked = []
    current_path = start
    while visitnt:
        smallest = current_path
        current_cost = distance[current_path]["cost"]
        walked.append(current_path)
        for path, cost in graph.get(current_path).items():
            if path in visitnt and distance[path]["cost"] >= cost + current_cost:
                walked.append(path)
                smallest = path
                distance[path]["cost"] = cost + current_cost
                distance[path]["path"] = walked.copy()
                walked.pop()

        visitnt.remove(current_path)

        if smallest == current_path and visitnt:
            current_path = visitnt[0]
            walked = distance[current_path]["path"][:-1]
        else:
            current_path = smallest

    print_dijstra_pretty(distance)
```

```python
gdjk = Graph()
gdjk2 = Graph()
generate_val_part2(gdjk)
generate_val_part2v2(gdjk2)
print("\nDijkstra")
print(gdjk)
dijkstra(1, gdjk)
print(gdjk2)
dijkstra(1, gdjk2)
```

```
Dijkstra
1 - 2 (10), 3 (15), 6 (5)
2 - 3 (7)
3 - 4 (7), 6 (10)
4 - 5 (7)
6 - 4 (5)
5 - 6 (13)

1 -> 2 : 10
1 -> 3 : 15
1 -> 6 : 5
1 : 0
1 -> 6 -> 4 : 10
1 -> 6 -> 4 -> 5 : 17

1 - 2 (1)
6 - 7 (13)
2 - 4 (52), 3 (12)
3 - 5 (3), 6 (98)
4 - 5 (52)
7 - 4 (26)
5 - 6 (24)

1 -> 2 : 1
1 : 0
1 -> 2 -> 3 -> 6 -> 7 : 124
1 -> 2 -> 3 -> 6 : 111
1 -> 2 -> 4 : 53
1 -> 2 -> 3 : 13
1 -> 2 -> 3 -> 5 : 16
```

Create a dictionary that store all path and mark it as not visited, then travel through the paths and compare it's cost and updating the path with the least cost, and repeat until all location are visited.

The function first visit all of the graph's vertecies (n), then it loop through all of the vertecies' neighbor(logn), so the time complexity is O(n + logn)

Write the Kruskal's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.

```python
def kruskal(graph: Graph):
    # return a list of paths with increasing costs
    sorted_graph = sort_graph(graph)
    forest = []

    while len(sorted_graph) > 0:
        # get the least expensive path
        fr, to, cost = sorted_graph.pop(0)
        from_belong = find_belong(forest, fr)
        to_belong = find_belong(forest, to)

        # check if the new tree created a circle if yes skip
        # and check for race condition when forest is empty
        if (from_belong != to_belong) or (from_belong == -1 or to_belong == -1):
            new_node = (fr, to, cost)
            from_tree = [] if from_belong == -1 else forest[from_belong]
            to_tree = [] if to_belong == -1 else forest[to_belong]

            from_tree.append(new_node)
            from_tree.extend(to_tree)
            if from_belong == -1:
                forest.append(from_tree)
            else:
                forest[from_belong] = from_tree
                del forest[to_belong]

    # print result
    for i in forest[0]:
        print(f"{i[0]} -- {i[1]} : {i[2]}")

    print("\n")
```

```python
gkru = Graph()
gkru2 = Graph()
generate_val_part2_2(gkru)
generate_val_part2_2(gkru2)
print("\nKruskal")
print(gkru)
kruskal(gkru)
print(gkru2)
kruskal(gkru2)
```

```
Kruskal
1 - 2 (10), 3 (15), 6 (5)
2 - 1 (10), 3 (7)
3 - 1 (15), 2 (7), 4 (7), 6 (10)
6 - 1 (5), 3 (10), 4 (5), 5 (13)
4 - 3 (7), 5 (7), 6 (5)
5 - 4 (7), 6 (13)

1 -- 2 : 10
1 -- 3 : 15
3 -- 6 : 10
6 -- 5 : 13
4 -- 5 : 7


1 - 2 (10), 3 (15), 6 (5)
2 - 1 (10), 3 (7)
3 - 1 (15), 2 (7), 4 (7), 6 (10)
6 - 1 (5), 3 (10), 4 (5), 5 (13)
4 - 3 (7), 5 (7), 6 (5)
5 - 4 (7), 6 (13)

1 -- 2 : 10
1 -- 3 : 15
3 -- 6 : 10
6 -- 5 : 13
4 -- 5 : 7
```

This function first sort the graph and then it created a tree, then it go through all the paths in the order of least expensive to most expensive, and add it to the tree, if the path creates a circle then it will reject that path and keep on repeating until it've gone through all the path.

First this function sort the graph, then it go through all of the paths of the graph in the sorted order, because of that it have O(nlogn)

11. Write the Prim's algorithm and briefly explain the algorithm. Perform algorithm analysis to find the time complexity.

```python
def prims(graph: Graph):
    tree = []
    unvisited = []

    # choose a random starting vertex, set is unordered
    start = set(graph.get_all_vertex()).pop()
    next_node = (start, start, float('inf'))
    tree.append((start, start, 0))

    while True:
        dirty = False
        for i in graph.get(start):
            if not any(i == x for x, _, _ in tree):
                cost = graph.get_cost(start, i)
                if cost < next_node[2]:
                    dirty = True
                    next_node = (start, i, cost)
                if not any(i == x for _, x, _ in unvisited):
                    unvisited.append((start, i, cost))
        # if dirty is false that mean we've hit a dead end
        if not dirty:
            if unvisited:
                next_node = unvisited.pop()
            else:
                break

        start = next_node[1]

        tree.append(next_node)
        unvisited = [ele for ele in unvisited if start != ele[1]]

        next_node = (0, 0, float('inf'))

    # Pretty print tree
    for i in tree:
        print(f"{i[0]} -- {i[1]}: {i[2]}")
```

```python
gprim = Graph()
gprim2 = Graph()
generate_val_part2_2(gprim)
generate_val_part2_2v2(gprim2)
print("\nPrims")
print(gprim)
prims(gprim)
print("\n")
print(gprim2)
prims(gprim2)
```

```
Prims
1 - 2 (10), 3 (15), 6 (5)
2 - 1 (10), 3 (7)
3 - 1 (15), 2 (7), 4 (7), 6 (10)
6 - 1 (5), 3 (10), 4 (5), 5 (13)
4 - 3 (7), 5 (7), 6 (5)
5 - 4 (7), 6 (13)

1 -- 1: 0
1 -- 6: 5
6 -- 4: 5
4 -- 3: 7
3 -- 2: 7
6 -- 5: 13


1 - 2 (1)
2 - 1 (1), 4 (52), 3 (12)
6 - 7 (13), 3 (98), 5 (24)
7 - 6 (13), 4 (26)
4 - 2 (52), 5 (52), 7 (26)
3 - 2 (12), 5 (3), 6 (98)
5 - 3 (3), 4 (52), 6 (24)

1 -- 1: 0
1 -- 2: 1
2 -- 3: 12
3 -- 5: 3
5 -- 6: 24
6 -- 7: 13
7 -- 4: 26
```

This function first choose a random starting point, and then it traverse to it’s neighbor with the lowest cost, it’s then repeat until it have traverse all of the paths.

This function uses two loop, first loop will continue until all of the path are traverse, and the 2^{nd} loop inside it will traverse from the starting vertex to all of it’s neighbor, even though there is a nested loop because of the inside loop keeping track of where it have go through the loops wouldn’t run twice, so the resulting big o is O(nlogn)