CMPSC 412 – Lab-8 (50 points)
Graph - Basics

Due date: 11/01/2022

Lab Exercises: Implementing Graphs, BFS, DFS and Dijkstra's algorithm

Exercise-1:

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

1. Function to generate the list of all edges
   #def generate_edges(graph):

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

```
44  def generate_edges(graph: Graph):
43      for _ in range(0, 10):
42          x = y = 0
41          while x == y:
40              x = randint(0,5)
39              y = randint(0,5)
38          graph.add_vertex(x, y)
37
36      graph.add_vertex(6, None)
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER: VARIABLES TERMINAL

```
➜ Lab8 python main.py
2 - 5, 1, 4
5 - 2, 1, 3
1 - 0, 5, 4, 2, 3
0 - 1, 4
4 - 0, 1, 2
3 - 1, 5
6

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

This function randomly generated two points and add them in the graph. Because of the requirement of minimal 2 examples, I opt to use random number generator instead of hard coding the values because test hard coded value twice don't make sense, vertex 6 are there to make sure there is at least 1 isolated node in the graph for the next example, while the extra while loops are there to prevent duplicate values.

2. Function to calculate isolated nodes of a given graph
#def find_isolated_nodes(graph):

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

```
6   def find_isolated_nodes(graph: Graph):
7       isolated = []
8       verticies = graph.get_all_vertex()
9       for i in verticies:
10          if not graph.get(i):
11              isolated.append(i)
12
13      return isolated
14
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER: VARIABLES TERMINAL

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

this function work by looking at all the vertices and whichever vertex don't have connections is the isolated

3. Function to find a path from a start vertex to an end vertex
   #def find_path(start_vertex, end_vertex, path=None):

```
p1 = []
p2 = []
find_path(5, 0, p1, g1)
find_path(5, 0, p2, g2)
print(p1)
print(p2)
```

```
5   def find_path(start: int, end: int, path:list, graph: Graph):
6       path.append(start)
7       re = None
8       if start == end:
9           return 0
10
11      for i in graph.get(start):
12          if i in path:
13              continue
14          re = find_path(i, end, path, graph)
15          if re == 0:
16              return 0
17
18      if re == None:
19          path.pop()
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER: VARIABLES TERMINAL

```
find single path
[5, 2, 1, 0]
[5, 1, 3, 2, 0]
```

this function recursively go through all verticies connected to the start vertex and check if it is what its looking for, if it is then the function will return code 0 and stop continue to search, if it have gone through all possible path and still haven't found the solution, the path will be pop as the call stack getting releases and finally the path list will become empty

4. Function to find all the paths between a start vertex to an end vertex
   #def find_all_paths (start_vertex, end_vertex, path=[]):

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

```
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()
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER: VARIABLES TERMINAL

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

this function will recursively go through all possible path to find the end from the start, once it found one it will append walked list into the final path list, then it pop walked list as it returns so the walked list can continue to be use for subsequence recursion with differents values.

5. Function to check if a graph is a connected graph.
   #def is_connected(vertices_encountered=**None**,start_vertex=**None**):

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

```
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)

PROBLEMS  OUTPUT  DEBUG CONSOLE  JUPYTER: VARIABLES  TERMINAL

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

this function will recursively walks through all connected path and add it to the vert_encountered list if a path isn't already in it.

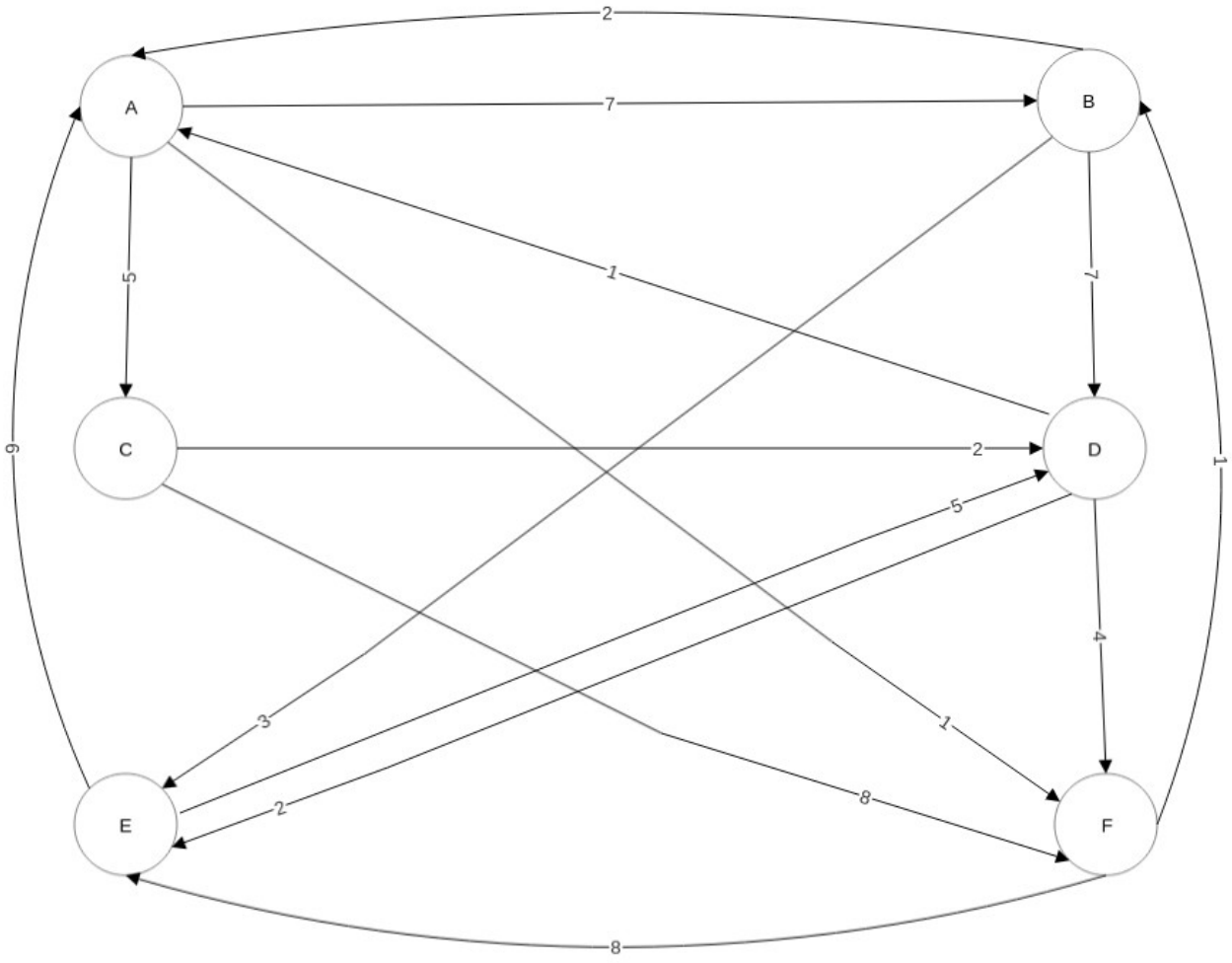
Note: each function should take a graph as a parameter along with other required parameters. Check each function with an example. You might require more examples (test examples (minimum: 2)) for the completion of this assignment. Some functions in the file might not take graph as an input parameter. Explain each function in 2-3 lines.

Attach the code and screenshots of your results here.

Exercise-2:
Draw the graph corresponding to the following adjacency matrix. Attach the screenshot of the image here (if drawn on a paper)

	A	B	C	D	E	F
A		7	5			1
B	2			7	3	
C		2				8
D	1				2	4
E	6			5		
F		1			8	

Attach the code and screenshots of your results here. (no code cause this is a drawing)



Exercise-3:
Implement the graph corresponding to the following list of edges

from	to	cost
1	2	10
1	3	15
1	6	5
2	3	7
3	4	7
3	6	10
4	5	7
6	4	5
5	6	13

1. Ignoring the weights, (program) implement/perform a breadth first search on the graph from the previous question.

```
 7  def BFS(start, graph: Graph,path: list = None):
 8      q = [start]
 9      visited = [start]
10      while q:
11          i = q.pop()
12          path.append(i)
13
14          for j in graph.get(i):
15              if j not in visited:
16                  q.append(j)
17                  visited.append(j)
18
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
BFS
['A', 'F', 'C', 'B', 'E', 'D']
```

2. (Program) Implement/Perform a depth first search for the above graph.

```
gcost = Graph()
pcost = []
generate_val_graph(gcost)
print("\nDFS")
DFS(1, gcost, pcost)
print(pcost)
```

main.py > ...

```
69
68  def DFS(start, graph: Graph, path:list = None):
67      s = [start]
66      while s:
65          i = s.pop()
64          if i not in path:
63              path.append(i)
62              s.extend(graph.get(i).keys() - path)
61
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL

```
DFS
[1, 6, 4, 5, 3, 2]
```

3. (Program) Implement Dijkstra's algorithm to the graph shown above.

main.py > ...

```
196
  1  def dijkstra(start, graph: Graph):
  2      distance = {}
  3      visitnt = []
  4      for i in graph.get_all_vertex_flatten():
  5          visitnt.append(i)
  6          if i == start:
  7              distance[i] = {"path": [start], "cost": 0}
  8          else:
  9              distance[i] = {"path": [start], "cost": float("inf")}
 10      walked = []
 11      current_path = start
 12      while visitnt:
 13          smallest = current_path
 14          current_cost = distance[current_path]["cost"]
 15          walked.append(current_path)
 16          for path, cost in graph.get(current_path).items():
 17              if path in visitnt and distance[path]["cost"] >= cost + current_cost:
 18                  walked.append(path)
 19                  smallest = path
 20                  distance[path]["cost"] = cost + current_cost
 21                  distance[path]["path"] = walked.copy()
 22                  walked.pop()
 23
 24          visitnt.remove(current_path)
 25
 26          if smallest == current_path and visitnt:
 27              current_path = visitnt[0]
 28              walked = distance[current_path]["path"][:-1]
 29          else:
 30              current_path = smallest
 31
 32      return distance
 33
```

PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER TERMINAL zsh

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

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

```
gdij = Graph()
generate_val_graph(gdij)
print("\nDijkstra")
print(gdij)
distance = dijkstra(1, gdij)
print_dijstra_pretty(distance)
```

Attach the code and screenshots of your results here.

Deliverables: Report, codes and the demonstration video (~3 minutes)
For video demonstration, answer the following questions:

1. In exercise-1, explain how Function to find all the paths between a start vertex to an end vertex works?
2. Explain the program for BFS, DFS and Dijkstra's algorithm?