Week 2 Coding Assignment

Question 1:

```
class UndirectedGraph:
    def __init__(self, no_of_vertices=None):
        self.no_of_vertices = no_of_vertices
        self.adj_list = {}  # Adjacency list for the graph
        self.k = 0  # for counting the no of edges

    def __str__(self):
        string = []
        # counting the no of edges
        for i in self.adj_list.keys():
            for j in self.adj_list[i]:
                if j >= i + 1:
                    self.k = self.k + 1
```

In the above code we have defined the class ***UndirectedGraph*** where the ***self.adj_list*** is the adjacency list for the graph which is defined as a dictionary and also counting of the edges is shown in the code.

```
def addNode(self, node_no):
    self.node_no = node_no
    # if no of vertices is specified checking to see node_no is
    # less than the number of nodes
    if self.no_of_vertices != None:
        if node_no <= self.no_of_vertices:
            self.adj_list[node_no] = []
        else:
            try:
                raise Exception('Node index cannot exceed number of
nodes')
            except Exception as inst:
                print(type(inst))
                print(inst)
    # For a free graph assigning a empty list to the node
    else:
        self.adj_list[node_no] = []
    return self
```

In the above code **addNode** function is defined which adds node to both free and bounded (bounded by no of nodes) graphs. It also checks whether a node was already added before if not it adds it to the list of the node

```
def addEdge(self, *nodes):
    self.node = [*nodes]
    # When no of vertices are specified
    if self.no_of_vertices != None:
        for i in range(1, self.no_of_vertices + 1):
            if i not in self.adj_list.keys():  # adding an node for the
edge if its was added before
                self.adj_list[i] = []
        for i in range(1, self.no_of_vertices + 1):
            # updating the list for the nodes when it has not been
added before
            if i == self.node[0] and self.node[1] not in
self.adj_list[i]:
```

```
                self.adj_list[i].append(self.node[1])
            if i == self.node[1] and self.node[0] not in
self.adj_list[i]:
                self.adj_list[i].append(self.node[0])
    # Free graphs we only add the node no for the edges specified
    else:
        if self.node[0] not in self.adj_list.keys():  # Checking
whether a node if it was not already added
            self.adj_list[self.node[0]] = []
        if self.node[1] not in self.adj_list.keys():
            self.adj_list[self.node[1]] = []

        if self.node[1] not in self.adj_list[self.node[0]]:
            self.adj_list[self.node[0]].append(self.node[1])
        if self.node[0] not in self.adj_list[self.node[1]]:
            self.adj_list[self.node[1]].append(self.node[0])

    return self
```

<u>addEdge</u> function is defined which adds edges to graphs both free and unbounded. For bounded graphs the edges are added and nodes are created if it was not already there in adjacency list same occurs for the free graph.

```
def __add__(self, other):
    # checking whether a single argument was provided
    if type(other) == int:

        self.adj_list[other] = []
    else:
        # if not we store it in a list and use it later
        self.node = list(other)
        # Same logic as used in addEdge method
        if self.no_of_vertices == None:

            if self.node[0] not in self.adj_list.keys():
                self.adj_list[self.node[0]] = []
            if self.node[1] not in self.adj_list.keys():
                self.adj_list[self.node[1]] = []
            if self.node[1] not in self.adj_list[self.node[0]]:
                self.adj_list[self.node[0]].append(self.node[1])
            if self.node[0] not in self.adj_list[self.node[1]]:
                self.adj_list[self.node[1]].append(self.node[0])
        else:
            for i in range(1, self.no_of_vertices + 1):
                if i not in self.adj_list.keys():
                    self.adj_list[i] = []
            for i in range(1, self.no_of_vertices + 1):
                if i == self.node[0] and self.node[1] not in
self.adj_list[i]:
                    self.adj_list[i].append(self.node[1])

                if i == self.node[1] and self.node[0] not in
self.adj_list[i]:
                    self.adj_list[i].append(self.node[0])
```

Operator overload is performed also type(other) checks whether the argument is a +ve integer or a tuple to perform similar operations corresponding to addNode and addEdge methods

```python
def plotDegDist(self):
    k = 0
    degree_dist = {}  # storing the degree distribution
    # storing  node degrees for all the nodes
    self.node_degrees = [len(self.adj_list[i]) for i in
self.adj_list.keys()]
    # populating degree distribution  which stores the degrees as keys
    # and the number  of vertices with that degree as its value
    for i in self.node_degrees:
        for j in self.adj_list.keys():
            if i == len(self.adj_list[j]):
                k = k + 1
        degree_dist[i] = k
        k = 0

    degree_list = [i * degree_dist[i] for i in degree_dist.keys()]
    # finding aout average degree of the graph
    avg_degree = sum(degree_list) / len(self.adj_list.keys())
    # node degrees
    x = [i for i in range(len(self.adj_list.keys()))]
    # proportion of vertices
    y = [0 for i in x]
    for i in range(len(x)):
        if x[i] in degree_dist.keys():
            y[i] = degree_dist[i] / len(self.adj_list.keys())
        else:
            y[i] = 0
```

The above code performs the plotting before that a dictionary named **degree_dist** is created to create whose keys are the different degrees of the graph and the value corresponding are the no of such degree vertices. X values and Y values are calculated suitably to perform the plotting where the list x contains the node degrees and the corresponding y value contains the proportion of such vertices.

Question 2:

```python
class EERRandomGraph(UndirectedGraph):
    def __init__(self,no_of_vertices):
        self.vertices=no_of_vertices
        self.adj_list={}
        self.prob_par=0
        UndirectedGraph.__init__(self,self.vertices)
    def sample(self,prob_par):
        #self.adj_list={}
        #print(self.no_of_vertices)
        self.prob_par=prob_par
        for i in range(1,self.vertices+1):
            self.adj_list[i]=[]
        for i in range(1,self.vertices):
            for j in range(i+1,self.vertices+1):
                r =random.random()
                if r <prob_par:
                    self.adj_list[i].append(j)
                    self.adj_list[j].append(i)
                #print("i=",i,"j=",j)
        #print(self.adj_list)
        return self
```

Class **EERRandomGraph** is created as a derived class of **UndirectedGraph** which is a random graph created by a parameter **prob_par** and no of nodes specified. This class has a method **Sample** creates a graph by adding edges randomly where the randomness depends upon the value **r** if the value of r is less than **prob_par** we add the edge otherwise we skip it.

Question 3:

```
def BFSUtil(self, temp, v, visited):
    # Mark the current vertex as visited
    visited[v-1] = True

    # Store the vertex to list
    temp.append(v)

    # Repeat for all vertices adjacent
    # to this vertex v
    for i in self.adj_list[v]:
        if visited[i-1] == False:
            # Update the list
            temp = self.BFSUtil(temp, i, visited)
    return temp

def isConnected(self):
    visited = []
    cc = []
    for i in range(self.no_of_vertices):
        visited.append(False)
    for v in range(self.no_of_vertices):
        if visited[v] == False:
            temp = []
            cc.append(self.BFSUtil(temp, v+1, visited))
    #print(cc)
    if len(cc)==1:
        return True
    else:
        return False
```

We have defined a method **isConnected** which uses the function **BFSUtil** which uses BFS to find out whether the graph is connected or not. The **isConnected** method returns all the connected components and hence if the variable **cc** contains only a single list then we return true otherwise false.

```
k=0
x=np.arange(0.0,0.1,0.01)
ratio_dist={i: 0 for i in x}

for p in x:
 for i in range(1000):
     g=EERRandomGraph(100)
     g.sample(p)
     r=g.isConnected()
     if r==True:
         k=k+1
 ratio_dist[p]=k
 k=0

ratio_proportion=[int(ratio_dist[i])/1000 for i in ratio_dist.keys()]
```

```
fig,ax=plt.subplots()
ax.plot(x,ratio_proportion,color="b")
ax.set_xlabel("p")
ax.set_ylabel("fraction of runs G(100, p) is connected")
ax.set_title("Connectedness of a G(100, p) as function of p")
plt.grid(True, which='both')
ax.set_ylim([0,1])
plt.axvline(x=np.log(100)/100,color="r",label="Theoretical threshold")
plt.yticks(np.arange(0,1.2,0.2))
plt.xticks(np.arange(0,0.12,0.02))
ax.legend()
plt.show()
```

We plot the graph to verify the threshold to check at what range of the probability the random graph becomes conncted.

Question 4:

```
def oneTwoCompomentSizes(self):
    visited = []
    cc = []
    for i in range(self.no_of_vertices):
        visited.append(False)
    for v in range(self.no_of_vertices):
        if visited[v] == False:
            temp = []
            cc.append(self.BFSUtil(temp, v+1, visited))
    len_cc=[len(i) for i in cc ]
    len_cc.sort(reverse=True)

    return [len_cc[0],len_cc[1]]
```

oneTwoCompomentSizes It is the same method as **isConnected** the only difference this time is it returns the length of the two biggest size components.