Programming Assignment 1

Daniel Raw

9/21/2020

CSCI303

```
import sys
from random import random

nVal = float(input("Input n "))
while nVal < 2:
    nVal = float(input("Input n "))
if nVal > 1:
    pVal = float(input("Choose an input for p that is between [0,1] "))
    while (pVal < 0 or pVal > 1):
        pVal = float(input("Choose an input for p that is between [0,1]
"))

    #1: Generating a random graph passing n and p
    def random_graph(n,p): #Using an agency adjacency list
        G = list()

        for node in range(n): #create an empty list for adjacent node
            G.append(set()) #establish a set

        for node in range(n):
            for adjacent_node in range(n):
                q = random() #generate random number
                if ( q < p and node != adjacent_node ):
                    G[node].add(adjacent_node)
                    G[adjacent_node].add(node)

        for i in range(len(G)): #change to list
            G[i] = list(G[i])
        return G

    #2: Computing the size of the largest conected component
    def large_cc(t, graph):
        levels = {t : 0} #levels dictionary
        parent = {t : None} #parent dictionary
        track = [t] #tracks the nodes in component
        frontier =[t] #frontier list
        i = 1

        while frontier: #BFS traversal from class
```

```
            next_it = []
            for x in frontier:
                for y in graph[x]:
                    if (y not in levels):
                        levels[y] = i
                        parent[y] = x
                        next_it.append(y)
                        track.append(y) #append the tracker
            frontier = next_it #set frontier to next_it
            i = i + 1
    return track #return the tracker than checks the nodes
        #if (len(track) <= t):
        #    return 1
        #return 0

    def main(n,p,t = None):
        n = int(n)
        tracker = list() #make tracker a list
        adjacent_list = random_graph(n,p)
        if (t is None):
            t = float(input("Choose a whole number that is greater than 1 for
t. "))
            while t < 1: #make sure t is greater than 1
                t = float(input("Again, please choose a number that is greater
than 1 for t. "))
            t = int(t)
        for i in range(n):
            if (i not in tracker):
                checker = large_cc(i, adjacent_list) #pass i and and
random_graph
                for i in checker: #to update the tracker
                    if (i in tracker):
                        exit("counted one node more than once")
                    tracker.append(i)
                if (len(checker) >= t): #if the length is larger than t
                    return 1
    return 0

    #3: Testing algorithm
    def test(c, n = 40):
        iteration = 0
        i = 1
    #    z = 500
        p = float(c)/float(n)
        #for i in range(z):
        while i <= 500: #generate 500 graphs
```

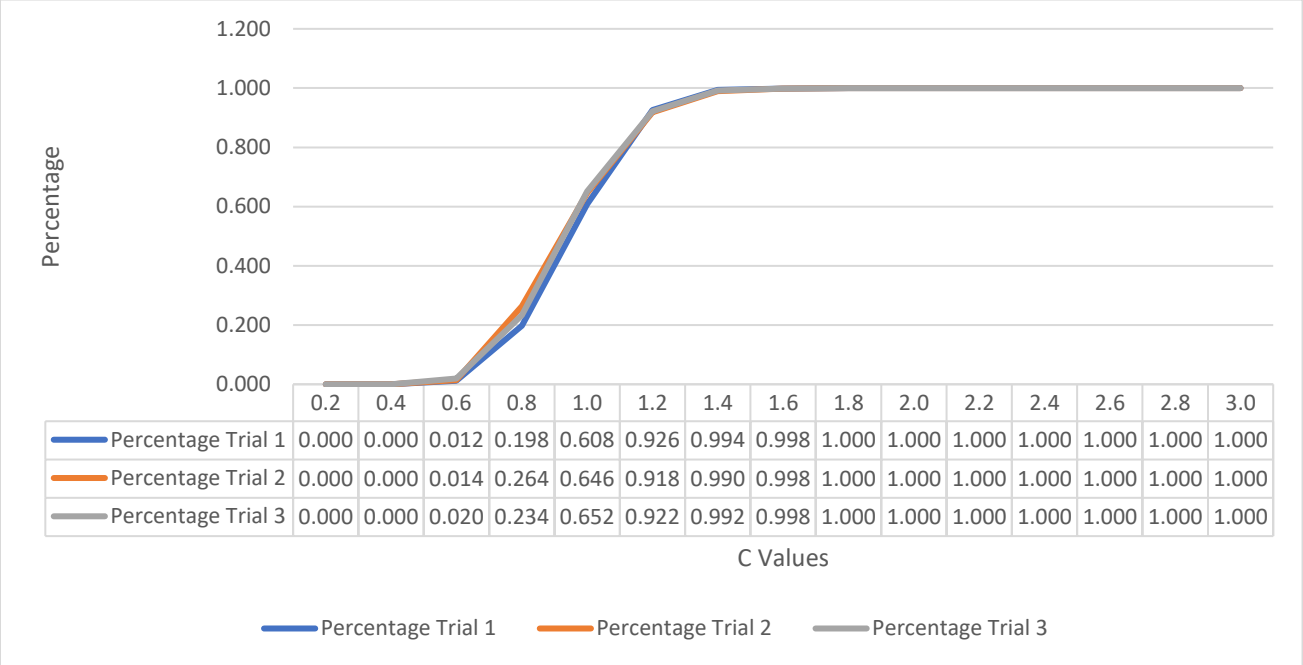
```
        val = main(n,p,t = 30)
        if (val == 1):
            iteration = iteration + 1
        i = i + 1
    return float(iteration/500)

def varTest(): #test to visibly see points and make sure code runs
properly
    c = 0.2
    print("For each c in the range [0.2,3.0] and incrementing by 0.2,
generating 500 graphs ")
    while c <= 3: #when the loop stops
        graphPercent = test(c)
        print("The c-value is {} and the percentage is {} out of 500
graphs".format(c,graphPercent))
        c = c + 0.2 #increment by 2
        c = round(c, 1)
    print("Generated 7500 Erdos-Renyi random graphs, and 500 graphs for
each c")
print(main(nVal,pVal))
varTest()
#random_graph(4,0.5)

def exit(msg):
    sys.exit(msg)
```

For question number one, I created a function called random_graph that creates an adjacency list. The variable q tracks the random number that is generated. The array list is used to check every vertices and its corresponding connected vertices as well. The two nested for loops go through each vertex to see if there is a connection with two vertices. For question number two, my large_cc function traverses by BFS. The track variable stores every node that is traversed in the algorithm into a list, and is returned at the end. In my main function, the tracker similarly stores all of the traversed nodes in the graph into a list, but importantly, it does not do a BFS traversal for nodes that have already been traversed. The BFS traversal returns a list of all of the nodes that are connected to each other, so hence, it does not have to traverse the nodes that have already been traversed because the connected component has already been accounted for.

If at the end of the traversal, if length of the variable checker is greater than t, then 1 is returned, otherwise, 0 is returned.



	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
Percentage Trial 1	0.000	0.000	0.012	0.198	0.608	0.926	0.994	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Percentage Trial 2	0.000	0.000	0.014	0.264	0.646	0.918	0.990	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Percentage Trial 3	0.000	0.000	0.020	0.234	0.652	0.922	0.992	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000