Donovan Guelde
CSCI 5352
PS3

1. $Q = \sum_r (e_{rr} - a_r^2)$ Eq. (7.76)

e_{rr} is given by the diagonals of the table given in Problem 1, since this value represents the fraction of all edges that connect like vertex to like vertex (in other words, a given edge e that is included in a diagonal has one endpoint connected to a man of ethnicity r, and the other to a woman of ethnicity r). a_r, the number of edges that connect to a given vertex type r, can be found by: $A_{rr} + \sum_{s \neq r}(A_{rs} + A_{sr})$. This represents the number of edges with at least one endpoint connecting to vertex type r.

This gives us the following values:

Ethnicity	e_{rr}	a_r	${a_r}^2$
Black	0.258	0.352	0.124
Hispanic	0.157	0.292	0.085
White	0.306	0.494	0.244
Other	0.016	0.119	0.014

So, by Eq.(7.76), Q = 0.27

Overall, the community exhibits distinct homophily. The only outlier is the *other* vertex type, which is not surprising, considering that this vertex type is actually a collection of vertices of varying types. These smaller subtypes may be small minorities of the community as a whole.

2. a. A line graph with n vertices (n-1 total edges, 2n-2 total endpoints), whose nodes are divided into two contiguous groups, r and $\neg r$ results in the following configuration:

Component	(e_{rr})	(a_r)
r	$\frac{r-1}{n-1}$	$\frac{2r-1}{2n-2}$
$\neg r$	$\frac{n-r-1}{n-1}$	$\frac{2n-2r-1}{2n-2}$

e_{rr} is the number of nodes that link vertices of the same component.
a_r is calculated by $\frac{number\ of\ endpoints\ attached\ to\ a\ component's\ vertices}{total\ number\ of\ endpoints\ in\ the\ graph}$
$Q = \sum_r (e_{rr} - {a_2}^2)$

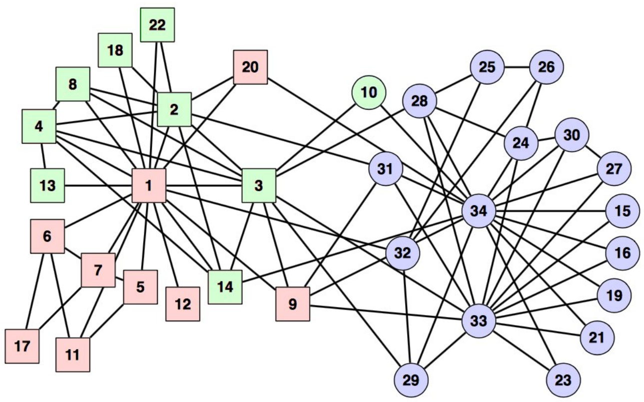
$Q = \frac{r-1}{n-1} - (\frac{2r-1}{2n-2})^2 + \frac{n-r-1}{n-1} - (\frac{2n-2r-2}{2n-2})^2$

$Q = \frac{3-4n+4rn-4r^2}{2(n-1)^2}$

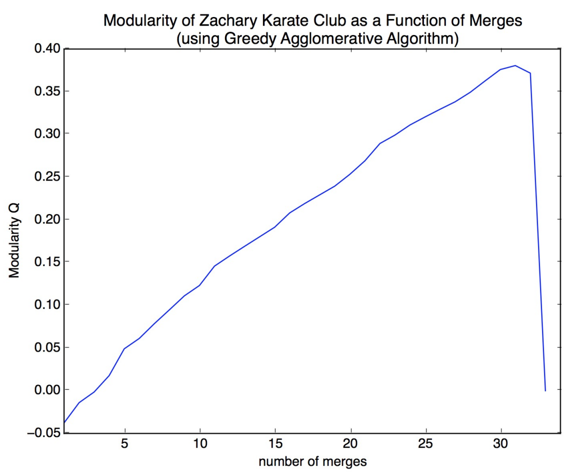
2. b. If the graph has n nodes where n is even, then each component has $r = \neg r = \frac{n}{2}$ nodes. To maximize Q, we can use the first and second derivatives of the result from 2.a. $\frac{d}{dr}\frac{3-4n+4rn-4r^2}{2(n-1)^2} = \frac{2(n-2r)}{(n-1)^2}$ which equals 0 at $r = \frac{n}{2}$, and $\frac{d^2}{du^2} = \frac{-4}{(n-1)^2}$, which is negative for

all values of r, so $r = \frac{n}{2}$ is a global maximum for the equation $Q = \frac{3-4n+4rn-4r^2}{2(n-1)^2}$. This tells us that, when n is even, the division resulting in optimal modularity is one where both components have an equal number of nodes.

3. Using the greedy agglomerative algorithm on the Zachary Karate Club network, the maximum Q score achieved was 0.3806 with three groups (after 31 merges), as shown here:



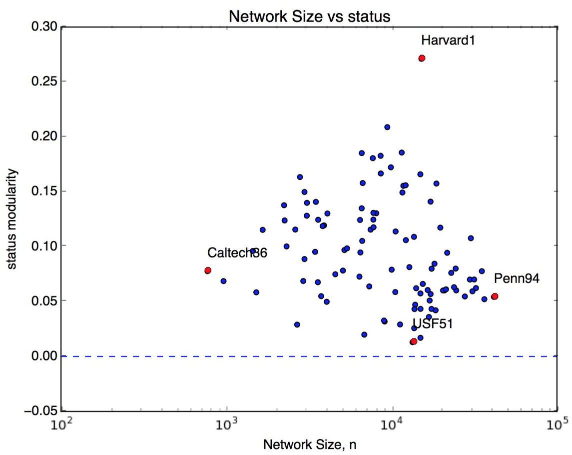
Q as a function of number of merges gives:

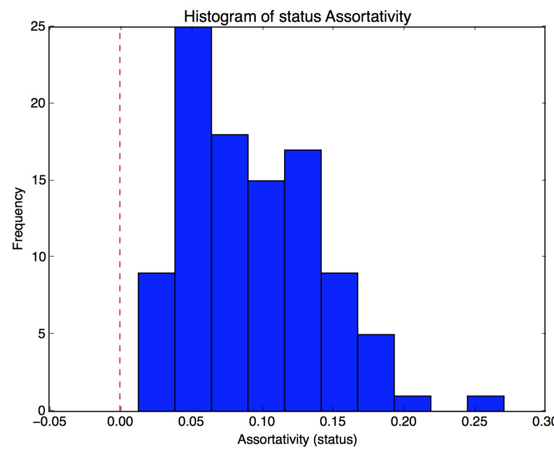


The Normalized Mutual Information is $\frac{2I(C,C')}{H(C)+H(C')}$
where $H(C) = -\frac{17}{34}log_2\frac{17}{34} - \frac{17}{34}log_2\frac{17}{34} = 1$
$H(C') = -\frac{8}{34}log_2\frac{8}{34} - \frac{9}{34}log_2\frac{9}{34} - \frac{17}{34}log_2\frac{17}{34} = 1.498751$
and $I(C,C') = H(C) - H(C|C') = 1 - (\frac{8}{34}log_2 1 - \frac{8}{34}log_2\frac{8}{9} - \frac{1}{34}log_2\frac{1}{9} - \frac{1}{34}log_2\frac{1}{17} - \frac{16}{34}log_2\frac{16}{17})$
$= 1.0 - .0.294594 = 0.705406$
$NMI = \frac{2(0.705406)}{2.498751} = 0.564607$

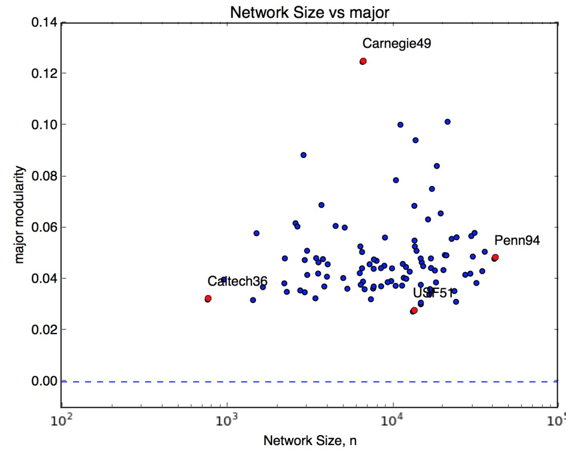
Since NMI = 0.564607, this would suggest that, without knowing the 'truth on the ground',' it is possible to make relatively accurate, but not perfect, predictions about group membership with the greedy agglomerative algorithm. It is interesting that the greedy algorithm gave three partitions, while the truth on the ground was made up of only two. Nodes 2 and 3 seem to be the driving cause of this added cluster, forming many triangles with each other and surrounding nodes, with node 1 doing the same for the remaining nodes in the original (red) partition.

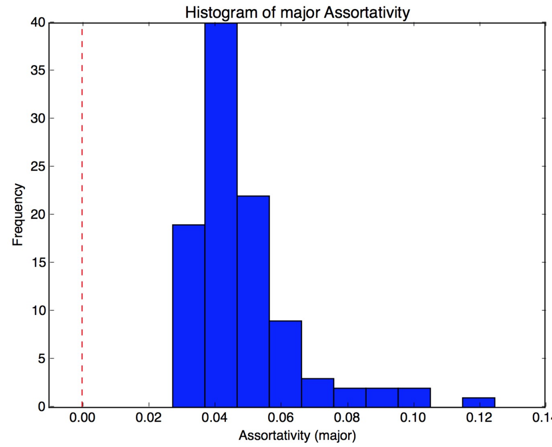
4. Assortativity/modularity was calculated using the iGraph library for Python, as the modularity and assortativity functions therein are calculated using the same methods as presented in class and lecture notes.



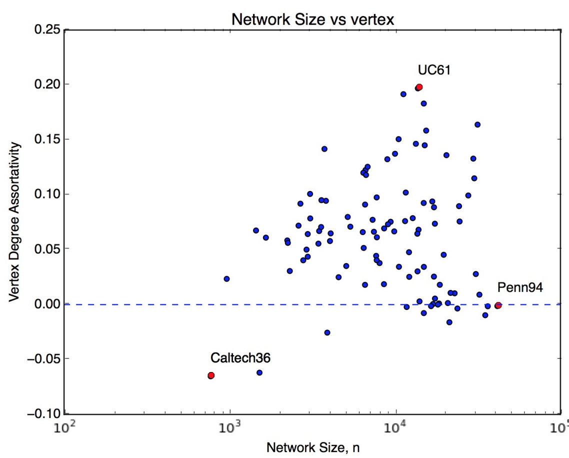


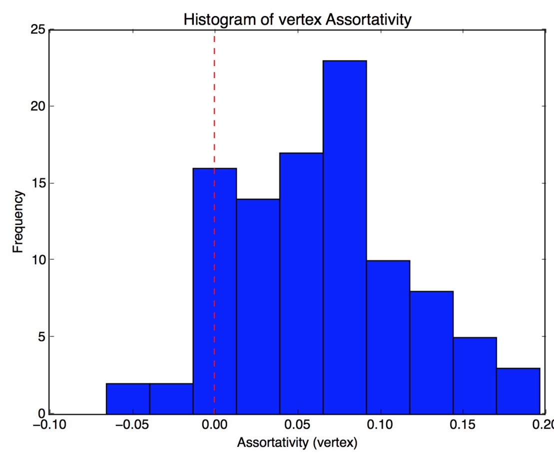
Network size and modularity of status show distinct homophily. Networks of all sizes exhibit positive modularity with respect to status (faculty, student, alumni, etc). The most common degree of assortativity is around the 0.05 range, but the histogram has a rather long, positive tail. This seems logical, since current students most likely connect with other students, faculty to faculty, etc. Many universities most likely have a policy regarding faculty/student interactions outside of official business, so this may act to reinforce the homophily of the networks by actively discouraging connections between some groups.





Networks also exhibit homophily with regards to major. The distribution is considerably more narrow than seen in the status data, with 40% of networks having assortativity with respect to major at around 0.045. Again, this is not surprising, considering that students most likely spend a lot of time with other students of the same major. It is impossible to tell from the data given if these are purely 'friendship' connections, or if the data reflects a more academic connection between nodes, such as project or study groups.



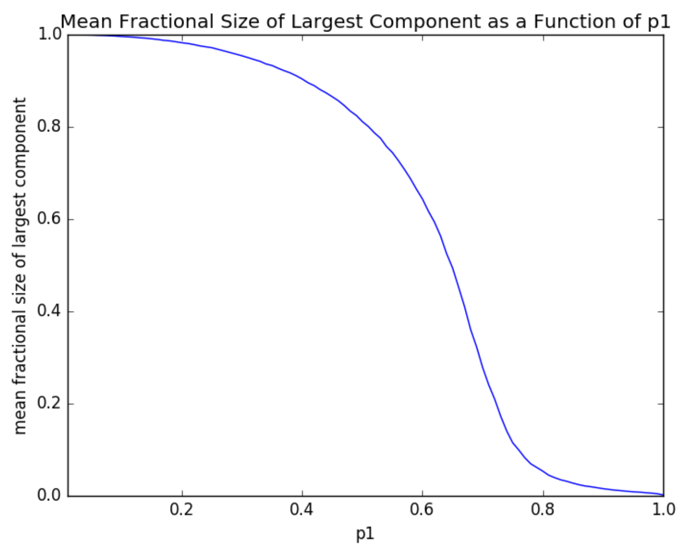


Assortativity with respect to vertex degree exhibits what seems to be the widest range of assortativity of all data sets examined, with some universities showing negative assortativity. However, the histogram shows a wide distribution from 0.0 to 0.75, and with a long positive tail. The lower vertex degree associations could be caused by the friendship paradox, as discussed earlier in class. Since the average person's friends have more friends than she does, it follows that nodes link to nodes with a dissimilar degree number. This does not explain the wide distribution, however.

6. To compute fractional sized of configuration model graphs, the networkx library for Python was used for convenience, as it has built-in functions for the generation of configuration model random graphs and finding the connected components therein.

Part 1: To determine the mean fractional size of the largest component for a network with $n = 10^4$ vertices, and with $p_1 = 0.6$ and $p_3 = 1 - p_1$, 2663 iterations were performed. Each iteration consisted of the generation of a random graph by the networkx.configuration_model() function, using randomly generated degree sequences, matching the probabilities given. The number 2663 was chosen to achieve a 99% confidence level and a confidence interval of 2.5*. Under these conditions, the mean fractional size of the specified random network was 0.64204.

Part 2: Again using the networkx library, 666 iterations were performed for each p_1 value, from $p_1 = 0.01$ to $p_1 = 1.0$. 666 iterations were done to achieve a confidence level of 99% and a confidence interval of 5*, giving the following results:



The phase change, where the giant component disappears, happens as p_1 grows larger than 0.6 .

* per http://www.surveysystem.com/sscalc.htm

```python
# Author: Donovan Guelde
# CSCI 5352 PS3 Question 3
# references: online documentation for numpy,
#               http://stackoverflow.com/questions/9754729/remove-object-from-a-list-of-objects-in-python
# Collaborators: None

import numpy as np
import os
import matplotlib.pyplot as plt
import networkx as nx

#fileName="test.txt"
fileName="../../CSCI5352_Data/karate_club_edges.txt"

def inModGroup(i,maxModularityGroups):
	if i in maxModularityGroups[0]:
		return 0
	if i in maxModularityGroups[1]:
		return 1
	if i in maxModularityGroups[2]:
		return 2

class Network:
	def __init__(self,fileName):
		self.groups = [] #empty array at instantiation, filled and updated as merges are performed
		self.n = 0 #number of nodes in network
		self.associationMatrix = self.readFile(fileName) #simple graph
		self.m = np.sum(self.associationMatrix)/2 #number edges
		self.regularization = (2*float(self.m)) #so we only have to calculate it once...
		self.eMatrix = self.get_e_matrix()
	def readFile(self,fileName):
		with open(fileName,'r') as f: #make 2d numpy array of appropriate size
			temp=-1
			lastNode=-1
			for line in f:
				line = line.split()
				temp = max(int(line[0]),int(line[1]))
				if (temp>lastNode):
					lastNode=temp #finds the highest numbered node
			associationMatrix = np.zeros((lastNode,lastNode))
			self.n = int(lastNode) #assumes no gaps in node labelling
			f.seek(0,0)
			lines = f.readlines()
			for line in lines:
				line = line.split()
				associationMatrix[int(line[0])-1][int(line[1])-1] = 1 #make it undirected...
				associationMatrix[int(line[1])-1][int(line[0])-1] = 1
			for index in range(0,self.n): #self.groups is a list of lists
				self.groups.append([[]]) #add empty list for every vertex in graph
				self.groups[index][0] = index+1 #place every vertex in its own group
		f.close()
		return (associationMatrix)

	def inGroup(self,i): #returns group that node i belongs to
```

```
            group=0
            node = i
            for index in range (0,len(self.groups)):
                if (node in self.groups[index]):
                    group=index
                    break
            return group

    def get_e_matrix(self): #updates e matrix (used after merge is performed)
            numberGroups=0
            for index in range(0,len(self.groups)):
                if (self.groups[index][0]): #if group has member/members
                    numberGroups+=1
            eMatrix = np.zeros((numberGroups,numberGroups))
            for r in range (0,numberGroups):
                for s in range (0,numberGroups):
                    tempSum=0.0
                    for i in range (0,self.n):
                        if(self.inGroup(i+1) == r):
                            for j in range(0,self.n):
                                if (self.associationMatrix[i][j]==1):
                                    if (self.inGroup(j+1) == s):
                                        tempSum+=1
                    if (tempSum!=0):
                        eMatrix[r][s]=(tempSum/self.regularization)
            return eMatrix

    def findDeltaQ(self,u,v): #returns delta Q between groups u and v
            a_u = np.sum(self.eMatrix[u])
            a_v = np.sum(self.eMatrix[v])
            return (2*(self.eMatrix[u][v]-(a_u*a_v)))

    def findGreatestDeltaQ(self): #returns (greatest delta Q,
            #                       index of first group to merge, index of second group to merge)
            deltaQ = (float('-inf'),0,0)
            for index in range(0,len(self.groups)):
                for index2 in range (index+1,len(self.groups)):
                    temp = self.findDeltaQ(index,index2)
                    if (deltaQ[0] < temp):
                        deltaQ = (temp,index,index2)
            return deltaQ

    def mergeGroups(self,r,s): #merge groups r and s into r, delete s
            r=int(r)
            s=int(s)
            for item in self.groups[s]:
                self.groups[r].append(item)
            del self.groups[s]
            return

    def getQ(self): #returns Q of network
            sum=0
            for index in range(0,len(self.groups)):
                sum2=0
                for index2 in range(0,len(self.groups)):
                    sum2+=self.eMatrix[index][index2]
                sum+=self.eMatrix[index][index]-pow(sum2,2)
            return sum
```

```
def main():
	graph = Network(fileName)
	counter=0
	maxModularity=float('-inf')
	modularity=np.zeros((graph.n))
	while (len(graph.groups)>1):
		temp=graph.findGreatestDeltaQ() #temp is a triple, (max delta Q, group to merge, group to merge)
		Q = graph.getQ()
		graph.mergeGroups(temp[1],temp[2])
		modularity[counter] = Q
		print Q
		print graph.groups
		counter+=1
		graph.eMatrix=graph.get_e_matrix()
		if (maxModularity<graph.getQ()):
			maxModularity=graph.getQ()
			length = len(graph.groups)
			maxModularityGroups=[[]]*length
			#print graph.getQ()
			for index in range(0,length):
				maxModularityGroups[index] = graph.groups[index]
	print maxModularity
	print maxModularityGroups

	plt.ylabel('Modularity Q')
	plt.xlabel('number of merges')
	plt.xlim(1,34)
	plt.title('Modularity of Zachary Karate Club as a Function of Merges\n (using Greedy Agglomerative Algorithm)')
	plt.plot(modularity)
	plt.savefig("karateMerge.jpg")

main()
```

```
# Author: Donovan Guelde
# CSCI 5352 PS3 Question 4
# references: online documentation for numpy and igraph
# Collaborators: None

import numpy as np
import os
import igraph

class Network:
	def __init__(self,fileName):
		self.n = 0
		self.gender=[]
		self.status=[]
		self.major = []
		self.vertexDegree=[]
		self.associationMatrix = self.readFile(fileName)
		self.g = igraph.Graph.Adjacency((self.associationMatrix == 1).tolist())
		self.m = np.sum(self.associationMatrix)/2.0
		self.regularization = 1/(2*self.m) #caluclate once
	def readFile(self,fileName):
		with open("./facebook100txt/"+fileName+"_attr.txt",'r') as f: #get n and attributes from _attr.txt
			counter=0
			for line in f:
```

```python
                counter+=1
            self.n = counter-1
            associationMatrix = np.zeros((self.n,self.n))
            self.gender = [0]*self.n #arrays to track gender, status, major and degree of vertexes
            self.status = [0]*self.n
            self.major = [0]*self.n
            self.vertexDegree = [0]*self.n
            f.seek(0,0)
            f.next() #skip the label row
            counter=0
            for line in f: #populate the attribute arrays
                line = map(int,line.split())
                self.gender[counter] = int(line[2]) #gender of vertexes where index=vertex
                self.status[counter] = int(line[1]) #ditto...
                self.major[counter] = int(line[3])
                counter+=1
        f.close()

        with open("./facebook100txt/"+fileName+".txt",'r') as f: #construct association matrix
            lines = f.readlines()
            for line in lines:
                line = line.split()
                associationMatrix[int(line[0])-1][int(line[1])-1] = 1
                associationMatrix[int(line[1])-1][int(line[0])-1] = 1 #make it undirected
        f.close()
        for index in range(0,self.n): #populate the vertex degree array,
            self.vertexDegree[index] = np.sum(associationMatrix[index],axis=0)
        return associationMatrix

    def getQ(self,attribute): #returns Q of network
        if attribute == "gender":
            membership = self.gender
        if attribute == "status":
            membership = self.status
        if attribute == "major":
            membership = self.major
        Q = igraph.Graph.modularity(self.g,membership)
        return Q

    def calculateAssortativity(self):
        assortativityCoefficient=igraph.Graph.assortativity(self.g,self.vertexDegree)
        return assortativityCoefficient

def main():
    plotArray=np.empty((100,2)) #an array of points to plot
    nameArray=[""]*100 #array to hold names of schools where [index] corresponds to plotArray[index]
    nextUniversity = [2]
    lastUniversity = [2]
    genderModularity=[""]*100
    statusModularity=[""]*100
    majorModularity=[""]*100
    vertexAssortativity=[""]*100
    names=[""]*100
    nValues = [""]*100
    counter=0
    for file in os.listdir("./facebook100txt/"):
        if (file != ".DS_Store"):
            nextFile, fileExtension = os.path.splitext(file)
```

```
            nextUniversity = nextFile.split('_')
            if (str(nextUniversity[0]) != str(lastUniversity[0])):
                nextGraph = Network(nextUniversity[0])
                names[counter]=nextUniversity[0]
                nValues[counter]=nextGraph.n
                genderModularity[counter]=nextGraph.getQ("gender")
                statusModularity[counter]=nextGraph.getQ("status")
                majorModularity[counter]=nextGraph.getQ("major")
                vertexAssortativity[counter]=nextGraph.calculateAssortativity()
                nameArray[counter] = str(nextUniversity[0])
                lastUniversity=nextUniversity
                counter+=1
    with open("./results/genderModularity.txt","w") as f:
        np.savetxt(f,genderModularity,fmt='%s')
    f.close()
    with open("./results/statusModularity.txt","w") as f:
        np.savetxt(f,statusModularity,fmt='%s')
    f.close()
    with open("./results/majorModularity.txt","w") as f:
        np.savetxt(f,majorModularity,fmt='%s')
    f.close()
    with open("./results/vertexAssortativity.txt","w") as f:
        np.savetxt(f,vertexAssortativity,fmt='%s')
    f.close()
    with open("./results/names.txt","w") as f:
        np.savetxt(f,names,fmt='%s')
    f.close()
    with open("./results/nValues.txt","w") as f:
        np.savetxt(f,nValues,fmt='%s')
    f.close()

main()
```

```
# Donovan Guelde
# CSCI 5352
# EC #6
# references: networkx online documentation, https://docs.python.org/2/library/functions.html#max

import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random

DOPART1=1
DOPART2=1
#calculate mean fractional size per part 1, Q6
if (DOPART1):
    NUMBERITERATIONS=2663 #confidence level-99, confidence interval-2.5
    resultsArray = np.zeros(NUMBERITERATIONS)
    for index in range(0,NUMBERITERATIONS):
        degreeSequence=[0]*1000
        degreeSequence[0]=1
        while (not nx.is_valid_degree_sequence(degreeSequence)):
            for index2 in range (0,1000): #generate degree sequence
                rollTheDice=random.random()
                if (rollTheDice<=.6):
                    degreeSequence[index2]=1
                else:
```

```
                                        degreeSequence[index2]=3
                        graph = nx.configuration_model(degreeSequence)
                        resultsArray[index] = len(max(nx.connected_component_subgraphs(graph),key=len))
                print "mean of largest component after",NUMBERITERATIONS,"iterations =",np.mean(resultsArray)
                print "mean fractional size =",np.mean(resultsArray)/1000



if (DOPART2):
        NUMBERITERATIONS=666 #confidence level-99, confindence interval-5
        resultsArray = np.zeros(100)
        p1Values = np.zeros(100)
        p1=.01
        counter=0
        while (p1<=1.01):
                iterationResultsArray = np.zeros(NUMBERITERATIONS)
                for index in range(0,NUMBERITERATIONS):
                        degreeSequence=[0]*1000
                        degreeSequence[0]=1
                        while (not nx.is_valid_degree_sequence(degreeSequence)):
                                for index2 in range (0,1000): #generate degree sequence
                                        rollTheDice=random.random()
                                        if (rollTheDice<=p1):
                                                degreeSequence[index2]=1
                                        else:
                                                degreeSequence[index2]=3
                        graph = nx.configuration_model(degreeSequence)
                        iterationResultsArray[index] = len(max(nx.connected_component_subgraphs(graph),key=len))
                resultsArray[counter]=np.mean(iterationResultsArray)/1000
                p1Values[counter]=p1
                p1+=.01
                counter+=1

        plt.ylabel('mean fractional size of largest component')
        plt.xlabel('p1')
        plt.title('Mean Fractional Size of Largest Component as a Function of p1')
        plt.xlim(.01,1)
        plt.plot(p1Values,resultsArray)
        plt.savefig("meanFractionalSize.png")
        plt.close()
```

```
# Donovan Guelde
# CSCI 5352 PS3
# Plotter for PS3, Q4

import numpy as np
import matplotlib.pyplot as plt

def readData(fileName):
        counter=0
        with open(fileName,'r') as f:
                for line in f:
                        counter+=1
```

```
        f.seek(0,0)
        data = np.zeros((counter)) # array to  hold data
        counter=0
        maxDataPoint = float('-inf')
        maxIndex=0
        minDataPoint = float('inf')
        nimIndex=0
        for line in f:
            data[counter] = float(line)
            if data[counter] > maxDataPoint:
                maxDataPoint=data[counter]
                maxIndex=counter
            if data[counter] < minDataPoint:
                minDataPoint=data[counter]
                minIndex=counter
            counter+=1
    return data, int(maxIndex), int(minIndex)

def readNames(fileName):
    counter=0
    with open(fileName,'r') as f:
        for line in f:
            counter+=1
        f.seek(0,0)
        data = [""]*counter
        counter=0
        for line in f:
            data[counter] = line
            counter+=1
    return data
def getData(attribute):
    if attribute == "major":
        data, maxPoint, minPoint = readData("./results/majorModularity.txt")
    if attribute == "status":
        data, maxPoint, minPoint = readData("./results/statusModularity.txt")
    if attribute == "vertex":
        data, maxPoint, minPoint = readData("./results/vertexAssortativity.txt")
    return data,int(maxPoint),int(minPoint)



def main():

    nArray,nMax,nMin = readData("./results/nValues.txt")
    names = readNames("./results/names.txt")
    attributes = ["major","status","vertex"]

    for item in attributes:
        # scatter plots
        data,maxPoint,minPoint = getData(item)
        if item == "vertex":
            plt.ylabel('Vertex Degree Assortativity')
        else:
            plt.ylabel(item+" modularity")
        plt.xlabel('Network Size, n')
        plt.title('Network Size vs '+item)
        plt.scatter(nArray,data)
        if item=='major':
```

```
            plt.ylim(-.01,.14)
        plt.text(float(nArray[maxPoint]),float(data[maxPoint]),names[maxPoint])
        plt.plot(float(nArray[maxPoint]),float(data[maxPoint]),'o',mfc='red')
        plt.text(float(nArray[minPoint]),float(data[minPoint]),names[minPoint])
        plt.plot(float(nArray[minPoint]),float(data[minPoint]),'o',mfc='red')
        plt.text(float(nArray[nMax]),float(data[nMax]),names[nMax])
        plt.plot(float(nArray[nMax]),float(data[nMax]),'o',mfc='red')
        plt.text(float(nArray[nMin]),float(data[nMin]),names[nMin])
        plt.plot(float(nArray[nMin]),float(data[nMin]),'o',mfc='red')
        plt.axhline(0,linestyle='dashed')
        plt.xscale('log')
        plt.savefig(item+".jpg")
        plt.clf()
        plt.close()

        # histograms
        plt.hist(data)
        if item=='status':
            plt.xlim(-.05,.30)
        if item=='major':
            plt.xlim(-.01,.14)
        plt.title('Histogram of '+item+' Assortativity')
        plt.xlabel('Assortativity ('+item+')')
        plt.ylabel('Frequency')
        plt.axvline(0,linestyle='dashed', color='red')
        plt.savefig(item+"Hist.jpg")
        plt.clf()
        plt.close()

main()
```