**Introduction:**

In Lab 6 we were tasked in finding the most important vertices of a graph created from a text file. To do this we have to implement some algorithms that find the probability of visiting vertices and display those vertices, once we determine the most important, we remove it and re run the algorithms to find the second most important.

**Proposed Solution Design and Implementation:**

Method randomWalk(G,steps):

This method given a adjacency list graph and the amount of steps finds the probability of visiting each vertice in the graph and returns that list. First it creates a list the size of the amount of vertices then we select a random vertices in that list, then we create a visited list that keeps track of every time we visit a vertices and an empty list called N. then we iterate using a for loop steps amount of times and we add one to the visited list the vertices we visit, then we append all the vertices connected to the one were on to the list N and make a random selection in that list and set it equal to v then we clear the list N and do it over again. Once were done iterating we create a list by dividing the visited list by the amount of steps we took and round every number in the list to 6 decimals and return the list. The random walk time runs in O(n) n being the amount of steps, but the assumption is made that the amount of steps is much larger then the size of the list

Method iterativeP(G);

This method receives a adjacency matrix and returns the probability list. First it creates a list the size of the amount of vertices in the graph and fills that with the probability of picking any of the vertices(1/v) then we create the transition matrix which is the same size as the graph, then for every element in T we calculate the out degree of the vertices of the graph if the out degree is zero we set everything in that vertices connection to 1/v if its not we set all the edges to 1/out degree. After the transition matrix is created we have to find the dot product of the probability matrix and the transition matrix and keep doing that, I decided to iterate 1000 times I round the probability list to 6 digits and return it. This Method runs in O(N^2) because we have to loop through every element in the array n being the number of vertices in the graph

Method createGraphFromText(text,T):

I decided to create a method to create a graph from a text file it takes in a type T and text as a string. Then I use the python .split function to turn the function into a list once it’s a list we check if t is a ajececny list or matrix and depending on which it is it creates a graph the size of the last element of the list which is the last vertice then we iterate through the length of the list by 2 and use the insert\_edge function in either graph and insert the I as the source and the i+1 as the destination for the edge. Then we return g. This method runs in O(n/2) n being the amount of edges in the file.

Method outdegree(G,v)

This method takes in a graph and vertice and returns the out degree of the vertice.This method runs in O(N)

Method removeInfluence(G,v)

This method given a graph and vertieces removes all the influence of that vertice on the graph. First we have to figure out the type of the graph then for the length of either graph we usre the delete edge function for either type of graph. This Method runs in O(n) being the number of vertices in the graph because we have to remove every edge connecting to vertiec v

Method findVimp(G,iters):

This method given a graph and a number of iterations prints the most important vertice with the probability of visiting that vertices for that many iterations. First it looks at what type of graph it has if it’s an ajecency list it sets p equal to the random walk method for 10000 steps, if its an ajecency matrix we set p equal to the iterativeP method. Then for every iteration we loop through all the vertices and find the greatest probability in the list and set v to it. Print the iteration number vertice and the probability use the removeInfluence method of the graph and vertice, then re run either the random walk or iterativeP algorithims.This method is highly dependent on the amount of iterations and which algorithim you decide to do, but not counting the other algorithims it runs in O(n)

Method Main:

The main method is pretty simple, first we open and read the file, use the createGraphFromText method and make both a adjacency list and matrix then run the findVimp method on both of those.

**Experimental results:**

This lab was a lot easier due to the fact that we were given the pseudo code for the algorithm thus the problem of actually coding them was substantially easier and I didn’t really have any problems with it, However I did have some problem removing influence as I was planning and creating a method to remove the edges and vertices from the graph but I realized that both graphs had a delete edge function that made it much simpler. Also some trouble was getting the text file into a list that we can then manipulate into graphs I had tested out a couple different methods but the best one was to use the python function .split() which splits a string at line breaks and spaces and turns the results into a list. Also for testing of all the methods I used much smaller graphs that way debugging was much easier and problems would occur because I would try to remove edges of graphs that no longer had any connections, I realized that these problems wouldn’t occur in the actual running of the program with huge graph sizes but the program would have to be adjusted for graphs of smaller sizes.

**Conclusions:**

This lab was essential to me in learning to modify and work with graphs, as well as the many uses of graphs in real life, as something like this program is an essential part to search engines and other things. This also shows that one problem can have totally different solotuins and different types of algorthims, the random walk being a randomization algorithim and the iterative being computational.

Appendix:

import numpy as np

import matplotlib.pyplot as plt

import math

import random

import graph\_AM as AMgraph

import graph\_AL as ALgraph

def removeInfluence(G,v):

#Given a graph and vertice removes all influence of vertice v in graph g

if type(G) is ALgraph.Graph:

for i in range(len(G.al)):#deletes the edges of al graph

G.delete\_edge(i,v)

else:

for i in range(len(G.am)):

G.delete\_edge(i,v)#deltes the edges of an am graph

def findVimp(G, iters):

#Given a graph and how many iterations prints out the most important vertice in the graph after each iteration

if type(G) is ALgraph.Graph:#figures out the type of the grapj

p = randomWalk(G,100000)

print("Using random walk method and adjacency list representation")

else:

p = iterativeP(G)

print("Using iterative method and adjacency matrix representation")

v = 0

for j in range(iters):

for i in range(len(p)):#loop used to figure out gratest vertive

if p[i] > p[v]:

v = i

print("iteration", j,"most important vertex:",v,"with p =", p[v])

removeInfluence(G,v)#removes the influence of the most important vertie

if type(G) is ALgraph.Graph:

p = randomWalk(G,10000)

else:

p = iterativeP(G)

v = 0 #resets verice

def createGraphFromText(text, T):

#gets a string representing a textfile and a type of tragph called T and creates a graph with the file based on the type

E = list(text.split())#turns string into list

if T is ALgraph.Graph:

g = ALgraph.Graph(int(E[-1])+1) # creates graph for al graph

else:

g = AMgraph.Graph(int(E[-1])+1)#creates graph for am graph

for i in range(0,len(E),2):

g.insert\_edge(int(E[i]),int(E[i+1]))#inserts edges into the graph

return g

def outDegree(G,i):

#given graph and vertice i returns the out degree of i

count = 0

for j in range(len(G.am[i])):

if G.am[i][j] != -1:

count += 1

return count

def randomWalk(G,steps):

#calculates the probability of visiting vertices of graph G in n number of steps

V = []

for i in range(len(G.al)):

if G.al[i] != []:

V.append(i)

v = random.choice(V)

visited = np.zeros(len(G.al))

N = []

for i in range(steps):

visited[v] += 1

for e in G.al[v]:

N.append(e.dest)

if not N:

N = V

u = random.choice(N)

v = u

N.clear()

p = visited/steps

p = [round(num, 6) for num in p]

return p

def iterativeP(G):

#calculates the probability of vesiting each matrix given graph G as a ajacency matrix

p = np.zeros(len(G.am))

p = p + 1/len(G.am)

T = np.zeros((len(G.am),len(G.am)))

for i in range(len(T)):

out = outDegree(G , i)

if out == 0:

for j in range(len(T[i])):

T[i][j] = 1/len(G.am)

for j in range(len(T[i])):

if G.am[i][j] != -1:

T[i][j] = 1/out

for i in range(1000):

p = np.dot(p,T)

p = np.round(p,5)

return p

if \_\_name\_\_ == "\_\_main\_\_":

s = open('facebook\_combined.txt', 'r', encoding = "utf8")# opens and reads the facebook combined text

text = s.read() #reads the text

G = createGraphFromText(text,ALgraph.Graph) # creates a graph for al case

g = createGraphFromText(text,AMgraph.Graph)# creates a graph for am case

s.close()

plt.close("all")

findVimp(g,10)#finds the most important vertice over 10 iterations

findVimp(G,10)

**I, Estevan Ramos, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.**