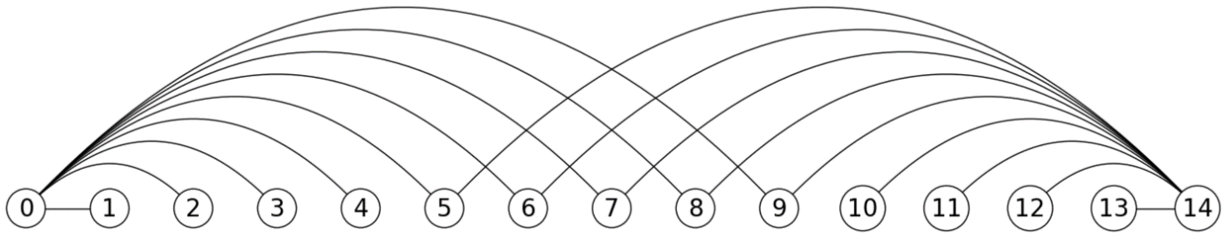
Lab V

Finding the Most Influential Node using Simulation and Iteration



Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

Introduction

The objective of this laboratory is to create a program which reads a text document that has the source and destination of a node and then create a graph in an adjacency list and adjacency matrix representation. Having that said, the main purpose of this program will be to find the most influential node among the graph, then we will remove the influence of the most important node to get the second most important node.

The algorithms that will be used are the random walk and an iterative approach. The following methods are going to be used to achieve the lab:

- ***Build_graph():***
  - reads the text document "Facebook_combined.txt" and returns a graph.
- ***Random_walk(G,steps):***
  - Simulation algorithm that selects a random vertex v in V, then "walk" from v to a random neighbor u, then "walk" again to a neighbor of u and so on.
  - Returns a list p which contains the probability of each vertex to be visit at any given time.
- ***Neighbors(E):***
  - Creates a list containing all the neighbors of v.
- ***Iterative_p(G,convergence):***
  - Iterative algorithm that performs elementary probabilistic computation to find which vertex has the highest probability to be visit.
  - Returns a list p which contains the probability of each vertex to be visit at any given time.
- ***Out_degrees(G)***
  - Returns a list containing the edges that are going out from v in V.
- ***Transition_matrix(G)***
  - Returns a 2d array containing the probability of going from vertex i to vertex j.
- ***Remove_influence(G,Vimp)***
  - Method that removes all neighbor u in v to remove the influence of the most important vertex.
- ***Get_info(G,v)***
  - Returns the largest value and its index in p.

Proposed Solution, Design and Implementation

Build_graph()

First, I started creating this method because this is the one that creates the graph. Although, for simplicity's sake I made two methods, one that creates the graph in adjacency list representation and the other creates the graph in adjacency matrix representation.

Both methods (AL/AM) follow the same steps:

- First, we create an empty graph.
- Second, we create a variable that reads the text document.
- Third, we create a for loop that visits each line of the text document and for when there is no longer a line to read.
- Fourth, within this for loop I use the split method to create an array in which the first element is the source and the second element is the destination.
- Lastly, call the insert_edge () method to add the edges to the graph.

Random_walk(G,steps):

This method receives a graph in adjacency list representation and the amount of steps that we will walk in G. The purpose of this method is to pick a random vertex v in G, then "walk" randomly to one of its neighbor u, then walk again to a random neighbor of a neighbor of u and so on.

To create this method we must follow the following steps:

- First we must create a variable v which will choose a random vertex in G.
- Second, we will create a visited list which will have the purpose of storing how many times a vertex has been visited.
- Third, we will create a for loop which determines how many steps we will take from one neighbor to another.
- Fourth, within this for loop we will increase by one the vertex that we are visiting.
- Fifth, we will call the neighbors method to return a list with the neighbors of v and it will be stored in the variable N.

- Sixth, we will make a condition which will be true if vertex v has no neighbors, that is, it has no influence, we will create a list of all vertices of G and it will be stored in N.
- Seventh, we will pick a random (neighbor) vertex of N and store it in u.
- Eighth, we will store the vertex of u in v.
- Finally, when we have finished walking we will divide all the content of the visited list with the steps we gave, this will be stored in p. Note that p will then be a probability list that we will visit a certain vertex in p at any time.

Neighbors(E):

This method receives a reference to a vertex in G and the purpose of this method is to return a full list of all the neighbors of v.

To create this method we must do the following:

- First, we will create an empty list.
- Second, we will do a for loop which will visit all the edges in E.
- Third, we will depend on E.'s neighbors.
- Finally, we will return the method.

Iterative_p(G,convergence):

This method receives a graph in adjacency matrix representation and the total number of times that we are going to converge. The purpose of this method is to achieve similar results to the "Random Walk" simulation algorithm by performing elementary probabilistic computations. In other words, this method will do a repetitive multiplication of the probability vector (or 1D matrix) p and a transition matrix T

To create this method we must do the following:

- First, we create size variable called “size” that stores the size of the graph so that the code is easy to read and organized.
- Second, we create a list “p” that stores the initial probability of all vertices, by this I mean that all vertices have a 1/size probability of being visited (regardless of their influence).

- Third, we create a variable “T” that stores an array of probabilities of going from vertex i to vertex j. Note: I created a helper method to make the iterative_p method look cleaner and easier to read and understand.
- Fourth, we create a for loop which will repeat until convergence and within this for loop a matrix multiplication p = p * T will be done.
- Return the list p which is the probability that we will visit a certain vertex in p at any time.

Transition_matrix(G):

This method receives a graph representing matrix adjacency and its purpose is to create a transition matrix where T[i,j] is the probability of going from vertex i to vertex j.

To create this method we must do the following:

- First, we create a variable that stores the size of the graph to make the method look cleaner and more readable.
- Second, we will create a 2d array of floats called "T" that contains zeros.
- Third, we will create a list called "out" that contains the edges that come out of all v's in G.
- Fourth, we will create two for loops, one will visit vertex i and the other will visit vertex j (nested loop).
- Fifth, within the nested loop we will create three conditions:
    - The first condition will be true if vertex i has no influence, that is, it has no neighbor / edge, in which case we will make T [i, j] equal to 1/size.
    - The second condition will be true if there is an edge that goes from i to j, in which case we will make T [i, j] equal to 1/out.
    - The last condition will be true if there is no vertex from i to j, in which case T [i, j] is equal to 0.
- Finally, we will return T that vertex i will go to vertex j at any given time.

Out_degrees(G):

This method receives a graph in adjacency matrix representation and returns a list that contains the edges that come out of all v's in G.

To create this method we must do the following:

- First, we will create a list "L" that will contain zeros and will have the purpose of storing the out degrees of all vertices in G.
- Second, we will make two for loops: one that visits vertices i and a nested for loop that visits vertices j.
- Third, we will create a condition that will be true if there is an edge that goes from i to j, in which case we will increment by one to the outdegree of vertex i (L[i] += 1).
- Finally, we will return L.

Remove_influence(G,Vimp):

This method receives a graph and a reference to the most important vertex Vimp and its purpose is to remove the influence of Vimp so that we can find the second most important Vimp in the next computations. This method is divided into two methods, this because the procedure/steps to delete the edges in the two representations (AL/AM) is different.

To remove the edges using a graph in AL we must do the following:

- First, we create a while loop that will stop until Vimp has no neighbors. Note: I used a while loop because when I tried to delete the edges with a for loop only half of the edges were erased, this is because when you erased an edge it acts like the pop method from queues and stacks.
- Finally, I called the method delete_edge(Vimp,G[i][0]) where Vimp is the source and G[i][0] is the destination.

To remove the edges using a graph in AM we must do the following:

- First, we create a for loop that will stop until we have visit all the edges/neighbors in Vimp
- Finally, we create a condition which will be true if there is an edge going from Vimp to i, then we will call the method delete_edge(Vimp,i) where Vimp is the source and I the destination.

Get_info(p):

This method receives a list p of probabilities of visiting vertex i at any time and its purpose is to return the largest element in p and its index.

To create this method I used the following numpy functions:

- Np.argmax(p) → Returns the index of the largest element in p
- Np.amax(p) → Returns the largest element in p.

Time Complexity

Build graph Method (AL/AM)	O(N)
Get info (AL/AM)	O(N)
Remove Influence (AL)	O(|E|)
Remove Influence (AM)	O(|E|)

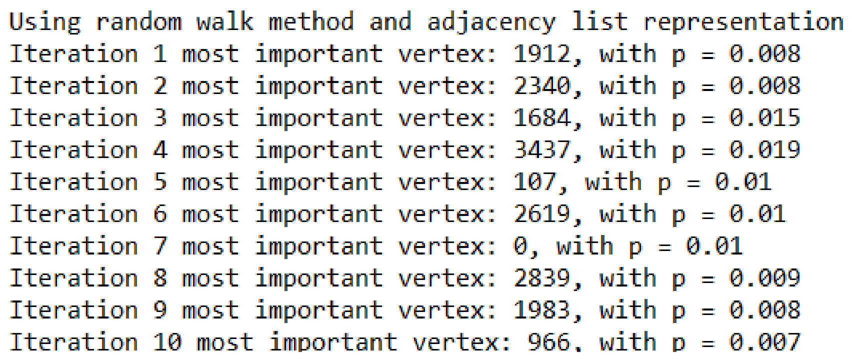
Random Walk Algorithm	
Neighbors	O(|E|)
Random Walk	O(N)

Iterative Algorithm	
Out Degree	O(|V|+|E|)
Transition Matrix	$O(V^2)$
Iterative_p	O(N)

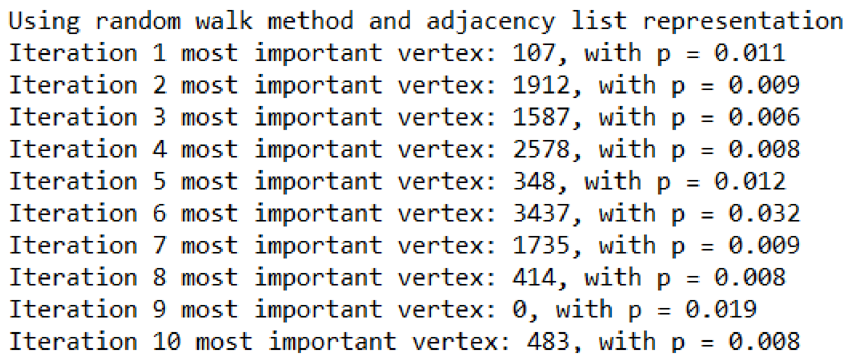
Experimental Results

Random Walk

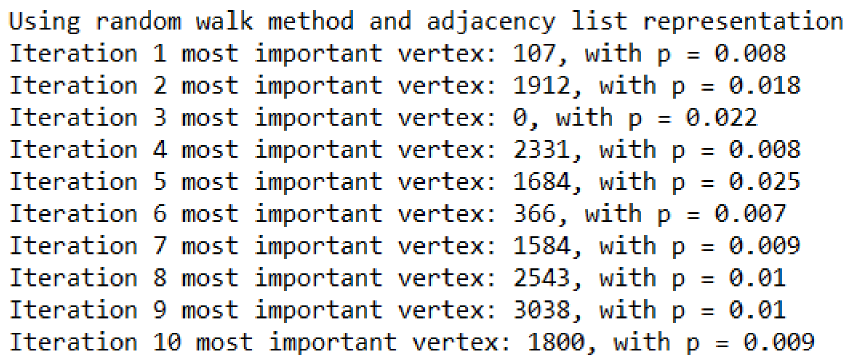
As expected, the outputs of the "random walk" simulation algorithm will be different each time we run the code, this is because this algorithm is based on choosing a random vertex constantly, so the results will not always be the same. Even so, it should be noted that although the algorithm is based on randomness, it is capable of obtaining (at least half) the most important vertices of the graph.



```
Using random walk method and adjacency list representation
Iteration 1 most important vertex: 1912, with p = 0.008
Iteration 2 most important vertex: 2340, with p = 0.008
Iteration 3 most important vertex: 1684, with p = 0.015
Iteration 4 most important vertex: 3437, with p = 0.019
Iteration 5 most important vertex: 107, with p = 0.01
Iteration 6 most important vertex: 2619, with p = 0.01
Iteration 7 most important vertex: 0, with p = 0.01
Iteration 8 most important vertex: 2839, with p = 0.009
Iteration 9 most important vertex: 1983, with p = 0.008
Iteration 10 most important vertex: 966, with p = 0.007
```



```
Using random walk method and adjacency list representation
Iteration 1 most important vertex: 107, with p = 0.011
Iteration 2 most important vertex: 1912, with p = 0.009
Iteration 3 most important vertex: 1587, with p = 0.006
Iteration 4 most important vertex: 2578, with p = 0.008
Iteration 5 most important vertex: 348, with p = 0.012
Iteration 6 most important vertex: 3437, with p = 0.032
Iteration 7 most important vertex: 1735, with p = 0.009
Iteration 8 most important vertex: 414, with p = 0.008
Iteration 9 most important vertex: 0, with p = 0.019
Iteration 10 most important vertex: 483, with p = 0.008
```

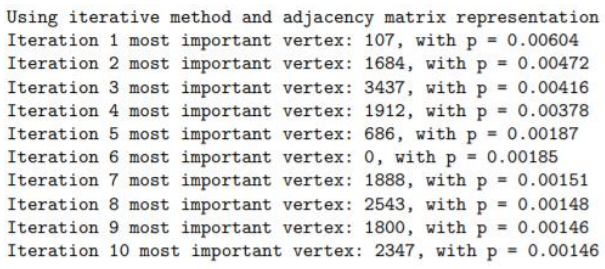


```
Using random walk method and adjacency list representation
Iteration 1 most important vertex: 107, with p = 0.008
Iteration 2 most important vertex: 1912, with p = 0.018
Iteration 3 most important vertex: 0, with p = 0.022
Iteration 4 most important vertex: 2331, with p = 0.008
Iteration 5 most important vertex: 1684, with p = 0.025
Iteration 6 most important vertex: 366, with p = 0.007
Iteration 7 most important vertex: 1584, with p = 0.009
Iteration 8 most important vertex: 2543, with p = 0.01
Iteration 9 most important vertex: 3038, with p = 0.01
Iteration 10 most important vertex: 1800, with p = 0.009
```

Iterative Algorithm

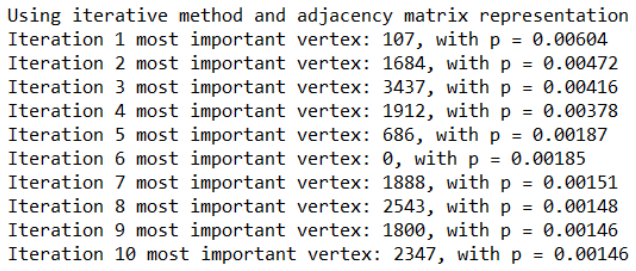
Here are two images, a sample of the first inputs of what should be in our code (this output was provided by Dr. Fuentes) and the other image shows the output of my program.

Expected Result



```
Using iterative method and adjacency matrix representation
Iteration 1 most important vertex: 107, with p = 0.00604
Iteration 2 most important vertex: 1684, with p = 0.00472
Iteration 3 most important vertex: 3437, with p = 0.00416
Iteration 4 most important vertex: 1912, with p = 0.00378
Iteration 5 most important vertex: 686, with p = 0.00187
Iteration 6 most important vertex: 0, with p = 0.00185
Iteration 7 most important vertex: 1888, with p = 0.00151
Iteration 8 most important vertex: 2543, with p = 0.00148
Iteration 9 most important vertex: 1800, with p = 0.00146
Iteration 10 most important vertex: 2347, with p = 0.00146
```

Actual Result



```
Using iterative method and adjacency matrix representation
Iteration 1 most important vertex: 107, with p = 0.00604
Iteration 2 most important vertex: 1684, with p = 0.00472
Iteration 3 most important vertex: 3437, with p = 0.00416
Iteration 4 most important vertex: 1912, with p = 0.00378
Iteration 5 most important vertex: 686, with p = 0.00187
Iteration 6 most important vertex: 0, with p = 0.00185
Iteration 7 most important vertex: 1888, with p = 0.00151
Iteration 8 most important vertex: 2543, with p = 0.00148
Iteration 9 most important vertex: 1800, with p = 0.00146
Iteration 10 most important vertex: 2347, with p = 0.00146
```

As can be seen, the results of the iteration algorithm are identical to that of Dr. Fuentes. The reason this happened is simple, this algorithm, unlike the random walk algorithm, computes the problem in a more precise and exact way and does not use the random element at all. In other words, elementary probabilistic calculations were used to solve the problem.

Conclusion

In conclusion, this laboratory helped me to practice my knowledge and understanding about graphs. Also, when I finished the laboratory I realized how interesting and efficient this data structure is, because we were able to go into detail and understand how the algorithms of Facebook or any social network really work to find the most popular user. On the other hand, I was able to identify a big difference between the simulation algorithm and the iterative one, this difference is that the simulation algorithm solves the problem almost instantly but its results are not so accurate while the iterative approach solves the problem in a very long time but it is accurate. This is because the simulation algorithm is based on randomness, while the other uses rules and mathematical formulas. Lastly, I never believed that a problem of this type could be solved using randomness, much less thought that it would be faster than one that uses precise mathematical elements.

Academic Honesty Certification

I 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.

[illegible]

Oswaldo Escobedo

Appendix

```
#   Course: CS 2302
#   Assignment: Lab VI
#   Author: Oswaldo Escobedo
#   Instructor: Dr. Fuentes
#   TA: Harshavardhini Bagavathyraj
#   Date of Last Modification: 04/27/2020
#   Purpose of the Program: To implement two algorithms to identify
#   the most influential node.

import numpy as np
import matplotlib.pyplot as plt
import math
import random
import graph_AL as AL
import graph_AM as AM

#%% Creation of Graphs in AL and AM

def build_graphAM():
    g_am = AM.Graph(4039)
    f = open('facebook_combined.txt','r')
    for line in f: # Iterates until there is no line left to read
        v = line.split(' ') # Array of size 2 containing source and dest
        g_am.insert_edge(int(v[0]),int(v[1]))
    f.close()
    return g_am

def build_graphAL():
    g_al = AL.Graph(4039)
    f = open('facebook_combined.txt','r')
    for line in f: # Iterates until there is no line left to read in the document
        v = line.split(' ') # Array of size 2 containing source and dest
        g_al.insert_edge(int(v[0]),int(v[1]))
    f.close()
    return g_al


#%% Random Walk Algorithm Section

def neighbors(E):
    L = []
    for i in E:
        L.append(i.dest) # Appends neighbors
    return L

def random_walk(G,steps=1000):
    v = random.randint(0,len(G.al)) # Picks a random vertex
    visited = [0] * len(G.al) # Contains how many times have we visit vertex v
    for i in range(steps):
        visited[v] += 1 # Vertex v has been visited
        N = neighbors(G.al[v])
        if len(N) == 0: # Vertex v has no neighbors
            N = np.arange(0,len(G.al)) # List that contains all V in G
        u = random.choice(N) # Randomly choose a neighbor
        v = u # We change vertex v to be neighbor u, this to traverse the graph
    p = np.divide(visited,steps)
    return p # Returns a list of probabilities that we will visit i.

def get_info_rw(p):
    return np.argmax(p),np.amax(p) # Returns the largest value and its index.

def remove_influence_rw(G,v):
    while len(G.al[v]) != 0:
        G.delete_edge(v,G.al[v][0].dest) # Deletes neighbors from Vimp to remove influence


#%% Iterative Algorithm Section

def out_degrees(G):
    L = [0 for i in range(G.am.shape[0])]
    for v in range(G.am.shape[0]):
        for e in range(G.am.shape[1]):
            if G.am[v][e] != -1: # Vertex has an edge from v to u
                L[v] += 1
    return L  # Returns a list containing the outdegrees of all vertices

def transition_matrix(G):
    size = len(G.am)
    T = np.zeros((size,size),dtype=np.float64) # Matrix that contains zeros of size 4039x4039
    out = out_degrees(G)
    for i in range(G.am.shape[0]):
        for j in range(G.am.shape[1]):
            if out[i] == 0: # Vertex i has no neighbors and influence
                T[i,j] = 1/size
            elif G.am[i,j] != -1: # There is an edge going from i to j
                T[i,j] = 1/out[i]
            else: # There is no edge going from i to j
                T[i,j] = 0
    return T # Returns a matrix of probabilities tat vertex i will go to vertex j

def iterative_p(G,convergence=1000):
    size = len(G.am)
    p = np.array(size*[1/size]) # List that contains the initial probability of all v's in V
    T = transition_matrix(G) # Matrix containing probabilities of going from vertex i to vertex j
    for i in range(convergence): # Iterates until the probablities ov visiting vertex i are exact
        p = np.dot(p,T)
    return p

def get_info_ip(p):
    return np.argmax(p),round(np.amax(p),5) # Returns the largest value and its index

def remove_influence_ip(G,v):
    for i in range(G.am.shape[1]):
        if G.am[v][i] != -1:
            G.delete_edge(v,i) # Deletes neighbors/influence from Vimp


#%% Main Method
if __name__ == "__main__":

    plt.close("all")

    g_al = build_graphAL()
    g_am = build_graphAM()

    print('\nUsing random walk method and adjacency list representation')
    for i in range(10):
        p = random_walk(g_al)
        V_imp,val = get_info_rw(p)
        print('Iteration {} most important vertex: {}, with p = {}'.format(i+1,V_imp,val))
        remove_influence_rw(g_al,V_imp)

    print('\nUsing iterative method and adjacency matrix representation')
    for i in range(10):
        p = iterative_p(g_am)
        V_imp,val = get_info_ip(p)
        print('Iteration {} most important vertex: {}, with p = {}'.format(i+1,V_imp,val))
        remove_influence_ip(g_am,V_imp)
```