Degrees of Separation in a Social Circle of Facebook

# Data:

The data set is from the Stanford data set and contains data about social groups. The data was collected anonymously, and all identifying values were converted to be anonymous. The data is broken up into nodes of features, circles, and ego networks. This dataset will represent the connectivity of approximately 4000 people who use the social platform. There are approximately 88,000 edges in this platform. Since the connections in this data set are anonymous, we cannot determine what that represents but who is connected by political affiliation.

# Research Question:

The purpose of this project is to examine the connectivity of Facebook social groups. Specifically using degrees of separation and statistics to determine how connected the people are.

# Code:

Imports:

Text

Description automatically generated

This includes the necessary imports as well as the setup for the modules which will be covered later.

Main:

Text

Description automatically generated

This is the main method of my program. The first part calls the “read\_file” function and sets up what the maximum nodes are in the function. Using this information, I created the adjacency list by pushing in the values from the data set into an adjacency graph format. Next is to set up a vector to take in the distances and a tuple which contains the distance value and the start and end point for that distance. Once that is done, I called the “get\_distances” function to make edits and use the degrees of freedom function. Next, I created a “remove\_zero\_distances” which removed all the times the program ran having a same start and end person. This was needed to pass into the average and median functions.

get\_distances function:

Text

Description automatically generatedThe “get\_distances” function takes in the distances vector, tuple, and adjacency list. This function loops through every possible starting point and then from that starting point to all of the possible ending points after. The function then calls the “degrees” function which will return an option of either Some(distance) or None. This value will then be pushed into both the “distances” and “start\_end\_vec” tuple. Since “None” cannot be pushed into the vector it is just not added unless it is a 0 to start\_end\_vec.

Degrees Function:

Text

Description automatically generated

This is the “degrees” function which takes in the adjacency list, start point and end point while returning an option. First thing is to import and establish the queue for the push and pop system we will use later. Next, we need to make a vector of all the points we have visited or not. The vector is populated with all false for the length of the adjacency list. Then we set the start point in the visited vector to be True. Then we pushed into the back of the queue the start point and a distance of 0. Then is a while loop which acts kind of like a base case for a recursive function. As soon as the node is equal to the endpoint we return Some of the distance. If that is not the case, then find the next neighbor in the adjacency list and visit that point and put it into the queue, and add 1 to the distance. This continues until the function reaches the end point or it will return None.

Read\_file Function:

Text

Description automatically generated

The “read\_file” function is pretty simple, it just reads the file and splits the csv. Then it puts the start point and all of its connections into a tuple and returns it.

Remove\_zero function:

Graphical user interface, text

Description automatically generated

This function is simple as well, it returns a vector of the distances but without the 0s. This is needed so that the average and median functions aren’t influenced by distances that go from the same start to endpoints.

The next few functions are mathematical functions to analyze the distances and degrees of separation.

# Output:

This is the output:

Text

Description automatically generated

This is what the final program outputs. The functions from before take in the distance functions or tuple with distances and start and endpoints. The average distance was 4 and the median was also 4. There was a distance greater than 6 which means it fails the 6 degrees of separation test. The maximum distance was 17 and went from start point 498 to 850. The average and median distance being 4 means that most of the distances fell under those 6 degrees of separation goal. The program runs in approximately 3 minutes and 20 seconds.

# References:

Chat GPT Usage:

My input:

Explain the process for degrees of separation.

Chat GPT Output:

The concept of degrees of separation is used to describe the number of steps it takes to connect any two individuals in a social network. The process for determining degrees of separation typically involves identifying the connections between individuals in the network and counting the number of steps it takes to link any two individuals.

Here is a general process for calculating degrees of separation:

1. Define the network: Determine the individuals or entities in the network and the relationships that connect them.
2. Identify the starting and ending points: Choose two individuals in the network that you want to connect.
3. Find the shortest path: Using a search algorithm, such as breadth-first search or depth-first search, find the shortest path that connects the two individuals. The algorithm will typically start at the starting point and explore the network until it reaches the ending point.
4. Count the steps: Count the number of steps or connections it takes to move from the starting point to the ending point along the shortest path. This number represents the degrees of separation between the two individuals.

For example, in a network of friends, if you want to determine the degrees of separation between yourself and a celebrity, you would start by identifying all of the people you know who may have a connection to the celebrity. You would then use a search algorithm to find the shortest path that connects you to the celebrity. The number of connections in the shortest path would represent the degrees of separation between you and the celebrity.

This helped me to further understand how degrees of separation works and give me an idea of pseudocode.

# Full Code:

use std::collections::VecDeque;

use std::fs::File;

use std::io::BufRead;

use std::io::BufReader;

mod my\_module;

use crate::my\_module::average;

use crate::my\_module::median;

use crate::my\_module::six\_degrees;

use crate::my\_module::max;

fn main() {

    let my\_list = read\_file("facebook\_combined.csv");

    let mut n = 4039;

    let mut adj\_list : Vec<Vec<usize>> = vec![vec![]; n];

    for (v,w) in my\_list.iter() {

        let v\_us = usize::try\_from(\*v).unwrap();

        let w\_us = usize::try\_from(\*w).unwrap();

        adj\_list[v\_us].push(w\_us);

    };

    let mut distances = Vec::new();

    let mut start\_end\_vec: Vec<(usize,usize,usize)> = Vec::new();

    get\_distances(&mut distances, &mut start\_end\_vec, &adj\_list);

    let mut remove\_zero\_distances = remove\_zero(&mut distances);

    average(&remove\_zero\_distances);

    median(remove\_zero\_distances.to\_vec());

    six\_degrees(&distances);

    max(&distances, start\_end\_vec);

}

fn get\_distances(distances: &mut Vec<usize>, start\_end\_vec: &mut Vec<(usize,usize,usize)>, adj\_list: &Vec<Vec<usize>>) {

    for start in 0..adj\_list.len() {

        for end in start..adj\_list.len() {

            if let Some(x) = degrees(&adj\_list, start, end) {

                let current\_distance = degrees(&adj\_list, start, end).unwrap();

                let mut start\_end = vec![current\_distance,start,end];

                start\_end\_vec.push((current\_distance, start, end));

                distances.push(current\_distance);

                //println!("The distance between {} and {} is {}", start, end, current\_distance);

            }

            else {

                //println!("There is no path from {} to {} ", start, end);

                start\_end\_vec.push((0, start, end));

            }

        }

    }

}

fn read\_file(path: &str) -> Vec<(usize, usize)> {

    let mut entries: Vec<(usize, usize)> = Vec::new();

    let file = File::open(path).expect("Could not open file");

    let buf\_reader = std::io::BufReader::new(file).lines();

    for line in buf\_reader {

        let line\_str = line.expect("Error reading");

        let v: Vec<&str> = line\_str.trim().split(',').collect();

        if v.len() == 1{

                continue;

        }

        let x = v[0].parse::<usize>().unwrap();

        let y = v[1].parse::<usize>().unwrap();

        entries.push((x, y));

    }

    return entries;

}

fn degrees(adj\_list: &Vec<Vec<usize>>, start: usize, end: usize) -> Option<usize> {

    let mut queue = VecDeque::new();

    let mut visits = vec![false; adj\_list.len()]; // creates vector of visited points

    //setting the start point to be visited and in queue

    visits[start] = true;

    queue.push\_back((start, 0));

    while let Some((node, distance)) = queue.pop\_front() {

        if node == end {

            return Some(distance);

        }

        for &neighbor in &adj\_list[node]{

            if !visits[neighbor] {

                visits[neighbor] = true;

                queue.push\_back((neighbor, distance+1));

            }

        }

    }

    return None;

}

fn remove\_zero(distances: &mut Vec<usize>) -> &Vec<usize> {

    distances.retain(|&x| x != 0);

    return distances;

}

#[test]

fn test\_zero() {

    let test\_adj = vec![vec![0], vec![2], vec![1]];

    assert\_eq!(degrees(&test\_adj,0,0), Some(0));

}

#[test]

fn test\_one() {

    let test\_adj = vec![vec![0], vec![2], vec![1]];

    assert\_eq!(degrees(&test\_adj,1,2), Some(1));

}

#[test]

fn test\_None() {

    let test\_adj = vec![vec![0], vec![2], vec![1]];

    assert\_eq!(degrees(&test\_adj, 0,2), None);

}

Module:

pub fn average(remove\_zero\_distances: &Vec<usize>) {

    let mut total = 0;

    for x in 0..remove\_zero\_distances.len() {

        total += remove\_zero\_distances[x];

    }

    let avg = total/remove\_zero\_distances.len();

    println!("The average distance is {}", avg);

}

pub fn median(mut remove\_zero\_distances:Vec<usize>) {

    remove\_zero\_distances.sort();

   let middle = remove\_zero\_distances.len()/2;

   let median\_vec = remove\_zero\_distances[middle];

   println!("The median is {}", median\_vec);

}

pub fn six\_degrees(distances: &Vec<usize>) {

    for x in 0..distances.len() {

        if distances[x] > 6 {

            println!("There is a distance that is greater than 6, therefore failing the 6 degrees of separation");

            break;

        }

    }

}

pub fn max(distances: &Vec<usize>, start\_end\_vec: Vec<(usize,usize,usize)>) {

    let mut index = 0;

    let mut highest = 0;

    for x in 0..distances.len() {

        if distances[x] > highest{

            index = x;

            highest = distances[x];

        }

    }

    println!("The maximum distance is {} and the start point is {} with the end point {}", highest, start\_end\_vec[index].1, start\_end\_vec[index].2);

}