Anup De

Final Project – DS 210

**Dataset Information**

My dataset was accessed via the network catalogue, repository, and centrifuge from <https://networks.skewed.de/> . It is a dataset about Facebook.com friendships for individuals employed at a specific company. This dataset was created by Michael Fire, Rami Puzis, and Yuval Elovici for their paper “Organization Mining Using Online Social Networks.” This specific dataset regards the Facebook connections of an international telecommunication company based in North America. The authors chose not to reveal the identity of any of the companies in the study, but give a brief explanation of the company’s important details. The authors identified 1,429 employees who indicated on their Facebook pages that they worked for this specific company. Between these 1,429 employees, there were 32,876 links. These 1429 employees represent 1429 nodes in the graph and 32,876 friendship links are the node connections, or edges of the graph. It is an undirected graph because the study only took friends that follow each other on Facebook.

The list of connections was downloaded from: <https://data4goodlab.github.io/MichaelFire/#section3>

An article on the dataset, along with the 5 other datasets used in the larger study can be found: <https://arxiv.org/pdf/1303.3741.pdf>

The study included an image of this graph, which is included below:

A picture containing map

Description automatically generated

I chose this specific company out of the 6 companies used in the study because of its ideal size. Other datasets were too large for my computer to run, or too small for the project requirements. This company, a medium-sized company according to the study, was the ideal size for this project.

**Project Purpose**

The project goal was to understand some information regarding the distances between nodes in this graph. I searched the entire dataset for connections between all nodes and all other nodes. This search was done taking into account the fact that the graph is undirected. This search finds the distance between a node and only those nodes higher than that node. This would cover all possible combinations because the graph is undirected and the order of node does not matter.

**How the Code Works**

The file is read using the following lines of code:

fn read\_file(path: &str) -> Vec<(u16, u16)> {  //read\_file function to create result Vector of tuples that represent the start and end nodes

    let mut result: Vec<(u16, u16)> = 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::<u16>().unwrap()-1;

        let y = v[1].parse::<u16>().unwrap()-1;

        result.push((x, y));

    }

    return result;

}

The function is called upon using the path of the file and split by comma from the csv file. From there the individual x and y (start and end) numbers are pushed as a tuple into the vector of tuples called result.

The adjacency list is made using the following lines of code:

fn make\_adjacency(result:Vec<(u16, u16)>) -> Vec<Vec<usize>> {

    let num\_nodes = result[result.len()-1].0+1;

    let mut adjacency\_list : Vec<Vec<usize>> = vecvec![];num\_nodes as usize];

    for (a,b) in result.iter() {

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

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

        adjacency\_list[a\_us].push(b\_us);

        adjacency\_list[b\_us].push(a\_us)

    };

    return adjacency\_list

}

Taking in the result, this function first finds the size of the adjacency list. The size of the adjacency list is the number of nodes, as determined by the num\_nodes variable. From there, each connected node is added to the index of every start node. The process is reversed to account for the fact that the graph is undirected.

The distance between a given start and end point is found using the following lines of code:

fn find\_dist(adjacency\_list: &Vec<Vec<usize>>, start: usize, end: usize) -> Option<usize> {

    let mut queue = VecDeque::new();

    let mut visits = vec![false; adjacency\_list.len()];

    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 &adjacency\_list[node]{

            if !visits[neighbor] {

                visits[neighbor] = true;

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

            }

        }

    }

    None

}

A queue is created starting with the node and an initialized distance of 0. A vector of visited status is also created which is initialized as false. Running a loop until the node the function is on is equal to the given end node, the function pops the current location, and reads true for all of the neighbors of that node. If one of those neighbors is the end node, then the function can return the distance. Otherwise the distance increases by 1. If there is no possible connection, then the Option None is returned.

To find the distance between all possible connections of nodes, the following lines of code were run:

fn make\_all\_distances(adjacency\_list: &Vec<Vec<usize>>, distances: &mut Vec<usize>, all\_info: &mut Vec<(usize,usize,usize)>) {

    let mut index = 0;

    let mut now\_on = 0;

    let length = adjacency\_list.len();

    for start in 0..length{

        // loop through all possible connection options

        for end in start..length{

            if let Some(x) = find\_dist(&adjacency\_list, start, end) {

                distances[index] = find\_dist(&adjacency\_list, start, end).unwrap();

                //println!("the distance between {} and {} is {:?}. that was the {} distance",start, end, distances[index],now\_on);

                all\_info.push((distances[index], start, end));

                index+=1;

                now\_on+=1;

            }

            else {

                index+=1;

                now\_on+=1;

                //println!("there is no path between {} and {}", start, end);

                all\_info.push((0,start, end));

            }

        }

    }

}

The function make\_all\_distances takes in the adjacency list created and some created vectors called distances and all\_info. Both are initialized empty and get information stored in them in this function. The distances vector simply stores all the distances, while the all\_info vector is a vector of tuples that stores the distance, the start, and the end of that combination. Iterating through a nested for loop, the find\_dist function is called on the adjacency list with the respective start and end nodes from the loops. If it has a value, then the distance, the start, and the end are pushed into the all\_info vector and the distance is recorded into the distance vector. A print statement can be included here to print out all the distances. If there is no distance, then this is also recorded as a placeholder 0 distance (to be taken away later) and a print statement can be included here to report that there is no path between those nodes.

Now that there is the distances for each possible combination, some math can be done to find information regarding the graph. These functions were placed in a module in a separate file. The code for those functions are found below:

pub fn delete\_nones(all\_info: &Vec<(usize,usize,usize)>) -> Vec<(usize,usize,usize)> {

    let all\_info\_no\_zero: Vec<\_> = all\_info.iter().filter(|&(c, \_, \_)| c != &0usize).map(|&a| a).collect();

    return all\_info\_no\_zero;

}

pub fn find\_average(all\_info\_no\_zero: &Vec<(usize,usize,usize)>) -> f64 {

    let mut sum = 0;

    for i in all\_info\_no\_zero.iter() {

        sum += i.0

    }

    let avg = (sum / all\_info\_no\_zero.len()) as f64;

    return avg

}

pub fn six\_deg\_check(all\_info: &Vec<(usize,usize,usize)>) {

    let mut num\_above = 0;

    for i in all\_info.iter() {

        if i.0 > 6 {

            println!("Going from {} to {} will be more than 6 degrees. It has a distance of {}",i.1,i.2,i.0);

            num\_above += 1

        }

    }

    println!("there are {:?} pairs of nodes that have more than 6 degrees separating them", num\_above);

}

pub fn highest\_dist(all\_info: &Vec<(usize,usize,usize)>)  {

    let mut highest\_dist = (0,0,0);

    for i in all\_info.iter() {

        if i.0 > highest\_dist.0 {

            highest\_dist = \*i

        }

    }

    println!("the farthest distance between two nodes is {:?} between {} and {}", highest\_dist.0, highest\_dist.1, highest\_dist.2);

}

pub fn find\_median(mut all\_info\_no\_zero: Vec<(usize,usize,usize)>) {

    let vec\_len = all\_info\_no\_zero.len();

    all\_info\_no\_zero.sort\_by\_key(|f| f.0);

    if vec\_len % 2 == 0 {

        let middle = vec\_len / 2;

        let median = all\_info\_no\_zero[middle].0;

        println!("the median distance is {:?}",median)

    }

    else {

        let lower\_middle = (vec\_len /2 ) as f64 - 0.5;

        let upper\_middle = (vec\_len /2 ) as f64 + 0.5;

        let median = (all\_info\_no\_zero[upper\_middle as usize].0 + all\_info\_no\_zero[lower\_middle as usize].0)/2;

        println!("the median distance is {:?}",median)

    }

}

pub fn stdev(all\_info\_no\_zero: &Vec<(usize,usize,usize)>) {

    let mean = find\_average(&all\_info\_no\_zero);

    let mut dev = 0.0;

    for i in all\_info\_no\_zero.iter() {

        dev += (i.0 as f64 - mean) \* (i.0 as f64 - mean);

    }

    let variance = dev / (all\_info\_no\_zero.len()) as f64;

    let stdev = variance.sqrt();

    println!("the standard deviation is {:?}", stdev);

}

These functions output various information about the distances. A six degree check is done in the so-named function to determine the number of connections which require more 6 degrees to travel between each other. A print-out is done for each of these instances. The highest distance amongst these is also found.

To find the standard deviation, median, and average, all the distances of 0 were removed first. These distances represented nodes connected to themselves which do not make sense in the context of this question in which people cannot befriend themselves. The function delete\_nones does this in which the code iterates through all of all\_info, looking for a 0 at the beginning of the tuple and removes it, mapping this onto all of the vector. From there the respective functions, find\_average, find\_median, and stdev compute the average, median, and stdev of the distances.

**Output**

Below is the output of the code. The main is running all the functions described in the two paragraphs above.

![Text

Description automatically generated

The code runs in about 90 seconds. Its complexity is of nlog(n).

The output describes the average distance, which is the average distance between every node and every other node. Only one combination takes more than 6 degrees to reach each other and that is the combination from node 256 to node 789. It is the highest distance. The standard deviation is 0.849.

This shows that the company’s employees on Facebook are pretty well connected as most of the employees have few degrees separating them. There were also no islands in the data as everyone can eventually get everywhere.

It is possible to output the adjacency list, and the list of distances, and the all\_info vector which is a vector of tuples that gives the distance, start, and end nodes for every combination. These have all been included at the end of the report after the entire code in the Additional Information section.

**Entire Code:**

use std::fs::File;

use std::io::{BufRead, BufReader};

use std::num::ParseIntError;

use std::collections::VecDeque;

mod sep\_module;

use crate::sep\_module::delete\_nones;

use crate::sep\_module::find\_average;

use crate::sep\_module::six\_deg\_check;

use crate::sep\_module::highest\_dist;

use crate::sep\_module::find\_median;

use crate::sep\_module::stdev;

fn main() -> Result<(), Box<dyn std::error::Error>> {

    let result = read\_file("C:\\Users\\anupd\\Documents\\2 Semester\\DS 210\\FinalProject\\src\\M1Anonymized.csv");

    //println!("{:?}",result[result.len()-1].0+1);

    let adjacency\_list = make\_adjacency(result);

    let mut distances = vec0;(adjacency\_list.len() \* (adjacency\_list.len()+1))/2];

    let mut all\_info: Vec<(usize,usize,usize)> = Vec::new();

    make\_all\_distances(&adjacency\_list, &mut distances, &mut all\_info);

    //println!("{:?}-----------------------------------------------", distances);

    //println!("{:?}-----------------------------------------------", all\_info) ;

    //println!("{:?}-----------------------------------------------",adjacency\_list);

    println!("{:?}", complete\_info(&all\_info));

    Ok(())

}

fn read\_file(path: &str) -> Vec<(u16, u16)> {  //read\_file function to create result Vector of tuples that represent the start and end nodes

    let mut result: Vec<(u16, u16)> = 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::<u16>().unwrap()-1;

        let y = v[1].parse::<u16>().unwrap()-1;

        result.push((x, y));

    }

    return result;

}

fn make\_adjacency(result:Vec<(u16, u16)>) -> Vec<Vec<usize>> {

    let num\_nodes = result[result.len()-1].0+1;

    let mut adjacency\_list : Vec<Vec<usize>> = vec![vec![];num\_nodes as usize];

    for (a,b) in result.iter() {

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

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

        adjacency\_list[a\_us].push(b\_us);

        adjacency\_list[b\_us].push(a\_us)

    };

    return adjacency\_list

}

fn make\_all\_distances(adjacency\_list: &Vec<Vec<usize>>, distances: &mut Vec<usize>, all\_info: &mut Vec<(usize,usize,usize)>) {

    let mut index = 0;

    let mut now\_on = 0;

    let length = adjacency\_list.len();

    for start in 0..length{

        // loop through all possible connection options

        for end in start..length{

            if let Some(x) = find\_dist(&adjacency\_list, start, end) {

                distances[index] = find\_dist(&adjacency\_list, start, end).unwrap();

                //println!("the distance between {} and {} is {:?}. that was the {} distance",start, end, distances[index],now\_on);

                all\_info.push((distances[index], start, end));

                index+=1;

                now\_on+=1;

            }

            else {

                index+=1;

                now\_on+=1;

                //println!("there is no path between {} and {}", start, end);

                all\_info.push((0,start, end));

            }

        }

    }

    //println!("{:?}",all\_info)

}

fn find\_dist(adjacency\_list: &Vec<Vec<usize>>, start: usize, end: usize) -> Option<usize> {

    let mut queue = VecDeque::new();

    let mut visits = vec![false; adjacency\_list.len()];

    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 &adjacency\_list[node]{

            if !visits[neighbor] {

                visits[neighbor] = true;

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

            }

        }

    }

    None

}

fn complete\_info(all\_info: &Vec<(usize,usize,usize)>) {

    let all\_info\_no\_zero = delete\_nones(&all\_info);

    println!("the average distance is {:?}", find\_average(&all\_info\_no\_zero));

    println!("{:?}", six\_deg\_check(all\_info));

    println!("{:?}", highest\_dist(all\_info));

    println!("{:?}", find\_median(all\_info.to\_vec()));

    println!("{:?}", stdev(&all\_info\_no\_zero))

}

#[test]

fn dist0() {

    let mut adjacency\_list : Vec<Vec<usize>> =  vec![vec![],vec![1], vec![3], vec![2]];

    assert\_eq!(find\_dist(&adjacency\_list,1,1), Some(0));

}

#[test]

fn dist1() {

    let mut adjacency\_list : Vec<Vec<usize>> =  vec![vec![],vec![1], vec![3], vec![2]];

    assert\_eq!(find\_dist(&adjacency\_list, 2,3), Some(1))

}

#[test]

fn distna() {

    let mut adjacency\_list : Vec<Vec<usize>> =  vec![vec![],vec![1], vec![3], vec![2]];

    assert\_eq!(find\_dist(&adjacency\_list, 1,2),None)

}

File for sep\_module

pub fn delete\_nones(all\_info: &Vec<(usize,usize,usize)>) -> Vec<(usize,usize,usize)> {

    let all\_info\_no\_zero: Vec<\_> = all\_info.iter().filter(|&(c, \_, \_)| c != &0usize).map(|&a| a).collect();

    return all\_info\_no\_zero;

}

pub fn find\_average(all\_info\_no\_zero: &Vec<(usize,usize,usize)>) -> f64 {

    let mut sum = 0;

    for i in all\_info\_no\_zero.iter() {

        sum += i.0

    }

    let avg = (sum / all\_info\_no\_zero.len()) as f64;

    return avg

}

pub fn six\_deg\_check(all\_info: &Vec<(usize,usize,usize)>) {

    let mut num\_above = 0;

    for i in all\_info.iter() {

        if i.0 > 6 {

            println!("Going from {} to {} will be more than 6 degrees. It has a distance of {}",i.1,i.2,i.0);

            num\_above += 1

        }

    }

    println!("there are {:?} pairs of nodes that have more than 6 degrees separating them", num\_above);

}

pub fn highest\_dist(all\_info: &Vec<(usize,usize,usize)>)  {

    let mut highest\_dist = (0,0,0);

    for i in all\_info.iter() {

        if i.0 > highest\_dist.0 {

            highest\_dist = \*i

        }

    }

    println!("the farthest distance between two nodes is {:?} between {} and {}", highest\_dist.0, highest\_dist.1, highest\_dist.2);

}

pub fn find\_median(mut all\_info\_no\_zero: Vec<(usize,usize,usize)>) {

    let vec\_len = all\_info\_no\_zero.len();

    all\_info\_no\_zero.sort\_by\_key(|f| f.0);

    if vec\_len % 2 == 0 {

        let middle = vec\_len / 2;

        let median = all\_info\_no\_zero[middle].0;

        println!("the median distance is {:?}",median)

    }

    else {

        let lower\_middle = (vec\_len /2 ) as f64 - 0.5;

        let upper\_middle = (vec\_len /2 ) as f64 + 0.5;

        let median = (all\_info\_no\_zero[upper\_middle as usize].0 + all\_info\_no\_zero[lower\_middle as usize].0)/2;

        println!("the median distance is {:?}",median)

    }

}

pub fn stdev(all\_info\_no\_zero: &Vec<(usize,usize,usize)>) {

    let mean = find\_average(&all\_info\_no\_zero);

    let mut dev = 0.0;

    for i in all\_info\_no\_zero.iter() {

        dev += (i.0 as f64 - mean) \* (i.0 as f64 - mean);

    }

    let variance = dev / (all\_info\_no\_zero.len()) as f64;

    let stdev = variance.sqrt();

    println!("the standard deviation is {:?}", stdev);

}

**Additional Information**

Adjacency List Output snippet: ![Background pattern

Description automatically generated

Distances Output snippet:

Background pattern

Description automatically generated

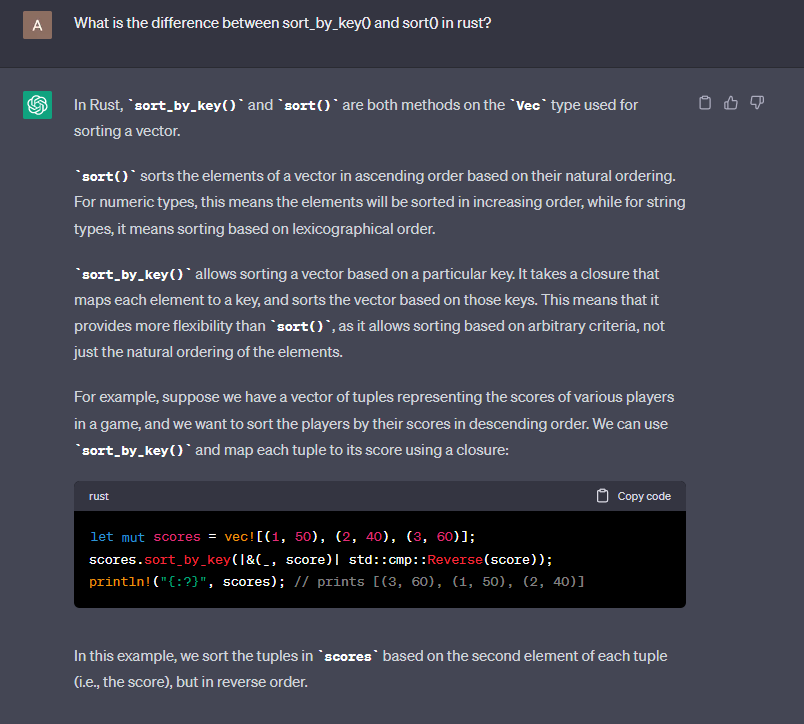
All\_info Output snippet:

A picture containing text

Description automatically generated

Generative AI Assistance Policy

In accordance with the **terribly worded and pathetic Boston University Faculty of Computing and Data Science Generative AI Assistance Policy**, I am including all uses of “AI tools” that were used in my project.



When trying to sort the distances from least to greatest in the vector of tuples, I sought the use of OpenAI’s ChatGPT to find the proper use of sorting. While the information provided by ChatGPT is not always accurate, the speed and accessibility of information enhanced my decision to use the program. Though ChatGPT outputted the code above, I did not use any of it. This is because the code was extremely complicated, and I did not understand what the syntax did. However the explanation did help my decision to use sort\_by\_key.

This should satisfy the **unfortunately deplorable Boston University Faculty of Computing and Data Science Generative AI Assistance Policy**