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:

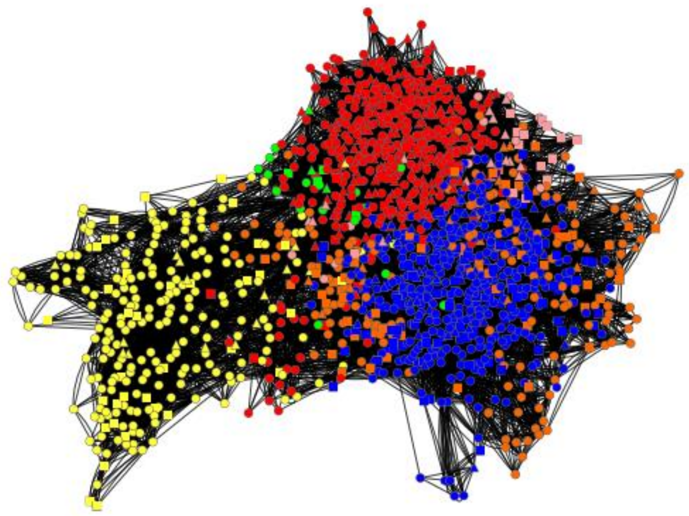


Figure 3: **M1 Company**: *Blue and Orange nodes* - R&D divisions. *Red nodes* - senior management. *Yellow nodes* - international consultants and support engineers. *Green nodes* - North American headquarter employees.

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:

```rust
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:

```rust
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
}
```

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:

```rust
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:

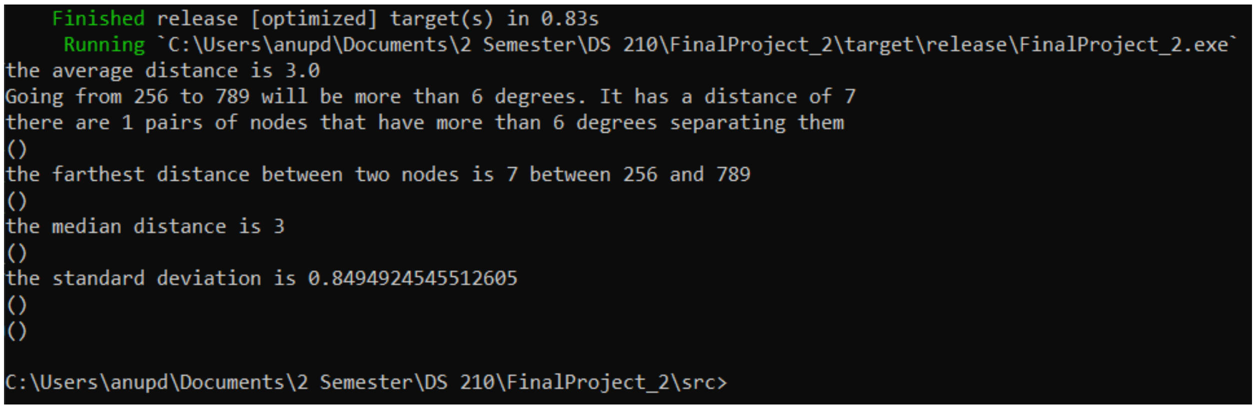
```rust
pub fn delete_nones(all_info: &Vec<(usize,usize,usize)>) ->
Vec<(usize,usize,usize)> {

    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.



```
    Finished release [optimized] target(s) in 0.83s
     Running `C:\Users\anupd\Documents\2 Semester\DS 210\FinalProject_2\target\release\FinalProject_2.exe`
the average distance is 3.0
Going from 256 to 789 will be more than 6 degrees. It has a distance of 7
there are 1 pairs of nodes that have more than 6 degrees separating them
()
the farthest distance between two nodes is 7 between 256 and 789
()
the median distance is 3
()
the standard deviation is 0.8494924545512605
()
()

C:\Users\anupd\Documents\2 Semester\DS 210\FinalProject_2\src>
```

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:

```rust
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 = vec![0;(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(())

}
```

```rust
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));
            }

        }
```

```rust
    }
    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)

}
```

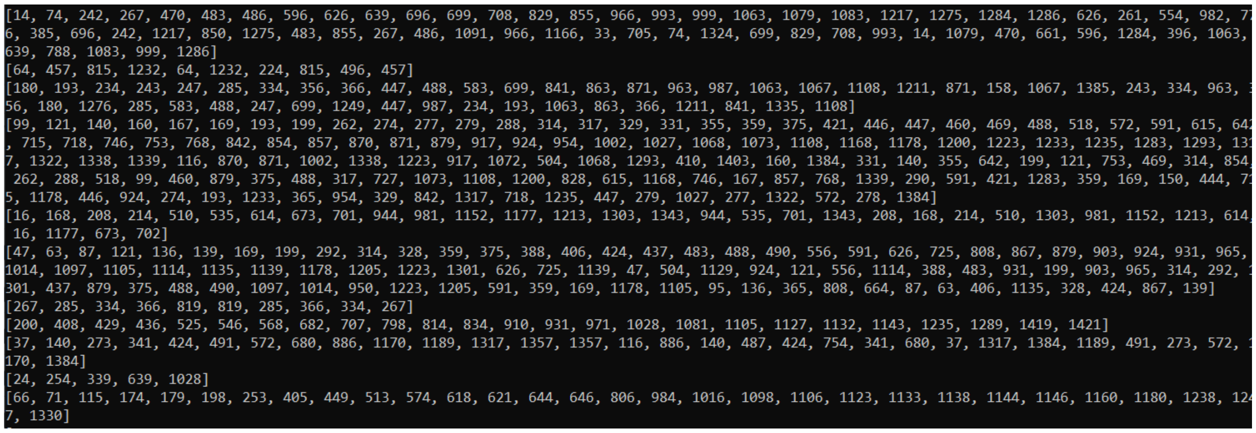
File for sep_module

```rust
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);
```

```rust
}


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);

}
```

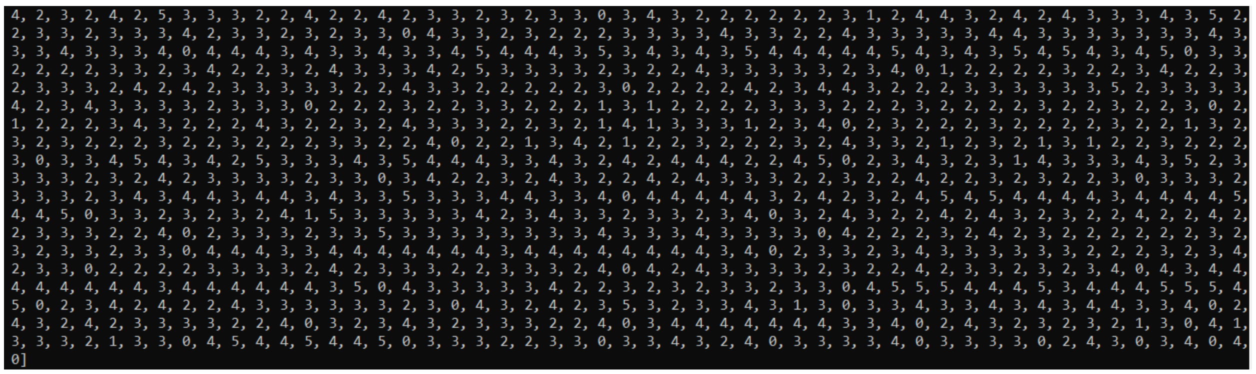
Additional Information

Adjacency List Output snippet:



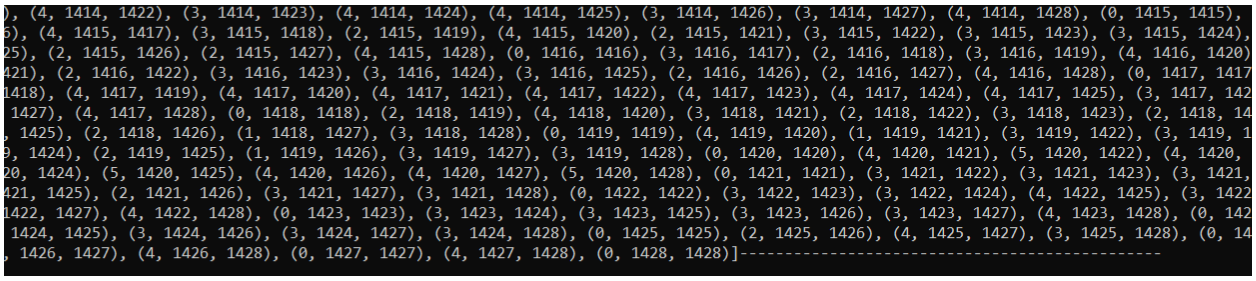
```
[14, 74, 242, 267, 470, 483, 486, 596, 626, 639, 696, 699, 708, 829, 855, 966, 993, 999, 1063, 1079, 1083, 1217, 1275, 1284, 1286, 626, 261, 554, 982, 7
6, 385, 696, 242, 1217, 850, 1275, 483, 855, 267, 486, 1091, 966, 1166, 33, 705, 74, 1324, 699, 829, 708, 993, 14, 1079, 470, 661, 596, 1284, 396, 1063,
639, 788, 1083, 999, 1286]
[64, 457, 815, 1232, 64, 1232, 224, 815, 496, 457]
[180, 193, 234, 243, 247, 285, 334, 356, 366, 447, 488, 583, 699, 841, 863, 871, 963, 987, 1063, 1067, 1108, 1211, 871, 158, 1067, 1385, 243, 334, 963,
56, 180, 1276, 285, 583, 488, 247, 699, 1249, 447, 987, 234, 193, 1063, 863, 366, 1211, 841, 1335, 1108]
[99, 121, 140, 160, 167, 169, 193, 199, 262, 274, 277, 279, 288, 314, 317, 329, 331, 355, 359, 375, 421, 446, 447, 460, 469, 488, 518, 572, 591, 615, 64
, 715, 718, 746, 753, 768, 842, 854, 857, 870, 871, 879, 917, 924, 954, 1002, 1027, 1068, 1073, 1108, 1168, 1178, 1200, 1223, 1233, 1235, 1283, 1293, 13
7, 1322, 1338, 1339, 116, 870, 871, 1002, 1338, 1223, 917, 1072, 504, 1068, 1293, 410, 1403, 160, 1384, 331, 140, 355, 642, 199, 121, 753, 469, 314, 854,
 262, 288, 518, 99, 460, 879, 375, 488, 317, 727, 1073, 1108, 1200, 828, 615, 1168, 746, 167, 857, 768, 1339, 290, 591, 421, 1283, 359, 169, 150, 444, 7
5, 1178, 446, 924, 274, 193, 1233, 365, 954, 329, 842, 1317, 718, 1235, 447, 279, 1027, 277, 1322, 572, 278, 1384]
[16, 168, 208, 214, 510, 535, 614, 673, 701, 944, 981, 1152, 1177, 1213, 1303, 1343, 944, 535, 701, 1343, 208, 168, 214, 510, 1303, 981, 1152, 1213, 614,
 16, 1177, 673, 702]
[47, 63, 87, 121, 136, 139, 169, 199, 292, 314, 328, 359, 375, 388, 406, 424, 437, 483, 488, 490, 556, 591, 626, 725, 808, 867, 879, 903, 924, 931, 965,
1014, 1097, 1105, 1114, 1135, 1139, 1178, 1205, 1223, 1301, 626, 725, 1139, 47, 504, 1129, 924, 121, 556, 1114, 388, 483, 931, 199, 903, 965, 314, 292, 1
301, 437, 879, 375, 488, 490, 1097, 1014, 950, 1223, 1205, 591, 359, 169, 1178, 1105, 95, 136, 365, 808, 664, 87, 63, 406, 1135, 328, 424, 867, 139]
[267, 285, 334, 366, 819, 819, 285, 366, 334, 267]
[200, 408, 429, 436, 525, 546, 568, 682, 707, 798, 814, 834, 910, 931, 971, 1028, 1081, 1105, 1127, 1132, 1143, 1235, 1289, 1419, 1421]
[37, 140, 273, 341, 424, 491, 572, 680, 886, 1170, 1189, 1317, 1357, 1357, 116, 886, 140, 487, 424, 754, 341, 680, 37, 1317, 1384, 1189, 491, 273, 572, 1
170, 1384]
[24, 254, 339, 639, 1028]
[66, 71, 115, 174, 179, 198, 253, 405, 449, 513, 574, 618, 621, 644, 646, 806, 984, 1016, 1098, 1106, 1123, 1133, 1138, 1144, 1146, 1160, 1180, 1238, 12
7, 1330]
```

Distances Output snippet:



```
 4, 2, 3, 2, 4, 2, 5, 3, 3, 3, 2, 2, 4, 2, 2, 4, 2, 3, 3, 2, 3, 2, 3, 3, 0, 3, 4, 3, 2, 2, 2, 2, 2, 2, 3, 1, 2, 4, 4, 3, 2, 4, 2, 4, 3, 3, 3, 4, 3, 5, 2,
 2, 3, 3, 2, 3, 3, 3, 4, 2, 3, 3, 2, 3, 2, 3, 3, 0, 4, 3, 3, 2, 3, 2, 2, 2, 3, 3, 3, 3, 4, 3, 3, 2, 2, 4, 3, 3, 3, 3, 3, 4, 4, 3, 3, 3, 3, 3, 3, 3, 4, 3,
 3, 3, 4, 3, 3, 3, 4, 0, 4, 4, 4, 3, 4, 3, 3, 4, 3, 3, 4, 5, 4, 4, 4, 3, 5, 3, 4, 3, 4, 3, 5, 4, 4, 4, 4, 4, 5, 4, 3, 4, 3, 5, 4, 5, 4, 3, 4, 5, 0, 3, 3,
 2, 2, 2, 2, 3, 3, 2, 3, 4, 2, 2, 3, 2, 4, 3, 3, 3, 4, 2, 5, 3, 3, 3, 3, 2, 3, 2, 2, 4, 3, 3, 3, 3, 3, 2, 3, 4, 0, 1, 2, 2, 2, 2, 3, 2, 2, 3, 4, 2, 2, 3,
 2, 3, 3, 3, 2, 4, 2, 4, 2, 3, 3, 3, 3, 3, 2, 2, 4, 3, 3, 2, 2, 2, 2, 2, 3, 0, 2, 2, 2, 2, 4, 2, 3, 4, 4, 3, 2, 2, 2, 3, 3, 3, 3, 3, 3, 5, 2, 3, 3, 3, 3,
 4, 2, 3, 4, 3, 3, 3, 3, 2, 3, 3, 3, 0, 2, 2, 2, 3, 2, 2, 3, 3, 2, 2, 2, 1, 3, 1, 2, 2, 2, 2, 3, 3, 3, 2, 2, 2, 3, 2, 2, 2, 2, 3, 2, 2, 3, 2, 2, 3, 0, 2,
 1, 2, 2, 2, 3, 4, 3, 2, 2, 2, 4, 3, 2, 2, 3, 2, 4, 3, 3, 3, 2, 2, 3, 2, 1, 4, 1, 3, 3, 3, 1, 2, 3, 4, 0, 2, 3, 2, 2, 2, 3, 2, 2, 2, 2, 3, 2, 2, 1, 3, 2,
 3, 2, 3, 2, 2, 2, 3, 2, 2, 3, 2, 2, 2, 3, 3, 2, 2, 4, 0, 2, 2, 1, 3, 4, 2, 1, 2, 2, 3, 2, 2, 2, 3, 2, 4, 3, 3, 2, 1, 2, 3, 2, 1, 3, 1, 2, 2, 3, 2, 2, 2,
 3, 0, 3, 3, 4, 5, 4, 3, 4, 2, 5, 3, 3, 3, 4, 3, 5, 4, 4, 4, 3, 3, 4, 3, 2, 4, 2, 4, 4, 4, 2, 2, 4, 5, 0, 2, 3, 4, 3, 2, 3, 1, 4, 3, 3, 3, 4, 3, 5, 2, 3,
 3, 3, 3, 2, 3, 2, 4, 2, 3, 3, 3, 3, 2, 3, 3, 0, 3, 4, 2, 2, 3, 2, 4, 3, 2, 2, 4, 2, 4, 3, 3, 3, 2, 2, 3, 2, 2, 4, 2, 2, 3, 2, 3, 2, 2, 3, 0, 3, 3, 3, 2,
 3, 3, 3, 2, 3, 4, 3, 4, 4, 3, 4, 4, 3, 4, 3, 3, 5, 3, 3, 3, 4, 4, 3, 3, 4, 0, 4, 4, 4, 4, 4, 3, 2, 4, 2, 3, 2, 4, 5, 4, 5, 4, 4, 4, 4, 3, 4, 4, 4, 4, 5,
 4, 4, 5, 0, 3, 3, 2, 3, 2, 3, 2, 4, 1, 5, 3, 3, 3, 3, 3, 4, 2, 3, 4, 3, 3, 2, 3, 3, 2, 3, 4, 0, 3, 2, 4, 3, 2, 2, 4, 2, 4, 3, 2, 3, 2, 2, 4, 2, 2, 4, 2,
 2, 3, 3, 3, 2, 2, 4, 0, 2, 3, 3, 3, 2, 3, 3, 5, 3, 3, 3, 3, 3, 3, 3, 3, 4, 3, 3, 3, 4, 3, 3, 3, 3, 0, 4, 2, 2, 2, 3, 2, 4, 2, 3, 2, 2, 2, 2, 2, 2, 3, 2,
 3, 2, 3, 3, 2, 3, 3, 0, 4, 4, 4, 3, 3, 4, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 3, 4, 0, 2, 3, 3, 2, 3, 4, 3, 3, 3, 3, 3, 3, 2, 2, 2, 3, 2, 3, 4,
 2, 3, 3, 0, 2, 2, 2, 2, 3, 3, 3, 3, 2, 4, 2, 3, 3, 3, 2, 2, 3, 3, 3, 2, 4, 0, 4, 2, 4, 3, 3, 3, 3, 2, 3, 2, 2, 4, 2, 3, 3, 2, 3, 2, 3, 4, 0, 4, 3, 4, 4,
 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 3, 5, 0, 4, 3, 3, 3, 3, 3, 4, 2, 2, 3, 2, 3, 2, 3, 3, 2, 3, 3, 0, 4, 5, 5, 5, 4, 4, 4, 5, 3, 4, 4, 4, 5, 5, 5, 4,
 5, 0, 2, 3, 4, 2, 4, 2, 2, 4, 3, 3, 3, 3, 3, 3, 2, 3, 0, 4, 3, 2, 4, 2, 3, 5, 3, 2, 3, 3, 4, 3, 1, 3, 0, 3, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 0, 2,
 4, 3, 2, 4, 2, 3, 3, 3, 3, 2, 2, 4, 0, 3, 2, 3, 4, 3, 2, 3, 3, 3, 2, 2, 4, 0, 3, 4, 4, 4, 4, 4, 4, 4, 3, 3, 4, 0, 2, 4, 3, 2, 3, 2, 3, 2, 1, 3, 0, 4, 1,
 3, 3, 3, 2, 1, 3, 3, 0, 4, 5, 4, 4, 5, 4, 4, 5, 0, 3, 3, 3, 2, 2, 3, 3, 0, 3, 3, 4, 3, 2, 4, 0, 3, 3, 3, 3, 4, 0, 3, 3, 3, 3, 0, 2, 4, 3, 0, 3, 4, 0, 4,
 0]
```

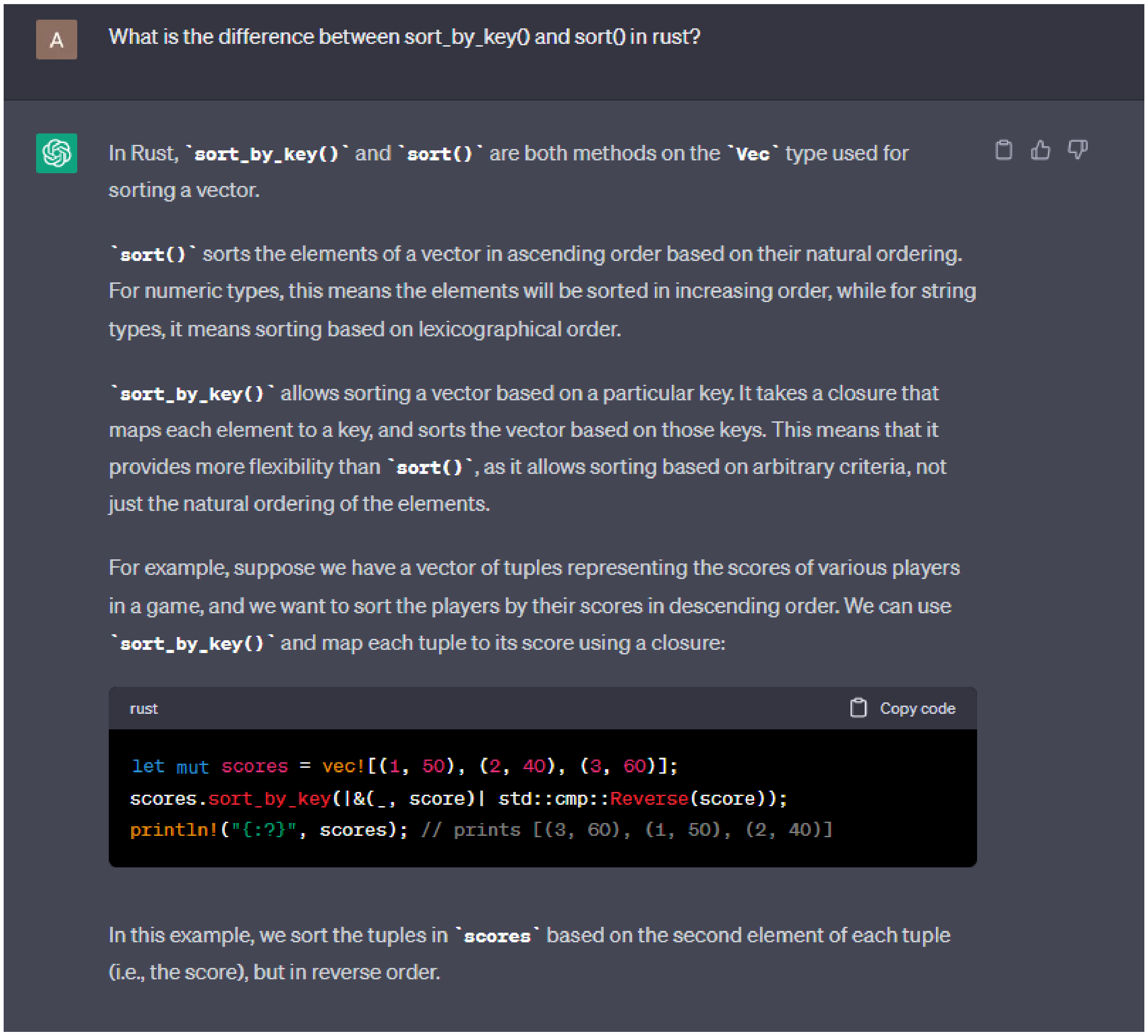
All_info Output snippet:



```
), (4, 1414, 1422), (3, 1414, 1423), (4, 1414, 1424), (4, 1414, 1425), (3, 1414, 1426), (3, 1414, 1427), (4, 1414, 1428), (0, 1415, 1415),
6), (4, 1415, 1417), (3, 1415, 1418), (2, 1415, 1419), (4, 1415, 1420), (2, 1415, 1421), (3, 1415, 1422), (3, 1415, 1423), (3, 1415, 1424),
25), (2, 1415, 1426), (2, 1415, 1427), (4, 1415, 1428), (0, 1416, 1416), (3, 1416, 1417), (2, 1416, 1418), (3, 1416, 1419), (4, 1416, 1420)
421), (2, 1416, 1422), (3, 1416, 1423), (3, 1416, 1424), (3, 1416, 1425), (2, 1416, 1426), (2, 1416, 1427), (4, 1416, 1428), (0, 1417, 1417
1418), (4, 1417, 1419), (4, 1417, 1420), (4, 1417, 1421), (4, 1417, 1422), (4, 1417, 1423), (4, 1417, 1424), (4, 1417, 1425), (3, 1417, 142
 1427), (4, 1417, 1428), (0, 1418, 1418), (2, 1418, 1419), (4, 1418, 1420), (3, 1418, 1421), (2, 1418, 1422), (3, 1418, 1423), (2, 1418, 14
, 1425), (2, 1418, 1426), (1, 1418, 1427), (3, 1418, 1428), (0, 1419, 1419), (4, 1419, 1420), (1, 1419, 1421), (3, 1419, 1422), (3, 1419, 1
9, 1424), (2, 1419, 1425), (1, 1419, 1426), (3, 1419, 1427), (3, 1419, 1428), (0, 1420, 1420), (4, 1420, 1421), (5, 1420, 1422), (4, 1420,
20, 1424), (5, 1420, 1425), (4, 1420, 1426), (4, 1420, 1427), (5, 1420, 1428), (0, 1421, 1421), (3, 1421, 1422), (3, 1421, 1423), (3, 1421,
421, 1425), (2, 1421, 1426), (3, 1421, 1427), (3, 1421, 1428), (0, 1422, 1422), (3, 1422, 1423), (3, 1422, 1424), (4, 1422, 1425), (3, 1422
1422, 1427), (4, 1422, 1428), (0, 1423, 1423), (3, 1423, 1424), (3, 1423, 1425), (3, 1423, 1426), (3, 1423, 1427), (4, 1423, 1428), (0, 142
 1424, 1425), (3, 1424, 1426), (3, 1424, 1427), (3, 1424, 1428), (0, 1425, 1425), (2, 1425, 1426), (4, 1425, 1427), (3, 1425, 1428), (0, 14
, 1426, 1427), (4, 1426, 1428), (0, 1427, 1427), (4, 1427, 1428), (0, 1428, 1428)]-----------------------------------------------
```

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.



What is the difference between sort_by_key() and sort() in rust?

In Rust, `sort_by_key()` and `sort()` are both methods on the `Vec` type used for sorting a vector.

`sort()` sorts the elements of a vector in ascending order based on their natural ordering. For numeric types, this means the elements will be sorted in increasing order, while for string types, it means sorting based on lexicographical order.

`sort_by_key()` allows sorting a vector based on a particular key. It takes a closure that maps each element to a key, and sorts the vector based on those keys. This means that it provides more flexibility than `sort()`, as it allows sorting based on arbitrary criteria, not just the natural ordering of the elements.

For example, suppose we have a vector of tuples representing the scores of various players in a game, and we want to sort the players by their scores in descending order. We can use `sort_by_key()` and map each tuple to its score using a closure:

```rust
let mut scores = vec![(1, 50), (2, 40), (3, 60)];
scores.sort_by_key(|&(_, score)| std::cmp::Reverse(score));
println!("{:?}", scores); // prints [(3, 60), (1, 50), (2, 40)]
```

In this example, we sort the tuples in `scores` based on the second element of each tuple (i.e., the score), but in reverse order.

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**