Mark Castro
Prof. Kontothanassis
DS210
4 May 2025

Final Project Writeup

A. Project Overview

I am asking how connected different graphs are. There is a graph of twitter interactions, where there are different graphs for likes, retweets, and mentions. Can I travel farther on likes, retweets, or mentions?

My data set comes from the stanford dataset. https://snap.stanford.edu/data/higgs-twitter.html I only am using 100k lines of the dataset.

B. Data Processing

I loaded my dataset into rust with a text file. No cleaning was necessary, but I skipped the "time" index because it has no relevance to this project. It is stored in Edge and Adjacency lists.

C. Code Structure

I have one module for the graph struct, which makes graphs and stores them as adjacency lists.

I have an enum for the data type which helps me sort it into which graph it needs to be added to. It keeps everything as a mention, retweet, or reply. I also have a struct for the graphs, which stores the size of the graph and the adjacency list of each graph.

The program makes a graph based on which enum the data belongs to, and then finds the average distance you can "walk" on each graph.

D. Tests

This test checks that the function can reverse the edge list properly to make the undirected graph.

```
#[test]
fn does_reverse_edges_work() {
    let test_vec = vec!((9,0), (8,1), (7,2), (6,3));
    assert_eq!(reverse_edges(&test_vec), vec!((0,9), (1,8), (2,7),
(3,6)));
}
This test checks that for this sample data set, the index map is 7 long,
because then it is probably making it right. The sample data set contains
7 unique users.
#[test]
fn is_idx_map_7_long() {
    assert_eq!(make_index_map("src/test-set.txt").len(), 7);
}
```

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests src/main.rs (target/debug/deps/final_project-c75aade445789d99)

running 2 tests
test does_reverse_edges_work ... ok
test is_idx_map_7_long ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

E. Results

```
    Finished `release` profile [optimized] target(s) in 0.79s
     Running `target/release/project`
the average path length for retweeting is 13.142840609231055
the average path length for mentioning is 9.957940916042824
the average path lenth for replying is 4.73625170998632
Time it took: 2810.417586929s
```

This tells us that we can travel an average of 13.14 nodes via the retweet graph, 9.96 nodes via the mention graph, and 4.74 nodes for the reply graph.

F. Usage Instructions
The runtime is around 45 minutes. There are no special instructions, as it just runs.

G. AI-Assistance Disclosure and Other Citations
I did not use AI for help with this project. I attended Joey Russionello's office hours for help.

Code Writeup:

Importing modules
```
mod graph;
use crate::graph::Graph;
```

Makes a map for the retweets, replies, and mentions. Prints the average path lengths for the retweet map, mention map, and replying map. Also returns the time it took for the program to run.
```
fn main() {
    let (retweet_vec, reply_vec, mention_vec) =
read_file("src/testing.txt");
    let before = SystemTime::now();
    println!("the average path length for retweeting is {:?}",
average_path(retweet_vec.adjacency_list.len(),
retweet_vec.adjacency_list));
    println!("the average path length for mentioning is {:?}",
average_path(mention_vec.adjacency_list.len(),
mention_vec.adjacency_list));
    println!("the average path lenth for replying is {:?}",
average_path(reply_vec.adjacency_list.len(),
reply_vec.adjacency_list));
    let after = SystemTime::now();
    let difference = after.duration_since(before);
    let difference = difference.expect("Did the clock go back?");
    println!("Time it took: {:?}", difference);
}

use std::fs::File;
use std::io::prelude::*;
use std::collections::HashMap;
use std::collections::VecDeque;
use std::time::SystemTime;
```

Type aliasing for the graph functions
```
type Vertex = usize;
type AdjacencyLists = Vec<Vec<Vertex>>;
type ListOfEdges = Vec<(Vertex,Vertex)>;
```

Makes an enum to hold the interaction types listed in the graph
```
#[derive(Debug, Clone, Copy)]
enum InteractionType {
    Mention,
    Retweet
    Reply,
```

```
}
```

Makes a function that reads the file and separates each interaction type into three separate graphs. They include an adjacency list that is undirected and unweighted. A match statement is used to determine which graph to put each datum in. First, it makes an edge list using the graph module functions and then makes the graphs.

```
fn read_file(path: &str) -> (Graph, Graph, Graph) {
    let index_map = make_index_map(path);
    let mut edge_list_retweet : ListOfEdges = Vec::new();
    let mut edge_list_reply : ListOfEdges = Vec::new();
    let mut edge_list_mention : ListOfEdges = 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();
        let original_user = v[0].parse::<usize>().unwrap();
        let to_user = v[1].parse::<usize>().unwrap();
        let interaction_type = v[3].parse::<String>().unwrap();
        let mut interaction_enum = None;
        if interaction_type == String::from("MT") {
            interaction_enum = Some(InteractionType::Mention);
        }
        else if interaction_type == String::from("RT") {
            interaction_enum = Some(InteractionType::Retweet);
        }
        else if interaction_type == String::from("RE") {
            interaction_enum = Some(InteractionType::Reply);
        }
        match interaction_enum {
            Some(InteractionType::Mention) => {
                edge_list_mention.push((original_user, to_user));
            }
            Some(InteractionType::Reply) => {
                edge_list_reply.push((original_user, to_user));
            }
            Some(InteractionType::Retweet) => {
                edge_list_retweet.push((original_user, to_user));
            }
            None => println!("None type passed in")
        }
    }

    let size = index_map.len();
```

```
    let retweet_vec : Graph = Graph::create_undirected(size,
&edge_list_retweet, index_map.clone());
    let mention_vec : Graph = Graph::create_undirected(size,
&edge_list_mention, index_map.clone());
    let reply_vec : Graph = Graph::create_undirected(size,
&edge_list_reply, index_map.clone());

    return (retweet_vec, reply_vec, mention_vec)
}
```

This makes a map of the indexes for the points as the numbers corresponding to each user don't start at 0. It gives each "user" an index starting at 0 and stores it in a hashmap. The graphs will use these indexes to store the adjacency lists.

```
fn make_index_map(path:&str) -> HashMap<usize, usize> {
    let mut counter = 0;
    let mut index_map : HashMap<usize, usize> = HashMap::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();
        let original_user = v[0].parse::<usize>().unwrap();
        if let None = index_map.get(&original_user) {
            index_map.insert(original_user, counter);
            counter += 1;
        }
        let to_user = v[1].parse::<usize>().unwrap();
        if let None = index_map.get(&to_user) {
            index_map.insert(to_user, counter);
            counter += 1;
        }
    }
    return index_map
}
```

This uses BFS to compute the farthest point from a point. It uses a queue to track which points it has visited. It also changes the user id's for the indexes at which they are stored. It stores all the distances in a vector and returns the longest one.

```
fn compute_distance_bfs(start: Vertex,
adjacency_list:&AdjacencyLists) -> usize {
    let index_map = make_index_map("src/testing.txt");
    let mut counting_vector : Vec<usize> = Vec::new();
```

```
    let mut distance : Vec<Option<u32>> =
vec![None;adjacency_list.len()+1];
    distance[start] = Some(0);
    let mut queue : VecDeque<Vertex> = VecDeque::new();
    queue.push_back(start);
    while let Some(v) = queue.pop_front() {
        for u in adjacency_list[v].iter() {
            let u_idx = index_map.get(&u).expect("There's an error
finding the index");
            if let None = distance[*u_idx] {
                distance[*u_idx] = Some (distance[v].unwrap()+1);
                queue.push_back(*u_idx);
            }
        }
    }
    for v in 0..adjacency_list.len() {
        if let Some(_k) = distance[v] {
            counting_vector.push(distance[v].unwrap() as usize)
        }
    }
    counting_vector.sort_by(|a,b| b.cmp(&a));
    return counting_vector[0]
}
```

This takes all the points in the set and finds the longest distance for each one. It then adds all the distances together and divides by how many items are in the set. This returns the average distance of nodes you can travel through the graph.

```
fn average_path(n:usize,adjacency_list: Vec<Vec<usize>>) -> f64 {
    let mut counting_vector : Vec<usize> = Vec::new();
    for i in 0..n {
        if adjacency_list[i].len() != 0 {
            counting_vector.push(compute_distance_bfs(i,
&adjacency_list));
        }
    }
    let mut counter = 0;
    for i in &counting_vector {
        counter += *i
    }
    return counter as f64/(counting_vector.len() as f64)
}
```

Graph.rs module

```
type Vertex = usize;
type AdjacencyLists = Vec<Vec<Vertex>>;
type ListOfEdges = Vec<(Vertex,Vertex)>;
use std::collections::HashMap;

#[derive(Debug)]
pub struct Graph {
    pub n: usize,
    pub adjacency_list: AdjacencyLists,
}
```

Switches the order of a vector of tuples.

```
fn reverse_edges(list:&ListOfEdges) -> ListOfEdges {
    let mut new_list = Vec::new();
    for (u,v) in list {
        new_list.push((*v,*u));
    }
    return new_list
}

impl Graph{
```

Adds edges to an adjacency list for each edge in an edge list.

```
    fn add_directed_edges(&mut self, edges:&ListOfEdges, index_map :
HashMap<usize, usize>) {
        for (u,v) in edges {
            let idx = index_map.get(&u).expect("An error with finding
index in graph.rs");
            self.adjacency_list[*idx].push(*v);
        }
    }
```

Makes a new graph with the directed edges from an edge list using the add directed edges function.

```
    fn create_directed(n:usize, edges:&ListOfEdges, index_map:
HashMap<usize, usize>) -> Graph {
        let mut g = Graph{n,adjacency_list:vec![Vec::new();n]};
        g.add_directed_edges(edges, index_map);
        return g
    }
```

Makes a directed graph and then an undirected graph by reversing the edges and adding those too.

```
    pub fn create_undirected(n:usize, edges:&ListOfEdges,
index_map:HashMap<usize, usize>) -> Graph {
```

```
        let mut g = Self::create_directed(n, edges,
index_map.clone());
        g.add_directed_edges(&reverse_edges(edges), index_map);
        return g
    }
}
```