Top 10 nodes with highest betweenness centrality:

1. Node 107: 0.2462

2. Node 1684: 0.1784

3. Node 1912: 0.1492

4. Node 3437: 0.1283

5. Node 0: 0.0831

6. Node 348: 0.0433

7. Node 686: 0.0357

8. Node 414: 0.0215

9. Node 1465: 0.0172

10. Node 698: 0.0163

Top 10 nodes with highest degree distribution:

1. Node 107: 1045

2. Node 1684: 792

3. Node 1912: 755

4. Node 3437: 547

5. Node 0: 347

6. Node 2543: 294

7. Node 2347: 291

8. Node 1888: 254

9. Node 1800: 245

10. Node 1663: 235

Top 10 nodes with most neighbors at distance 2:

1. Node 58: 2903

2. Node 171: 2145

3. Node 563: 2029

4. Node 428: 2005

5. Node 1534: 1947

6. Node 1666: 1944

7. Node 1171: 1899

8. Node 1726: 1801

9. Node 1450: 1799

10. Node 1419: 1798

use std::error::Error;

use std::fs::File;

use std::io::prelude::\*;

use std::path::Path;

use std::collections::{HashMap, BinaryHeap};

use std::cmp::Reverse;

// Define the struct for the graph

struct MyGraph {

adjacency\_list: Vec<Vec<usize>>,

}

impl MyGraph {

fn new() -> MyGraph {

MyGraph { adjacency\_list: vec![] }

}

fn add\_edge(&mut self, src: usize, dest: usize) {

while self.adjacency\_list.len() <= src.max(dest) {

self.adjacency\_list.push(vec![]);

}

if !self.adjacency\_list[src].contains(&dest) {

self.adjacency\_list[src].push(dest);

}

if !self.adjacency\_list[dest].contains(&src) {

self.adjacency\_list[dest].push(src);

}

}

fn degree\_distribution(&self) -> Vec<usize> {

let mut degrees = vec![0; self.adjacency\_list.len()];

for (vertex, neighbors) in self.adjacency\_list.iter().enumerate() {

degrees[vertex] = neighbors.len();

}

degrees

}

fn neighbors\_at\_distance\_2(&self) -> Vec<usize> {

let mut neighbors\_at\_distance\_2 = vec![0; self.adjacency\_list.len()];

for (vertex, neighbors) in self.adjacency\_list.iter().enumerate() {

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

visited[vertex] = true;

for &neighbor in neighbors {

visited[neighbor] = true;

for &second\_hop\_neighbor in &self.adjacency\_list[neighbor] {

if !visited[second\_hop\_neighbor] && !neighbors.contains(&second\_hop\_neighbor) {

neighbors\_at\_distance\_2[vertex] += 1;

visited[second\_hop\_neighbor] = true;

}

}

}

}

neighbors\_at\_distance\_2

}

// Function to calculate betweenness centrality

fn betweenness\_centrality(adjacency\_list: &[Vec<usize>], all\_shortest\_paths: &[(usize, usize, Vec<usize>)]) -> HashMap<usize, f64> {

// Count the number of times each node has appeared in the middle of a vector(exclude the times it starts or ends at that node)

let mut node\_counts: HashMap<usize, usize> = HashMap::new();

for (\_, \_, path) in all\_shortest\_paths {

for &node in path.iter().skip(1).take(path.len() - 2) {

\*node\_counts.entry(node).or\_insert(0) += 1;

}

}

// Calculate betweenness centrality for each node

let num\_nodes = adjacency\_list.len();

let mut betweenness\_centralities = HashMap::new();

for (node, count) in &node\_counts {

//divid the count by the total number of shortest paths

let centrality = \*count as f64 / (num\_nodes \* (num\_nodes - 1)) as f64;

betweenness\_centralities.insert(\*node, centrality);

}

betweenness\_centralities

}

}

// Function to read the graph from the file

fn read\_graph(file\_path: &str) -> Result<MyGraph, Box<dyn Error>> {

let mut graph = MyGraph::new();

let path = Path::new(file\_path);

let file = File::open(&path)?;

let reader = std::io::BufReader::new(file);

for line in reader.lines() {

let line = line?;

let parts: Vec<\_> = line.split\_whitespace().collect();

if parts.len() == 2 {

let src: usize = parts[0].parse()?;

let dest: usize = parts[1].parse()?;

graph.add\_edge(src, dest);

} else {

return Err(From::from("Each line in the file must contain exactly two numbers"));

}

}

Ok(graph)

}

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

// Read the graph from the file

let file\_path = "facebook\_combined.txt";

let graph = match read\_graph(file\_path) {

Ok(graph) => graph,

Err(err) => {

eprintln!("Error reading graph from file: {}", err);

return Err(err.into());

}

};

// Count degree distribution

let degree\_distribution = graph.degree\_distribution();

// Count number of neighbors at distance 2

let neighbors\_at\_distance\_2 = graph.neighbors\_at\_distance\_2();

// Calculate betweenness centrality

let mut all\_shortest\_paths = Vec::new();

for start\_node in 0..graph.adjacency\_list.len() {

let shortest\_paths = dijkstra\_shortest\_paths(&graph.adjacency\_list, start\_node);

for (node, path) in shortest\_paths.iter().enumerate() {

if !path.is\_empty() {

all\_shortest\_paths.push((start\_node, node, path.clone()));

}

}

}

let betweenness = MyGraph::betweenness\_centrality(&graph.adjacency\_list, &all\_shortest\_paths);

// Sort the nodes by degree centrality in descending order

let mut sorted\_degree\_centrality: Vec<\_> = betweenness.iter().collect();

sorted\_degree\_centrality.sort\_by(|(\_, a), (\_, b)| b.partial\_cmp(a).unwrap());

// Print the 10 nodes with the highest betweenness centrality

println!("Top 10 nodes with highest betweenness centrality:");

for (i, (node, centrality)) in sorted\_degree\_centrality.iter().take(10).enumerate() {

println!("{}. Node {}: {:.4}", i + 1, node, centrality);

}

// Sort the nodes by degree distribution in descending order

let mut sorted\_degree\_distribution: Vec<\_> = degree\_distribution.iter().enumerate().collect();

sorted\_degree\_distribution.sort\_by(|(\_, a), (\_, b)| b.partial\_cmp(a).unwrap());

// Print the 10 nodes with the highest degree distribution

println!("Top 10 nodes with highest degree distribution:");

for (i, (vertex, degree)) in sorted\_degree\_distribution.iter().take(10).enumerate() {

println!("{}. Node {}: {}", i + 1, vertex, degree);

}

// Sort the nodes by number of neighbors at distance 2 in descending order

let mut sorted\_neighbors\_at\_distance\_2: Vec<\_> = neighbors\_at\_distance\_2.iter().enumerate().collect();

sorted\_neighbors\_at\_distance\_2.sort\_by(|(\_, a), (\_, b)| b.partial\_cmp(a).unwrap());

// Print the 10 nodes with the most neighbors at distance 2

println!("Top 10 nodes with most neighbors at distance 2:");

for (i, (vertex, neighbors)) in sorted\_neighbors\_at\_distance\_2.iter().take(10).enumerate() {

println!("{}. Node {}: {}", i + 1, vertex, neighbors);

}

Ok(())

}

fn dijkstra\_shortest\_paths(adj\_list: &[Vec<usize>], start\_node: usize) -> Vec<Vec<usize>> {

let num\_nodes = adj\_list.len();

let mut distances = vec![usize::MAX; num\_nodes];

let mut shortest\_paths = vec![vec![]; num\_nodes];

distances[start\_node] = 0;

let mut heap = BinaryHeap::new();

heap.push(Reverse((0, start\_node)));

while let Some(Reverse((dist, node))) = heap.pop() {

if dist > distances[node] {

continue;

}

for &next\_node in &adj\_list[node] {

let next\_dist = dist + 1;

if next\_dist < distances[next\_node] {

distances[next\_node] = next\_dist;

heap.push(Reverse((next\_dist, next\_node)));

shortest\_paths[next\_node] = vec![node];

} else if next\_dist == distances[next\_node] {

shortest\_paths[next\_node].push(node);

}

}

// Update the shortest paths in reverse direction (for undirected graph)

for &prev\_node in adj\_list[node].iter() {

let prev\_dist = dist + 1;

if prev\_dist < distances[prev\_node] {

distances[prev\_node] = prev\_dist;

heap.push(Reverse((prev\_dist, prev\_node)));

shortest\_paths[prev\_node] = vec![node];

} else if prev\_dist == distances[prev\_node] {

shortest\_paths[prev\_node].push(node);

}

}

}

// Backtrack to construct the actual shortest paths

for node in 0..num\_nodes {

if shortest\_paths[node].len() > 0 {

let mut path = vec![node];

let mut current\_node = node;

while current\_node != start\_node {

let prev\_node = shortest\_paths[current\_node][0];

path.insert(0, prev\_node);

current\_node = prev\_node;

}

shortest\_paths[node] = path;

}

}

// Remove duplicate nodes in paths

for path in &mut shortest\_paths {

path.dedup();

}

shortest\_paths

}