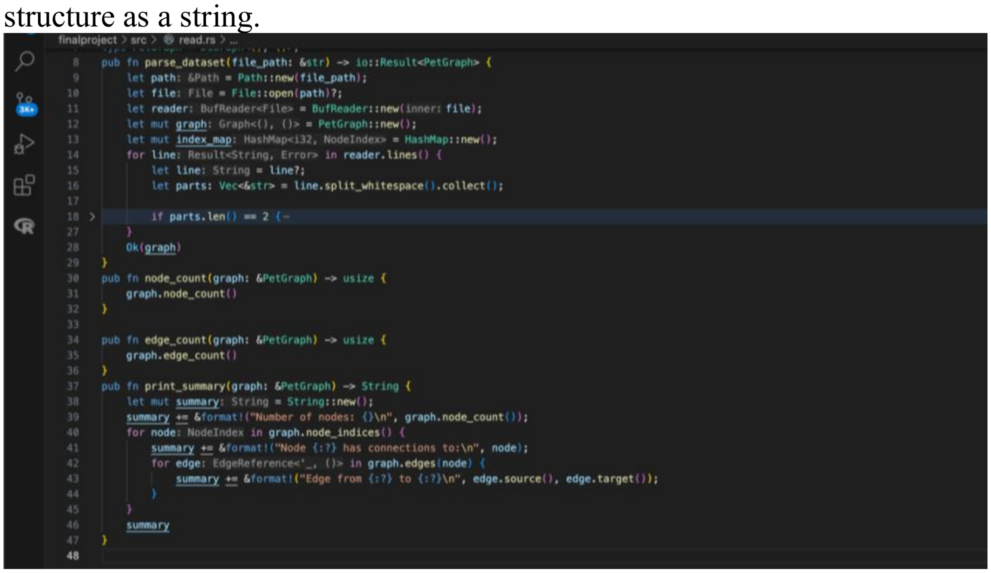
Analyzing Wikipedia Administrator Elections Using Rust

Purpose: The purpose of this project was to utilize Rust to analyze the Wikipedia vote network dataset and more specifically the elections and voting behavior in the Wikipedia community. I did this using the Breadth First Search algorithm and centrality measures.

Data: This dataset contained Wikipedia voting data from the establishment of Wikipedia up until January 2008. It contains voter history data. There are 7115 nodes and 103689 edges. The nodes in the network represent a Wikipedia user. The data is directed. An example of what this means is that there is a directed edge from node a to node b. In terms of the data this means user a voted for user b.

I divided my code into four modules: main.rs, alg.rs, alg2.rs, and read.rs. Each module has several dependencies that were imported for working graphs, parallel processing, efficient set operations, storing data, etc.

read.rs
This module provides functions to read data from a file and construct a directed graph. It also prints node and edge counts that I used to ensure that my data was being processed correctly. The parse dataset function reads data from a file and then constructs a directed graph based on the data. The print summary function refers to a graph and then creates a summary of the graph's structure as a string.

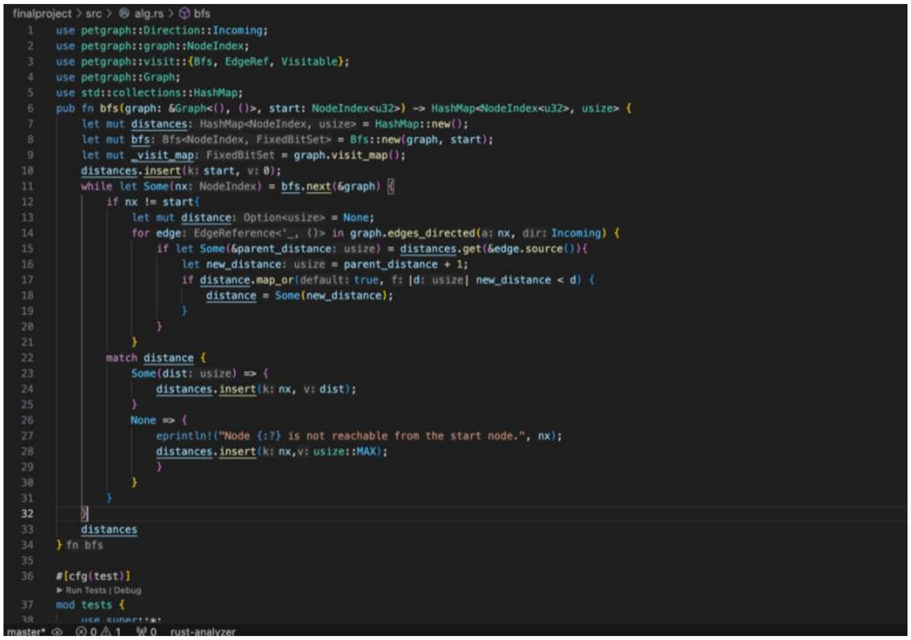



```rust
pub fn parse_dataset(file_path: &str) -> io::Result<PetGraph> {
    let path: &Path = Path::new(file_path);
    let file: File = File::open(path)?;
    let reader: BufReader<File> = BufReader::new(inner: file);
    let mut graph: Graph<(), ()> = PetGraph::new();
    let mut index_map: HashMap<i32, NodeIndex> = HashMap::new();
    for line: Result<String, Error> in reader.lines() {
        let line: String = line?;
        let parts: Vec<&str> = line.split_whitespace().collect();

        if parts.len() == 2 {…
    }
    Ok(graph)
}
pub fn node_count(graph: &PetGraph) -> usize {
    graph.node_count()
}

pub fn edge_count(graph: &PetGraph) -> usize {
    graph.edge_count()
}
pub fn print_summary(graph: &PetGraph) -> String {
    let mut summary: String = String::new();
    summary += &format!("Number of nodes: {}\n", graph.node_count());
    for node: NodeIndex in graph.node_indices() {
        summary += &format!("Node {:?} has connections to:\n", node);
        for edge: EdgeReference<'_, ()> in graph.edges(node) {
            summary += &format!("Edge from {:?} to {:?}\n", edge.source(), edge.target());
        }
    }
    summary
}
```

alg.rs
This module provides a 'bfs' function. It performs a breadth first search algorithm on a directed graph. It starts from a specific node and then calculates the distance from that node to all the other nodes that are reachable. It also has a test to verify the correctness of the BFS implementation. It does this by comparing the expected distances with the actual distances in a test graph.

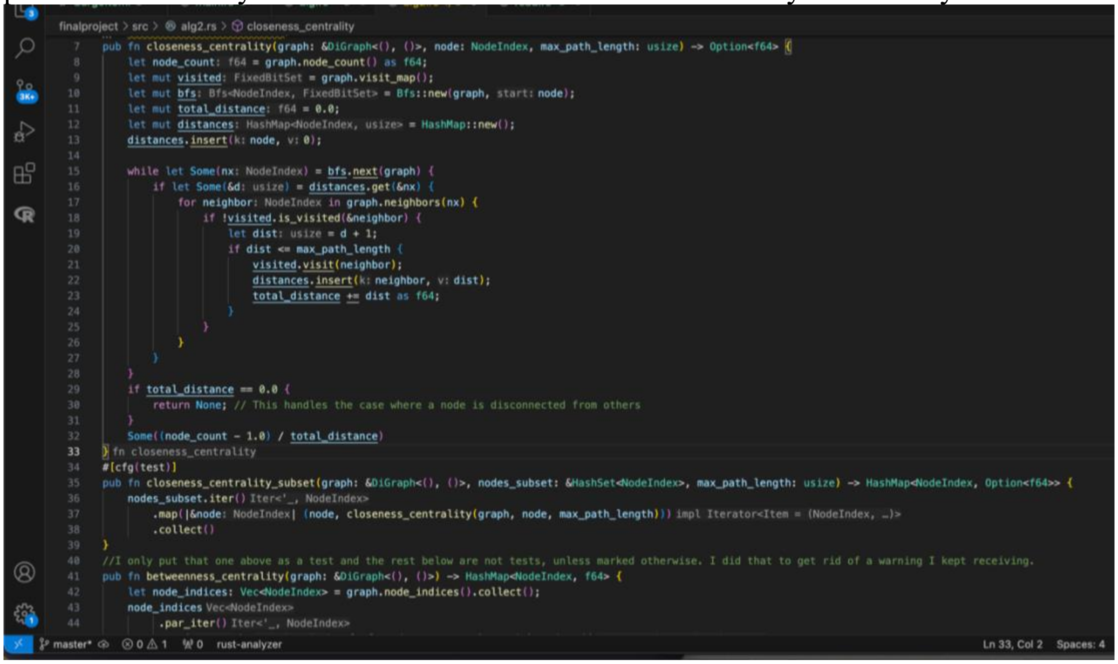


```
finalproject > src > alg.rs > bfs
1   use petgraph::Direction::Incoming;
2   use petgraph::graph::NodeIndex;
3   use petgraph::visit::{Bfs, EdgeRef, Visitable};
4   use petgraph::Graph;
5   use std::collections::HashMap;
6   pub fn bfs(graph: &Graph<(), ()>, start: NodeIndex<u32>) -> HashMap<NodeIndex<u32>, usize> {
7       let mut distances: HashMap<NodeIndex, usize> = HashMap::new();
8       let mut bfs: Bfs<NodeIndex, FixedBitSet> = Bfs::new(graph, start);
9       let mut _visit_map: FixedBitSet = graph.visit_map();
10      distances.insert(k: start, v: 0);
11      while let Some(nx: NodeIndex) = bfs.next(&graph) {
12          if nx != start{
13              let mut distance: Option<usize> = None;
14              for edge: EdgeReference<'_, ()> in graph.edges_directed(a: nx, dir: Incoming) {
15                  if let Some(&parent_distance: usize) = distances.get(&edge.source()){
16                      let new_distance: usize = parent_distance + 1;
17                      if distance.map_or(default: true, f: |d: usize| new_distance < d) {
18                          distance = Some(new_distance);
19                      }
20                  }
21              }
22          match distance {
23              Some(dist: usize) => {
24                  distances.insert(k: nx, v: dist);
25              }
26              None => {
27                  eprintln!("Node {:?} is not reachable from the start node.", nx);
28                  distances.insert(k: nx,v: usize::MAX);
29                  }
30              }
31          }
32      }
33      distances
34  } fn bfs
35
36  #[cfg(test)]
    ▶ Run Tests | Debug
37  mod tests {
38      use super::*;
master*  ⊗ 0 ⚠ 1  ⚡ 0  rust-analyzer
```

The 6th line performs a BFS traversal on a graph from a specific node which is marked by 'start.' It then returns a HashMap with the distances from the start node to all the reachable nodes.
In line 11, for each node visited excluding the start node it initializes the distance as none, then iterates through the incoming edges and calculates the distance based on the parent node's distance. If a shorter distance is found it will update it and if it is not reachable the code will print that as well. There is also a test module which created a directed graph with nodes and edges and asserts that the distances match the expected value based on the graph's structure.

alg2.rs

This module provides functions to calculate closeness centrality and betweenness centrality for nodes in a directed graph. It also includes unit tests to validate the correctness of these functions. The closeness centrality function calculates the closeness centrality of a specified node in a directed graph. Closeness centrality measures how close a node is to all other nodes in a graph. This function calculates centrality based on the lengths of these paths and returns it as an 'Option<f64>' if the node is disconnected then it returns none. This is in lines 7-32. I had to make a function called closeness_centrality_subset which calculates closeness centrality for a subset of nodes in a graph. I did this because it was taking too long to run, and I found that having these functions sped up the process. The betweenness centrality function used parallel processing to calculate the betweenness centrality for each node then reduced the results using the merge centralities function. I used a variant of the Dijkstra's algorithm to compute shortest paths and centrality values. This module also had test to verify the validity of the function.

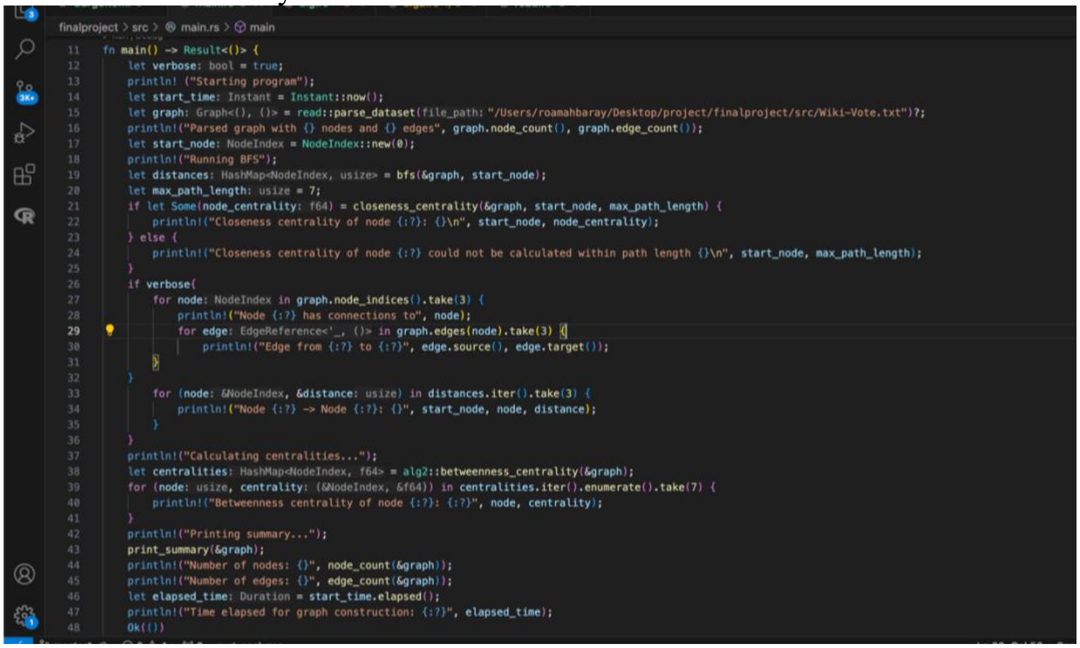


```
finalproject > src > alg2.rs > closeness_centrality
pub fn closeness_centrality(graph: &DiGraph<(), ()>, node: NodeIndex, max_path_length: usize) -> Option<f64> {
    let node_count: f64 = graph.node_count() as f64;
    let mut visited: FixedBitSet = graph.visit_map();
    let mut bfs: Bfs<NodeIndex, FixedBitSet> = Bfs::new(graph, start: node);
    let mut total_distance: f64 = 0.0;
    let mut distances: HashMap<NodeIndex, usize> = HashMap::new();
    distances.insert(k: node, v: 0);

    while let Some(nx: NodeIndex) = bfs.next(graph) {
        if let Some(&d: usize) = distances.get(&nx) {
            for neighbor: NodeIndex in graph.neighbors(nx) {
                if !visited.is_visited(&neighbor) {
                    let dist: usize = d + 1;
                    if dist <= max_path_length {
                        visited.visit(neighbor);
                        distances.insert(k: neighbor, v: dist);
                        total_distance += dist as f64;
                    }
                }
            }
        }
    }
    if total_distance == 0.0 {
        return None; // This handles the case where a node is disconnected from others
    }
    Some((node_count - 1.0) / total_distance)
} fn closeness_centrality
#[cfg(test)]
pub fn closeness_centrality_subset(graph: &DiGraph<(), ()>, nodes_subset: &HashSet<NodeIndex>, max_path_length: usize) -> HashMap<NodeIndex, Option<f64>> {
    nodes_subset.iter() Iter<'_, NodeIndex>
        .map(|&node: NodeIndex| (node, closeness_centrality(graph, node, max_path_length))) impl Iterator<Item = (NodeIndex, …)>
        .collect()
}
//I only put that one above as a test and the rest below are not tests, unless marked otherwise. I did that to get rid of a warning I kept receiving.
pub fn betweenness_centrality(graph: &DiGraph<(), ()>) -> HashMap<NodeIndex, f64> {
    let node_indices: Vec<NodeIndex> = graph.node_indices().collect();
    node_indices Vec<NodeIndex>
        .par_iter() Iter<'_, NodeIndex>
master* ⊗ 0 ⚠ 1 ⚡0 rust-analyzer Ln 33, Col 2 Spaces: 4
```

main.rs

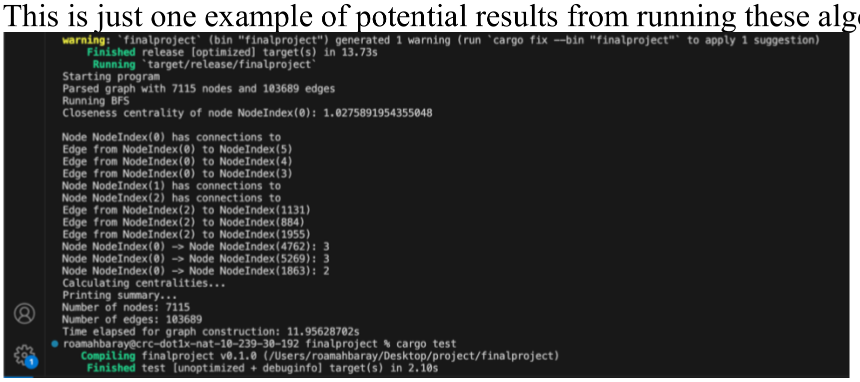
This file reads a graph dataset, performs BFS traversal, calculates centrality measures, and prints various statistics/information about the graph. I declared the three modules I implemented in separate files. I had it record the current time for performance measurement which is in the 14th line. I had to do this because I was having trouble with efficiency. Then the dataset is read and parsed and a graph is created. Again, I had the number of nodes and edges printed to make sure that my graph was parsed properly. BFS traversal is performed from a specified node and calculates the distances to other nodes. I had to put a limit on how many were printed otherwise the function would take too long to run. Then the closeness centrality is calculated and printed as well as other summary statistics.



finalproject > src > main.rs > main

```rust
fn main() -> Result<()> {
    let verbose: bool = true;
    println! ("Starting program");
    let start_time: Instant = Instant::now();
    let graph: Graph<(), ()> = read::parse_dataset(file_path: "/Users/roamahbaray/Desktop/project/finalproject/src/Wiki-Vote.txt")?;
    println!("Parsed graph with {} nodes and {} edges", graph.node_count(), graph.edge_count());
    let start_node: NodeIndex = NodeIndex::new(0);
    println!("Running BFS");
    let distances: HashMap<NodeIndex, usize> = bfs(&graph, start_node);
    let max_path_length: usize = 7;
    if let Some(node_centrality: f64) = closeness_centrality(&graph, start_node, max_path_length) {
        println!("Closeness centrality of node {:?}: {}\n", start_node, node_centrality);
    } else {
        println!("Closeness centrality of node {:?} could not be calculated within path length {}\n", start_node, max_path_length);
    }
    if verbose{
        for node: NodeIndex in graph.node_indices().take(3) {
            println!("Node {:?} has connections to", node);
            for edge: EdgeReference<'_, ()> in graph.edges(node).take(3) {
                println!("Edge from {:?} to {:?}", edge.source(), edge.target());
            }
        }
        for (node: &NodeIndex, &distance: usize) in distances.iter().take(3) {
            println!("Node {:?} -> Node {:?}: {}", start_node, node, distance);
        }
    }
    println!("Calculating centralities...");
    let centralities: HashMap<NodeIndex, f64> = alg2::betweenness_centrality(&graph);
    for (node: usize, centrality: (&NodeIndex, &f64)) in centralities.iter().enumerate().take(7) {
        println!("Betweenness centrality of node {:?}: {:?}", node, centrality);
    }
    println!("Printing summary...");
    print_summary(&graph);
    println!("Number of nodes: {}", node_count(&graph));
    println!("Number of edges: {}", edge_count(&graph));
    let elapsed_time: Duration = start_time.elapsed();
    println!("Time elapsed for graph construction: {:?}", elapsed_time);
    Ok(())
```

Results:
This is just one example of potential results from running these algorithms on this dataset.



```
warning: `finalproject` (bin "finalproject") generated 1 warning (run `cargo fix --bin "finalproject"` to apply 1 suggestion)
    Finished release [optimized] target(s) in 13.73s
     Running `target/release/finalproject`
Starting program
Parsed graph with 7115 nodes and 103689 edges
Running BFS
Closeness centrality of node NodeIndex(0): 1.0275891954355048

Node NodeIndex(0) has connections to
Edge from NodeIndex(0) to NodeIndex(5)
Edge from NodeIndex(0) to NodeIndex(4)
Edge from NodeIndex(0) to NodeIndex(3)
Node NodeIndex(1) has connections to
Node NodeIndex(2) has connections to
Edge from NodeIndex(2) to NodeIndex(1131)
Edge from NodeIndex(2) to NodeIndex(884)
Edge from NodeIndex(2) to NodeIndex(1955)
Node NodeIndex(0) -> Node NodeIndex(4762): 3
Node NodeIndex(0) -> Node NodeIndex(5269): 3
Node NodeIndex(0) -> Node NodeIndex(1863): 2
Calculating centralities...
Printing summary...
Number of nodes: 7115
Number of edges: 103689
Time elapsed for graph construction: 11.95628702s
roamahbaray@crc-dot1x-nat-10-239-30-192 finalproject % cargo test
   Compiling finalproject v0.1.0 (/Users/roamahbaray/Desktop/project/finalproject)
    Finished test [unoptimized + debuginfo] target(s) in 2.10s
```

The results indicate that the program successfully parsed the dataset from the file. It has 7115 nodes and 103689 edges which is what it is supposed to be. The BFS traversal was initiated starting from the first node. The closeness centrality of NodeIndex(0) is 1.0276. A higher value suggests that NodeIndex(0) is relatively close to other nodes. Then the program prints information about the connections of the first three nodes, like Node 0 has connections to Node 5, 4, and 3. It also displayed the distances from NodeIndex 0 to specific nodes. In order to run this, I used cargo run –release. I also found it helpful to use cargo clean and cargo build before running the code. At first I was using just cargo run but that was not efficient so I did cargo run –release.

I chose these methods because it is commonly used in social network analysis and graph theory which is interesting to me. I learned about how to implement these algorithms. I ran into a lot of issues with the run time as my laptop is very slow and old. That made it difficult to implement these algorithms, however they are now successfully running.