

The question I am answering is: How does a user's closeness centrality relate to their ability to influence other users within a limited number of steps? What is the relationship between closeness centrality and reach of a node in a social network graph?

<https://snap.stanford.edu/data/feather-lastfm-social.html>

I loaded the data using my `read_graph` function in `main.rs`, using the `petgraph` library in Rust.

Code Structure

`Main.rs` has my main function that runs the methods and functions of `centrality.rs` and `graph.rs`. It loads the graph, calculates connected components, calculates closeness centrality, and compares the reach of users based on their centrality.

`graph.rs` contains the `Graphs` struct and its `impl` block which contains the methods of graph algorithms of BFS, closeness centrality, connected components, and finding the number of users a user can reach. It also has the `read_graph` function to load an undirected graph given a CSV file.

`centrality.rs` contains functions for computing centrality-related analysis. It has functions to find the most and least central user of a component, to find the largest component, and to find the most and least central users of multiple components.

Key Functions & Types (Structs, Enums, Traits, etc)

For each non-trivial item, restate its purpose, inputs and outputs, and **Core logic and key components**

How do modules/functions interact to produce your results?

`Central_finder` finds the most central user in each connected component. Input is a list of connected components (a list of node numbers) and a `HashMap` that contains (user, closeness centrality score.) pairs. Output is a vector containing the most central nodes from each component. For each component in the list of components, the user with the highest closeness score is found and added to a vector containing all found users at the end of the loop.

`least_central_finder` finds the least central user in each connected component. Input is a list of connected components (a list of node numbers) and a `HashMap` that contains (user, closeness centrality score) pairs. Output is a vector containing the least central nodes from each component. For each component, the user with the lowest closeness score is found and added to a vector that stores all such users.

`most_central_user` finds the most central user in a single connected component. Input is a vector of node numbers (a single component) and a `HashMap` of (user, closeness centrality

score) pairs. Output is the user with the highest closeness score. The function loops through the users in the component and keeps track of the one with the highest score.

least_central_user finds the least central user in a single connected component. Input is a vector of node numbers (a single component) and a HashMap of (user, closeness centrality score) pairs. Output is the user with the lowest closeness score. The function loops through the users in the component and keeps track of the one with the lowest score.

largest_component returns the largest component from a list of connected components. Input is a list of components (each a vector of node numbers). Output is the component with the most users. The function compares the size of each component and returns the one with the largest length.

Graphs::new creates a new Graphs object from a Petgraph graph. Input is a Graph<u32, (), Undirected> and output is a Graphs struct containing the graph. The function returns a struct where the graph is stored as a field.

Graphs::bfs performs BFS from a starting node for a given number of steps. Input is the starting node number and a step limit. Output is a vector of visited node numbers. The function uses a queue to explore the graph from the starting node and stops when the step count is reached.

Graphs::closeness_centrality calculates the closeness centrality for every node in the graph. The input is the graph (self). Output is a HashMap with (node number, closeness score) pairs. The function runs a BFS from each node, calculates the total distance to all other nodes, and applies the closeness formula.

Graphs::connected_components finds all connected components in the graph. Input is the graph (self). Output is a vector of components, and each component is a list of node numbers. The function runs BFS from every unvisited node and grouped the reached nodes into components.

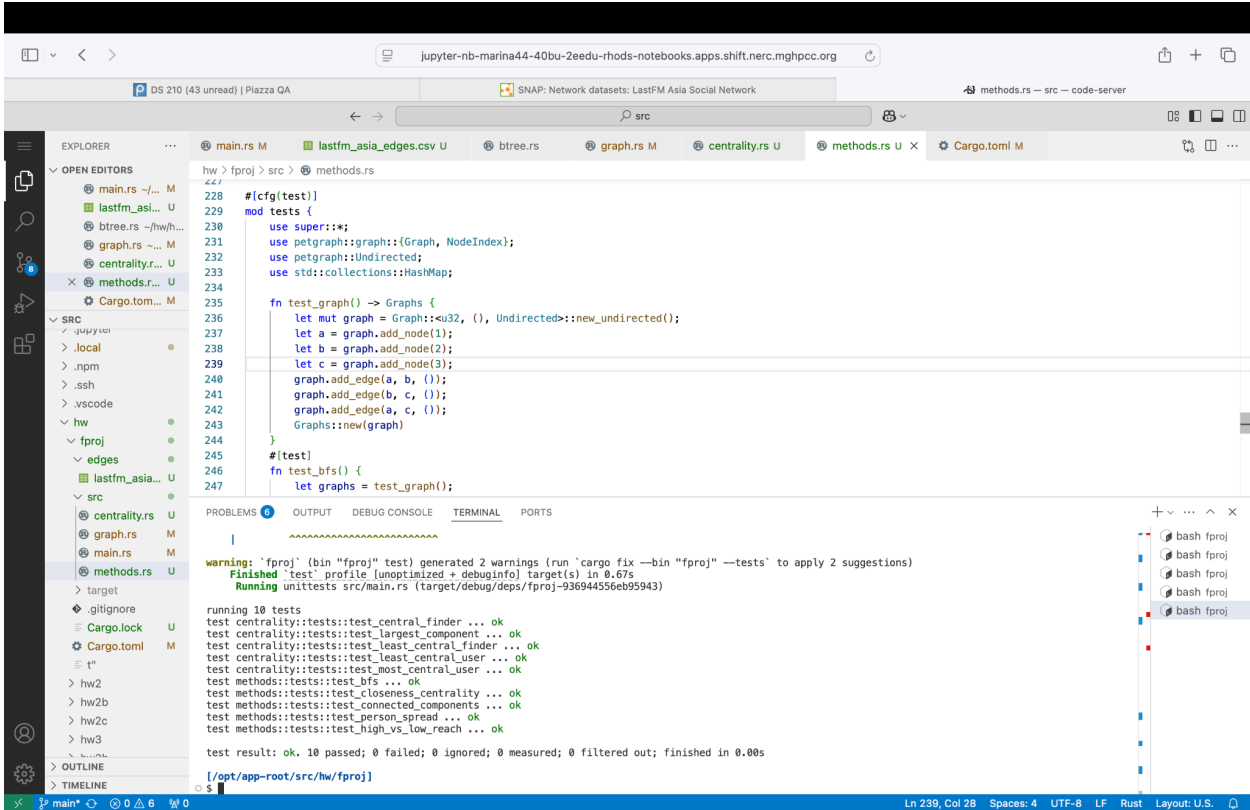
Graphs::person_spread finds how many people a given user can reach in a certain number of steps. Input is a user's node number and a number of steps. Output is the number of unique nodes reached. The function uses a limited-depth BFS to find the number of reachable nodes within the step limit.

Graphs::high_vs_low_reach compares the reach of the most central users and the least central users within each component. Input is a list of components, a HashMap of (node, closeness score) pairs, and a number of steps. Output prints the average number of users reached by the most and least central users. The function uses central_finder and least_central_finder to find the target users and then calls person_spread to calculate how many users they can reach.

read_graph loads a graph from a CSV file. Input is a file path. Output is a Petgraph Graph<u32, (), Undirected> with the nodes and edges from the file. The function reads each line, parses the

node numbers, creates the graph using a HashMap to track existing nodes, and connects them with edges.

Tests



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor in the center. The code editor displays Rust code for testing a graph library. The code defines a test module with a function `test_graph` that creates a graph with 3 nodes and 3 edges. It then runs a series of tests to verify the graph's properties.

```
228 #[cfg(test)]
229 mod tests {
230     use super::*;
231     use petgraph::graph::Graph, NodeIndex;
232     use petgraph::Undirected;
233     use std::collections::HashMap;
234
235     fn test_graph() -> Graphs {
236         let mut graph = Graph::new(3, Undirected::new_undirected());
237         let a = graph.add_node(1);
238         let b = graph.add_node(2);
239         let c = graph.add_node(3);
240         graph.add_edge(a, b, ());
241         graph.add_edge(b, c, ());
242         graph.add_edge(a, c, ());
243         Graphs::new(graph)
244     }
245
246     #[test]
247     fn test_bfs() {
248         let graphs = test_graph();
249     }
250 }
```

The terminal output shows the results of running the tests:

```
running 10 tests
test centrality::tests::test_central_finder ... ok
test centrality::tests::test_largest_component ... ok
test centrality::tests::test_least_central_finder ... ok
test centrality::tests::test_least_central_user ... ok
test centrality::tests::test_most_central_user ... ok
test methods::tests::test_bfs ... ok
test methods::tests::test_closeness_centrality ... ok
test methods::tests::test_connected_components ... ok
test methods::tests::test_person_spread ... ok
test methods::tests::test_high_vs_low_reach ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

[/opt/app-root/src/hw/fproj]
```

test_central_finder

Makes sure that the most central user found in each connected component is accurate, and has a component [1,2,3] with node 2 having the highest closeness score.

test_least_central_finder

Makes sure that the least central user found in each connected component is accurate, and has a component [1,2,3] with node 3 having the lowest closeness score.

Test_most_central_user

Makes sure that the most central user found within a single component is correct.

test_least_central_user

Makes sure that the least central user found within a single component is correct.

test_largest_component

Makes sure that the component with the most nodes is correctly identified from a list of components.

test_graph creates a new graph.

test_bfs

Makes sure that BFS visits all reachable nodes from a given starting point in 1 step.

test_closeness centrality

Makes sure that closeness centrality is calculated correctly.

test_connected_components

Makes sure that a graph is correctly classified as one connected component.

test_person_spread

Makes sure that the number of users a person can reach in a given number of steps found is accurate, using a node 1 in a graph which can reach nodes 2 and 3 in 1 step.

test_high_vs_low_reach

Makes sure that the high_vs_low_reach function works properly.

Results

The screenshot displays a Jupyter Notebook environment with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like `main.rs`, `lastfm_asi...`, `btrees.rs`, `graph.rs`, `centrality.rs`, `methods.rs`, and `Cargo.toml`. The code editor shows the `methods.rs` file with Rust code for graph analysis. The code includes comments and function definitions for calculating the largest component, BFS, closeness centrality, and person spread. The execution output shows the results of running the code, including the number of components, the number of users reached by the most and least central users, and the average number of users reached by the most central users.

```
hw > fproj > src > methods.rs
26 //output is a vector of visited nodes
198 //within a given amount of steps
199 //the input is a list of connected components
200 //a hash map of (node number, score) pairs, and the number of steps
201 //the output prints the average number of users reached by the most and least central users
202 pub fn high_vs_low_reach(&self, components: &Vec<Vec<u32>>, closeness_map: &HashMap<u32, f64>, steps: u32) {
203     let most_central = central_finder(components, closeness_map);
204     let least_central = least_central_finder(components, closeness_map);
205
206     let mut most_sum = 0;
207     let mut least_sum = 0;
208     let num_components = most_central.len().min(least_central.len());
209
210     //within each component, determine the number of users each can reach
211     for i in 0..num_components {
212         let most_reach = self.person_spread(most_central[i], steps);
213         let least_reach = self.person_spread(least_central[i], steps);
214         most_sum += most_reach;
215         least_sum += least_reach;
216     }
217
218     let most_avg = most_sum as f64 / num_components as f64;
219     let least_avg = least_sum as f64 / num_components as f64;
220
221     println!("In {} steps, the most central users reached {} people on average.", steps, most_avg);
222 }
223
224 https://doc.rust-lang.org/cargo/reference/profiles.html#default-profiles (cmd + click)
error: could not compile `fproj` (bin `fproj`) due to 4 previous errors
[./opt/app-root/src/hw/fproj]
$ cargo build
Compiling fproj v0.1.0 (/opt/app-root/src/hw/fproj)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.40s
[./opt/app-root/src/hw/fproj]
$ cargo run
Running `target/debug/fproj`
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
In 3 steps, the most central users reached 11.66 people on average.
In 3 steps, the least central users reached 4.61 people on average.
The most central user reached 191 people during BFS in 3 steps
The least central user reached 9 people during BFS in 3 steps
[./opt/app-root/src/hw/fproj]
```

reached 191 people and the least central user reached 9 people during BFS given 3 steps, showing that centrality plays a very significant role in the reach of a node within a graph.

To build and run the code: Cargo build, cargo run

The runtime should be near instant. I took a sample of the first 1000 rows of the dataset, which cut down on run time.

AI-Assistance Disclosure and Other Citations

I used chatgpt for assistance on syncing Github with Rust. I learned how to set up an ssh key by writing the following code in the terminal

```
ssh-keygen -t ed25519 -C "your\_email@example.com"
```

```
git remote set-url origin git@github.com:your-username/your-repo-name.git
```

cat ~/.ssh/id_ed25519.pub. And pasting the output of this pasted into the terminal into “New SSH key” on Github in Settings.

Out of class resources I used:

https://john-cd.com/rust_howto/categories/data-structures/graph.html

<https://docs.rs/petgraph/latest/petgraph/graph/struct.Graph.html>

<https://depth-first.com/articles/2020/02/03/graphs-in-rust-an-introduction-to-petgraph>

<https://crates.io/crates/petgraph>