DS 210 Project Write up

This project analyzes the Gnutella peer-to-peer (P2P) network using graph-based centrality measures to identify the most important nodes in terms of communication efficiency and traffic bridging. Specifically, we compute **closeness** and **betweenness** centrality scores for each node to understand their influence and structural role in the network.

Used ChatGPT to create a framework for the project and generate ideas on how to tackle the analysis.

Dataset:

- **Source:** SNAP (Stanford Large Network Dataset Collection)
- **Name:** Gnutella P2P network from August 8, 2002
- **Format:** TSV with directed edges (FromNodeId, ToNodeId)
- **Size:** 6,301 nodes and 20,777 edges
- **Link:** https://snap.stanford.edu/data/p2p-Gnutella08.html
- **File Used:** `data/p2p-Gnutella08.txt`

Data Processing:

Loading data set:
The dataset `p2p-Gnutella08.txt` was in tab separated form with each line representing a directed edge in the form:

FromNodeID ToNodeID

Metadata was skipped and the rest of the lines were parsed into directed edges. Each one represented a connection between 2 peers in the Gnutella network. Used buffer reader (BufReader) to read the file line by line. For each line, the following was done. Whitespace Split: Split on whitespace to extract the two node IDs. Type Conversion: Parsed the node IDs from `&str` to `u32`. Node Mapping: Used a `HashMap<u32, NodeIndex>` to assign each unique raw node ID to a `petgraph::graph::NodeIndex` used internally by the graph structure. Graph

Construction: Inserted each edge into a `petgraph::DiGraph<u32, ()>` — a directed graph with nodes labeled by their original `u32` IDs and no edge weights. This approach guaranteed thatNodes are uniquely indexed and reused. The graph reflects the true topology of the original dataset. Duplicate or malformed lines are safely ignored.

Graph representation:

The graph was stored using `petgraph::DiGraph<u32, ()>`. Each node stores its original `u32` ID, enabling reverse mapping for interpretation later. The graph is **directed**, as P2P queries are unidirectional (a node may connect to another without reciprocation). Edge weights were unused (set to null), since only structural analysis was performed

After processing all of the information, the graph was loaded and became the base for analysing centrality and calculations and visualization. The total nodes were 6301 and the total edges were 20,777.

1. `Graph.rs` — Graph Construction and Data Loading

Purpose:
To load the Gnutella dataset from a file and construct a directed graph using `petgraph`.

```
pub type Graph = DiGraph<u32, ()>;
```

- Represents the network graph with `u32` node labels and unit-weight edges.

Function:

pub fn load_graph(path: &str) -> (Graph, HashMap<u32, NodeIndex>)

Input is a path to the data file

Output is a Graph and a map of raw node IDs to internal node indices

```rust
pub fn compute_closeness(graph: &Graph) -> HashMap<NodeIndex, f64> {
    let n = graph.node_count();
    let mut scores = HashMap::new();

    for node in graph.node_indices() {
        let mut visited = HashMap::new();
        let mut queue = VecDeque::new();

        visited.insert(node, 0);
        queue.push_back(node);

        while let Some(current) = queue.pop_front() {
            let dist = visited[&current];
            for neighbor in graph.neighbors(current) {
                if !visited.contains_key(&neighbor) {
                    visited.insert(neighbor, dist + 1);
                    queue.push_back(neighbor);
                }
            }
        }
```

```
        let reachable = visited.len();
        let total_distance: usize = visited.values().sum();

        let closeness = if reachable > 1 && total_distance > 0 {
            (reachable as f64 - 1.0) / total_distance as f64
        } else {
            0.0
        };

        println!(
            "Node {}: reachable = {}, total_dist = {}, closeness = {}",
            node.index(),
            reachable,
            total_distance,
            closeness
        );

        scores.insert(node, closeness);
    }

    scores
}
```

Input: Graph

Output: Map of node → closeness score

Core Logic: For each node, perform BFS to compute shortest paths to all reachable nodes. Unreachable nodes are handled gracefully with a score of 0.0.

Compute betweenness:

```rust
pub fn compute_betweenness(graph: &Graph) -> HashMap<NodeIndex, f64> {

   let mut betweenness = HashMap::new();



   for s in graph.node_indices() {

       let mut stack = Vec::new();

       let mut pred: HashMap<NodeIndex, Vec<NodeIndex>> = HashMap::new();

       let mut sigma: HashMap<NodeIndex, usize> = HashMap::new();

       let mut dist: HashMap<NodeIndex, isize> = HashMap::new();



       for v in graph.node_indices() {

           pred.insert(v, Vec::new());

           sigma.insert(v, 0);

           dist.insert(v, -1);

       }



       sigma.insert(s, 1);

       dist.insert(s, 0);
```

```
    let mut queue = VecDeque::new();
    queue.push_back(s);

    while let Some(v) = queue.pop_front() {
        stack.push(v);
        for w in graph.neighbors(v) {
            if dist[&w] < 0 {
                queue.push_back(w);
                dist.insert(w, dist[&v] + 1);
            }
            if dist[&w] == dist[&v] + 1 {
                sigma.insert(w, sigma[&w] + sigma[&v]);
                pred.get_mut(&w).unwrap().push(v);
            }
        }
    }

    let mut delta: HashMap<NodeIndex, f64> = HashMap::new();
    for v in graph.node_indices() {
        delta.insert(v, 0.0);
    }

    while let Some(w) = stack.pop() {
```

```
            for &v in &pred[&w] {
                let ratio = (sigma[&v] as f64) / (sigma[&w] as f64);
                delta.insert(v, delta[&v] + ratio * (1.0 + delta[&w]));
            }
            if w != s {
                *betweenness.entry(w).or_insert(0.0) += delta[&w];
            }
        }
    }

    for val in betweenness.values_mut() {
        *val /= 2.0; // normalization for undirected graphs (here we keep it simple)
    }
    betweenness
}
```

Input: Graph

Output: Map of node → betweenness score

Logic:
Implements **Brandes' algorithm**, an efficient way to compute betweenness and for each source node it count shortest paths to all other nodes and accumulate contributions to betweenness of intermediate nodes and normalizes final scores by dividing by 2 (since we're approximating undirected behavior in a directed graph)

visualise.rs

```rust
pub fn draw_bar_chart(

    title: &str,

    filename: &str,

    scores: &HashMap<NodeIndex, f64>,

    top_n: usize,

) -> Result<(), Box<dyn std::error::Error>> {

    let root = BitMapBackend::new(filename, (800, 600)).into_drawing_area();

    root.fill(&WHITE)?;



    let mut sorted: Vec<_> = scores.iter().collect();

    sorted.sort_by(|a, b| b.1.partial_cmp(a.1).unwrap());



    let top_nodes: Vec<_> = sorted.iter().take(top_n).collect();

    let max_score = top_nodes.iter().map(|(_, &v)| v).fold(0./0., f64::max);



    let labels: Vec<String> = top_nodes.iter().map(|(idx, _)|
idx.index().to_string()).collect();

    let values: Vec<f64> = top_nodes.iter().map(|(_, &v)| v).collect();



    let mut chart = ChartBuilder::on(&root)

        .caption(title, ("sans-serif", 30).into_font())

        .margin(40)

        .x_label_area_size(40)

        .y_label_area_size(60)
```

```
    .build_cartesian_2d(
        0..top_n as i32,
        0.0..(max_score * 1.1),
    )?;

chart.configure_mesh()
    .x_labels(top_n)
    .x_label_formatter(&|x| {
        let index = *x as usize;
        if index < labels.len() {
            labels[index].clone()
        } else {
            "".to_string()
        }
    })
    .y_desc("Score")
    .x_desc("Node ID")
    .axis_desc_style(("sans-serif", 18))
    .draw()?;

chart.draw_series(
    values.iter().enumerate().take(top_n).map(|(i, &val)| {
        Rectangle::new(
            [(i as i32, 0.0), ((i + 1) as i32, val)],
```

```
                BLUE.filled(),

            )

        }),

    )?;



    root.present()?;

    Ok(())

}
```

Purpose: To get a visualization of the top nodes based on the centrality metrics.

Inputs:

- Title (for the chart)
  Output filename
- Centrality scores
- Number of top nodes to display

Logic:

Sorts the top N scores then Maps node indices to labels Uses plotters crate to draw bar charts with x-axis as Node ID and y-axis as Score and then saves result as a PNG file

main():

```
fn main() {

    let (graph, id_map) = load_graph("data/p2p-Gnutella08.txt");

    println!("Graph loaded with {} nodes and {} edges", graph.node_count(),
graph.edge_count());



    let closeness = compute_closeness(&graph);
```

```
    let betweenness = compute_betweenness(&graph);



    display_top("Closeness Centrality", &closeness);

    display_top("Betweenness Centrality", &betweenness);

    draw_bar_chart(

        "Top 10 Nodes by Closeness Centrality",

        "closeness_chart.png",

        &closeness,

        10,

    ).unwrap();



    draw_bar_chart(

        "Top 10 Nodes by Betweenness Centrality",

        "betweenness_chart.png",

        &betweenness,

        10,

    ).unwrap();



    println!("Charts saved as closeness_chart.png and betweenness_chart.png");

}
```

Orchestrates the full pipeline: loading the graph, computing metrics, and rendering visualizations. display_top is a helper to print the top 10 results for each centrality type display_top is a helper to print the top 10 results for each centrality type and centrality results are calculated once and used for both display and visualization.

In-Function Comments & Struct Documentation

Throughout the codebase:

Each module starts with a 1-line header comment summarizing its purpose. Each function includes inline comments explaining: What it does, Inputs/outputs, High-level logic (BFS loops, shortest-path accumulation, graph drawing steps)

- **All data structures** like `HashMap<NodeIndex, f64>` are documented in usage to clarify mapping and storage behavior

D. Tests (Expanded)

Testing focused on validating the correctness and integrity of the centrality analysis functions. Since this project involves mathematical computations over graph structures, we designed tests to ensure:

1. Centrality scores fall within reasonable numeric ranges
2. No invalid outputs (e.g., negative values or NaNs)
3. The program remains stable on the full Gnutella dataset

1. Test Location

Tests were implemented directly within main.rs using Rust's built-in #[cfg(test)] module system to keep things simple and self-contained. This eliminates the need for a separate lib.rs or external integration test structure.

2. Output Example

running 2 tests test tests::test_betweenness_nonnegative ... ok test tests::test_closeness_nonnegative ... ok test result: ok. 2 passed; 0 failed; 0 ignored CopyEdit

Actual Output:

```
test tests::test_closeness_nonnegative ... ok
test tests::test_betweenness_nonnegative has been running for over 60 seconds
test tests::test_betweenness_nonnegative ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 104.06s
```

Closeness Validity: Ensures that shortest path aggregation logic is correct, and that scores aren't erroneously flipped negative due to math or type conversion. **Betweenness Stability:** Verifies that the Brandes algorithm implementation accumulates scores correctly across all nodes and doesn't allow underflow or logical bugs. These basic tests act as sanity checks and are especially important when scaling analysis to large real-world networks.

1. Closeness Centrality

Top 10 Nodes by Closeness

Node ID	Closeness Score
5728	1.000
5729	1.000
1043	1.000
4583	1.000
4452	1.000
270	1.000
843	1.000
1363	1.000

975	1.000
439	1.000

Interpretation:

The top-ranked nodes all had a closeness score of 1.0, suggesting they are part of a tightly connected core or reachable subgraph within the network. These nodes likely represent supernodes — well-connected peers that can reach most others with very short paths. The uniformity of scores may reflect a plateau in connectivity: a subset of the graph is tightly interconnected, while other areas are sparse or unreachable. Further improvements (e.g., better normalization or a focus on subgraphs) could help reveal more nuanced variation in closeness.

Node ID	Betweenness Score
7629	1.63×10^9
2501	1.56×10^9
6935	1.41×10^9
13222	1.35×10^9
10072	1.13×10^9
12031	1.04×10^9

4572	0.99×10^9
2875	0.93×10^9
5765	0.90×10^9
3456	0.88×10^9

Interpretation:

These nodes act as critical bridges in the network — information likely flows through them more often than others. The steep drop-off in betweenness scores shows a power-law distribution: a few nodes dominate the traffic paths, while most nodes play a marginal role in routing. From a network resilience perspective, these high-betweenness nodes are vulnerable points. Removing them could significantly fragment or isolate parts of the Gnutella network.
These findings are consistent with P2P architectures from the early 2000s, which often relied on supernodes to maintain efficiency.

Usage Requirements

1. Requirements

- Rust (latest stable version recommended)
- Cargo (comes with Rust)
- Internet connection for initial dependency install

2. Installation Steps

1. Clone or download the project repository.
2. Navigate to the project root directory:
    a. cd ds_project
3. Ensure the dataset is placed in: data/p2p-Gnutella08.txt

4. Build the project (in release mode): cargo build --release
5. Cargo run –release