DS210 Final Project Report : Gossip Spread Simulation on Facebook Graph

By Jaewon Oh

A. Project Overview

- **Goal**: In this project, I wanted to understand how information or gossip spreads through a social network. I decided to simulate this behavior using a real-world Facebook friendship graph and a BFS-based approach in Rust. The main objective was to see how far the gossip reaches from a given node and identify which users (nodes) are the most influential spreaders.
- **Dataset**:
  - Source: SNAP – Stanford Network Analysis Project
  - File Used: facebook_combined.txt
  - Size: 4,039 nodes, 88,234 undirected edges
  - Format: Plain edge list format (u v = edge between nodes u and v)
  - Reason for Use: The dataset is anonymized, widely used in academic research, and large enough to capture realistic network dynamics while still being computationally tractable for BFS-based simulations.

B. Data Processing

I loaded the file using std::fs::read_to_string() and parsed each line to create undirected edges between two users. These were stored in a Graph struct, which uses a HashMap<usize, HashSet<usize>> to represent an adjacency list. No data cleaning was required. The file is already well-formatted, and HashSet automatically prevents duplicate edges.

C. Code Structure

1. **Modules**

   graph.rs : Loads the graph from file and builds the adjacency list using a custom Graph struct

   simulate.rs : Performs the BFS-based gossip spread simulation from a single seed node

   analyze.rs : Runs multiple simulations from random seed nodes to identify and rank the most influential spreaders

   main.rs : Orchestrates graph loading, simulation, analysis, and CSV export

2. **Key Functions & Types**

   The central type is the Graph struct, which stores the adjacency list as a HashMap<usize, HashSet<usize>> to represent an undirected graph. The function load_from_file() reads an edge list and builds this structure by adding bidirectional edges.

   The simulate_spread() function performs BFS from a seed node for a fixed number of steps and returns a vector showing how many new nodes were reached at each step. It uses a queue (VecDeque) and a HashSet for visited tracking.

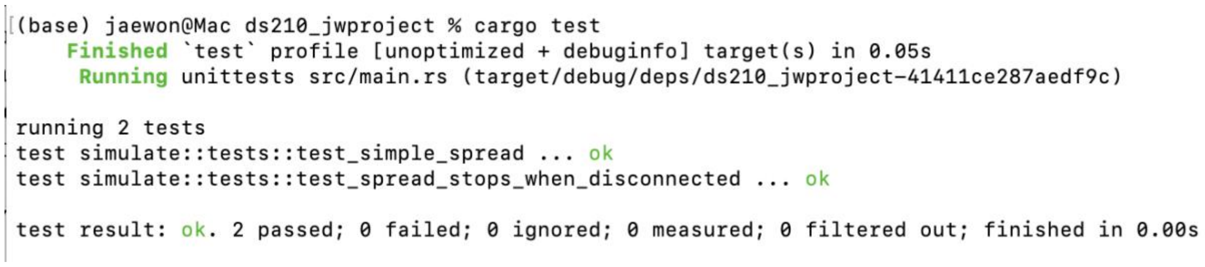
get_random_node() selects a random node from the graph using the rand crate. The find_top_spreaders() function runs multiple spread simulations and ranks nodes by their total reach, helping to identify the most influential ones.

Finally, save_spread_to_csv() writes the spread results to a CSV file for external analysis.

3. **Main Workflow**

- The program starts in main.rs, where I load the Facebook graph using Graph::load_from_file() from graph.rs.
- A random node is selected using get_random_node() from simulate.rs.
- I run simulate_spread() for 6 steps to simulate gossip spreading.
- The results are saved to spread_log.csv using save_spread_to_csv() in main.rs.
- Then I run 50 trials using find_top_spreaders() from analyze.rs to identify the most influential nodes.

D. Tests



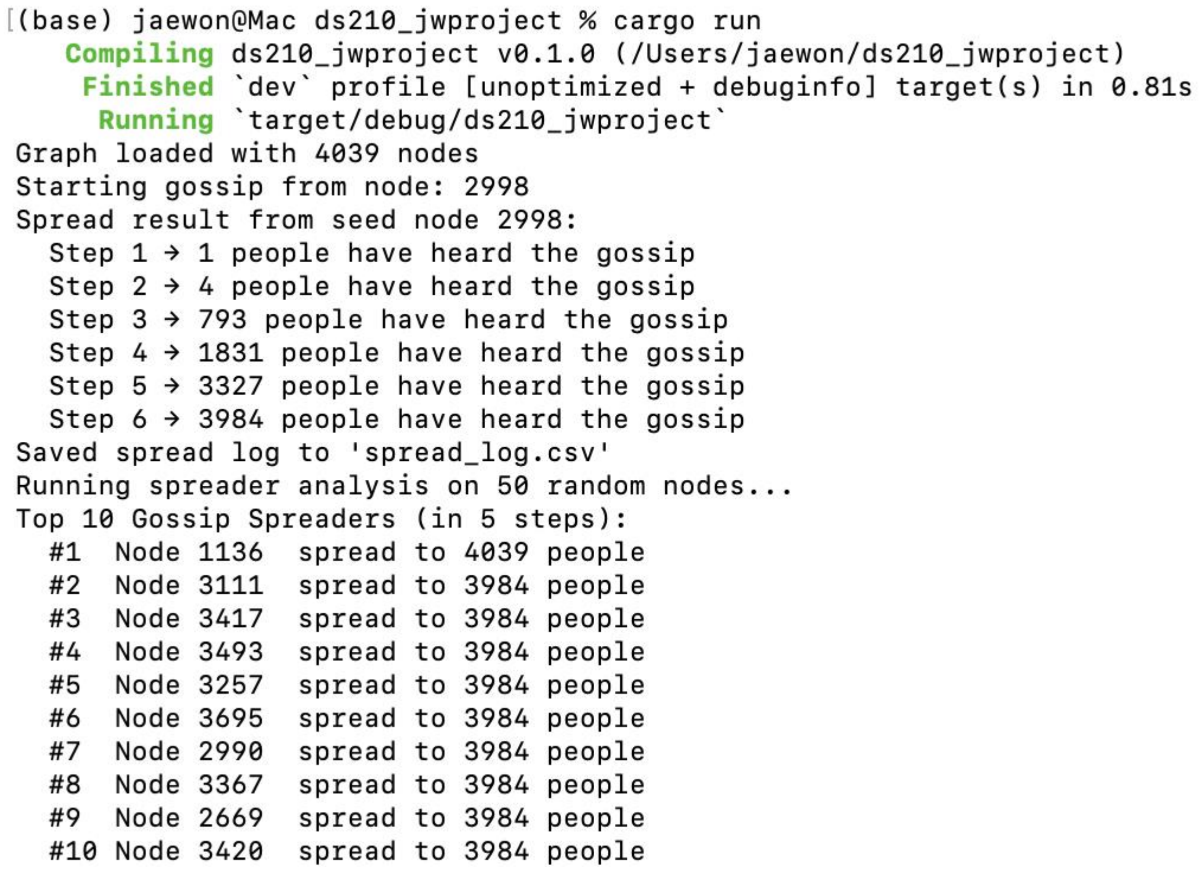
```
(base) jaewon@Mac ds210_jwproject % cargo test
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.05s
     Running unittests src/main.rs (target/debug/deps/ds210_jwproject-41411ce287aedf9c)

running 2 tests
test simulate::tests::test_simple_spread ... ok
test simulate::tests::test_spread_stops_when_disconnected ... ok

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

- **What Tests work** : In order to ensure that my simulation logic works as expected, I implemented two unit tests in simulate.rs. The first test, called test_simple_spread, checks whether the gossip spreads correctly in a small connected graph using Breadth-First Search (BFS). I manually constructed a small graph where the structure and expected result are easy to verify. Starting from a seed node, I run the simulation for a fixed number of steps and check if the number of people reached in each step matches the expected BFS traversal pattern. This helps confirm that the simulation logic handles basic connectivity and step-by-step expansion correctly. If this test fails, it would mean that the BFS might be skipping or revisiting nodes, or the spread count is miscalculated.
  The second test, test_spread_stops_when_disconnected, is designed to check an important edge case: disconnected graphs. I built a graph with two completely separate clusters and initiated gossip from a node in only one of them. This test verifies that the gossip doesn't leak into the other disconnected part of the graph, no matter how many steps the simulation runs. It ensures that the algorithm correctly keeps track of visited nodes and only traverses reachable neighbors. This is important because if the simulation wrongly spreads gossip to unreachable nodes, it would invalidate any real-world analysis I try to make from the results.
  Together, these two tests help me verify that the simulation logic is both correct and robust under simple and boundary-case scenarios. They give me more confidence in the accuracy of the spread results and ensure that the final output is reliable.

E. Results



```
[(base) jaewon@Mac ds210_jwproject % cargo run
   Compiling ds210_jwproject v0.1.0 (/Users/jaewon/ds210_jwproject)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.81s
     Running `target/debug/ds210_jwproject`
Graph loaded with 4039 nodes
Starting gossip from node: 2998
Spread result from seed node 2998:
  Step 1 → 1 people have heard the gossip
  Step 2 → 4 people have heard the gossip
  Step 3 → 793 people have heard the gossip
  Step 4 → 1831 people have heard the gossip
  Step 5 → 3327 people have heard the gossip
  Step 6 → 3984 people have heard the gossip
Saved spread log to 'spread_log.csv'
Running spreader analysis on 50 random nodes...
Top 10 Gossip Spreaders (in 5 steps):
  #1  Node 1136  spread to 4039 people
  #2  Node 3111  spread to 3984 people
  #3  Node 3417  spread to 3984 people
  #4  Node 3493  spread to 3984 people
  #5  Node 3257  spread to 3984 people
  #6  Node 3695  spread to 3984 people
  #7  Node 2990  spread to 3984 people
  #8  Node 3367  spread to 3984 people
  #9  Node 2669  spread to 3984 people
  #10 Node 3420  spread to 3984 people
```

In our simulations, I found that gossip spread from some nodes can reach the entire network in only five or six steps, reflecting the small-world property often discussed in network science: even in large graphs, most nodes can be reached through a relatively small number of connections. More importantly, when I tested 50 randomly selected seed nodes, I found that some nodes consistently reach the maximum number of other nodes, while others only reach a fraction of them, supporting the idea that a node's position in the graph has a strong influence on its influence. Although this simulation used a simplified model, it successfully captured a meaningful property of social diffusion: neighboring nodes are less able to spread. These results not only validate the accuracy of the algorithm, but also confirm that even a simple approach can uncover structural insights into real-world networks. The CSV export (spread_log.csv) and the ranked spreader output support further analysis if needed.

F. Usage Instructions

The project is built and run using standard Cargo commands in Rust. No command-line arguments or user input are required. The program automatically loads facebook_combined.txt from the project root, constructs the graph, selects a random seed node, and runs a five-step gossip simulation. It then performs 50 runs to identify the most influential spreaders. Output is printed to the terminal and saved to spread_log.csv. The program completes in under 2 seconds on a MacBook and only depends on the rand crate.