Collaborators: None

# DS210 Final Project Report

## Project Overview

**Goal**: The data set provides mutual follower relationships for streaming Twitch users, which were analyzed to identify large cliques of streamers. These cliques are assumed to be groups of streamers that regularly interact with each other. The project analyzes these cliques to identify trends in how viewership is distributed between streamers in the clique, suggesting whether these streamer groups generate a diffusion or a concentration of viewership among streamers.

**Dataset**: https://snap.stanford.edu/data/twitch-social-networks.html, which contains multiple datasets for different languages (German, British English, Spanish, French, Portuguese, and Russian). Each dataset can be selected individually through terminal inputs.

| Dataset statistics | | | | | | |
|---|---|---|---|---|---|---|
| | DE | EN | ES | FR | PT | RU |
| Nodes | 9,498 | 7,126 | 4,648 | 6,549 | 1,912 | 4,385 |
| Edges | 153,138 | 35,324 | 59,382 | 112,666 | 31,299 | 37,304 |
| Density | 0.003 | 0.002 | 0.006 | 0.005 | 0.017 | 0.004 |
| Transitvity | 0.047 | 0.042 | 0.084 | 0.054 | 0.131 | 0.049 |

## Data Processing (File Reading Module)

**Loading Edges:** The edges file contains rows of two vertices that are mutually connected. This data is loaded into Rust via the csv crate's reader and added to a HashMap. For each pair, the code first treats the first vertex as the key and the second vertex as the value, before repeating the reverse process to save the mutual relation and create the undirected graph. Values are saved as a HashSet for efficient lookup, since the order of connections does not matter for the Bron-Kerbosch algorithm. If a key does not yet exist, the code creates the HashSet before adding the relevant value.

Collaborators: None

**Loading Vertices:** Data was loaded into Rust using the csv crate's reader and serde deserialization (details in node cleaning). All rows were loaded and deserialized into a custom struct called NodeStats and then placed in a vector of NodeStats. This vector was referenced to reconstruct the identified cliques, which were found using the node's u32 values from the prior loading of the edges file.

**Data Cleaning:** When deserializing each row from the target file into the NodeStats struct, the fields "id" and "days" were not kept. "Id" references the internal Twitch ID for the user, which was not used by the edges file, which used the field "new_id". "Days" was not defined by the dataset and was therefore removed to avoid ambiguity.

## Core Structure

**Modules:**

The "file_reading" module houses the functions for reading both files; the structure of these reads can be found in the Data Processing section.

The "data_analysis" module houses the NodeStats struct and all functions related to modifying or visualizing the data contained inside NodeStats fields.

The "bron-kerbosch" module houses the central clique identification algorithm, alongside small- and large-scale tests against a verified version of the algorithm code.

The "copied_alg" code contains this verified version of the algorithm code, taken from Rosetta Code, and was separated for clarity.

All functions were separated into modules to reduce clutter on the main.rs file and clearly define the purpose of each function at a glance.

**Key Functions & Types (in order of appearance in main function)**

csv_to_hashmap (function):

- Loads the edges CSV file and processes the data into an undirected graph.
- Input: the file path of the edges CSV file as a &str.
- Output: A HashMap<u32, HashSet<u32>> that functions as the undirected graph
- Core logic: for each two-vertex row, the vertices are inputted into the HashMap as both keys and values to save the mutual connection and form an undirected graph.

run_bron_kerbosch (function):

Collaborators: None

- The maximal clique identification algorithm identifies all cliques from the undirected graph and returns all cliques that pass the hardcoded minimum value (currently hardcoded at 10 to reduce computation time).
- Inputs:
    - r, p, x: 3 HashSets that track the clique currently being built, the unprocessed vertices remaining, and the vertices that have already been processed (r, p, x, respectively). r and x are initially empty, while p is initially all nodes in the graph.
    - graph: A HashMap of u32 keys and HashSets containing u32s that functions as the undirected map.
    - cliques: A vector of u32 vectors, initially empty. This is the output with all identified cliques.
    - min_value: A minimum value threshold. Cliques will only be saved and outputted if they are greater than or equal to this value.
- Output: The cliques vector of vectors, now filled with identified cliques.
- Core logic:
    - The algorithm first checks if all possible candidate nodes in p and x are empty, and if so, the clique is pushed to the output vector if it passes the minimum value threshold.
    - If there are candidates left, the algorithm chooses a central node based on the candidate with the most remaining neighbors (pivoting to reduce computation).
    - Using this central node, candidates are redefined as all unprocessed nodes that are not neighbors to the central node (all neighbors are considered by considering the central node).
    - The algorithm is recursively called on updated inputs:
        - The central node is added to r
        - p is now all the unprocessed neighbors of the central node
        - x is now all the processed neighbors of the central node
    - The central node is now moved from p to x, since it has been processed.

NodeStats (struct):

- Contains the relevant id (new_id: u32), viewership (views: u32), maturity rating (mature: bool), and partner status (partner: bool) for each vertex/node.

Collaborators: None

- Read from the target csv file via serde deserialization in the "load_target_file_replace_u32_cliques" function.
- Streamlines the analysis process by allowing each clique to be represented by a struct that already contains all relevant statistics.

deserialize_bool (function):

- A custom deserialization function to work around the target file using capital letters when defining the bool values.
- Input: a capitalized bool value from the target CSV file (True, False).
- Output: a Rust-readable bool value (true, false) or error.
- Core logic: matches True and False to true and false, or returns an error if neither value was inputted.
- *Code lifted from StackOverflow answer, cited in code and in report citations

load_target_file_replace_u32_cliques (function):

- Rebuilds the u32 cliques (identified using just the u32 ID for computational efficiency) as cliques of NodeStats for continuing analysis.
- Inputs: the path to the target CSV file as a &str and the vector of vectors containing the u32 cliques.
- Output: a vector of vectors, where each u32 ID from the inputted cliques vector of vectors has been replaced by the NodeStats struct with the matching ID.
- Core logic:
    - Load the target CSV file into a vector of NodeStats using serde deserialization, to reference later.
    - For each clique, find the matching NodeStats entry for each node by matching the ID values. Push these NodeStats structs to a clique vector.
    - Push each clique vector to the output vector to rebuild the original vector of vectors structure.

viewership_distribution (function):

- Identifies the distribution of total viewership amongst vertices (streamers) in each clique.
- Inputs: the vector of NodeStats vectors that represents the identified cliques.
- Outputs: a vector of vectors of tuples, each tuple containing the ID of the vertex and the percentage of the clique's total viewership the vertex represents.

Collaborators: None

- Core logic:
    - For each clique vector, accumulate the clique's total viewership through a fold closure on the views field of each NodeStats struct.
    - Using this sum of viewership, iterate back through the NodeStats and push the result of (vertex.views / sum_viewership) to a new vector of f32 values for the clique.
    - Push each clique's vector to the output vector to maintain the vector of vectors structure.

plot_viewership_distributions (function):

- Plots the viewership distributions of each clique onto its own bar chart using Plotters.
- Input: a vector of vectors containing (u32, f32) tuples, relating to the vertex ID and percentage of total clique viewership each vertex represents.
- Output: a viewership_distributions.png file containing Plotters bar charts for each clique.
- Core logic:
    - Identify the number of cliques and sub-divide the generated drawing space into enough spaces for all cliques
    - For each clique, format the chart context and mesh before drawing the series as red rectangles based on the number of vertices in each clique and their f32 values.
    - Return the drawn space as a PNG file.

**Main Workflow:**

User inputs set the language for analysis (and therefore the files to read), and the minimum size each clique needs to be saved for analysis.

↓

File_reading module's csv_to_hashmap function reads the edges file into a HashMap that functions as the undirected graph.

↓

The graph is passed into the bron_kerbosch module's run_bron_kerbosch alongside related HashSets and the user-set minimum value. The Bron-Kerbosch algorithm returns a vector containing vectors that contain vertex IDs for all identified cliques that are greater than or equal to the minimum value variable.

↓

*Terminal print of how many image files will be created, requires user input to continue*

Collaborators: None

↓

The vector of clique vectors is passed to the data_analysis module's load_target_file_replace_u32_cliques, which reads the target.csv file and reconstructs the vectors using each ID's related NodeStats struct.

↓

The vector of clique vectors using NodeStats is passed to the data_analysis module's viewership_distribution function, which reconstructs the vector of vectors format, replacing each NodeStats struct with a tuple of the vertex's ID and the percentage of total clique viewership the vertex is responsible for.

↓

This vector of vectors of tuples is passed to the data_analysis module's plot_viewership_distribution, which uses 16 vector chunks to generate image files with bar charts that visualize each clique vector's viewership distribution.

## Tests

**Cargo test output:**

```
running 5 tests
test data_analysis::tests::test_distributions ... ok
test bron_kerbosch::tests::test_alg_smallscale ... ok
test file_reading::tests::test_edge_reading ... ok
test file_reading::tests::test_target_reading ... ok
test bron_kerbosch::tests::test_alg_largescale ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 1.56s
```

Test *test_distributions* ensures that the replacement of NodeStats vectors with tuples representing the vertex's percentage of total clique viewership is accurate and structured properly. Without this, the plotting would not be accurate, and all inferences from the plotted data would be useless.

Test *test_alg_smallscale* tests my implementation of the Bron-Kerbosch algorithm against a verified version copied from Rosetta Code. This small-scale version also prints out the cliques found to help a user understand the process. This code confirms that the algorithm itself works.

Test *test_alg_largescale* also checks against the Rosetta Code algorithm, but uses one of the datasets to do so on a larger scale. This test confirms that the file reading/graph creation and the algorithm work properly together.

Collaborators: None

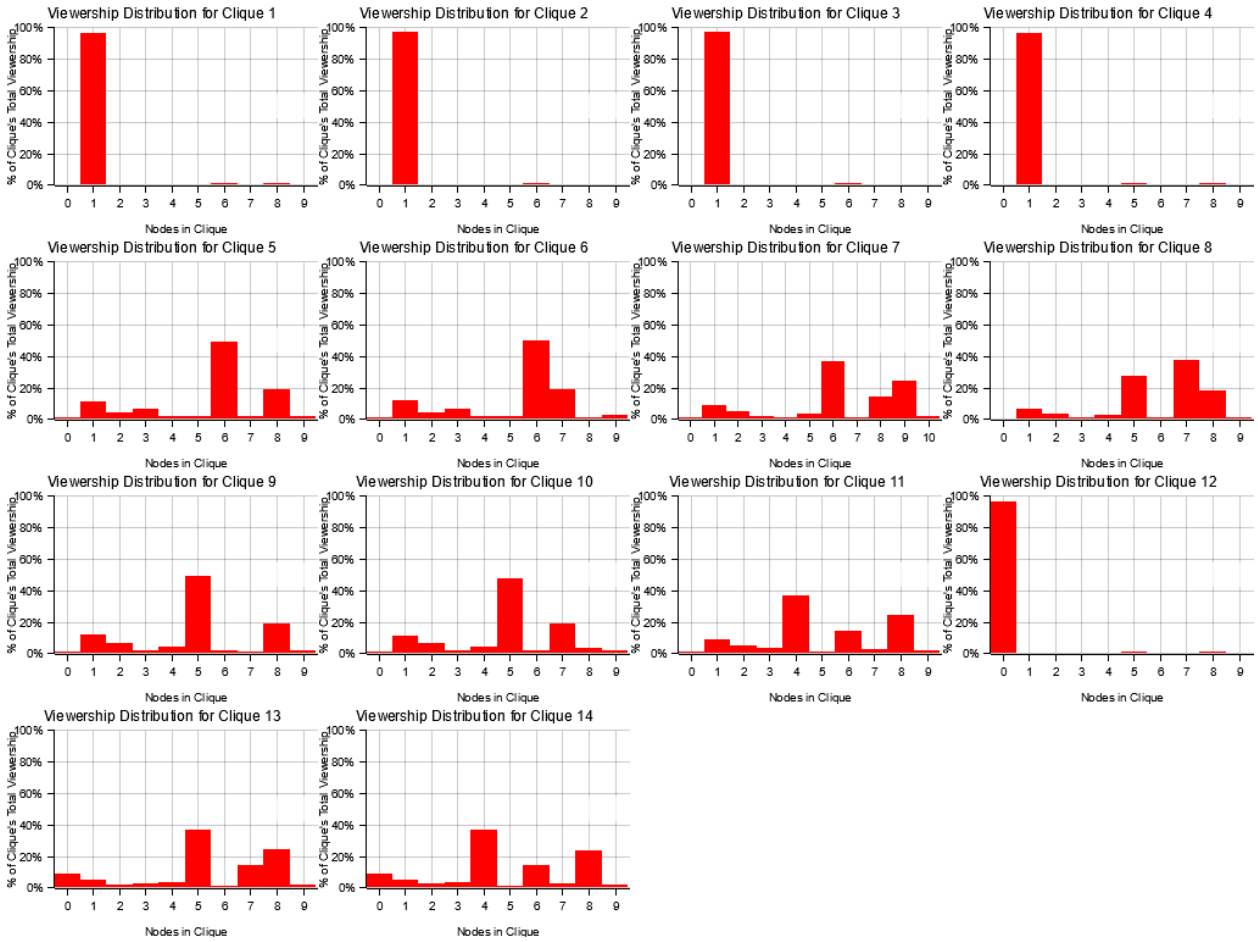
Test *test_edge_reading* checks that the edge file is being properly read by using a temporarily created small CSV file. Ensuring that the edge file and graph creation are working properly is paramount to the code, as it is the initial block upon which everything else is built.

Test *test_target_reading* checks the code's accuracy in replacing u32 cliques with NodeStats cliques through deserialization. The test provides a small temporary CSV file that mimics the structure of the target.csv files in the dataset. This is also vital because inaccurate replacement of a u32 clique with a NodeStats clique would mean that the cliques fed into the plotting functions would be meaningless.

## Results

The code could produce many results, I will explore one example.

Example Output for British English Dataset and a minimum value of 10:




Interpretation:

Based on the charts produced, for the British English dataset, these streamer groups do not result in a significant diffusion of viewership between streamers. In multiple graphs, some streamers' percentage of the total clique's viewership is so small that the bar is unable to be graphed. This suggests closely connected streamer groups are not correlated with equality in viewership among

Collaborators: None

the clique, but the statistical significance and direction of this correlation would need to be analyzed before concluding whether forming streamer groups is an effective way to grow an audience. However, this preliminary data is not positive.

## Usage Descriptions

Code can be built by using cargo run within the project_code subfolder. The terminal will display requests for which dataset you wish to analyze, which are separated by language. This can be input into the terminal through an integer. The terminal will also prompt for a minimum value that the user wants each clique to be in order to be saved for analysis, also input as an integer into the terminal.
Once the code identifies cliques based on these parameters, it will display a message regarding how many cliques were identified and how many image files will be created. This requires a terminal input of "y" to continue, ensuring a user doesn't end up with hundreds of image files accidentally.
Image files will be output in the project_code subfolder.

**Expected Runtime (and recommendation):** Code runtime depends on the input parameters. For a consistently fast runtime and only 1 image file, please choose "1" for the language parameter (British English) and "10" for the minimum clique size. This results in 14 identified cliques and a single image file being generated. This code should run within 5 seconds.

## AI Disclosure and Citations

Generative AI was used only for debugging purposes, directly through VSCode's Co-Pilot, with code suggestions/auto-completion turned off. I believe this falls under debugging and is allowed to be skipped.
My Bron-Kerbosch Algorithm builds on the general steps outlined in this YouTube explanation:
https://www.youtube.com/watch?v=j_uQChgo72I
Custom Serde deserialization was taken from this StackOverflow question:
https://stackoverflow.com/questions/70114905/how-to-deserialize-a-string-field-to-bool
The Plotters Guide was referenced to construct the plotting code:
https://plotters-rs.github.io/book/basic/basic_data_plotting.html
Lecture notes and example code were also referenced throughout the project.