Zihao Guo

DS210

12/12/2023

Project Report

Data Link: https://snap.stanford.edu/data/p2p-Gnutella08.html

This data is connection data with 2 columns. The left column is the subject ID and the right column is the object ID. It outputs that the left ID knows the right ID in a **DIRECTED** way.

Output data:

```
Total sixth-degree paths count: 6420200
Mean sixth-degree connections per vertex: 1018.9176321218854
Proportion of vertex pairs with a sixth-degree connection: 0.3234659149593287
Variance of sixth-degree connections per vertex: 784678.0743137739
Standard deviation is:885.8205655288061
The average distance between pairs of vertices in the graph is: 4.642991785658397
Most Similar Pair: (6100, 6103) with Jaccard Coefficient: 1
Most Dissimilar Pair: (1317, 667) with Jaccard Coefficient: 0.006535947712418301
```

Data explanation:

Total sixth degree: In all the combinations that are listed by node, there are only 6420200 paths which is exactly 6 steps from one ID to another.

Mean sixth degree: The total number of six steps reached divided by the number of nodes, which means that each node will have about 1019 paths on average.

Proportion: Total number of six steps divided by the total number of paths. 32.35 percent of the total path is a six-step path.

Variance: The difference between each node's sixth-degree path compared to the mean of the sixth-degree path.

Standard deviation: The measure of the average distance from the mean, in the same units as the data itself.

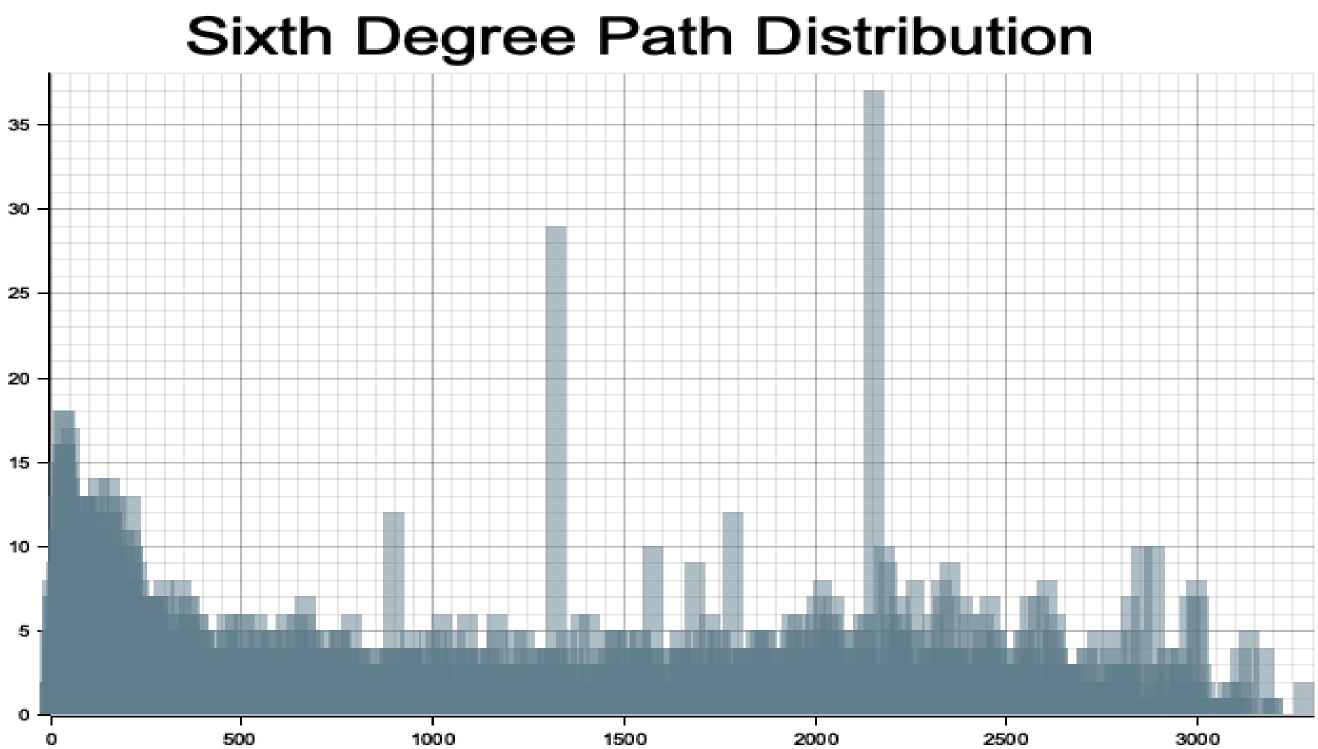
Avg steps: The average steps that one node needs to reach another node.

Similar and dissimilar pair: The similar and dissimilar pair will use the Jaccard coefficient from the domain of 0 to 1 to find the most related and unrelated pairs. 0 would be considered the most not related and 1 would be the most relative.

Data analysis:

From the mean and proportion of vertex pairs with a sixth-degree connection, we can conclude the connection map of the data is a dense network, which means that if a person wants to reach out to another person, it may cost a lot of steps to connect. The high variance and standard deviation shows that there are different type of people in the data. Some people are very socialized and they can connect with people directly or within a few steps (at least less than six steps). Also, some people may not be good at social networking, and it’s hard for them to connect with people, so they may need to connect with more than six steps. When we connect both groups of people, we can see that there are more socialized people than unsocialized people, from the average distance between pairs of vertices in the graph, we can see that the average step cost is 4.64, which is less than six steps. We also made a graph that analyzes the number of sixth steps that each person needs and the frequency of people.

(The image is on the next page):



The peaks that appear on the graph show that there may be some underlying structure or community clusters where nodes within the same cluster have a similar number of sixth-degree connections. The spread of width distribution also indicates that people may be socialized or unsocialized, which means that some people need more sixth-degree paths and some don't. The skewness shows that it's a slightly right skew. We can conclude that there are more nodes with fewer sixth-degree connection, and we can also confirm the average number of steps that a node need to another node. There are some outliers on both the left and right sides. We can conclude that there are some people who know a lot of connections that are in the data. Some people need a lot of sixth-degree paths and only know a few people in the dataset.

Function explaining:

Main.rs: basically just mod and crate all the functions that will conduct a result call them, and print out the result

Graph.rs:

1. Read graph: input the data split them into the correct format, and set two variables: from and to (column 1 and column 2).
2. Build graph: use hashmap to store nodes and connect them with hashset to form a graph.

Seperation.rs:

1. Calculate sixth-degree stats: use some of the functions on the bottom to calculate the stats of sixth degree that will be called in the main function
2. Bfs for sixth degree: using bfs algorithm to calculate all of the paths that will have exactly six steps, and we can use this count to find the stat that will be calculated in the previous function
3. Average distance: The function will use bfs for avg distance function to find the shortest path of distance from one node to another node. After that, we will sum it up and find the average distance for each of them.
4. Bfs directed: basic bfs algorithm that will calculate the path from one node to another node with the shortest distance.
5. Plot function: using the plot to find x = the sixth-degree paths that a node needs and y= the frequency of node that needs these amount of path

Similaryity.rs:

1. Find extreme pair: find the pair that will have the most similar and dissimilar using the calculation of Jaccard similarity
2. Jaccard similarity: find the intersection of two nodes and divide the union of two nodes. The domine will be between 0 to 1, which 0 is the dissimilar and 1 is the most similar.

AI using: use ChatGPT to find out how to draw a graph in rust

ChatGPT

Creating a histogram in Rust typically involves several steps. Below is an overall structure on how you would go about creating a histogram:

1. **Data Collection:**
   - First, you need to have a dataset from which you'll calculate the frequency of each distinct value or range of values. This could be a simple vector of integers or floats, or a more complex data structure if you're dealing with multidimensional data.
2. **Frequency Calculation:**
   - Count the occurrences of each unique value or the number of values that fall within each bin range. This can be done by iterating over your data and using a `HashMap` or `Vec` to keep track of counts.
3. **Setting up the Graphing Environment:**
   - Choose a Rust plotting library, such as `plotters`, which is versatile and can create a variety of plots including histograms.
   - Add the necessary crate to your `Cargo.toml` file and import it into your Rust file.
4. **Graph Configuration:**
   - Determine the dimensions of your histogram and the labeling for the x-axis (bins) and y-axis (frequency).
   - Set up the drawing backend, which can be an image file, a GUI window, or an interactive environment, depending on the library capabilities and your needs.
5. **Drawing the Histogram:**
   - Using the plotting library functions, draw the bars of the histogram where each bar represents a bin and its height corresponds to the frequency of the data in that bin.
   - Configure the style, width, colors, and other visual aspects of the bars.
6. **Adding Labels and Titles:**
   - Add titles, axis labels, and any necessary annotations to make the histogram informative.
7. **Outputting the Histogram:**
   - Render the histogram to the chosen output, which might be saving to a file or displaying on the screen.
8. **Error Handling:**
   - Ensure that your code gracefully handles any errors that may occur during data processing or drawing, such as through the use of `Result<T, E>` types and `match` or `if let` constructions.

rust

Copy code

```rust
use plotters::prelude::*;

fn create_histogram(data: &[usize], output_file: &str) -> Result<(), Box<d
    let root_area = BitMapBackend::new(output_file, (640, 480)).into_drawi
    root_area.fill(&WHITE)?;

    let max_value = *data.iter().max().unwrap();
    let min_value = *data.iter().min().unwrap();

    let mut frequencies = vec![0; max_value + 1];
    for &value in data {
        frequencies[value] += 1;
    }

    let histogram_data: Vec<(usize, usize)> = frequencies.iter().enumerate

    let mut chart = ChartBuilder::on(&root_area)
        .caption("Histogram", ("sans-serif", 40))
        .x_label_area_size(40)
        .y_label_area_size(40)
        .build_cartesian_2d(min_value..max_value, 0..*frequencies.iter().m

    chart.configure_mesh().draw()?;

    chart.draw_series(
        histogram_data.iter().map(|&(x, y)| {
            let x0 = x as i32;
            let y0 = 0;
            let x1 = (x + 1) as i32;
            let y1 = y as i32;
            Rectangle::new([(x0, y0), (x1, y1)], RED.filled())
        }),
    )?;

    root_area.present()?;

    Ok(())
}
```