Itai Merlin

DS210

30 April 2024

My dataset: https://snap.stanford.edu/data/facebook_large.zip

Prompt: “How often are friends of my friends my friends?

Explanation of my dataset: The graph comprises of 22,470 nodes (Facebook pages) and 171,002 edges (mutual likes). This dataset represents a large social network of Facebook pages, where vertices are Facebook pages and edges represent mutual likes between them. This dataset is interesting because it reflects real-world social interactions and is a rich source for exploring how interests (in the form of page likes) create connections between people online.

Summary of Project:

For this project, I wrote code that analyzed a Facebook dataset where people's interests on social media were turned into a graph by converting people into vertices and connecting them if he/she had some interest in common based on the info in the dataset.

functions.rs:

This is responsible for reading JSON data from a file using the load_data function, which efficiently handles all of the inputs.

- **“Parse_person”:** an important function that takes a person's name and interests from a JSON object and then returns the data.
- **“Serde_json”:** is used to parse JSON into a vector.
- **“Load_data”:** This function reads JSON data from a specified file path.
- **Error Handling:** This module also returns a “Result type” that either contains the successfully parsed data or an error.

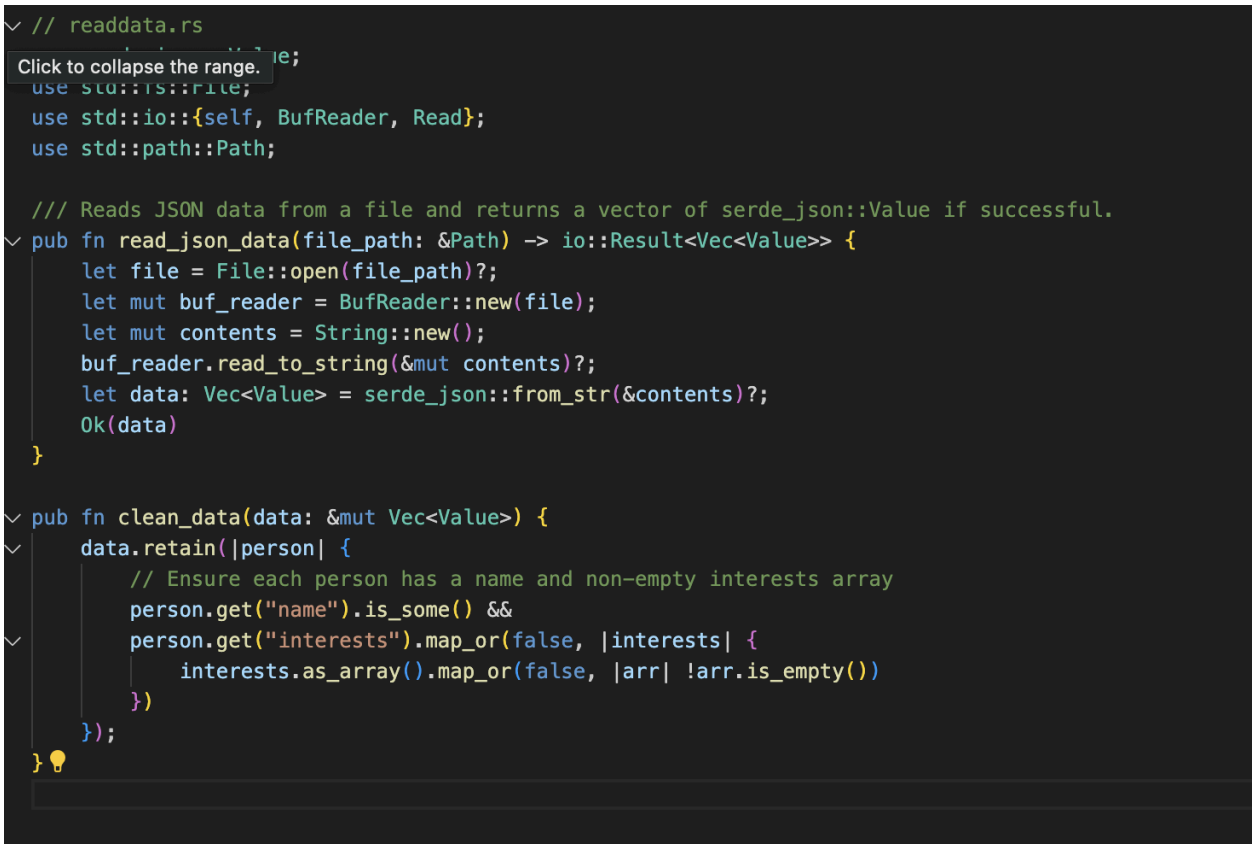
Essentially, function.rs parses the data and allows it to become very manipulatable. Because of function.rs, the code is eventually able to create the graph with nodes.

```rust
// functions.rs
use serde_json::Value;
use std::fs::File;
use std::io::{self, BufReader, Read};
use std::path::Path;
use std::error::Error;
use std::collections::HashSet;

/// Reads JSON data from a file and returns a vector of serde_json::Value if successful.
pub fn load_data(file_path: &Path) -> Result<Vec<Value>, Box<dyn Error>> {
    let file: File = File::open(file_path)?;
    let mut buf_reader: BufReader<File> = BufReader::new(inner: file);
    let mut contents: String = String::new();
    buf_reader.read_to_string(buf: &mut contents)?;
    serde_json::from_str(&contents).map_err(Into::into)
}

/// Parses a JSON object into a tuple of a person's name and a vector of their interests.
/// Additionally filters out any unnecessary data if needed.
pub fn parse_person(person: &Value) -> Result<(String, HashSet<String>), &'static str> {
    let name: String = person["name"].as_str().ok_or("Missing name in data")?.to_string();
    let interests: HashSet<String> = person["interests"].as_array()
        .ok_or("Missing interests in data")?
        .iter()
        .filter_map(|v| v.as_str().map(String::from))
        .collect::<HashSet<_>>();
    Ok((name, interests))
}

/// Placeholder for data cleaning function, which could be implemented as needed.
pub fn clean_data(data: &mut Vec<Value>) {
    // Implement specific data cleaning steps here.
}
```

readdata.rs:

Similar in functionality to functions.rs, readdata.rs includes a function to read JSON data from a file.

Data Cleaning Placeholder: This code also has a data cleaning function that ensures that each person has a name and non-empty interests array. It allows for only "clean" data to be analyzed and eventually implemented into the graph.



```rust
// readdata.rs
Click to collapse the range.
use std::io::{self, BufReader, Read};
use std::path::Path;

/// Reads JSON data from a file and returns a vector of serde_json::Value if successful.
pub fn read_json_data(file_path: &Path) -> io::Result<Vec<Value>> {
    let file = File::open(file_path)?;
    let mut buf_reader = BufReader::new(file);
    let mut contents = String::new();
    buf_reader.read_to_string(&mut contents)?;
    let data: Vec<Value> = serde_json::from_str(&contents)?;
    Ok(data)
}

pub fn clean_data(data: &mut Vec<Value>) {
    data.retain(|person| {
        // Ensure each person has a name and non-empty interests array
        person.get("name").is_some() &&
        person.get("interests").map_or(false, |interests| {
            interests.as_array().map_or(false, |arr| !arr.is_empty())
        })
    });
}
```

main.rs:

This is the main part of my code that initializes the data and creates the graph of nodes and edges.

- It uses functions.rs to load and parse the dataset.
- It initializes an undirected graph (UnGraph) where each person is a node. Edges between nodes are created based on shared interests between individuals, determined by comparing their interest sets.

- The graph is constructed by iterating over each person, adding them as nodes, and then iterating over pairs of people to add edges for those who share interests.
- It prints the graph by using the DOT format and provides a simple count of the nodes and edges.

How it works:

1. **Data Loading:** It begins by defining the path to the JSON data file and then uses the “load_data” function from functions.rs to read and parse the data into a vector.
2. **Graph Initialization:** It then initializes a graph using petgraph's UnGraph class. Each person in the dataset is added as a node in the graph.
3. **Node Creation:** For each person, it takes their name and interests using the “parse_person” function. Each person's name is added to the graph as a node.
4. **Edge Creation Based on Shared Interests:** It then implements a nested loop to compare the interests of each pair of individuals in the dataset. If two people share at least one interest, an edge is created between their corresponding nodes in the graph. The edge is created using a HashSet for each person's interests to efficiently check for intersections.
5. **Graph Output and Analysis:** After the graph is built, it prints the graph in DOT format using petgraph's Dot module. It also prints a simple count of nodes and edges, providing basic insights into the graph's structure.

```rust
fn main() {
    let path: &Path = Path::new("/Users/itaimerlin/Downloads/facebook_large/musae_facebook_features.json");
    let people = functions::load_data(&path).expect("Could not load data");

    let mut graph = UnGraph::<String, &str>::new_undirected();

    // Build nodes and edges based on shared interests
    let mut name_to_node: HashMap<{unknown}, ({unknown}, …)> = std::collections::HashMap::new();
    for person in &people {
        let (name, interests) = functions::parse_person(person).expect("Error parsing person data");
        let node_id = graph.add_node(name.clone());
        name_to_node.insert(k: name, v: (node_id, interests));
    }

    // Create edges for shared interests
    for (name_a: &{unknown}, (node_id_a: &{unknown}, interests_a: &{unknown})) in &name_to_node {
        for (name_b: &{unknown}, (node_id_b: &{unknown}, interests_b: &{unknown})) in &name_to_node {
            if name_a != name_b && !interests_a.is_disjoint(interests_b) {
                graph.add_edge(*node_id_a, *node_id_b, "shared interest");
            }
        }
    }

    // Output the graph in DOT format
    println!("{}", Dot::with_config(&graph, &[Config::EdgeNoLabel]));
    println!("Graph has {} nodes and {} edges", graph.node_count(), graph.edge_count());
} fn main
```