Project Report

Github repo: https://github.com/Kaiyiliu233/DS210_project

1. Dataset:

The Enron Email Dataset: https://www.kaggle.com/datasets/amank56/enron-clean-dataset
The Enron Email Dataset is a comprehensive collection of approximately 500,000 emails from around 150 Enron employees, primarily senior management, spanning from 2000 to 2002. This dataset allows me to explore the organizational communication patterns, identify key influencers, and understand the underlying structure of a large corporation using graph Theory.

This is a cleaned version of the Enron Email Dataset, it is too large to put on to the github, so I left a link here. Also, I combined multiple csv file into one csv file. It will also be available in the zip file I submitted to the Gradescope.

Sample row of a dataset:

```
,date,sender,recipient1,subject,text
0,2001-05-14 16:39:00
07:00,phillip.allen@enron.com,tim.belden@enron.com,,"['', 'Here is our
forecast', '', ' ']"
```

2. Project Goal:
   The primary objective of this project is to analyze and interpret email communication patterns within an organization by using graph-based algorithms. This project is able to:
   (1) Extract Relevant Information: Efficiently parse email data from CSV files to extract essential fields such as sender, recipients, date, subject, and email body.
   (2) Build a Directed Communication Network: Represent the parsed email data as a directed, unweighted graph where nodes correspond to unique email addresses and edges denote the direction of communication (from sender to recipient).
   (3) Degree Calculations: Compute in-degrees and out-degrees for each node to identify prolific senders and frequent recipients, providing insights into individual communication behaviors.
   (4) Community Detection: Apply the Label Propagation Algorithm (LPA) to identify distinct communities or clusters within the network.
   (5) Top Senders and Recipients: Pinpoint individuals with the highest out-degrees and in-

degrees to recognize key communicators, potential information hubs, and influential personnel within the organization.

3. Algorithm explained
   The algorithm used in this project is Label Propagation Algorithm (LPA). LPA is employed to identify communities within an organization's email communication network. By detecting these communities, the project aims to uncover underlying organizational structures.

Initially, each node (email address) is assigned to a unique label. This ensures that initially, every node is its own community. Then, the nodes are processes in a randomly shuffled order to prevent any bias in label updated. For each node, examine the labels of its immediate neighbors and adopt the label that appears most frequently. In cases of ties, a label is chosen at random from the most frequent labels.
Then, repeat the label update process until no labels change during an entire iteration or until a predefined maximum number of iterations is reached. The algorithm concludes when labels stabilize, indicating that communities have been detected, or when it reaches the iteration limit to prevent infinite loops.

4. Result, and Result interpretation

```
Successfully parsed 405968 emails.
--- Out-Degree Statistics ---
Total Nodes: 27179
Total Out-Degree: 64113
Average Out-Degree: 2.36
Maximum Out-Degree: 718
Minimum Out-Degree: 0

--- In-Degree Statistics ---
Total Nodes: 27179
Total In-Degree: 64113
Average In-Degree: 2.36
Maximum In-Degree: 1274
Minimum In-Degree: 0

--- Top 10 Senders (Prolific Communicators) ---
1. vince.kaminski@enron.com - Sent 718 emails
2. tana.jones@enron.com - Sent 643 emails
3. sara.shackleton@enron.com - Sent 630 emails
4. jeff.dasovich@enron.com - Sent 551 emails
5. debra.perlingiere@enron.com - Sent 501 emails
6. kay.mann@enron.com - Sent 461 emails
7. gerald.nemec@enron.com - Sent 451 emails
```

```
8. chris.germany@enron.com - Sent 430 emails
9. mark.taylor@enron.com - Sent 409 emails
10. richard.sanders@enron.com - Sent 402 emails

--- Top 10 Recipients (Information Hubs) ---
1. klay@enron.com - Received 1274 emails
2. kenneth.lay@enron.com - Received 609 emails
3. jeff.skilling@enron.com - Received 567 emails
4. sara.shackleton@enron.com - Received 558 emails
5. tana.jones@enron.com - Received 527 emails
6. jeff.dasovich@enron.com - Received 510 emails
7. louise.kitchen@enron.com - Received 454 emails
8. gerald.nemec@enron.com - Received 415 emails
9. mark.taylor@enron.com - Received 394 emails
10. sally.beck@enron.com - Received 381 emails

Total Detected Communities: 9535
Total Nodes: 27179
Average Community Size: 2.85
Largest Community Size: 10652
Smallest Community Size: 1

--- Largest Community ---
Community Label: louise.kitchen@enron.com
Number of Members: 10652
Top 10 Members: ["tonyborelli@bandm.com", "greg_priest@smartforce.com",
"jas@beggslane.com", "dick.jenkins@enron.com",
"sanjeev.karkhanis@ubs.com", "charles.weldon@enron.com",
"tina.ward@enron.com", "jimmystorey@fbcc.com", "ava.garcia@enron.com",
"michael@eeg.com"]

--- Smallest Community ---
Community Label: timothy.norton@enron.com
Number of Members: 1
Member: ["timothy.norton@enron.com"]
```

In the coding block above, I have presented the result of a single run. Please note that result may differ for community detection, since LPA has a certain degree of randomness when tranversing the node. However, average community size remains constant.

The output above shows that 405,968 emails is parsed to construct a directed, unweighted communication network comprising 27,179 unique nodes (email addresses). The subsequent evaluation focused on degree statistics to identify key communicators and information hubs, followed by community detection using the Label Propagation Algorithm (LPA) to uncover underlying organizational structures. This section presents the results from a single execution of the analysis and provides a comprehensive interpretation of the findings.

Out-Degree Analysis
Total Nodes: 27,179
Represents the unique email addresses involved in the communication network.
Total Out-Degree: 64,113
Indicates the total number of outgoing emails sent within the network.
Average Out-Degree: 2.36
On average, each node sends approximately 2.36 emails.
Maximum Out-Degree: 718
The highest number of emails sent by a single node, suggesting a highly active communicator or a central department.
Minimum Out-Degree: 0
Nodes with no outgoing emails, potentially indicating roles with limited communication responsibilities or data anomalies.

Top 10 Senders (Prolific Communicators):
Individuals with high out-degrees are identified as prolific senders within the organization. These individuals are pivotal in initiating communications, coordinating activities, and disseminating information. These individuals play a crucial role in spreading information across departments or teams, ensuring organizational coherence.
(1) vince.kaminski@enron.com - Sent 718 emails
(2) tana.jones@enron.com - Sent 643 emails
(3) sara.shackleton@enron.com - Sent 630 emails
(4) jeff.dasovich@enron.com - Sent 551 emails
(5) debra.perlingiere@enron.com - Sent 501 emails
(6) kay.mann@enron.com - Sent 461 emails
(7) gerald.nemec@enron.com - Sent 451 emails
(8) chris.germany@enron.com - Sent 430 emails
(9) mark.taylor@enron.com - Sent 409 emails
(10) richard.sanders@enron.com - Sent 402 emails

In-Degree Analysis
Total Nodes: 27,179
Confirms that all nodes are accounted for within the analysis.
Total In-Degree: 64,113
Reflects the total number of incoming emails received within the network.
Average In-Degree: 2.36
On average, each node receives approximately 2.36 emails.

Maximum In-Degree: 1,274
The highest number of emails received by a single node, indicating a central information hub or a key executive.
Minimum In-Degree: 0
Nodes with no incoming emails, potentially indicating roles with limited reception responsibilities or data inconsistencies.

Top 10 Recipients (Information Hubs):
Individuals with high in-degrees serve as information hubs, centralizing communications and information intake within the organization. Some high in-degree email addresses may represent shared mailboxes (e.g., support teams), aggregating communications for efficiency.
(1) klay@enron.com - Received 1,274 emails
(2) kenneth.lay@enron.com - Received 609 emails
(3) jeff.skilling@enron.com - Received 567 emails
(4) sara.shackleton@enron.com - Received 558 emails
(5) tana.jones@enron.com - Received 527 emails
(6) jeff.dasovich@enron.com - Received 510 emails
(7) louise.kitchen@enron.com - Received 454 emails
(8) gerald.nemec@enron.com - Received 415 emails
(9) mark.taylor@enron.com - Received 394 emails
(10) sally.beck@enron.com - Received 381 emails

Community Detection (The output of this section will change each run due to the nature of the randomness of the algorithm)

- **Total Detected Communities:** 9,535
  A high number of communities suggests a fragmented or highly decentralized communication structure within the organization.
- **Total Nodes:** 27,179
  Confirms that all nodes are encompassed within the detected communities.
- **Average Community Size:** 2.85
  Indicates that most communities are small, comprising approximately 3 members on average.
- **Largest Community Size:** 10,652
  A dominant community likely representing a central hub or a highly interconnected group within the organization.
- **Smallest Community Size:** 1
  Numerous singleton communities indicate isolated nodes with no communication links.

Largest Community (This solution changed in each run)
The largest community comprises 10,652 members, indicating a highly interconnected group within the organization. The high membership count underscores its significance as a major communication hub, facilitating widespread information dissemination and coordination.

- **Community Label:** louise.kitchen@enron.com
- **Number of Members:** 10,652
- **Top 10 Members:** `["tonyborelli@bandm.com", "greg_priest@smartforce.com", "jas@beggslane.com", "dick.jenkins@enron.com", "sanjeev.karkhanis@ubs.com", "charles.weldon@enron.com", "tina.ward@enron.com", "jimmystorey@fbcc.com", "ava.garcia@enron.com", "michael@eeg.com"]`

Smallest Community:
The smallest community consists of a single member, indicating an isolated node within the communication network. This could signify several scenarios:

- **Community Label:** timothy.norton@enron.com
- **Number of Members:** 1
- **Member:** `["timothy.norton@enron.com"]`

Code Explanation
`Email.rs`
The objective of this segment is to parse and preprocess email communication data stored in a CSV (Comma-Separated Values) file.

```rust
use serde::Deserialize;
use std::fs::File;
use std::error::Error;
use csv::ReaderBuilder;

pub struct EmailRecord {
#[serde(rename = "")]
        pub index: usize,
        pub date: String,
        pub sender: String,
        pub recipient1: String,
        pub subject: String,
        pub text: String,
}

#[derive(Debug)]
pub struct ParsedEmail {
        pub from: String,
        pub to: Vec<String>,
}

pub fn parse_recipients(recipient: &str) -> Vec<String> {
    recipient
        .split(',')
        .map(|s| s.trim().to_lowercase())
        .filter(|s| !s.is_empty())
        .collect()
}

pub fn read_csv(file_path: &str) -> Result<Vec<ParsedEmail>, Box<dyn
Error>> {
    let file = File::open(file_path)?;
    let mut rdr = ReaderBuilder::new()
        .has_headers(true)
        .from_reader(file);

    let mut parsed_emails = Vec::new();
    let mut failed_parses = 0;

    for result in rdr.deserialize() {
        let record: EmailRecord = match result {
            Ok(rec) => rec,
            Err(e) => {
                eprintln!("Failed to deserialize a record: {}", e);
                failed_parses += 1;
                continue;
            }
        };

        let recipients = parse_recipients(&record.recipient1);

        if record.sender.is_empty() || recipients.is_empty() {
            eprintln!(
                "Incomplete record found at index {}: sender='{}',
recipient1='{}'",
                record.index, record.sender, record.recipient1
            );
            failed_parses += 1;
            continue;
        }

        let parsed_email = ParsedEmail {
            from: record.sender.clone(),
            to: recipients,
        };


        parsed_emails.push(parsed_email);
    }

    println!(
        "Successfully parsed {} emails.",
        parsed_emails.len()
    );
    if failed_parses > 0 {
        println!("Failed to parse {} records.", failed_parses);
    }

    Ok(parsed_emails)
}
```

```
use serde::Deserialize;
use std::fs::File;
use std::error::Error;
use csv::ReaderBuilder;
```

- `serde::Deserialize`: Utilized for deserializing CSV records into Rust structs. `Serde` is a powerful serialization framework that facilitates converting data between different formats.
- `std::fs::File`: Provides functionalities to handle file operations, such as opening and reading files.
- `std::error::Error`: A trait that represents general errors. It allows the `read_csv` function to return any type of error that implements this trait, enhancing flexibility in error handling.
- `csv::ReaderBuilder`: Part of the `csv` crate, it enables the creation and configuration of CSV readers with customizable settings, such as header handling and delimiter specification.

```
#[derive(Debug, Deserialize)]
pub struct EmailRecord {
    #[serde(rename = "")]
    pub index: usize,
    pub date: String,
    pub sender: String,
    pub recipient1: String,
    pub subject: String,
    pub text: String,
}
```

Represents each record (row) in the CSV file. This struct maps directly to the CSV columns, facilitating the deserialization process.

- `index: usize` : Corresponds to the first unnamed column in the CSV, acting as an index or unique identifier for each email record. The `#[serde(rename = "")]` attribute indicates that this field should capture the first unnamed column.
- `date: String` : Stores the date when the email was sent.
- `sender: String` : Contains the email address of the sender.
- `recipient1: String` : Holds the primary recipient(s) of the email. This field may contain multiple recipients separated by commas.
- `subject: String` : Captures the subject line of the email.
- `text: String` : Includes the body text of the email.

```
#[derive(Debug)]
pub struct ParsedEmail {
    pub from: String,
    pub to: Vec<String>,
}
```

Simplifies the email record by focusing only on the sender (`from`) and the list of recipients (`to`). This abstraction is beneficial for constructing communication networks where only the direction of communication (from sender to recipients) is essential.

- `from: String` : Represents the sender's email address.
- `to: Vec<String>` : A vector containing the email addresses of all recipients. Using a vector allows handling multiple recipients per email.

```
pub fn parse_recipients(recipient: &str) -> Vec<String> {
    recipient
        .split(',')
        .map(|s| s.trim().to_lowercase())
        .filter(|s| !s.is_empty())
        .collect()
}
```

Processes the `recipient1` field from each email record, extracting individual recipient email addresses.

- `split(',')`: Splits the recipient string by commas, accommodating multiple recipients in a single field.
- `map(|s| s.trim().to_lowercase())`: Trims whitespace from each recipient address and converts it to lowercase to maintain consistency and prevent duplication due to case differences.
- `filter(|s| !s.is_empty())`: Excludes any empty strings resulting from potential trailing commas or malformed entries.
- `collect()`: Gathers the processed recipient addresses into a `Vec<String>`.

```rust
pub fn read_csv(file_path: &str) -> Result<Vec<ParsedEmail>, Box<dyn
Error>> {
    let file = File::open(file_path)?;
    let mut rdr = ReaderBuilder::new()
        .has_headers(true)
        .from_reader(file);

    let mut parsed_emails = Vec::new();
    let mut failed_parses = 0;

    for result in rdr.deserialize() {
        let record: EmailRecord = match result {
            Ok(rec) => rec,
            Err(e) => {
                eprintln!("Failed to deserialize a record: {}", e);
                failed_parses += 1;
                continue;
            }
        };

        // Parse recipients
        let recipients = parse_recipients(&record.recipient1);

        // Check for missing sender or recipients
        if record.sender.is_empty() || recipients.is_empty() {
            eprintln!(
                "Incomplete record found at index {}: sender='{}',
recipient1='{}'",
                record.index, record.sender, record.recipient1
            );
            failed_parses += 1;
            continue;
        }

        // Create ParsedEmail instance
        let parsed_email = ParsedEmail {
            from: record.sender.clone(),
            to: recipients,
        };
        // For debugging: print first few parsed emails
        if parsed_emails.len() < 5 {
            println!("{:?}", parsed_email);
        }

        parsed_emails.push(parsed_email);
    }

    println!(
        "Successfully parsed {} emails.",
        parsed_emails.len()
    );
    if failed_parses > 0 {
        println!("Failed to parse {} records.", failed_parses);
    }

    Ok(parsed_emails)
}
```

Reads the CSV file containing email records, deserializes each row into the `EmailRecord` struct, processes the recipient information, and constructs a vector of `ParsedEmail` instances for further analysis.

- File Opening `File::open(file_path)?`: Attempts to open the CSV file at the specified `file_path`. The `?` operator propagates any errors encountered during this process.
- CSV Reader Initialization `ReaderBuilder::new().has_headers(true).from_reader(file)`: Configures the CSV reader to expect headers in the first row and initializes it with the opened file.
- Initialization of Containers:
  - `parsed_emails`: A vector to store successfully parsed `ParsedEmail` instances.

  - `failed_parses`: A counter to track the number of records that failed to parse due to errors or missing information.
- Iterating Over CSV Records `for result in rdr.deserialize()`: Iterates over each row in the CSV, attempting to deserialize it into an `EmailRecord`.
- Deserialization and Error Handling:
  - Successful Deserialization (`Ok(rec)`): Proceeds to process the record.
  - Failed Deserialization (`Err(e)`): Logs an error message specifying the nature of the failure and increments the `failed_parses` counter. The `continue` statement skips to the next record without halting the entire parsing process.
- Recipient Parsing `parse_recipients(&record.recipient1)`: Processes the `recipient1` field to extract individual recipient email addresses.
- Validation of Essential Fields `if record.sender.is_empty() || recipients.is_empty()`: Checks for the presence of a sender and at least one recipient. If either is missing, logs an error message with the record's index and details, increments the `failed_parses` counter, and skips to the next record.
- Creation of `ParsedEmail` Instance `ParsedEmail { from: record.sender.clone(), to: recipients }`: Constructs a `ParsedEmail` instance using the sender and the vector of recipients.
- Appending to `parsed_emails` `parsed_emails.push(parsed_email)`: Adds the newly created `ParsedEmail` to the vector of parsed emails.
- Final Reporting `println!("Successfully parsed {} emails.", parsed_emails.len())`: Outputs the total number of successfully parsed emails. If any records failed to parse, logs the total number of such failures.
- Function Return `Ok(parsed_emails)`: Returns the vector of parsed emails encapsulated in a `Result` type, allowing the caller to handle potential errors gracefully.

`graph.rs`

This segment of the project focuses on constructing a directed, unweighted graph from parsed email data and detecting communities within the network using the Label Propagation Algorithm (LPA).

```rust
use std::collections::{HashSet, HashMap};
use crate::ParsedEmail;
use rand::seq::SliceRandom;
use rand::thread_rng;
```

- `std::collections::{HashSet, HashMap}`: Utilized for storing the adjacency list of the graph. `HashMap` maps each node to its set of neighbors (`HashSet`), ensuring efficient lookup and insertion operations.

- `crate::ParsedEmail`: References the `ParsedEmail` struct defined in another module (presumably `email.rs`). This struct encapsulates the sender and recipients of each email, forming the basis for graph construction.
- `rand::seq::SliceRandom` and `rand::thread_rng`: Imported from the `rand` crate to facilitate random shuffling of nodes during the Label Propagation process, ensuring unbiased updates.

```
#[derive(Debug)]
pub struct Graph {
    pub num_vertices: usize,
    pub adjacency_list: HashMap<String, HashSet<String>>,
}
```

Models the communication network as a directed graph where each node represents a unique email address, and edges denote the direction of communication (from sender to recipient).

- `num_vertices: usize`: Tracks the total number of unique nodes (email addresses) in the graph.
- `adjacency_list: HashMap<String, HashSet<String>>`: Represents the adjacency list, mapping each node to a set of its neighboring nodes (i.e., recipients).
  Then several functions created within the implementation of the graph struct:

```
pub fn new() -> Self {
    Graph {
        num_vertices: 0,
        adjacency_list: HashMap::new(),
    }
}
```

- Initializes an empty `Graph` instance with no vertices and an empty adjacency list. Sets `num_vertices` to `0` and creates an empty `HashMap` for `adjacency_list`.

```
pub fn add_edge(&mut self, from_node: String, to_node: String) {
    self.adjacency_list.entry(from_node.clone())
        .or_insert_with(|| {
            self.num_vertices += 1;
            HashSet::new()
        });
    self.adjacency_list.entry(to_node.clone())
```

```
            .or_insert_with(|| {
                self.num_vertices += 1;
                HashSet::new()
            });
        self.adjacency_list
            .get_mut(&from_node)
            .unwrap()
            .insert(to_node.clone());
}
```

Establishes a directed edge from `from_node` (sender) to `to_node` (recipient), effectively capturing a single email communication.
Insertion of Sender (`from_node`):
`self.adjacency_list.entry(from_node.clone())`: Accesses the entry for `from_node` in the adjacency list.
`.or_insert_with(|| { self.num_vertices += 1; HashSet::new() })`: If `from_node` does not exist in the adjacency list, inserts it with a new empty `HashSet` and increments the `num_vertices` counter.
Insertion of Recipient (`to_node`):
`self.adjacency_list.entry(to_node.clone())`: Accesses the entry for `to_node` in the adjacency list.
`.or_insert_with(|| { self.num_vertices += 1; HashSet::new() })`: If `to_node` does not exist in the adjacency list, inserts it with a new empty `HashSet` and increments the `num_vertices` counter.
Adding the Edge:
`self.adjacency_list.get_mut(&from_node).unwrap().insert(to_node.clone())`: Retrieves a mutable reference to the `HashSet` of neighbors for `from_node` and inserts `to_node` into this set, establishing the directed edge.

```
pub fn build_from_emails(parsed_emails: Vec<ParsedEmail>) -> Self {
    let mut graph = Graph::new();

    for email in parsed_emails {
        let sender = email.from;
        let recipients = email.to;

        for recipient in recipients {
            graph.add_edge(sender.clone(), recipient);
        }
    }
    graph
}
```

Constructs the entire communication graph by iterating over a vector of `ParsedEmail` instances, each containing a sender and a list of recipients.
Initialization: creates a new, empty `Graph` instance. For each `ParsedEmail`, extracts the `sender` and iterates through the `recipients`, adding an edge from the sender to each recipient using the `add_edge` method.
Returns the fully constructed `Graph` after processing all emails.

```
pub fn get_neighbors(&self, node: &String) -> Option<&HashSet<String>> {
    self.adjacency_list.get(node)
}
```

Retrieves the set of recipients (neighbors) that a particular node (sender) has communicated with.
Parameter node is a reference to the sender's email address whose neighbors are to be fetched.
This function will returns `Some(&HashSet<String>)` if the node exists in the `adjacency_list`, containing all its recipients. Returns `None` if the node is not present, indicating it has no outgoing edges or is isolated.

```
pub fn calculate_out_degrees(&self) -> HashMap<String, usize> {
    let mut out_degrees = HashMap::new();

    for node in self.adjacency_list.keys() {
        let degree = self
            .get_neighbors(node)
            .map_or(0, |neighbors| neighbors.len());
        out_degrees.insert(node.clone(), degree);
    }

    out_degrees
}
```

This function computes the out-degree for every node in the graph, representing the number of emails each sender has dispatched.
It loops through each node (sender) in the `adjacency_list`. For each node, retrieves its neighbors using `get_neighbors`. If the node has recipients, the out-degree is the count of recipients; otherwise, it's zero. Then inserts the node and its corresponding out-degree into the `out_degrees`

`HashMap`. This function will return a `HashMap<String, usize>` where each key is a sender's email address, and the value is their out-degree.

```
pub fn calculate_in_degrees(&self) -> HashMap<String, usize> {
    let mut in_degrees = HashMap::new();
    for node in self.adjacency_list.keys() {
        in_degrees.insert(node.clone(), 0);
    }
    for neighbors in self.adjacency_list.values() {
        for neighbor in neighbors {
            if let Some(count) = in_degrees.get_mut(neighbor) {
                *count += 1;
            }
        }
    }

    in_degrees
}
```

This function can compute the in-degree for every node in the graph, representing the number of emails each recipient has received. The function first sets the in-degree of every node to zero. Then iterates over each sender's recipients and increments the in-degree count for each recipient accordingly.
The function will return a `HashMap<String, usize>` where each key is a recipient's email address, and the value is their in-degree.

```
pub fn label_propagation(&self) -> HashMap<String, String> {
    let mut labels: HashMap<String, String> = self.adjacency_list
        .keys()
        .map(|node| (node.clone(), node.clone()))
        .collect();

    let mut rng = thread_rng();

    let max_iterations = 500;
    for iteration in 0..max_iterations {
        let mut changed = false;


        let mut nodes: Vec<&String> =
self.adjacency_list.keys().collect();
        nodes.shuffle(&mut rng);
        for node in nodes {
            let neighbors = match self.get_neighbors(node) {
                Some(neigh) => neigh,
                None => continue,
            };

            if neighbors.is_empty() {
                continue;
            }

            let mut label_counts: HashMap<&String, usize> =
HashMap::new();
            for neighbor in neighbors {
                if let Some(label) = labels.get(neighbor) {
                    *label_counts.entry(label).or_insert(0) += 1;
                }
            }


            if let Some((&max_label, &max_count)) =
label_counts.iter().max_by_key(|&(_, count)| count) {
                let current_label = labels.get(node).unwrap();
                if current_label != max_label {
                    labels.insert(node.clone(), max_label.clone());
                    changed = true;
                }
            }
        }
    }
    labels
}
```

Implements the **Label Propagation Algorithm (LPA)** to detect communities within the communication network. Each community represents a cluster of individuals with dense interconnections.

I first Assign each node a unique label identical to its email address, ensuring that every node starts as its own community. Then I Initializes a thread-local RNG (`thread_rng()`) to facilitate random shuffling of node processing order in each iteration. The runs for a maximum of 500 iterations to prevent infinite loops in cases where labels continue to change without convergence. Then collect all nodes and shuffles their order randomly. This randomization is crucial to avoid bias in label updates and promotes fair community formation.

For each node in the shuffled list, obtain the set of recipients (neighbors) the node has communicated with. Then tallies the frequency of each label among the node's neighbors. Identifies the label with the highest frequency and updates the node's label to this dominant label if it differs from the current label. This action signifies the node adopting the community label most prevalent among its neighbors.

A `changed` flag tracks whether any label changes occurred during the iteration. While not explicitly used to terminate the loop early, it indicates ongoing community adjustments.

After completing all iterations, the algorithm concludes, returning the final `labels` mapping each node to its detected community.

`main.rs`
The main workflow orchestrates the process of reading email data, constructing the graph, performing degree distribution analysis, identifying top communicators, detecting communities, and analyzing these communities.

```rust
pub mod email;
pub mod graph;

use email::{ParsedEmail, read_csv};
use graph::{Graph};
use std::error::Error;
use std::collections::{HashMap, HashSet};
```

`email` and `graph`: Import the custom modules responsible for email parsing and graph operations.

- **`std::error::Error`:** Facilitates error handling by allowing the `main` function to return any error type that implements the `Error` trait.
- **`std::collections::{HashMap, HashSet}`:** Provides efficient data structures for storing and managing degree distributions and community mappings.

Function: `analyze_degree_distribution`
This function computes and displays statistical information about the graph's out-degree and in-degree distributions.
This function computes the In degree and Out degree statisitics:

- Total Nodes: Number of unique email addresses in the graph.
- Total Degrees: Sum of all out-degrees and in-degrees, respectively.
- Average Degrees: Average number of emails sent and received per node.

- Maximum and Minimum Degrees: Highest and lowest number of emails sent and received by any node.

Functions: `identify_top_senders` and `identify_top_recipients`
Identifies and retrieves the top N senders and recipients based on their out-degree and in-degree, respectively.
This function converts the `HashMap` of degrees into a vector of `(String, usize)` tuples for sorting.The vector is sorted in descending order based on degree values, ensuring that the top communicators are at the forefront. Then extracts the top N entries from the sorted vector using `.take(top_n)`.
The function will returns a vector containing the top N senders or recipients along with their corresponding degree counts.

Function: `print_top_individuals`
This function displays the top N senders (prolific communicators) and recipients (information hubs) based on their out-degree and in-degree.
The function calls `identify_top_senders` to retrieve the top N senders, and calls `identify_top_recipients` to retrieve the top N recipients. Then prints the top senders and recipients in a formatted manner, enumerating each with their respective email counts.

Function: `analyze_communities`
Analyzes the detected communities by computing and displaying key statistics, providing a overview of the community structures within the communication network.
This function first determine the total number of detected communities by assessing the length of the `community_map`.
The function then extracts the size of each community by mapping over the values (vectors of members) in the `community_map`. The sizes are sorted in descending order to facilitate easy identification of the largest and smallest communities.
Total node is calculated by summing all members across all communities.
Average community size is calculated by dividing the number of communities by the total number of users.
Largest Community Size is calculated by finding the first element of the sorted community size.
The smallest community size is calculated by finding the last element of the sorted community size.
The statistics are then printed.

Function: `identify_extreme_communities`
Identifies and displays detailed information about the largest and smallest communities detected within the communication graph.
Largest Community Identification: Uses `max_by_key` to find the community with the highest number of members. Prints the community label, number of members, and lists the top 10 members

for inspection.
Smallest Community Identification: Uses `min_by_key` to find the community with the fewest members. Prints the community label, number of members, and lists the sole member if it's a singleton.

Main function:
I start by defining the path to the CSV file containing email data (`emaildata_100000_0.csv`). Then the `read_csv` function from the `email` module to parse the CSV file into a vector of `ParsedEmail` instances. `Graph::build_from_emails` method is used to construct the communication graph from the parsed emails.
Calling `analyze_degree_distribution` to compute and display statistics related to out-degree and in-degree distributions, providing a foundational understanding of communication patterns.
Then I retrieves out-degrees and in-degrees using `calculate_out_degrees` and `calculate_in_degrees`, identifies and prints the top 10 senders and recipients using `print_top_individuals`, highlighting key communicators within the organization.
Executes the `label_propagation` method to detect communities within the graph. Then I structures the community assignments into a `HashMap<String, Vec<String>>`, mapping each community label to its members. Then I calls `analyze_communities` to compute and display community-related statistics, and called `identify_extreme_communities` to find and display the largest and smallest communities.

Test method:

1. `test_out_degree_calculation`:
   Validates that out-degree calculations accurately reflect the number of emails sent by each sender.

- Constructs a sample set of `ParsedEmail` instances.
- Builds the graph and calculates out-degrees.
- Asserts that the calculated out-degrees match expected values.

2. `test_in_degree_calculation`:
   Validates that in-degree calculations accurately reflect the number of emails received by each recipient.

- Constructs a sample set of `ParsedEmail` instances.
- Builds the graph and calculates in-degrees.
- Asserts that the calculated in-degrees match expected values.

3. `test_self_loops_and_multiple_edges`:
   Ensures that the graph correctly handles self-loops and prevents duplicate edges.

- Constructs a graph with self-loops and duplicate email entries.
- Calculates out-degrees and in-degrees.
- Asserts that duplicate edges do not inflate degree counts and that self-loops are correctly accounted for.

4. `test_label_propagation_small_graph`:
   Validates the Label Propagation Algorithm on a controlled small graph to ensure it correctly identifies distinct communities.

- Constructs a graph with two distinct communities.
- Performs label propagation.
- Asserts that exactly two communities are detected, each with the expected number of members.