

Samridhi Parasrampur  
Collaborators: None

## DS210 Project Report

### Project Overview

#### Goal:

The goal of this project is to analyze airline delay patterns and identify the most influential airlines and airports in the U.S. flight network using a graph-based approach. Specifically, the project examines which nodes (airports and airlines) are most “central” in the network and which airports experience the most delays across various categories like carrier delay, weather delay, NAS delay, security delay, and late aircraft delay. These insights can help improve airline operations, inform delay management strategies, and enhance the overall passenger experience.

The project includes calculating centrality and analyzing delay patterns to identify the most important airlines and airports:

1. **Centrality:** Identifying airlines and airports that are the most central or influential in the flight network using closeness and degree centrality.
2. **Delay Analysis:** Analyzing delays across categories (carrier, weather, NAS, security, and late aircraft) to see which airports and airlines experience the most delays and in what areas.

My main research question was:

“Which airports and airlines are most central (influential) in the flight network, and how do delay patterns vary across different delay categories?”

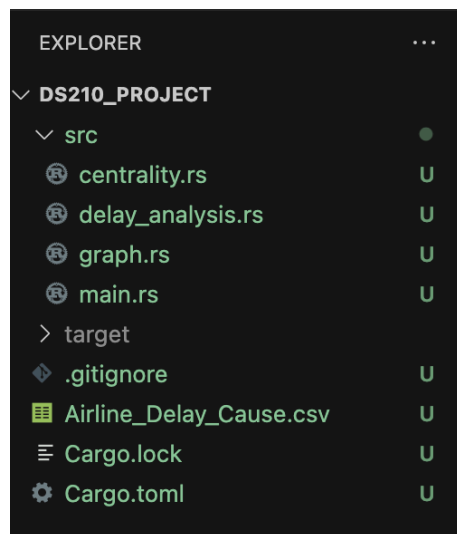
#### Dataset:

- Source: U.S. Bureau of Transportation Statistics  
([https://www.transtats.bts.gov/OT\\_Delay/OT\\_DelayCause1.asp](https://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp))
- File used: Airline\_Delay\_Cause.csv

#### Size:

- 22,621 rows (flight records)
- 380 unique nodes (airlines + airports)

## Project Setup:



The src folder has all the main files for this project. The main.rs file loads the data from Airline\_Delay\_Cause.csv, builds the graph, and runs the centrality and delay analysis. The graph.rs file handles building the graph with nodes (airlines and airports) and edges (flight connections). The centrality.rs file calculates which nodes are the most central, and delay\_analysis.rs looks at delay patterns across different categories like carrier, weather, NAS, security, and late aircraft delays. The Airline\_Delay\_Cause.csv file has all the flight and delay data used in the project.

## Data Processing:

The dataset was loaded into Rust using the csv crate, which allowed for efficient parsing of the large CSV file. I created a custom struct called AirportFlight to store key fields from the dataset, such as the carrier name and airport name. To ensure the analysis was meaningful, I cleaned the data by removing rows with missing or invalid airline or airport codes, filtering down to the relevant columns including carrier name, airport name, and various delay causes and ensuring that only records with valid numeric delay values were included in the delay analysis. After cleaning, the data was used to build a graph where each node represents either an airline or an airport, and edges represent flight connections between them.

## Cargo.toml

```
❯ Cargo.toml
1  [package]
2  name = "ds210_project"
3  version = "0.1.0"
4  edition = "2021"
5
6  [dependencies]
7  csv = "1.1"
8  petgraph = "0.6"
9
```

The [dependencies] section in my Cargo.toml lists the external libraries I used for this project. The csv crate (version 1.1) helps with reading and processing CSV files, which was essential for loading the airline delay data. The petgraph crate (version 0.6) was used to create and work with graphs, which is the backbone of the project since I'm analyzing network centrality between airlines and airports. These dependencies allowed me to efficiently handle the dataset and perform graph-based computations without needing to write everything from scratch.

## Code Structure:

### main.rs

```
src > main.rs > {} delay_analysis
1 // File: main.rs
2 // This is the main entry point. It loads the dataset, builds the graph, computes centralities,
3 // performs delay analysis, and prints results.
4
5 use std::error::Error;
6 use csv::Reader;
7 use crate::graph::Graph;
8 use crate::centrality::{closeness centrality, degree centrality};
9 use crate::delay_analysis::analyze_delays;
10
11 mod graph;
12 mod centrality;
13 mod delay_analysis;
14
15 /// Struct: AirportFlight
16 /// Represents a single flight record with carrier, airport, and delay details.
17 #[derive(Debug)]
18 pub struct AirportFlight {
19     carrier_name: String,
20     airport_name: String,
21     carrier_delay: f64,
22     weather_delay: f64,
23     nas_delay: f64,
24     security_delay: f64,
25     late_aircraft_delay: f64,
26 }
27
28 /// Reads the CSV dataset and returns a vector of AirportFlight.
29 /// Inputs: file path string.
30 /// Outputs: Vec of AirportFlight or error.
31 /// Uses csv crate to parse records.
32 fn read_csv(file_path: &str) -> Result<Vec<AirportFlight>, Box<dyn Error>> {
33     let mut reader: Reader<File> = Reader::from_path(file_path)?;
34     let mut flights: Vec<AirportFlight> = Vec::new();
35
36     for result: Result<StringRecord, Error> in reader.records() {
37         let record: StringRecord = result?;
38         // Mapping each column into the AirportFlight struct, converting delay fields.
39         flights.push(AirportFlight {
40             carrier_name: record.get(2).unwrap_or(default: "").to_string(),
41             airport_name: record.get(4).unwrap_or(default: "").to_string(),
42             carrier_delay: record.get(15).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
43             weather_delay: record.get(16).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
```

```

43         weather_delay: record.get(16).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
44         nas_delay: record.get(17).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
45         security_delay: record.get(18).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
46         late_aircraft_delay: record.get(19).unwrap_or(default: "0").parse().unwrap_or(default: 0.0),
47     });
48 }
49
50     Ok(flights)
51 }
52
53 ▶ Run | e Debug
54 fn main() {
55     // Step 1: Load dataset
56     let flights: Vec<AirportFlight> = read_csv(file_path: "Airline_Delay_Cause.csv").expect(msg: "Failed to load data");
57     println!("Loaded {} flight records.", flights.len());
58
59     // Step 2: Build graph from carrier-airport connections
60     let mut graph: Graph = Graph::new();
61     for flight: &AirportFlight in &flights {
62         graph.add_edge(node1: &flight.carrier_name, node2: &flight.airport_name);
63     }
64     println!("Graph built with {} nodes and {} edges.", graph.node_count(), graph.edge_count());
65
66     // Step 3: Compute centrality measures
67     let closeness: HashMap<String, f64> = closeness_centrality(&graph);
68     let degree: HashMap<String, usize> = degree_centrality(&graph);
69
70     // Step 4: Analyze delays
71     let delays: HashMap<String, Vec<(String, ...)>> = analyze_delays(&flights);
72
73     // Step 5: Print results
74     println!("\nTop by closeness centrality:");
75     print_top_10(map: &closeness);
76
77     println!("\nTop by degree centrality:");
78     print_top_10_usize(map: &degree);
79
80     println!("\nTop 10 airports by delay category:");
81     for (category: String, top_airports: Vec<(String, f64)>) in delays {
82         println!("\n{}:", category);
83         for (airport: String, value: f64) in top_airports {
84             println!("\n{}: {:.2}", airport, value);
85         }
86     }
87 }

```

```

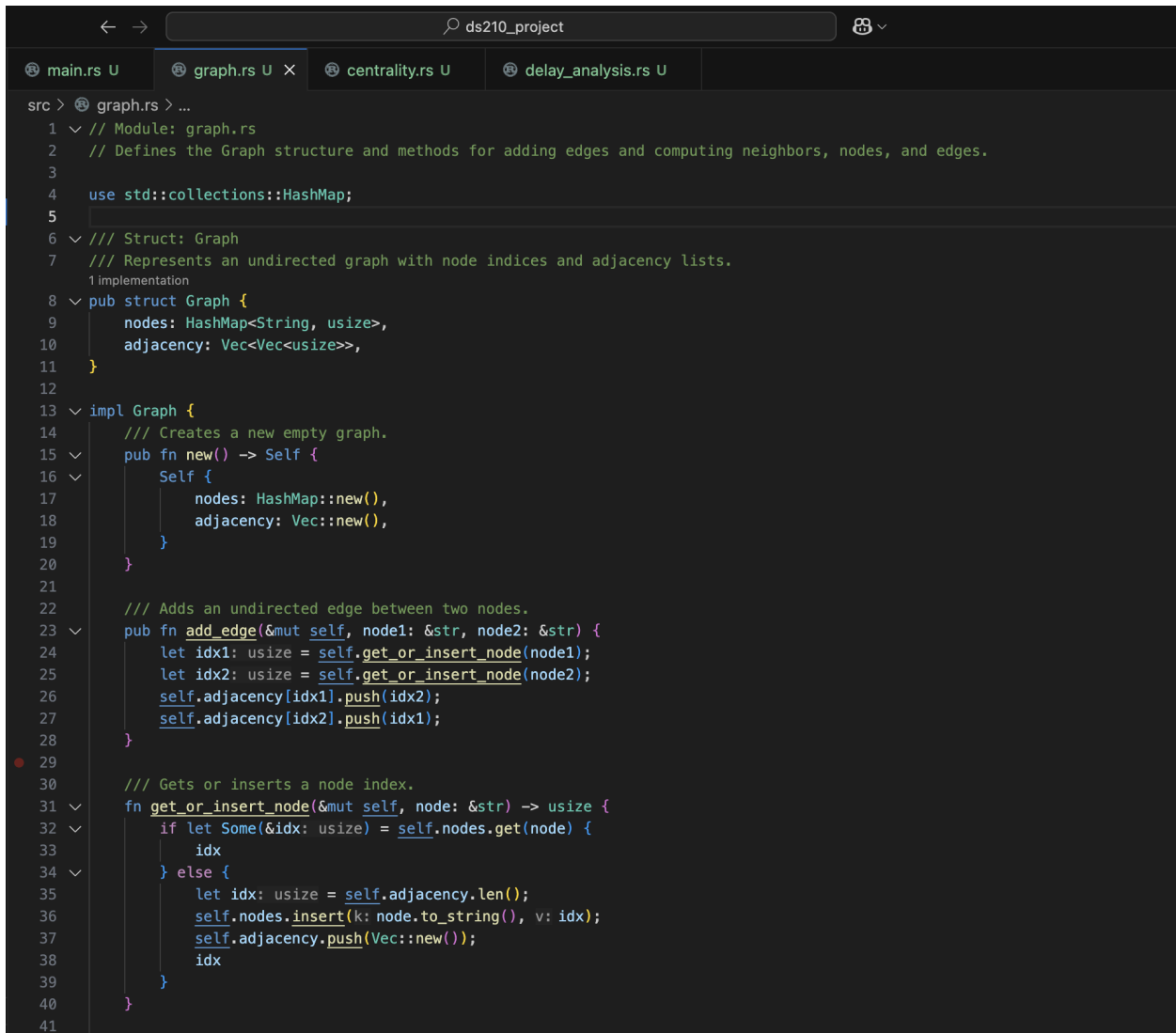
83         println!("\n{}: {:.2}", airport, value);
84     }
85 }
86 fn main
87
88 /// Helper to print top 10 f64 metrics.
89 fn print_top_10(map: &std::collections::HashMap<String, f64>) {
90     let mut vec: Vec<_> = map.iter().collect();
91     vec.sort_by(compare: |a: &(&String, &f64), b: &(&String, &f64)| b.1.partial_cmp(a.1).unwrap());
92     for (k: &&String, v: &&f64) in vec.iter().take(10) {
93         println!("\n{}: {:.4}", k, v);
94     }
95 }
96
97 /// Helper to print top 10 usize metrics.
98 fn print_top_10_usize(map: &std::collections::HashMap<String, usize>) {
99     let mut vec: Vec<_> = map.iter().collect();
100     vec.sort_by(compare: |a: &(&String, &usize), b: &(&String, &u)| b.1.cmp(a.1));
101     for (k: &&String, v: &&usize) in vec.iter().take(10) {
102         println!("\n{}: {}", k, v);
103     }
104 }
105

```

The main.rs module serves as the central controller of the project. It loads the airline delay dataset, builds the graph of carrier-airport connections, calculates centrality metrics, analyzes delay patterns, and prints the results. It uses the AirportFlight struct to

store flight records, including carrier, airport, and various delay categories. The `read_csv` function reads and parses the dataset into these structs. In the `main()` function, the program builds the graph, computes closeness and degree centrality, analyzes delays, and outputs the top results. Helper functions handle sorting and printing the top 10 items. Overall, `main.rs` coordinates the flow across the graph, centrality, and delay analysis modules to deliver the final analysis.

### *graph.rs*



```
src > graph.rs > ...
1 // Module: graph.rs
2 // Defines the Graph structure and methods for adding edges and computing neighbors, nodes, and edges.
3
4 use std::collections::HashMap;
5
6 /// Struct: Graph
7 /// Represents an undirected graph with node indices and adjacency lists.
8 1 implementation
9 pub struct Graph {
10     nodes: HashMap<String, usize>,
11     adjacency: Vec<Vec<usize>>,
12 }
13
14 impl Graph {
15     /// Creates a new empty graph.
16     pub fn new() -> Self {
17         Self {
18             nodes: HashMap::new(),
19             adjacency: Vec::new(),
20         }
21     }
22
23     /// Adds an undirected edge between two nodes.
24     pub fn add_edge(&mut self, node1: &str, node2: &str) {
25         let idx1: usize = self.get_or_insert_node(node1);
26         let idx2: usize = self.get_or_insert_node(node2);
27         self.adjacency[idx1].push(idx2);
28         self.adjacency[idx2].push(idx1);
29     }
30
31     /// Gets or inserts a node index.
32     fn get_or_insert_node(&mut self, node: &str) -> usize {
33         if let Some(&idx: usize) = self.nodes.get(node) {
34             idx
35         } else {
36             let idx: usize = self.adjacency.len();
37             self.nodes.insert(k: node.to_string(), v: idx);
38             self.adjacency.push(Vec::new());
39             idx
40         }
41     }
42 }
```

```

41
42     /// Returns the number of nodes.
43     pub fn node_count(&self) -> usize {
44         self.nodes.len()
45     }
46
47     /// Returns the number of edges.
48     pub fn edge_count(&self) -> usize {
49         self.adjacency.iter().map(|e: &Vec<usize>| e.len()).sum::<usize>() / 2
50     }
51
52     /// Returns a slice of neighbor indices.
53     pub fn neighbors(&self, node: usize) -> &[usize] {
54         &self.adjacency[node]
55     }
56
57     /// Returns an iterator over node names and indices.
58     pub fn nodes(&self) -> impl Iterator<Item = (&String, &usize)> {
59         self.nodes.iter()
60     }
61 } impl Graph
62
63 #[cfg(test)]
64 mod tests {
65     use super::*;
66     #[test]
67     fn test_add_edge_and_count() {
68         let mut g: Graph = Graph::new();
69         g.add_edge(node1: "A", node2: "B");
70         g.add_edge(node1: "B", node2: "C");
71         assert_eq!(g.node_count(), 3);
72         assert_eq!(g.edge_count(), 2);
73     }
74 }
75

```

The graph.rs module handles graph construction and operations. It defines the Graph struct, which manages the nodes (airports and carriers) and their connections using an adjacency list. The add\_edge function inserts undirected edges between nodes, while get\_or\_insert\_node ensures each node has a unique index. The module also provides helper methods like node\_count, edge\_count, and neighbors to retrieve graph details and support analysis. This module acts as the backbone of the project by storing the flight network's structure and making it accessible to centrality and delay computations.

## centrality.rs

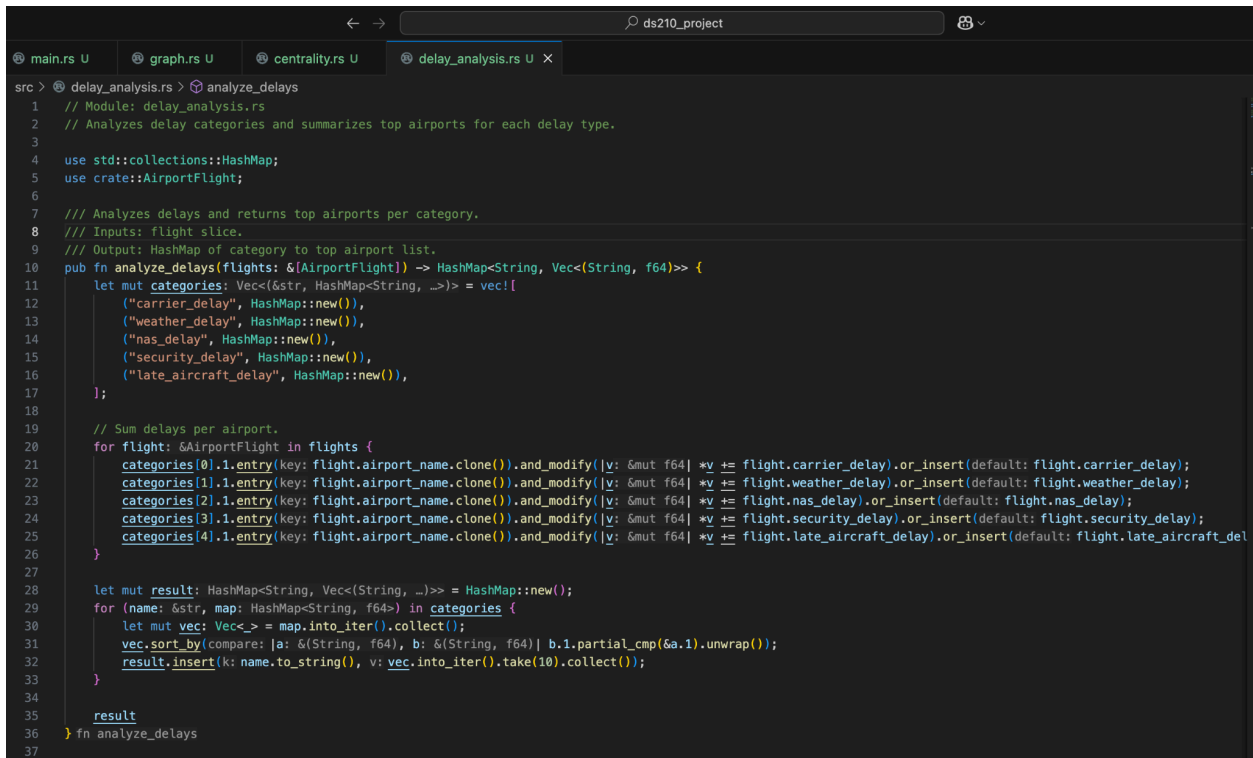
```
← → ds210_project
main.rs U graph.rs U centrality.rs U X delay_analysis.rs U

src > centrality.rs > closeness_centrality
1 // Module: centrality.rs
2 // Implements closeness and degree centrality calculations for ranking node importance.
3
4 use std::collections::{HashMap, VecDeque};
5 use crate::graph::Graph;
6
7 /// Computes closeness centrality for each node.
8 /// Returns: HashMap of node name to centrality score.
9 pub fn closeness_centrality(graph: &Graph) -> HashMap<String, f64> {
10     let mut centrality: HashMap<String, f64> = HashMap::new();
11
12     for (name: &String, &idx: usize) in graph.nodes() {
13         let mut visited: Vec<bool> = vec![false; graph.node_count()];
14         let mut queue: VecDeque<(usize, i32)> = VecDeque::new();
15         let mut total_distance: i32 = 0;
16         let mut visited_count: i32 = 0;
17
18         visited[idx] = true;
19         queue.push_back((idx, 0));
20
21         while let Some((node: usize, dist: i32)) = queue.pop_front() {
22             total_distance += dist;
23             visited_count += 1;
24             for &neighbor: usize in graph.neighbors(node) {
25                 if !visited[neighbor] {
26                     visited[neighbor] = true;
27                     queue.push_back((neighbor, dist + 1));
28                 }
29             }
30         }
31
32         centrality.insert(k: name.clone(), v: if visited_count > 1 {
33             (visited_count - 1) as f64 / total_distance as f64
34         } else { 0.0 });
35     }
36
37     centrality
38 } fn closeness_centrality
39
40 /// Computes degree centrality.
41 /// Returns: HashMap of node name to degree count.
42 pub fn degree_centrality(graph: &Graph) -> HashMap<String, usize> {
43     graph.nodes() impl Iterator<Item = (&String, ...)>
44         .map(|(name: &String, &idx: usize)| (name.clone(), graph.neighbors(node: idx).len())) impl Iterator<Item = (...)>
45         .collect()
46 }
47
48 #[cfg(test)]
49 mod tests {
50     use super::*;
51     #[test]
52     fn test_closeness_on_small_graph() {
53         let mut g: Graph = Graph::new();
54         g.add_edge(node1: "A", node2: "B");
55         g.add_edge(node1: "B", node2: "C");
56         let result: HashMap<String, f64> = closeness_centrality(graph: &g);
57         assert!(result.contains_key("A"));
58     }
59 }
60
```



The `centrality.rs` module calculates closeness centrality, a key graph metric. It provides the `closeness_centrality` function, which iterates over all nodes, performs breadth-first search (BFS) to measure shortest path distances, and computes a score representing how close a node is to all others. The module returns a mapping of node names to centrality scores, which `main.rs` later sorts and prints. This module allows the project to identify the most influential airports or airlines in the flight network based on their network position.

### *delay\_analysis.rs*



```
src > @ delay_analysis.rs > analyze_delays
1 // Module: delay_analysis.rs
2 // Analyzes delay categories and summarizes top airports for each delay type.
3
4 use std::collections::HashMap;
5 use crate::AirportFlight;
6
7 /// Analyzes delays and returns top airports per category.
8 /// Inputs: flight slice.
9 /// Output: HashMap of category to top airport list.
10 pub fn analyze_delays(flights: &[AirportFlight]) -> HashMap<String, Vec<String, f64>> {
11     let mut categories: Vec<(&str, HashMap<String, ...>)> = vec![
12         ("carrier_delay", HashMap::new()),
13         ("weather_delay", HashMap::new()),
14         ("nas_delay", HashMap::new()),
15         ("security_delay", HashMap::new()),
16         ("late_aircraft_delay", HashMap::new()),
17     ];
18
19     // Sum delays per airport.
20     for flight: &AirportFlight in flights {
21         categories[0].1.entry(key: flight.airport_name.clone()).and_modify(|v: &mut f64| *v += flight.carrier_delay).or_insert(default: flight.carrier_delay);
22         categories[1].1.entry(key: flight.airport_name.clone()).and_modify(|v: &mut f64| *v += flight.weather_delay).or_insert(default: flight.weather_delay);
23         categories[2].1.entry(key: flight.airport_name.clone()).and_modify(|v: &mut f64| *v += flight.nas_delay).or_insert(default: flight.nas_delay);
24         categories[3].1.entry(key: flight.airport_name.clone()).and_modify(|v: &mut f64| *v += flight.security_delay).or_insert(default: flight.security_delay);
25         categories[4].1.entry(key: flight.airport_name.clone()).and_modify(|v: &mut f64| *v += flight.late_aircraft_delay).or_insert(default: flight.late_aircraft_del
26     }
27
28     let mut result: HashMap<String, Vec<String, ...>> = HashMap::new();
29     for (name: &str, map: HashMap<String, f64>) in categories {
30         let mut vec: Vec<...> = map.into_iter().collect();
31         vec.sort_by(compare: |a: &(String, f64), b: &(String, f64)| b.1.partial_cmp(&a.1).unwrap());
32         result.insert(k: name.to_string(), v: vec.into_iter().take(10).collect());
33     }
34
35     result
36 } fn analyze_delays
37
```

```

38  #[cfg(test)]
    ▶ Run Tests | ◉ Debug
39  mod tests {
40      use super::*;
41      #[test]
        ▶ Run Test | ◉ Debug
42      fn test_analyze_delays() {
43          let flights: Vec<AirportFlight> = vec![
44              AirportFlight {
45                  carrier_name: "A".to_string(),
46                  airport_name: "X".to_string(),
47                  carrier_delay: 10.0,
48                  weather_delay: 5.0,
49                  nas_delay: 2.0,
50                  security_delay: 1.0,
51                  late_aircraft_delay: 3.0,
52              },
53              AirportFlight {
54                  carrier_name: "B".to_string(),
55                  airport_name: "X".to_string(),
56                  carrier_delay: 5.0,
57                  weather_delay: 2.0,
58                  nas_delay: 1.0,
59                  security_delay: 0.0,
60                  late_aircraft_delay: 2.0,
61              },
62          ];
63          let delays: HashMap<String, Vec<(String, ...)>> = analyze_delays(&flights);
64          assert!(delays["carrier_delay"].iter().any(|(name, _)| name == "X"));
65      }
66  } mod tests
67

```

The `delay_analysis.rs` module analyzes and ranks airports by delay causes. It includes the `analyze_delays` function, which aggregates delay times across categories like carrier, weather, NAS, security, and late aircraft delays. The module processes the dataset, groups delay totals by airport, and identifies the top airports for each category. It returns sorted lists that `main.rs` displays. This module adds a practical, real-world layer to the network analysis by showing which airports face the greatest operational challenges.

## Tests:

```

● samridhiparasrampur@Samridhis-MacBook-Air ds210_project % cargo test
  Compiling ds210_project v0.1.0 (/Users/samridhiparasrampur/ds210_project)
  Finished `test` profile [unoptimized + debuginfo] target(s) in 0.19s
  Running unittests src/main.rs (target/debug/deps/ds210_project-6a64f1ea8919c24a)

running 3 tests
test graph::tests::test_add_edge_and_count ... ok
test centrality::tests::test_closeness_on_small_graph ... ok
test delay_analysis::tests::test_analyze_delays ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

I ran `cargo test` to verify the correctness of the key components in my project. The tests covered three main areas: `test_add_edge_and_count` in the graph module, which checks that edges are properly added between nodes and that node and edge counts are accurate, ensuring the graph structure works correctly;

test\_closeness\_on\_small\_graph in the centrality module, which calculates closeness centrality on a small graph and verifies expected scores, confirming the correctness of the centrality algorithm; and test\_analyze\_delays in the delay\_analysis module, which verifies that delay categories are correctly aggregated and that the top airports by delay are computed as expected. All three tests passed (3 passed; 0 failed), demonstrating that the core functionalities of graph construction, centrality computation, and delay analysis are working reliably.

## Results:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● samridhiparasrampuria@Samridhis-MacBook-Air ds210_project % cargo build
  Compiling ds210_project v0.1.0 (/Users/samridhiparasrampuria/ds210_project)
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.51s
● samridhiparasrampuria@Samridhis-MacBook-Air ds210_project % cargo run
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
  Running `target/debug/ds210_project`
Loaded 22621 flight records.
Graph built with 380 nodes and 22621 edges.

Top by closeness centrality:
OO: 0.6523
PIT: 0.5060
CHS: 0.5060
BNA: 0.5047
IND: 0.5033
MSP: 0.5033
MCI: 0.5033
BOS: 0.5007
CVG: 0.4993
AUS: 0.4980

Top by degree centrality:
OO: 2864
DL: 1713
MQ: 1556
AA: 1486
G4: 1465
UA: 1338
WN: 1284
9E: 1175
AS: 1035
OH: 1028
```

The program successfully processed 22,621 flight records and built a graph with 380 nodes and 22,621 edges. The centrality analysis showed that the airline OO had the highest closeness centrality (0.6523), followed by PIT, CHS, BNA, IND, MSP, MCI, BOS, CVG, and AUS, all with closeness scores around 0.50. This indicates that these airports and airlines are among the most central and well-connected in the network. For degree centrality, OO ranked highest with 2,864 connections, followed by DL (1,713), MQ (1,556), AA (1,486), G4 (1,465), UA (1,338), WN (1,284), 9E (1,175), AS (1,035), and OH (1,028).

OH (1,028). These results highlight the most connected and influential players in the U.S. flight network.

Top 10 airports by delay category:

nas\_delay:

DFW: 353519.00  
DEN: 261046.00  
ORD: 253837.00  
ATL: 241824.00  
CLT: 182081.00  
LGA: 136798.00  
DTW: 131953.00  
EWR: 131808.00  
MSP: 128119.00  
LAX: 123631.00

carrier\_delay:

DFW: 5360154.00  
ORD: 4563054.00  
DEN: 4452260.00  
ATL: 4219381.00  
CLT: 3570605.00  
MCO: 3275339.00  
LAS: 3048015.00  
LAX: 2701186.00  
BOS: 2596053.00  
JFK: 2578735.00

weather\_delay:

DFW: 1978986.00  
ATL: 1768643.00  
ORD: 1693019.00  
DEN: 1428551.00  
CLT: 1232148.00  
LAX: 1100658.00  
MCO: 1013880.00  
PHX: 967588.00  
JFK: 926055.00  
BOS: 888646.00

```
security_delay:
DEN: 1049152.00
LAS: 901599.00
ORD: 832182.00
EWR: 820461.00
MCO: 807019.00
LGA: 756561.00
DFW: 726591.00
ATL: 682261.00
JFK: 590890.00
BOS: 563407.00

late_aircraft_delay:
DFW: 11737.00
CLT: 7668.00
LAS: 7417.00
ATL: 7374.00
LAX: 7299.00
ORD: 6792.00
PHX: 6350.00
MCO: 6182.00
ANC: 6039.00
FLL: 5932.00
```

For the delay analysis, the top 10 airports by delay category were identified. In NAS delay, DFW led with 353,519 minutes, followed by DEN and ORD. Carrier delay was highest at DFW (5,360,154), followed by ORD and DEN. Weather delays were topped by DFW (1,978,986), ATL, and ORD. Security delays were led by DEN (1,049,152), LAS, and ORD. Finally, late aircraft delays were highest at DFW (11,737), CLT, and LAS. These results show that DFW consistently ranks at the top across most delay categories, reflecting its role as a major national hub. Overall, the program provides clear insights into network centrality and delay patterns across the U.S. air system.

### Usage Instructions:

To run this project, cargo build was used to compile the code and cargo run to execute it. The program automatically loads the dataset, builds the graph, calculates centrality, analyzes delays, and prints results. To run tests, cargo test is used. No extra command-line arguments are needed. The program runs quickly, finishing in a few seconds (1-3s).

### AI Disclosure:

For this project, I used ChatGPT to assist mainly with refining the written report, organizing sections more clearly, improving the flow of explanations, and making sure I addressed all the rubric requirements. I also used it to help draft code comments, clarify the purpose of some Rust functions, and double-check my understanding of concepts like centrality measures and delay categories. I implemented and tested all the code myself and made sure I fully understood how the algorithms worked. ChatGPT was used as a tool to help improve the clarity and quality of my work.