Instructions to run:

1. Please run the code with "**cargo run –release**" due to the large size and complexity of the dataset and code. The run time is approximately 10 mins for PageRank, and 2 mins for Kmeans Clustering.
2. Please use "**cargo test**" to run the tests.

My Goals:

I designed this project to analyze academic paper citation data using PageRank and K-Means Clustering algorithms. The dataset chosen include information about paper citations: edges in a directed graph where nodes represent papers.

1. **Rank Papers**: Use the PageRank algorithm to determine the relative importance of each paper in the citation network, based on the number of times it is being cited.
2. **Cluster Papers**: Apply K-means clustering to group similar papers based on their citation relationships, intended to discover patterns or themes within the citation network. Potentially finding a subgroup where the papers have similar research topics.

My findings:

1. I computed the pagerank of the graph with 10 random walks and 50 iterations. Then printed out the top 5 most cited papers. The pagerank results show some of the most influential papers in the dataset because, although the PageRank values are numerically small, they are normalized such that the sum of all PageRank scores equals 1. (A pagerank of ~0.004 is significant in a dataset of over 400000 nodes.)
2. I performed K-means clustering on the graph with various values of k and each for 10 iterations. Regardless of the chosen k, the results consistently showed that nearly all nodes were assigned to the 1 cluster, while the remaining clusters contained only the centroid itself. I believe this outcome stems from the relatively sparse and evenly distributed citation structure between the papers in the dataset. This makes sense in the real-world context, as paper citations tend to be less frequent and more spread out compared to other types of networks, such as social media, where connections are more dense and interconnected. This hypothesis is supported by examining the adjacency list of the graph and the result I got from my PageRank algorithm (Only a few key papers are frequently cited by many others, most are sparsely connected). As a result, the clustering did not reveal meaningful subgroups of research papers, likely because such subgroups do not exist within this particular dataset.

Structure of this Project:

1. **main.rs**: This is where all modules are used and run. It coordinates the execution of both the PageRank and K-means clustering algorithms. It reads the citation dataset from a CSV file, performs analysis, and prints the results. My tests are also written inside this file, stored in its own test module to ensure that it only runs with "cargo test".
2. **graph.rs**: This module defines the Graph struct and functions to manage the directed graph representing paper citations. It includes functionality for building the graph from a CSV file and running a random walk simulation on the graph. The rule for the walk is if the current node has no neighbors, it jumps to a random node in the graph. If the current node has neighbors, it jumps to one of its neighbors with 90% probability, and jumps to a random node in the graph with 10% probability.
3. **clustering.rs**: This module contains the logic for performing K-means clustering on the graph data. It also includes methods for converting the adjacency lists (citations) into feature vectors that store each node's connection with the rest of the nodes, and assigning papers to clusters.

Functions and Tests

1.
   ```
   fn create_directed_graph(edges: Vec<(Node, Node)>) -> Graph
   ```

   This function creates a directed graph from a list of edges. Each edge is represented as a tuple `(from, to)`, where `from` and `to` are the starting and ending nodes, respectively. The graph is stored into the Graph struct as an adjacency list using a `HashMap`, where each key is a node and the value is a vector of its neighbors.

2.
   ```
   fn random_walk(&self, current: &Node, steps: usize) -> Node
   ```

   This function performs a random walk on the graph starting from a given node, it returns the node where the random walk ends after the specified number of steps.

   The walker:

   - Jumps to a random node if the current node has no outgoing edges.
   - With 90% probability, moves to a random neighbor of the current node.
   - With 10% probability, jumps to a random node in the entire graph.

3.
   ```
   fn read_graph_from_csv(filename: &str) -> Result<Graph, Box<dyn
   std::error::Error>>
   ```

   Reads a graph from a CSV file and constructs its adjacency list. The CSV file have tab-separated values (`\t`) with at least two columns representing edges. Invalid or malformed rows are skipped with warnings. The function returns a `Graph` object or an error if the file cannot be read.

4.
   ```
   fn kmeans_clustering(adjacency_lists: &HashMap<Node, Vec<Node>>, k: usize,
   iterations: usize) -> Vec<Cluster>
   ```

Performs k-means clustering on a graph represented by adjacency lists using helper functions. It outputs a vector of `Cluster` objects, where each cluster has: “centroids” - The node closest to the cluster's centroid. “Members” - A set of nodes assigned to the cluster.

5. `fn convert_to_feature_vectors(adjacency_lists: &HashMap<Node, Vec<Node>>, nodes: &[&Node]) -> Vec<Vec<f64>>`

   Converts each node's adjacency list into a binary feature vector, where each feature indicates whether the node has a direct edge to another node in the graph.

6. `fn find_closest_node(centroid: &Vec<f64>, data: &[Vec<f64>], nodes: &[&Node]) -> String`

   Computes the Euclidean distance between the centroid (a feature vector) and each node's feature vector. - Returns the name of the node with the smallest distance to the centroid, effectively assigning it as the representative of that cluster.

7. `fn page_rank(graph: &Graph, steps: usize, iterations: usize) -> HashMap<String, f64>`

   For each node in the graph, performs multiple random walks, then tracks where each random walk terminates in a HashMap. Calculates the PageRank of each node as the proportion of walks ending at that node over the total number of walks. Returns a HashMap where the keys are node IDs, and the values are their corresponding PageRank scores.

8. `fn main() -> Result<(), Box<dyn Error>>`

   Reading a graph from a CSV file, computing PageRank, and performing k-means clustering. Print out messages to indicate what step it is at when running the code.

9. `fn test_pagerank_sum_to_one()`

   Ensures that the PageRank values from the citations.csv sum to 1, as they represent a probability distribution. Asserts that the sum of all PageRank scores is approximately 1 within a small margin of error because of the small rounding errors from floating-point precision limitations in computer systems.

10. `test_kmeans_clustering()`

    Creates a small directed graph with predefined edges. Runs the KMeans clustering function with 2 clusters and 5 iterations. Asserts that: The number of clusters is 2. Each cluster has at least one member. Each cluster has an assigned centroid.