Project description:

The Road Network Analysis and Optimization project aim to develop a robust system for analyzing and optimizing road networks using graph algorithms and data analysis techniques. The project focuses on processing road network data , performing graph analysis tasks such as BFS and DFS, selecting key nodes based on degree, and evaluating network performance metrics.

Potential use cases:

- Urban Planning: Analyzing road networks for urban development, traffic management, and infrastructure planning.
- Transportation Management: Optimizing routes, identifying congestion points, and improving public transportation systems.
- Logistics and Supply Chain: Analyzing supply chain routes, optimizing delivery networks, and reducing transportation costs.
- Emergency Response: Identifying critical nodes for emergency response planning and resource allocation during disasters or emergencies.

The Road Network Analysis and Optimization project will result in a software tool or system that enables users to analyze, optimize, and visualize road networks for various applications. The project aims to provide valuable insights and recommendations for improving network efficiency, reliability, and performance in real-world scenarios.

Explanation of code:

This project has been split into two modules, one called dfsbfs.rs, and another called readparse.rs. The first module defines a graph structure using an adjacency list representation and implements several graph-related operations such as adding edges, BFS and DFS searching algorithms, computing top-degree nodes and calculating average path lengths. The second module utilizes the graph from dfsbfs.rs, populates a graph object, and writes the graph's adjacency list to a CSV file. And finally within the main.rs, data is analyzed and results are outputted.

dfsbfs.rs:

1. First, I import the Hashmap and VecDeque data structures from the Rust standard library. HashMap is used to store the adjacency list representing the graph, and Vecque is used for the queue operation in the BFS algorithm that will be implemented later.

2. Next, a struct called 'Graph' is created, with one single field, a HashMap called 'adjacency_list'. Each key is an i32, that represents a node, and its value (Vec<i32>) represents a list of nodes with which it is connected.

3. My first function is called add_edge, and it modifies the graph noted by &mut self. It takes two parameters, both of type i32, and adds a directed edge from node src (short for source) to node dst (short for destination), it inserts dst into the adjacency list of src, and if it is not already in the HashMap, it initializes a new empty vector for it.

- *self.adjacency_list.entry(src)* accesses the adjacency_list field of the Graph. The 'entry' method is called with src as the argument. The method returns an 'entry' into the hashmap (Graph) for key src.
- *or_insert_with(Vec::new( ))*, is then called on the entry– This checks if the entry for the given key 'src' exists, if it doesn't, a new value is inserted, in this case, a new empty vector (*Vec::new()),* returning a mutable reference to the value in the entry.
- Finally with *.push(dst),* the destination node 'dst' is appended to the vector associated with the source node src, effectively adding an edge from src to dst.

4. My next function begins the BFS: I chose to use BFS to find the shortest path in my given data set.

- It initializes a mutable HashMap to store the distances from the start_node to other nodes. And a mutable queue (Vecdeque) to manage nodes as they are visited. Start node is added to the queue and the distance to itself is 0 – explaining line 23 and 24.

- After, a loop is created that continues to iterate as long as there are nodes in the queue.
- *while let Some(current_node) = queue.pop_front()*, uses pattern matching to check if there are nodes to process, and the *queue.pop_front()* method removes the first element from the queue (node that will be processed). If the queue is not empty it returns Some(value), where 'value' is the removed element. If the queue is empty, 'none' is returned. Once Some(value) is returned, it is matched and assigned to *current_node*.
- *self.adjacency_list.get(&current_node),* retrieves the list of neighbors of *current_node* from the graph's adjacency list.

- Then, with another loop, for each neighbor of *current_node, if !distances.contains_key(&neighbor),* checks if the neighbor has or has not been visited.
- Within the loop, *if let Some(neighbors) = self.adjacency_list.get(&current_node)* is used to look up the current_node in the Graph HashMap; if current_node exists as a key, it returns a reference to the vector containing the neighbors of 'current_node'.
- For each neighbor of current_node, *if !distances.contains_key(&neighbor)* checks if the neighbor has been visited, if not *distances.insert(neighbor, distances[&current_node] + 1)* inserts the distance of the neighbor.
- The last step *queue.push_back(neighbor)* then adds said neighbor to the queue. The 'distances' HashMap is returned at the end.

5. The next function written is for DFS. Finds a path from start_node to target_node, which are the inputs it will take in. It returns an Option<i32> which will contain the depth of the path between these nodes.

- Two variables are first initialized, 'stack' which is a vector of tuples, each tuple indicating the start_node, and the depth to the target_node. This depth is initialized as 0. The second variable is a HashMap that keeps track of all already visited nodes.
- Similar to the previous function, the DFS loop continues to iterate as long as there are nodes in the stack.
- *stack.pop(),* removes the last element from the stack, and processes that node. The next two lines check if the current node has already been visited, and then if the current node is the target node– in that case the depth is returned. The *current_node* is then marked as visited. This is where the neighbors get processed.
- *self.adjacency_list.get(&current_node)* retrieves the neighbors of the current node, and then for each neighbor in neighbors, the code checks if the neighbors has been visited , and adds the neighbors to the stack, incrementing the depth.

6. I created a top_degree_nodes function to help identify highly connected nodes, which is then used in the main.rs. It finds the nodes with the highest number of connections .

- initializes a vector 'node_degree' to store tuples of node IDs and their corresponding degrees. It populates 'node_degrees' by iterating over the adjacency list, mapping each node to its degree. Line 69 and 70, uses sort_by and into_iter to take the top n nodes with the highest degrees and returns them in a vector.

7. The average_path_length_subset function, offers insights into the average distance or accessibility between nodes in a specified subset, aiding in network analysis and optimization. The average_path_length function uses the BFS algorithm to calculate the average path length from a subset of nodes (subset because the data set was too large).

- I initialize my variables total_average and count. Which are used in the final calculation (total_average/ count = average).
- I use a loop that iterates over each node, sums the bfs distance from the current node to all other reachable nodes in the graph, and totals it. Number of paths is determined by taking the length of the 'distances' map, and subtracting 1 (the start node).
- Averages are updated by incrementing the total_average, by dividing the total distance to the number of paths found prior. The count of nodes is also incremented +1. After calculating all nodes in the subset, the function divides 'total' by 'num_paths' and returns that value.

readparse.rs:

This is where the txt file with my road data is converted into a graph. I have an extra function that turns the txt file into a csv file as well – which may make it easier to share and import the graph data. In this module, I have two imports, one for file handling, and the other for reading and writing of files.

1. The first function is called from_file, reads the txt file and populates the graph. It takes a string as the input (file path), and outputs a Graph.
   - file_path: &str: Parameter for the file path.
   - Graph::new(): Initializes an empty graph.
   - File::open(file_path)?: Opens the file at the provided path, returning an error if the operation fails (? operator propagates the error).
   - BufReader::new(file): Wraps the file in a buffer to optimize reading operations.
   - reader.lines().enumerate(): Iterates over lines of the file, with each line wrapped in a result to handle potential errors. enumerate() adds an index to each line.
   - if index > 0: Skips the first line assuming it's a header.
   - split_whitespace().collect(): Splits each line into parts based on whitespace.
   - match (parts[0].parse::<i32>(), parts[1].parse::<i32>()): Tries to parse the two parts of the line as integers.
   - graph.add_edge(src, dst): Adds an edge to the graph if both numbers are successfully parsed.

- eprintln!: Prints an error message to standard error if the line format is invalid.

2. The second function is the conversion to a csv file
   - File::create(file_path)?: Creates a new file for writing.
   - writeln!(file, "FromNodeId,ToNodeId")?: Writes the CSV header.
   - for (src, destinations) in &graph.adjacency_list: Iterates over the adjacency list of the graph.
   - writeln!(file, "{},{}", src, dst)?: Writes each source node and its destination nodes as rows in the CSV file.

Main.rs:

Inputting of data, and some parameters for analysis:

- let input_path = "roadNet-PA.txt";: Specifies the input file path containing the road network data.
- let output_csv_path = "roadNet-PA.csv";: Specifies the output CSV file path where the converted graph data will be saved.
- let graph = from_file(input_path)?;: Reads the graph data from the input file using the from_file function defined in the readparse module. The ? operator is used for error handling if reading the file fails.
- The start and target node are arbitrary values chosen for demonstration of my bfs/dfs algorithms
- to_csv(&graph, output_csv_path)?;: Converts the graph data (graph) to CSV format using the to_csv function defined in the readparse module and writes it to the specified output CSV file. The ? operator is used for error handling.

Graph Analysis:

- let bfs_distances = graph.bfs(start_node);: Performs a breadth-first search (BFS) starting from the start_node to find distances to all other reachable nodes in the graph.
- if let Some(distance) = bfs_distances.get(&target_node) { ... }: Checks if there is a path from the start_node to the target_node using BFS and prints the shortest path distance if a path exists.
- let dfs_result = graph.dfs(start_node, target_node);: Performs a depth-first search (DFS) from the start_node to the target_node to find a path between them.
- if let Some(depth) = dfs_result { ... }: Checks if a path is found using DFS and prints the depth (number of edges) of the path if found.

Graph Analysis and Node Selection:

- let center_nodes = graph.top_degree_nodes(100);: Selects the top 100 nodes with the highest degree in the graph using the top_degree_nodes function. This is done because data set was too large, and run time was long
- let average_path_length = graph.average_path_length_subset(&center_nodes);: Calculates the average path length from the selected center nodes to all other reachable nodes in the graph using the average_path_length_subset function.

Finally, the program prints the results of BFS, DFS, and the average path length from the top center nodes, and it returns Ok(()) if all operations complete successfully.

- BFS prints, the shortest path, and its length
- DFS prints possible path, and depth of said path