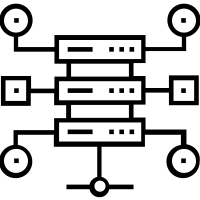
**Assignment No: \_5\_\_**

**Date: 03/ 05 /2025**

**Title: Implementation and Performance Comparison of Tree Data Structures for Efficient File System Management**

(Title based on the application domain and the data structure you will be implementing)

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment Type of Submission:** |  |  |  |
| **Group** | Yes | Yuxuan Song  24207239  Yijun Liu  24202574  Deepak Shelke  24208478 | **50**  **50**  **0** |

****

1. **Problem Domain Description:**

1This report investigates the application of graph data structures to model and solve real-world routing problems in transportation networks. The simulated scenario involves a network of virtual cities, where each city is connected to others by roads of varying distances. Such a system mimics practical use cases found in logistics, urban planning, or navigation systems, where route optimization and distance calculation between locations are essential.

To emulate a realistic environment, a dataset comprising 300 unique cities and 10,000 direct road connections was generated. Each city is assigned a distinct alphabetic identifier ranging from single-letter names (e.g., A, B, C) to double-letter combinations (e.g., AA, AB, AC), ensuring a wide variety of node identifiers. The connections between cities are randomly assigned to form an undirected, weighted graph where each edge represents a two-way road with a randomly generated distance between 10 and 999 kilometers. These connections are stored in a CSV file named routes.csv, where each line represents a direct route between two cities along with its associated distance.

Unlike naive representations such as adjacency matrices, which consume excessive memory for sparse graphs, this experiment adopts an adjacency list approach using hash-based maps. This representation allows for efficient graph traversal and edge lookup operations, making it highly suitable for handling large and dynamically changing route networks. The graph structure is subsequently used to compute shortest paths between city pairs using Dijkstra’s algorithm, a well-established solution for single-source shortest path problems in graphs with non-negative edge weights.

The program uses city Q as the source and city I as the destination to compute the shortest path between them. Based on the generated graph and the application of Dijkstra’s algorithm, the resulting optimal route is: Q -> AK -> AB -> EV -> T -> I, representing the path with the minimum total distance among all possible routes connecting the two cities. And the shortest distance is 140.

This experiment not only demonstrates how graph data structures can effectively represent and operate on large-scale city networks but also sets the foundation for integrating further real-world features such as traffic conditions, directional constraints, or real-time route updates in future extensions.

1. **Theoretical Foundations of the Data Structure(s) utilised**

Unlike hash tables or arrays, Java does not provide a built-in Graph class in its standard libraries. Therefore, graph data structures must be implemented manually using combinations of native Java collections such as HashMap, ArrayList, and PriorityQueue. A graph is a non-linear data structure that consists of nodes (also called vertices) and edges connecting pairs of nodes. Graphs can be directed or undirected, weighted or unweighted, and may support various traversal and pathfinding operations.

In our implementation, we represent a directed weighted graph, where each edge has an associated non-negative cost (or distance) and direction from one city to another. Each city is modeled as a node (String), and its connections to neighboring cities are stored in an adjacency list, implemented via a HashMap<String, List<CityRoute>>. Each CityRoute object holds the destination city and the weight (distance) of the edge. This structure offers a space-efficient way to model sparse graphs, where not all nodes are connected to every other node.

Graphs can represent a wide range of real-world scenarios such as road networks, social networks, and dependency hierarchies. There are several common graph types:

Undirected Graph: Edges have no direction. If A is connected to B, B is also connected to A.

Directed Graph (Digraph): Edges have direction, meaning a path from A to B does not imply a path from B to A.

Weighted Graph: Each edge has a cost, distance, or weight.

Unweighted Graph: All edges are treated as having equal cost.

Cyclic/ Acyclic Graph: Indicates whether or not cycles exist within the graph.

To compute the shortest path between two cities, our program implements Dijkstra’s algorithm, a well-established graph algorithm for finding the minimum-cost path from a source node to all other nodes in a graph with non-negative edge weights. The algorithm initializes the distance to all cities as infinity, except the source node which is set to zero. It then iteratively selects the node with the smallest tentative distance using a priority queue (implemented via PriorityQueue<CityRoute>) and updates the distances of its neighbors if a shorter path is found.

Dijkstra’s algorithm guarantees an optimal solution in graphs with non-negative edge weights and operates with a time complexity of O((V + E) log V) when used in conjunction with a binary heap priority queue, where V is the number of vertices and E is the number of edges. In our system, the dijkstra() method returns both the shortest path and the total distance, supporting use cases such as routing and logistics optimization between any two cities.

By explicitly modeling the graph structure in Java and applying Dijkstra’s algorithm, the program can handle complex city networks with up to 10,000 routes across 300 cities, demonstrating both scalability and practical application of graph theory to real-world routing problems.

1. **Analysis/Design (UML Diagram(s))**
2. **Code Implementation (please add your TA -** [Furqan.rustam1@gmail.com](mailto:Furqan.rustam1@gmail.com) **– as a collaborator)**

GitHub (link):https://github.com/WolfClarence/Group9\_Assignment5

This is a new repository, please join as the collaborator.

This report presents a generic graph implementation in Java, designed to support both directed and undirected graphs. The class is built with flexibility and scalability in mind, ensuring that it can efficiently handle large datasets, such as those with over 100,000 nodes and 1,000,000 edges. The graph utilizes an adjacency list structure backed by a HashMap<String, List<Edge>>, where each node is mapped to a list of its outgoing edges. This approach offers efficient space utilization and enables fast lookups, insertions, and traversals, which is particularly beneficial for large, sparse graphs.

The graph provides the following core features:

addEdge(from, to, weight): Adds a weighted edge between two nodes. In undirected graphs, edges are added bidirectionally.

dijkstra(start, end): Computes the shortest path using Dijkstra's algorithm. It maintains a priority queue to always expand the node with the current shortest distance and uses a Map<String, Integer> for distance tracking and a Map<String, String> for reconstructing the path.

getShortestDistance(start, end): Returns the total distance of the shortest path computed by Dijkstra.

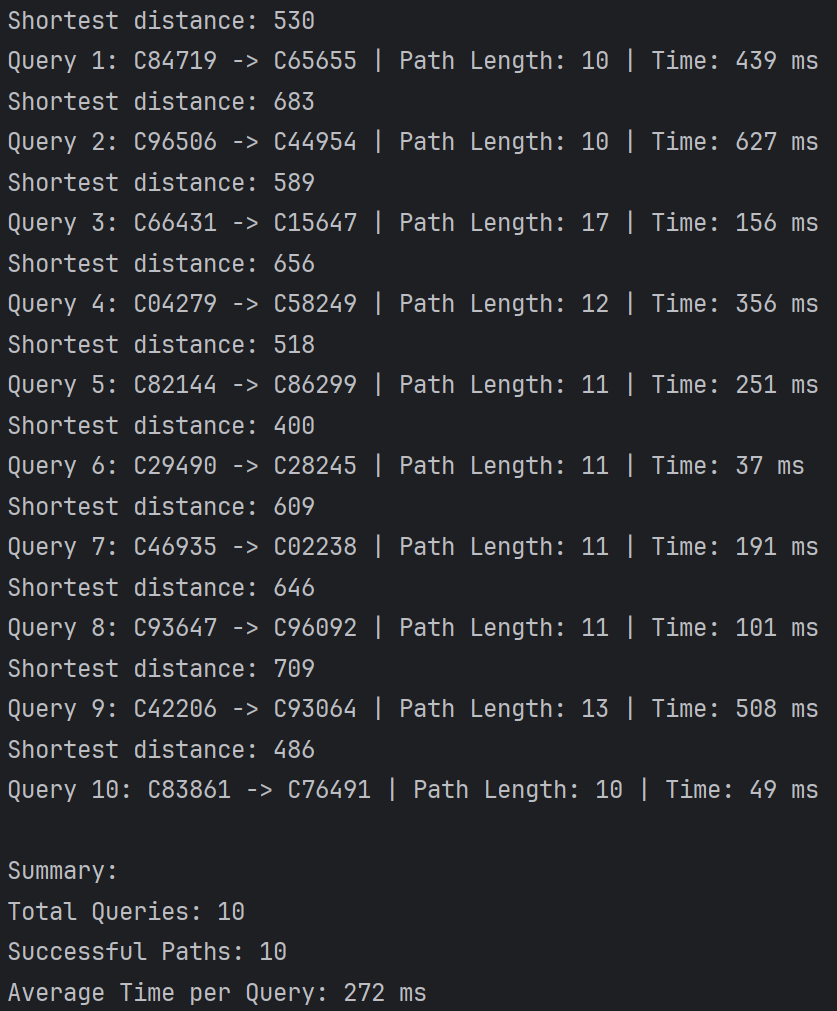
In terms of algorithm design, Dijkstra’s algorithm is implemented using a min-heap (PriorityQueue), ensuring efficient extraction of the node with the lowest tentative distance. The algorithm has a time complexity of O((V + E) log V), which is suitable for large graphs where the number of edges is much greater than the number of nodes.

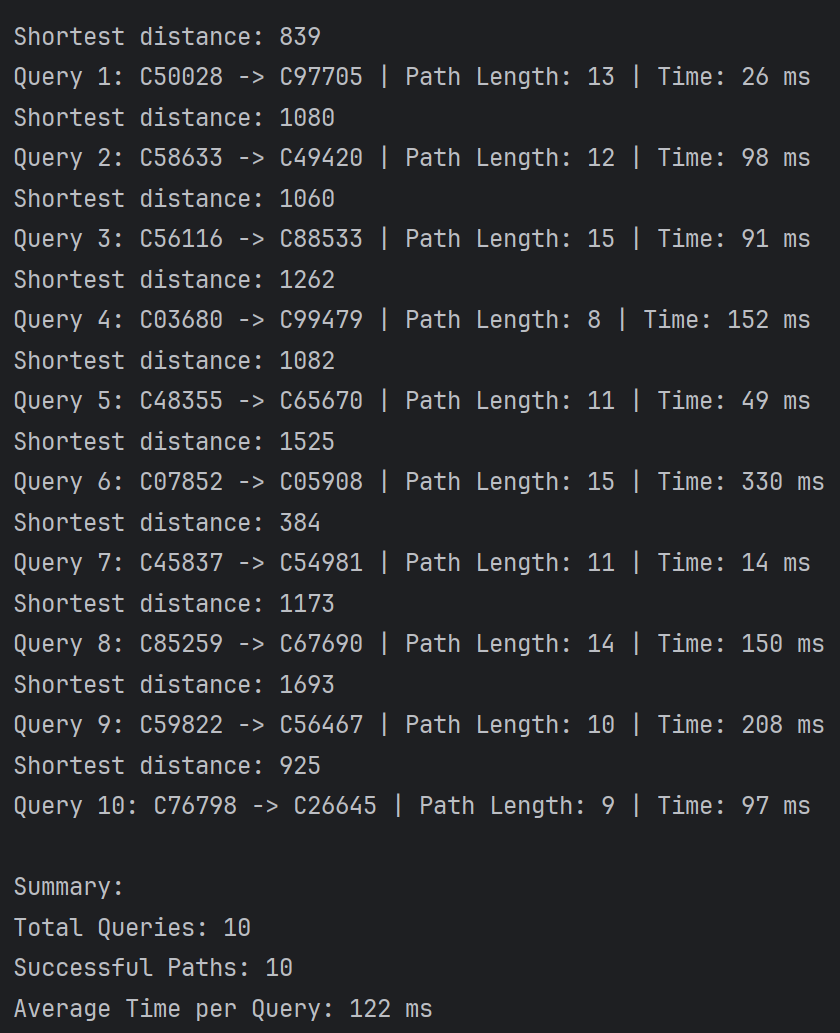
To assess the performance of this graph implementation, a test scenario was constructed with 100,000 randomly generated nodes and 1,000,000 edges. Ten random queries were executed to calculate the shortest path between pairs of randomly selected cities. This setup allowed us to compare the performance of directed versus undirected graphs. The undirected graph, with its bidirectional edges, often found shorter paths and had a higher success rate in pathfinding. On the other hand, the directed graph, where movement is restricted to one direction per edge, resulted in longer or even unreachable paths.

The observed difference in performance can be attributed to the structural properties of the graphs. In an undirected graph, bidirectional edges allow more flexibility, which generally results in faster convergence of Dijkstra’s algorithm. This leads to a shorter query time (122 ms on average) compared to the directed graph (272 ms on average). The directed graph’s edges, being one-way, limit the available paths, causing longer query times and lower success rates. Additionally, 10 queries were selected to strike a balance between statistical relevance and practical execution time. Each query's execution time was in the hundred-millisecond range, so 10 queries provided enough data for meaningful results without excessively prolonging the testing process.

In conclusion, this graph implementation demonstrates key principles of graph theory, such as adjacency lists and shortest path calculations. The comparison between directed and undirected graphs highlights the impact of graph structure on algorithm efficiency and runtime. This implementation is both functional and educational, providing insights into how different graph structures influence the behavior and performance of algorithms like Dijkstra’s.

Sample output:





1. **Video of the Implementation running**

youtube link:

Comments:

This is a brief overview of the code execution and result display. The results are shown in the console.

**Please save as pdf and submit on Brightspace**

**Students belonging to the same group** please **submit the same file .**