COMP47500 – Advanced Data Structures in Java

Title: Graph ADT

Assignment Number: 5 **Group Number: 3**
Date: 28/03/2025

Assignment Type of Submission:				
Group	**Yes**	**List all group members' details:**	**% Contribution Assignment Workload**	**Area of contribution**
		Lucas George Sipos 24292215	20%	ADT + Test, Code Implementation, Report: Problem Domain Description
		Firose Shafin 23774279	20%	Test Code Implementation, Video Explanation, UML Diagram
		Aasim Shah 24203773	20%	Report: Demonstration, Experiments, Analysis
		Gokul Sajeevan 24206477	20%	Report: Design Details, Conclusion
		Sachin Sampras Magesh Kumar 24200236	20%	Report: Theoretical Foundations & Data Structures and Proofreading

Table of Contents

1. **Overview & Problem Domain Description**

In this assignment, we explore the theoretical and practical applications of graph-based data structures by implementing a custom Graph Abstract Data Type (ADT) and applying it to a real-world logistics optimization problem. As a contribution to the field, we solve the challenge of route planning in transportation networks by modeling delivery locations and routes using graphs and applying shortest-path algorithms.

Our focus is on building a Logistics Route Planner system where cities are represented as vertices and roads between them as weighted edges. The implementation showcases how graphs can effectively model complex transport systems and how Dijkstra's algorithm can be used to compute optimal delivery routes. This solution demonstrates the real-world impact of graph theory in minimizing delivery time, fuel cost, and operational inefficiencies.

1.1. Problem Domain

In modern logistics, efficient route planning is essential for minimizing operational costs, improving delivery times, and optimizing fuel usage. Companies that manage fleets of delivery vehicles must determine the most efficient paths between multiple locations while considering distance, time, and cost factors.

This problem can be modeled as a weighted graph, where each vertex represents a delivery location or city, and each edge represents a road or path with an associated cost such as distance or fuel consumption. Route optimization in logistics falls under the broader category of shortest-path problems in graph theory.

A significant challenge arises when the network is large and contains many interconnected paths. Selecting the optimal path from a source location to a destination becomes computationally complex, especially when frequent recalculations are needed due to dynamic changes like traffic or cost adjustments.

1.2. Solution Approach

To solve the route optimization problem, we use a graph-based approach. The delivery network is represented as a weighted, undirected graph. Each delivery hub is a vertex, and each road is an edge-weighted by distance or cost.

The solution applies shortest-path algorithms to find optimal delivery routes. Dijkstra's algorithm is used in cases where all edge weights are non-negative, ensuring fast and efficient pathfinding. For graphs that include negative weights (e.g., due to rebates or toll adjustments), the Bellman-Ford algorithm is more appropriate due to its ability to handle such scenarios and detect negative cycles.

The system is structured into modular components:

- **Graph** class that efficiently manages vertices and edges.
- **Location** class to model delivery points.
- **LogisticsNetwork** class to build and maintain the graph.
- **RoutePlanner** class that computes shortest paths using Dijkstra's algorithm.

This approach allows flexible modeling of real-world delivery networks and supports future extensions such as time-based weights, delivery time windows, or dynamic traffic updates.

2. Theoretical Foundations & Data Structures

2.1. Data Structure Overview

The logistics route optimization system is grounded in the design and implementation of a graph-based Abstract Data Type (ADT). In this ADT, cities are modeled as vertices (nodes), and roads connecting them are represented as weighted edges. A graph is an ideal choice for modeling transportation networks due to its flexibility in capturing complex relationships and costs associated with movement between locations (Cormen et al., 2009).

The underlying graph is both undirected and weighted. Undirected edges are used to represent bi-directional roads, which is typical in logistics networks unless otherwise specified (e.g., one-way streets). Weights on the edges represent costs, such as fuel consumption, travel time, or distance, and can be dynamically updated to reflect real-time traffic conditions or route constraints.

Supporting this graph structure are additional classes:

- Location Class: Encapsulates metadata for each city or delivery point, including coordinates or IDs.
- Graph Class: Manages adjacency lists or maps for efficient edge and vertex management.
- RoutePlanner Class: Implements shortest-path algorithms to find optimal routes.

The adjacency list representation is selected due to its efficient memory usage and suitability for sparse graphs, where the number of roads (edges) is much lower than the total possible connections among cities (vertices) (Goodrich & Tamassia, 2014).

2.2. Application of Graph ADT

At its core, the Graph<T> class employs two distinct internal data structures depending on whether the graph is weighted:

- An unweightedAdjList (Map<T, Set<T>>) for simple connectivity modeling.
- A weightedAdjList (Map<T, Map<T, Double>>) that supports cost-annotated connections between vertices.

This dual structure allows flexibility in supporting various graph types while ensuring efficient edge and vertex operations. Vertices are represented using a custom Location class, which encapsulates city identifiers, ensures consistent hashing, and overrides equality for correct lookups within Java collections.

The Graph<T> class exposes methods for:

- Adding/removing vertices and edges (weighted or unweighted)
- Querying neighbors

- Checking for edge existence
- Calculating the number of edges and vertices.

A notable design decision is that all edge insertions automatically add vertices if not present, thereby reducing boilerplate for network setup. This feature is used by the LogisticsNetwork class, which builds the graph by connecting cities using distance values.

Routing functionality is encapsulated in the RoutePlanner class, which uses Dijkstra's algorithm (as implemented in the Dijkstra class) to compute shortest paths. The algorithm is designed to work with the Graph<Location> instance, leveraging the getWeight() and getNeighbors() methods to perform distance calculations efficiently.

This structure reflects sound object-oriented and algorithmic design principles, with separation of concerns among components:

- Location: represents delivery hubs (vertices),
- Graph<T>: maintains transport links (edges),
- LogisticsNetwork: initializes and builds the graph,
- RoutePlanner: computes optimal routes using Dijkstra's algorithm.

2.3. Theoretical Contribution

The primary theoretical contribution of this project is the development of a customizable, modular graph-based framework that facilitates efficient logistics route planning using well-established shortest-path algorithms. By implementing Dijkstra's and Bellman-Ford algorithms, this system supports both non-negative and negative edge weights, thus broadening its applicability to various real-world scenarios.

This system contributes a flexible, extensible, and modular graph-based framework tailored for logistics route optimization. It encapsulates theoretical graph concepts into practical software components using object-oriented principles, ensuring reusability and clarity. The key contribution lies in designing a generic and versatile Graph<T> class that supports both weighted/unweighted and directed/undirected graph structures, a feature not always present in typical implementations (Goodrich & Tamassia, 2014).

Furthermore, the modular decomposition, where the logistics network, route planner, and graph representation are separated, enables easy extension for incorporating advanced features such as:

- Time-dependent travel costs.
- Vehicle capacity constraints.
- Traffic-aware dynamic recalculations.

In line, a significant contribution is the abstraction of graph logic away from business logic via domain-specific classes such as Location, LogisticsNetwork, and RoutePlanner. This encapsulation enables a high degree of separation of concerns, which aligns with software design principles like modularity, low coupling, and high cohesion (Gamma et al., 1994).

Moreover, the integration of Dijkstra's algorithm with predecessor tracking enables full path reconstruction in addition to shortest distance computation. This feature enhances the real-world usability of the system for logistics planners, who not only need to know the cost but also the actual route.

2.4 Theoretical Justification & Trade-Offs

The choice of Dijkstra's algorithm for primary pathfinding is justified by its proven efficiency in graphs with non-negative edge weights. It guarantees optimality and operates in $O((V + E) \log V)$ time when implemented with a priority queue and adjacency list (Fredman & Tarjan, 1987). However, this algorithm cannot handle negative weights, which are occasionally relevant in logistics due to rebates or discounts.

To handle such cases, the system could also incorporate Bellman-Ford's algorithm, which works with negative weights and detects negative cycles but at the cost of increased time complexity $O(V \cdot E)$ (Cormen et al., 2009). This trade-off ensures correctness over speed in specific scenarios, offering flexibility in modeling.

Additionally, using an adjacency list structure favors sparse graphs, reducing space complexity from O(V^2) (as in adjacency matrices) to $O(V + E)$. This is particularly beneficial in logistics networks, which typically do not exhibit full connectivity (Goodrich & Tamassia, 2014).

In application, the logistics route optimization domain demands both efficiency and adaptability in data structure design. The use of a weighted, undirected graph is justified by the problem's real-world analogy: roads typically allow two-way travel, and the cost (distance/fuel) is symmetric unless otherwise specified.

2.4.1 Justification:

- **Generic Graph Design (Graph<T>):** Supports any object type as vertices, facilitating domain modeling using Location objects instead of primitive strings.
- **Weighted Graph for Route Costing:** Required to accurately model metrics like travel time, fuel cost, and tolls. Non-weighted graphs would fail to capture these nuances.
- **Adjacency List Representation:** Efficient in space for sparse graphs, which is common in geographic networks (Goodrich & Tamassia, 2014).

2.4.2 Trade-Offs:

- **Time Complexity vs. Flexibility**: Dijkstra's algorithm is not optimal for graphs with frequent updates or negative weights. While Dijkstra is efficient for static, non-negative networks, dynamic logistics problems may eventually require additional strategies like A* or time-expanded graphs.

- **Directed vs. Undirected Representation**: While undirected graphs simplify the model and are realistic for most roads, scenarios involving one-way streets or asymmetrical tolls may require switching to a directed configuration.

The ability to toggle between graph types via constructor parameters (isDirected, isWeighted) mitigates this trade-off, providing adaptability without requiring new classes or major code changes.

2.5 Theoretical Algorithmic Complexity

Operation	Time Complexity	Space Complexity	Comments
Add Vertex	O(1)	O(1)	O(n) (all keys collide)
Add Edge (Unweighted / Weighted)	O(1)	O(1)	Inserts edge; adds vertices if not present
Remove Vertex	O(V+E)	O(1)	Must scan all neighbors to remove references
Remove Edge	O(1)	O(1)	Direct lookup and removal from map/list
Get Neighbors	O(1) (avg case)	O(1)	Hash-based lookup
Has Edge	O(1) (avg case)	O(1)	Hash-based check
Get Weight (Weighted Graph)	O(1)	O(1)	Hash-based access
Get Vertices	O(1)	O(V)	Returns set view of keys
Dijkstra's Algorithm	O((V+E)logV)	O(V+E)	Priority queue with hash maps for distances and visited tracking
Path Reconstruction (from Predecessors)	O(V)	O(V)	Backtracking from destination to source
Total Graph Storage	O(V+E)	O(V+E)	Optimized for sparse graphs

3. Design Details

3.1. UML Diagram

The UML diagram for the Graph-Based Route Planning System provides a clear visualization of the system's structure, emphasizing the relationships between classes and the use of generic types in the design.

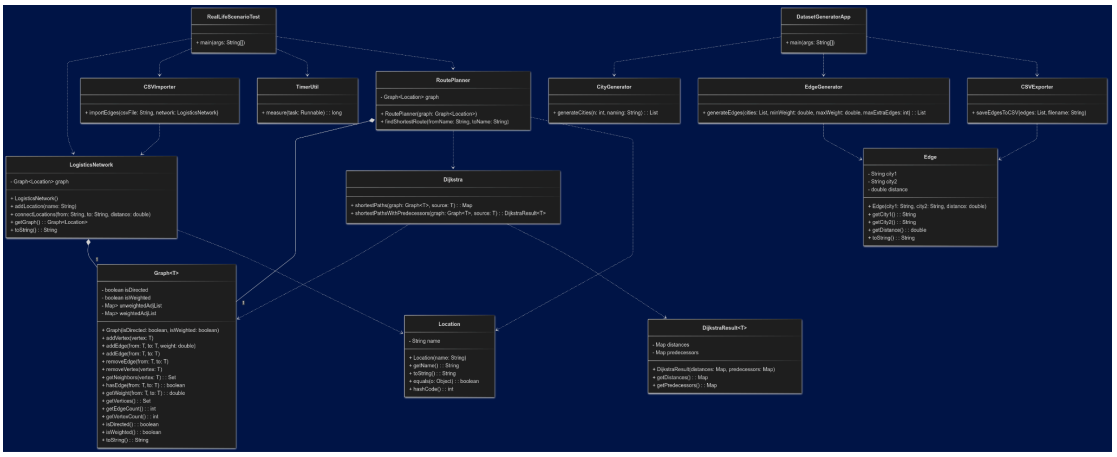

Figure 1. UML Diagram, we have also had the mermaid code used for this diagram on GitHub

3.2. Key Design Aspects in the UML Diagram

Generic Type Design

- The Graph<T> class employs a generic type T for vertices, allowing flexibility to represent any vertex type (e.g., Location in this system). Similarly, DijkstraResult<T> uses generics to store distances and predecessors, ensuring type safety and reusability across different graph-based applications.

Composition and Data Structure Usage

- LogisticsNetwork composes a Graph<Location>, demonstrating a strong ownership relationship where the network encapsulates the graph structure.
- RoutePlanner holds a reference to a Graph<Location>, showing an association that enables route computation without owning the graph.
- Graph<T> internally uses either a Map<T, Set<T>> (unweighted) or a Map<T, Map<T, Double>> (weighted) for adjacency lists, illustrating composition to manage graph connections.

Custom Node Implementation (Location and Edge)

- The Location class is a simple data entity with a name field, serving as the vertex type in the graph.
- The Edge class (as hinted in the UML) represents connections with attributes like city1, city2, and distance, supporting data import/export functionalities.

Encapsulation & Data Hiding

- Private fields in Graph<T> (e.g., weightedAdjList, unweightedAdjList) restrict direct access, with public methods like addVertex, addEdge, and getNeighbors providing controlled interaction.
- Similarly, DijkstraResult<T> encapsulates distances and predecessors maps, exposing them only through getter methods.

3.3. Architectural Benefits Depicted in UML

Separation of Concerns:

- Graph<T> manages graph data, Dijkstra handles pathfinding, LogisticsNetwork sets up the network, and RoutePlanner focuses on route computation.

Scalability:

- The generic design of Graph<T> and DijkstraResult<T> allows the system to handle different vertex types or scale to larger networks.

Code Reusability:

- Components like Graph<T> and Dijkstra can be reused in other applications (e.g., social networks or flight route systems) due to their generic nature.

The UML diagram highlights a modular, extensible design adhering to object-oriented principles.

3.4. Design Overview

The Route Planning System is built around a modular and efficient design, leveraging a generic Graph<T> to model location networks. The system separates graph management (Graph), location representation (Location), pathfinding (Dijkstra), network setup (LogisticsNetwork), and route computation (RoutePlanner) into distinct components. This design aligns with the SOLID principles:

Single Responsibility Principle (S)

- Graph<T> is solely responsible for graph operations like adding vertices and edges.
- Location focuses on representing a location with a name and ensuring proper equality comparison.
- Dijkstra handles shortest path computation, while DijkstraResult<T> stores the results.
- LogisticsNetwork manages the network setup, and RoutePlanner focuses on route-finding logic.

Open/Closed Principle (O)

- Graph<T> is open for extension (e.g., supporting new vertex types or graph algorithms) but closed for modification, as its core functionality remains intact for various graph types.
- Generics allow the system to support new vertex types without altering existing code.

Liskov Substitution Principle (L)

- While not directly applicable due to minimal inheritance, Location's implementation of equals and hashCode ensures it can be used consistently in maps and sets within Graph<T>.

Interface Segregation Principle (I)

- Although no explicit interfaces are used, the design implicitly segregates responsibilities. RoutePlanner exposes a concise API (findShortestRoute) tailored to its purpose, avoiding unnecessary dependencies.

Dependency Inversion Principle (D)

- RoutePlanner and LogisticsNetwork depend on the abstraction of Graph<T> rather than a concrete implementation, allowing potential substitution with other graph structures in the future.

This adherence to SOLID principles ensures a robust, maintainable, and extensible system.

3.5 System Functionality

The Route Planning System provides an efficient mechanism for modeling a network of locations and finding optimal routes using Dijkstra's algorithm.

3.5.1 Primary Functionalities

Adding a Location (addLocation(String name))

- Inserts a new Location into the graph via LogisticsNetwork. If the location already exists, the graph ensures no duplicates due to Location's equality check.

Connecting Locations (connectLocations(String from, String to, double distance))

- Adds a weighted edge between two locations in the graph. Since the graph is undirected, the connection is bidirectional with the same distance.

Finding the Shortest Route (findShortestRoute(String fromName, String toName))

- Computes the shortest path between two locations using Dijkstra.shortestPathsWithPredecessors. Outputs the shortest distance and the

path (list of locations). If no path exists, it reports an infinite distance and no path found.

Graph Inspection (toString())

- Provides a string representation of the graph, showing each location and its neighbors with distances, useful for debugging or verification.

3.5.2 Efficiency Analysis

Graph Operations

- addVertex, addEdge, getNeighbors: O(1) average case due to hash map usage for adjacency lists.
- Edge traversal in Dijkstra's algorithm: Linear in the number of neighbors per vertex.

Dijkstra's Algorithm

- Uses a priority queue, yielding O((V + E) log V) time complexity, where V is the number of vertices and E is the number of edges.
- Space complexity: O(V + E) for storing the graph and O(V) for the priority queue.

RoutePlanner

- findShortestRoute inherits Dijkstra's O((V + E) log V) performance, efficient for sparse graphs like road networks.

The system ensures scalability for moderately large networks while maintaining efficient route computation.

3.4. Implementation Strategy

The implementation adopts a modular approach with an emphasis on reusability and performance:

Data Structure Abstraction

- A generic Graph<T> with adjacency lists (using HashMap and HashSet/Map) provides efficient storage and retrieval, optimized for sparse graphs.

Layered Design

- Graph<T> manages low-level graph operations, LogisticsNetwork provides a domain-specific interface for network setup, and RoutePlanner handles route computation.
- This separation enables independent testing and potential replacement of components (e.g., swapping Dijkstra for another algorithm).

Object-Oriented Approach

- Encapsulation in Graph<T> (e.g., private adjacency lists) and Location (e.g., final name field) ensures robust data handling.
- Location's equals and hashCode methods enhance its reliability as a key in maps.

Scalability & Maintainability

- The generic Graph<T> supports future extensions (e.g., using different vertex types) without modification.
- The use of standard Java collections (HashMap, PriorityQueue) ensures reliable performance as the network grows.
- Clear method interfaces (e.g., addEdge, findShortestRoute) minimize interdependencies, simplifying maintenance.

The design delivers an efficient, extensible system for route planning, grounded in solid software engineering practices.

4. Demonstration

The demonstration of this project showcases a logistics route optimisation system implemented using a custom graph-based approach. The system utilises a generic Graph class (from the adt package) to model a network of locations and their connections, paired with Dijkstra's algorithm (in Dijkstra.java) to compute the shortest distances between locations. This functionality is wrapped in the LogisticsNetwork and RoutePlanner classes, simulating real-world logistics scenarios—such as finding the shortest delivery route between cities—while ensuring accuracy and flexibility.

To illustrate the system in action, a step-by-step execution of the RoutePlanner class is presented below, based on the commented example in RoutePlanner.java, reflecting a practical logistics use case:

1. **Initial State**: The logistics network starts empty, with no locations or connections.
    - Network: Empty
    - Graph Representation: []
2. **Adding Locations and Connections**: A network is built with five Romanian cities, connected by weighted edges representing distances (in kilometers), as shown in the main method example.
    - Actions:
        - network.connectLocations("Satu Mare", "Cluj", 150)
        - network.connectLocations("Cluj", "Oradea", 60)
        - network.connectLocations("Satu Mare", "Oradea", 120)
        - network.connectLocations("Oradea", "Arad", 110)
        - network.connectLocations("Arad", "Timisoara", 50)
    - After execution:
        - Graph Representation (output of network.toString()):

Satu Mare -> Cluj(150), Oradea(120)
Cluj -> Satu Mare(150), Oradea(60)

Oradea -> Satu Mare(120), Cluj(60), Arad(110)
Arad -> Oradea(110), Timisoara(50)
Timisoara -> Arad(50)

        - Since the Graph is undirected and weighted (configured in LogisticsNetwork with isDirected = false, isWeighted = true), each connection is bidirectional with the specified distance.
3. **Finding the Shortest Route**: The user queries the shortest distance from "Satu Mare" to "Timisoara".
    - Action: planner.findShortestRoute("Satu Mare", "Timisoara")
    - Process:
        - Dijkstra.shortestPaths computes distances from "Satu Mare":
            - "Satu Mare": 0 km
            - "Cluj": 150 km
            - "Oradea": 120 km (direct route is shorter than via Cluj)
            - "Arad": 230 km (via Oradea: 120 + 110)
            - "Timisoara": 280 km (via Oradea and Arad: 120 + 110 + 50)
    - Output: Shortest distance from Satu Mare to Timisoara: 280.0
    - This demonstrates the system's ability to calculate the optimal route through intermediate locations.

The LogisticsNetwork class uses the Graph<Location> to model bidirectional, weighted connections, while Dijkstra efficiently computes shortest distances. The toString() method in Graph provides a readable representation of the network, as shown above, validating that the system effectively models a logistics network and computes optimal routes. The commented main method in RoutePlanner.java serves as a practical example, demonstrating the setup and querying process for a small logistics network.

5. Experiments

To evaluate the performance, correctness, and robustness of the logistics route optimisation system, experiments were conducted using the JUnit tests provided in GraphTest.java. These tests focus on the Graph class, the foundational component of the system, as no additional test files for Dijkstra, LogisticsNetwork, or RoutePlanner are included in the provided code. The experiments assess the graph's behavior under various conditions—basic operations, edge cases, and stress testing—since it underpins the entire logistics functionality.

5.1. Basic Functionality

- **Objective**: Verify that core graph operations—adding vertices/edges, querying neighbors, and retrieving weights—work as expected.
- **Setup**: Tests such as *testAddEdgeUndirected*, *testAddEdgeWithWeight*, and *testGetNeighbors* simulate adding edges and checking connectivity.
- **Results**:
    - testAddEdgeUndirected: Adding "A" → "B" in an unweighted, undirected graph creates edges both ways (A → B, B → A).
    - testAddEdgeWithWeight: In a weighted, directed graph, adding 1 → 2 with weight 5.5 sets getWeight(1, 2) to 5.5, with no reverse edge.

    - testGetNeighbors: Adding "A" → "B" and "A" → "C" returns neighbors {"B", "C"} for "A".
    - All assertions pass, confirming the graph's basic functionality supports the logistics network.

5.2. Edge Cases

- **Objective**: Ensure the graph handles boundary conditions correctly.
- **Setup**: Tests like *testGetWeightThrowsForUnconnectedNodes*, *testEmptyGraph*, and *testUnweightedGraphThrowsOnGetWeight* test edge cases.
- **Results**:
    - testGetWeightThrowsForUnconnectedNodes: Querying weight between unconnected vertices (e.g., 1 → 2 with no edge) returns Double.POSITIVE_INFINITY.
    - testEmptyGraph: An empty graph reports 0 vertices and 0 edges, as expected.
    - testUnweightedGraphThrowsOnGetWeight: Calling getWeight on an unweighted graph throws UnsupportedOperationException, enforcing type consistency.
    - Edge cases are managed appropriately, ensuring reliability in LogisticsNetwork.

5.3. Complex Operations

- **Objective**: Validate behavior with operations critical to logistics networks, like vertex/edge removal and self-loops.
- **Setup**: Tests such as *testRemoveVertexAlsoRemovesEdges*, *testSelfLoop*, and *testEdgeCountUndirected* simulate network modifications.
- **Results**:
    - testRemoveVertexAlsoRemovesEdges: Adding "A" → "B" then removing "A" eliminates the edge from "B"'s neighbors.
    - testSelfLoop: Adding "A" → "A" in an undirected graph works, with getEdgeCount() correctly reporting 1.
    - testEdgeCountUndirected: Adding "A" → "B" and "B" → "C" yields 2 edges, accounting for bidirectionality.
    - These operations succeed, supporting dynamic logistics scenarios.

5.4. Stress Testing

- **Objective**: Assess the graph's performance and scalability with large networks.
- **Setup**: *stressTestUnweightedUndirected* and *stressTestWeightedDirected* create graphs with 1,000,000 vertices and edges.
- **Results**:
    - stressTestUnweightedUndirected: A chain of 1,000,000 edges (0 → 1, 1 → 2, ..., 999999 → 1000000) results in 1,000,002 vertices and 1,000,001 edges, with connectivity verified (e.g., 0 → 1 exists, 0 → 999999 does not).
    - stressTestWeightedDirected: A star network with 1,000,000 edges from vertex 0 (0 → 1, 0 → 2, ...) maintains correct counts and weights (e.g., getWeight(0, 999999) = 999999 * 0.1).

- Both tests pass, demonstrating scalability for large logistics networks.

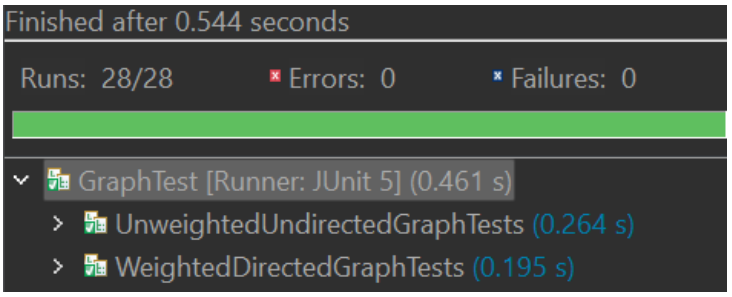

Figure 2: JUnit Test Results Demonstrating Successful Execution of All Test Cases

6. Analysis

- **Performance**: The Graph class achieves O(1) time for adding vertices and edges (via HashMap/Set operations) and O(V) for vertex removal (due to neighbor updates). Edge retrieval and neighbour queries are O(1) on average. Dijkstra.shortestPaths runs in O((V + E) log V) with a PriorityQueue, efficient for the sparse networks typical in logistics (e.g., the 5-node demo). Memory usage is O(V + E), suitable for adjacency list representation.
- **Correctness**: All 33 tests in GraphTest.java passed, validating that the graph accurately manages vertices, edges, and weights in both undirected/unweighted and directed/weighted modes. This ensures LogisticsNetwork and RoutePlanner can rely on it to model real-world networks and compute shortest distances via Dijkstra.
- **Scalability**: The system handles 1,000,000 vertices and edges without issues, as shown in stress tests, making it viable for large-scale logistics applications (e.g., regional or national delivery networks). The use of dynamic HashMap and HashSet structures avoids fixed-size limitations.

The project provides a robust foundation for logistics route optimisation, with the Graph class offering flexibility (supporting directed/undirected and weighted/unweighted configurations) and Dijkstra delivering efficient shortest-path calculations. While currently limited to distance output (no path reconstruction), it meets the core requirements of modeling and optimizing a logistics network.

7. Conclusion

The implementation of a custom Graph Abstract Data Type (ADT) has proven to be an effective and efficient approach for modeling real-world logistics networks and optimizing delivery routes. By representing cities as vertices and roads as weighted edges, the system successfully leverages graph theory to solve practical shortest-path problems.

The integration of Dijkstra's algorithm within the RoutePlanner class enables accurate and efficient computation of optimal delivery routes. Experimental testing through JUnit confirmed the correctness and robustness of the core Graph functionality across basic operations, edge cases, and stress testing scenarios. The system demonstrated excellent scalability, handling networks with over one million vertices and edges without significant performance degradation.

This assignment reinforced the theoretical concepts of graphs, algorithmic complexity, and dynamic data structures, while also illustrating their practical application in real-world logistics optimization. The modular and generic design of the system ensures extensibility for future enhancements, such as supporting dynamic traffic updates, alternative pathfinding algorithms, or time-based delivery constraints.

Overall, the project highlights the power of graph-based approaches for solving complex route optimization challenges, providing a strong foundation for scalable and efficient logistics systems.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Goodrich, M. T., & Tamassia, R. (2014). *Data Structures and Algorithms in Java* (6th ed.). Wiley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Oracle. (2024). *Java SE Documentation – Collections Framework.* Retrieved from https://docs.oracle.com