**Assignment 5**

**COMP47500 Adv. Data Structures in Java**

**Campus Navigation System: Graph-Based Pathfinding Using Dijkstra’s Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment Type of Submission:** |  |  |  |
| **Group** | Yes | **List all group members’ details:**  Anuj Lalitkumar Lamba  24202506  Harshal Chandrakant More  24200354  Khush Sushil Poddar  24200202  Omkar Sanwatsarkar  24202103  Shikhar Singh Negi  24203095 | **% Contribution**  **Assignment Workload**  20%  20%  20%  20%  20% |

**A logo of a university

AI-generated content may be incorrect.**

**1. Problem Domain Description**

Modern university campuses often cover a large geographic area with dozens of interconnected buildings such as lecture halls, libraries, labs, administrative blocks, and cafeterias. For students, faculty, staff, and visitors, navigating this space efficiently is essential. A common challenge is determining the shortest or fastest route from one building to another — especially under time constraints, such as getting to class, meetings, or events on time.

To solve this, we developed a **Campus Navigation System** that uses graphs to model the campus layout and implements **Dijkstra’s Algorithm** to find the shortest path between two locations. Each **location** is represented as a **node**, and each **walkable path** between locations is represented as a **weighted edge**, where the weight denotes distance or estimated time.

This problem is practical and closely aligned with how pathfinding is handled in real-world applications such as:

* Google Maps or Apple Maps (for outdoor and indoor navigation)
* Transportation and logistics systems
* Emergency evacuation planning
* Campus navigation apps for new students or visitors

By using a graph-based approach, we ensure that the system can handle complex layouts, provide accurate routing, and support future extensions like blocked paths, accessibility preferences, or alternate routes.

This project not only addresses a real need but also demonstrates how fundamental computer science concepts specifically graph data structures and algorithms, can be applied to build efficient, interactive, and scalable solutions.

## 2. Theoretical Foundations of the Data Structure.

At the core of the Campus Navigation System lies the **graph data structure**, a fundamental abstraction in computer science used to model relationships between pairs of objects. In our case, **locations on campus are modeled as vertices (nodes)** and the **connections between them as edges**, forming an **undirected, weighted graph**.

### **2.1 Graphs: Structure and Purpose**

A graph consists of a set of vertices V and edges E. Each edge may carry a weight, often representing cost, distance, or time. Our implementation uses an **adjacency list representation**, where each location maintains a list of neighboring locations and the weights of the paths that connect them. This approach is memory-efficient and suitable for **sparse graphs**, which is typical of most real-world maps (i.e., not every building is directly connected to every other building).

This graph structure enables us to:

* Model flexible campus layouts
* Add/remove locations or paths dynamically
* Attach custom data to each node or edge (e.g., accessibility flags)

### **2.2 Dijkstra’s Algorithm: Pathfinding Logic**

To compute the shortest path between two locations, we implemented **Dijkstra’s Algorithm**, a classic **greedy algorithm** that solves the **single-source shortest path problem** for graphs with non-negative edge weights.

#### Algorithm Overview:

1. Assign a tentative distance value to every node (0 for the start node, ∞ for all others).
2. Set the starting node as current. Mark all others unvisited.
3. For the current node, consider all its neighbors and calculate tentative distances.
4. Once all neighbors are considered, mark the current node as visited.
5. Repeat the process for the unvisited node with the smallest tentative distance.
6. Stop when the destination node is marked visited.

#### Time Complexity:

* with a min-priority queue (as used in this implementation).

### **2.3 Theoretical Significance**

Using graphs and Dijkstra’s algorithm in a campus navigation setting demonstrates a strong alignment between **theory and real-world application**. The graph abstraction allows us to:

* Efficiently represent complex map structures
* Support dynamic routing
* Scale with increasing nodes (campus expansion)

Meanwhile, Dijkstra’s guarantees that the **optimal path** is found using a deterministic, performance-efficient method.

This foundation provides a solid platform not only for campus navigation but also for potential extensions like:

* Real-time traffic updates
* Accessibility-aware pathfinding
* Obstacle avoidance (e.g., construction zones)

## 3. Design Overview

The Campus Navigation System is built using an **object-oriented design** that emphasizes modularity, scalability, and ease of testing. Each component of the system has a clearly defined responsibility, making the system intuitive to understand and maintain.

### **3.1 Core Design Principles**

* **Modularity**: Each class handles a specific part of the system — locations, paths, graph logic, algorithms, user interaction, and test automation are cleanly separated.
* **Abstraction**: Core logic (e.g., graph and pathfinding) is decoupled from the user interface (console), allowing future GUI or web integrations.
* **Reusability**: The same graph structure and pathfinding logic can be used for other map-based applications (e.g., logistics routing or indoor navigation).
* **Extensibility**: Additional features such as alternate route suggestions, obstacle marking, or time-based path updates can be easily added.

| **Component** | **Description** |
| --- | --- |
| Location | Represents a single building or point on campus. Each location is treated as a node in the graph. |
| Path | Represents a connection between two locations, with a weight indicating the distance. |
| Graph | Manages the adjacency list, supporting dynamic addition of locations and paths. |
| Dijkstra | Implements Dijkstra’s shortest path algorithm to calculate optimal routes. |
| CampusNavigationApp | Acts as the main application driver. Handles user input, interaction logic, and connects all components. |
| TestCaseLoader | Contains 12 predefined test cases and file-based logging for automated verification of correctness and performance. |

### **3.3 User Interaction Flow**

1. **Startup**: The user is presented with a console menu.
2. **Setup**: Users can add buildings and paths manually or use preloaded sample data.
3. **Query**: Users input a starting and ending location.
4. **Output**: The system computes and displays the shortest path with the total distance.
5. **Testing**: Users can run automated tests and export the results to a text log.

### **3.4 Sample Use Case**

A student wants to get from the **Library** to the **Lecture Hall**. After launching the system:

* They choose to load the default map (preloaded in ()).
* Then select the shortest route option and input "Library" as the start and "Lecture Hall" as the end.
* The system returns:

A screenshot of a computer

AI-generated content may be incorrect.**UML diagram**

**4. Code & Implementation:**

**GitHub Link:**

**Video Link:**

Contributors:

Khush Shushil Poddar

Shikhar Singh Negi

Harshal Chandrakant More

Anuj Lalitkumar Lamba

Omkar Satwanskar

## Set of Experiments and Results

To validate the correctness, performance, and flexibility of the Campus Navigation System, we conducted a comprehensive set of experiments. These experiments were designed to test both typical and edge-case scenarios in campus navigation, using a variety of graph structures and pathfinding conditions.

### **🔹 5.1 Experimental Setup**

* **Testing Tool:** TestCaseLoader.java automates 12 predefined scenarios.
* **Input Types:** Graph nodes (locations), weighted edges (distances), start and end nodes.
* **Output:** Path printed to the console and saved to test\_log.txt file.
* **Evaluation Criteria:**
  + Correctness of path
  + Optimality of total distance
  + Handling of disconnected or edge cases
  + Algorithm stability and speed for larger graphs

5.2 Sample Test Cases

| **Test Case** | **Description** | **Expected Output** |
| --- | --- | --- |
| 1 | Direct path (A → B) | Path: A → B, Distance: 5 |
| 2 | Two-step path | Path: A → B → C, Distance: 7 |
| 3 | No connection between nodes | “No path found” |
| 4 | Multiple equal-cost paths | One of: A → B → D or A → C → D, Distance: 4 |
| 5 | Circular graph | Path: A → C, Distance: 1 |
| 6 | Long chain (A to F) | Path of 6 steps, Distance: 5 |
| 7 | Shortcut with lower cost | Chooses A → C → B instead of A → B directly |
| 8 | Disconnected clusters | “No path found” |
| 9 | Star-shaped graph | Correct central routing via node X |
| 10 | Avoiding high-cost edge | Takes cheaper 2-step path |
| 11 | Same start and end node | “Already at destination” |
| 12 | Large graph (15 nodes) | Distance: 14, tested performance |

### 5.3 Manual Console Testing

In addition to the automated test suite, we manually verified the system’s behavior through the interactive console interface. This allowed us to validate usability, correctness, and edge case handling from the perspective of an end user.

The manual testing process followed a structured workflow:

* **Step 1: Adding Custom Buildings**  
  Using the “Add Location” menu option, we input named buildings such as "Library", "Cafeteria", and "Lecture Hall" into the system. Each was internally stored as a Location node in the graph.
* **Step 2: Defining Campus Paths**  
  We then used the “Add Path” function to define realistic, weighted edges between those buildings, such as "Library" → "Cafeteria" (distance: 4)" and "Cafeteria" → "Lecture Hall" (distance: 6)". These were recorded as Path instances.
* **Step 3: Querying Navigation Paths**  
  By selecting the “Find Shortest Route” option and entering a start and destination location, the system invoked Dijkstra’s algorithm and displayed the shortest path along with the total distance. For example:
* **Step 4: Verifying Results**  
  We manually calculated expected outcomes and verified that the output path and distance matched. Additionally, attempts to route to nonexistent or disconnected nodes triggered appropriate error messages (e.g., “No path found”).

These tests demonstrated that the application was intuitive, functionally accurate, and robust against user input errors.

### **5.4 Performance Evaluation**

The performance of the system was evaluated based on responsiveness and algorithm efficiency across varying graph sizes:

* **Small Graphs (3–5 nodes):**  
  Pathfinding results were computed instantaneously (< 1 millisecond), even with multiple route options. These scenarios simulate a local building cluster, such as three adjacent lecture halls.
* **Moderate Graphs (10–15 nodes, linear topology):**  
  When testing a longer, linear graph simulating a full campus layout with 15 buildings, the system consistently returned results in under 5 milliseconds. The underlying use of a priority queue in Dijkstra’s algorithm ensures efficient performance, even as the graph size grows.
* **Disconnected Graphs & Edge Handling:**  
  Attempts to route between isolated clusters or self-loops were handled gracefully. The system:
  + Correctly reported “No path found” for disconnected nodes.
  + Returned a single node with zero distance when the start and destination were the same.
  + Never crashed or produced incorrect paths under stress conditions.

### **5.5 Key Observations**

The following insights emerged from both automated and manual testing:

* **Algorithm Reliability:**  
  Dijkstra’s algorithm proved to be a dependable choice for weighted, undirected campus graphs. It consistently returned the optimal route in all scenarios where a valid path existed.
* **Code Modularity:**  
  The clear separation of concerns between Graph, Path, Dijkstra, and the UI logic in CampusNavigationApp allowed for isolated testing, easy debugging, and future extensibility (e.g., replacing Dijkstra with A\*).
* **Logging & Testability:**  
  File-based logging in test\_log.txt enabled seamless documentation of all automated tests. This not only supported repeatability and transparency but also made the results suitable for inclusion in the final report or presentation.
* **User Experience:**  
  The console interface, while basic, was effective in guiding users through graph creation and navigation. Future enhancements could include a GUI or voice-based interface, but the core interaction model was solid.

## Conclusion

The Campus Navigation System successfully demonstrates how graph data structures and Dijkstra’s algorithm can be applied to a real-world problem: finding the shortest path between locations on a university campus. By modeling buildings as nodes and paths as weighted edges, the system provides fast and accurate navigation with a user-friendly console interface.

The code is modular and maintainable, with clear separation between data structures, algorithms, user interaction, and testing. Both manual and automated tests confirmed that the system performs reliably, handles edge cases correctly, and scales well with graph size.

Overall, this project bridges theoretical computer science and practical application, showing how algorithms like Dijkstra can be used to solve meaningful problems efficiently and effectively.