DSA II

ITBNM-2110-0062

N.A.S.R Dayananda

Contents

DSA II 1
1.0 Introduction 3
1.1Project Overview 3
1.2Problem Scenario 3
1.3Implementation Approach 3
2.0 Graph Design 3
2.1 Network Structure 3
2.2 Connectivity 3
2.3 Graph Representation 4
3.0 Implementation 5
3.1 Choice of Language and Tools 5
3.2 Data Structures 5
3.3 Algorithm Logic (Dijkstra's Algorithm) 5
3.4 Error Handling 6
4.0 Testing & Results 6
4.1 Test Environment 6
4.2 Test Case 1: 6
4.3 Test Case 2: 7
5.0 Conclusion 7
6.0 Appendix 8

1.0 Introduction

1.1Project Overview

The goal of this project is to develop a route optimization tool for a local delivery service. In the logistics industry, finding the most efficient path is crucial to minimize fuel consumption and reduce delivery times. Therefore, this project aims to calculate the shortest distance from a central warehouse to various customer locations using algorithmic techniques.

1.2Problem Scenario

Real-world delivery networks are not uniform; distances between locations vary significantly. To represent this, I modeled the delivery area as a weighted graph connecting six key locations, including a Warehouse, City Center, and Suburban districts. The challenge is to navigate this network efficiently, ensuring that the calculated route is always the shortest possible path based on road distance in kilometers.

1.3Implementation Approach

To address this challenge, I selected Dijkstra's Algorithm, as it is the standard and most reliable method for finding shortest paths in graphs with positive weights. The application is implemented using the Java programming language. It operates as a command-line interface (CLI) tool where the user can specify a starting point, and the system outputs the optimal route and total distance to all other destinations in the network.

2.0 Graph Design

2.1 Network Structure

For the purpose of this assignment, I designed a weighted, undirected graph that represents a delivery zone covering a generic city layout. The graph consists of 6 nodes (intersections/locations) and 8 edges (roads).

Nodes: I assigned meaningful names to each node to simulate a real-world scenario. "Node A" serves as the Warehouse (Starting Point), while other nodes represent delivery destinations like the "City Center" and "Suburban" areas.

Edges & Weights: The connections between nodes represent the roads. I assigned weights to these edges to represent the distance in kilometers. For example, the distance from the Warehouse (A) to the North District (D) is set to 7km.

2.2 Connectivity

A key requirement for this design was connectivity. I ensured that the graph is fully connected, meaning there is at least one path from the Warehouse to every other node in the system. There are no isolated locations, ensuring that the delivery driver can reach any destination within the network.

2.3 Graph Representation

Below is the breakdown of the locations used in the design:

A: Warehouse (Source)

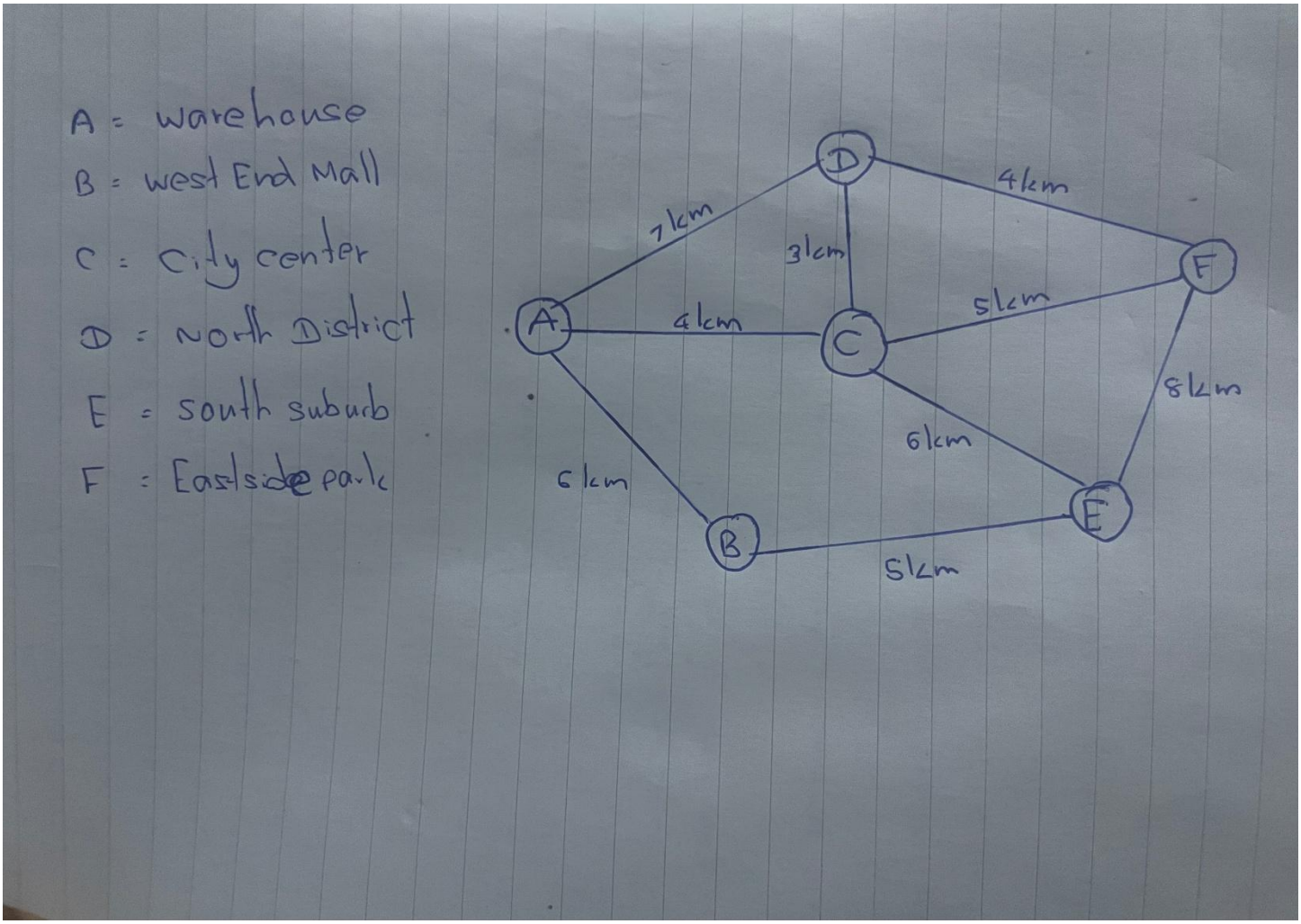
B: West End Mall

C: City Center

D: North District

E: South Suburb

F: Eastside Park



3.0 Implementation

3.1 Choice of Language and Tools

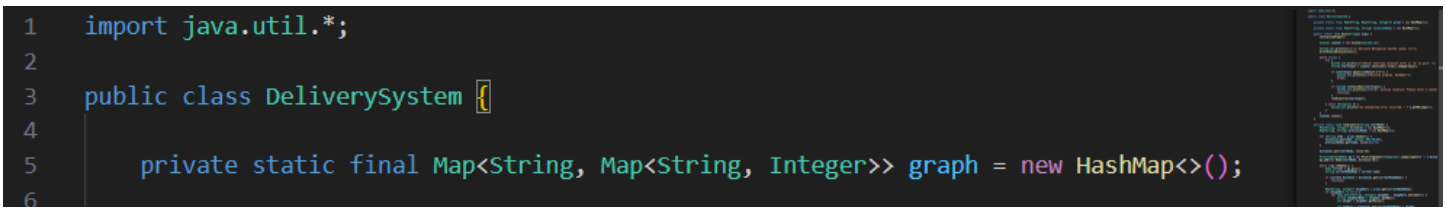
For the development of this delivery system, I chose Java as the programming language. The main reason for this selection is Java's robust "Collections Framework," which provides built-in data structures like Maps and Queues that are essential for implementing graph algorithms efficiently. I used the Scanner class to handle user inputs via the command line interface (CLI).

3.2 Data Structures

Used To represent the graph (the delivery network) in the code, I avoided using a complex matrix. Instead, I used an Adjacency List implementation.

HashMap: I used a HashMap<String, Map<String, Integer>> to store the graph. This allows the program to look up any location (Node) and immediately see its neighbors and the distances to them.

PriorityQueue: This was used to implement Dijkstra's algorithm. It helps in selecting the node with the smallest tentative distance efficiently, which significantly speeds up the search process.



```
1  import java.util.*;
2
3  public class DeliverySystem {
4
5      private static final Map<String, Map<String, Integer>> graph = new HashMap<>();
6
```

3.3 Algorithm Logic (Dijkstra's Algorithm)

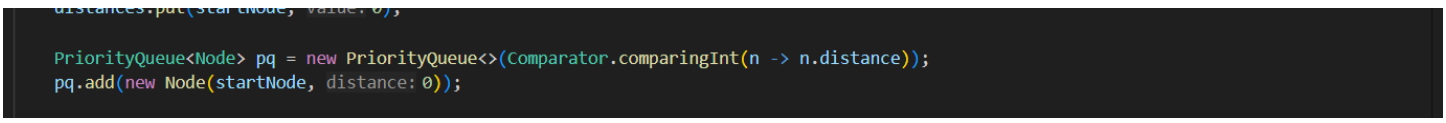
The core functionality relies on Dijkstra's Algorithm. Here is how I implemented the logic:

Initialization: The program sets the distance to the starting node as 0 and all other nodes to Infinity.

Exploration: It uses the Priority Queue to visit the nearest unvisited node.

Relaxation: For each neighbor, the program calculates if the new path is shorter than the currently known path. If it is, the distance is updated.

Backtracking: To display the path (e.g., A -> C -> D), I maintained a previousNodes map. After reaching the destination, the code traces back from the end node to the start node to reconstruct the route.



```
PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.distance));
pq.add(new Node(startNode, distance: 0));
```

3.4 Error Handling

To ensure the system is user-friendly and doesn't crash easily, I implemented several error-handling mechanisms:

Input Validation: The program checks if the user's input matches a valid node key (A-F). If the user enters an invalid letter, a custom error message is displayed.

Runtime Safety: The main logic is wrapped in try-catch blocks to handle any unexpected runtime exceptions gracefully.

4.0 Testing & Results

4.1 Test Environment

The implementation was tested on a Windows environment using the standard Command Prompt (CMD). I compiled and ran the DeliverySystem.java file to verify its functionality. The main goal of testing was to ensure that the Dijkstra algorithm correctly calculates the shortest distances and that the system handles invalid inputs without crashing.

4.2 Test Case 1:

Valid Input (Normal Operation) For the first test, I entered "A" (Warehouse) as the starting location. As expected, the system successfully calculated the shortest paths to all other destinations.

Input: A

Observation: The program listed B, C, D, E, and F with their correct cumulative distances and the exact path sequence. For example, the path to the "South Suburb (E)" was correctly identified as A -> B -> E with a total distance of 11km.

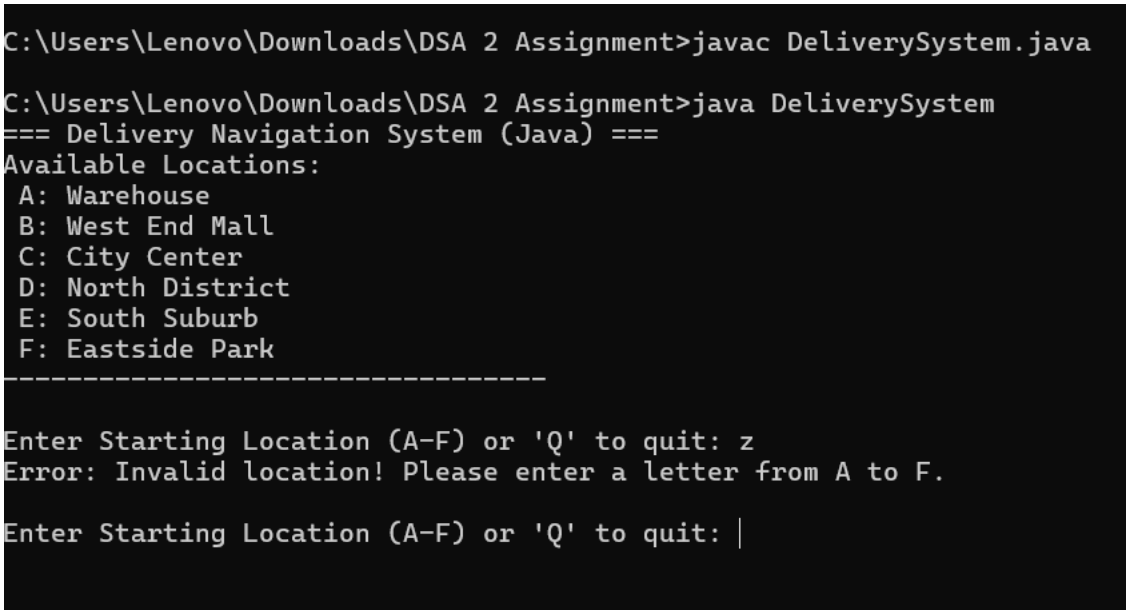
```
C:\Users\Lenovo\Downloads\DSA 2 Assignment>javac DeliverySystem.java

C:\Users\Lenovo\Downloads\DSA 2 Assignment>java DeliverySystem
=== Delivery Navigation System (Java) ===
Available Locations:
 A: Warehouse
 B: West End Mall
 C: City Center
 D: North District
 E: South Suburb
 F: Eastside Park
--------------------------------

Enter Starting Location (A-F) or 'Q' to quit: A

Shortest paths from Warehouse (A):
Destination              | Dist (km)  | Path
-------------------------------------------------------------
West End Mall (B)        | 6          | A -> B
City Center (C)          | 4          | A -> C
North District (D)       | 7          | A -> D
South Suburb (E)         | 10         | A -> C -> E
Eastside Park (F)        | 9          | A -> C -> F

Enter Starting Location (A-F) or 'Q' to quit:
```

4.3 Test Case 2:

Error Handling (Invalid Input) To test the system's robustness, I entered an invalid location "Z".

Input: Z

Observation: The system detected that "Z" is not part of the graph and displayed the error message: "Error: Invalid location! Please enter a letter from A to F." The program did not crash and allowed me to try again.



5.0 Conclusion

In conclusion, this assignment successfully demonstrated how Data Structures and Algorithms can be applied to solve real-world logistics problems. By modeling a delivery network as a weighted graph and implementing Dijkstra’s Algorithm, I was able to build a functional tool that helps find the most efficient delivery routes.

Throughout the development process, I learned the importance of choosing the right data structures. Using a Hash Map for the graph made the code cleaner and easier to manage than using a 2D array. Additionally, handling edge cases, such as invalid user inputs, proved to be essential for creating a professional-grade application. The final system meets all the requirements specified in the assignment guideline.

6.0 Appendix

DeliverySystem.java > ...

```
Click to add a breakpoint
l.*;

public class DeliverySystem {

    private static final Map<String, Map<String, Integer>> graph = new HashMap<>();

    private static final Map<String, String> locationNames = new HashMap<>();

    Run | Debug
    public static void main(String[] args) {
        initializeGraph();

        Scanner scanner = new Scanner(System.in);

        System.out.println(x: "=== Delivery Navigation System (Java) ===");
        printAvailableLocations();

        while (true) {
            try {
                System.out.print(s: "\nEnter Starting Location (A-F) or 'Q' to quit: ");
                String startInput = scanner.nextLine().trim().toUpperCase();

                if (startInput.equals(anObject: "Q")) {
                    System.out.println(x: "Exiting program. Goodbye!");
                    break;
                }

                if (!graph.containsKey(startInput)) {
                    System.out.println(x: "Error: Invalid location! Please enter a letter from A to F.");
                    continue;
                }
                runDijkstra(startInput);

            } catch (Exception e) {
                System.out.println("An unexpected error occurred: " + e.getMessage());
            }
        }
Click to add a breakpoint
.close();
    }

    private static void runDijkstra(String startNode) {
        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> previousNodes = new HashMap<>();

        for (String node : graph.keySet()) {
            distances.put(node, Integer.MAX_VALUE);
            previousNodes.put(node, value: null);
        }

        distances.put(startNode, value: 0);

        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.distance));
        pq.add(new Node(startNode, distance: 0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
            String currentNodeName = current.name;

            if (current.distance > distances.get(currentNodeName)) {
                continue;
            }

            Map<String, Integer> neighbors = graph.get(currentNodeName);
            if (neighbors != null) {
                for (Map.Entry<String, Integer> neighbor : neighbors.entrySet()) {
                    String neighborName = neighbor.getKey();
                    int weight = neighbor.getValue();
```

```
                int newDist = distances.get(currentNodeName) + weight;

                if (newDist < distances.get(neighborName)) {
                    distances.put(neighborName, newDist);
                    previousNodes.put(neighborName, currentNodeName);
                    pq.add(new Node(neighborName, newDist));
                }
            }
        }
    }

    printResults(startNode, distances, previousNodes);
}


private static void initializeGraph() {
    addPath(from: "A", to: "B", weight: 6); addPath(from: "A", to: "C", weight: 4); addPath(from: "A", to: "D", weigh… 7);
    addPath(from: "B", to: "A", weight: 6); addPath(from: "B", to: "E", weight: 5);
    addPath(from: "C", to: "A", weight: 4); addPath(from: "C", to: "D", weight: 3); addPath(from: "C", to: "E", weigh… 6); addPath("
    addPath(from: "D", to: "A", weight: 7); addPath(from: "D", to: "C", weight: 3); addPath(from: "D", to: "F", weigh… 4);
    addPath(from: "E", to: "B", weight: 5); addPath(from: "E", to: "C", weight: 6); addPath(from: "E", to: "F", weigh… 8);
    addPath(from: "F", to: "D", weight: 4); addPath(from: "F", to: "C", weight: 5); addPath(from: "F", to: "E", weigh… 8);

    locationNames.put(key: "A", value: "Warehouse");
    locationNames.put(key: "B", value: "West End Mall");
    locationNames.put(key: "C", value: "City Center");
    locationNames.put(key: "D", value: "North District");
    locationNames.put(key: "E", value: "South Suburb");
    locationNames.put(key: "F", value: "Eastside Park");
}

private static void addPath(String from, String to, int weight) {
    graph.putIfAbsent(from, new HashMap<>());
    graph.get(from).put(to, weight);
}

private static void printAvailableLocations() {
    System.out.println(x: "Available Locations:");
    for (String key : new TreeSet<>(locationNames.keySet())) {
        System.out.println(" " + key + ": " + locationNames.get(key));
    }
    System.out.println(x: "---------------------------------");
}

private static void printResults(String startNode, Map<String, Integer> distances, Map<String, String> previousNodes) {
    System.out.println("\nShortest paths from " + locationNames.get(startNode) + " (" + startNode + "):");
    System.out.printf(format: "%-25s | %-10s | %s%n", ...args: "Destination", "Dist (km)", "Path");
    System.out.println(x: "------------------------------------------------------------");

    for (String targetNode : new TreeSet<>(graph.keySet())) {
        if (targetNode.equals(startNode)) continue;

        int dist = distances.get(targetNode);
        String path = getPathString(previousNodes, startNode, targetNode);
        String destName = locationNames.get(targetNode) + " (" + targetNode + ")";

        System.out.printf(format: "%-25s | %-10d | %s%n", destName, dist, path);
    }
}

private static String getPathString(Map<String, String> previousNodes, String start, String end) {
    List<String> path = new ArrayList<>();
    String current = end;

    while (current != null) {
        path.add(current);
```

```
        while (current != null) {
            path.add(current);
            if (current.equals(start)) break;
            current = previousNodes.get(current);
        }

        Collections.reverse(path);

        if (!path.isEmpty() && path.get(index: 0).equals(start)) {
            return String.join(delimiter: " -> ", path);
        }
        return "No Path";
    }
    static class Node {
        String name;
        int distance;

        public Node(String name, int distance) {
            this.name = name;
            this.distance = distance;
        }
    }
}
```