|  |  |  |
| --- | --- | --- |
|  | **KONGU ENGINEERING COLLEGE**  (Autonomous)  Perundurai, Erode – 638 060  **DEPARTMENT OF INFORMATION TECHNOLOGY** | KEC | Kongu Engineering College |

**TRAVELPLANNER**

**AN MICRO PROJECT REPORT**

**for**

**DATA STRUCTURES USING JAVA (22ITC32)**

**Submitted by**

**A.SRIMADHANRAJA(23ITR155)**

**D.SUBIKSHAN(23ITR160)**

**V.K.SUVETHA(23ITR163)**

|  |  |  |
| --- | --- | --- |
|  | **KONGU ENGINEERING COLLEGE**  (Autonomous)  Perundurai, Erode – 638 060  **DEPARTMENT OF INFORMATION TECHNOLOGY** | KEC | Kongu Engineering College |

**BONAFIDE CERTIFICATE**

Name : A.Srimadhanraja(23ITR155)

D.Subikshan(23ITR160)

V.K.Suvetha(23ITR163)

Course Code : **22ITC31**

Course Name : **JAVA PROGRAMMING**

Semester : **III**

Certified that this is a bonafide record of work for application project done by the above students for **22ITC32 – JAVA PROGRAMMING** during the academic year **2024 - 2025.**

Submitted for the Viva Voce Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Faculty In-Charge Head of the Department**

# ABSTRACT

This project presents a **Travel Planner** application that uses **graph theory** to help users navigate between cities based on routes and distances. Built with **Java AWT**, the interface is simple and intuitive. Cities are represented as nodes, and routes as edges with distances, enabling efficient travel planning.

The application calculates the **shortest path** between cities using **Dijkstra’s algorithm**. Users can input cities, add routes, and quickly find the most efficient path. This ensures an optimal travel route for any selected pair of cities.

The system also includes a **city information module**, where users can learn about **landmarks** and **historical sites**. This adds valuable context to the travel planning experience. It transforms the tool from a basic planner into a more comprehensive travel assistant.

Designed with **usability** in mind, the system is easy to interact with. The Java AWT components like dropdowns and buttons make navigation smooth. The application is scalable, allowing dynamic updates of cities and routes, making it a practical tool for efficient travel planning.

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER**  **No.** | **TITLE** | **PAGE**  **No.** |
|  | **ABSTRACT** | **III** |
| **1** | **INTRODUCTION**  1.1 Motivation  1.2 Objective  1.3 Methodology Overview | **5**  5  5  5 |
| **2** | **PROBLEM STATEMENT** | 7 |
| **3** | **METHODOLOGY**  3.1. Graph Representation  3.2.Memory and performance  3.3.Scalable Architecture  3.4.Database Integration (Future**)**: | **8**  8  8  10  11  13 |
| **4** | **IMPLEMENTATION**  4.1 Coding and Dependencies | **15**  15 |
| **5** | **RESULTS AND DISCUSSION** | 22 |
| **6** | **CONCLUSION AND FUTURE WORK** | 25 |

## 1. INTRODUCTION

### The \*\*Travel Planner\*\* application is a graphical tool designed to simplify travel route planning by visualizing cities as nodes and routes as edges in a graph. With the growing need for efficient navigation through urban and regional areas, this system addresses the challenge by utilizing \*\*Dijkstra’s shortest path algorithm\*\* to calculate optimal travel paths. It allows users to manage city information, add new routes, and access detailed descriptions of cities, enhancing the overall travel planning experience. Built with \*\*Java AWT\*\*, the user interface is both straightforward and flexible, providing a cross-platform solution for efficient and dynamic route planning.

### 1.1 Motivation

### The motivation for this project comes from the need for efficient travel planning in regions with multiple interconnected cities. Traditional tools often lack customization and real-time adaptability for route optimization. By leveraging graph theory and Dijkstra’s algorithm, the \*\*Travel Planner\*\* app provides optimized travel paths. It also integrates city information to enrich the planning experience and offers flexibility for future enhancements, such as real-time data and route recommendations.

### 1.2 Objective

The objective of a Real Estate Property Management System is to provide a centralized platform for efficient property management, enabling users to list, search, and manage properties seamlessly.

It aims to simplify workflows, automate routine tasks, enhance user satisfaction, and improve decision-making for property managers, agents, landlords, and tenants. Additionally, it strives to ensure transparency, accessibility, and accuracy in property-related transactions and data management.

### 1.3 Methodology Overview

The methodology for addressing the problem statement involves enhancing both the **User Interface (UI)** and **Data Management**, as well as implementing specific features to improve user experience, system performance, and reliability. The following sections describe the main areas of focus for the methodology.

# 2.Problem Statement

The main challenge addressed by the \*\*Travel Planner\*\* application is helping travelers efficiently plan their journeys by finding the shortest and most cost-effective routes between cities. This becomes particularly important when navigating unfamiliar regions, where determining the optimal path can be difficult. The application aims to simplify this process by offering users the ability to quickly calculate the best routes using \*\*Dijkstra’s algorithm\*\*.

In addition to route optimization, the application also provides valuable city-specific information, enriching the travel planning experience. It balances \*\*usability\*\*, \*\*efficiency\*\*, and \*\*accuracy\*\*, ensuring that it caters to a wide range of users—from those with limited technical knowledge to experienced travelers. The tool is designed to meet diverse travel planning needs while maintaining ease of use and providing reliable results.

The tool is designed to meet diverse travel planning needs while maintaining ease of use, offering flexibility for future enhancements like real-time traffic data and personalized recommendations.

# 3.Methodology

3.1. **Graph Representation**:

* The cities and routes are represented as nodes and edges in a graph. As the system grows, managing this graph efficiently becomes crucial.
* **Adjacency lists** are used to represent the graph, which is ideal for sparse graphs (fewer routes). Using a **hash map** for city names ensures fast access and updates.

3.2. **Memory and Performance**:

* The implementation should be optimized to minimize memory usage while processing large numbers of cities and routes. Algorithms like **Dijkstra’s algorithm** should be optimized to handle large data sets efficiently.
* **Caching** frequently computed shortest paths can speed up repetitive queries.

3.3. **Scalable Architecture**:

* The graph model should be adaptable to accommodate new cities and routes easily without significant rework.
* The **dijkstra algorithm** uses priority queues, and by ensuring that the queue operations are optimized (through efficient data structures like **binary heaps**), we can minimize processing time for large graphs.

3.4. **Database Integration (Future)**:

* For better scalability, future iterations of the system could integrate with a **database** (e.g., **MySQL**, **PostgreSQL**) to store city and route data persistently. This would allow users to store their travel plans and access them later.

## 4.IMPLEMENTATION

**4.1 Coding and Dependencies**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.util.ArrayList;

import java.util.HashMap;

public class TravelManagementSystem extends JFrame implements ActionListener {

// Declare UI components

private JTextField tfFirstName, tfTelephone, tfCost, tfTax, tfSubtotal, tfTotal;

private JTextArea taAddress;

private JComboBox<String> departureChoice, destinationChoice, accommodationChoice;

private JCheckBox cbAirportTax, cbAirMiles, cbInsurance, cbLuggage;

private ArrayList<JCheckBox> foodCheckboxes = new ArrayList<>();

private HashMap<String, Double> destinationCosts = new HashMap<>();

private HashMap<String, Double> accommodationCosts = new HashMap<>();

private HashMap<String, Double> foodCosts = new HashMap<>();

private ArrayList<String> selectedFoodItems = new ArrayList<>();

public TravelManagementSystem() {

// Frame properties

setTitle("Travel Management System");

setSize(700, 800);

setLayout(new GridBagLayout());

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

GridBagConstraints gbc = new GridBagConstraints();

gbc.insets = new Insets(5, 5, 5, 5);

// Customer Details Panel

gbc.gridx = 0;

gbc.gridy = 0;

gbc.gridwidth = 2;

add(new JLabel("Customer Details"), gbc);

gbc.gridwidth = 1;

gbc.gridy++;

add(new JLabel("Name:"), gbc);

tfFirstName = new JTextField(25);

gbc.gridx = 1;

add(tfFirstName, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Address:"), gbc);

taAddress = new JTextArea(3, 25);

JScrollPane scrollPane = new JScrollPane(taAddress);

gbc.gridx = 1;

add(scrollPane, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Telephone:"), gbc);

tfTelephone = new JTextField(25);

tfTelephone.addKeyListener(new KeyAdapter() {

public void keyTyped(KeyEvent e) {

if (!Character.isDigit(e.getKeyChar()) || tfTelephone.getText().length() >= 10) {

e.consume();

}

}

});

gbc.gridx = 1;

add(tfTelephone, gbc);

// Travel Details Panel

gbc.gridx = 0;

gbc.gridy++;

gbc.gridwidth = 2;

add(new JLabel("Travel Details"), gbc);

gbc.gridwidth = 1;

gbc.gridy++;

add(new JLabel("Departure:"), gbc);

departureChoice = new JComboBox<>(new String[] {

"MM Airport", "YYZ - Toronto Pearson", "LAX - Los Angeles International",

"JFK - New York JFK", "LHR - London Heathrow"

});

gbc.gridx = 1;

add(departureChoice, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Destination:"), gbc);

destinationChoice = new JComboBox<>(new String[] {

"Canada - 7 Days in Ottawa", "USA - 5 Days in NYC", "France - 4 Days in Paris",

"Australia - 10 Days in Sydney", "Japan - 6 Days in Tokyo",

"Germany - 8 Days in Berlin", "Brazil - 7 Days in Rio", "South Africa - 5 Days in Cape Town"

});

gbc.gridx = 1;

add(destinationChoice, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Accommodation:"), gbc);

accommodationChoice = new JComboBox<>(new String[] {

"Single", "Double", "Family Suite", "Luxury Suite"

});

gbc.gridx = 1;

add(accommodationChoice, gbc);

// Additional Services Checkbox Panel

JPanel checkboxPanel = new JPanel(new GridLayout(2, 2, 10, 10));

cbAirportTax = new JCheckBox("Airport Tax");

cbAirMiles = new JCheckBox("Air Miles Over 20000");

cbInsurance = new JCheckBox("Traveling Insurance");

cbLuggage = new JCheckBox("Extra Luggage");

checkboxPanel.add(cbAirportTax);

checkboxPanel.add(cbAirMiles);

checkboxPanel.add(cbInsurance);

checkboxPanel.add(cbLuggage);

gbc.gridx = 0;

gbc.gridy++;

gbc.gridwidth = 2;

add(checkboxPanel, gbc);

// Food Accommodation Panel

gbc.gridy++;

gbc.gridwidth = 2;

add(new JLabel("Food Accommodation (Select any):"), gbc);

JPanel foodPanel = new JPanel(new GridLayout(3, 2, 10, 10));

foodCosts.put("Pizza", 10.0);

foodCosts.put("Pasta", 15.0);

foodCosts.put("Salad", 8.0);

foodCosts.put("Sandwich", 12.0);

foodCosts.put("Dessert", 5.0);

foodCosts.put("Beverage", 3.0);

for (String foodItem : foodCosts.keySet()) {

JCheckBox foodCheckbox = new JCheckBox(foodItem);

foodCheckbox.addItemListener(e -> {

if (e.getStateChange() == ItemEvent.SELECTED) {

selectedFoodItems.add(foodItem);

} else {

selectedFoodItems.remove(foodItem);

}

});

foodCheckboxes.add(foodCheckbox);

foodPanel.add(foodCheckbox);

}

gbc.gridy++;

add(foodPanel, gbc);

// Cost and Receipt Panel

gbc.gridy++;

gbc.gridwidth = 2;

add(new JLabel("Cost Summary"), gbc);

gbc.gridwidth = 1;

gbc.gridy++;

add(new JLabel("Cost:"), gbc);

tfCost = new JTextField("0", 15);

tfCost.setEditable(false);

gbc.gridx = 1;

add(tfCost, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Tax:"), gbc);

tfTax = new JTextField("0", 15);

tfTax.setEditable(false);

gbc.gridx = 1;

add(tfTax, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Subtotal:"), gbc);

tfSubtotal = new JTextField("0", 15);

tfSubtotal.setEditable(false);

gbc.gridx = 1;

add(tfSubtotal, gbc);

gbc.gridx = 0;

gbc.gridy++;

add(new JLabel("Total:"), gbc);

tfTotal = new JTextField("0", 15);

tfTotal.setEditable(false);

gbc.gridx = 1;

add(tfTotal, gbc);

// Buttons

gbc.gridx = 0;

gbc.gridy++;

gbc.gridwidth = 2;

JPanel buttonPanel = new JPanel(new FlowLayout());

JButton btnCalculate = new JButton("Calculate");

btnCalculate.addActionListener(this);

buttonPanel.add(btnCalculate);

JButton btnReset = new JButton("Reset");

btnReset.addActionListener(this);

buttonPanel.add(btnReset);

JButton btnReceipt = new JButton("Receipt");

btnReceipt.addActionListener(this);

buttonPanel.add(btnReceipt);

add(buttonPanel, gbc);

// Initialize destination and accommodation costs

destinationCosts.put("Canada - 7 Days in Ottawa", 500.0);

destinationCosts.put("USA - 5 Days in NYC", 600.0);

destinationCosts.put("France - 4 Days in Paris", 700.0);

destinationCosts.put("Australia - 10 Days in Sydney", 800.0);

destinationCosts.put("Japan - 6 Days in Tokyo", 900.0);

destinationCosts.put("Germany - 8 Days in Berlin", 850.0);

destinationCosts.put("Brazil - 7 Days in Rio", 750.0);

destinationCosts.put("South Africa - 5 Days in Cape Town", 780.0);

accommodationCosts.put("Single", 100.0);

accommodationCosts.put("Double", 150.0);

accommodationCosts.put("Family Suite", 200.0);

accommodationCosts.put("Luxury Suite", 300.0);

setVisible(true);

}

@Override

public void actionPerformed(ActionEvent e) {

String command = e.getActionCommand();

switch (command) {

case "Calculate":

calculateCosts();

break;

case "Reset":

resetFields();

break;

case "Receipt":

showReceipt();

break;

default:

break;

}

}

private void calculateCosts() {

double baseCost = destinationCosts.get(destinationChoice.getSelectedItem());

double accommodationCost = accommodationCosts.get(accommodationChoice.getSelectedItem());

double totalCost = baseCost + accommodationCost;

if (cbAirportTax.isSelected()) {

totalCost += 50;

}

if (cbAirMiles.isSelected()) {

totalCost += 25;

}

if (cbInsurance.isSelected()) {

totalCost += 100;

}

if (cbLuggage.isSelected()) {

totalCost += 30;

}

// Calculate food costs

double foodCost = 0;

for (String foodItem : selectedFoodItems) {

foodCost += foodCosts.get(foodItem);

}

totalCost += foodCost;

// Update cost fields

tfCost.setText(String.valueOf(totalCost));

tfTax.setText(String.valueOf(totalCost \* 0.05)); // Assuming 5% tax

tfSubtotal.setText(String.valueOf(totalCost));

tfTotal.setText(String.valueOf(totalCost + (totalCost \* 0.05)));

}

private void resetFields() {

tfFirstName.setText("");

taAddress.setText("");

tfTelephone.setText("");

tfCost.setText("0");

tfTax.setText("0");

tfSubtotal.setText("0");

tfTotal.setText("0");

cbAirportTax.setSelected(false);

cbAirMiles.setSelected(false);

cbInsurance.setSelected(false);

cbLuggage.setSelected(false);

departureChoice.setSelectedIndex(0);

destinationChoice.setSelectedIndex(0);

accommodationChoice.setSelectedIndex(0);

selectedFoodItems.clear();

for (JCheckBox foodCheckbox : foodCheckboxes) {

foodCheckbox.setSelected(false);

}

}

private void showReceipt() {

StringBuilder receipt = new StringBuilder("Receipt:\n");

receipt.append("Name: ").append(tfFirstName.getText()).append("\n");

receipt.append("Address: ").append(taAddress.getText()).append("\n");

receipt.append("Telephone: ").append(tfTelephone.getText()).append("\n");

receipt.append("Destination: ").append(destinationChoice.getSelectedItem()).append("\n");

receipt.append("Accommodation: ").append(accommodationChoice.getSelectedItem()).append("\n");

receipt.append("Total Cost: ").append(tfTotal.getText()).append("\n");

JOptionPane.showMessageDialog(this, receipt.toString(), "Receipt", JOptionPane.INFORMATION\_MESSAGE);

}

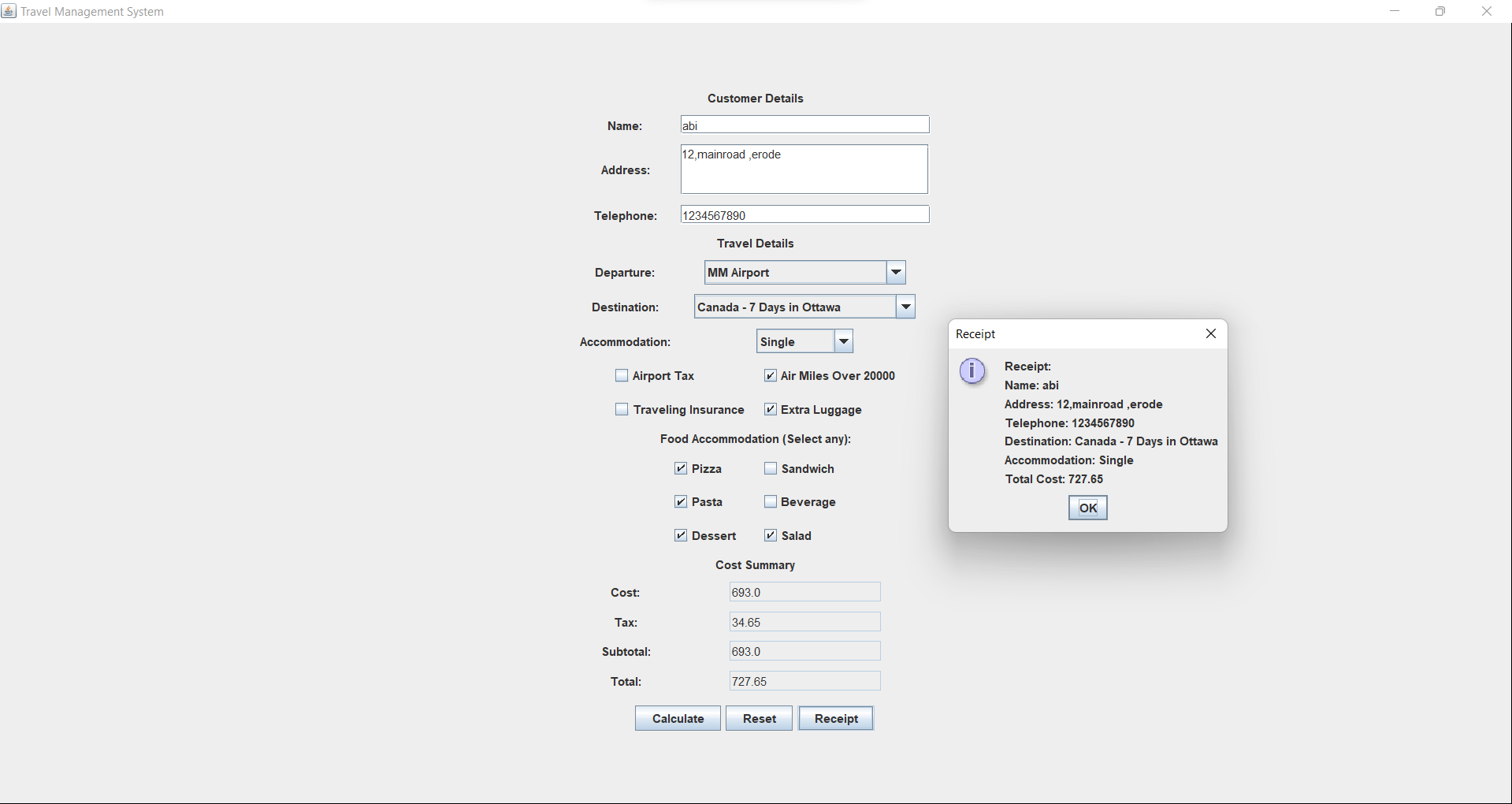
public static void main(String[] args) {

new TravelManagementSystem();

}

}

5. **RESULTS AND DISCUSSION**

****

**6.CONCLUSION AND FUTURE WORKS**

Conclusion

The \*\*Travel Planner\*\* application effectively addresses the need for efficient journey planning by utilizing \*\*Dijkstra's algorithm\*\* to compute the shortest routes between cities. By incorporating \*\*city-specific information\*\*, the application not only helps travelers optimize their routes but also enriches their experience with useful insights into the destinations they plan to visit.

The user interface, built with \*\*Java AWT\*\*, ensures ease of use and accessibility, making the tool suitable for a wide range of users, from casual travelers to those planning more detailed itineraries. Overall, the application is a practical solution for simplifying travel planning and enhancing the overall user experience.

Future Work

While the \*\*Travel Planner\*\* application offers a solid foundation for route optimization, there are several avenues for future improvement.

One potential enhancement is the integration of \*\*real-time traffic data\*\* to offer more accurate route suggestions based on current conditions. This would allow the app to dynamically adjust routes based on congestion, accidents, or road closures, providing even more efficient travel plans.

Another area for development is the expansion of the city database, including more cities and additional features such as \*\*weather updates\*\* or \*\*local event information\*\*, which could further enrich the user’s travel experience. Additionally, the interface could be enhanced with \*\*interactive maps\*\* and \*\*visualizations\*\*, making it even more user-friendly and intuitive.

On the downside, while the current approach is effective for basic route planning, scaling the application to handle a larger dataset (e.g., a national or global network of cities) could introduce performance issues.

Similarly, the reliance on a static city database could limit the system's ability to stay up to date with new routes or cities. As the system evolves, these challenges will need to be addressed to maintain its effectiveness and reliability.