Flight Route Planner

A Project Report

Submitted to

Dr. Maajid Bashir

(Associate Professor)

Submitted by

Karan

(24MCA20147)

in partial fulfilment for the award of the degree of

Master of Computer Application

Chandigarh University

ACKNOWLEDGEMENT

With the submission of this project, we would like to express our gratitude towards all the people who provided us with their valuable assistance in the course of completion of the project. It gives us immense pleasure in submitting this project ―SCHOOL NOTICE BOARD‖. We have developed this project as a minor project for 2nd Semester. We are highly grateful to the esteemed University faculty for giving us required knowledge to finish our project and we wish to express our profound gratitude and sincere thanks to **Dr. Maajid Bashir** (The Project Supervisor), our project guide, without whose valuable guidance and constructive criticism this project would have been impossible. We are highly grateful to other faculty members of Department of Computer Science as they are the one who taught us the basics of project making. We are grateful to our family and to all friends who helped us in making this project possible with their positive and enthusiastic attitude towards us.

At last, but not the least we consider ourself proud to be a part of University Institute of Computing, Chandigarh University.

INTRODUCTION TO PROJECT

In today's digital era, where travel and logistics play a major role in everyday life, the need for effective route planning tools is more important than ever. The **Flight Route Planner** is a Python- based desktop application designed to simulate real-world flight networks and compute the **cheapest route between two cities**. Built using **Dijkstra's algorithm**, the application helps users visualize how optimal routes are selected based on cost efficiency.

The application uses **Tkinter** for the graphical user interface, allowing users to select their departure and arrival cities from a predefined list. After selection, the system calculates the most cost-effective flight route and **displays the result both textually and graphically** using **NetworkX** and **Matplotlib**. The graph representation includes all city connections and highlights the selected route with a detailed cost analysis.

This project is an excellent integration of **algorithmic concepts, GUI development, and data visualization**, showcasing how academic knowledge can be applied to solve practical, real-world problems.

Problem Statement

In the aviation industry, as well as in broader transportation and logistics domains, finding the most efficient and economical path between two points is critical. However, when multiple routes exist, and each route has different costs or weights, determining the cheapest option becomes computationally complex.

Most people rely on automated systems for such route optimization without understanding how those systems operate internally. This creates a knowledge gap, especially for students or learners studying algorithms like **Dijkstra's**.

Therefore, the project addresses the following core problems:

- How to represent a flight network between multiple cities efficiently?
- How to calculate the **cheapest flight route** between two cities with varying travel costs?
- How to **visually represent** the network and the selected optimal path in an intuitive and user-friendly format?
   How to build an interactive GUI that allows users to easily engage with the system?

By solving these problems, the project helps users understand and visualize the power of shortest- path algorithms in a real-world context.

Project Motivation

The motivation behind this project arises from a desire to bridge the gap between theoretical computer science and practical applications. Students often learn graph traversal algorithms like Dijkstra's in isolation, without seeing how they operate in practical scenarios like flight planning, GPS navigation, or logistics.

Key motivations include:

- **Educational Value**: Offering a hands-on way for students to learn and observe shortest path algorithms in action.
- **Visualization Impact**: Visual tools make it easier to comprehend complex data structures like graphs, enhancing both teaching and learning experiences.
- **Skill Integration**: Encouraging the integration of multiple skills—data structures, algorithm design, GUI development, and visualization libraries in Python.
- **Practical Simulation**: Simulating a real-world scenario like flight planning, which could be expanded into a full-fledged travel management system.

Ultimately, the project not only serves as a valuable learning tool but also as a prototype for building scalable route-planning systems.

REQUIREMENTS SPECIFICATION

SOFTWARE REQUIREMENTS

Operating System	: 64bit WINDOWS Operating System, X64-based processor
Language	: Python

HARDWARE REQUIREMENTS

Processor	: Intel Celeron CPU N3060 @1.60GHz or Above
RAM	: 4.00 GB or Above
Hard Disk	: 1 TB
Compact Disk	: CD-ROM, CD-R, CD-RW
Input device	: Keyboard

Tools & Technologies Used

Category	Tool / Technology	Purpose
Programming Language	Python	Core language for implementing the logic and GUI
GUI Library	Tkinter	Built-in Python library used to design the user interface
Graph Algorithm	Dijkstra's Algorithm (via custom implementation with `heapq`)	Used to compute the shortest/cheapest path between cities
Graph Processing Library	NetworkX	For creating, managing, and analyzing the flight graph (nodes and edges)
Visualization Library	Matplotlib	Used to visually render the flight network and shortest path route
Priority Queue	**heapq** (Python standard library)	Optimizes Dijkstra's algorithm by selecting the next city with minimum cost efficiently
Error Dialogs / Message Boxes	tkinter.messagebox	Used to display input validation errors and prompts
Python IDE	Any (e.g., **PyCharm**, **VS Code**, or	For developing and testing the Python code **IDLE**)
Operating System	Platform-independent (Windows, macOS, Linux)	Can be run on any OS that supports Python and required libraries

Project Scope

The **Flight Route Planner** project aims to develop a desktop-based application that allows users to determine the **cheapest flight route** between two Indian cities using **Dijkstra's shortest path algorithm**. The system is designed for educational and practical demonstration purposes, focusing on route optimization in transportation or travel systems.

Key Objectives:

- Implement a graph-based route planner using **Dijkstra's algorithm**.
- Allow users to **select source and destination cities** via a graphical interface.
- **Display the shortest route** and total travel cost using textual and visual outputs.
- Enhance user experience with a clean, interactive GUI.

In-Scope:

- Static predefined graph of Indian cities and flight costs.
- GUI built with **Tkinter** for city selection and output.
- Graph visualization using **NetworkX and Matplotlib**.
- Cost calculation and real-time path rendering.
- Error handling for incomplete or invalid input.

Out-of-Scope:

- Real-time data from airline APIs.
- Ticket booking or live flight schedules.

- Web or mobile application interface (this version is desktop-based).
- Dynamic graph updates based on user inputs.

Long-Term Vision (Future Scope):

- Integrate real-world APIs to fetch live flight data.
- Build a web/mobile version using frameworks like Flask, Django, or React Native.
- Include additional filters like time, airlines, or layovers.
- Expand to a global route planner with multiple transport modes (bus, train, etc.).

Key Features

Feature	Description
1. GUI-Based City Selection	Users can choose **departure and arrival cities** using a clean and interactive interface built with **Tkinter**.
2. Dijkstra's Algorithm	Calculates the **cheapest path** using a highly efficient implementation of **Dijkstra's shortest path algorithm** with a priority queue.
3. Cost Calculation	Displays the **total travel cost** between the selected cities based on the graph data.
4. Visual Route Display	Uses **NetworkX** and **Matplotlib** to draw the complete flight network and **highlight the shortest path** in red.
5. Error Handling	Provides **input validation** and shows pop-up error messages if cities are not selected correctly.
6. Custom Styling	Color-coded nodes and edges make the visual graph clear and visually appealing.
7. Responsive Layout	The application window is centered and resizes appropriately across different screen sizes.
8. Path Tracing	Displays the **exact route (city-to-city)** in order, along with the cost.
9. Modular Code Structure	Clean separation of logic (algorithm), UI (Tkinter), and visualization (Matplotlib + NetworkX).
10. Educational Purpose	Ideal for **demonstrating graph theory and pathfinding algorithms** in real-world applications.

Challenges & Solutions

Challenge	Solution

1. Implementing Dijkstra's Algorithm Efficiently	Used Python's `heapq` module to implement a **priority queue**, ensuring that the algorithm runs efficiently even with multiple nodes and connections.
2. Creating an Interactive and Clean GUI	Utilized **Tkinter** to build a simple, intuitive, and user-friendly GUI layout with proper grouping for inputs and outputs.
3. Handling Invalid or Incomplete Inputs	Integrated **input validation** and user prompts using `messagebox` to prevent the application from crashing or returning incorrect paths.
4. Visualizing the Graph and Highlighted Path Clearly	Combined **NetworkX** for graph structure and **Matplotlib** for visualization. Applied color-coding and custom layout (`spring_layout`) to make paths and nodes more understandable.

5. Keeping UI Responsive with Limited the scope of the graph to a fixed set of cities and used **optimized layout** Complex Visualizations **calculations** to ensure the UI remains responsive.

6. Preventing Confusion in Used **directed edges** (`DiGraph`) and clearly labeled weights to show the direction and Bidirectional Paths cost of flights.

7. Ensuring Readability Across Adjusted **window centering and layout spacing** to make the app look consistent Different Devices across different screen sizes.

8. Combining Multiple Carefully managed integration of **Tkinter, NetworkX, and Matplotlib** by calling Libraries Smoothly visualization in a non-blocking way after user interaction.

CODE OF IMPLEMENTATION

```
import heapq
import matplotlib.pyplot as plt import
networkx as nx import
tkinter as tk
from tkinter import messagebox

# Dijkstra's algorithm def
dijkstra(graph,        start):
queue = [(0, start)]
    distances = {node: float('inf') for node in graph} distances[start] = 0 previous_nodes
    = {node: None for node in graph}

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]: continue
```

```
        for    neighbor,    weight    in    graph[current_node].items():    distance    =
            current_distance + weight
            if distance < distances[neighbor]: distances[neighbor]
                = distance previous_nodes[neighbor] =
                current_node heapq.heappush(queue, (distance,
                neighbor))
    return distances, previous_nodes def
shortest_path(graph, start, end):
    distances, previous_nodes = dijkstra(graph, start) path = []
    step = end

    while    step:    path.append(step)
        step = previous_nodes[step]

   path.reverse() return path,
   distances[end] def
   calculate_shortest_path():
   if start_city.get() ==
   "None" or end_city.get()
   == "None":
   messagebox.showerror("I
   nvalid Input",
        "Please   select   both   start   and   end
    cities.") else: start = start_city.get() end =
        end_city.get()
        path,    cost    =    shortest_path(graph,    start,    end)    result_label.config(
            text=f"Shortest path: {' → '.join(path)}\nTotal cost: ₹{cost:,}", fg="black"
        )
        visualize_path(path, start, end)

       def visualize_path(path, start, end): G =
    nx.DiGraph()

    for node in graph:
        for neighbor, weight in graph[node].items():
            G.add_edge(node, neighbor, weight=weight)

    pos          =          nx.spring_layout(G,          k=0.5)
    plt.figure(figsize=(12, 9))
    nx.draw(G,    pos,    with_labels=True,    node_color='#AED6F1',    node_size=3000,
            font_size=10, font_weight='bold', edge_color='#7FB3D5', width=1.5, arrows=True)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_color='black')
```

```
path_edges = [(path[i], path[i+1]) for i in range(len(path)-1)] nx.draw_networkx_nodes(G, pos,
nodelist=path, node_color='#E74C3C')
    nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='#E74C3C', width=3) plt.title(f"Cheapest
    Flight Route: {start} to {end}\nTotal Cost: ₹{sum(graph[u][v] for u, v in
path_edges):,}", color='black') plt.show()

graph = {
    'Delhi': {'Mumbai': 4, 'Ahmedabad': 8, 'Bangalore': 12, 'Hyderabad': 15, 'Chennai': 3, 'Kolkata':
18, 'Kochi': 7, 'Chandigarh': 10},
    'Mumbai': {'Bangalore': 8, 'Hyderabad': 5, 'Delhi': 9, 'Ahmedabad': 14, 'Chennai': 2, 'Kolkata': 11,
'Kochi': 6, 'Chandigarh': 13},
    'Bangalore': {'Hyderabad': 7, 'Kolkata': 4, 'Delhi': 16, 'Mumbai': 17, 'Chennai': 5, 'Kochi': 19,
'Ahmedabad': 1, 'Chandigarh': 20},
    'Hyderabad': {'Chennai': 9, 'Kochi': 2, 'Delhi': 10, 'Mumbai': 3, 'Bangalore': 6, 'Kolkata': 12, 'Ahmedabad': 18,
    'Chandigarh': 14},
  'Chennai': {'Kolkata': 10, 'Delhi': 11, 'Mumbai': 7, 'Bangalore': 13, 'Hyderabad': 8, 'Kochi': 4,
'Ahmedabad': 15, 'Chandigarh': 17},
    'Kolkata': {'Kochi': 2, 'Delhi': 5, 'Mumbai': 9, 'Bangalore': 3, 'Hyderabad': 16, 'Chennai': 6,
'Ahmedabad': 12, 'Chandigarh': 19},
'Kochi': {'Ahmedabad': 1, 'Delhi': 14, 'Mumbai': 18, 'Bangalore': 10, 'Hyderabad': 4, 'Chennai': 7,
'Kolkata': 13, 'Chandigarh': 20},
    'Ahmedabad': {'Delhi': 8, 'Chandigarh': 7, 'Mumbai': 11, 'Bangalore': 2, 'Hyderabad': 17,
'Chennai': 12, 'Kolkata': 5, 'Kochi': 16},
    'Chandigarh': {'Bangalore': 2, 'Kolkata': 6, 'Delhi': 15, 'Mumbai': 10, 'Hyderabad': 13, 'Chennai':
18, 'Kochi': 3, 'Ahmedabad': 19}
        }

# GUI root =
tk.Tk()
root.title("Flight Route Planner") root.configure(bg='#EAEDED')

window_width = 900 window_height = 700 screen_width =
root.winfo_screenwidth() screen_height = root.winfo_screenheight() center_x
= int(screen_width/2 - window_width/2) center_y = int(screen_height/2 -
window_height/2)
root.geometry(f'{window_width}x{window_height}+{center_x}+{center_y
}')

title_frame = tk.Frame(root, bg='#FF9933') title_frame.pack(fill=tk.X,
pady=(0, 10))
tk.Label(title_frame, text="Flight Route Planner", font=('Arial', 16, 'bold'), bg='#FF9933',
        fg='black').pack()
```

start_city = tk.StringVar(value="None") end_city = tk.StringVar(value="None")

selection_frame = tk.Frame(root, bg='#EAEDED')

selection_frame.pack(pady=10)

start_frame = tk.Frame(selection_frame, bg='#EAEDED') start_frame.pack(side=tk.LEFT, padx=20)
tk.Label(start_frame, text="Departure City:", font=('Arial', 11, 'bold'),

bg='#EAEDED', fg='black').pack() end_frame = tk.Frame(selection_frame,

bg='#EAEDED') end_frame.pack(side=tk.LEFT, padx=20)

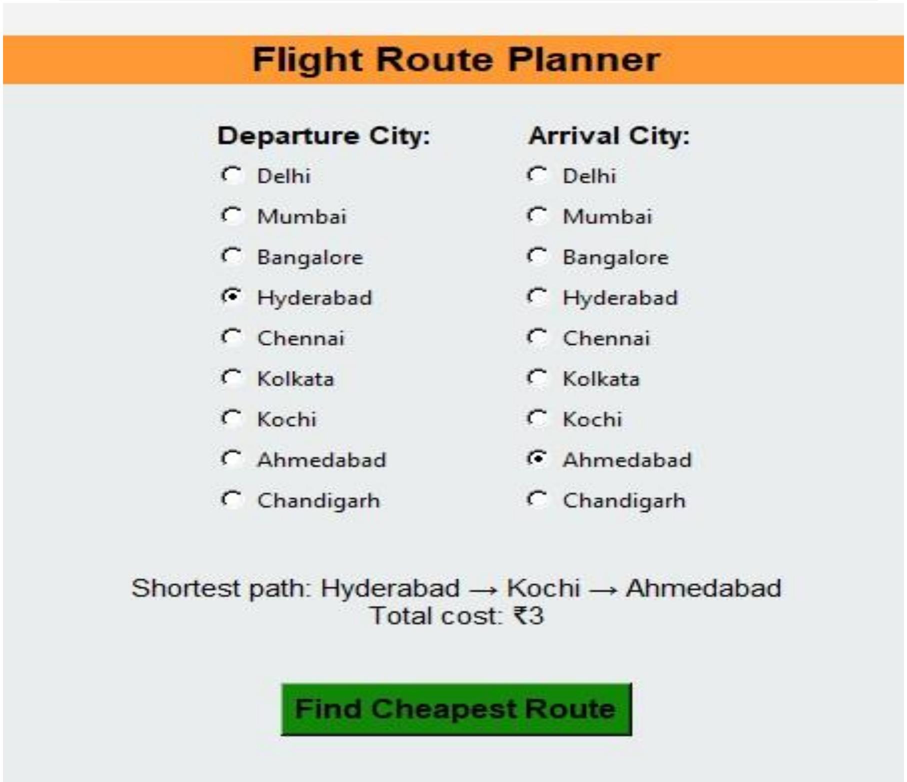
tk.Label(end_frame, text="Arrival City:", font=('Arial', 11, 'bold'), bg='#EAEDED', fg='black').pack()

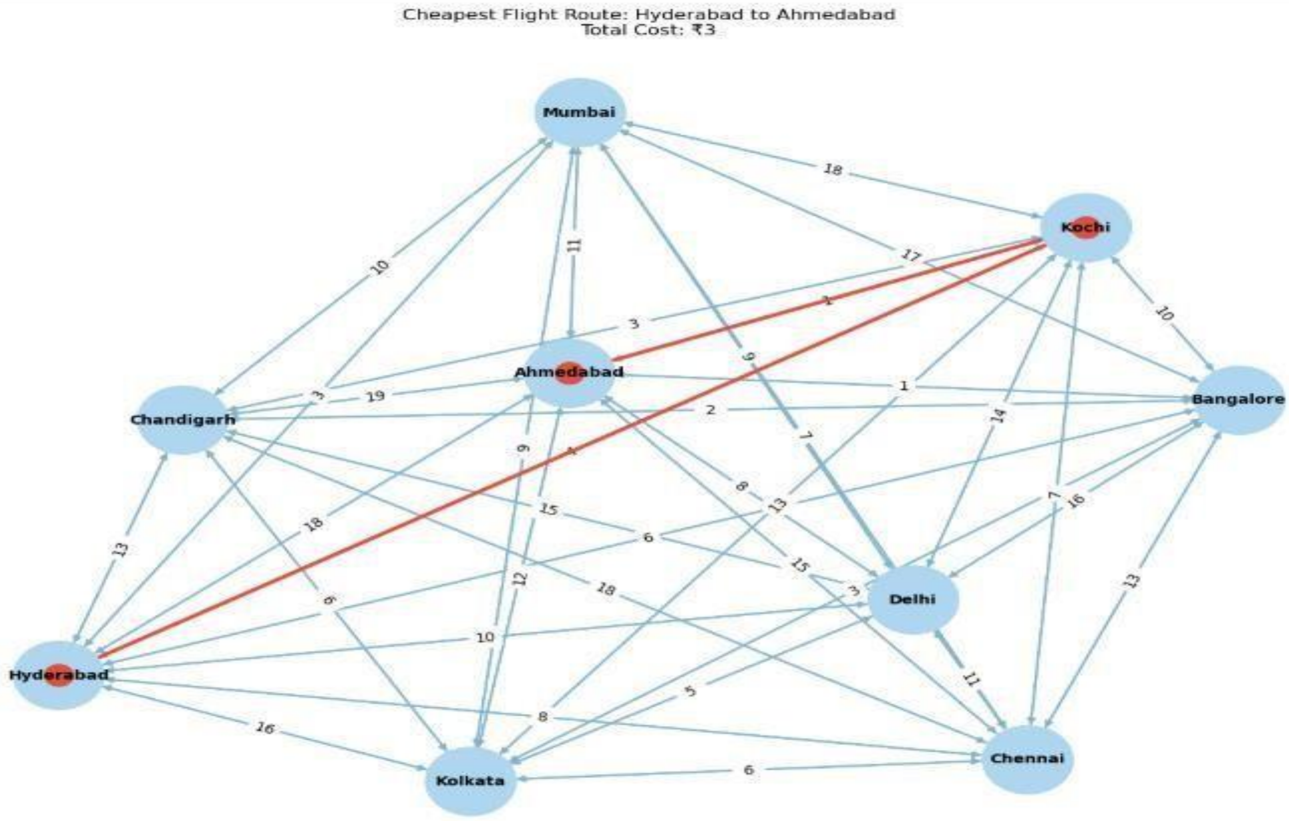
for city in graph.keys(): tk.Radiobutton(start_frame, text=city, variable=start_city, value=city, bg='#EAEDED',
fg='black', anchor=tk.W).pack(fill=tk.X) tk.Radiobutton(end_frame, text=city, variable=end_city, value=city, bg='#EAEDED', fg='black', anchor=tk.W).pack(fill=tk.X)

result_label = tk.Label(root, text="Your flight route will appear here.", wraplength=800,
font=('Arial', 11), bg='#EAEDED',
fg='black')
result_label.pack(pady=20)
tk.Button(root, text="Find Cheapest Route", command=calculate_shortest_path, bg='#138808', fg='black', font=('Arial', 12, 'bold')).pack(pady=10)
def update_display(*args): pass # could be used to show dynamic selections
start_city.trace_add('write', update_display)
end_city.trace_add('write', update_display) root.mainloop()

RESULT


Flight Route Planner
Departure City:
Delhi
Mumbai
Bangalore
Hyderabad
Chennai
Kolkata
Kochi
Ahmedabad
Chandigarh
Arrival City:
Delhi
Mumbai
Bangalore
Hyderabad
Chennai
Kolkata
Kochi
Ahmedabad
Chandigarh
Your flight route will appear here.
Find Cheapest Route


Flight Route Planner
Departure City:
Delhi
Mumbai
Bangalore
Hyderabad
Chennai
Kolkata
Kochi
Ahmedabad
Chandigarh
Arrival City:
Delhi
Mumbai
Bangalore
Hyderabad
Chennai
Kolkata
Kochi
Ahmedabad
Chandigarh
Shortest path: Hyderabad → Kochi → Ahmedabad
Total cost: ₹3
Find Cheapest Route



CONCULSION

The **Flight Route Planner** is a successful demonstration of how algorithms can be practically applied to real-life problems in route optimization. By using **Dijkstra's algorithm**, the application effectively identifies the cheapest route between selected cities. The use of **Tkinter** for GUI and **NetworkX + Matplotlib** for graph visualization ensures that users receive both interactivity and clarity.

This project fulfills both educational and functional goals—it teaches users how pathfinding algorithms work while providing a hands-on, visual, and interactive experience. Moreover, it lays the foundation for more advanced features such as:

- Real-time data integration (e.g., actual flight prices),
- Time-based route optimization,
- Multi-stop route planning,
- Incorporation of more cities and geographic maps.

In conclusion, this project not only solves a specific problem but also opens up avenues for learning, innovation, and further development in the fields of **algorithmic design**, **software development**, and **data visualization**.

References

1. **Dijkstra's Algorithm** ○ Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
    ○ GeeksforGeeks: Dijkstra's Algorithm
2. **Tkinter Documentation** ○ Tkinter (Python GUI) Official Docs ○ Tutorialspoint: Python Tkinter Tutorial
3. **NetworkX & Matplotlib** ○ NetworkX Documentation ○ Matplotlib Documentation
    ○ Python Graph Tutorials - NetworkX
4. **Python Priority Queue (heapq)** ○ Python Official Docs - heapq Module
5. **Graph Theory Concepts** ○ Brilliant.org: Graph Theory
6. **Inspiration & Example Projects** ○ GitHub Repositories and Medium Articles featuring similar pathfinding visualizations for educational use.