UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

NAAC GRADE A+ ACCREDITED UNIVERSITY

Project Report

Topic: Route Finder

Submitted By:
Name: Anuj
UID: 24MCI10020
Class: MCA (AIML)
Section: 3/A

Submitted To:
Dr. Ashaq Hussain Bhat

University Institute of Computing
Chandigarh University, Gharuan, Mohali

INDEX

1. Introduction

2. Aim

3. Objective

4. Programming Language & Tools Used

5. Implementation

6. Dependencies

7. Output

8. Conclusion

9. Learning Outcome

10. Future Scope

1. Introduction

In the age of smart and intelligent systems, route optimization is crucial in logistics, navigation, and transportation. The 'Route Finder' project is designed to find and display optimal paths between cities using Artificial Intelligence search techniques. It allows users to find paths using BFS, DFS, and A* algorithms and visually renders the results using interactive maps with Folium.

2. Aim

To develop an AI-based interactive system that identifies optimal or near-optimal paths between two cities using various search algorithms with real-time visualization.

3. Objective

- Implement and compare BFS, DFS, and A* algorithms.
- Display selected routes on a map using Folium.
- Develop an easy-to-use GUI using Tkinter.
- Provide interactive route planning between popular US cities.

4. Programming Language & Tools Used

- Python
- Tkinter
- NetworkX
- Folium
- Webbrowser
- Geopy

5. Implementation

The project allows the user to select source and destination cities from dropdowns in a GUI. Depending on the selected search algorithm, the system computes a route:

- BFS: Explores neighbors level by level.
- DFS: Explores as far as possible along each branch.

- A*: Uses heuristic (Euclidean distance) and actual distance for optimal search.

The result is shown on an interactive Folium map.

```
import tkinter as tk
from tkinter import messagebox
import networkx as nx
import folium
import webbrowser
import random
import time
from geopy.geocoders import Nominatim


# Global Variables
city_locations = {
    "New York": (40.7128, -74.0060),
    "Los Angeles": (34.0522, -118.2437),
    "Chicago": (41.8781, -87.6298),
    "Houston": (29.7604, -95.3698),
    "Miami": (25.7617, -80.1918),
    "Dallas": (32.7767, -96.7970),
    "San Francisco": (37.7749, -122.4194),
    "Las Vegas": (36.1699, -115.1398),
    "Denver": (39.7392, -104.9903),
    "Seattle": (47.6062, -122.3321),
}
```

```
# Graph Representation
graph = nx.Graph()
for city in city_locations:
    graph.add_node(city, pos=city_locations[city])


# Adding random weighted edges to simulate real distances
edges = [
    ("New York", "Chicago"), ("New York", "Miami"),
    ("Los Angeles", "San Francisco"), ("Los Angeles", "Las Vegas"),
    ("Chicago", "Houston"), ("Chicago", "Dallas"),
    ("Houston", "Miami"), ("Houston", "Dallas"),
    ("San Francisco", "Seattle"), ("Las Vegas", "Denver"),
    ("Denver", "Seattle"), ("Dallas", "Las Vegas"),
]
for edge in edges:
    graph.add_edge(edge[0], edge[1], weight=random.randint(500, 2000))


# Tkinter UI
class RouteFinderApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Interactive Route Finder")
        self.root.geometry("600x500")
        self.start_city = tk.StringVar()
        self.end_city = tk.StringVar()
        # UI Elements
        tk.Label(root, text="Start City:").pack()
        self.start_dropdown = tk.OptionMenu(root, self.start_city, *city_locations.keys())
        self.start_dropdown.pack()

        tk.Label(root, text="End City:").pack()
        self.end_dropdown = tk.OptionMenu(root, self.end_city, *city_locations.keys())
        self.end_dropdown.pack()

        # Buttons
        tk.Button(root, text="Find Route (BFS)", command=self.find_route_bfs).pack(pady=5)
        tk.Button(root, text="Find Route (DFS)", command=self.find_route_dfs).pack(pady=5)
        tk.Button(root, text="Find Best Route (A*)", command=self.find_route_astar).pack(pady=5)

    def find_route_bfs(self):
        self.find_route("BFS")

    def find_route_dfs(self):
        self.find_route("DFS")

    def find_route_astar(self):
        self.find_route("A*")
    def find_route(self, algorithm):
        start = self.start_city.get()
        end = self.end_city.get()
        if not start or not end:
            messagebox.showerror("Error", "Please select both start and end cities!")
            return

        if algorithm == "BFS":
            path = self.bfs(start, end)
        elif algorithm == "DFS":
            path = self.dfs(start, end)
        elif algorithm == "A*":
            path = self.astar(start, end)
        else:
            return

        if path:
            messagebox.showinfo("Success", f"Route found using {algorithm}: {' → 
'.join(path)}")
            self.show_route_on_map(path)
        else:
            messagebox.showerror("Error", "No route found!")

    def bfs(self, start, end):
```

```
    queue = [[start]]
    visited = set()

    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node in visited:
            continue
        for neighbor in graph.neighbors(node):
            new_path = list(path)
            new_path.append(neighbor)
            queue.append(new_path)
            if neighbor == end:
                return new_path
        visited.add(node)
    return None

def dfs(self, start, end):
    stack = [[start]]
    visited = set()

    while stack:
        path = stack.pop()
        node = path[-1]
        if node in visited:
            continue
        for neighbor in graph.neighbors(node):
            new_path = list(path)
            new_path.append(neighbor)
            stack.append(new_path)
            if neighbor == end:
                return new_path
        visited.add(node)
    return None

def astar(self, start, end):
    def heuristic(city1, city2):
        lat1, lon1 = city_locations[city1]
        lat2, lon2 = city_locations[city2]
        return ((lat1 - lat2) ** 2 + (lon1 - lon2) ** 2) ** 0.5  # Euclidean Distance

    open_list = [(0, [start])]
    visited = set()

    while open_list:
        cost, path = min(open_list, key=lambda x: x[0])
        open_list.remove((cost, path))
        node = path[-1]
        if node in visited:
            continue
        if node == end:
            return path
        for neighbor in graph.neighbors(node):
            new_path = list(path)
            new_path.append(neighbor)
            new_cost = cost + graph[node][neighbor]['weight'] + heuristic(neighbor, end)
            open_list.append((new_cost, new_path))
        visited.add(node)
    return None

def show_route_on_map(self, path):
    # Generate Map
    m = folium.Map(location=city_locations[path[0]], zoom_start=4)
    for city in path:
        folium.Marker(city_locations[city], popup=city).add_to(m)

    for i in range(len(path) - 1):
        folium.PolyLine([city_locations[path[i]], city_locations[path[i + 1]]],
color="blue").add_to(m)

    map_path = "route_map.html"
    m.save(map_path)
    webbrowser.open(map_path)

# Start Tkinter App
```

```
if __name__ == "__main__":
    root = tk.Tk()
    app = RouteFinderApp(root)
    root.mainloop()
```

6. Dependencies

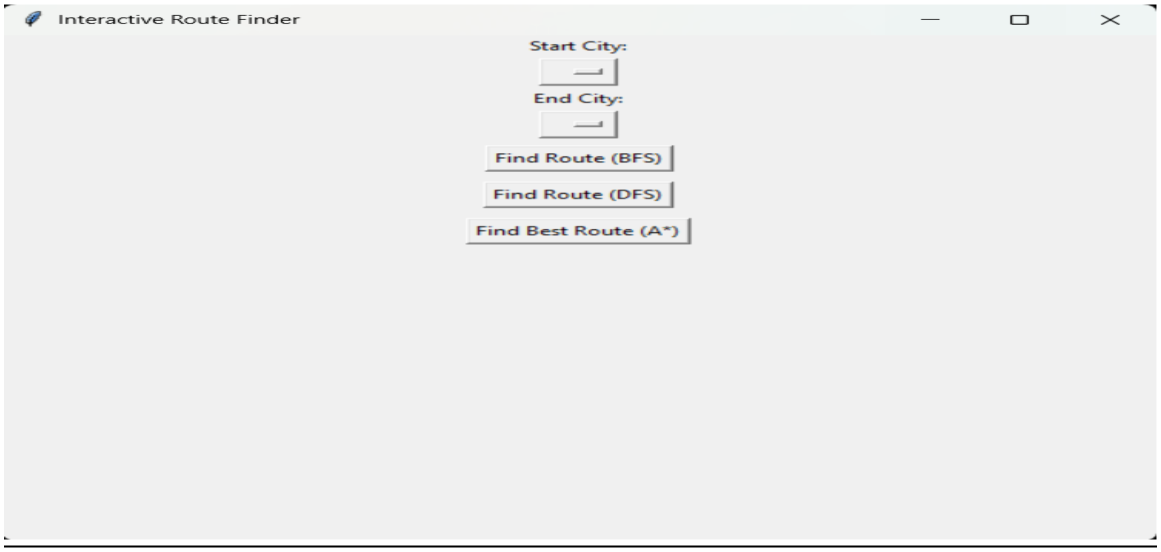
- Tkinter
- NetworkX
- Folium
- Random
- Webbrowser
- Geopy (optional)

7. Output

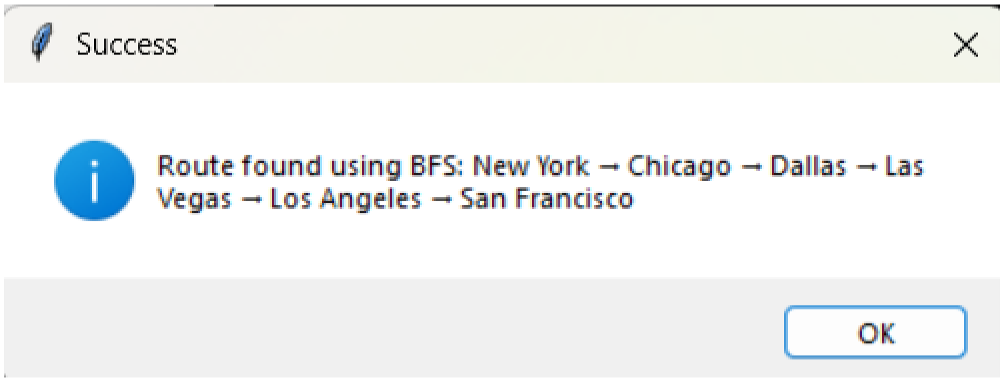
After selecting the cities and algorithm, the application:

- Displays the selected route in a popup message.
- Launches a web browser with an interactive map showing the route.

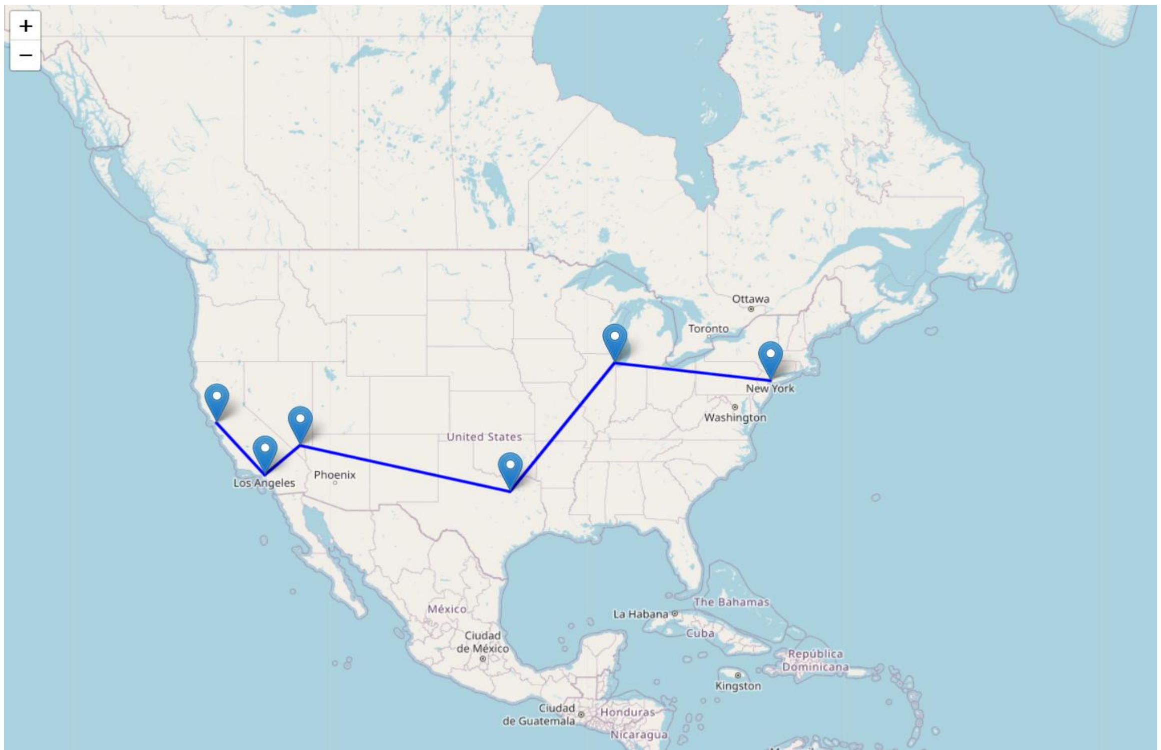
Example Output:



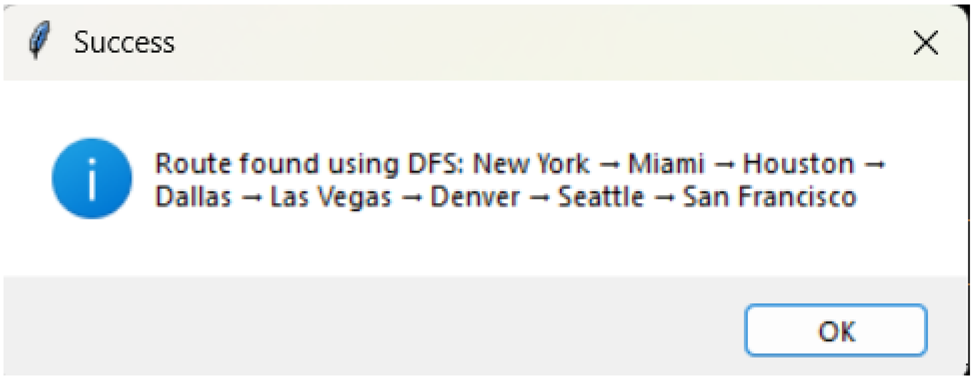
BFS:-



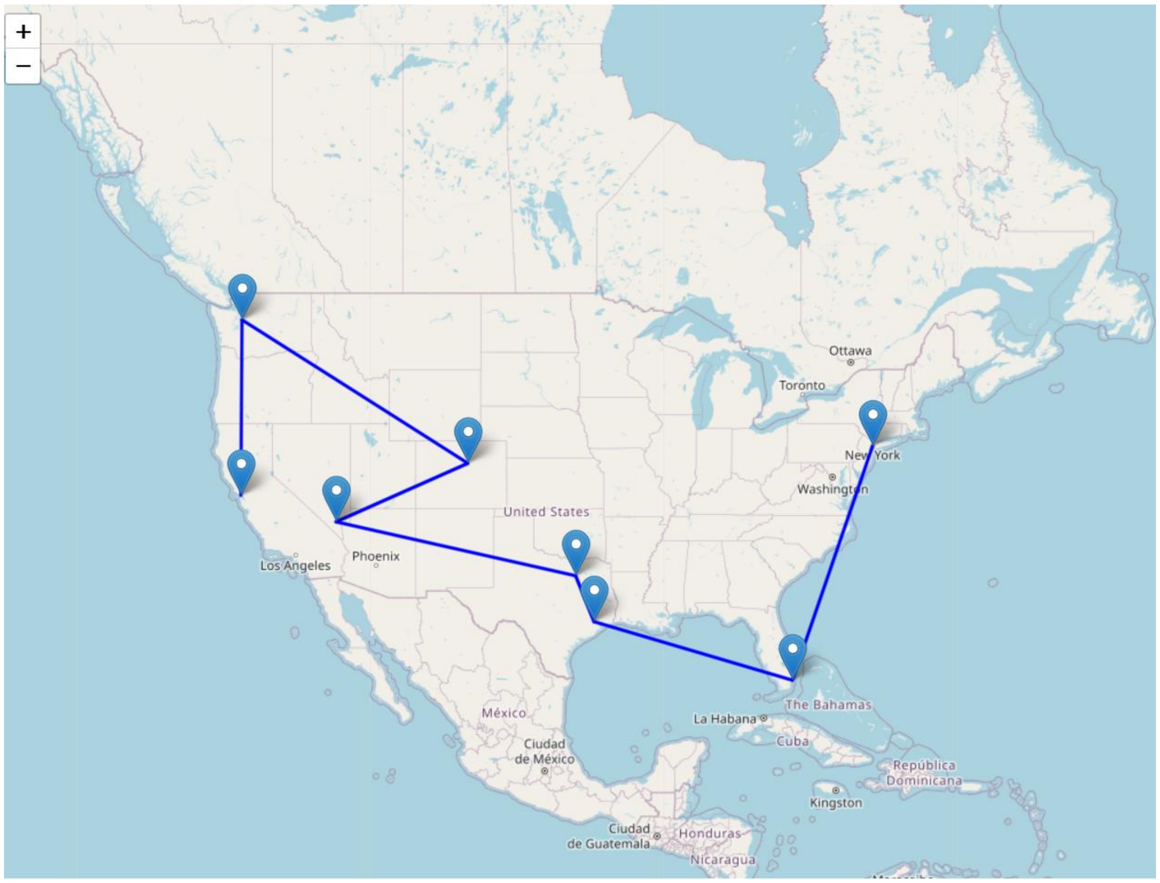
Route found using BFS: New York → Chicago → Dallas → Las Vegas → Los Angeles → San Francisco



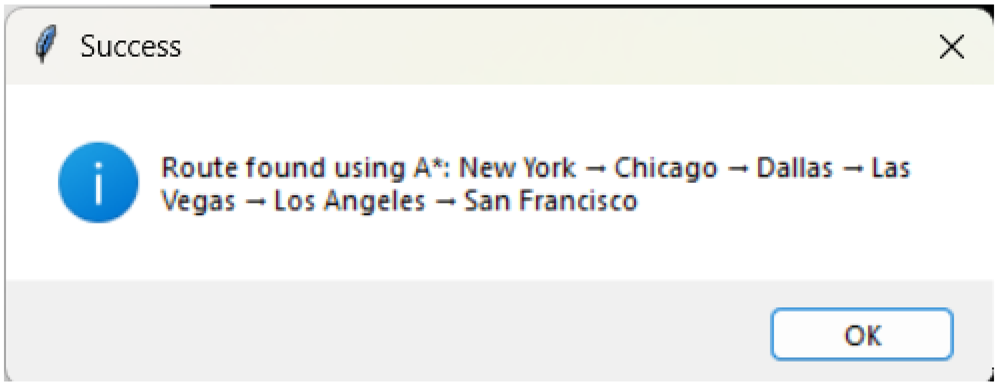
DFS:-


Success

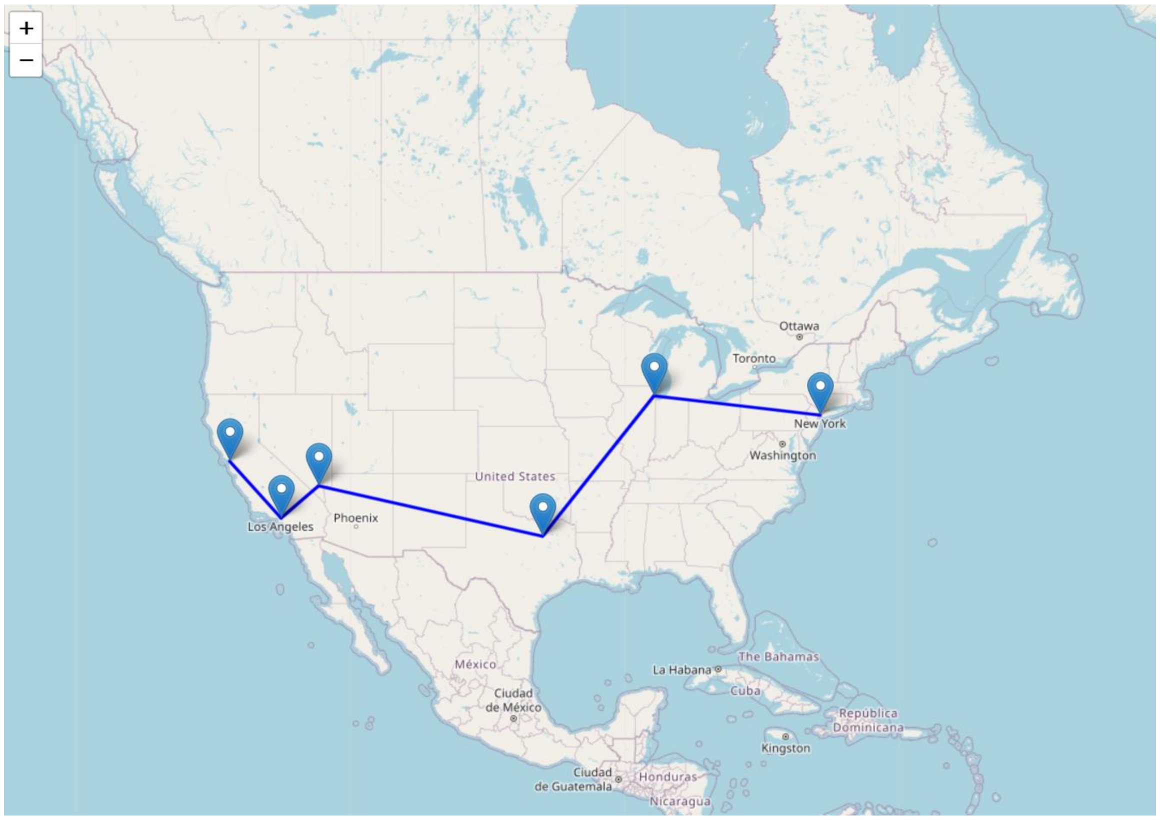
Route found using DFS: New York → Miami → Houston → Dallas → Las Vegas → Denver → Seattle → San Francisco



A*:-


Success

Route found using A*: New York → Chicago → Dallas → Las Vegas → Los Angeles → San Francisco



8. Conclusion

The Route Finder project simplifies the problem of finding paths between cities using AI search algorithms. It visually shows paths with maps and is helpful in educational tools, logistics, and navigation system prototypes.

9. Learning Outcome

- Application of AI search algorithms in real life.

- Practical use of GUI and mapping libraries in Python.

- Visualization of graph traversal and heuristic-based decision making.

- Integration of backend logic with frontend interface.

10. Future Scope

- Add live traffic data and travel time estimation.

- Use real map APIs like Google Maps.

- Support more cities and custom map layers.

- Allow saving/exporting routes.

GitHub Link: https://github.com/anujdhiman28/Route-Finder.git