Academic Year: 2023-2024(Odd Semester)

Department: Electronics and Communication Engineering (ECE)

V Semester Electronics and Computer Engineering (EAC)

19CSE459 – Advanced Algorithms and Analysis Project

Path Finding Visualizer for Drone Delivery

Team Members:

Konduru Praveen Karthik – BL.EN.U4EAC21035

Taduvai Satvik Gupta – BL.EN.U4EAC21075

Aditya Thelu – BL.EN.U4EAC21076

Introduction:

In real-time logistics and transportation, efficient route planning is critical to optimize resources and reduce delivery time. This project aims to develop a comprehensive pathfinding visualizer specifically tailored for drone delivery services. By integrating various algorithms and visualization techniques, this tool assists in selecting optimal delivery locations, calculating distances, and determining the most efficient route for drone navigation. This project amalgamates critical algorithms and visualization techniques to create a Visualizer that assists in optimizing drone delivery routes, contributing to the advancement of efficient and intelligent logistics solutions.

Problem Statement: Optimize the Drone's route to visit selected locations.

User Input:

- Number of Locations: Specify the number of locations to visit.
- Location Details: Enter names, coordinates, weights, and profits for each location.
- Starting Location: Choose the initial position of the drone.
- Weight Capacity: Define the drone's weight-carrying limit.

Constraints:

- Limited weight capacity and the need to maximize profit.

Algorithms Used:

1) 0/1 Knapsack:

- To select the locations based on weight and profit capacity.
- Dynamic programming to optimize the selection process.

2) Euclidean Distance:

- Establish pairwise distances between selected locations.
- Displayed the distance matrix for better understanding.

3) Travelling Salesman Problem:

- To find the optimal path to minimize total distance.
- Used the Brute-force method for simplicity.
- Highlighted the optimal path on the directed graph.

Knapsack Algorithm:

- The Knapsack Algorithm is a dynamic programming technique used for optimization problems where items have associated weights and values.
- The selected locations become the subset and are further used in the project, specifically for calculating the distance matrix and optimizing the drone's travel path using the Traveling Salesman Problem (TSP) algorithm.
- Time Complexity: O(N*W)

  N: No. of Items Available

  W: Capacity of Drone

Euclidean Distance and Distance Matrix:

- Euclidean distance is used to calculate the distance between two points in a two-dimensional space.
- Euclidean distance is utilized to construct the distance matrix, which is later used in the Traveling Salesman Problem (TSP) and for displaying the distances in the visual representation of the optimal path.
- The distance matrix is essential for solving the Traveling Salesman Problem using the brute-force method. It provides the distances between all locations, enabling the algorithm to find the optimal path.

- Time Complexity: O(N^2)

  N: No. of points obtained after applying Knapsack

Travelling Salesman Problem:

- The TSP is employed to solve the challenge of finding the shortest possible path that connects all selected locations and returns to the starting point.
- The TSP algorithm outputs the optimal path that the drone should follow to minimize the total distance traveled. This path ensures that each selected location is visited exactly once, and the drone returns to the starting point.
- In the context of the scenario, the TSP helps optimize the drone's travel path, ensuring efficient coverage of all selected locations. This contributes to minimizing the overall travel time and enhancing the drone's effectiveness in transporting materials or collecting data.
- Time Complexity: O(N!)

  N: No. of points obtained after applying Knapsack

Code:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.spatial import distance
from scipy.optimize import linear_sum_assignment
import itertools


# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return distance.euclidean(point1, point2)

# Knapsack Algorithm considering both weight and profit
def knapsack(items, capacity):
    n = len(items)
    dp = [[0 for _ in range(int(capacity) + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
```

```
        for w in range(1, int(capacity) + 1):
            weight, profit, coordinates = items[i - 1][1], items[i - 1][0], items[i - 1][2]
            if weight <= w:
                dp[i][w] = max(dp[i - 1][w], profit + dp[i - 1][w - int(weight)])
            else:
                dp[i][w] = dp[i - 1][w]

    selected_items = []
    w = int(capacity)
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:
            selected_items.append(items[i - 1])
            w -= int(items[i - 1][1])

    return selected_items

def distance_matrix(point_dict):
    point_names = list(point_dict.keys())
    num_points = len(point_names)
    matrix = np.zeros((num_points, num_points))

    for i in range(num_points):
        for j in range(num_points):
            matrix[i, j] = euclidean_distance(point_dict[point_names[i]],
point_dict[point_names[j]])

    return matrix, point_names

# Function to calculate total distance of the path
def calculate_path_distance(graph, path):
    distance = 0
    for i in range(len(path) - 1):
        distance += graph[path[i]][path[i + 1]]
    distance += graph[path[-1]][path[0]]  # Return to the starting point
    return distance

# Function to solve TSP using brute-force method
def tsp_brute_force(graph, start):
    num_nodes = len(graph)
    nodes = list(range(num_nodes))
    nodes.remove(start)

    min_distance = float('inf')
    min_path = None

    for perm in itertools.permutations(nodes):
```

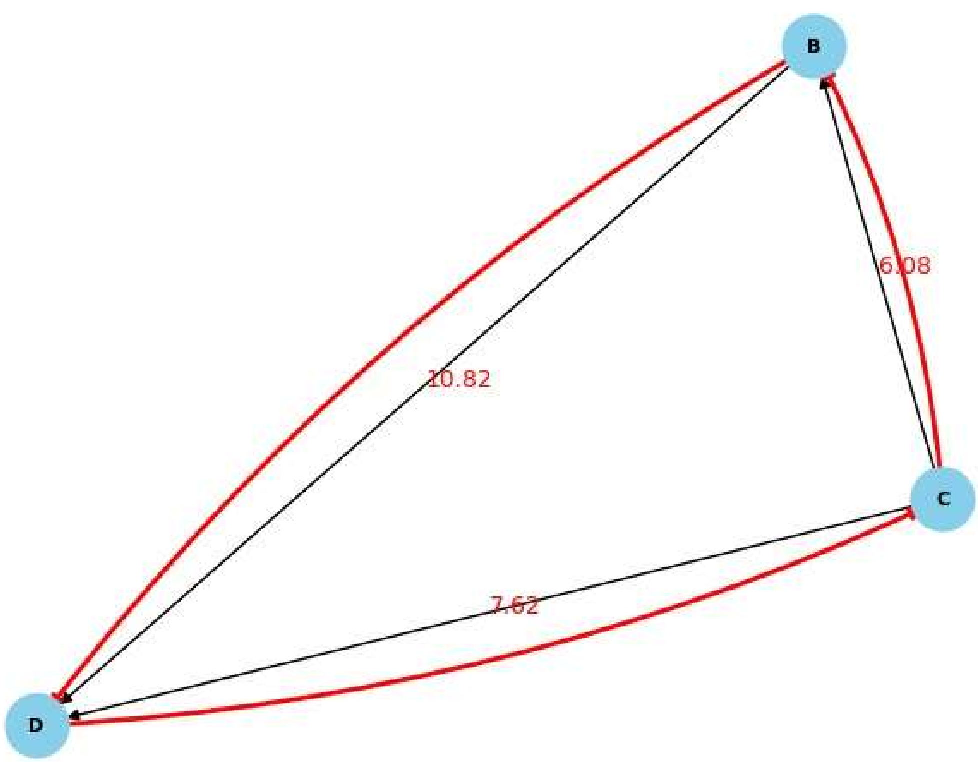
```
        path = [start] + list(perm)
        distance = calculate_path_distance(graph, path)
        if distance < min_distance:
            min_distance = distance
            min_path = path

    return min_path, min_distance


def visualize_optimal_path(graph, optimal_path, point_names,locations):
    G = nx.DiGraph()  # Use DiGraph to represent a directed graph

    for i, name in enumerate(point_names):
        G.add_node(name, pos=(locations[name][0], locations[name][1]))

    for i in range(len(graph)):
        for j in range(len(graph[i])):
            if i < j:
                G.add_edge(point_names[i], point_names[j], weight=graph[i][j])

    pos = nx.get_node_attributes(G, 'pos')

    # Draw the graph
    nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700,
node_color='skyblue', font_color='black', font_size=8)

    # Highlight the optimal path
    edges = [(point_names[optimal_path[i]], point_names[optimal_path[i + 1]]) for i in
range(len(optimal_path) - 1)]
    edges.append((point_names[optimal_path[-1]], point_names[optimal_path[0]]))  # Connect
back to the starting point

    # Draw directed edges for the optimal path
    nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='r', width=2,
connectionstyle='arc3,rad=0.1')

    # Display distances for the red arcs
    for edge in edges:
        distance = graph[point_names.index(edge[0])][point_names.index(edge[1])]
        plt.text((pos[edge[0]][0] + pos[edge[1]][0]) / 2, (pos[edge[0]][1] + pos[edge[1]][1]) / 2,
f'{distance:.2f}', color='red')

    # Display the plot
    plt.show()
```

```
# Example Input
def main():
    num_locations = int(input("Enter the number of locations: "))
    locations = {}
    sel_loc = {}
    items = []
    for i in range(num_locations):
        name = input(f"Enter name for location {i + 1}: ")
        coordinates = input(f"Enter coordinates for {name} (x y): ").split()
        if len(coordinates) != 2:
            print("Invalid coordinates. Please enter two space-separated values.")
            exit(1)
        x, y = map(float, coordinates)
        weight = float(input(f"Enter weight for {name}: "))
        profit = float(input(f"Enter profit for {name}: "))
        locations[name] = (x, y)
        items.append((profit, weight, (x, y)))

    start_location = input("Enter the starting location: ")

    if start_location not in locations:
        print("Start location not found in provided locations. Adding start location to the
locations.")
        coordinates = input(f"Enter coordinates for {start_location} (x y): ").split()
        if len(coordinates) != 2:
            print("Invalid coordinates. Please enter two space-separated values.")
            exit(1)
        x, y = map(float, coordinates)
        locations[start_location] = (x, y)

    # Knapsack to find selected locations based on weight and profit capacity
    capacity = float(input("Enter the weight capacity of the drone: "))
    selected_items = knapsack(items, capacity)

    selected_locations = [name for profit, weight, coord in selected_items for name,
coordinates in locations.items() if
                          coordinates == coord]

    print("Selected Coordinates:")
    for loc in selected_locations:
        print(f"{loc}: {locations[loc]}")
        sel_loc[loc] = locations[loc]


    # Add start location to the graph if it's not already present
    if start_location not in selected_locations:
```

```
        selected_locations += [start_location]  # Using the + operator for concatenation instead
of append

    for loc in selected_locations:
        sel_loc[loc] = locations[loc]


    # Calculate distance matrix
    dist_matrix, point_names = distance_matrix(sel_loc)

    # Display the distance matrix
    print("Distance Matrix:")
    print("    ", end="")
    for name in point_names:
        print(f"{name:4}", end=" ")
    print()

    for i, row in enumerate(dist_matrix):
        print(f"{point_names[i]} |", end=" ")
        for distance in row:
            print(f"{distance:4.2f}", end=" ")
        print()

    # Convert labels to indices for ease of computation
    label_to_index = {name: i for i, name in enumerate(point_names)}
    start_index = label_to_index[start_location]

    # Convert the labels in the graph to indices
    graph_indices = [[dist_matrix[label_to_index[label_i]][label_to_index[label_j]] for label_j
in point_names] for label_i in point_names]

    # Solve TSP using brute-force method
    optimal_path, min_distance = tsp_brute_force(graph_indices, start_index)

    # Convert indices back to labels for display
    optimal_path_labels = [point_names[i] for i in optimal_path]

    # Display the result
    print("Optimal Path:", " -> ".join(optimal_path_labels), " (Total Distance:", min_distance,
")")

    visualize_optimal_path(graph_indices, optimal_path, point_names,locations)
if __name__== "__main__":
    main()
```

Result:

```
Enter the number of locations: 3
Enter name for location 1: A
Enter coordinates for A (x y): 4 3
Enter weight for A: 5
Enter profit for A: 1
Enter name for location 2: B
Enter coordinates for B (x y): 6 9
Enter weight for B: 4
Enter profit for B: 6
Enter name for location 3: C
Enter coordinates for C (x y): 7 3
Enter weight for C: 4
Enter profit for C: 8
Enter the starting location: D
Start location not found in provided locations. Adding start location to the locations.
Enter coordinates for D (x y): 0 0
Enter the weight capacity of the drone: 10
Selected Coordinates:
C: (7.0, 3.0)
B: (6.0, 9.0)
Distance Matrix:
    C    B    D
C | 0.00 6.08 7.62
B | 6.08 0.00 10.82
D | 7.62 10.82 0.00
Optimal Path: D -> C -> B  (Total Distance: 24.515189462554098 )
```



Applications:

- **Drone Delivery Services:** Companies engaged in logistics and e-commerce can employ this tool to plan efficient routes for drone deliveries. It aids in optimizing delivery schedules, reducing delivery times, and enhancing last-mile delivery efficiency.
- **Warehouse Management:** For large warehouses or distribution centers, this tool can assist in optimizing drone routes for inventory management, picking items, and organizing deliveries within the facility.
- **Medical Supply Delivery:** Healthcare facilities, especially in remote areas or during emergencies, can benefit from optimized drone routes for delivering medical supplies, vaccines, or life-saving equipment promptly.
- **Emergency Response:** During natural disasters or emergencies, drones equipped with supplies or equipment can navigate optimally planned routes to deliver aid and assistance to affected areas efficiently.

Future Scope:

- Time Complexity can be reduced using Better Algorithms.
- A proper User Interface can be developed using HTML and CSS.
- By Integrating it with G Maps we can make it to use in the Realtime.
- Integrating machine learning models to analyze historical delivery data and environmental factors, allowing the system to learn and improve route optimizations continuously.