Programming Assignment-1

Name: Dyuti Dasmahapatra **Enrollment no : 210424**

Write a program to implement the following:

i. Build a graph (dynamically) consisting of nodes/vertices (representing cities in random (x,y) locations in a 2D Cartesian plane). Randomly assign cardinality (connections/edges between two nodes with randomly initialized positive weights).
This graph should contain at least 10 nodes and 15 connections/edges.

Input:

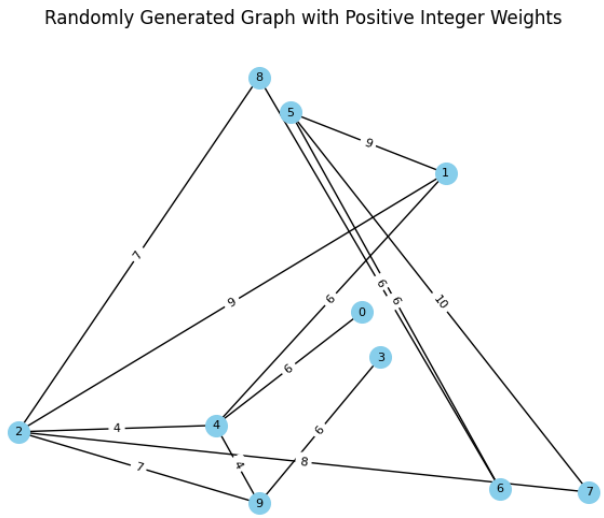
```
G = nx.Graph()

num_nodes = 10
for node_id in range(num_nodes):
    x = random.uniform(0, 12)
    y = random.uniform(0, 12)
    G.add_node(node_id, x=x, y=y)

num_edges = 15
for _ in range(num_edges):
    node1, node2 = random.sample(G.nodes(), 2)
    weight = random.randint(1, 10)
    G.add_edge(node1, node2, weight=weight)

# Plot the graph
pos = {node: (G.nodes[node]['x'], G.nodes[node]['y']) for node in G.nodes()}
nx.draw(G, pos, with_labels=True, node_size=200, node_color='skyblue', font_size=8,
font_color='black')
edge_labels = {(edge[0], edge[1]): G.edges[edge]['weight'] for edge in G.edges()}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)
plt.title('Randomly Generated Graph with Positive Integer Weights')
plt.show()
```

Output:



ii. Choose the Start and the Goal nodes.

Input:

```
nodes = list(G.nodes())

start_node = random.choice(nodes)
goal_node = random.choice(nodes)

while start_node == goal_node:
    goal_node = random.choice(nodes)

print("Start Node:", start_node)
print("Goal Node:", goal_node)
```

Output:

```
Start Node: 3
Goal Node: 6
```

iii. Compute admission heuristic using straight-line-distance as a measure of heuristic. Compute the heuristic values for each node in the graph.

Input:

```
def euclidean_distance(node1, node2):
    x1, y1 = pos[node1]
    x2, y2 = pos[node2]
    return round(math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2))

# Calculate the actual costs (sum of weights) from the start node to each node
def calculate_actual_costs(graph, start):
    actual_costs = {}
    for node in graph.nodes():
        if node == start:
            actual_costs[node] = 0
        else:
            path = nx.shortest_path(graph, source=start, target=node, weight='weight')
            cost = sum([graph[path[i]][path[i+1]]['weight'] for i in range(len(path)-1)])

            if cost == 0:
                cost = float('inf')

            actual_costs[node] = cost
    return actual_costs

# Calculate actual costs
actual_costs = calculate_actual_costs(G, start_node)

# Calculate Euclidean heuristics and admissibility
euclidean_heuristics = {}
data = []

for node in G.nodes():
    euclidean_heuristic = euclidean_distance(node, goal_node)
    euclidean_heuristics[node] = euclidean_heuristic
    admissible = actual_costs[node] >= euclidean_heuristic

    data.append([node, euclidean_heuristic, actual_costs[node], admissible])

# Create a DataFrame and display it using tabulate
```

```
df = pd.DataFrame(data, columns=["Node", "Euclidean Heuristic", "Actual Cost",
"Admissible"])
df["Actual Cost"].replace(0, float('inf'), inplace=True)
df.loc[df["Actual Cost"] == float('inf'), "Admissible"] = True
table = tabulate(df, headers='keys', tablefmt='grid')
print(table)
```

Output:

	Node	Euclidean Heuristic	Actual Cost	Admissible
0	0	4	16	True
1	1	7	16	True
2	2	8	13	True
3	3	3	inf	True
4	4	5	10	True
5	5	8	25	True
6	6	0	26	True
7	7	2	21	True
8	8	9	20	True
9	9	4	6	True

iv. Apply A* algorithm to compute the optimal distance from Start to the Goal node. Show the complete solution path (all the nodes in the sequence from Start to Goal in the optimal path). Show the optimal path cost.

Input:

```
open_list = []
closed_set = set()

#A* algorithm
def custom_astar(graph, start, goal, heuristic):
    open_list = [(0, start)]
    came_from = {}
    g_score = {node: float('inf') for node in graph.nodes()}
    g_score[start] = 0
```

```
    while open_list:
        _, current_node = open_list.pop(0)

        if current_node == goal:
            path = [current_node]
            while current_node in came_from:
                current_node = came_from[current_node]
                path.append(current_node)
            path.reverse()
            return path

        closed_set.add(current_node)

        for neighbor in graph.neighbors(current_node):
            if neighbor in closed_set:
                continue

            tentative_g_score = g_score[current_node] + graph[current_node][neighbor]['weight']

            if neighbor not in [node for _, node in open_list] or tentative_g_score <
g_score[neighbor]:
                came_from[neighbor] = current_node
                g_score[neighbor] = tentative_g_score
                f_score = g_score[neighbor] + heuristic(neighbor, goal)
                open_list.append((f_score, neighbor))
                open_list.sort(key=lambda x: x[0])

    return None

optimal_path = custom_astar(G, start_node, goal_node, heuristic=euclidean_distance)
total_cost = sum([G[optimal_path[i]][optimal_path[i + 1]]['weight'] for i in
range(len(optimal_path) - 1)])

print("Complete Solution Path:", optimal_path)
print("Optimal Path Cost:", total_cost)
```

Output:

```
Complete Solution Path: [3, 9, 2, 8, 6]
Optimal Path Cost: 26
```

v. Show the state of the Open and Closed lists.

Input:

```
print("\nState of Open List:")
print(open_list)

print("\nState of Closed Set:")
print(closed_set)
```

Output:

```
State of Open List:
[]

State of Closed Set:
{0, 1, 2, 3, 4, 7, 8, 9}
```