DATA STRUCTURES & ALGORITHMS

TRAVELLING SALESMAN'S PROBLEM ASSIGNMENT

GROUP Q

GROUP MEMBER	REGISTRATION NUMBER
MWANJE SIMON PETER	24/U/07633/PS
ATIM REINA MICHAL	24/U/03845/PS
KAJJABWANGU JOSEPH LESTER	24/U/05090/PS
MUTEBI KENNEDY	24/U/07472/PS
AYESIGA MALCOM	24/U/04104/PS

GITHUB REPOSITORY LINK: *https://github.com/lolesterrr/Travelling_Salesman*

Language of choice:PYTHON

External libraries used include;

- *Numpy*
- *matplotlib*

1. GRAPH REPRESENTATION & DATA STRUCTURE

This Travelling Salesman's Problem involves 7 cities with edges representing the distances between them. To efficiently store the graph, we used the **ADJACENCY MATRIX**.

We improvised and used actual city names for easier understanding.

Adjacency Matrix

```
# Define city names and corresponding labels
city_names = ["Austin", "Birmingham", "Chicago", "Denver", "Edinburgh", "Frankfurt", "Glasgow"]
labels = ["A", "B", "C", "D", "E", "F", "G"]

# Define adjacency matrix (distances) with np.inf for no direct route
distances = np.array([
[np.inf,12,10,np.inf,np.inf,np.inf,12],#city 1
[12,np.inf,8,np.inf,np.inf,np.inf,np.inf],#city 2
[10,8,np.inf,11,3,np.inf,9],#city 3
[np.inf,12,11,np.inf,11,10,np.inf],#city 4
[np.inf,np.inf,3,11,np.inf,6,7],#city 5
[np.inf,np.inf,np.inf,10,6,np.inf,9],#city 6
[12,np.inf,9,np.inf,7,9,np.inf]#city 7
])

# Convert to Pandas DataFrame for better visualization
adj_matrix = pd.DataFrame(distances, index=city_names, columns=labels)
print(adj_matrix)
```

The above matrix shows the distances between cities. The graph is represented as a list of lists where the index of the outer list represents the city, and the inner list represents the distances to other cities. np.inf indicates no direct connection/route between two cities.

Justification for Using Adjacency Matrix

- **Fast lookups:** Checking distances between cities takes **O(1)** time.
- **Space efficiency:** Works well for a small number of cities (**O(n^2) space complexity**).
- **Direct representation:** Matrix format aligns naturally with TSP formulation.

2. CLASSICAL TSP SOLUTION

USING DYNAMIC PROGRAMMING (HELD-KARP ALGORITHM)

We used **Dynamic Programming** as our classical solution, which solves the TSP optimally in **O(2^n * n) time**. This approach uses memoization to speed up recursive calls without redundant calculations.

Our code implements a solution to the Traveling Salesman Problem (TSP) using dynamic programming. The TSP aims to find the shortest possible route that visits a set of cities exactly once and returns to the starting city. The algorithm begins by initializing a memoization table, dp, to store the minimum costs for visiting subsets of cities

```
dp = np.full((all_sets, n), np.inf)
dp[1][0] = 0 # Starting at city 0 (Austin)
```

It then iterates over all possible subsets of cities using bit manipulation, updating the table with the minimum cost to reach each subset by considering all possible transitions between cities:

```
for subset in range(all_sets):
for current in range(n):
if not (subset & (1 << current)): # If current city is not in subset
continue
# Check possible previous cities
for prev in range(n):
if prev == current or not (subset & (1 << prev)):
continue
prev_subset = subset & ~(1 << current) # Remove current from subset
dp[subset][current] = min(
dp[subset][current],
dp[prev_subset][prev] + matrix[prev][current]
)
```

Once the table is filled, the algorithm reconstructs the optimal path by tracing back through the table to determine the sequence of cities that yields the shortest route:

```
path = [0] # Start at city 0
subset = final_state
current = last_index
while current != 0:
path.append(current)
prev_subset = subset & ~(1 << current)
# Find the previous city in optimal path
for prev in range(n):
if subset & (1 << prev) and dp[subset][current] == dp[prev_subset][prev] + matrix[prev][current]:
current = prev
subset = prev_subset
break

path.append(0) # Return to start city
```

The final output includes the optimal route and its total cost, providing an efficient solution for small to moderately sized TSP instances.

Sample Output/ Final Tour basing off current parameters used in the Matrix

Optimal Route: Austin → Birmingham → Denver → Frankfurt → Glasgow → Edinburgh → Chicago → Austin

Minimum Cost: 63.0

3. SELF-ORGANIZING MAP (SOM) APPROACH TO TSP

A Self-Organizing Map (SOM) is a type of artificial neural network that approximates the shortest TSP route by learning city locations.

Our code implements a Self-Organizing Map (SOM) to approximate a solution for the Traveling Salesman Problem (TSP). It begins by initializing city coordinates and SOM nodes:

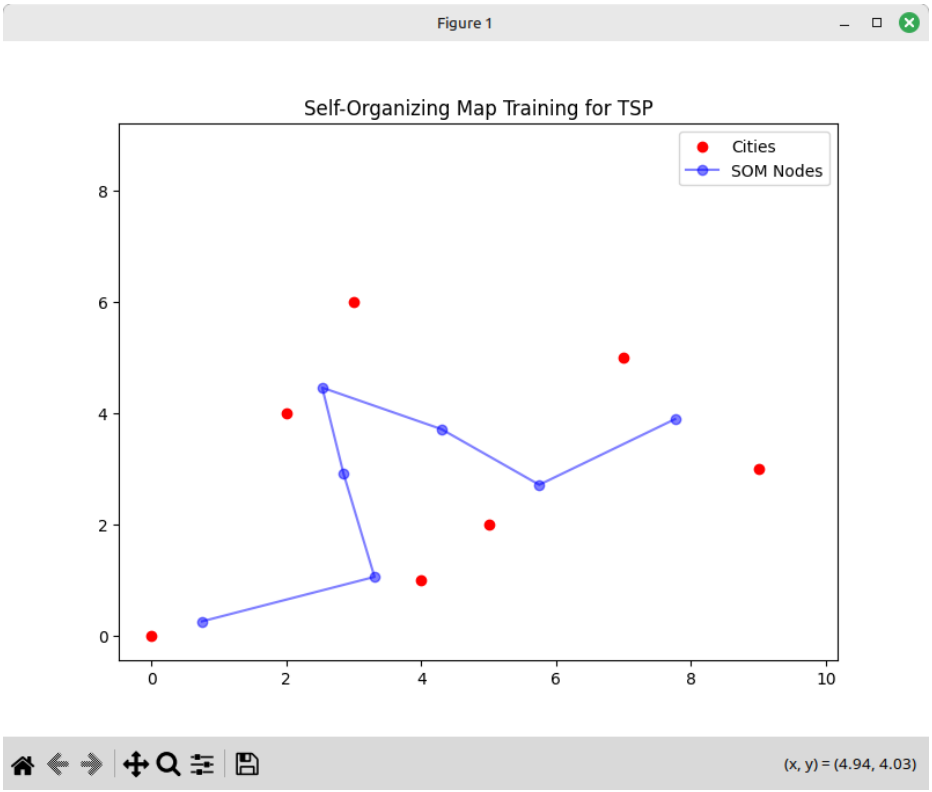
```
city_coords = {
"City1": (0, 0), # Starting city
"City2": (2, 4),
"City3": (5, 2),
"City4": (7, 5),
"City5": (3, 6),
"City6": (9, 3),
"City7": (4, 1)
}


som_nodes = np.random.rand(num_nodes, 2) * 10 # Scale within map
```

The SOM nodes are adjusted iteratively using an animation function that updates their positions based on proximity to randomly selected cities:

```
def update(frame):
global som_nodes, learning_rate, radius

city = coordinates[np.random.randint(len(city_coords))] # Random city selection
distances = np.linalg.norm(som_nodes - city, axis=1) # Distance to all nodes
winner_idx = np.argmin(distances) # Closest SOM node

# Adjust nodes in the neighborhood
for j in range(num_nodes):
distance_factor = np.exp(-np.linalg.norm(j - winner_idx) / radius)
som_nodes[j] += learning_rate * distance_factor * (city - som_nodes[j])

# Update learning rate and radius
learning_rate *= decay
radius *= decay
```

After training, the optimal tour is extracted by tracing through the SOM nodes

The final output is the approximated TSP tour using SOM.(full implementation within code repository)



Challenges with the SOM Approach

- Parameter tuning for learning rate and radius is crucial for achieving good results.
- The SOM approach may not always converge to the optimal solution, especially for larger instances of the TSP.

Comparison Report of TSP Solutions both Dynammic Programming and SOM Approach

The Traveling Salesman Problem (TSP) is a complex optimization challenge where the objective is to find the shortest possible route visiting all given cities exactly once and returning to the starting point. In this assignment, we explored two approaches: the exact **Held-Karp Dynamic Programming (DP) method** and the approximate **Self-Organizing Map (SOM) neural network approach**.

The **Held-Karp Algorithm**, based on dynamic programming, systematically computes the optimal path by storing intermediate results and avoiding redundant calculations. This method guarantees the shortest path but has an exponential time complexity of O(2^n * n), making it computationally expensive for large problem instances.

The **Self-Organizing Map (SOM) Approach**, on the other hand, is a heuristic method that attempts to approximate a good solution rather than the absolute best. It models neurons that adjust their positions based on input cities over multiple iterations. The learning process involves winner selection, adaptation, and neighborhood decay, refining the estimated path over time. However, since it does not explore all possible routes, it does not guarantee the most efficient tour.

Computational Complexity and Performance

One key factor in choosing between these approaches is computational efficiency. The **Held-Karp DP algorithm** requires O(2^n * n) time, which makes it infeasible for large-scale problems due to its

exponential growth. However, it provides an exact solution, which is crucial in applications requiring precise optimization.

Conversely, the **SOM approach** runs in O(n * iterations), where iterations is the number of training steps. This makes it significantly more scalable for large graphs but sacrifices exactness. The trade-off is between computational feasibility and solution optimality, depending on the problem's constraints and available resources.

4. ANALYSIS AND COMPARISON

a) ROUTE QUALITY

Algorithm	Solution Type	Accuracy
Dynamic Programming	Exact	Finds the shortest route
SOM Approach	Approximate	May be suboptimal

b) COMPLEXITY

Algorithm	Time Complexity	Space Complexity
Dynamic Programming	O(2^n * n)	O(2^n * n)
SOM Approach	O(n * iterations)	O(n)

c) PRACTICAL CONSIDERATIONS

- **Its recommended to use DP** for **small graphs (n ≤ 20)**.
- **Use SOM** for **large graphs (n > 20)** where approximation is acceptable.