DATA STRUCTURE ASSIGNMENT

GROUP U

STUDENT NAME	STUDENT NUMBER	REGESTRATION NUMBER
BARIGYE ROMEO	2400704269	24/U/04269/PS
OKEDI ISMAIL MUSA	2400710601	24/U/10601/EVE
AINEBYOONA DATIVAH	2400702898	24/U/02898/EVE
NANTALE CECILIA	2400724555	24/U/24555/PS
KIGOZI ALLAN	2400725792	24/U/25792/PS
WALERA EMMANUEL	2400701410	24/U/1410

Link to the Repository : https://github.com/OKEDI820/Data-Structure-Assignment-Group-U

1. TSP Representation and Data Structures
   Deliverables:
   • A short description of your chosen data structure (code snippet or pseudocode).
   The chosen data structure is Adjacency Matrix

```
graph = [
    [0, 12, 10, 8, 12, 3, 9],  Distances from City 1
    [12, 0, 12, 11, 6, 7, 9],    Distances from City 2
    [10, 12, 0, 11, 10, 11, 6],  Distances from City 3
    [8, 11, 11, 0, 7, 9, 12],  Distances from City 4
    [12, 6, 10, 7, 0, 9, 10],   Distances from City 5
    [3, 7, 11, 9, 9, 0, 11],     Distances from City 6
    [9, 9, 6, 12, 10, 11, 0]     Distances from City 7
]
```

An adjacency matrix is a 2D array where each row and column represent a city.

The value at (i, j) represents the distance between city i and city j.

If no direct route exists, the value is infinity.

• A brief explanation (1–2 paragraphs) justifying your choice of representation

Efficiency: O(1) Lookup Time: The adjacency matrix allows for constant time (O(1)) lookup of distances between any two cities. This is crucial for TSP algorithms that frequently query distances to calculate the total travel distance and determine the next city to visit. Example: To find the distance between City 1 and City 3, you simply access graph[0][2], which returns 10 units**. Simplicity**: Straightforward Implementation, The adjacency matrix is easy to implement and understand. Each row and column correspond to a city, and the value at the intersection represents the distance between those cities. Example: The first row [0, 12, 10, 8, 12, 3, 9] represents the distances from City 1 to all other cities. **Bidirectional Edges**: Symmetric Representation: The matrix naturally supports symmetric distances, meaning the distance from city i to city j is the same as from city j to city i. This is important for undirected graphs like the one used in TSP. Example: The distance from City 1 to City 2 is 12 units, and the distance from City 2 to City 1 is also 12 units, represented by graph[0][1] and graph[1][0] respectively.

Direct Access in Constant Time (O(1)):Since the matrix is stored in memory as a 2D array, accessing any element graph[i][j] takes O(1) time. This is much faster than searching through an adjacency list, which may require O(n) time to find a specific connection. **No Need for Iteration**: In an adjacency list, to find the distance between two cities, you may need to traverse a linked list. In an adjacency matrix, the value is already stored at a fixed index, eliminating unnecessary computation.

2. **Self-Organizing Map (SOM) Approach for TSP**

The Traveling Salesman Problem (TSP) seeks the shortest route that visits a set of cities exactly once and returns to the origin. While classical methods like dynamic programming guarantee optimal solutions, their exponential time complexity makes them impractical for large instances. The Self-Organizing Map (SOM), a type of unsupervised neural network, offers a heuristic approach to approximate TSP solutions efficiently. SOMs excel at preserving topological relationships, making them suitable for organizing cities into a coherent route.

A Self-Organizing Map (SOM) is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional representation of the input space. To adapt an SOM to solve the Traveling Salesman Problem (TSP), the following steps are typically followed:

1. **Initializing Neurons**: Neurons are initialized randomly or in a circular layout to represent potential positions of cities.
2. **Representing Cities**: Cities are represented as points in a 2D space.
3. **Neighborhood Function**: A neighborhood function defines the influence of a winning neuron on its neighbors. This function typically decays over time.
4. **Learning Rate**: The learning rate controls the adjustment of neuron positions and also decays over time.
5. **Training Loop**: The SOM is trained iteratively by presenting cities to the network, finding the closest neuron (winner), and updating the winner and its neighbors.

<u>Implementation of the SOM approach to solve the TSP</u>

Parameter Tuning

-Learning Rate: The initial learning rate and its decay schedule significantly impact the convergence of the SOM. A high learning rate may cause instability, while a low learning rate may slow down convergence.

Neighborhood Radius: The initial neighborhood radius and its decay schedule also affect the quality of the solution. A large radius may lead to sub-optimal routes, while a small radius may cause the SOM to converge prematurely.

Sub-optimal **Convergence**

-Local Minima: The SOM may converge to a sub-optimal route if the parameters decay too quickly or if the initial neuron positions are not well-distributed.

Training Iterations: The number of training iterations (epochs) needs to be sufficient to allow the SOM to explore the solution space and converge to a good solution.

How SOM Works for TSP

- A Self-Organizing Map (SOM) consists of a ring of neurons, where:
- Each neuron represents a potential position in the optimal TSP route.
- Neurons are initialized randomly and updated over multiple iterations.
- The network learns by adjusting neuron positions toward city locations, forming an approximate tour.

Challenges and Limitations

1. Parameter Sensitivity:

- The learning rate and neighborhood decay rate strongly affect performance.
- Poor tuning can lead to sub-optimal convergence.

2. No Guarantee of Optimality

- Unlike Dynamic Programming, SOM does not always find the shortest route.
- It provides an approximation that is often good enough for large-scale problems.

3. Computational Cost

- SOM is faster than exact algorithms but still requires multiple iterations for refinement.

Advantages of Using SOM for TSP

Scalability – Works well for large n (100+ cities).
Faster than exact methods – Approximate solutions in O(n × epochs) time.
Inspired by biological learning – Uses unsupervised learning to refine solutions.

When to Use SOM for TSP?

For small problems (n < 15), exact methods (e.g., Dynamic Programming) are preferable.

For large-scale problems (n > 50), SOM provides a good balance between speed and solution quality.

4.Analysis and Comparison
• A short report (1–2 pages) containing:
Comparison of route distances from both methods.
The exact approach (Dynamic Programming) is shorter than the SOM approach.
The SOM route is close but sub-optimal due to its heuristic nature.
The Traveling Salesman Problem (TSP) was solved using two different methods that is Classical Approach (Dynamic Programming - DP) and Self-Organizing Map (SOM - Neural Network). The results are compared based on: Route Distance (shorter is better), Computational Complexity (how fast the method runs), Practical Usability (which approach is better for large datasets)

Final Routes Obtained

Dynamic Programming finds the exact shortest path (0, 5, 1, 4, 3, 2, 6, 0).

SOM finds a near-optimal solution (≈68 units), but not always the best.

SOM is significantly faster for large-scale problems, while DP is better for small graphs.

Time/complexity analysis of both approaches.
1.Classical TSP Solution (Dynamic Programming - DP)

Time Complexity: O $(n^2 \times 2^n)$

Space Complexity: O $(n \times 2^n)$

Advantage: Finds the exact shortest path.

2.Self-Organizing Map (SOM Approach)

Time Complexity: O (n × epochs) (where epochs is the number of training iterations).

Space Complexity: O (n) (only neuron positions are stored).

Advantage: Works well for large datasets ($n > 50$).

Disadvantage: Does not guarantee the optimal route, only an approximation.

Disadvantage: Exponential growth makes it impractical for $n > 20$.

Discussion on trade-offs between classical and heuristic methods.
When to Use the Dynamic Programming Approach

- ➢ Small-scale problems ($n < 15$) where finding the exact optimal path is important.
- ➢ Situations where precision is required (e.g., circuit board design, vehicle routing in a small city).

When to Use the SOM Approach

- Large-scale problems (n > 50) where an approximate solution is acceptable.
- Real-time applications like drone path optimization, delivery routing, logistics, AI-driven scheduling.
- Problems where time efficiency matters more than getting the absolute best path.

Suggestions for improvements or extensions.

1. Hybrid Approach (SOM + Local Optimization)

Use SOM for an initial approximation, then apply 2-opt local search to refine the route.

This balances speed and accuracy, reducing errors in the SOM route.

2. Alternative Neighborhood Function

Instead of Gaussian decay, try Inverse Distance Weighting (IDW) for smoother convergence.

This ensures neurons adjust proportionally to distance, improving stability.

3. Advanced Heuristics (Ant Colony, Simulated Annealing)

Ant Colony Optimization (ACO) mimics real-world route optimization by modeling pheromone trails.

Simulated Annealing (SA) can further improve the SOM route by fine-tuning neuron placements.

Conclusion

The Dynamic Programming (DP) approach guarantees the shortest route but is slow for large problems.

The Self-Organizing Map (SOM) is faster but gives an approximate solution.

A hybrid approach (SOM + Local Search) may offer the best trade-off between speed and accuracy.

For small problems ($n < 15$), Dynamic Programming is best.

For large problems ($n > 50$), SOM-based methods are more practical.