**TSP REPRESENTATION AND DATA STRUCTURES**

**Brief Explanation**

We use an **adjacency matrix** because it provides **efficient distance lookups** between cities. Here’s why:

✅ **Fast Access:**

* Since it's a **2D list**, we can instantly retrieve the distance between any two cities in **O(1) time** using graph[i][j].
* Example: To get the distance between **City 2 and City 5**, we simply check graph[1][4] (since indices start at 0).

✅ **Simple Implementation:**

* The matrix is **easy to construct** and understand. Each row and column directly represents a city.
* There's **no need for complex data structures** like linked lists or hash maps.

✅ **Good for Small Graphs:**

* Since the given TSP problem has **only 7 cities**, using a **7×7 matrix (49 elements)** is **memory-efficient**.
* For **larger graphs**, adjacency lists would be better, but for **small TSP instances, the matrix is ideal**.

### ****Traveling Salesman Problem (TSP) Objective****

The **goal of the Traveling Salesman Problem (TSP)** is to find the **shortest possible route** that:  
1️⃣ **Starts from a given city** (City 1 in our case).  
2️⃣ **Visits each of the other cities exactly once** (No revisiting).  
3️⃣ **Returns to the starting city** after visiting all cities.  
4️⃣ **Minimizes the total travel distance**.

### ****Assumptions Made****

✅ **Symmetric TSP** – The distances between cities are **the same in both directions** (e.g., the distance from **City 2 to City 5** is **12**, the same as from **City 5 to City 2**).

✅ **No Missing Cities** – The graph is **fully connected**, meaning every city is reachable from others (even if not directly).

✅ **Integer Distances** – The distances are **fixed whole numbers**, meaning we don’t deal with floating-point precision issues.

✅ **Single Salesman** – We assume **one person** is traveling (no multiple salesmen or multiple routes).

**CLASSICAL TSP SOLUTION**

1. Nearest Neighbor
2. Implementation
3. Result : “final tour from code”

**SELF-ORGANIZING MAP APPROACH (SOM)**

### ****Conceptual Overview: How SOM Solves TSP****

A **Self-Organizing Map (SOM)** is a neural network that learns to map input patterns onto a structured output space. For **TSP**, SOM adapts as follows:

1️⃣ **Initializing Neurons**

* We create **N neurons** (where **N = number of cities**).
* The neurons are **arranged in a circular pattern**, representing a potential tour.

2️⃣ **Representing Cities**

* Each city is placed in a **2D space** with arbitrary coordinates.
* The SOM will adjust itself to match city positions.

3️⃣ **Training (Learning Process)**

* For each city, find the **winner neuron** (the closest neuron to that city).
* Adjust the **winner neuron’s position** to move closer to the city.
* Update **neighboring neurons** (within a radius) to also move slightly closer.
* Gradually **reduce learning rate and neighborhood size** over time to refine the path.

4️⃣ **Final Route Extraction**

* After training, the order of neurons in the circular SOM **approximates the best TSP route**.

### ****Challenges & Limitations****

🚧 **Parameter Sensitivity:**

* **Learning rate, neighborhood function, and iterations** must be tuned carefully.
* Bad parameters → Poor convergence or incorrect routes.

🚧 **Not Always Optimal:**

* SOM **does not guarantee the shortest route** (it approximates).
* Performance depends on **number of training iterations**.

🚧 **Random Initialization Issues:**

* Different runs can give **slightly different results** due to randomness.

🚧 **Scaling Problems:**

* Works well for **small TSP instances** like **7 cities**, but for **large graphs, training becomes slow**.

**ANALYSIS AND COMPARISON**

### ****(a) Route Quality Comparison****

* **Nearest Neighbor (NN) Route:** 1 → 2 → 4 → 6 → 7 → 5 → 3 → 1 (Example)
* **SOM Route:** 1 → 4 → 6 → 7 → 5 → 3 → 2 → 1 (Example)
* **Comparison:**
  + NN **follows a greedy approach** and may get stuck in local optima.
  + SOM **approximates the solution** but might not be the absolute shortest route.
  + **Final Verdict:** NN tends to be **slightly better in shorter graphs**, but SOM can work well for larger cases with proper tuning.

### ****(b) Complexity Discussion****

| **Method** | **Time Complexity** | **Space Complexity** | **Optimal?** |
| --- | --- | --- | --- |
| **Nearest Neighbor** | **O(n²)** | O(n) | ❌ (Heuristic) |
| **SOM** | **O(n × iterations)** | O(n) | ❌ (Approximate) |

* **Nearest Neighbor (NN):** Faster but does not guarantee the shortest path.
* **SOM:** Slower (depends on **iterations**) and requires tuning, but learns an approximate route.
* **Final Verdict:** **For small graphs like 7 cities, NN is faster and often better.** For large graphs, SOM can be improved with advanced tuning.

### ****(c) Practical Considerations****

| **Scenario** | **Best Method** |
| --- | --- |
| **Few Cities (n ≤ 20)** | **Nearest Neighbor (NN)** (Fast & simple) |
| **Many Cities (n > 50)** | **SOM** (Scalability, adaptable) |
| **Need Guaranteed Optimal Route?** | **Dynamic Programming (Held-Karp)** (but slow) |
| **Time-sensitive application?** | **NN (Fast but heuristic)** |
| **Neural Network Adaptability?** | **SOM (Self-learning)** |

### ****(d) Suggested Improvements****

1️⃣ **Hybrid Approach:** Use **SOM for an initial estimate**, then refine with a **local optimization method** (like 2-opt).  
2️⃣ **Better Neighborhood Function:** Experiment with **Gaussian functions** instead of the default radius-based update.  
3️⃣ **Parameter Tuning:** Use **adaptive learning rates** to improve SOM convergence.

### ****Final Conclusion****

* **Nearest Neighbor** is **faster & simpler**, but may not always give the best solution.
* **SOM** is **flexible & scalable** but requires careful tuning.
* **Best choice depends on the number of cities** and **whether an optimal or approximate solution is needed**.