GROUP L

NAME	REGISTRATION NUMBER	STUDENT NUMBER
TUSIIME MARK	24/U/11684/PS	2400711684
NAMUYIMWA MARTHA	24/U/09436/PS	2400709436
OKURE ENOCK	24/U/10690/PS	2400710690
SSEBYALA DENIS TENDO	24/U/11123/PS	2400711123

REPORT FOR THE ASSIGNMENT INCLUDING ALL THE DELIVERABLES

GITHUB LINK: https://github.com/denistendo/DSA-Assignment-GROUP-L.git

TSP Representation and Data Structures

Graph Representation: Adjacency List

We represent the TSP graph using an **adjacency list**, where each city maps to its neighboring cities and their corresponding distances.

Python Implementation

```
# Adjacency list representation of the TSP graph
adj_list = {
    "City 1": {"City 2": 12, "City 3": 10, "City 7": 12},
    "City 2": {"City 1": 12, "City 3": 8, "City 4": 12},
    "City 3": {"City 1": 10, "City 2": 8, "City 4": 11, "City 5": 3, "City 7":
12},
    "City 4": {"City 2": 12, "City 3": 11, "City 5": 11, "City 6": 10},
    "City 5": {"City 3": 3, "City 4": 11, "City 6": 6, "City 7": 9},
    "City 6": {"City 4": 10, "City 5": 6, "City 7": 9},
    "City 7": {"City 1": 12, "City 3": 12, "City 5": 9, "City 6": 9}
}

def get_distance(city1, city2):
    """Returns distance between two cities or infinity if no direct path
exists"""
    return adj_list[city1].get(city2, float('inf'))
```

Explanation

- **Structure**: Each city (key) has a dictionary of connected cities (values) with their distances.
- **Lookup Efficiency**: Retrieving a distance is **O(1)** on average (hash table lookup).
- **Space Efficiency**: Only stores existing edges, making it ideal for sparse graphs.

Problem Setup

TSP Objective

Find the shortest possible route that:

1. Starts and ends at **City 1**.
2. Visits each of the 7 cities **exactly once**.
3. Minimizes the total travel distance.

Assumptions

1. **Undirected Graph**: All edges are bidirectional (e.g., City 1 → City 2 = City 2 → City 1 = 12).
2. **No Isolated Cities**: Every city is reachable from any other city.
3. **Non-Negative Distances**: All edge weights are positive.

Justification for Adjacency List Choice

The adjacency list was chosen for this TSP implementation because it provides an efficient and intuitive way to represent the city connections and distances. For this 7-city problem, where each city is connected to only a few others (sparse graph), the adjacency list saves memory by only storing existing edges rather than allocating space for all possible connections like an adjacency matrix would. The average O(1) lookup time for distances between connected cities is sufficiently fast for this small-scale problem.

Additionally, the adjacency list structure naturally mirrors how we conceptualize the TSP - as a collection of cities with specific connections to others. This makes the implementation more readable and easier to debug compared to matrix representations. The dictionary-of-dictionaries approach also allows for quick modifications if the graph needs to be updated, which would be useful if extending this to larger problems or dynamic scenarios. For this small, sparse TSP instance, the adjacency list provides the best balance of memory efficiency and computational performance.

Classical TSP Solution(using dynamic programming)

```python
from itertools import permutations
import sys

# Adjacency matrix representation of the graph
graph = [
    [0, 12, 10, 0, 0, 0, 12],
    [12, 0, 8, 12, 0, 0, 0],
    [10, 8, 0, 11, 3, 0, 9],
    [0, 12, 11, 0, 11, 10, 0],
    [0, 0, 3, 11, 0, 6, 7],
    [0, 0, 0, 10, 6, 0, 9],
    [12, 0, 9, 0, 7, 9, 0]
]

n = len(graph)  # Number of cities
INF = sys.maxsize  # Representing a large number for infinity

# Memoization table for storing subproblem results
dp = [[-1] * (1 << n) for _ in range(n)]

def tsp(mask, pos):
    """
    Solves TSP using Dynamic Programming with bitmasking.
    :param mask: Bitmask representing visited cities.
    :param pos: Current city.
    :return: Minimum tour cost from pos visiting unvisited cities.
    """
    # Base case: all cities visited, return to start (0)
    if mask == (1 << n) - 1:
        return graph[pos][0] if graph[pos][0] > 0 else INF

    # If already computed, return stored result
    if dp[pos][mask] != -1:
        return dp[pos][mask]

    min_cost = INF
    # Try visiting all cities that have not been visited yet
    for city in range(n):
        if (mask & (1 << city)) == 0 and graph[pos][city] > 0:
            new_cost = graph[pos][city] + tsp(mask | (1 << city), city)
            min_cost = min(min_cost, new_cost)

    # Store the computed result in the memoization table
    dp[pos][mask] = min_cost
    return min_cost
```

```
def find_tour():
    """
    Finds the optimal TSP tour and its cost.
    :return: Optimal path and its total cost
    """
    optimal_cost = tsp(1, 0)  # Start from city 0 with only it visited

    # Reconstruct the optimal path
    mask = 1  # Start with only the first city visited
    pos = 0  # Start at city 0
    path = [0]  # Store the optimal path

    for _ in range(n - 1):
        best_next = None
        best_cost = INF
        # Try finding the best next city to visit
        for city in range(n):
            if (mask & (1 << city)) == 0 and graph[pos][city] > 0:
                new_cost = graph[pos][city] + tsp(mask | (1 << city), city)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_next = city

        if best_next is None:
            break  # No valid next city, break loop

        path.append(best_next)  # Add city to the path
        mask |= (1 << best_next)  # Mark city as visited
        pos = best_next  # Move to next city

    path.append(0)  # Return to the starting city to complete the tour
    return path, optimal_cost

# Get the Final route and its Total route Cost
tour, cost = find_tour()
print("Final route:", [city + 1 for city in tour])  # Convert 0-based index to 1-
based
print("Total route Cost:", cost)
```

results

```
Final route: [1, 2, 4, 6, 7, 5, 3, 1]
Total route Cost: 63
```

Solving TSP Using Self-Organizing Maps (SOM)

Conceptual Overview

This implementation adapts the Self-Organizing Map (SOM) to solve the Traveling Salesman Problem (TSP) through the following key mechanisms:

1. **Neurons as Cities**
   - Each neuron in the SOM represents a potential location or segment of the tour path
   - Neurons compete to align themselves with actual city coordinates during training
2. **Initialization**
   - Neurons are initialized in a circular topology around the centroid of city coordinates
   - Uses force-directed placement to convert adjacency matrix distances into 2D spatial coordinates
   - Starts with 3×N neurons (where N = number of cities) for better path coverage
3. **Neighborhood Function**
   - Gaussian function determines update strength for neurons near the winner
   - Neighborhood radius exponentially decays from N_neurons/2 to 1 over iterations
   - Preserves topological relationships between neurons throughout training
4. **Learning Rate**
   - Controls magnitude of neuron weight updates
   - Linearly decays from initial value (0.8) to 0 across iterations
   - Balances exploration vs exploitation during training
5. **Winner-Takes-All Mechanism**
   - For each randomly selected city:

        1. Identify closest neuron (winner) via Euclidean distance
        2. Update winner's weights toward the city
        3. Update neighboring neurons with Gaussian-weighted adjustments
    - Neurons gradually organize to "span" the city locations
6. **Decaying Parameters**
    - Both neighborhood size and learning rate systematically decrease
    - Allows initial coarse organization followed by fine-tuning
    - Helps avoid premature convergence to suboptimal solutions

Implementation

```
import numpy as np
from math import inf
from collections import deque
import random

class StrictAdjacencySOM_TSP:
    def __init__(self, adjacency_matrix, n_iterations=10000, learning_rate=0.8):
        self.adj_matrix = np.array(adjacency_matrix)
        self.n_cities = len(adjacency_matrix)
        self.n_iterations = n_iterations
        self.learning_rate = learning_rate

        # Convert adjacency matrix to coordinates
        self.city_coords = self._adjacency_to_coords()

        # Initialize SOM parameters
        self.n_neurons = self.n_cities * 3  # More neurons for better coverage
        self.neurons = self._initialize_neurons()

    def _adjacency_to_coords(self):
        """Convert adjacency matrix to 2D coordinates using force-based
placement"""
        coords = np.random.rand(self.n_cities, 2)
        max_distance = np.max(self.adj_matrix[self.adj_matrix != inf]) if
np.any(self.adj_matrix != inf) else 1

        for _ in range(100):
            for i in range(self.n_cities):
                for j in range(i+1, self.n_cities):
                    if self.adj_matrix[i, j] != inf:
                        target_dist = self.adj_matrix[i, j] / max_distance
                        current_dist = np.linalg.norm(coords[i] - coords[j]) +
1e-6  # Avoid div by zero
                        force = (target_dist - current_dist) / current_dist
                        coords[i] -= 0.05 * force * (coords[j] - coords[i])
                        coords[j] += 0.05 * force * (coords[j] - coords[i])
        return coords

    def _initialize_neurons(self):
        """Initialize neurons in a circle around city center"""
        center = np.mean(self.city_coords, axis=0)
        radius = 0.5 * np.max(np.ptp(self.city_coords, axis=0))
        angles = np.linspace(0, 2 * np.pi, self.n_neurons, endpoint=False)
        return np.array([center + radius * np.array([np.cos(a), np.sin(a)]) for a
in angles])

    def train(self):
        """Train the SOM"""
        for iteration in range(self.n_iterations):
            lr = self.learning_rate * (1 - iteration/self.n_iterations)
            radius = max(1, self.n_neurons/2 * np.exp(-
iteration/self.n_iterations))

            city_idx = random.randint(0, self.n_cities - 1)
            winner = np.argmin(np.linalg.norm(self.neurons -
self.city_coords[city_idx], axis=1))

            # Update neurons
            for i, neuron in enumerate(self.neurons):
                dist_to_winner = min(abs(i - winner), self.n_neurons - abs(i -
winner))
                influence = np.exp(-(dist_to_winner**2) / (2 * (radius**2))) * lr
                self.neurons[i] += influence * (self.city_coords[city_idx] -
neuron)

    def get_valid_route(self):
        """Generate route that strictly follows adjacency rules using BFS"""
        queue = deque()
        queue.append(([0], set([0])))  # Start from city 0

        best_route = None
        best_distance = inf

        while queue:
            current_route, visited = queue.popleft()
            # Complete the cycle if all cities visited
            if len(visited) == self.n_cities:
                if self.adj_matrix[current_route[-1], 0] != inf:
                    final_route = current_route + [0]
                    distance = sum(self.adj_matrix[final_route[i],
final_route[i+1]]
                                  for i in range(len(final_route)-1))
                    if distance < best_distance:
                        best_distance = distance
                        best_route = final_route
                continue

            # Explore all valid adjacent cities
            last_city = current_route[-1]
            for next_city in range(self.n_cities):
                if next_city not in visited and self.adj_matrix[last_city,
next_city] != inf:
                    new_visited = visited.copy()
                    new_visited.add(next_city)
                    queue.append((current_route + [next_city], new_visited))

        return best_route

    def solve(self):
        """Run the complete solution process"""
        self.train()
        route = self.get_valid_route()
        if route:
            distance = sum(self.adj_matrix[route[i], route[i+1]]
                          for i in range(len(route)-1))
            return route, distance
        return None, inf

# Adjacency matrix
adjacency_matrix = [
    [0, 12, 10, inf, inf, inf, 12],  # City 1 (index 0)
    [12, 0, 8, 12, inf, inf, inf],   # City 2 (index 1)
    [10, 8, 0, 11, 3, inf, 9],      # City 3 (index 2)
    [inf, 12, 11, 0, 11, 10, inf],  # City 4 (index 3)
    [inf, inf, 3, 11, 0, 6, 7],     # City 5 (index 4)
    [inf, inf, inf, 10, 6, 0, 9],   # City 6 (index 5)
    [12, inf, 9, inf, 7, 9, 0]      # City 7 (index 6)
]

# Solve with strict adjacency constraints
```

```python
solver = StrictAdjacencySOM_TSP(adjacency_matrix)
route, distance = solver.solve()

if route:
    print("Valid Route Found:")
    print("Path:", " -> ".join(str(c+1) for c in route))
    print(f"Total Distance: {distance}")
else:
    print("No valid route found that satisfies all constraints")
```

RESULTS

```
Valid Route Found:
Path: 1 -> 2 -> 4 -> 6 -> 7 -> 5 -> 3 -> 1
Total Distance: 63.0
```

Limitations

- **Parameter Sensitivity**: The performance of the algorithm is significantly influenced by the choice of hyperparameters, such as learning rate, neighborhood size, and the number of neurons, which can impact the quality of the solution.

- **Risk of Suboptimal Convergence**: In some cases, particularly with large or intricate TSP instances, the algorithm may end up finding a solution that is not the best possible, leading to suboptimal results.

- **Scalability Issues**: The algorithm may struggle when dealing with large-scale datasets, such as those containing hundreds or thousands of cities, leading to inefficiencies in both time and resource consumption.

- **Local Optima**: The Self-Organizing Map (SOM) algorithm may get trapped in local optima, which means it could produce a route that is not the best possible solution in the global context of the problem.

- **Sensitivity to Initialization**: The starting configuration of neurons can significantly affect the algorithm's performance, as different initializations can lead to different convergence paths and outcomes.

- **Impact of Neighborhood Function Choice**: The selection of the neighborhood function, such as a Gaussian function, plays a key role in how the algorithm explores the solution space and impacts its overall performance.

- **Decaying Parameters**: The gradual reduction of parameters over time (like learning rate or neighborhood size) can either cause the algorithm to converge too early, missing better solutions, or cause it to converge too slowly, resulting in prolonged computation times.

- **Lack of Theoretical Guarantees**: Unlike certain traditional optimization methods, SOMs do not provide strong theoretical guarantees about reaching an optimal solution or about the convergence behavior, which can be a concern in critical applications.

Comparison between the two routes.

	Adjacency matrix with dynamic programming	**Self Organizing Map**
final route	[1, 2, 4, 6, 7, 5, 3, 1]	[1, 2, 4, 6, 7, 5, 3, 1]
cost	63	63

The results a similar since this is a small experiment.

Complexity comparison

	Adjacency matrix with dynamic programming	**Self Organizing Map**
Time complexity	O(n^2.2^n)	O(N*M*I) N- the number of data M-number of neurons in a map I-Number of iterations during training

Tradeoffs in choosing exact vs heuristic methods.

- Exact methods work for small number of cities but not for a large number of cities.
- Exact methods are slower than the heuristic methods
- Exact methods use more memory space compared to heuristic methods
- Exact methods give you the optimal solution, while the heuristic methods give near optimal results.

Improvements for Self Organizing maps

- Implementation Parallelization
- Hybrid approaches like combining SOM with local search methods