GROUP U MEMBERS

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

GRAPH

```
graph = [
    [0, 12, 10, 8, 12, 3, 9],
    [12, 0, 12, 11, 6, 7, 9],
    [10, 12, 0, 11, 10, 11, 6],
    [8, 11, 11, 0, 7, 9, 12],
    [12, 6, 10, 7, 0, 9, 10],
    [3, 7, 11, 9, 9, 0, 11],
    [9, 9, 6, 12, 10, 11, 0]
]
```

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.

JUSTIFYING YOUR CHOICE OF REPRESENTATION

EFFICIENCY: LOOKUP TIME- THE ADJACENCY MATRIX ALLOWS FOR CONSTANT TIME LOOKUP OF DISTANCES BETWEEN ANY TWO CITIES.

SIMPLICITY: 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.

DIRECT ACCESS IN CONSTANT TIME : SINCE THE MATRIX IS STORED IN MEMORY AS A 2D ARRAY, ACCESSING ANY ELEMENT GRAPH TAKES A SHORT TIME.

. **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.

Classical TSP

```python
import itertools

def held_karp(graph):
    """
    Solves the Traveling Salesman Problem using the Held-Karp algorithm (Dynamic Programming).

    Parameters:
    graph (list of list of int): Adjacency matrix representing the distances between cities.

    Returns:
    tuple: A tuple containing the optimal cost and the optimal path.
    """
    n = len(graph)
    C = {}

    # Base case: Start at city 0, then visit another city k
    for k in range(1, n):
        C[(1 << k, k)] = (graph[0][k], 0)

    # Iterate over subsets of increasing size
    for subset_size in range(2, n):
        for subset in itertools.combinations(range(1, n), subset_size):
            bits = sum(1 << bit for bit in subset)  # Correct bitmask calculation
            for k in subset:
                prev_bits = bits & ~(1 << k)
                res = []

                for m in subset:
                    if m == k:
```

```python
                    if m == k:
                        continue
                    if (prev_bits, m) in C:
                        res.append((C[(prev_bits, m)][0] + graph[m][k], m))

                C[(bits, k)] = min(res) if res else (float('inf'), -1)  # Avoid KeyError

    # Find the minimum cost to visit all cities and return to the start
    bits = (1 << n) - 2  # All cities visited except city 0
    res = []
    for k in range(1, n):
        if (bits, k) in C:
            res.append((C[(bits, k)][0] + graph[k][0], k))

    opt, parent = min(res)

    # Path reconstruction
    path = [0]
    while parent != 0:
        path.append(parent)
        next_bits = bits & ~(1 << parent)  # Remove current city
        parent = C.get((bits, parent), (None, 0))[1]  # Ensure key exists
        bits = next_bits

    path.append(0)

    return opt, path
```

```python
    path.append(0)

    return opt, path

# Adjacency matrix representing the distances between cities
graph = [
    [0, 12, 10, 8, 12, 3, 9],
    [12, 0, 12, 11, 6, 7, 9],
    [10, 12, 0, 11, 10, 11, 6],
    [8, 11, 11, 0, 7, 9, 12],
    [12, 6, 10, 7, 0, 9, 10],
    [3, 7, 11, 9, 9, 0, 11],
    [9, 9, 6, 12, 10, 11, 0]
]

# Solve the TSP using the Held-Karp algorithm
opt_cost, opt_path = held_karp(graph)

# Output the final route and total distance
print(f"Optimal Cost: {opt_cost}")
print(f"Optimal Path: {opt_path}")


#Optimal Cost: 49
#Optimal Path: [0, 5, 1, 4, 3, 2, 6, 0]
```

SOM

```python
import random
import math

def euclidean_distance(a, b):
    """Computes the Euclidean distance between two points."""
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)

def som_tsp(cities, max_epochs=1000, initial_lr=0.8, initial_radius=3):
    """
    Solves the Traveling Salesman Problem using a Self-Organizing Map (SOM) without NumPy.

    Parameters:
    cities (list of tuples): List of city coordinates (x, y).
    max_epochs (int): Number of training iterations.
    initial_lr (float): Initial learning rate.
    initial_radius (int): Initial neighborhood radius.

    Returns:
    list: The approximate route found by the SOM.
    """
    n = len(cities)

    # Initialize neurons in a circular arrangement
    theta = [2 * math.pi * i / n for i in range(n)]
    mean_x = sum(x for x, y in cities) / n
    mean_y = sum(y for x, y in cities) / n
    scale = max(max(x for x, y in cities) - min(x for x, y in cities),
                max(y for x, y in cities) - min(y for x, y in cities)) / 2
    neurons = [(mean_x + scale * math.cos(t), mean_y + scale * math.sin(t)) for t in theta]

    for epoch in range(max_epochs):
        lr = initial_lr * (1 - epoch / max_epochs)  # Decay learning rate
        radius = max(1, int(initial_radius * (1 - epoch / max_epochs)))  # Decay neighborhood radius

        for city in cities:
            # Find winner neuron (closest to city)
            distances = [euclidean_distance(neuron, city) for neuron in neurons]
            winner = distances.index(min(distances))
```

```python
            # Update winner and neighbors
            for i in range(-radius, radius + 1):
                neighbor = (winner + i) % n
                influence = math.exp(- (i ** 2) / (2 * (radius ** 2)))  # Gaussian function

                # Move neuron toward the city
                neurons[neighbor] = (
                    neurons[neighbor][0] + lr * influence * (city[0] - neurons[neighbor][0]),
                    neurons[neighbor][1] + lr * influence * (city[1] - neurons[neighbor][1])
                )

    # Find nearest-neighbor mapping between neurons and cities
    route = []
    remaining = set(range(n))
    for city in cities:
        distances = {idx: euclidean_distance(neurons[idx], city) for idx in remaining}
        best_match = min(distances, key=distances.get)
        route.append(best_match)
        remaining.remove(best_match)

    return route

# Example input (list of city coordinates)
cities = [(0, 0), (1, 2), (3, 3), (6, 5), (8, 8), (10, 10), (12, 12)]

# Train the SOM on the given TSP graph data
route = som_tsp(cities)

# Function to calculate the total distance of a given route
def calculate_total_distance(route, graph):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += graph[route[i]][route[i + 1]]
    total_distance += graph[route[-1]][route[0]]  # Return to the starting city
    return total_distance
```

```python
# Function to calculate the total distance of a given route
def calculate_total_distance(route, graph):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += graph[route[i]][route[i + 1]]
    total_distance += graph[route[-1]][route[0]]  # Return to the starting city
    return total_distance

# Adjacency matrix representing the distances between cities
graph = [
    [0, 12, 10, 8, 12, 3, 9],
    [12, 0, 12, 11, 6, 7, 9],
    [10, 12, 0, 11, 10, 11, 6],
    [8, 11, 11, 0, 7, 9, 12],
    [12, 6, 10, 7, 0, 9, 10],
    [3, 7, 11, 9, 9, 0, 11],
    [9, 9, 6, 12, 10, 11, 0]
]

# Convert route indices to city indices
city_route = [int(i) for i in route]

# Calculate the total distance of the route
total_distance = calculate_total_distance(city_route, graph)

# Output the final route and total distance
print(f"Approximate Route: {city_route}")
print(f"Total Distance: {total_distance} units")

#Approximate Route: [3, 4, 2, 5, 1, 0, 6]
#Total Distance: 68 units
```

Self-Organizing Map (SOM) Approach for TSP

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.

Implementation of the SOM approach to solve the TSP

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 :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

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.

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)

SOM IS SIGNIFICANTLY FASTER FOR LARGE-SCALE PROBLEMS, WHILE DP IS BETTER FOR SMALL GRAPHS

FINAL ROUTES OBTAINED

DYNAMIC PROGRAMMING FINDS THE EXACT SHORTEST PATH (0, 5, 1, 4, 3, 2, 6, 0). – 49 UNITS

SOM FINDS A NEAR-OPTIMAL SOLUTION (≈68 UNITS), BUT NOT ALWAYS THE BEST.

.

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

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.