**Name: Hasnain Haider**

**Sap: 56626**

**1. Introduction**

The **Barnacle Mating Optimizer (BMO)** is an algorithm that mimics the way barnacles (marine animals) mate in nature. Barnacles have a unique way of mating, where they use long "mating appendages" to find and connect with each other. These appendages allow barnacles to search for other mates over long distances.

The BMO algorithm uses this behavior to solve optimization problems, like the **Traveling Salesman Problem (TSP)**. In TSP, the goal is to find the shortest path that visits each city exactly once and returns to the starting point.

In simple terms, BMO works by creating a population of possible routes (or solutions) to the TSP. These solutions are then "mated" (combined) in a way that produces better solutions over time. The algorithm continues to evolve the population of routes until it finds the shortest possible path.

It is like having a group of barnacles that keep searching for better connections (routes) until they find the most efficient one.

**2. Problem Statement**

The **Traveling Salesman Problem (TSP)** involves finding the shortest route that visits a given set of cities exactly once and returns to the starting point. In this project, the problem includes 8 cities in Pakistan: Karachi, Lahore, Islamabad, Peshawar , Quetta, Multan , Sialkot ,Faisalabad. The goal is to find the shortest path that covers all these cities using a provided distance matrix.

This problem is ideal for the **Barnacle Mating Optimizer (BMO)** because it has many possible routes (8! = 40,320 combinations), making it challenging to solve using simple methods. BMO is effective in exploring and finding better solutions efficiently by simulating the mating behavior of barnacles.

### ****3. Methodology****

To solve the TSP using the **Barnacle Mating Optimizer (BMO)**, we followed these steps:

* **City Representation**: Cities are indexed, and distances are taken from a predefined distance matrix.
* **Distance Calculation**: Total route distance is calculated by adding the distances between consecutive cities.
* **Initial Population**: Random routes (city orders) are generated.
* **Mating Process**: Routes combine with others to create new ones, simulating barnacle mating.
* **Fitness Evaluation**: Routes with shorter distances are considered better (higher fitness).
* **Selection**: The best routes are kept, and the process is repeated to improve the solution over time.

### ****4. Implementation****

The BMO algorithm was implemented in Python. A distance matrix of 11 Pakistani cities was used. Each city was treated as a node, and routes were represented as permutations of city indices. The algorithm simulates mating by combining two routes to generate new ones, evaluates fitness based on total travel distance, and iteratively improves the population over generations.

[**Code in python**](file:///C:\Users\turi\Desktop\aoa%20project\code.docx)

**import random**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Define city names and distance matrix for 8 cities**

**city\_names = [**

**"Karachi", "Lahore", "Islamabad", "Peshawar", "Quetta",**

**"Multan", "Sialkot", "Faisalabad"**

**]**

**distance\_matrix = [**

**[0, 1210, 1410, 1520, 690, 900, 1225, 1150],**

**[1210, 0, 375, 510, 960, 340, 125, 140],**

**[1410, 375, 0, 185, 820, 550, 300, 250],**

**[1520, 510, 185, 0, 950, 680, 460, 400],**

**[690, 960, 820, 950, 0, 600, 990, 870],**

**[900, 340, 550, 680, 600, 0, 410, 280],**

**[1225, 125, 300, 460, 990, 410, 0, 200],**

**[1150, 140, 250, 400, 870, 280, 200, 0]**

**]**

**# Define the TSP functions**

**def calculate\_total\_distance(route):**

**total = 0**

**for i in range(len(route)):**

**total += distance\_matrix[route[i]][route[(i + 1) % len(route)]]**

**return total**

**def initialize\_population(size, num\_cities):**

**return [random.sample(range(num\_cities), num\_cities) for \_ in range(size)]**

**def mate(parent1, parent2):**

**start, end = sorted(random.sample(range(len(parent1)), 2))**

**child = [-1] \* len(parent1)**

**child[start:end+1] = parent1[start:end+1]**

**p2\_index = 0**

**for i in range(len(parent1)):**

**if child[i] == -1:**

**while parent2[p2\_index] in child:**

**p2\_index += 1**

**child[i] = parent2[p2\_index]**

**return child**

**def mutate(route):**

**i, j = random.sample(range(len(route)), 2)**

**route[i], route[j] = route[j], route[i]**

**return route**

**def bmo\_tsp(pop\_size=50, generations=200, mutation\_rate=0.2):**

**num\_cities = len(distance\_matrix)**

**population = initialize\_population(pop\_size, num\_cities)**

**best\_solution = None**

**best\_distance = float('inf')**

**fitness\_history = []**

**for gen in range(generations):**

**population = sorted(population, key=calculate\_total\_distance)**

**if calculate\_total\_distance(population[0]) < best\_distance:**

**best\_solution = population[0]**

**best\_distance = calculate\_total\_distance(population[0])**

**new\_population = population[:5]  # Elitism**

**while len(new\_population) < pop\_size:**

**parent1, parent2 = random.sample(population[:25], 2)**

**child = mate(parent1, parent2)**

**if random.random() < mutation\_rate:**

**child = mutate(child)**

**new\_population.append(child)**

**population = new\_population**

**fitness\_history.append(best\_distance)**

**return best\_solution, best\_distance, fitness\_history**

**# Run optimizer**

**best\_route, best\_dist, fitness = bmo\_tsp()**

**# Display results**

**print(" Best route:")**

**for city in best\_route:**

**print(city\_names[city], end=" → ")**

**print(city\_names[best\_route[0]])**

**print(f" Total Distance: {best\_dist} km")**

### ****Space Complexity****

**O(n × population size)**  
The algorithm stores a population of different possible routes. Each route includes all cities, so the total memory usage depends on the number of cities (n) and how many routes are being processed in each generation.

**Time Complexity**

**O (g × p × n)**

Where:

* g = number of generations
* p = population size (number of routes in each generation)
* n = number of cities  
  This is because, in every generation, the algorithm evaluates the fitness (distance) of each route and creates new routes through mating.

**Application**

* **Logistics and delivery**: Optimizing truck or van routes to reduce fuel and time.
* **Travel planning**: Best route through multiple cities for tourism or business.
* **Computer networks**: Laying out cables or connections between nodes efficiently.
* **Metaheuristic testing**: TSP is a common benchmark for testing optimization algorithms like BMO, GA, ACO, etc.

**Limitation**

* **Scalability**: As the number of cities increases, the algorithm becomes slower and uses more memory.
* **No guarantee of global optimum**: It gives a very good solution, but not always the absolute best.
* **Random behavior**: Since it’s based on random mating, results can vary slightly each time the program runs.
* **Parameter sensitivity**: The performance depends on settings like population size and generations, which may require tuning.