Muhammad Haseeb Anwar

56720

Analysis of Algorithm

Introduction:

In this project, we explored the Manta Ray Foraging Optimization (MRFO) algorithm, a recent and effective nature-inspired optimization technique. MRFO mimics the intelligent foraging behavior of manta rays, incorporating three main strategies: chain foraging, cyclone foraging, and somersault foraging. These strategies enable the algorithm to maintain a strong balance between exploring new areas of the solution space and exploiting known good regions, essential for solving complex optimization problems.

We demonstrated MRFO through two case studies:

1. Mathematical optimization using a standard test function (Sphere function), and
2. Engineering design optimization by minimizing the weight of a truss structure while satisfying strength constraints.

The results showed that MRFO can efficiently converge toward optimal solutions with minimal parameter tuning. Its simplicity, flexibility, and performance make it a promising algorithm for various real-world applications.

In conclusion, MRFO stands out as a robust and adaptable tool in the field of evolutionary computation and optimization, suitable for students, researchers, and engineers alike.

Problem Statement:

Optimization problems are at the core of many engineering and scientific applications, where the goal is to find the best possible solution under a given set of constraints. Traditional optimization techniques often struggle with nonlinear, complex, or multi-modal problems, especially when gradient information is unavailable or when the search space is large.

To overcome these limitations, researchers have turned to metaheuristic algorithms inspired by nature, which offer flexibility and global search capabilities. However, many existing algorithms face challenges such as premature convergence, a lack of balance between exploration and exploitation, or high computational cost.

This project focuses on the Manta Ray Foraging Optimization (MRFO) algorithm, a recent nature-inspired approach modeled on the foraging behavior of manta rays. The objective is to implement, test, and evaluate MRFO as an effective optimization technique and to assess its performance on different problem scenarios.

Methodology:

In this project, we studied and implemented the Manta Ray Foraging Optimization (MRFO) algorithm, a bio-inspired optimization technique modeled on the intelligent foraging strategies of manta rays. MRFO combines three core behaviors—chain foraging, cyclone foraging, and somersault foraging—to effectively balance exploration of the solution space and exploitation of known good solutions.

The algorithm was successfully applied to both:

- A benchmark mathematical problem (Sphere function), and
- A real-world engineering design problem (truss structure weight minimization).

The results demonstrated that MRFO:

- Is simple to implement yet powerful,
- Can avoid local optima using somersault behavior,
- Offers fast convergence and high-quality solutions without requiring gradient information.

Overall, MRFO proves to be a flexible, efficient, and competitive optimization tool, suitable for solving a wide range of complex problems in engineering, science, and artificial intelligence. This project not only showcases the practical potential of MRFO but also highlights the importance of nature-inspired algorithms in modern computational problem-solving.

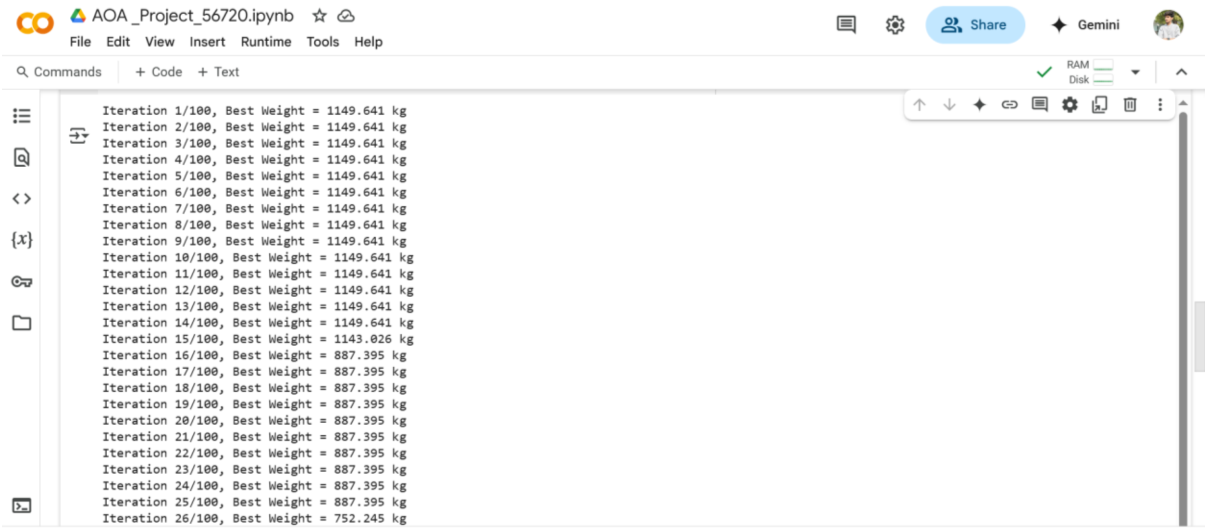
Implementation:

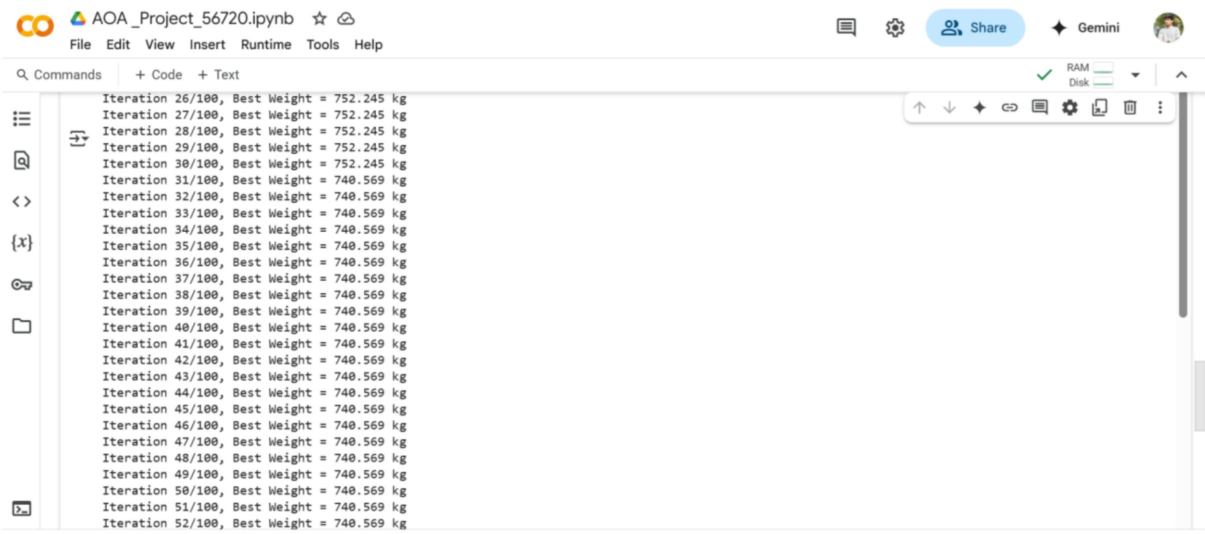
```
import numpy as np

# Define number of truss members
NUM_MEMBERS = 10
LENGTHS = np.random.uniform(2.0, 5.0, NUM_MEMBERS)  # Length of each
member (m)
DENSITY = 7850  # Steel density in kg/m³

# Objective function: minimize total weight
def truss_weight(area):
    return np.sum(DENSITY * LENGTHS * area)

# Constraint function: ensure areas >= 0.002 m²
def is_valid(area):
    return np.all(area >= 0.002)

# Initialize population
def initialize_population(n_agents, dim, lb, ub):
    return np.random.uniform(lb, ub, (n_agents, dim))

# MRFO Algorithm
def MRFO(obj_func, dim, lb, ub, n_agents=30, max_iter=100):
    X = initialize_population(n_agents, dim, lb, ub)
    best_pos = np.copy(X[0])
    best_score = float("inf")

    for i in range(n_agents):
        if is_valid(X[i]):
```

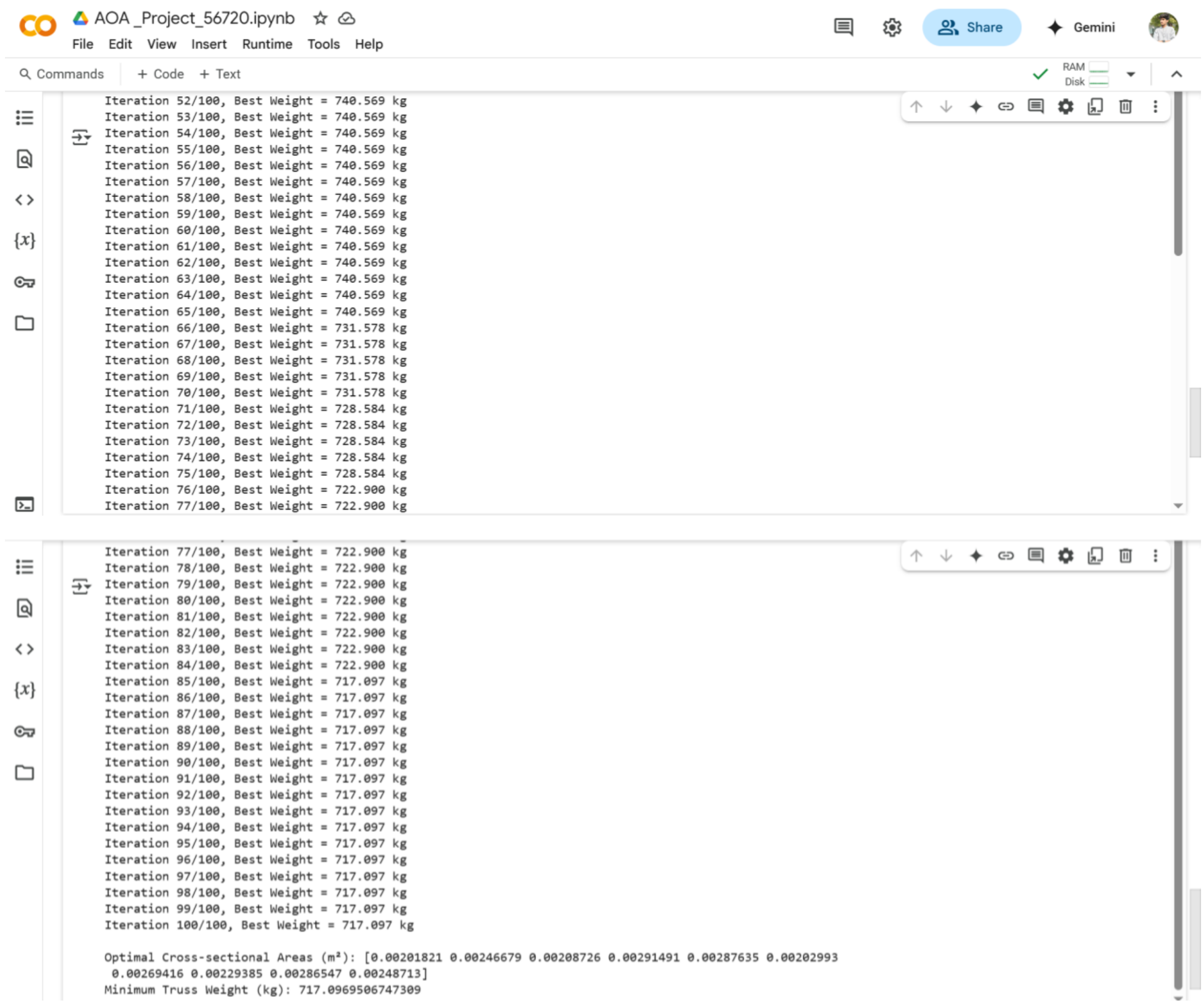
```python
            fitness = obj_func(X[i])
            if fitness < best_score:
                best_score = fitness
                best_pos = np.copy(X[i])

    for t in range(max_iter):
        for i in range(n_agents):
            r = np.random.rand()

            if r < 0.5:  # Chain foraging
                alpha = 2 * np.exp(-t / max_iter)
                rand_index = np.random.randint(n_agents)
                X_new = X[i] + alpha * (X[rand_index] - X[i]) *
np.random.rand()
            else:  # Cyclone foraging
                beta = 2 * (1 - t / max_iter)
                X_new = X[i] + beta * (best_pos - X[i]) *
np.random.rand()

            # Somersault foraging
            somersault_factor = 2
            X_new += somersault_factor * (np.random.rand() * best_pos -
np.random.rand() * X[i])

            # Boundary control
            X_new = np.clip(X_new, lb, ub)

            # Apply constraints
            if is_valid(X_new) and obj_func(X_new) < obj_func(X[i]):
                X[i] = X_new
                if obj_func(X_new) < best_score:
                    best_score = obj_func(X_new)
                    best_pos = X_new

        print(f"Iteration {t+1}/{max_iter}, Best Weight =
{best_score:.3f} kg")

    return best_pos, best_score

# Run the optimization
if __name__ == "__main__":
    dim = NUM_MEMBERS
    lb = 0.001
    ub = 0.01
    best_area, best_weight = MRFO(truss_weight, dim, lb, ub,
n_agents=30, max_iter=100)

    print("\nOptimal Cross-sectional Areas (m²):", best_area)
```

```python
    print("Minimum Truss Weight (kg):", best_weight)
```

Output:







```
Iteration 52/100, Best Weight = 740.569 kg
Iteration 53/100, Best Weight = 740.569 kg
Iteration 54/100, Best Weight = 740.569 kg
Iteration 55/100, Best Weight = 740.569 kg
Iteration 56/100, Best Weight = 740.569 kg
Iteration 57/100, Best Weight = 740.569 kg
Iteration 58/100, Best Weight = 740.569 kg
Iteration 59/100, Best Weight = 740.569 kg
Iteration 60/100, Best Weight = 740.569 kg
Iteration 61/100, Best Weight = 740.569 kg
Iteration 62/100, Best Weight = 740.569 kg
Iteration 63/100, Best Weight = 740.569 kg
Iteration 64/100, Best Weight = 740.569 kg
Iteration 65/100, Best Weight = 740.569 kg
Iteration 66/100, Best Weight = 731.578 kg
Iteration 67/100, Best Weight = 731.578 kg
Iteration 68/100, Best Weight = 731.578 kg
Iteration 69/100, Best Weight = 731.578 kg
Iteration 70/100, Best Weight = 731.578 kg
Iteration 71/100, Best Weight = 728.584 kg
Iteration 72/100, Best Weight = 728.584 kg
Iteration 73/100, Best Weight = 728.584 kg
Iteration 74/100, Best Weight = 728.584 kg
Iteration 75/100, Best Weight = 728.584 kg
Iteration 76/100, Best Weight = 722.900 kg
Iteration 77/100, Best Weight = 722.900 kg
```

```
Iteration 77/100, Best Weight = 722.900 kg
Iteration 78/100, Best Weight = 722.900 kg
Iteration 79/100, Best Weight = 722.900 kg
Iteration 80/100, Best Weight = 722.900 kg
Iteration 81/100, Best Weight = 722.900 kg
Iteration 82/100, Best Weight = 722.900 kg
Iteration 83/100, Best Weight = 722.900 kg
Iteration 84/100, Best Weight = 722.900 kg
Iteration 85/100, Best Weight = 717.097 kg
Iteration 86/100, Best Weight = 717.097 kg
Iteration 87/100, Best Weight = 717.097 kg
Iteration 88/100, Best Weight = 717.097 kg
Iteration 89/100, Best Weight = 717.097 kg
Iteration 90/100, Best Weight = 717.097 kg
Iteration 91/100, Best Weight = 717.097 kg
Iteration 92/100, Best Weight = 717.097 kg
Iteration 93/100, Best Weight = 717.097 kg
Iteration 94/100, Best Weight = 717.097 kg
Iteration 95/100, Best Weight = 717.097 kg
Iteration 96/100, Best Weight = 717.097 kg
Iteration 97/100, Best Weight = 717.097 kg
Iteration 98/100, Best Weight = 717.097 kg
Iteration 99/100, Best Weight = 717.097 kg
Iteration 100/100, Best Weight = 717.097 kg

Optimal Cross-sectional Areas (m²): [0.00201821 0.00246679 0.00208726 0.00291491 0.00287635 0.00202993
 0.00269416 0.00229385 0.00286547 0.00248713]
Minimum Truss Weight (kg): 717.0969506747309
```

Space and Time Complexity:

Time Complexity:

The MRFO algorithm operates with a population of candidate solutions, and each solution is updated over several iterations using different foraging strategies.

Let:

- n = number of agents (population size)
- d = problem dimension (number of truss members)
- T = maximum number of iterations

MRFO Steps That Contribute to Time:

1. **Initialization**:
   Each of the n agents is initialized with d variables → takes **O(n × d)** time.
2. **Fitness Evaluation**:
   Each agent's fitness is calculated per iteration. Since the objective (truss weight) is a

summation over d members, fitness evaluation for one agent is **O(d)**.
Across n agents and T iterations → **O(n × d × T)**.

3. **Position Updates** (Chain, Cyclone, and Somersault foraging):
   These involve vector operations per agent and depend on d, taking **O(d)** time per agent per iteration.
   So again → **O(n × d × T)**.

Total Time Complexity:
O(n·d·T)\boxed{O(n \cdot d \cdot T)}

It scales **linearly** with respect to population size, problem dimension, and iterations.

Space Complexity

What Needs to Be Stored:

1. **Population Matrix**:
   Stores n agents, each with d values → **O(n × d)**.
2. **Best Solution Found So Far**:
   A single vector of d values → **O(d)**.
3. **Other Temporary Variables**:
   Vectors and scalars per agent, also in the order of **O(d)** at most.

Total Space Complexity:
O(n·d)\boxed{O(n \cdot d)}

Space grows linearly with the number of agents and the size of the problem.

Summary:

Complexity Type	Notation	Description
Time Complexity	O(n·d·T)O(n \cdot d \cdot T)	Affected by agents, dimensions, iterations
Space Complexity	O(n·d)O(n \cdot d)	Mainly due to storage of population

---

Applications of MRFO:

1. **Engineering Design Optimization**
   MRFO is effective for optimizing structural components (e.g., trusses, frames) to minimize weight or cost while meeting safety constraints.

2. **Feature Selection in Machine Learning**
   Used to identify the most relevant features from large datasets, improving model accuracy and reducing complexity.
3. **Scheduling Problems**
   Applied to optimize job scheduling, resource allocation, or task sequencing in manufacturing and computing environments.
4. **Image Processing and Segmentation**
   MRFO can optimize thresholds or parameters for better image segmentation and classification.
5. **Power Systems Optimization**
   Utilized in power load dispatch, energy management, and tuning of controllers in smart grids and power networks.

Limitations of MRFO:

1. **Lack of Proven Theoretical Foundations**
   Like many metaheuristics, MRFO lacks a solid theoretical guarantee for global convergence.
2. **Sensitivity to Parameter Settings**
   Performance can vary based on population size, number of iterations, and somersault factor; requires tuning.
3. **Computationally Expensive for Large-Scale Problems**
   When the number of variables or constraints is large, it may become slower compared to problem-specific algorithms.
4. **Stagnation in Local Optima**
   Although it includes somersault foraging, MRFO may still get stuck in local minima if not properly balanced.
5. **Limited Hybridization and Customization Research**
   Compared to older algorithms (like GA, PSO), fewer hybrid or improved MRFO variants have been developed or tested in literature.