VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shiv sundar sah(1BM22CS258)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Shiv sundar sah(1BM22CS258),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sneha s bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Shivsundarsah/BIS-LAB-1BM22CS258.git

2024/10/24
Thursday

Genetic Algorithm for optimization problem

· The goal is to find the value of x (within a defined) that maximum this function.

Pseudocode :

1. Initalize population of size pop-size with random values in range [x-range-low, x-range-high]
2. For generation in range(generations):
   a) Evaluate fitness of each individual:
      For each individual x_i in population:
         Fitness(x_i) = x_i ^ 2

   b) Select two parents based on fitness values (roulette-wheel selection):
      total_fitness = sum of all fitness values
      For each individual x_i in population:
         probability(x_i) = fitness(x_i) / total_fitness.

   Select two parent (p1, p2) based on probability

   c) perform crossover with probability crossover-rate:
      if random number < crossover rate:
         alpha = random value between 0 & 1
         offspring 1 = alpha * p1 + (1 - alpha) * p2
         offspring 2 = alpha * p2 + (1 - alpha) * p1

   ELSE:
      offspring = p1
      offspring = p2

d) Apply mutation with probability mutation rate:
For each offspring:
if random_number < mutation_rate:
offspring = random value in range [x_range_low, x_range_high]

e) Update population with the new offspring

3) Terminate after maximum generation or convergence

4) Output the best individual and fitness value.

new_population.extend([offspring, offspring])
population = new_population[:pop_size].

LAB:-1 Genetic Algorithm for optimization problems

Genetic algorithm code:-

```
import numpy as np
import random

def objective_function(x):
    return x ** 2

population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
range_min = -10
range_max = 10

# Create initial population
def initialize_population(size, min_val, max_val):
    return np.random.uniform(min_val, max_val, size)

# Evaluate fitness of the population
def evaluate_fitness(population):
    return np.array([objective_function(x) for x in population])

# Selection using roulette-wheel method
def selection(population, fitness):
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    return population[np.random.choice(range(len(population)), size=2, p=probabilities)]
```

```
# Crossover between two parents
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        return (parent1 + parent2) / 2  # Simple averaging for crossover
    return parent1  # No crossover

# Mutation of an individual
def mutate(individual):
    if random.random() < mutation_rate:
        return np.random.uniform(range_min, range_max)
    return individual

# Genetic Algorithm function
def genetic_algorithm():
    # Step 1: Initialize population
    population = initialize_population(population_size, range_min, range_max)

    for generation in range(num_generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        best_index = np.argmax(fitness)
        best_solution = population[best_index]
        best_fitness = fitness[best_index]

        # print(f"Generation {generation + 1}: Best Solution = {best_solution}, Fitness =
{best_fitness}")
```

```
        # Step 3: Create new population
        new_population = []
        for _ in range(population_size):
            # Select parents
            parent1, parent2 = selection(population, fitness)
            # Crossover to create offspring
            offspring = crossover(parent1, parent2)
            # Mutate offspring
            offspring = mutate(offspring)
            new_population.append(offspring)

        # Step 6: Replace old population with new population
        population = np.array(new_population)

    return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm()
print(f"Best Solution Found: {best_solution}, Fitness: {best_fitness}")
```

OUTPUT:-

best solution found :- -7.290085674589, fitness: 82.305974562

{2024/11/7 Thursday}

2) Particle Swarm optimization

Purpose

Initialize swarm with N particles
For each particle:

Initialize position swarm[i].positions randomly within problem bounds

Initialize velocity swarm[i].velocity randomly
Initialize best position swarm[i].bestpos = swarm[i].position

Initialize best Fitness swarm[i].bestfitness = Fitness(swarm[i].position)

w = inertia weight
c_1 = cognitive co-efficient
c_2 = social co-efficient
Minx = Lower bound of search space
Maxx = upper bound of search space

```
for iter in maxiter
    for i in N
        r1 = random number between 0 & 1
        r2 = random number between 0 & 1
        swarm[i].velocity = w*swarm[i].velocity +
            r1*c1*(swarm[i].bestpos -
            swarm[i].pos) + r2*c2*(bestpos
            - swarm - swarm[i].pos)

        swarm[i].pos += swarm[i].velocity
        if swarm[i].pos < minx
            swarm[i].pos = minx
        elif swarm[i]
```

swarm[i].pos = max x

current_fitness = fitness(swarm[i].pos)

if current_fitness < swarm[i].best_fitness

swarm[i].best_fitness = current_fitness.

swarm[i].best_pos = swarm[i].pos

if current_fitness < best_fitness_swarm

best_fitness_swarm = current_fitness

best_position_swarm = swarm[i].pos

Return best_pos_swarm

Lab:-2 Particle swarm optimization for function optimization:

code for pso

```
import numpy as np
import random

# Define the optimization problem (Rastrigin Function)
def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Particle Swarm Optimization (PSO) implementation
class Particle:
    def __init__(self, dimension, lower_bound, upper_bound):
        # Initialize the particle position and velocity randomly
        self.position = np.random.uniform(lower_bound, upper_bound, dimension)
        self.velocity = np.random.uniform(-1, 1, dimension)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin(self.position)

    def update_velocity(self, global_best_position, w, c1, c2):
        # Update the velocity of the particle
        r1 = np.random.rand(len(self.position))
        r2 = np.random.rand(len(self.position))

        # Inertia term
        inertia = w * self.velocity

        # Cognitive term (individual best)
```

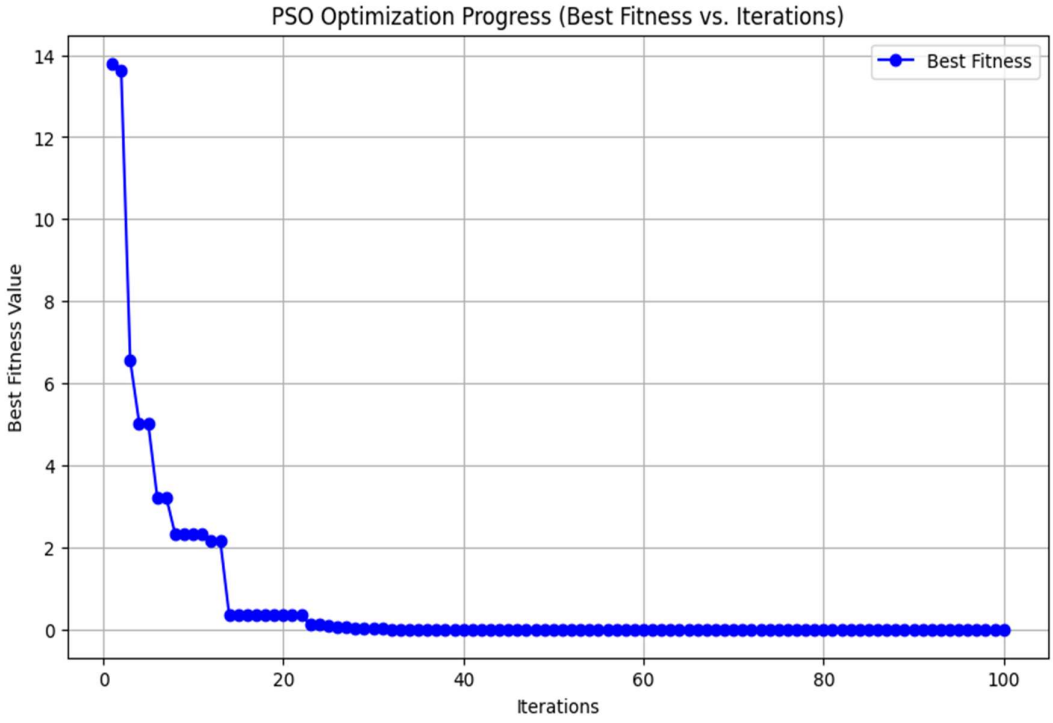
```
        cognitive = c1 * r1 * (self.best_position - self.position)

        # Social term (global best)
        social = c2 * r2 * (global_best_position - self.position)

        # Update velocity
        self.velocity = inertia + cognitive + social

    def update_position(self, lower_bound, upper_bound):
        # Update the position of the particle
        self.position = self.position + self.velocity

        # Ensure the particle stays within the bounds
        self.position = np.clip(self.position, lower_bound, upper_bound)

    def evaluate(self):
        # Evaluate the fitness of the particle
        fitness = rastrigin(self.position)

        # Update the particle's best position if necessary
        if fitness < self.best_value:
            self.best_value = fitness
            self.best_position = np.copy(self.position)

def particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles=30,
max_iter=100, w=0.5, c1=1.5, c2=1.5):
    # Initialize particles
    particles = [Particle(dim, lower_bound, upper_bound) for _ in range(num_particles)]

    # Initialize the global best position and value
    global_best_position = particles[0].best_position
```

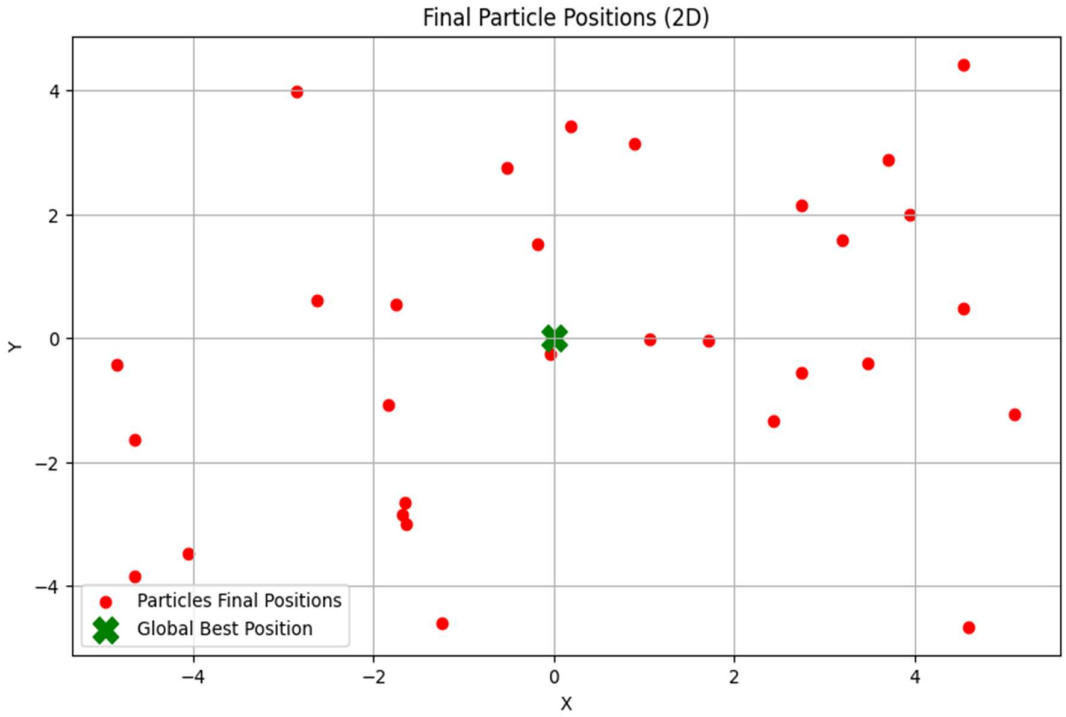
```
    global_best_value = particles[0].best_value

    for i in range(max_iter):
        # Update each particle
        for particle in particles:
            particle.update_velocity(global_best_position, w, c1, c2)
            particle.update_position(lower_bound, upper_bound)
            particle.evaluate()

            # Update global best position if needed
            if particle.best_value < global_best_value:
                global_best_value = particle.best_value
                global_best_position = np.copy(particle.best_position)

        # Optionally print the progress
        if (i+1 ) % 10 == 0:
            print(f"Iteration {i+1 }/{max_iter} - Best Fitness: {global_best_value}")

    return global_best_position, global_best_value

# Set the parameters for the PSO algorithm
dim = 2              # Number of dimensions for the function
lower_bound = -5.12    # Lower bound of the search space
upper_bound = 5.12     # Upper bound of the search space
num_particles = 30     # Number of particles in the swarm
max_iter = 100         # Number of iterations

# Run the PSO
best_position, best_value = particle_swarm_optimization(dim, lower_bound, upper_bound,
num_particles, max_iter)
```

```
# Output the best solution found
print("\nBest Solution Found:")
print("Position:", best_position)
print("Fitness:", best_value)
```

OUTPUT:

```
Iteration 10/100 - Best Fitness: 2.3145203625443997
Iteration 20/100 - Best Fitness: 0.34026142761705813
Iteration 30/100 - Best Fitness: 0.0158886712260653
Iteration 40/100 - Best Fitness: 5.572809527620848e-06
Iteration 50/100 - Best Fitness: 3.493363465167931e-08
Iteration 60/100 - Best Fitness: 2.8475000135586015e-11
Iteration 70/100 - Best Fitness: 1.4210854715202004e-14
Iteration 80/100 - Best Fitness: 0.0
Iteration 90/100 - Best Fitness: 0.0
Iteration 100/100 - Best Fitness: 0.0

Best Solution Found:
Position: [ 1.64289135e-09 -1.88899730e-09]
Fitness: 0.0
```




Final Particle Positions (2D)
Particles Final Positions
Global Best Position
X
Y
4
2
0
−2
−4
−4
−2
0
2
4

(2024/11/14)
Thursday

3) Ant colony optimization for traveling Salesman problem.

1) Define the problem - Create a set of cities with coordinate.

Pseudocode.

```
cities = array of city coordinates
function calculate_distance(cities):
    num_cities = length(cities)
    distance = matrix of size num_cities x num_cities.

    for each city i:
        for each city j:
            distance[i][j] = Euclidean
distance between city i and city j
return distance.
```

The function distance computes the pairwise Euclidean distance between all cities.

2) Initialize parameters

Pseudocode.

```
num_ants = no. of ants.
alpha = influence of pheromone
beta = influence of heuristic (inverse distance)
rho = evaporation rate of pheromone
num_iterations = no. of iterations
initial_pheromone = initial pheromone value
                    on all edge
```

distance = calculate distance (cities)

pheromone_matrix = matrix of size num_cities x num_cities filled with initial pheromone

eta = matrix of heuristic information (1/distance)

* Set up the algorithm parameter. Such as the number of ant, the influence of pheromone and heuristic information, the pheromone evaporation rate and the no. of iteration.

* Initialize the pheromone matrix where all values start with the same initial pheromone level.
* Calculate the heuristic matrix eta, which is the inverse of the distance matrix.

3) Heuristic function

function heuristic(distance):

create an empty matrix eta of the same size as distance.

For i=0 to number.

eta[i][j] = 1/distance[i][j]

else.

eta[i][j] = 0

return heuristic matrix (eta).

4) construct-solution(pheromone eta)

→ Each ant starts at a random city and probabilistic-ally construct a tour by selecting the next city using the pheromone & heristic information

```
Pheromone *= (1-rho)
    for each tour in all_tours:
        tour_length = compute total length of the
tour using distance.
        for each city pair in the tour:
            Pheromone [city1, city2] += 1/tour_length

best_length = computer total length of best_tour
    for each city pair in best_tour:
        pheromone [city1, city2] += 1/best_length.
```

plot_route(cities, best_tour)

→ The plot_route function generates a graphical representation of best tour found by ants.

Input:

```
distance_matrix = np.array([
        [0, 10, 12, 11]
        [10, 0, 13, 15]
        [12, 13, 0, 10]
        [11, 15, 10, 0]
    ])
```

Sample output

Best tour (0, 3, 2, 1)

Best tour length: 36

Lab:-3 Ant Colony Optimization for Travelling Salesman problem:

CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Define the Problem: Create a set of cities with their coordinates
cities = np.array([
    [0, 0],  # City 0
    [1, 5],  # City 1
    [5, 1],  # City 2
    [6, 4],  # City 3
    [7, 8],  # City 4
])

# Calculate the distance matrix between each pair of cities
def calculate_distances(cities):
    num_cities = len(cities)
    distances = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(num_cities):
            distances[i][j] = np.linalg.norm(cities[i] - cities[j])

    return distances

distances = calculate_distances(cities)

# 2. Initialize Parameters
```

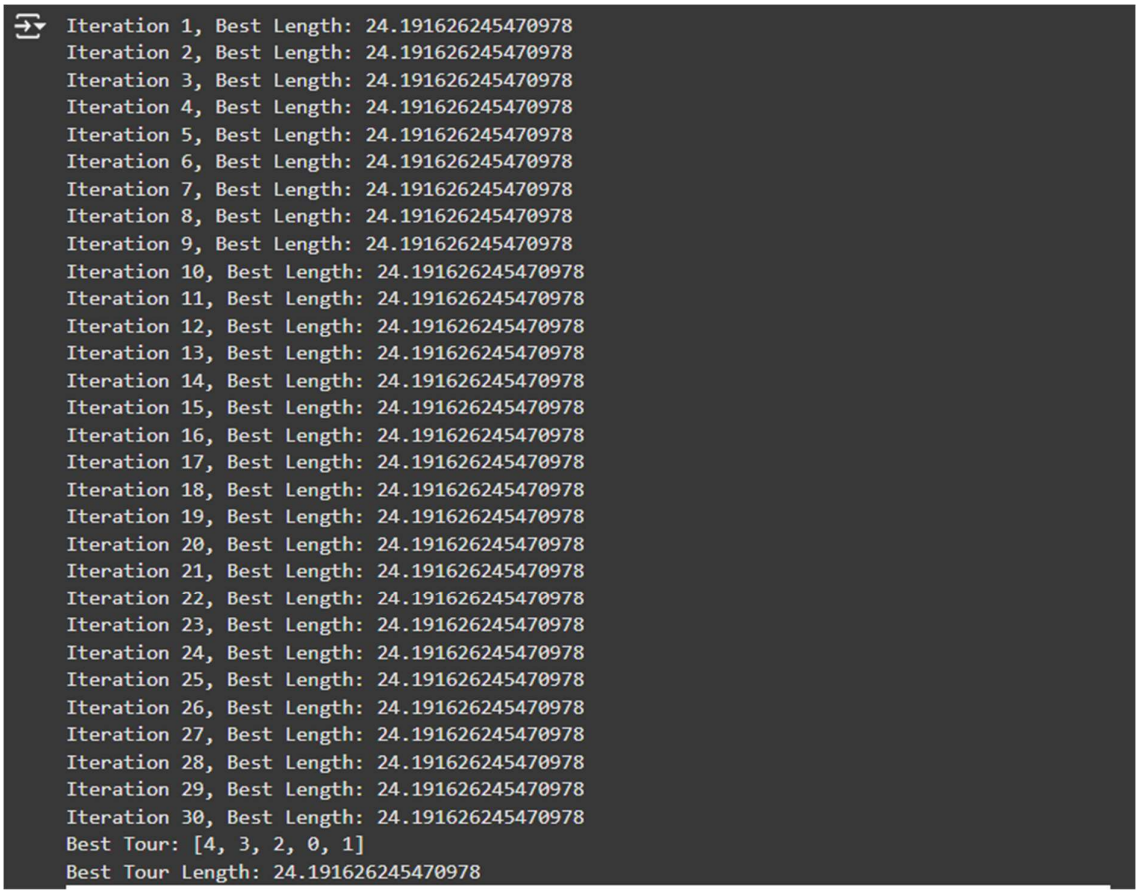
```
num_ants = 10
num_cities = len(cities)
alpha = 1.0  # Influence of pheromone
beta = 5.0   # Influence of heuristic (inverse distance)
rho = 0.5    # Evaporation rate
num_iterations = 30
initial_pheromone = 1.0

# Pheromone matrix initialization
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# 3. Heuristic information (Inverse of distance)
def heuristic(distances):
    with np.errstate(divide='ignore'):  # Ignore division by zero
        return 1 / distances

eta = heuristic(distances)

# 4. Choose next city probabilistically based on pheromone and heuristic info
def choose_next_city(pheromone, eta, visited):
    probs = []
    for j in range(num_cities):
        if j not in visited:
            pheromone_ij = pheromone[visited[-1], j] ** alpha
            heuristic_ij = eta[visited[-1], j] ** beta
            probs.append(pheromone_ij * heuristic_ij)
        else:
            probs.append(0)
    probs = np.array(probs)
    return np.random.choice(range(num_cities), p=probs / probs.sum())
```

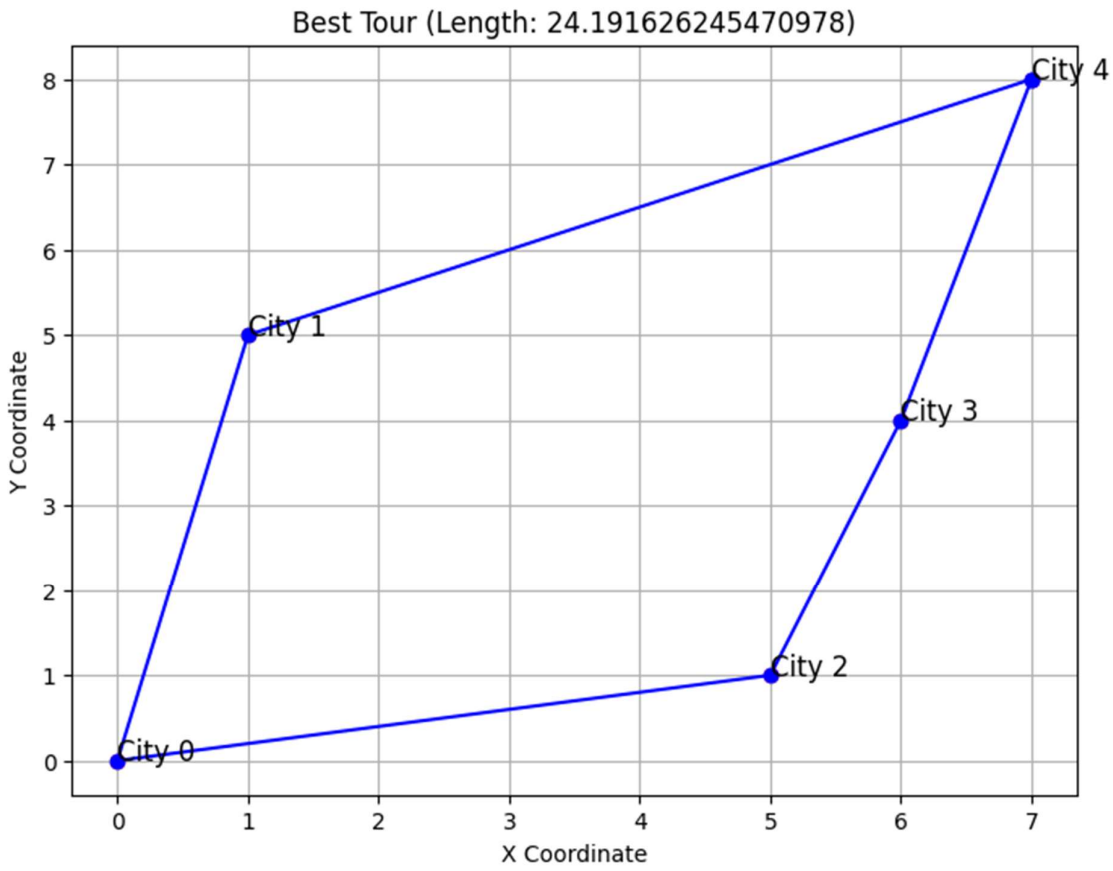
```
# Construct solution for a single ant
def construct_solution(pheromone, eta):
    tour = [np.random.randint(0, num_cities)]
    while len(tour) < num_cities:
        next_city = choose_next_city(pheromone, eta, tour)
        tour.append(next_city)
    return tour

# 5. Update pheromones after all ants have constructed their tours
def update_pheromones(pheromone, all_tours, distances, best_tour):
    pheromone *= (1 - rho)  # Evaporate pheromones

    # Add pheromones for each ant's tour
    for tour in all_tours:
        tour_length = sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)])
        for i in range(-1, num_cities - 1):
            pheromone[tour[i], tour[i + 1]] += 1.0 / tour_length

    # Increase pheromones on the best tour
    best_length = sum([distances[best_tour[i], best_tour[i + 1]] for i in range(-1, num_cities - 1)])
    for i in range(-1, num_cities - 1):
        pheromone[best_tour[i], best_tour[i + 1]] += 1.0 / best_length

# 6. Main ACO Loop: Iterate over multiple iterations to find the best solution
def run_aco(distances, num_iterations):
    pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_tours = [construct_solution(pheromone, eta) for _ in range(num_ants)]
```

```
        all_lengths = [sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)]) for tour in
all_tours]

        current_best_length = min(all_lengths)
        current_best_tour = all_tours[all_lengths.index(current_best_length)]

        if current_best_length < best_length:
            best_length = current_best_length
            best_tour = current_best_tour

        update_pheromones(pheromone, all_tours, distances, best_tour)

        print(f"Iteration {iteration + 1}, Best Length: {best_length}")

    return best_tour, best_length

# Run the ACO algorithm
best_tour, best_length = run_aco(distances, num_iterations)

# 7. Output the Best Solution
print(f"Best Tour: {best_tour}")
print(f"Best Tour Length: {best_length}")

# 8. Plot the Best Route
def plot_route(cities, best_tour):
    plt.figure(figsize=(8, 6))
    for i in range(len(cities)):
        plt.scatter(cities[i][0], cities[i][1], color='red')
        plt.text(cities[i][0], cities[i][1], f"City {i}", fontsize=12)

    # Plot the tour as lines connecting the cities
```

```
    tour_cities = np.array([cities[i] for i in best_tour] + [cities[best_tour[0]]])  # Complete the loop by returning to the start

    plt.plot(tour_cities[:, 0], tour_cities[:, 1], linestyle='-', marker='o', color='blue')


    plt.title(f"Best Tour (Length: {best_length})")

    plt.xlabel("X Coordinate")

    plt.ylabel("Y Coordinate")

    plt.grid(True)

    plt.show()


# Call the plot function

plot_route(cities, best_tour)
```

OUTPUT:




Best Tour (Length: 24.191626245470978)
City 0
City 1
City 2
City 3
City 4
X Coordinate
Y Coordinate
0
1
2
3
4
5
6
7
8

21/11/24
Thursday

4) Cuckoo search (CS)

1) Define the problems

The problem is defined by an objective function. In this case, we use the sphere function.

Pseudo codes:

Objective function f(x) = sum(x_i^2)

2) Initialize parameter

Parameters such as the number of nest, no. of iterations, discovery rate, and search space boundaries are initialized.

Pseudo code:

```
num_nest = 25
num_iterations = 100
discovery_rate = 0.25
dim = 5
lower_bound = -10
upper_bound = 10
```

3) Initialize population:

A population of nest is randomly initialize with the defined boundary.

Pseudo code

```
nest = random.uniform(lower_bound, upper_bound,
                      num_nest, dim).
Fitness = [evaluate_fitness(nest) for nest in nest]
```

4) Evaluate Fitness:

The fitness of each nest is evaluate using the objective function. The best nest is selected based on the lowest fitness value.

Pseudocode

```
best_nest = min(nest, by = Fitness)
best_fitness = min(fitness)
```

5) Generate New Solutions

New solution are generated by a levy flight, which help explore the search space more effectively.

Pseudocode

```
for each nest:
    step_size = levy_flight(lambda)
    new_solution = nest + step_size * (nest - best_nest)
new_solution = clip(new_solution, lower_bound, upper_bound)
fitness evaluate_fitness(new_solution).
```

6) Abandon worst nest.

x some of the worst nest are abandoned and replaced with random solution based on the discovery rate

Pseudocode:

for each nest - with a probability discovery - rate
new - random - solution = random - uniform (lower bound, upper - bound)
fitness = evaluate - fitness (new - random - solution)

7) iterate.

The algorithm repeats the process of generating new solution updating nest, and evaluate fitness over multiple iterations.

Pseudocode:

for iteration in range (num - iterations),
generate - new - solution ( )
update - best - nest ( )
track - best - fitness ( )

8) output the Best Solution.

After completing the iterations, the nest best found it returned, representing the optimal solution

Pseudocode:-

return best - nest, best fitness

Application:

* Structural optimization, Aerospace design, feature selection, Hyper parameter Tuning.

Sneha SB 28/11/24

LAB-4 : Cuckoo Search (CS):

CODE:

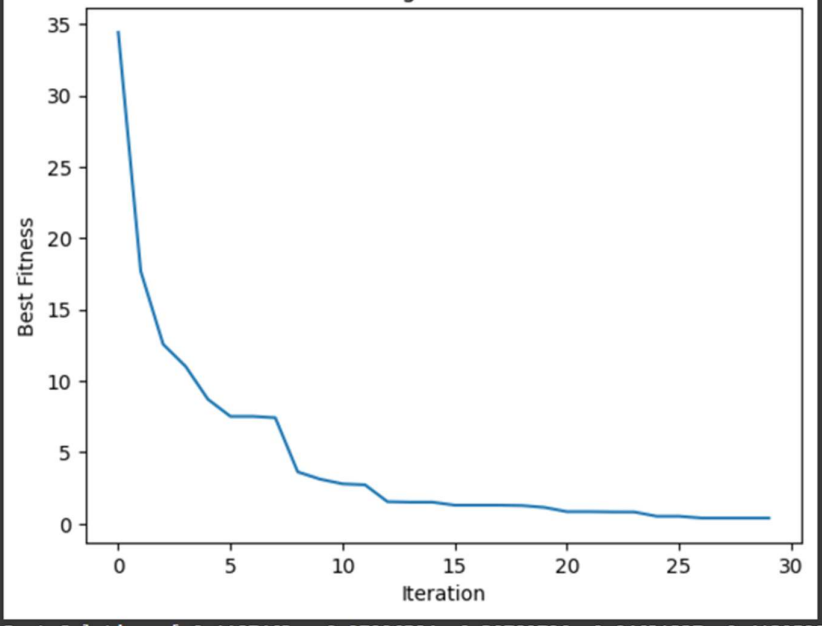
```
#cuckoo search
import numpy as np
import random
import math
import matplotlib.pyplot as plt

# Define a sample function to optimize (Sphere function in this case)
def objective_function(x):
    return np.sum(x ** 2)

# Lévy flight function
def levy_flight(Lambda):
    sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
            (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, size=1)
    v = np.random.normal(0, sigma_v, size=1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

# Cuckoo Search algorithm
def cuckoo_search(num_nests=25, num_iterations=100, discovery_rate=0.25, dim=5, lower_bound=-
10, upper_bound=10):
    # Initialize nests
    nests = np.random.uniform(lower_bound, upper_bound, (num_nests, dim))
    fitness = np.array([objective_function(nest) for nest in nests])

    # Get the current best nest
```

```
best_nest_idx = np.argmin(fitness)
best_nest = nests[best_nest_idx].copy()
best_fitness = fitness[best_nest_idx]

Lambda = 1.5  # Parameter for Lévy flights
fitness_history = []  # To track fitness at each iteration

for iteration in range(num_iterations):
    # Generate new solutions via Lévy flight
    for i in range(num_nests):
        step_size = levy_flight(Lambda)
        new_solution = nests[i] + step_size * (nests[i] - best_nest)
        new_solution = np.clip(new_solution, lower_bound, upper_bound)
        new_fitness = objective_function(new_solution)

        # Replace nest if new solution is better
        if new_fitness < fitness[i]:
            nests[i] = new_solution
            fitness[i] = new_fitness

    # Discover some nests with probability 'discovery_rate'
    random_nests = np.random.choice(num_nests, int(discovery_rate * num_nests), replace=False)
    for nest_idx in random_nests:
        nests[nest_idx] = np.random.uniform(lower_bound, upper_bound, dim)
        fitness[nest_idx] = objective_function(nests[nest_idx])

    # Update the best nest
    current_best_idx = np.argmin(fitness)
    if fitness[current_best_idx] < best_fitness:
        best_fitness = fitness[current_best_idx]
        best_nest = nests[current_best_idx].copy()
```

```
        fitness_history.append(best_fitness)
        print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

    plt.plot(fitness_history)
    plt.title('Fitness Convergence Over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Best Fitness')
    plt.show()
    return best_nest, best_fitness

best_nest, best_fitness = cuckoo_search(num_nests=30, num_iterations=100, dim=10,
lower_bound=-5, upper_bound=5)
print("Best Solution:", best_nest)
print("Best Fitness:", best_fitness)
```

OUTPUT:

```
Iteration 1/30, Best Fitness: 34.421347350368414
Iteration 2/30, Best Fitness: 17.701267864864427
Iteration 3/30, Best Fitness: 12.572246094152595
Iteration 4/30, Best Fitness: 11.025968548544025
Iteration 5/30, Best Fitness: 8.713786692960158
Iteration 6/30, Best Fitness: 7.5206125475077785
Iteration 7/30, Best Fitness: 7.5206125475077785
Iteration 8/30, Best Fitness: 7.426062303628502
Iteration 9/30, Best Fitness: 3.6305424687807872
Iteration 10/30, Best Fitness: 3.122312407680085
Iteration 11/30, Best Fitness: 2.7935374916676268
Iteration 12/30, Best Fitness: 2.7258275326189683
Iteration 13/30, Best Fitness: 1.5451154817432429
Iteration 14/30, Best Fitness: 1.5138101828809285
Iteration 15/30, Best Fitness: 1.5138101828809285
Iteration 16/30, Best Fitness: 1.300269684490209
Iteration 17/30, Best Fitness: 1.300269684490209
Iteration 18/30, Best Fitness: 1.300269684490209
Iteration 19/30, Best Fitness: 1.2738498249584989
Iteration 20/30, Best Fitness: 1.1445834652176474
Iteration 21/30, Best Fitness: 0.8487556087655604
Iteration 22/30, Best Fitness: 0.8487556087655604
Iteration 23/30, Best Fitness: 0.8289231635578032
Iteration 24/30, Best Fitness: 0.8242402471719793
Iteration 25/30, Best Fitness: 0.5258270013075049
Iteration 26/30, Best Fitness: 0.5258270013075049
Iteration 27/30, Best Fitness: 0.3996236442626478
Iteration 28/30, Best Fitness: 0.3996236442626478
Iteration 29/30, Best Fitness: 0.3996236442626478
Iteration 30/30, Best Fitness: 0.3996236442626478
```




```
Best Solution: [ 0.1187463  -0.27226584  0.30789726 -0.04631337 -0.41395998  0.0566571
 -0.05240855 -0.11176247  0.15695263 -0.00202369]
Best Fitness: 0.3996236442626478
```

2024/11/28
Thursday

Grey wolf optimizer (GWO)
Algorithm

1) Define the problem (objective function)

Define the objective function that takes position of a wolf (a candidate solution) and returns the function's value at that position. The value is called the fitness. Here.

Pseudocode:

Obj_fun = x^2

```
FUNCTION objective_function(position):
    sum = 0
    For i in range(0, len(position)):
        sum += position[i]^2
    Return sum
```

2) Initialize the parameters:

Set the number of wolves (population size) the number of iteration and the bounds which the wolves will search for the optimal solutions

Pseudocodes

```
num_wolves = 30
num_dimension = 5
num_iteration = 100
bounds = [-10, 10]      # lb = -10, ub = 10,
```

3) Initialize population

Generate random position for all the wolves in the population within the defined bounds.

Pseudocode:- Function initialize_wolve(num_wolves, num_dimension, bounds)

```
Population = []
For i in range(num_wolves):
    position = []
    for d in range(num_dimension):
        position[d] = random_value_in(bounds)
    Population.append(position)
Return population
```

Explanation:- Each wolves position is initialize randomly within the search space bounds. Each wolf is a candidate solution represented by a vector in multi-dimensional space.

4) Evaluate Fitness (Rank wolves as Alpha, Beta, Delta)

Pseudocode:-

```
Fuction evaluate_fitness(population):
    alpha_fitness = infinity
    beta_fitness = infinity
    delta_fitness = infinity
for wolf in population:
    fitness = objective_function(wolf_position)
    if fitness < alpha_fitness:
        delta_fitness = beta_fitness
        delta_position = beta_position
```

beta_fitness = alpha_fitness
beta_position = alpha_positia
alpha_fitness = fitness
alpha_position = wolf_position.

Else if:
fitness < beta_fitness
delta_fitness = beta_fitness
delta_position = beta_position

else
if fitness < delta_fitness.
delta_fitness = fitness.
delta_position = wolf_positions

Return. alpha_position, beta_position, delta position

Explanation:- Each wolf's fitness is calculated using the objective function & the wolves are ranked. The top three wolves (with the best fitness values are classified as alpha, beta and delta wolves

Dd = |C1 * alpha_position - x[i]|
Db = |C2 * B_position - x[i]|
Ds = |C3 * s_position - x[i]|
C1, C2, C3 = 2 * r2

2) update the fitness of each wolf
fitness[i] = f(x[i])
b) sort the wolves based on their fitness value and update α, β, δ.

5) Return the best solution found (α wolf) after max iteration

5. Grey Wolf optimizer(GWO)

LAB-5 : Grey Wolf Optimizer (GWO):

CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the Problem (a mathematical function to optimize)
def objective_function(x):
    return np.sum(x**2)  # Example: Sphere function (minimize sum of squares)

# Step 2: Initialize Parameters
num_wolves = 5  # Number of wolves in the pack
num_dimensions = 2  # Number of dimensions (for the optimization problem)
num_iterations = 30  # Number of iterations
lb = -10  # Lower bound of search space
ub = 10  # Upper bound of search space

# Step 3: Initialize Population (Generate initial positions randomly)
wolves = np.random.uniform(lb, ub, (num_wolves, num_dimensions))

# Initialize alpha, beta, delta wolves
alpha_pos = np.zeros(num_dimensions)
beta_pos = np.zeros(num_dimensions)
delta_pos = np.zeros(num_dimensions)

alpha_score = float('inf')  # Best (alpha) score
beta_score = float('inf')   # Second best (beta) score
delta_score = float('inf')  # Third best (delta) score

# To store the alpha score over iterations for graphing
```

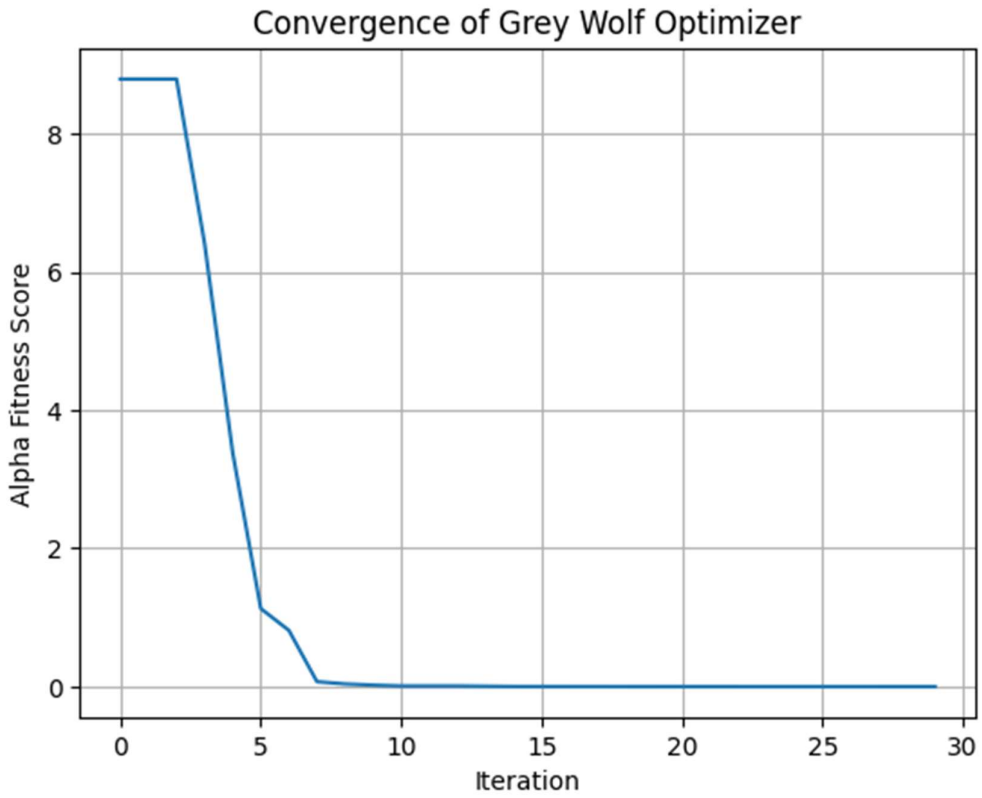
```
alpha_score_history = []

# Step 4: Evaluate Fitness and assign Alpha, Beta, Delta wolves
def evaluate_fitness():
    global alpha_pos, beta_pos, delta_pos, alpha_score, beta_score, delta_score

    for wolf in wolves:
        fitness = objective_function(wolf)

        # Update Alpha, Beta, Delta wolves based on fitness
        if fitness < alpha_score:
            delta_score = beta_score
            delta_pos = beta_pos.copy()

            beta_score = alpha_score
            beta_pos = alpha_pos.copy()

            alpha_score = fitness
            alpha_pos = wolf.copy()
        elif fitness < beta_score:
            delta_score = beta_score
            delta_pos = beta_pos.copy()

            beta_score = fitness
            beta_pos = wolf.copy()
        elif fitness < delta_score:
            delta_score = fitness
            delta_pos = wolf.copy()

# Step 5: Update Positions
def update_positions(iteration):
```

```
a = 2 - iteration * (2 / num_iterations)  # a decreases linearly from 2 to 0

for i in range(num_wolves):
    for j in range(num_dimensions):
        r1 = np.random.random()
        r2 = np.random.random()

        # Position update based on alpha
        A1 = 2 * a * r1 - a
        C1 = 2 * r2
        D_alpha = abs(C1 * alpha_pos[j] - wolves[i, j])
        X1 = alpha_pos[j] - A1 * D_alpha

        # Position update based on beta
        r1 = np.random.random()
        r2 = np.random.random()
        A2 = 2 * a * r1 - a
        C2 = 2 * r2
        D_beta = abs(C2 * beta_pos[j] - wolves[i, j])
        X2 = beta_pos[j] - A2 * D_beta

        # Position update based on delta
        r1 = np.random.random()
        r2 = np.random.random()
        A3 = 2 * a * r1 - a
        C3 = 2 * r2
        D_delta = abs(C3 * delta_pos[j] - wolves[i, j])
        X3 = delta_pos[j] - A3 * D_delta

        # Update wolf position
        wolves[i, j] = (X1 + X2 + X3) / 3
```

```
            # Apply boundary constraints
            wolves[i, j] = np.clip(wolves[i, j], lb, ub)

# Step 6: Iterate (repeat evaluation and position updating)
for iteration in range(num_iterations):
    evaluate_fitness()  # Evaluate fitness of each wolf
    update_positions(iteration)  # Update positions based on alpha, beta, delta

    # Record the alpha score for this iteration
    alpha_score_history.append(alpha_score)

    # Optional: Print current best score
    print(f"Iteration {iteration+1}/{num_iterations}, Alpha Score: {alpha_score}")

# Step 7: Output the Best Solution
print("Best Solution:", alpha_pos)
print("Best Solution Fitness:", alpha_score)

# Plotting the convergence graph
plt.plot(alpha_score_history)
plt.title('Convergence of Grey Wolf Optimizer')
plt.xlabel('Iteration')
plt.ylabel('Alpha Fitness Score')
plt.grid(True)
plt.show()
```

OUTPUT:

```
Iteration 1/30, Alpha Score: 8.789922247101906
Iteration 2/30, Alpha Score: 8.789922247101906
Iteration 3/30, Alpha Score: 8.789922247101906
Iteration 4/30, Alpha Score: 6.409956649485766
Iteration 5/30, Alpha Score: 3.383929841190778
Iteration 6/30, Alpha Score: 1.1292299489236237
Iteration 7/30, Alpha Score: 0.8136628488047792
Iteration 8/30, Alpha Score: 0.07110881373527288
Iteration 9/30, Alpha Score: 0.03823180120070083
Iteration 10/30, Alpha Score: 0.021111314445105462
Iteration 11/30, Alpha Score: 0.00874782100259989
Iteration 12/30, Alpha Score: 0.00874782100259989
Iteration 13/30, Alpha Score: 0.00874782100259989
Iteration 14/30, Alpha Score: 0.005066807028932165
Iteration 15/30, Alpha Score: 0.0011746187200998674
Iteration 16/30, Alpha Score: 0.0011746187200998674
Iteration 17/30, Alpha Score: 0.0008078646351838173
Iteration 18/30, Alpha Score: 0.0008078646351838173
Iteration 19/30, Alpha Score: 0.0006302256737926024
Iteration 20/30, Alpha Score: 0.0005272190797352655
Iteration 21/30, Alpha Score: 0.00035614966782860404
Iteration 22/30, Alpha Score: 0.0003270119398391142
Iteration 23/30, Alpha Score: 0.00022723766847392013
Iteration 24/30, Alpha Score: 0.00022152382849585967
Iteration 25/30, Alpha Score: 0.00022152382849585967
Iteration 26/30, Alpha Score: 0.00020102313789207912
Iteration 27/30, Alpha Score: 0.0001974565833678501
Iteration 28/30, Alpha Score: 0.0001547675581999543
Iteration 29/30, Alpha Score: 0.00014751518222697009
Iteration 30/30, Alpha Score: 0.00014751518222697009
Best Solution: [ 0.00643925 -0.01029812]
Best Solution Fitness: 0.00014751518222697009
```


Convergence of Grey Wolf Optimizer
Alpha Fitness Score
Iteration
0
2
4
6
8
0
5
10
15
20
25
30

6) parallel Cellular Algorithm and programs.

Function parallel cellular Algorithm

```
Define objective function()
    Initialize Grid (Grid size)
    For each cell in the grid:
        Randomly initialize cells position in the
        solution space

# Evaluate Fitness
    For each cell in the grid.
        Evaluate Fitness (cell)

# Update States.
    For iteration = 1 to max Iterations
        For each cell in the grid.
            neighbours = Get neighbours (cell, neighbo
            new state = calculate new state (
                        cell, neighbours)
            cell.state = new state

        For each in the grid:
            Evaluate Fitness (cell

        Track Best solution()

    Output Best Solution()
```

Initialize parameters.

```
Set GridSize = (rows, cols)
Set NumCell = rows * cols.
Set maxIteration = 100
Set convergence Threshold = 0.001
```

Function Initialize Grid (Gridsize):
    Initialize cells with random positions

Function Evaluate Fitness (cell):
    cell.fitness = objective_function(cell.state)

Function Get neighbours (cell, neighbourhood)
    return the list of neighbouring cells.

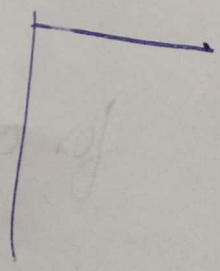
Function calculate new state (cell, neighbours)
    Return new state for the cell based
    on predefined

Function Track Best Solution()
    Best Solution = min(best solution, cell)

Application

-> weather modeling

-> ~~Natural disaster~~



Lab:-6

Parallel Cellular Algorithms and Programs

CODE:

```
#pcap import
numpy as np


# Define the problem: A simple op miza on func on (e.g., Sphere Func on) def
op miza on_func on(posi on):
    """Example: Sphere Func on for minimiza on."""
return sum(x**2 for x in posi on)


# Ini alize Parameters
GRID_SIZE = (10, 10)  # Grid size (rows, columns)
NEIGHBORHOOD_RADIUS = 1  # Moore neighborhood radius
DIMENSIONS = 2  # Number of dimensions in the solu on space
ITERATIONS = 30  # Number of itera ons


# Ini alize Popula on def ini alize_popula on(grid_size, dimensions):     """Ini alize a grid
with random posi ons."""     popula on = np.random.uniform(-10, 10, size=(grid_size[0],
grid_size[1], dimensions))     return popula on


# Evaluate Fitness def
evaluate_fitness(popula on):
    """Calculate the fitness of all cells."""     fitness =
np.zeros((popula on.shape[0], popula on.shape[1]))     for i in
range(popula on.shape[0]):         for j in range(popula
on.shape[1]):
            fitness[i, j] = op miza on_func on(popula on[i, j])
    return fitness
```

Lab:-6

```
# Get Neighborhood def
get_neighborhood(grid, x, y, radius):
   """Get the neighbors of a cell within the specified
radius."""     neighbors = []     for i in range(-radius, radius +
1):         for j in range(-radius, radius + 1):             if i == 0
and j == 0:                 con nue  # Skip the current cell
          ni, nj = x + i, y + j             if 0 <= ni <
grid.shape[0] and 0 <= nj < grid.shape[1]:
              neighbors.append((ni, nj))
return neighbors


# Update States def
update_states(popula on, fitness):
   """Update the state of each cell based on its neighbors."""     new_popula on =
np.copy(popula on)    for i in range(popula on.shape[0]):       for j in range(popula
on.shape[1]):                         neighbors  =  get_neighborhood(popula  on,  i,  j,
NEIGHBORHOOD_RADIUS)
          best_neighbor = popula on[i, j]
best_fitness = fitness[i, j]


          # Find the best posi on among neighbors
          for ni, nj in neighbors:
             if fitness[ni, nj] < best_fitness:
best_fitness = fitness[ni, nj]
best_neighbor = popula on[ni, nj]             # Update
the cell state (move towards the best neighbor)
new_popula on[i, j] = (popula on[i, j] +
best_neighbor) / 2  # Average posi on     return
new_popula on
```

Lab:-6

```
# Main Algorithm def
parallel_cellular_algorithm():
   """Implementa on of the Parallel Cellular Algorithm."""
popula on = ini alize_popula on(GRID_SIZE, DIMENSIONS)
best_solu on = None
   best_fitness = float('inf')

   for itera on in range(ITERATIONS):
      # Evaluate fitness        fitness =
evaluate_fitness(popula on)

      # Track the best solu on        min_fitness = np.min(fitness)        if min_fitness <
best_fitness:            best_fitness = min_fitness            best_solu on = popula
on[np.unravel_index(np.argmin(fitness), fitness.shape)]

      # Update states based on neighbors
popula on = update_states(popula on, fitness)

      # Print progress        print(f"Itera on {itera on + 1}: Best
Fitness = {best_fitness}")

   print("\nBest Solu on Found:")     print(f"Posi on:
{best_solu on}, Fitness: {best_fitness}")

# Run the algorithm
if __name__ == "__main__":
parallel_cellular_algorithm()
```

OUTPUT:

Lab:-6

```
Iteration 1: Best Fitness = 0.43918427791098213
Iteration 2: Best Fitness = 0.43918427791098213
Iteration 3: Best Fitness = 0.062221279350329436
Iteration 4: Best Fitness = 0.030149522005462108
Iteration 5: Best Fitness = 0.015791278460696168
Iteration 6: Best Fitness = 0.0025499667118763104
Iteration 7: Best Fitness = 0.0025499667118763104
Iteration 8: Best Fitness = 0.00019007166980743008
Iteration 9: Best Fitness = 0.00019007166980743008
Iteration 10: Best Fitness = 1.0432171933623911e-05
Iteration 11: Best Fitness = 8.406928148912647e-06
Iteration 12: Best Fitness = 5.511032710180021e-07
Iteration 13: Best Fitness = 4.3084388056725156e-07
Iteration 14: Best Fitness = 2.315054420755622e-07
Iteration 15: Best Fitness = 5.245753459404661e-08
Iteration 16: Best Fitness = 5.245753459404661e-08
Iteration 17: Best Fitness = 4.341357920017173e-08
Iteration 18: Best Fitness = 1.145644119860328e-08
Iteration 19: Best Fitness = 3.147791691706415e-09
Iteration 20: Best Fitness = 2.8192306881167533e-09
Iteration 21: Best Fitness = 9.788374665398935e-11
Iteration 22: Best Fitness = 9.788374665398935e-11
Iteration 23: Best Fitness = 9.788374665398935e-11
Iteration 24: Best Fitness = 9.788374665398935e-11
Iteration 25: Best Fitness = 7.537171686605552e-11
Iteration 26: Best Fitness = 7.234639306921671e-11
Iteration 27: Best Fitness = 7.028872029493468e-11
Iteration 28: Best Fitness = 3.340290444524624e-11
Iteration 29: Best Fitness = 1.4953679944431498e-11
Iteration 30: Best Fitness = 1.0817118995466254e-11

Best Solution Found:
Position: [-2.92599538e-06 -1.50188883e-06], Fitness: 1.0817118995466254e-11
```

→ Optimization via Gene Expression Algorithm:

Function 'Gene Expression Algorithm

#Step 1
Define objective function()

#Step 2: Initialize parameters.
set population = 100, NumGenes = 10, mutation Rate = 0.05,

CrossoverRate = 0.7, NumGeneration = 1000

# step 3: Initialize population
population = Initialize population(population.size, NumGenes)

#Step 4: Evaluate fitness
For each individual in population:
Evaluate Fitness(individual)

For generation in 1 to Num Generations.
Selected = select individuals(population)
offspring = crossover(selected)
mutate offspring(offspring, mutationRate)

for each individual in offspring:
individual.solution = Decode Genes(individual.genes)

Evaluate fitness(individual)
Track Best Solution()

Output Best solution.

```
Function Initialize Population (populationSize, numGenes):
    Return [generate Random Genes(numGenes)]

Function Evaluate Fitness (individual):
    individual.fitness = objectiveFunction(individual.genes)

Function crossover(selected)
    Return [perform crossover(selected[i], selected[i+1])

Function mutate offspring (offspring, mutationRate):
    for each individual in offspring:
        if Random() < mutationRate

Function TrackBestSolution():
    bestSolution = min(bestSolution, individuals)
```

Lab:-7

Optimization on via Gene Expression Algorithms

CODE:

```
import numpy as np import
random


# 1. Define the Problem: Op miza on Func on (e.g., Sphere Func on)
def op miza on_func on(solu on):
    """Sphere Func on for minimiza on (fitness evalua on)."""
return sum(x**2 for x in solu on)


# 2. Ini alize Parameters
POPULATION_SIZE = 50  # Number of gene c sequences (solu ons)
GENES = 5  # Number of genes per solu on
MUTATION_RATE = 0.1  # Probability of muta on
CROSSOVER_RATE = 0.7  # Probability of crossover
GENERATIONS = 30  # Number of genera ons to evolve


# 3. Ini alize Popula on def ini
alize_popula on(pop_size, genes):
    """Generate ini al popula on of random gene c sequences."""
return np.random.uniform(-10, 10, (pop_size, genes))


# 4. Evaluate Fitness def
evaluate_fitness(popula on):
    """Evaluate the fitness of each gene c sequence."""      fitness = [op
miza on_func on(solu on) for solu on in popula on]
    return np.array(fitness)
```

```
# 5. Selec on: Tournament Selec on
def select_parents(popula on, fitness, num_parents):
"""Select parents using tournament selec on."""
parents = []     for _ in range(num_parents):
       tournament = random.sample(range(len(popula on)), 3)  # Randomly select 3 candidates
best = min(tournament, key=lambda idx: fitness[idx])          parents.append(popula on[best])
return np.array(parents)

# 6. Crossover: Single-Point Crossover
def crossover(parents, crossover_rate):
   """Perform crossover between pairs of parents."""
offspring = []     for i in range(0, len(parents), 2):
       if i + 1 >= len(parents):
          break         parent1, parent2 = parents[i],
parents[i + 1]         if random.random() <
crossover_rate:
          point = random.randint(1, len(parent1) - 1)  # Single crossover
point             child1 = np.concatenate((parent1[:point], parent2[point:]))
child2 = np.concatenate((parent2[:point], parent1[point:]))
       else:
          child1, child2 = parent1, parent2  # No
crossover         offspring.extend([child1, child2])
return np.array(offspring)

# 7. Muta on def mutate(offspring, muta
on_rate):     """Apply muta on to introduce
variability."""     for i in range(len(offspring)):
for j in range(len(offspring[i])):
          if random.random() < muta on_rate:
```

```
            offspring[i][j] += np.random.uniform(-1, 1)  # Random small change
return offspring

# 8. Gene Expression: Func onal Solu on (No transforma on needed for this case) def
gene_expression(popula on):
    """Translate gene c sequences into func onal solu ons."""      return
popula on  # Gene c sequences directly represent solu ons here.

# 9. Main Func on: Gene Expression Algorithm
def gene_expression_algorithm():
    """Implementa on of Gene Expression Algorithm for op miza on."""
    # Ini alize popula on          popula on = ini alize_popula
on(POPULATION_SIZE, GENES)     best_solu on = None
    best_fitness = float('inf')

    for genera on in range(GENERATIONS):
        # Evaluate fitness         fitness =
evaluate_fitness(popula on)

        # Track the best solu on         min_fitness_idx
= np.argmin(fitness)       if fitness[min_fitness_idx]
<  best_fitness:                          best_fitness  =
fitness[min_fitness_idx]                 best_solu on =
popula on[min_fitness_idx]

        # Selec on         parents = select_parents(popula on, fitness,
POPULATION_SIZE // 2)

        # Crossover
```

```
        offspring = crossover(parents, CROSSOVER_RATE)
offspring = mutate(offspring, MUTATION_RATE)
        # Gene Expression         popula on =
gene_expression(offspring)

        # Print progress         print(f"Genera on {genera on + 1}: Best
Fitness = {best_fitness}")

   # Output the best solu on         print("\nBest Solu on
Found:")         print(f"Posi on: {best_solu on}, Fitness:
{best_fitness}")

if __name__ == "__main__":
gene_expression_algorithm()
```

OUTPUT:

```
Generation 1: Best Fitness = 55.82997756903893
Generation 2: Best Fitness = 26.410565738143625
Generation 3: Best Fitness = 21.857647823851615
Generation 4: Best Fitness = 20.016914182036285
Generation 5: Best Fitness = 20.016914182036285
Generation 6: Best Fitness = 20.016914182036285
Generation 7: Best Fitness = 13.81760087982789
Generation 8: Best Fitness = 13.81760087982789
Generation 9: Best Fitness = 12.077725051361178
Generation 10: Best Fitness = 10.461698723345474
Generation 11: Best Fitness = 8.933105023570093
Generation 12: Best Fitness = 6.619449963941974
Generation 13: Best Fitness = 3.1567413435369454
Generation 14: Best Fitness = 3.1567413435369454
Generation 15: Best Fitness = 3.1567413435369454
Generation 16: Best Fitness = 2.74585545305795
Generation 17: Best Fitness = 2.7031453676198964
Generation 18: Best Fitness = 2.078188177116774
Generation 19: Best Fitness = 1.5193087227027497
Generation 20: Best Fitness = 1.4413606561895607
Generation 21: Best Fitness = 0.8501569187378994
Generation 22: Best Fitness = 0.4209372164676112
Generation 23: Best Fitness = 0.3893761873774093
Generation 24: Best Fitness = 0.3893761873774093
Generation 25: Best Fitness = 0.3893761873774093
Generation 26: Best Fitness = 0.3741053651316379
Generation 27: Best Fitness = 0.1381555631914642
Generation 28: Best Fitness = 0.12238160343023853
Generation 29: Best Fitness = 0.12238160343023853
Generation 30: Best Fitness = 0.12238160343023853

Best Solution Found:
Position: [-0.03614343 -0.00257499  0.02260677  0.31412563  0.14792784], Fitness: 0.12238160343023853
```