VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sanjeet Prajwal Pandit (1BM22CS241)

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 **Sanjeet Prajwal Pandit (1BM22CS241),** 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/Sanjeet-108/BIS_LAB

<u>Program 1:</u> Genetic Algorithm for Optimization on Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

<u>Implementation Steps:</u>

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:

Page No.

Date. / /

(1) Genetic Algorithm for optimization problems:

```
function geneticAlgorithm():
    Initialize parameters (population_size (100), mutation_rate (10%), crossover_rate (70%),
        num_generations (50), range_min (-10), range_max (10))
    population = InitializePopulation(population_size, range_min, range_max)

    for generation in 1 to num_generations:
        fitness = EvaluateFitness(population)
        new_population = [ ]

        for i from 1 to population_size/2:
            parent1, parent2 = Selection(population, fitness)
            offspring1, offspring2 = Crossover(parent1, parent2)
            new_population.append(Mutate(offspring1))
            new_population.append(Mutate(offspring2))

        population = new_population
        best_fitness = Max(fitness)
        best_solution = population [ArgMax(fitness)]
        print("Generation", generation, "Best solution = ", best_solution,
              "Fitness = ", best_fitness)

    return best_solution, FitnessFunction(best_solution)
```

Page No.
Date. / /

→ (fitness) evaluation
function takes population as input and computes fitness for each individual by applying fitness_function. It returns an array of fitness values (return np.array([fitness_function(x) for x in population]))

→ total_fitness = np.sum(fitness)
selection_probs = fitness / total_fitness
return population [np.random.choice(len(population), size=2, p=selection_probs)]

(Selection) using roulette wheel
→ It calculates total fitness
→ computes probability of selection for each individual
→ uses these probabilities to select two parents for reproduction

→ (crossover)
→ this function combines two parents to create offspring
→ If random number less than crossover rate, it performs weighted combination of two parents
→ If not, it returns parent unchanged (no crossover)

→ (Mutation)
→ If random number less than mutation rate, it generates new random individual within specified range
→ If not, it returns individual unchanged

Code:

```
#lab-2: genetic
import numpy as np
import random

# Objective function to maximize
def objective_function(x):
    return x ** 2

# Initialize parameters
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: -9.290037411642935, Fitness: 86.30479510972536
```

Program 2: Particle Swarm Optimization for Function Optimization

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Page No.
Date. / /

(2) Particle Swarm Optimization (PSO)

(1) Define problem

$$f(x) = A\cdot n + \sum_{i=1}^{n} \left[x_i^2 - A\cdot\cos(2\pi x_i) \right]$$

where $A = 10$, n = dimensionality of input x

x_i is value of ith variable in input vector

Goal: to minimize the function

(2) Initialize parameters

$n \rightarrow$ no. of particles

$w \rightarrow$ Inertia weight (impact of prev velocity on curr velocity b/w 0.4 and 0.9)

$c1 \rightarrow$ cognitive coeff (controls particles own best position)

$c2 \rightarrow$ social coeff (best known position by swarm)

(3) Initialize particles

$x_i \rightarrow$ position

$v \rightarrow$ velocity

personal best position (pi)

personal best fitness (f(pi))

(4) Evaluate fitness

Rastrigin function:

fitness for curr position: score = f(x_i);

if score < personal_best_score[i]:

update personal best-position[i] to x_i

update personal_best-score[i] to score

(5) Update velocities and positions

generate random values, r1, r2, between 0 and 1

update velocity (w* vi(t) + c1*r1*(pers.best-pos[i] - pos[i]) +

(2) Particle Swarm Optimization (PSO)

(1) Define problem

$$f(a) = A \cdot n + \sum_{i=1}^{n} [x_i^2 - A \cdot \cos(2\pi x_i)]$$

where $A = 10$, n = dimensionality of input x

x_i is value of ith variable in input vector

Goal: to minimize the function

(2) Initialize parameters

$n \rightarrow$ no. of vehicles

$w \rightarrow$ Inertia weight (impact of prev velocity on curr velocity b/w 0.4 and 0.9)

$c1 \rightarrow$ cognitive coeff (controls particles own best position)

$c2 \rightarrow$ social coeff (best known position by swarm)

(3) Initialize particles

$x_i \rightarrow$ position

$v \rightarrow$ velocity

personal best position (p_i)

personal best fitness ($f(p_i)$)

(4) Evaluate fitness

Rastrigin function:

fitness for curr position: score = f(x_i);

if score < personal_best_score[i]:

update personal best_position[i] to x_i

update personal_best_score[i] to score

(5) Update velocities and positions

generate random values, r1, r2, between 0 and 1

update velocity (w* vi(t)+ c1*r1*(pers_best_pos[i] - pos[i]) + c2*r2*(global_best_pos - pos[i])

update position

Page No.
Date. / /

(6) Iterate

for each iteration t (t = 1 to num_iterations):

for each particle i (i = 1 to num_particles):

calculate fitness score at new position

if score is better than personal best, update global best solution

(7) Output best solution

After, return global best position & score

global minimum occurs at $x_i = 0$ (all $x \Rightarrow 0$)

i.e $An + (0 - A(n)) = An - An = 0$

7/11/24

Code:

```
#lab-3: 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: 1.1103296669969005
Iteration 20/100 - Best Fitness: 0.020031338560627887
Iteration 30/100 - Best Fitness: 2.788695226740856e-06
Iteration 40/100 - Best Fitness: 1.0778596895022474e-06
Iteration 50/100 - Best Fitness: 6.450946443692374e-10
Iteration 60/100 - Best Fitness: 2.0463630789890885e-11
Iteration 70/100 - Best Fitness: 1.0658141036401503e-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: [-1.63024230e-09  1.14735681e-09]
Fitness: 0.0
```

<u>Program 3:</u> Ant Colony Optimization for the Traveling Salesman Problem

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

<u>Implementation Steps:</u>

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:

Page No.
Date. / /

(3) Ant Colony optimization for Travelling Salesman problem

(i) Initialize TSP with cities coordinates
A list of cities with their (x,y) coordinates
eg city 0 : (0,0) → starting point
city 1 : (1,5)
city 2 : (3,1)
city 3 : (5,3)
city 4 : (6,6)
Distance matrix : between each pair of cities
$$d[i,j] = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

(ii) Calculate distance matrix for all pairs of cities
$$dist = \begin{pmatrix} 0 & d_{01} & d_{02} & d_{03} & d_{04} \\ d_{10} & 0 & d_{12} & d_{13} & d_{14} \\ d_{20} & d_{21} & 0 & d_{23} & d_{24} \\ d_{30} & d_{31} & d_{32} & 0 & d_{34} \\ d_{40} & d_{41} & d_{42} & d_{43} & 0 \end{pmatrix}$$

(d_{ij} represents euclidean distance b/w i & j)

(iii) Initialize Ant Colony Parameters
→ No. of ants - how many ants explore solution space in each iteration, each ant construct a tour through cities
→ Pheromone Importance (alpha α) - higher val means ants will prefer those paths with stronger pheromone trails
→ Distance Importance (beta) - dist b/w cities, higher val encourages ants to choose shorter paths
→ Pheromone Evaporation Rate (rho) - pheromones naturally evaporate over time. A higher evaporation rate leads to

Page No.
Date. / /

(iv) Iterate over solution construction process (Main ACO Loop)
for each iteration (1 to max-iterations), alg performs foll steps:
(a) Initialize lists for ants solution and costs (here costs update pheromone trails)
(b) construct sol^ns for each ant

choose-next-city()

$$P(i) = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{k \in unvisited cities}(\tau_{ik})^{\alpha} \cdot (\eta_{jk})^{\beta}}$$

$\tau_{ij} \rightarrow$ pheromone level from city i to j
$\eta_{ij} \rightarrow$ heuristic val $\left(\eta_{ij} = \frac{1}{d_{ij}}\right)$ (inverse of dist b/w cities i & j)

$\alpha, \beta \rightarrow$ parameters (Pheromone & dist)
visits cities until all are visited & calculates tour's cost
(c) update best sol^n (if shorter, becomes best sol^n)
(d) update pheromones (based on 5) rho

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{1}{\text{cost from } i \text{ to } j}$$

(e) Print iteration process

(v) output best solution and cost
best sol^n $\rightarrow$ sequence of cities that constitutes shortest tour
best cost $\rightarrow$ total dist of tour

Ex best route $\rightarrow$ [0, 1, 4, 3, 2] (order in which cities are iterated)

best cost $\rightarrow$ 19.35 $(0 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 0)$

$(d_{01} = 5.1,\ d_{14} = 5.1,\ d_{43} = 3.16,\ d_{32} = 2.83,\ d_{20} = 3.16)$
$(\Rightarrow 5.1 + 5.1 + 3.16 + 2.83 + 3.16 = 19.35)$

Code:

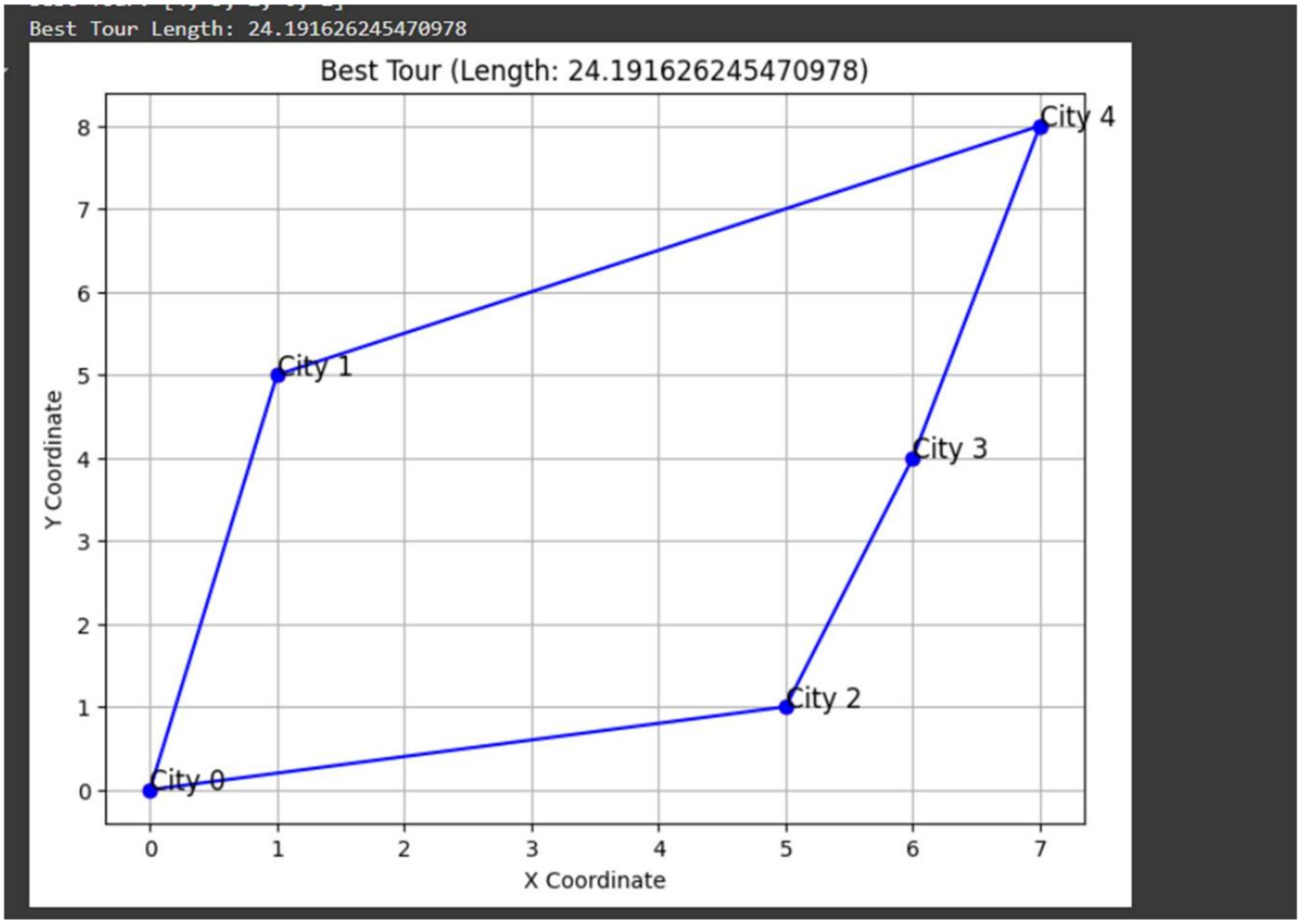
```
#ant colony
import numpy as np
import matplotlib.pyplot as plt

# 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:

```
Iteration 1, Best Length: 24.191626245470978
Iteration 2, Best Length: 24.191626245470978
Iteration 3, Best Length: 24.191626245470978
Iteration 4, Best Length: 24.191626245470978
Iteration 5, Best Length: 24.191626245470978
Iteration 6, Best Length: 24.191626245470978
Iteration 7, Best Length: 24.191626245470978
Iteration 8, Best Length: 24.191626245470978
Iteration 9, Best Length: 24.191626245470978
Iteration 10, Best Length: 24.191626245470978
Iteration 11, Best Length: 24.191626245470978
Iteration 12, Best Length: 24.191626245470978
Iteration 13, Best Length: 24.191626245470978
Iteration 14, Best Length: 24.191626245470978
Iteration 15, Best Length: 24.191626245470978
Iteration 16, Best Length: 24.191626245470978
Iteration 17, Best Length: 24.191626245470978
Iteration 18, Best Length: 24.191626245470978
Iteration 19, Best Length: 24.191626245470978
Iteration 20, Best Length: 24.191626245470978
Iteration 21, Best Length: 24.191626245470978
Iteration 22, Best Length: 24.191626245470978
Iteration 23, Best Length: 24.191626245470978
Iteration 24, Best Length: 24.191626245470978
Iteration 25, Best Length: 24.191626245470978
Iteration 26, Best Length: 24.191626245470978
Iteration 27, Best Length: 24.191626245470978
Iteration 28, Best Length: 24.191626245470978
Iteration 29, Best Length: 24.191626245470978
Iteration 30, Best Length: 24.191626245470978
Best Tour: [4, 3, 2, 0, 1]
Best Tour Length: 24.191626245470978
```



<u>Program 4:</u> Cuckoo Search (CS)

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

<u>Implementation Steps:</u>

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

(4) Cuckoo Search (CS):

Function CuckooSearch(Func, D, N, MaxIter, pa, alpha):

// Func: objective fxn (Σx_i^2) $f(x) = x_1^2 + x_2^2 + \dots + x_D^2$ (Sphere fn)

D: dimensionality of problem (no. of decision variables)

N: no. of nests (population size)

MaxIter: max no. of iterations

pa: prob of discovery (fraction of worst nests to replace)

alpha: step size for levy flight (scaling factor) //

nests = RandomlyInitialize(N, D) // N nests, each with D random values in [a,b]

fitness = EvaluateFitness(nests, Func)

best_nest = nests [IndexOfMinFitness (fitness)]

best_fitness = Min(fitness)

for iteration = 1 to MaxIter:

// generate new solutions (nests) using Levy flights

for each nest i in nests:

stepsize = alpha * LevyFlight (D)

new_nest = nests[i] + step_size

new_fitness = Func(new_nest)

if new_fitness < fitness[i]:

nests[i] = new_nest // replace old nest with new one

fitness[i] = new_fitness // update fitness of nest

// Abandon worst nests & replace them with new random nests

worst_nests = SelectWorstNests (fitness, pa)

ReplaceWithRandomNests (worstnests , D)

EvaluateFitness (worst_nests, Func)

//update best soln found so far
current_best_index = IndexOfMinFitness(fitness) //index of best nest
If fitness[current_best_index] < best_fitness:
    best_nest = nests[current_best_index] //update best nest
    best_fitness = fitness[current_best_index]

Return best_nest, best_fitness

function LevyFlight(D):
    return RandomNormal(0, 1, D) //D-dim step size

function RandomlyInitialize(N, D):
    return Random(N, D)

function EvaluateFitness(nests, Func):
    Return Apply(Func, nests)

Function SelectWorstNests(fitness, pa):
    Return SortByFitness(fitness)[-int(pa*N):] //select worst pa*N nests

    // Identify worst nests (highest fitness values)

Function ReplaceWithRandomNests(worst_nests, D):
    For each worst_nest in worst_nests
        worst_nest = Random(D) //Replace worst nest with new random solution

output:

[0.0751 0.0287 0.1106 0.0438 0.1728]

Code:

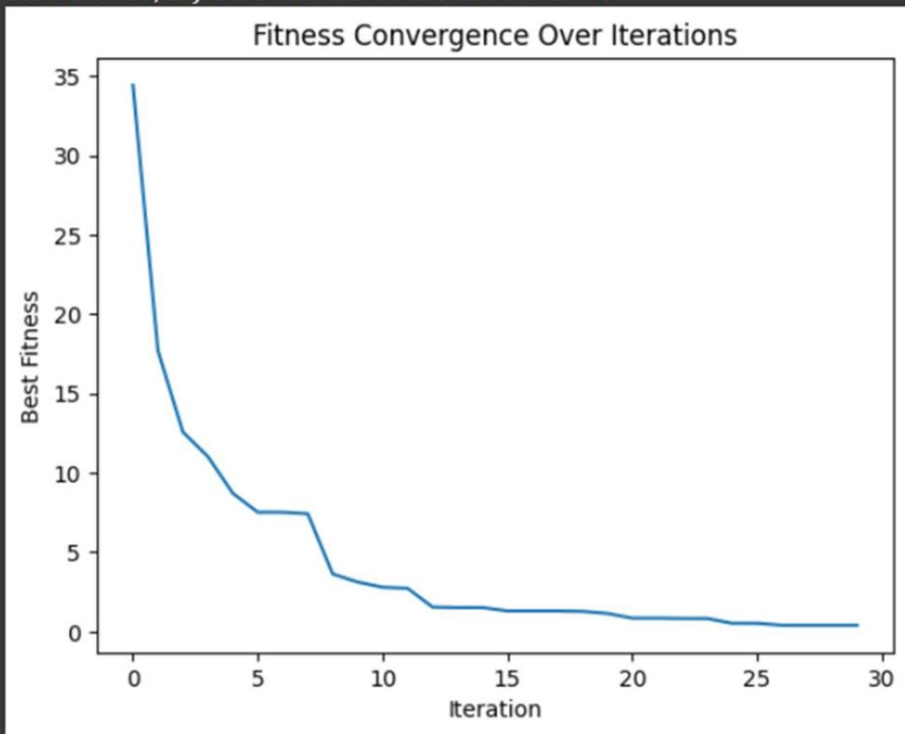
```
#cuckoo search
import numpy as np
import random
import math
import matplotlib.pyplot as plt

# 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()

        # Store fitness for plotting
        fitness_history.append(best_fitness)

        # Print the best solution at each iteration (optional)
        print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

    # Plot fitness convergence graph
    plt.plot(fitness_history)
    plt.title('Fitness Convergence Over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Best Fitness')
    plt.show()

    # Return the best solution found
    return best_nest, best_fitness

# Example usage
best_nest, best_fitness = cuckoo_search(num_nests=30, num_iterations=30, 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: 8.71378669296158
Iteration 6/30, Best Fitness: 7.5206125475077785
Iteration 7/30, Best Fitness: 7.5206125475077785
Iteration 8/30, Best Fitness: 7.42606230362850 2
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
```

```
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
```

<u>Program 5:</u> Grey Wolf Optimizer (GWO)

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

<u>Implementation Steps:</u>
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations

Algorithm:

Page No.
Date. / /

(5) Grey Wolf Optimizer (GWO)

Inputs:

objective fxn : f(x) to minimize/maximize

n: no. of wolves (population size)

max-iter: no of iterations

Bounds: lower & upper limits of search space

Outputs:

Best soln (Alpha wolf's position)

Fitness value (quality of solution)

(1) Initialize Alpha, Beta, Delta

alpha-posn = [0, 0, ... 0] (dimensional vector)

beta-posn = [0, 0, ... 0]

delta-posn = [0, 0, ... 0]

alpha-score = infinity

beta-score = infinity

delta-score = infinity

(2) Generate Initial population

→ Randomly initialize positions of n wolves within bounds

for each wolf in population:

generate random posn within bounds

(3) Evaluate fitness

→ calculate fitness of each wolf using obj fxn f(x)

for each wolf in population:

compute fitness = f(posn)

(Random coeffs (A, C) ensure balance between exploration & exploitation)

Page No.
Date. / /

else if fitness < delta-score :
  update delta

(5) Iterative optimization
  for t=1 to Max-iter:
    (1) update posns:
        each wolf posn is updated using influence of alpha, beta, and delta wolves
    (2) Boundary check:
        ensure updated posn remains within bounds
    (3) Re-evaluate fitness
        update alpha, beta, delta posns & scores based on new fitness values

(6) End iterations
  → After completing max-iter, return alpha wolf's position and fitness as best solution
    return alpha-posn (best soln) & alpha-score (fitness)

eg
  $f(x) = x^2$ in $[-10, 10]$
  random posns $(-7, 5, 2)$
  $\Rightarrow f(x) = 5^2 = 25$ (α)
     $f(x) = (-7)^2 = 49$ (β) & so on

  After many iterations, α converges to min soln
    (eg $x = 0 \rightarrow f(x) = 0$)

Applications -
  Engineering (eg design optimization)
  Data Analysis (feature selection)
  ML (hyperparameter tuning)

Code:

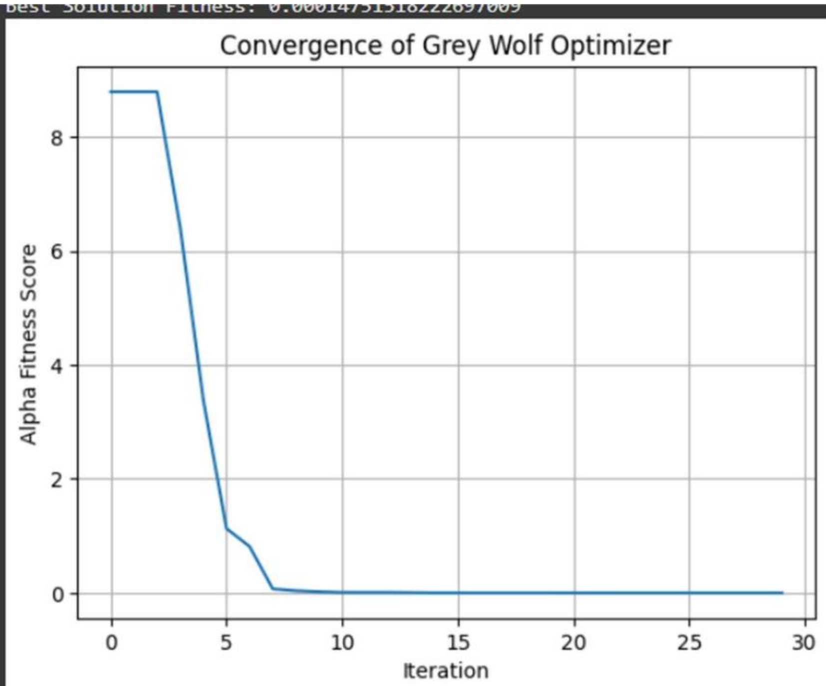
```
#GWO
import numpy as np
import matplotlib.pyplot as plt

# 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
```



Program 6: Parallel Cellular Algorithms and Programs

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Page No.
Date. / /

Output:

eg Alpha-posn (Best posn): [0.2, -0.1, 0.0, 0.3, -0.05]
Alpha-score (Best score): 0.155
$(\sum_{i=1}^{5} x_i^2)$ indicating algorithm found a near-optimal solution close to global minimum

---

(6) Parallel Cellular Algorithms & Program:

```
Function objective-function(x):
    return sum([xi^2 for xi in x])

Function get-neighbours(grid, i, j, grid-size):
    neighbors = []
    for di in [-1,0,1]:
        for dj in [-1,0,1]:
            if (di != 0 or dj != 0):
                ni = (i+di) mod grid-size
                nj = (j+dj) mod grid-size
                neighbors.append((ni,nj))
    return neighbors

function update-cell(currsol, neighborsol, perturbation):
    for each dim d in sol
        new sol = neighborsol + RANDOM(-perturbation range,
                                       + range)
return new-solution
```

Page No.
Date. / /

```
function parallel_cellular_Algorithms (grid-size, max-iter-sol, lb, ub):
    init grid with random sols in [lb, ub]
    init fitness values for all cells using obj-fxn
    set global-best-sol = none
    set global-best-fitness = infinity

    for t = 1 to max-iter:
        create an empty new-grid for updated soln
        for i ← 0 to grid-size - 1:
            for j ← 0 to grid-size - 1:
                neighbors = get-neighbors (grid, (i,j), grid-size)
                best neighbors = grid[i][j]
                best-fitness = fitness[i][j]
                for each (ni, nj) in neighbors:
                    if fitness[ni][nj] < best-fitness
                        best-fitness ← grid [ni][nj]
                        best-neighbor ← fitness [ni][nj]

                new sol = update-rule
                new fit = obj-fxn
        if newfit < fitness[i][j]
            new-grid[i][j] ← new-sol
            fitness[i][j] ← new-fit
    grid ← newgrid
        else new grid [i][j] ← grid[i][j]

        grid ← newgrid

        if fitness [i][j] < global-best-fitness.
            global-best-sol = grid[i][j]
            global-best-fitness = fitness[i][j]

    return global-best-sol, global-best-fitness
```

Code:

```
#pcap
import numpy as np

# Define the problem: A simple optimization function (e.g., Sphere Function)
def optimization_function(position):
    """Example: Sphere Function for minimization."""
    return sum(x**2 for x in position)

# Initialize Parameters
GRID_SIZE = (10, 10)  # Grid size (rows, columns)
NEIGHBORHOOD_RADIUS = 1  # Moore neighborhood radius
DIMENSIONS = 2  # Number of dimensions in the solution space
ITERATIONS = 30  # Number of iterations

# Initialize Population
def initialize_population(grid_size, dimensions):
    """Initialize a grid with random positions."""
    population = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1], dimensions))
    return population

# Evaluate Fitness
def evaluate_fitness(population):
    """Calculate the fitness of all cells."""
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

# 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:
                continue  # 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(population, fitness):
    """Update the state of each cell based on its neighbors."""
    new_population = np.copy(population)
```

```
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            neighbors = get_neighborhood(population, i, j, NEIGHBORHOOD_RADIUS)
            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]

            # Find the best position among neighbors
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness = fitness[ni, nj]
                    best_neighbor = population[ni, nj]

            # Update the cell state (move towards the best neighbor)
            new_population[i, j] = (population[i, j] + best_neighbor) / 2  # Average position
    return new_population

# Main Algorithm
def parallel_cellular_algorithm():
    """Implementation of the Parallel Cellular Algorithm."""
    population = initialize_population(GRID_SIZE, DIMENSIONS)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(ITERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness), fitness.shape)]

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

        # Print progress
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

    print("\nBest Solution Found:")
    print(f"Position: {best_solution}, Fitness: {best_fitness}")

# Run the algorithm
if _name___== "_main_":
    parallel_cellular_algorithm()
```

OUTPUT:

```
Iteration 1: Best Fitness = 0.43918427791098213
Iteration 2: Best Fitness = 0.43918427791098213
Iteration 3: Best Fitness = 0.06222127935032943б
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
```

<u>Program 7:</u> Optimization via Gene Expression Algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

<u>Implementation Steps:</u>

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Page No.
Date. / /

(7) Optimization via gene expression algorithms

population ← generate N random chromosomes
max_iter ← no. of iterations
mutation_rate ← rate of change in genetics
crossover_rate ← probability of crossover b/w parents
best_sol ← optimal gene
best_fitness ← fitness value of optimal gene

```
for generation in range (0, max_generations)
        phenotypes ← []
        for each chromosome in population:
                phenotype ← Decode genes of chromosome
                Append phenotype to phenotypes

        fitness score ← []
        for each phenotype in phenotypes:
                fitness ← Evaluate fitness of phenotype
                Append fitness to fitness score
                if fitness < best_fitness:
                        best_soln ← phenotype
                        best_fitness ← fitness
        parents ← select parents based on fitness score

        offspring ← []
        for i in range (0, length (parents), 2)
                if random() < crossover_rate:
                        child1, child2 ← perform crossover on
```

Page No.
Date. / /

(7) Optimization via gene expression algorithms

population ← generate N random chromosomes
max_iter ← no. of iterations
mutation_rate ← rate of change in genetics
crossover_rate ← probability of crossover b/w parents
best_sol ← optimal gene
best_fitness ← fitness value of optimal gene

for generation in range (0, max_generations)
    phenotypes ← [ ]
    for each chromosome in population:
        phenotype ← Decode genes of chromosome
        Append phenotype to phenotypes

    fitness score ← [ ]
    for each phenotype in phenotypes:
        fitness ← Evaluate fitness of phenotype
        Append fitness to fitness score
        if fitness < best_fitness:
            best_soln ← phenotype
            best_fitness ← fitness
    parents ← select parents based on fitness score

    offspring ← [ ]
    for i in range (0, length (parents), 2)
        if random() < crossover_rate:
            child1, child2 ← perform crossover on

Page No.
Date. / /

Append child1 and child2 to offspring

for each child in offspring:
    if random() < mutation_rate:
        mutate child

population ← offspring
print generation and best fitness

Code:

```
import numpy as np
import random

# 1. Define the Problem: Optimization Function (e.g., Sphere Function)
def optimization_function(solution):
    """Sphere Function for minimization (fitness evaluation)."""
    return sum(x**2 for x in solution)

# 2. Initialize Parameters
POPULATION_SIZE = 50  # Number of genetic sequences (solutions)
GENES = 5  # Number of genes per solution
MUTATION_RATE = 0.1  # Probability of mutation
CROSSOVER_RATE = 0.7  # Probability of crossover
GENERATIONS = 30  # Number of generations to evolve

# 3. Initialize Population
def initialize_population(pop_size, genes):
    """Generate initial population of random genetic sequences."""
    return np.random.uniform(-10, 10, (pop_size, genes))

# 4. Evaluate Fitness
def evaluate_fitness(population):
    """Evaluate the fitness of each genetic sequence."""
    fitness = [optimization_function(solution) for solution in population]
    return np.array(fitness)

# 5. Selection: Tournament Selection
def select_parents(population, fitness, num_parents):
    """Select parents using tournament selection."""
    parents = []
    for _ in range(num_parents):
        tournament = random.sample(range(len(population)), 3)  # Randomly select 3 candidates
        best = min(tournament, key=lambda idx: fitness[idx])
        parents.append(population[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. Mutation
def mutate(offspring, mutation_rate):
    """Apply mutation to introduce variability."""
    for i in range(len(offspring)):
        for j in range(len(offspring[i])):
            if random.random() < mutation_rate:
                offspring[i][j] += np.random.uniform(-1, 1)  # Random small change
    return offspring

# 8. Gene Expression: Functional Solution (No transformation needed for this case)
def gene_expression(population):
    """Translate genetic sequences into functional solutions."""
    return population  # Genetic sequences directly represent solutions here.

# 9. Main Function: Gene Expression Algorithm
def gene_expression_algorithm():
    """Implementation of Gene Expression Algorithm for optimization."""
    # Initialize population
    population = initialize_population(POPULATION_SIZE, GENES)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(GENERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        # Selection
        parents = select_parents(population, fitness, POPULATION_SIZE // 2)

        # Crossover
        offspring = crossover(parents, CROSSOVER_RATE)

        # Mutation
```

```
        offspring = mutate(offspring, MUTATION_RATE)

        # Gene Expression
        population = gene_expression(offspring)

        # Print progress
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    # Output the best solution
    print("\nBest Solution Found:")
    print(f"Position: {best_solution}, Fitness: {best_fitness}")

# 10. Run the Algorithm
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
```