# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB RECORD**

# Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**Shubham Maloo(1BM22CS343)**

***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 **Shubham Maloo(1BM22CS343),** 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.

| | |
|---|---|
| Sandhya A Kulkarni<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index



Github Link:
https://github.com/ShubhamMaloo00/1BM22CS343_BIS

**Program 1**
**Genetic Algorithm for Optimization Problems.**

Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

26/9/24 Lab - 1 (1)

Algorithm 1:

1.) Genetic Algorithm for optimization Problems

A Genetic Algorithm (GA) is a search heuristic function inspired by the process of natural selection and genetics. It is used to find approximate solutions to optimization and search problems. GAs mimic the process of biological evolution by using techniques such as selection crossover (recombination) and mutation.

Key Components.

1.) Population: A group of potential solutions to the problem.

2.) Chromosomes: Each individual solution in the population is often represented as a string (binary or real-valued).

3) Fitness function: A function that evaluates how good a solution is guiding the selection process.

4) Selection: The process of choosing the fittest individuals to reproduce and form the next generation.

5.) Crossover: A genetic operator used to combine the genetic information of two parents to generate new offspring.

6) Mutation : A genetic operator that introduces random changes to an individual's chromosomes to maintain genetic diversity.

Implementation steps.

1.) Define the problem: Create a mathematical function to optimize.

2.) Initialize parameters: Set the population size, mutation rate, crossover rate and number of generation.

3) Create Initial population: Generate an initial population of potential solution.

4.) Evaluate Fitness : Evaluate the fitness of each individual in the population.

5.) Selection : select individuals based on their fitness to reproduce.

6.) Fitness : To check they are fit or not.

7.) Crossover : Perform crossover between selected individuals to produce offspring.

8.) Mutation : Apply mutation to the offspring to maintain genetic diversity.

9.) Iteration : Repeat the valuation, selection crossover and mutation process for fixed number of generation

10.) Output the Best Solution : Track and output the best solution found during the generations.

3/10/24

# Lab - 2.

③

## Applications and Optimization Methods

1.) Engineering Design Optimization:

Application: structural optimization in civil engineering.

Optimization Method: GA's can optimize material distribution and geometrical configurations to maximize strength and minimize weight

2.) Scheduling Problems:

Applications: Job scheduling in manufacturing.

Optimization Method: GA's can optimize the sequence of jobs on machines to minimize makes pan or total completion time.

3.) Vechicle Routiong:

Application: Delivery route optimization. (eg. logistic companies).

Optimization Method: GAs help the most efficient route for delivery trucks, considering constraints like distance, time and veahicle capacity.

4.) Portfolio Optimization:

Application: Investment portfolio selection.

Optimization Method: GAs can maximize returns while minimizing risk by selecting optimal asset combinations based on historical data.

Code:

```
import numpy as np
import random

POPULATION_SIZE = 10
GENES = 8
GENERATIONS = 50
MUTATION_RATE = 0.1


def fitness_function(x):
    return x**2


def decode_chromosome(chromosome):
    return int("".join(map(str, chromosome)), 2)


def initialize_population():
    return [np.random.randint(0, 2, GENES).tolist() for _ in range(POPULATION_SIZE)]

def evaluate_fitness(population):
    return [fitness_function(decode_chromosome(individual)) for individual in population]


def select_parents(population, fitness):
    total_fitness = sum(fitness)
    probabilities = [f / total_fitness for f in fitness]
    selected = random.choices(population, weights=probabilities, k=2)
    return selected


def crossover(parent1, parent2):
    point = random.randint(1, GENES - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring2


def mutate(individual):
    for i in range(len(individual)):
        if random.random() < MUTATION_RATE:
            individual[i] = 1 - individual[i]  # Flip the bit
    return individual


def genetic_algorithm():
```

```
    # Step 1: Initialize population
    population = initialize_population()
    for generation in range(GENERATIONS):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Logging the best solution of the generation
        best_individual = population[np.argmax(fitness)]
        best_fitness = max(fitness)
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

        # Step 3: Create the next generation
        new_population = []
        while len(new_population) < POPULATION_SIZE:
            # Step 4: Select parents
            parent1, parent2 = select_parents(population, fitness)
            # Step 5: Crossover
            offspring1, offspring2 = crossover(parent1, parent2)
            # Step 6: Mutate
            new_population.append(mutate(offspring1))
            if len(new_population) < POPULATION_SIZE:
                new_population.append(mutate(offspring2))


        population = new_population


    fitness = evaluate_fitness(population)
    best_individual = population[np.argmax(fitness)]
    best_fitness = max(fitness)
    best_solution = decode_chromosome(best_individual)
    print("\nFinal Best Solution:")
    print(f"Chromosome: {best_individual}, Decoded: {best_solution}, Fitness: {best_fitness}")


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

**Program 2**
**Particle Swarm Optimization for Function Optimization.**

Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

24/10/24 Lab-3 (4)

1) Particle Swarm Optimization for Function
· Optimization

Particle Swarm optimization (PSO) is inspired by the social behaviour 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.

Algorithm:

1) Define the objective function:

- Purpose: Specify the function you want to optimize. This could be a minimization or maximization problem.
- Example: A common choice is the sphere function $f(x) = x_1^2 + x_2^2$ which has its global minimum at $(0,0)$.

2.) Initialize Parameters:

- Number of Particles: Decide how many candidate solutions (particles) to use.
- Number of Dimensions: Define how many parameters the optimization problem has (eg 2 for a 2D space).
- Inertia Weight (w): Controls the impact of the previous velocity on the current velocity. A higher value promotes exploration

- Cognitive coefficient ($c_1$): Governs the particles tendency to move towards its personal best position.
- Social coefficient ($c_2$): Governs the particle's tendency to move towards the global best position.
- Maximum Iteration: set a limit on how many times the algorithm will update positions and velocities.

3) Initialize Particles:

- Random Position Initialization: Generate random positions for each particles within the defined search space bounds (eg. between -10 and 10)

⇒ positions = np.random.uniform(-10, 10, (num_particles, num_dim))

- Random velocity Initialization: Assign random velocities to each particle, typically within a smaller range (eg. -1 to 1)

⇒ velocities = np.random.uniform(-1, 1, (num_particles, num_dim))

- Personal Bests: Initialize each particle's personal best position and value to its current position and the fitness value of that position

⇒ personal_best_positions = np.copy(positions)

4.) Determine Global Best:

- Find the Best: Evaluate all particles' personal best values and determine the global best position. This serves as a reference for all particles.

⇒ personal_best_values = np.array([objective_function(pos) for pos in positions])

⇒ global_best_position = personal_best_position[np.argmin(personal_best_values)]

⇒ global_best_value = min(personal_best_values).

6

5.) Iterate:

- Loop through Iterations: Repeat the following steps for a specified number of iteration (e.g. 100)

  - For Each Particle:

    - Update velocity:

      $$\Rightarrow v_i = w \cdot v_i + c_1 \cdot r_1 (p_i - x_i) + c_2 \cdot r_2 (g - x_i).$$

      - components:

        → $w \cdot v_i$ : The inertia component, carrying the previous velocity.

        → $c_1 \cdot r_1 (p_i - x_i)$: The cognitive componet, directing the particle towards ~~the global best~~ its personal best.

        → $c_2 r_2 (g - x_i)$: The social component, directing the particle towards the global best.

    - Update position:

      $$\Rightarrow x_i = x_i + v_i$$

    - Evaluate Fitness: compute the fitness of the new position using the objective function.

    - Update Personal Best: If the fitness of the new position is better than the particle's personal best, update the

• Update Global Best: (7)

- After all particles have update, check if any personal best is better than the current global best. If so update the global best.

Application:

1.) Machine Learning and Neural Network Training:

PSO helps in optimizing the parameters and weight of neural networks. Improving the performance of model in classification, regression and deep learning task without need for gradient Information.

2.) Image and signal processing:

PSO is employed in image segmentation, edge detection and other image processing tasks to enhance image quality or improving signal extraction in medical imaging, satellite imaging and more.

Output:

Best Position: [-3.096e-13, -3.4498e-13]

Best value: 2.14920... e-2

24/10

Code:
```
import random
import numpy as np

class Particle:
    def __init__(self, dim, bounds):
        self.position = np.array([random.uniform(bounds[0], bounds[1]) for _ in range(dim)])
        self.velocity = np.array([random.uniform(-1, 1) for _ in range(dim)])
        self.best_position = self.position.copy()
        self.best_score = float('inf')

    def update_velocity(self, global_best_position, w, c1, c2):
        inertia = w * self.velocity
        cognitive = c1 * random.random() * (self.best_position - self.position)
        social = c2 * random.random() * (global_best_position - self.position)
        self.velocity = inertia + cognitive + social

    def update_position(self, bounds):
        self.position = self.position + self.velocity
        self.position = np.clip(self.position, bounds[0], bounds[1])

    def evaluate(self, objective_function):
        score = objective_function(self.position)
        if score < self.best_score:
            self.best_score = score
            self.best_position = self.position.copy()


class PSO:
    def __init__(self, objective_function, dim, bounds, num_particles, max_iter, w=0.5, c1=1.5,
c2=1.5):
        self.objective_function = objective_function
        self.dim = dim
        self.bounds = bounds
        self.num_particles = num_particles
        self.max_iter = max_iter
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.global_best_position = None
        self.global_best_score = float('inf')
        self.particles = [Particle(dim, bounds) for _ in range(num_particles)]

    def optimize(self):
        for iteration in range(self.max_iter):
            for particle in self.particles:
                particle.evaluate(self.objective_function)
```

```
            if particle.best_score < self.global_best_score:
                self.global_best_score = particle.best_score
                self.global_best_position = particle.best_position.copy()

        for particle in self.particles:
            particle.update_velocity(self.global_best_position, self.w, self.c1, self.c2)
            particle.update_position(self.bounds)

        print(f"Iteration {iteration+1}/{self.max_iter}, Best Score: {self.global_best_score}")

    return self.global_best_position, self.global_best_score


def objective_function(x):
    return sum(xi**2 for xi in x)

dim = 2
bounds = (-10, 10)
num_particles = 30
max_iter = 100

pso = PSO(objective_function, dim, bounds, num_particles, max_iter)
best_position, best_score = pso.optimize()

print("Optimal Solution: ", best_position)
print("Best Score: ", best_score)
```

**Program 3**

**Ant Colony Optimization for the Traveling Salesman Problem**

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.

Algorithm:

7/11/24 Lab - 4 (8)

1) Ant Colony optimization for the Traveling Salesman Problem.

The foraging behaviour 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.

Pseudo Code for ACO

1.) Define the problem:

- Create a list of cities with their coordinates

$$cities = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$$

where $n$ is the number of cities and $x_i$ and $y_i$ is the coordinates.

- Calculate the distance Matrix ~~using~~ between all pairs of cities ~~and~~ using the Euclidean distance formula.

$$distance\,(i, j) = (x_i - x_j)^2 + (y_i - y_j)^2$$

2.) Initialize Parameters:

- Number of ants → nants.
- Number of iteration → niterations
- Pheromone importance factor → $\alpha$
- Heuristic importance factor → $\beta$
- Pheromone evaporation rate → $\rho$
- Initial pheromone value $\tau_0$ (typically set to 1.0)

9

- Exploration - exploitation balance q0, a threshold to decide wether to explore or exploit.

3.) Initialize Pheromone matrix:

Initialize the pheromone matrix T with the initial pheromone value T0: $T(i,j) = T0 \ \forall i,j$

Where $T(i,j)$ is the pheromone level on the edge between cities i and j.

4.) Main Loop (for each iteration):

for each iteration K, repeat the following steps.

① For Each Ant, construct a solution:

- start from a random city: Each ant starts at a randomly choosen city.

- select the next city:

$$P(i,j) = \frac{[T(i,j)]^{\alpha} \cdot [\text{distance}(i,j)]^{\beta}}{\sum_{K \in \text{allowed}} [T(i,K)]^{\alpha} \cdot [\text{distance}(i,K)]^{\beta}}$$

② Continue ~~untill~~ all city have been visited.

③ Evaluate the quality of the solution.

5.) Update Pheromone Trails:

① Evaporate pheromone on all edge.

Apply the evaporation rate ρ to all pheromone values: $T(i,j) \leftarrow (1-\rho) \cdot T(i,j)$ where ρ is the pheromone evaporation rate.

(2) Deposit new pheromone based on the quality of the tour. 10

For each ant's tour, deposit pheromone inversely proportional to the tour length.

$$T(i,j) \leftarrow T(i,j) + L_{tour} Q_3 .$$

6.) Keep the track of the Best solution: shortest tour and its corresponding length found by the ants during the current itr

- If an ant finds a shorter tour, update the best solution.

7.) Repeat steps 4-6 for a set Number of iterations or Until Convergence.

Repeat the main loop for niterations iterations or untill the algorithm converges.

8) Output the Best Solution.

After the algorithm finishes running, output the best solution found.

- Best Tour: The sequence of cities visited that corresponds to the shortest path.
- Best distance: The total length of the shortest path found.

Ou
Be
Le
Ap
1)
2)
3)
4)
5)
Pro

## Output:

(11)

Best tour found: [2, 4, 5, 3, 1, 0]

length of best tour: 18.861186377582317

## Application:

1) Travelling Salesman Problem.
2) Vechicle Routing problem
3) Job shop scheduling problem.
4) Network routing.
5) Robotics and Path Planning.

8/12
7/11

Probability, $P(i,j) = \dfrac{[T_{ij}]^{\alpha}\,[\eta_{ij}]^{\beta}}{\sum_{k \in \text{allowed}} [T_{ik}]^{\alpha} \cdot [\eta_{jk}]^{\beta}}$

$T_{ij}$ is the pheromone level on edge$(i,j)$

$\eta_{ij}$ is the heuristic information (usually 1/distance$_{ij}$)

$\alpha$, $\beta$ control the relative importance of pheromone vs heuristic information.

Code:

```
import random
import numpy as np

class AntColony:
    def __init__(self, dist_matrix, num_ants, num_iterations, alpha=1, beta=2, rho=0.1, q=100):
        self.dist_matrix = dist_matrix
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q = q
        self.num_cities = len(dist_matrix)
        self.pheromone_matrix = np.ones((self.num_cities, self.num_cities))
        np.fill_diagonal(self.pheromone_matrix, 0)

    def select_next_city(self, current_city, visited_cities):
        probabilities = []
        for city in range(self.num_cities):
            if city not in visited_cities:
                pheromone = self.pheromone_matrix[current_city][city] ** self.alpha
                distance = self.dist_matrix[current_city][city] ** self.beta
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)

        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabilities]

        return np.random.choice(range(self.num_cities), p=probabilities)

    def construct_solution(self):
        visited_cities = [random.randint(0, self.num_cities - 1)]
        while len(visited_cities) < self.num_cities:
            current_city = visited_cities[-1]
            next_city = self.select_next_city(current_city, visited_cities)
            visited_cities.append(next_city)
        visited_cities.append(visited_cities[0])
        return visited_cities

    def calculate_solution_length(self, solution):
        length = 0
        for i in range(len(solution) - 1):
            length += self.dist_matrix[solution[i]][solution[i + 1]]
```

```
        return length

    def update_pheromones(self, solutions, lengths):
        self.pheromone_matrix *= (1 - self.rho)
        for solution, length in zip(solutions, lengths):
            for i in range(len(solution) - 1):
                self.pheromone_matrix[solution[i]][solution[i + 1]] += self.q / length

    def optimize(self):
        best_solution = None
        best_length = float('inf')
        for _ in range(self.num_iterations):
            solutions = []
            lengths = []
            for _ in range(self.num_ants):
                solution = self.construct_solution()
                length = self.calculate_solution_length(solution)
                solutions.append(solution)
                lengths.append(length)
                if length < best_length:
                    best_solution = solution
                    best_length = length

            self.update_pheromones(solutions, lengths)

            print(f"Best Length in Iteration: {best_length}")

        return best_solution, best_length

def generate_distance_matrix(num_cities):
    matrix = np.random.randint(10, 100, size=(num_cities, num_cities))
    np.fill_diagonal(matrix, 0)
    return matrix


num_cities = 10
num_ants = 20
num_iterations = 100


dist_matrix = generate_distance_matrix(num_cities)


aco = AntColony(dist_matrix, num_ants, num_iterations)
best_solution, best_length = aco.optimize()

print("Best Solution (City Order):", best_solution)
```

print("Best Solution Length:", best_length)

**Program 4**

**Cuckoo Search (CS)**

Algorithm:

14/11/24 Lab-5 12

1:) Cuckoo Search (CS):

Cuckoo search is a nature inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Levy flight to generate new solutions, promoting global search capabilities and avoiding local minima.

Pseud code:

Objective: Minimize or Maximize a given objective function $f(x)$ where $x \in R^d$.

1:) Define the optimization problem:
Define the objective function $f(x)$ to optimize where $x = (x_1, x_2, \ldots x_d)$ is a vector in the search space.

2:) Set Algorithm Parameters:
num-nests: Number of nests (solutions) in the population.
pa: Discovery probability.
Max-iter: Maximum number of iteration.
Bounds: Define bounds ~~and~~ lb and ub for each dimension.

3:) Initialize Population:
Initialize a population of nests $X = \{x_i \mid i = 1, 2, \ldots num\text{-}nests\}$
With random positions within the bounds of search space

$$x_i = lb + (ub - lb) \cdot rand() \quad (13)$$

where $x_i$ is a position vector for nest $i$ and rand() generates a random vector within [0, 1]

4.) for each iteration (until max-iter):

(a) Evaluate a fitness of each nest:

Compute the fitness $f(x_i)$ for each nest $x_i$ based on the objective function.

(B) Generate New Solutions using Levy flights for each nest $x_i$, generate a new solution $x_{new}$ ~~which~~ using levy flights.

$$x_{new} = x_i + Levy(\lambda) \cdot (x_i - x_j)$$

where,

$x_j$ is randomly selecting nest.

$\alpha$ is a scaling factor.

Levy ($\lambda$) denotes a Levy flight step typically sampled from a distribution with finite variance

$$Levy(\lambda) \sim \frac{u}{v^{1/\lambda}}$$

where $u$ and $v$ are normally distributed random numbers and $\lambda$ is often set to 1.5.

(c) $x_i = x_{new}$ if $f(x_{new}) < f(x_i)$

14

(d) Abandon a fraction of worst Nests.

$$x_i = lb + (ub - lb) \cdot rand() \text{ for a fraction } pa \text{ of nests.}$$

(e) Track the best solution.

$$x_{best} = \arg\min_{x_i} f(x_i)$$

5.) Return the Best solution: $f(x_{best})$

Output:

Best Solution: [0.06265177, 0.02252894]

Best fitness: 0.0094327971102910004

Application:

1) Solving Travelling Salesman Problem
2) Image Processing
3) Routing optimization
4) Game strategy optimization
5) Knapsack problem
6) Job scheduling.

14/11

Code:

```
import numpy as np
import random
from scipy.special import gamma

def energy_function(x, y, theta):
    A = 1.5
    S = 1000
    optimal_distance = 0.0
    distance = np.sqrt(x**2 + y**2)
    energy = (A * S) / (1 + distance) * np.cos(np.radians(theta))
    return max(energy, 0)

class CuckooSearch:
    def __init__(self, fitness_function, lower_bound, upper_bound, population_size=25,
max_iter=100, pa=0.25):
        self.fitness_function = fitness_function
        self.lower_bound = np.array(lower_bound)
        self.upper_bound = np.array(upper_bound)
        self.population_size = population_size
        self.max_iter = max_iter
        self.pa = pa
        self.n_dim = len(lower_bound)
        self.population = np.random.uniform(self.lower_bound, self.upper_bound, (self.population_size,
self.n_dim))
        self.fitness = np.zeros(self.population_size)
        self.best_solution = None
        self.best_fitness = float('-inf')

    def levy_flight(self, x):
        beta = 1.5
        sigma = (gamma(1 + beta) * np.sin(np.pi * beta / 2) / (gamma((1 + beta) / 2) * beta * 2**((beta -
1) / 2)))**(1 / beta)
        u = np.random.normal(0, sigma, size=self.n_dim)
        v = np.random.normal(0, 1, size=self.n_dim)
        step = u / np.abs(v)**(1 / beta)
        new_solution = x + step * 0.01
        return np.clip(new_solution, self.lower_bound, self.upper_bound)

    def cuckoo_search(self):
        for iteration in range(self.max_iter):
            for i in range(self.population_size):
                new_solution = self.levy_flight(self.population[i])
                new_fitness = self.fitness_function(*new_solution)
                if new_fitness > self.fitness[i]:
                    self.population[i] = new_solution
```

```
                self.fitness[i] = new_fitness
            if new_fitness > self.best_fitness:
                self.best_solution = new_solution
                self.best_fitness = new_fitness

        for i in range(self.population_size):
            if random.random() < self.pa:
                self.population[i] = np.random.uniform(self.lower_bound, self.upper_bound, self.n_dim)
                self.fitness[i] = self.fitness_function(*self.population[i])

        print(f"Iteration {iteration+1}/{self.max_iter}, Best Fitness (Energy): {self.best_fitness}")

    return self.best_solution, self.best_fitness

lower_bound = [-10, -10, 0]
upper_bound = [10, 10, 90]
population_size = 25
max_iter = 100
pa = 0.25

cuckoo_search = CuckooSearch(energy_function, lower_bound, upper_bound, population_size,
max_iter, pa)
best_solution, best_fitness = cuckoo_search.cuckoo_search()

print("Optimal Solar Panel Position and Tilt Angle:")
print(f"Position (x, y): {best_solution[:2]}, Tilt Angle (theta): {best_solution[2]} degrees")
print(f"Optimal Energy (Fitness): {best_fitness} Watts")
```

**Program 5**

**Grey Wolf Optimizer (GWO):**

Algorithm:

21/11/24 Lab-6 (15)

1) Grey Wolf Optimizer Algorithm.

GWO algorithm is search intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves. It mimics leadership structure of alpha, beta, delta and omega wolves and their collaborative hunting strategies.

Pseudo code:

1.) Define Rastrigin function
$f(x) = A * \text{len}(x) + \text{sum}(x_i^2 - A\cos(2\pi x_i))$

2.) Initialise GWO parameters
N: no of wolves
dim: dimensionality of problem
max_iter: Maximum no of iterations
lb, ub: Bounds for search space.

3.) Initialize positions of wolves randomly
positions = random values between lb and ub for N wo[lves]

4.) Initialize α, β, δ wolves positions and fitne[ss] Scores.
alpha_pos = [0] * dim
beta_pos = [0] * dim
delta_pos = [0] * dim
alpha_score = +∞
beta_score = +∞
delta_score = +∞

(16)

5.) For each iteration ( t = 1 to max-iter)

a. for each wolf (i=1 to N)

evaluate fitness = f ( position[i])

update alpha, beta, delta wolves based on fitness

if (fitness < alpha-score)

update alpha-pos, alpha-score

else if (fitness < beta-score)

update beta-pos, beta-score

else if (fitness < delta-score)

update delta-pos, delta score

b. Update position of wolves

for each wolf i :

calculate coefficients A, C, using random values r1, r2.

update position [i] = position [i] - A* (alpha - position(i))

c. Decrease linearly over iteration

a = 2 - t ( 2/max-iter)

d. Print best fitness score at each iteration

6.) After max-iter :

output best position ( alpha - pos)

output best fitness (alpha - score)

7.) Visualization: (17)

Plot contours of Rastrigin function
plot best position found by GWO algorithm

Output:

Iteration 1/100, Best fitness : 3.70808050

Iteration 2/100, Best fitness : 3.70808050

⋮

Iteration 100/100, Best fitness : 1.77763568

Best Position : [ 8.30503 e-09 , 4.1557 e-09 ]

Best fitness : (Rastrigin value) : 1.77635.

Applications :

1.) Engineering Design Optimization:

Optimizes structural ~~structure~~ designs and mechanical components for better performance and reduced cost.

2.) Power Systems:

solved problems like optimal power flow load dispatcher and electrical machine design improve efficiency and minimize cost.

3.) Image Processing:

Applied in image segmentation and enhancement by optimizing thresholds ~~is~~ and improving visual quality.

Code:

```
import numpy as np

def objective_function(x):
    return np.sum(x**2)

def grey_wolf_optimizer(obj_func, dim, bounds, max_iter, pack_size):
    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)
    alpha_score = float("inf")
    beta_score = float("inf")
    delta_score = float("inf")
    wolves = np.random.uniform(bounds[0], bounds[1], (pack_size, dim))

    for t in range(max_iter):
        for i in range(pack_size):
            fitness = obj_func(wolves[i])
            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 = wolves[i].copy()
            elif fitness < beta_score:
                delta_score = beta_score
                delta_pos = beta_pos.copy()
                beta_score = fitness
                beta_pos = wolves[i].copy()
            elif fitness < delta_score:
                delta_score = fitness
                delta_pos = wolves[i].copy()

        a = 2 - t * (2 / max_iter)
        for i in range(pack_size):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                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

                r1, r2 = np.random.rand(), np.random.rand()
                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

            r1, r2 = np.random.rand(), np.random.rand()
            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

            wolves[i, j] = (X1 + X2 + X3) / 3
        wolves[i] = np.clip(wolves[i], bounds[0], bounds[1])

    return alpha_pos, alpha_score

dim = 5
bounds = (-10, 10)
max_iter = 100
pack_size = 20

best_position, best_score = grey_wolf_optimizer(objective_function, dim, bounds, max_iter,
pack_size)

print("Best Position:", best_position)
print("Best Score:", best_score)
```

**Program 6**

**Parallel Cellular Algorithms and Programs:**

Algorithm:

19/12/24

Lab - 7

(18)

1) 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 algorithm leverage the principles of cellular automata and parallel computing to solve ~~complex~~ complex optimization problem efficiently.

Pseudocode:

1.) Define Problem: Fitness function.

```
function fitness(solution):
    return fitness_value.
```

2.) Initialize: Grid size, iterations, population.

```
function initialize_population():
    return (random_solution() for each cell in
            grid_size).
```

3.) Update: Parallel state update for each cell

```
function update_cell(cell, neighbors):
    best = min(neighbours, key = fitness)
    return apply_update_rule(cell, best)
```

4.) Run Algorithm:

(9)

```
function run ()
    population = initialize_population ()
    for iteration in 1 to max_iteration:
        parallel-for each cell in population:
            neighbors = get neighbors (cell)
            cell.solution = update_cell (cell, neighbors)
            cell.fitness = fitness (cell.solution)

        best = min (population, key = fitness)
        if (best.fitness < convergence_threshold
                break;

        return best
```

5.) Main:

```
best_solution = run ()
output (best_solution).
```

Output:

Best solution found: Position: 0.011921942

Fitness: 0.0014213299

Application :

1.) Image processing : Parallel pixel updates for tasks like segmentation and edge detection.

2.) Wireless Sensor Network : Optimizes sensor placement and energy efficiency.

3.) Vechicle Routing : Optimizes delivery routes for minimal travel time.

4.) Genetic Research.

5.) Resource Management.

19/12

Code:

```
import numpy as np

def rastrigin_function(x, y):
    return 10 * 2 + (x**2 - 10 * np.cos(2 * np.pi * x)) + (y**2 - 10 * np.cos(2 * np.pi * y))

def initialize_grid(grid_size, bounds):
    return np.random.uniform(bounds[0], bounds[1], (grid_size, grid_size, 2))

def evaluate_grid(grid, fitness_function):
    fitness_grid = np.zeros((grid.shape[0], grid.shape[1]))
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            x, y = grid[i, j]
            fitness_grid[i, j] = fitness_function(x, y)
    return fitness_grid

def get_neighbors(grid, i, j):
    neighbors = [
        grid[(i - 1) % grid.shape[0], j],
        grid[(i + 1) % grid.shape[0], j],
        grid[i, (j - 1) % grid.shape[1]],
        grid[i, (j + 1) % grid.shape[1]],
    ]
    return neighbors

def update_grid(grid, fitness_grid, bounds):
    new_grid = grid.copy()
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            neighbors = get_neighbors(grid, i, j)
            best_neighbor = min(neighbors, key=lambda n: rastrigin_function(n[0], n[1]))
            if rastrigin_function(best_neighbor[0], best_neighbor[1]) < fitness_grid[i, j]:
                new_grid[i, j] = best_neighbor + np.random.uniform(-0.1, 0.1, size=2)
                new_grid[i, j] = np.clip(new_grid[i, j], bounds[0], bounds[1])
    return new_grid

def parallel_cellular_algorithm(fitness_function, grid_size, bounds, max_iter):
    grid = initialize_grid(grid_size, bounds)
    for _ in range(max_iter):
        fitness_grid = evaluate_grid(grid, fitness_function)
        grid = update_grid(grid, fitness_grid, bounds)
    best_cell = min(grid.reshape(-1, 2), key=lambda c: fitness_function(c[0], c[1]))
    best_fitness = fitness_function(best_cell[0], best_cell[1])
    return best_cell, best_fitness
```

```
grid_size = 10
bounds = (-5.12, 5.12)
max_iter = 100

best_solution, best_fitness = parallel_cellular_algorithm(rastrigin_function, grid_size, bounds,
max_iter)

print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

**Program 7**

**Optimization via Gene Expression Algorithms:**

Algorithm:

19/12/24 Lab - 8 21

1) Optimization via Gene Expression Algorithms:

Gene expression Algorithm (GEA) are inspired by the biological process of gene expression in living organism. This process involve the translation of genetic information encoded in DNA into functional proteins, In GEA, solution to optimization problems are encoded in a manner similar to genetic sequences.

Pseudo code:

```
import random

# fitness function
def fitness(solution):
        return fitness_value

# Initialize Population
def initialize_population(pop-size):
        return [random_solution() for _ in range
                                        (pop_size)]

# cross over between parents.
def cross over (p1, p2, state):
    return (compine(p1, p2), combine(p2, p1))
        if random.random() < rate
        else (p1, p2)
```

```
# Mutation of offspring
def mutate (offspring, rate):
    return [ mutate_gene (g) if random. random()
             < rate  else g  for g in offspring].

# selection
def selection (population):
    return roulette_wheel_selection (population,
           [ fitness (ind)  for ind in population]

# Run Algorithm
def run (gen_count , pop-size, cross rate, mut_rate,
         threshold ):

    pop = initialize_population (pop-size).

    for _ in range (gen_count):
        parents = selection (pop)
        pop = []

    for p1, p2  in  parents :
        offspring1, offspring 2  = crossover
                   ( p1, p2 , cross rate)

        pop. extend ( [ mutate ( offspring1,
                      mut_rate),
               mutate [ offspring 2, mut_rate)]
        best = min ( pop , key = fitness)
        if  fitness (best) <  threshold:
            break
    return   best .
```

# Main

best_solution = run (100, 50, 0.8, 0.05, 0.01) 23
Output ( first - solution ).

Output :

Best feature set : [0, 0, 1, 1]
Best fitness (Accuracy) : 10

Application:

1) Machine learning Model Training
2) portfolio optimization in finance
3.) feature , engineering in data science
4.) ~~feature engineering~~ Water distribution system Optimization
5.) Engineering desing optimization,

Code:

```
import numpy as np

def sphere_function(x):
    return np.sum(x**2)

def initialize_population(pop_size, gene_length, bounds):
    return np.random.uniform(bounds[0], bounds[1], (pop_size, gene_length))

def evaluate_fitness(population, fitness_function):
    return np.array([fitness_function(individual) for individual in population])

def select_parents(population, fitness, num_parents):
    sorted_indices = np.argsort(fitness)
    return population[sorted_indices[:num_parents]]

def crossover(parents, offspring_size):
    offspring = np.zeros(offspring_size)
    for i in range(offspring_size[0]):
        p1, p2 = np.random.choice(parents.shape[0], 2, replace=False)
        crossover_point = np.random.randint(1, offspring_size[1])
        offspring[i, :crossover_point] = parents[p1, :crossover_point]
        offspring[i, crossover_point:] = parents[p2, crossover_point:]
    return offspring

def mutate(offspring, mutation_rate, bounds):
    for i in range(offspring.shape[0]):
        for j in range(offspring.shape[1]):
            if np.random.rand() < mutation_rate:
                offspring[i, j] += np.random.uniform(-0.1, 0.1)
                offspring[i, j] = np.clip(offspring[i, j], bounds[0], bounds[1])
    return offspring

def gene_expression_algorithm(fitness_function, pop_size, gene_length, bounds, num_generations,
mutation_rate, num_parents):
    population = initialize_population(pop_size, gene_length, bounds)
    best_solution, best_fitness = None, float("inf")
    for _ in range(num_generations):
        fitness = evaluate_fitness(population, fitness_function)
        best_idx = np.argmin(fitness)
        if fitness[best_idx] < best_fitness:
            best_solution, best_fitness = population[best_idx], fitness[best_idx]
        parents = select_parents(population, fitness, num_parents)
        offspring_size = (pop_size - num_parents, gene_length)
        offspring = crossover(parents, offspring_size)
```

```
        offspring = mutate(offspring, mutation_rate, bounds)
        population[:num_parents] = parents
        population[num_parents:] = offspring
    return best_solution, best_fitness

pop_size = 50
gene_length = 5
bounds = (-5.12, 5.12)
num_generations = 100
mutation_rate = 0.1
num_parents = 10

best_solution, best_fitness = gene_expression_algorithm(sphere_function, pop_size, gene_length,
bounds, num_generations, mutation_rate, num_parents)

print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```