VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sindhuja Narasimhan (1BM22CS279)

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 **Sindhuja Narasimhan (1BM22CS279),** 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.

Saritha A.N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/sindhuja279/BISlab.git

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

classmate

Date

Page 3

03/10/24

Lab 2.

Algorithm for genetic algorithm

Step 1. Initialize parameters: set the population size, mutation rate, crossover rate & no of generations.

Step 2. Generate Initial population: create a random population of potential solutions within the given bounds.

Step 3. Evaluate Fitness: calculate the fitness of each individual in the population by evaluating the objective function.

Step 4: Select Parents: select the fittest individuals from the population to reproduce based on their fitness

Step 5: Crossover perform crossover between the selected parents to create new offspring, with a probability equal to the crossover rate

Step 6. Mutate: apply mutation to the offspring, with a probability equal to mutation rate, to introduce new traits

Step 7. Replace Least Fit: replace the least fit individuals in the population with the new offspring

Step 8. Repeat: repeat steps 3-7 for a fixed number of generations or until convergence criteria are met.

Step 9. Output Best soln: return the best soln found during the generations, which is the individual with the

classmate
Date
Page 4

→ The basic idea behind a genetic algorithm is to simulate the process of evolution by iteratively selecting, breeding and mutating a population of candidate solution.

→ The genetic algorithm is used for
- Optimizing problems such as scheduling, resource allo and portfolio optimization
- machine learning to optimize parameters of models such as neural networks and decision trees.
- engineering design: to optimize complex systems such as electronic circuits etc.
- computer network: to optimize network routing, schedu and resource allocation.

→ Optimizing Techniques:
- selection: selecting the fittest individuals from the population to reproduce.
- crossover: combining the genetic information of two parents to create a new offspring
- mutation: randomly changing the genetic information of an individual to introduce new traits.
- elitism: preserving the best solution from the previous generation to ensure that the best solution are not lost.
- Simulated binary crossover: A crossover technique that simulates the process of binary crossover to create new offspring

/10/24

2

Code:

```
import random
import numpy as np

def objective_function(x):
    # Define the objective function to be minimized
    return x**2 + 2*x + 1

def generate_initial_population(population_size, bounds):
    # Generate an initial population of potential solutions
    population = []
    for _ in range(population_size):
        x = random.uniform(bounds[0], bounds[1])
        population.append(x)
    return population

def evaluate_fitness(population):
    # Evaluate the fitness of each individual in the population
    fitness = []
    for x in population:
        fitness.append(objective_function(x))
    return fitness

def selection(population, fitness, num_parents):
    # Select individuals based on their fitness to reproduce
    parents = []
    for _ in range(num_parents):
        max_fitness_idx = np.argmax(fitness)
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -float('inf')  # Set fitness to -inf to avoid re-selection
    return parents

def crossover(parents, crossover_rate):
    # Perform crossover between selected individuals to produce offspring
    offspring = []
    for _ in range(len(parents) // 2):
        parent1, parent2 = random.sample(parents, 2)
        if random.random() < crossover_rate:
            x1, x2 = parent1, parent2
            x_offspring = (x1 + x2) / 2
            offspring.append(x_offspring)
    return offspring

def mutation(offspring, mutation_rate, bounds):
    # Apply mutation to the offspring to maintain genetic diversity
    for i in range(len(offspring)):
```

```
        if random.random() < mutation_rate:
            offspring[i] += random.uniform(-0.1, 0.1) * (bounds[1] - bounds[0])
            offspring[i] = max(bounds[0], min(offspring[i], bounds[1]))  # Ensure bounds
    return offspring

def genetic_algorithm(population_size, mutation_rate, crossover_rate, num_generations, bounds):
    population = generate_initial_population(population_size, bounds)
    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        parents = selection(population, fitness, population_size // 2)
        offspring = crossover(parents, crossover_rate)
        offspring = mutation(offspring, mutation_rate, bounds)
        population = offspring + parents
    best_solution = min(population, key=objective_function)
    return best_solution

# Set parameters
population_size = 100
mutation_rate = 0.1
crossover_rate = 0.5
num_generations = 100
bounds = (-10, 10)

# Run the Genetic Algorithm
best_solution = genetic_algorithm(population_size, mutation_rate, crossover_rate, num_generations,
bounds)
print("Best solution:", best_solution)
```

Program 2

Particle Swarm Optimization for Function Optimization

Algorithm:

Date: 07/11/24 Page: 8

Lab : Particle Swarm Optimization.

→ PSO is an optimization algorithm inspired by the social behaviour of animals such as bird flocking or fish schooling. The goal is to find optimal or near-optimal solutions to complex problems by having multiple "particles" explore the solution space. Each particle adjusts its position based on its own experience and the experience of its neighbours, ultimately converging towards the best solution

→ ~~Algorithm~~ Code Implementation

```
import numpy as np

def rastrigin_function(x):
    A = 10
    return A * len(x) + sum(x_i ** 2 - A * np.cos(2 * np.pi * x_i)
                            for x_i in x)

class Particle:
    def __init__(self, dim):
        self.position = np.random.uniform(-5.12, 5.12, dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin_function(self.position)

class PSO:
    def __init__(self, num_particles, dim, inertia_weight, cognitive_coeff,
                                         social_coeff, max_iter):
        self.num_particles = num_particles
        self.dim = dim
        self.inertia_weight = inertia_weight
```

Page: 9

```
        self.cognitive_coeff = cognitive_coeff
        self.social_coeff = social_coeff
        self.max_iter = max_iter
        self.swarm = [Particle(dim) for _ in range(num_particles)]
        self.global_best_position = self.swarm[0].best_position
        self.global_best_value = self.swarm[0].best_value

    def optimize(self):
        for iteration in range(self.max_iter):
            for particle in self.swarm:
                fitness_value = rastrigin_function(particle.position)
                if fitness_value < particle.best_value:
                    particle.best_value = fitness_value
                    particle.best_position = np.copy(particle.position)

                if fitness_value < self.global_best_value:
                    self.global_best_value = fitness_value
                    self.global_best_position = np.copy(particle.best_
                                                              position)

            for particle in self.swarm:
                r1 = np.random.rand(self.dim)
                r2 = np.random.rand(self.dim)

                particle.velocity = (self.inertia_weight * particle.velocity +
                        self.cognitive_coeff * r1 * (particle.best_position -
                                        particle.position) +
                        self.social_coeff * r2 * (self.global_best_pos))

                particle.position += particle.velocity
        return self.global_best_position, self.global_best_value.
```

Code:

```
import numpy as np

# Define the Rastrigin function
def rastrigin_function(x):
    A = 10
    return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

# Particle Swarm Optimization Algorithm
class Particle:
    def __init__(self, dim):
        self.position = np.random.uniform(-5.12, 5.12, dim)  # Random position
        self.velocity = np.random.uniform(-1, 1, dim)  # Random velocity
        self.best_position = np.copy(self.position)  # Personal best position
        self.best_value = rastrigin_function(self.position)  # Personal best value

class PSO:
    def __init__(self, num_particles, dim, inertia_weight, cognitive_coeff, social_coeff, max_iter):
        self.num_particles = num_particles
        self.dim = dim
        self.inertia_weight = inertia_weight
        self.cognitive_coeff = cognitive_coeff
        self.social_coeff = social_coeff
        self.max_iter = max_iter
        self.swarm = [Particle(dim) for _ in range(num_particles)]
        self.global_best_position = self.swarm[0].best_position
        self.global_best_value = self.swarm[0].best_value

    def optimize(self):
        for iteration in range(self.max_iter):
            for particle in self.swarm:
                # Evaluate fitness
                fitness_value = rastrigin_function(particle.position)

                # Update personal best
                if fitness_value < particle.best_value:
                    particle.best_value = fitness_value
                    particle.best_position = np.copy(particle.position)

                # Update global best
                if fitness_value < self.global_best_value:
                    self.global_best_value = fitness_value
                    self.global_best_position = np.copy(particle.best_position)

            # Update velocities and positions
            for particle in self.swarm:
                r1 = np.random.rand(self.dim)
```

```python
            r2 = np.random.rand(self.dim)

            # Update velocity
            particle.velocity = (self.inertia_weight * particle.velocity +
                                 self.cognitive_coeff * r1 * (particle.best_position - particle.position) +
                                 self.social_coeff * r2 * (self.global_best_position - particle.position))

            # Update position
            particle.position += particle.velocity

        return self.global_best_position, self.global_best_value

# Parameters
num_particles = 30  # Number of particles
dim = 2  # Dimensionality of the problem
inertia_weight = 0.7  # Inertia weight
cognitive_coeff = 1.5  # Cognitive coefficient
social_coeff = 1.5  # Social coefficient
max_iter = 100  # Maximum number of iterations

# Run PSO
pso = PSO(num_particles, dim, inertia_weight, cognitive_coeff, social_coeff, max_iter)
best_position, best_value = pso.optimize()

print("Best Position:", best_position)
print("Best Value:", best_value)
```

Program 3
Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:

14/11/24

Page 11

Lab:5 Ant Colony optimization

Ant colony optimization provides a biologically inspired approach to solving the Travelling Salesman problem by mimicking the natural behaviour of ants. By using pheromone trails and probabilistic path selection, ACO can effectively explore the solution space and converge towards optimal or near-optimal solutions for TSP.

```
import numpy as np
import random

class AntColony:
    def _init_(self, cities, num_ants, alpha, beta, rho,
                                                iterations):
        self.cities = cities
        self.num_cities = len(cities)
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.best_route = None
        self.best_distance = float('inf')

    def calculate_distance_matrix(self):
        distance_matrix = np.zeros((self.num_cities, self.num_cities))
        for i in range(self.num_cities):
            for j in range(self.num_cities):
                distance_matrix[i][j] = np.linalg.norm(np.array(
                    self.cities[i] - np.array(self.cities[j]))
        return distance_matrix.
```

classmate
Date
Page 12

```
def choose_next_city(self, current_city, visited):
    probabilities = []
    for city in range (self.num_cities):
        if city not in visited:
            pheromone = self.pheromone [current_city][city] ** self.alpha
            heuristic = (1/self.distance_matrix[current_city][city]) ** self.beta
            probabilities.append (pheromone * heuristic)
        else:
            probabilities.append(0)
    probabilities = np.array(probabilities)
    probabilities /= probabilities.sum()
    return np.random.choice(range(self.num_cities), p=
                            probabilities)

def construct_solution (self):
    for i in range(self.num_ants):
        visited = [0]
        for _ in range(1, self.num_cities):
            next_city = self.choose_next_city (visited [-1], visited)
            visited.append (next_city)
        visited.append(0)
        distance = self.calculate_route_distance (visited)
        if distance < self.best_distance:
            self.best_distance = distance
            self.best_route = visited

def calculate_route_distance(self, route):
    return sum(self.distance_matrix[route[i]][route[i+1]]
               for i in range(len(route)-1))

def update_pheromones(self):
    self.pheromone *= (1 - self.rho)
    for city in range(len(self.best_route)-1):
        self.pheromone[self.best_route[city]][self.best_route[city+1]] += 1
                                          / self.best_distance
```

classmate
Date
Page 13

```
def optimize (self):
    for _ in range(self.iterations):
        self.construct_solution()
        self.update_pheromones()
    return self.best_route, self.best_distance

if _name_ == "main":
    cities = [(0, 0), (1, 2), (2, 4), (3, 1), (4, 3)]
    num_ants = 10
    alpha = 1.0
    beta = 2.0
    rho = 0.5
    iterations = 10
    aco = AntColony(cities, num_ants, alpha, beta, rho, iterations)
    best_route, best_distance = aco.optimize()
    print ("Best Route:", best_route)
    print ("Best Distance", best_distance)
```

Output:

Best Route: [0, 1, 2, 4, 3, 0]

Best Distance: 12.10654940

13.11

Code:

```
import numpy as np
import random

class AntColony:
    def __init__(self, cities, num_ants, alpha, beta, rho, iterations):
        self.cities = cities
        self.num_cities = len(cities)
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.iterations = iterations
        self.pheromone = np.ones((self.num_cities, self.num_cities))  # Initial pheromone levels
        self.distance_matrix = self.calculate_distance_matrix()
        self.best_route = None
        self.best_distance = float('inf')

    def calculate_distance_matrix(self):
        distance_matrix = np.zeros((self.num_cities, self.num_cities))
        for i in range(self.num_cities):
            for j in range(self.num_cities):
                distance_matrix[i][j] = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
        return distance_matrix

    def choose_next_city(self, current_city, visited):
        probabilities = []
        for city in range(self.num_cities):
            if city not in visited:
                pheromone = self.pheromone[current_city][city] ** self.alpha
                heuristic = (1 / self.distance_matrix[current_city][city]) ** self.beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)
        probabilities = np.array(probabilities)
        probabilities /= probabilities.sum()  # Normalize
        return np.random.choice(range(self.num_cities), p=probabilities)
```

```
    def construct_solution(self):
        for _ in range(self.num_ants):
            visited = [0]  # Start from the first city
            for _ in range(1, self.num_cities):
                next_city = self.choose_next_city(visited[-1], visited)
                visited.append(next_city)
            visited.append(0)  # Return to the starting city
            distance = self.calculate_route_distance(visited)
            if distance < self.best_distance:
                self.best_distance = distance
                self.best_route = visited

    def calculate_route_distance(self, route):
        return sum(self.distance_matrix[route[i]][route[i + 1]] for i in range(len(route) - 1))

    def update_pheromones(self):
        # Evaporate pheromones
        self.pheromone *= (1 - self.rho)
        # Add pheromones based on the best route found
        for city in range(len(self.best_route) - 1):
            self.pheromone[self.best_route[city]][self.best_route[city + 1]] += 1 / self.best_distance

    def optimize(self):
        for _ in range(self.iterations):
            self.construct_solution()
            self.update_pheromones()
        return self.best_route, self.best_distance

# Example usage:
if __name__ == "__main__":
    # Define cities as (x, y) coordinates
    cities = [(0, 0), (1, 2), (2, 4), (3, 1), (4, 3)]

    # ACO parameters
    num_ants = 10
    alpha = 1.0  # importance of pheromone
    beta = 2.0   # importance of heuristic
```

```
rho = 0.5 # pheromone evaporation rate
iterations = 10

aco = AntColony(cities, num_ants, alpha, beta, rho, iterations)
best_route, best_distance = aco.optimize()

print("Best Route:", best_route)
print("Best Distance:", best_distance)
```

Program 4
Cuckoo Search (CS)

Algorithm:

14

21-11-24

Lab 6: Cuckoo Search.

Cuckoo search is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nest 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.

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

def levy_flight(beta = 1.5, d = 1):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi *
            beta/2) / gamma((1+beta)/2) * beta * np.cos(np.
            pi * beta/2) ** 2), 1/beta)
    u = np.random.normal(0, sigma_u, size = d)
    v = np.random.normal(0, 1, size = d)
    step = u / np.power(np.abs(v), 1/beta)
    return step.

def initialize_population(n_nests, n_dim, lower_bound,
                                          upper_bound):
    return np.random.uniform(lower_bound, upper_bound,
                                     (n_nests, n_dim))

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

def cuckoo_search(n_nests, n_dim, lower_bound, upper
                     bound, max_iter, pa=0.25):
    nest = initialize_population(n_nest, n_dim, lower_bound
                                         upper_bound)
```

15

```
    fitness = np.array([fitness_function(nest) for nest in
                                                  nests])
    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    for iteration in range(max_iter):
        for i in range(n_nests):
            step = levy_flight(d = n_dim)
            new_nest = nests[i] + step * (nests[i] - best_nest)
            new_nest = np.clip(new_nest, lower_bound,
                                   upper_bound)
            new_fitness = fitness_function(new_nest)
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness.
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness.

        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = np.random.uniform(lower_bound,
                                   upper_bound, n_dim)
                fitness[i] = fitness_functions(nests[i])

        if (iteration+1) % 100 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration+1}, Best Fitness: {best_fitness}
    return best_nest, best_fitness.

n_nests = 25
n_dim = 10
lower_bound = -5
upper_bound = -5.
```

Code:

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

# Levy flight function
def levy_flight(beta=1.5, d=1):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi * beta / 2) / gamma((1 + beta) / 2) * beta *
                        np.cos(np.pi * beta / 2) ** 2), 1 / beta)
    u = np.random.normal(0, sigma_u, size=d)
    v = np.random.normal(0, 1, size=d)
    step = u / np.power(np.abs(v), 1 / beta)
    return step

# Initialize population (nests)
def initialize_population(n_nests, n_dim, lower_bound, upper_bound):
    return np.random.uniform(lower_bound, upper_bound, (n_nests, n_dim))

# Fitness function (Example: Sphere function)
def fitness_function(x):
    return np.sum(x ** 2)

# Cuckoo Search Algorithm
def cuckoo_search(n_nests, n_dim, lower_bound, upper_bound, max_iter, pa=0.25):
    # Step 1: Initialize nests randomly
    nests = initialize_population(n_nests, n_dim, lower_bound, upper_bound)

    # Step 2: Evaluate fitness of all nests
    fitness = np.array([fitness_function(nest) for nest in nests])

    # Track the best solution found so far
    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    # Start iterations
    for iteration in range(max_iter):
        # Generate new solutions using Levy flight
        for i in range(n_nests):
            # Generate a new solution by Levy flight
            step = levy_flight(d=n_dim)
            new_nest = nests[i] + step * (nests[i] - best_nest)

            # Apply boundary conditions
            new_nest = np.clip(new_nest, lower_bound, upper_bound)

            # Evaluate the new solution
```

```
            new_fitness = fitness_function(new_nest)

            # If the new solution is better, replace the old nest
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

                # Update the best solution if necessary
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness

        # Abandon some of the worst nests and generate new random solutions
        for i in range(n_nests):
            if random.random() < pa:  # with probability pa
                nests[i] = np.random.uniform(lower_bound, upper_bound, n_dim)
                fitness[i] = fitness_function(nests[i])

        # Print progress every 100 iterations
        if (iteration + 1) % 100 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration + 1}, Best Fitness: {best_fitness}")

    return best_nest, best_fitness

# Parameters
n_nests = 25  # Number of nests
n_dim = 10  # Dimensionality of the problem
lower_bound = -5  # Lower bound for the search space
upper_bound = 5  # Upper bound for the search space
max_iter = 1000  # Maximum number of iterations
pa = 0.25  # Probability of abandoning the worst nests

# Run Cuckoo Search
best_solution, best_solution_fitness = cuckoo_search(n_nests, n_dim, lower_bound, upper_bound,
max_iter, pa)

print("\nBest solution found: ", best_solution)
print("Best fitness value: ", best_solution_fitness)
```

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

Algorithm:

Date
Page 17

28/11/24

Lab 7: Grey Wolf Optimizer

The grey wolf optimizer is a swarm intelligence algo inspired by the social hierarchy and hunting behaviour of grey wolves. It mimics the leadership structure of alpha, beta, & omega wolves and their collaborative hunting strategies

```
import numpy as np

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

def grey_wolf_optimizer(obj_func, dim, lb, ub, wolves_count, iterations):
    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_positions = np.random.uniform(lb, ub, (wolves_count, dim))

    for iteration in range(iterations):
        for i in range(wolves_count):
            fitness = obj_func(wolves_positions[i])
            if fitness < alpha_score:
                alpha_score, alpha_pos = fitness, wolves_positions[i].copy()
            elif fitness < beta_score:
                beta_score, beta_pos = fitness, wolves_positions[i].copy()
```

Page 18

```
elif fitness < delta_score:
    delta_score, delta_pos = fitness, wolves_positions[i].copy()

for i in range(wolves_count):
    for j in range(dim):
        a = 2 - 2 * (iteration / iterations)
        r1, r2 = np.random.rand(), np.random.rand()

        A1 = 2 * a * r1 - a
        C1 = 2 * r2

        D_alpha = abs(C1 * alpha_pos[j] - wolves_positions[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_positions[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_positions[i][j])
        X3 = delta_pos[j] - A3 * D_delta

        wolves_positions[i][j] = (X1 + X2 + X3) / 3

    wolves_positions[i] = np.clip(wolves_positions[i], lb, ub)

return alpha_pos, alpha_score
```

Page 19.

```
dimension = 5
lower_bound = -10
upper_bound = 10
wolves = 30
max_iteration = 50

best_position, best_score = grey_wolf_optimizer(
        objective_function, dimension, lower_bound,
        upper_bound, wolves, max_iteration)

print("Best position:", best_position)
print("Best score:", best_score)
```

Output

Best position: [6.39416683 6.40050065 -6.59530705
7.98219731 -7.16874410e-05]

Best Score: 2.40256295 34.

08.11.

Code:

```
import numpy as np

# Define the objective function to be minimized
def objective_function(x):
    return np.sum(x**2)  # Example: Sphere function (minimization problem)

# Grey Wolf Optimizer (GWO) implementation
def grey_wolf_optimizer(obj_func, dim, lb, ub, wolves_count, iterations):
    # Initialize alpha, beta, and delta wolves' positions
    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)

    # Initialize alpha, beta, and delta wolves' fitness values
    alpha_score = float("inf")  # Best fitness
    beta_score = float("inf")   # Second-best fitness
    delta_score = float("inf")  # Third-best fitness

    # Initialize the positions of all wolves
    wolves_positions = np.random.uniform(lb, ub, (wolves_count, dim))

    # Main loop for optimization
    for iteration in range(iterations):
        for i in range(wolves_count):
            # Calculate the fitness of the current wolf
            fitness = obj_func(wolves_positions[i])

            # Update alpha, beta, and delta wolves
            if fitness < alpha_score:
                alpha_score, alpha_pos = fitness, wolves_positions[i].copy()
            elif fitness < beta_score:
                beta_score, beta_pos = fitness, wolves_positions[i].copy()
            elif fitness < delta_score:
                delta_score, delta_pos = fitness, wolves_positions[i].copy()

        # Update the positions of wolves
        for i in range(wolves_count):
            for j in range(dim):
                # Coefficients
                a = 2 - 2 * (iteration / iterations)  # Linearly decreases from 2 to 0
                r1, r2 = np.random.rand(), np.random.rand()

                A1 = 2 * a * r1 - a
                C1 = 2 * r2
```

```
            D_alpha = abs(C1 * alpha_pos[j] - wolves_positions[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_positions[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_positions[i][j])
            X3 = delta_pos[j] - A3 * D_delta

            # Update position
            wolves_positions[i][j] = (X1 + X2 + X3) / 3

        # Enforce bounds
        wolves_positions[i] = np.clip(wolves_positions[i], lb, ub)


    # Return the best solution
    return alpha_pos, alpha_score

# Parameters
dimension = 5  # Number of variables
lower_bound = -10  # Lower bound of variables
upper_bound = 10   # Upper bound of variables
wolves = 30        # Number of wolves in the pack
max_iterations = 50  # Maximum number of iterations

# Run the GWO algorithm
best_position, best_score = grey_wolf_optimizer(
    objective_function, dimension, lower_bound, upper_bound, wolves, max_iterations
)

print("Best position:", best_position)
print("Best score:", best_score)
```

Program 6

Parallel Cellular Algorithms and Programs

Algorithm:

Page 20

19/12/24

Lab 8: Parallel Cellular Algorithm.

Parallel cellular algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner.

```
import numpy as np.
def rastrigin_function(x):
    A=10
    return A * len(x) + sum((xi ** 2 - A * np.cos(2 * np.pi * xi))
                            for xi in x)

class ParallelCellularAlgorithm:
    def_init_(self, grid_size=(5,5), dim=2, neighborhood=
                    'moore', iterations=100,):

        self.grid_size = grid_size
        self.dim = dim
        self.neighborhood = neighborhood
        self.bounds = bounds
        self.grid = np.random.uniform(bounds[0], bounds[1],
                                      grid_size[1], dim))
        self.fitness = np.zeros((grid_size[0], grid_size[1]))

def evaluate_fitness(self):
    for i in range(self.grid_size[0]):
        for j in range(self.grid_size[1]):
            self.fitness[i,j] = rastrigin_func(self.grid[i,j])

def get_neighbors(self, i, j):
    neighbors = []
    directions = []
    if self.neighborhood == 'moore':
        directions = [(-1,-1), (-1, 0), (-1,1)]
    elif self.neighborhood == 'von_neuman':
        directions = [(-1,0), (0,-1) (0,1)].
```

Page 21

```
update_states(self):
new_grid = self.grid.copy()
for i in range(self.grid_size[0]):
    for j in range(self.grid_size[1]):
        neighbors = self.get_neighbors(i,j)
self.grid = new_grid.

run(self):
for iteration in range(self.iterations):
    self.evaluate_fitness()
    self.update_states()
    print(f"Iteration {iteration+1}/{self.iterations}")

    print("Optimization complete")
    print("Best Solution: {self.best_solution}")
    print(f"Best Fitness: {self.best_fitness:.6f}")

if _name_ == "_main_":
    pca = ParallelCellularAlgorithm(grid_size=(10,10),
                dim=2, neighborhood='moore')
    pca.run()
```

Output.

Optimization Complete

Best Solution: [-0.0337945   -0.0340271]

Best Fitness: 0.451717

Code:

```
import numpy as np

def rastrigin_function(x):
    """
    Rastrigin function for optimization. It has a global minimum at x = 0.
    f(x) = 10n + sum(x_i^2 - 10 * cos(2 * pi * x_i))
    """
    A = 10
    return A * len(x) + sum((xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in x)

class ParallelCellularAlgorithm:
    def __init__(self, grid_size=(5, 5), dim=2, neighborhood='moore', iterations=100, bounds=(-5.12, 5.12)):
        """
        Initialize the parameters for the cellular algorithm.

        :param grid_size: Tuple defining the dimensions of the grid (rows, columns)
        :param dim: Dimensionality of the solution space
        :param neighborhood: Type of neighborhood ('moore' or 'von_neumann')
        :param iterations: Number of iterations to run
        :param bounds: Lower and upper bounds for the solution space
        """
        self.grid_size = grid_size
        self.dim = dim
        self.neighborhood = neighborhood
        self.iterations = iterations
        self.bounds = bounds

        # Initialize the grid with random solutions
        self.grid = np.random.uniform(bounds[0], bounds[1], size=(grid_size[0], grid_size[1], dim))
        self.fitness = np.zeros((grid_size[0], grid_size[1]))

        # Store the best solution and its fitness
        self.best_solution = None
        self.best_fitness = float('inf')

    def evaluate_fitness(self):
        """
        Evaluate fitness of each cell in the grid.
        """
        for i in range(self.grid_size[0]):
            for j in range(self.grid_size[1]):
                self.fitness[i, j] = rastrigin_function(self.grid[i, j])

                # Update the best solution
                if self.fitness[i, j] < self.best_fitness:
```

```
            self.best_fitness = self.fitness[i, j]
            self.best_solution = self.grid[i, j].copy()

    def get_neighbors(self, i, j):
        """
        Get neighboring cells based on the chosen neighborhood type.

        :param i: Row index of the current cell
        :param j: Column index of the current cell
        :return: List of neighboring cells
        """
        neighbors = []
        directions = []
        if self.neighborhood == 'moore':
            # Moore neighborhood: 8 neighbors (including diagonals)
            directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
        elif self.neighborhood == 'von_neumann':
            # Von Neumann neighborhood: 4 neighbors (up, down, left, right)
            directions = [(-1, 0), (0, -1), (0, 1), (1, 0)]

        for di, dj in directions:
            ni, nj = i + di, j + dj
            if 0 <= ni < self.grid_size[0] and 0 <= nj < self.grid_size[1]:
                neighbors.append(self.grid[ni, nj])
        return neighbors

    def update_states(self):
        """
        Update the state of each cell based on its neighbors.
        """
        new_grid = self.grid.copy()
        for i in range(self.grid_size[0]):
            for j in range(self.grid_size[1]):
                neighbors = self.get_neighbors(i, j)

                # Calculate the average position of neighbors
                avg_neighbor = np.mean(neighbors, axis=0)

                # Move the current cell slightly towards the average neighbor position
                new_grid[i, j] = self.grid[i, j] + 0.1 * (avg_neighbor - self.grid[i, j])

                # Ensure the new solution stays within bounds
                new_grid[i, j] = np.clip(new_grid[i, j], self.bounds[0], self.bounds[1])

        self.grid = new_grid

    def run(self):
        """
```

```
        Run the Parallel Cellular Algorithm.
        """
        for iteration in range(self.iterations):
            # Step 1: Evaluate fitness of the current grid
            self.evaluate_fitness()

            # Step 2: Update the states of all cells
            self.update_states()

            # Print progress
            print(f"Iteration {iteration + 1}/{self.iterations}, Best Fitness: {self.best_fitness:.6f}")

        print("Optimization complete.")
        print(f"Best Solution: {self.best_solution}")
        print(f"Best Fitness: {self.best_fitness:.6f}")

if __name__ == "__main__":
    # Initialize and run the algorithm
    pca = ParallelCellularAlgorithm(grid_size=(10, 10), dim=2, neighborhood='moore', iterations=50)
    pca.run()
```

Program 7

Optimization via Gene Expression Algorithms

Algorithm:

Date
Page 22

19/11/24

Lab 9: Optimization via Gene Expression Algorithm.

Gene Expression Algo are inspired by the biological process of gene expression in living organisms. The process involves the translation of genetic information encoded in DNA into functional proteins.

```
import numpy as np
import random
def rastrigin_function(x):
    A = 10
    return A * len(x) + sum((xi ** 2 - A * np.cos(2 * np.pi
                              * xi)) for xi in x)

class GeneExpressionAlgorithm:
    def __init__(self, pop_size = 100, gene_l = 10, bounds (-5.12, 5.12)):
    self.pop_size = pop_size
    self.gene_length = gene_length
    self.mutation_rate = mutation_rate
    self.bounds = bounds.

def random_gene_sequence(self):
    return np.random.uniform(self.bounds[0], self.bounds[1],
                      size = self.gene_length)

def evaluate_fitness(self, gene_sequence):
    return rastrigin_function(gene_sequence)

def selection(self):
    tournament_size = 3
    selected = []
    for _ in range(self.pop_size):
        tournament = random.sample(self.population, tournament_
                                                     size)
    return selected.
```

classmate
Date
Page 23.

```
def mutate(self, gene_sequence):
    for i in range(len(gene_sequence)):
        if random.random() < self.mutation_rate:
            gene_sequence[i] += np.random.uniform(-1.0, 1.0)
            gene_sequence[i] = np.clip(gene_sequence[i],
    return gene_sequence.                     self.bounds[i])

def gene_expression(self, gene_sequence):
    return gene_sequence.

def run(self):
    for generation in range(self.generations):
        fitness_values = [self.evaluate_fitness(ind) for ind
                                 in self.population]
        best_idx = np.argmin(fitness_value)
        selected_pop = self.selection()
        next_population = []
    for i in range(0, self.pop_size, 2):
        parent1 = select_pop[i]
        parent 2 = select_pop[i+1]
        offspring1, offspring2 = self.crossover(p1, p2)
        next_pop.append(self.mutate(offspring1,))
    self.population = next_population

    print("Optimization complete")
    print(f"Best Sol, {self.best_sol}")
    print(f"Best Fitness: {self.best_fitness:.6f}")

ga.run().
```

O/P Optimization Complete
Best Sol : [-0.02796834 -0.01908597]
Best Fitness: 0.226972.

Code:
```
import numpy as np
import random

def rastrigin_function(x):
    """
    Rastrigin function for optimization. It has a global minimum at x = 0.
    f(x) = 10n + sum(x_i^2 - 10 * cos(2 * pi * x_i))
    """
    A = 10
    return A * len(x) + sum((xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in x)

class GeneExpressionAlgorithm:
    def __init__(self, population_size=100, gene_length=10, mutation_rate=0.01, crossover_rate=0.7, generations=100, bounds=(-5.12, 5.12)):
        """
        Initialize the parameters for the Gene Expression Algorithm.

        :param population_size: Number of genetic sequences in the population
        :param gene_length: Length of each genetic sequence
        :param mutation_rate: Probability of mutation
        :param crossover_rate: Probability of crossover
        :param generations: Number of generations to evolve
        :param bounds: Lower and upper bounds for the solution space
        """
        self.population_size = population_size
        self.gene_length = gene_length
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.bounds = bounds

        # Initialize population with random solutions
        self.population = [self.random_gene_sequence() for _ in range(population_size)]
        self.best_solution = None
        self.best_fitness = float('inf')

    def random_gene_sequence(self):
        """Generate a random gene sequence within bounds."""
        return np.random.uniform(self.bounds[0], self.bounds[1], size=self.gene_length)

    def evaluate_fitness(self, gene_sequence):
        """Evaluate the fitness of a gene sequence using the Rastrigin function."""
        return rastrigin_function(gene_sequence)

    def selection(self):
        """Select individuals for reproduction using tournament selection."""
        tournament_size = 3
        selected = []
        for _ in range(self.population_size):
            tournament = random.sample(self.population, tournament_size)
            tournament_fitness = [self.evaluate_fitness(ind) for ind in tournament]
```

```
            winner = tournament[np.argmin(tournament_fitness)]
            selected.append(winner)
        return selected

    def crossover(self, parent1, parent2):
        """Perform crossover between two parents to produce offspring."""
        if random.random() < self.crossover_rate:
            point = random.randint(1, self.gene_length - 1)
            offspring1 = np.concatenate((parent1[:point], parent2[point:]))
            offspring2 = np.concatenate((parent2[:point], parent1[point:]))
            return offspring1, offspring2
        return parent1, parent2

    def mutate(self, gene_sequence):
        """Apply mutation to a gene sequence."""
        for i in range(len(gene_sequence)):
            if random.random() < self.mutation_rate:
                gene_sequence[i] += np.random.uniform(-1.0, 1.0)
                gene_sequence[i] = np.clip(gene_sequence[i], self.bounds[0], self.bounds[1])
        return gene_sequence

    def gene_expression(self, gene_sequence):
        """Translate a genetic sequence into a functional solution (no changes needed here)."""
        return gene_sequence

    def run(self):
        """Run the Gene Expression Algorithm."""
        for generation in range(self.generations):
            # Step 1: Evaluate fitness and find the best solution
            fitness_values = [self.evaluate_fitness(ind) for ind in self.population]
            best_idx = np.argmin(fitness_values)
            if fitness_values[best_idx] < self.best_fitness:
                self.best_fitness = fitness_values[best_idx]
                self.best_solution = self.population[best_idx].copy()

            # Step 2: Selection
            selected_population = self.selection()

            # Step 3: Crossover and Mutation
            next_population = []
            for i in range(0, self.population_size, 2):
                parent1 = selected_population[i]
                parent2 = selected_population[i + 1]
                offspring1, offspring2 = self.crossover(parent1, parent2)
                next_population.append(self.mutate(offspring1))
                next_population.append(self.mutate(offspring2))

            self.population = next_population
```

```
        print("Optimization complete.")
        print(f"Best Solution: {self.best_solution}")
        print(f"Best Fitness: {self.best_fitness:.6f}")

if __name__ == "__main__":
    # Initialize and run the Gene Expression Algorithm
    gea = GeneExpressionAlgorithm(population_size=50, gene_length=2, generations=50)
    gea.run()
```