VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sakshi Shetty(1BM22CS234)

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 **Sakshi Shetty (1BM22CS234),** 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

Sl. No.	Date	Experiment Title	Page No.
1		Genetic Algorithm for Optimization Problems	1-3
2		Particle Swarm Optimization for Function Optimization	3-6
3		Ant Colony Optimization for the Traveling Salesman Problem	6-9
4		Cuckoo Search Optimization	9-12
5		Grey Wolf Optimizer	12-15
6		Parallel Cellular Algorithms and Programs	15-19
7		Optimization via Gene Expression Algorithms	19-22

Github Link: https://github.com/Sakshishetty24/BIS

Program 1
Problem statement

Genetic Algorithm for Optimization 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.

Algorithm:

DOMS Page No. Date / /

Generic Algorithm for Optimization

1. Initialize parameters
Set population size, mutation rate, crossover rate, no. of generations and value range.

2. Create Initial population
Generate a random population of candidate solutions within the defined range.

3. Evaluate fitness:
Calculate the fitness of each individual using the objective function $f(x) = x^2$.

4. Track best solution:
Keep track of the best solution found so far.

5. Main loop (for each generation)

- Selection:
  select two parents from the population, using roulette - wheel selection based on fitness
- Crossover:
  Combine the parents to produce the offspring using a crossover method.
- Mutation:
  Apply random mutations to the offspring to introduce diversity.
- Update Population:
  Replace the old population with the new offspring.

6. Termination Condition:
Repeat the process for a fixed no. of generations.

```
FUNCTION GeneticAlgorithm ():
    SET population_size , num_generations , mutation_rate , crossover_r
        value_range.
    population = InitializePopulation (population_size, value_range)
    best_solution = None
    best_fitness = -∞

    FOR generation IN 1 TO num_generations:
        fitness = EvaluateFitness (population)
        current_best_index = ARGMAX (fitness)

        IF fitness [current_best_index] > best_fitness:
            best_fitness = fitness [current_best_index]
            best_solution = population [current_best_index]

        new_population = []
        FOR i IN 1 TO population_size:
            parent1 , parent2 = Selection (population , fitness)
            offspring = Crossover (parent1, parent2)
            offspring = Mutate (offspring , mutation_rate , value_rang
            APPEND offspring TO new_population

        population = new_population

    RETURN best_solution , best_fitness.
```

7. Output:
Return the best solution and its corresponding fitness value.

Code:

```
import numpy as np

def objective_function(x):
    return x ** 2

population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
value_range = (-10, 10)

def initialize_population(size, value_range):
    return np.random.uniform(value_range[0], value_range[1], size)

def evaluate_fitness(population):
    return np.array([objective_function(x) for x in population])

def selection(population, fitness):
    probabilities = fitness / fitness.sum()
    return population[np.random.choice(len(population), size=2, p=probabilities)]

def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        return (parent1 + parent2) / 2  # Simple averaging crossover
    return parent1 if np.random.rand() < 0.5 else parent2

def mutate(individual, mutation_rate, value_range):
    if np.random.rand() < mutation_rate:
        return np.random.uniform(value_range[0], value_range[1])
    return individual

def genetic_algorithm():
    population = initialize_population(population_size, value_range)
    best_solution = None
    best_fitness = -np.inf

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)

        current_best_index = np.argmax(fitness)
        if fitness[current_best_index] > best_fitness:
            best_fitness = fitness[current_best_index]
            best_solution = population[current_best_index]

        new_population = []
        for _ in range(population_size):
            parent1, parent2 = selection(population, fitness)
            offspring = crossover(parent1, parent2)
```

```python
            offspring = mutate(offspring, mutation_rate, value_range)
            new_population.append(offspring)

        population = np.array(new_population)

    return best_solution, best_fitness

best_solution, best_fitness = genetic_algorithm()

print(f"Best solution found: x = {best_solution:.2f}")
print(f"Maximum value of f(x) = x^2: f(x) = {best_fitness:.2f}")
```

```
Best solution found: x = 10.00
Maximum value of f(x) = x^2: f(x) = 99.95
```

Program 2
Problem statement

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.

Algorithm:

Date / /

Particle Swarm Optimization

1. Define objective_function (position):
return position [0]^2 + position [1]^2

2. Define Particle class:
Initialize particle:
    position = random numbers in [bounds[0], bounds[1]]
    velocity = random in [-1, 1]
    best_position = position
    best_score = objective_function (position)

Update velocity (global_best_position):
    velocity = inertia_weight * velocity + cognitive_component + social_component
    cognitive_component = cognitive_coef * r1 * (best_position - position)
    social_component = social_coef * r2 (global_best_position - position)

Update position ():
    position += velocity
    enforce bounds

Evaluate ():
    score = objective_function (position)
    if score < best_score:
        best_position = position
        best_score = score

3. Initialize PSO:
    swarm = list of num_particles particles.
    global_best_position = best_position of first particle.
    global_best_score = objective_function (global_best_position)

4. For each iteration (1 to max_iterations):
For each particle in swarm:
Update velocity (global_best_position)
Update position()
Evaluate()
If particle's best_score < global_best_score:
global_best_position = particle's best_position
global_best_score = particle's best_score;

5. Return global_best_position, global_best_score.

Code:
```
import numpy as np

# Define the objective function to be optimized (e.g., f(x, y) = x^2 + y^2)
def objective_function(position):
    return position[0] ** 2 + position[1] ** 2

# Particle class to represent each particle in the swarm
class Particle:
    def __init__(self, bounds):
        # Initialize position and velocity randomly within bounds
        self.position = np.array([np.random.uniform(bound[0], bound[1]) for bound in bounds])
        self.velocity = np.array([np.random.uniform(-1, 1) for _ in bounds])
        self.best_position = self.position.copy()
        self.best_score = objective_function(self.position)

    def update_velocity(self, global_best_position, inertia_weight, cognitive_coef, social_coef):
        # Generate random factors for stochastic update
        r1, r2 = np.random.rand(2)

        # Update velocity based on inertia, cognitive, and social components
        cognitive_component = cognitive_coef * r1 * (self.best_position - self.position)
        social_component = social_coef * r2 * (global_best_position - self.position)

        self.velocity = inertia_weight * self.velocity + cognitive_component + social_component

    def update_position(self, bounds):
        # Update position based on velocity
        self.position += self.velocity
        # Enforce boundary conditions
        for i in range(len(bounds)):
            if self.position[i] < bounds[i][0]:
                self.position[i] = bounds[i][0]
            elif self.position[i] > bounds[i][1]:
                self.position[i] = bounds[i][1]

    def evaluate(self):
        # Evaluate the fitness of the particle
        score = objective_function(self.position)
        # Update personal best if the new position is better
        if score < self.best_score:
            self.best_score = score
            self.best_position = self.position.copy()

# Particle Swarm Optimization (PSO) algorithm
def particle_swarm_optimization(num_particles, bounds, inertia_weight, cognitive_coef, social_coef,
max_iterations):
    # Initialize swarm
    swarm = [Particle(bounds) for _ in range(num_particles)]
```

```
    global_best_position = swarm[0].position.copy()
    global_best_score = objective_function(global_best_position)

    # Find initial global best
    for particle in swarm:
        if particle.best_score < global_best_score:
            global_best_score = particle.best_score
            global_best_position = particle.best_position.copy()

    # Optimization loop
    for iteration in range(max_iterations):
        for particle in swarm:
            # Update particle velocity and position
            particle.update_velocity(global_best_position, inertia_weight, cognitive_coef, social_coef)
            particle.update_position(bounds)
            particle.evaluate()

            # Update global best if the particle's best position is better
            if particle.best_score < global_best_score:
                global_best_score = particle.best_score
                global_best_position = particle.best_position.copy()

    # After all iterations, print the final global best score
    print(f"Final Iteration {max_iterations}, Global Best Score: {global_best_score}")
    return global_best_position, global_best_score

# Define the parameters for PSO
num_particles = 30                # Number of particles in the swarm
bounds = [(5, 10), (5, 10)]       # Bounds for the search space (e.g., x and y can vary from -10 to 10)
inertia_weight = 0.5              # Inertia weight
cognitive_coef = 1.5              # Cognitive (personal best) coefficient
social_coef = 1.5                 # Social (global best) coefficient
max_iterations = 100              # Number of iterations to run the algorithm

# Run PSO to find the minimum of the objective function
best_position, best_score = particle_swarm_optimization(num_particles, bounds, inertia_weight,
cognitive_coef, social_coef, max_iterations)

print(f"Best Position: {best_position}")
print(f"Best Score: {best_score}")
```

```
Final Iteration 100, Global Best Score: 50.0
Best Position: [5. 5.]
Best Score: 50.0
```

Program 3
Problem statement

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

Algorithm:

DOMS Page No.
Date / /

Ant Colony Optimization

```
Initialize cities, distances, pheromone, parameters and ants

best_path = none
best_distance = ∞

for i in iterations:
    list = []

        For ant in range(n_ants):
            path[0] = random(cities)
            while path != destination:
                current_city = path[-1]
                calculate_probabilities()
            path_distance += distance[current_city, next_city]

            if path_distance < best_distance:
                best_distance = path_distance
                best_path = path
        ant.distance = path_distance
        ant.path = path, list.add(path)
    update_pheromone()

return best_path and best_distance
```

Date / /

```
calculate_probabilities ()
    probabilities = []
    for next city in range (cities):
        if next_city not in path:
            pheromone = pher_matrix [current_city, next_city] ^ α
            heuristic = ( 1/ distance [current_city , next_city] ^ β
            probabilities append [next_city , pheromone * heuristic]
    else
        probabilities . append [(next_city, 0))]

    total = sum (probabilities)
    probabilities = [(city , prob/total if total>0 else 0))]

    selected_city = random. choices ([city for city, _ in probabilities]

        weights = [prob for _, prob in probabilities ], k=1 [0]
    return . selected_city

update_pheromones ().
    pheromone_matrix *= (1-γ)

    for path distance in (paths, distances)
        for i in range (len(path)-1)
            pheromone_matrix [path[i], path[i+1]]
                    += pheromone / distance
        pheromone_matrix [path[-1], path[0]]
                    += deposit / distance
```

14/11/24

Code:

```
import numpy as np
import random

class ACO:
    def __init__(self, n_ants, n_iterations, alpha, beta, rho, pheromone_deposit, cities):
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha  # importance of pheromone
        self.beta = beta    # importance of heuristic information
        self.rho = rho      # pheromone evaporation rate
        self.pheromone_deposit = pheromone_deposit  # pheromone deposit for the best path
        self.cities = cities
        self.num_cities = len(cities)
        self.distances = self.calculate_distances(cities)
        self.pheromone_matrix = np.ones((self.num_cities, self.num_cities))

    def calculate_distances(self, cities):
        distances = np.zeros((len(cities), len(cities)))
        for i, (x1, y1) in enumerate(cities):
            for j, (x2, y2) in enumerate(cities):
                distances[i, j] = np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
        return distances

    def run(self):
        best_distance = float('inf')
        best_path = None

        for iteration in range(self.n_iterations):
            all_paths = []
            all_distances = []

            for ant in range(self.n_ants):
                path = self.construct_solution()
                path_distance = self.calculate_path_distance(path)
                all_paths.append(path)
                all_distances.append(path_distance)

                if path_distance < best_distance:
                    best_distance = path_distance
                    best_path = path

            self.update_pheromones(all_paths, all_distances)
            print(f"Iteration {iteration+1}: Best Distance = {best_distance}")

        return best_path, best_distance

    def construct_solution(self):
        path = [random.randint(0, self.num_cities - 1)]
        while len(path) < self.num_cities:
            current_city = path[-1]
            next_city = self.choose_next_city(current_city, path)
```

```
            path.append(next_city)
        return path

    def choose_next_city(self, current_city, path):
        probabilities = []
        for next_city in range(self.num_cities):
            if next_city not in path:
                pheromone = self.pheromone_matrix[current_city, next_city] ** self.alpha
                heuristic = (1.0 / self.distances[current_city, next_city]) ** self.beta
                probabilities.append((next_city, pheromone * heuristic))
            else:
                probabilities.append((next_city, 0))

        total = sum(prob for _, prob in probabilities)
        probabilities = [(city, prob / total if total > 0 else 0) for city, prob in probabilities]

        selected_city = random.choices(
            [city for city, _ in probabilities],
            weights=[prob for _, prob in probabilities],
            k=1
        )[0]
        return selected_city

    def calculate_path_distance(self, path):
        distance = 0
        for i in range(len(path) - 1):
            distance += self.distances[path[i], path[i + 1]]
        distance += self.distances[path[-1], path[0]]  # return to start
        return distance

    def update_pheromones(self, paths, distances):
        self.pheromone_matrix *= (1 - self.rho)  # evaporate pheromones

        # deposit pheromones based on path quality
        for path, distance in zip(paths, distances):
            for i in range(len(path) - 1):
                self.pheromone_matrix[path[i], path[i + 1]] += self.pheromone_deposit / distance
            self.pheromone_matrix[path[-1], path[0]] += self.pheromone_deposit / distance  # return to start

# Example usage with random cities
cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(10)]
print(cities)
aco = ACO(n_ants=10, n_iterations=100, alpha=1, beta=2, rho=0.5, pheromone_deposit=10, cities=cities)
best_path, best_distance = aco.run()

print("\nBest Path:", best_path)
print("Best Distance:", best_distance)
```

```
Iteration 60: Best Distance = 326.99882689334635
Iteration 61: Best Distance = 326.99882689334635
Iteration 62: Best Distance = 326.99882689334635
Iteration 63: Best Distance = 326.99882689334635
Iteration 64: Best Distance = 326.99882689334635
Iteration 65: Best Distance = 326.99882689334635
Iteration 66: Best Distance = 326.99882689334635
Iteration 67: Best Distance = 326.99882689334635
Iteration 68: Best Distance = 326.99882689334635
Iteration 69: Best Distance = 326.99882689334635
Iteration 70: Best Distance = 326.99882689334635
Iteration 71: Best Distance = 326.99882689334635
Iteration 72: Best Distance = 326.99882689334635
Iteration 73: Best Distance = 326.99882689334635
Iteration 74: Best Distance = 326.99882689334635
Iteration 75: Best Distance = 326.99882689334635
Iteration 76: Best Distance = 326.99882689334635
Iteration 77: Best Distance = 326.99882689334635
Iteration 78: Best Distance = 326.99882689334635
Iteration 79: Best Distance = 326.99882689334635
Iteration 80: Best Distance = 326.99882689334635
Iteration 81: Best Distance = 326.99882689334635
Iteration 82: Best Distance = 326.99882689334635
Iteration 83: Best Distance = 326.99882689334635
Iteration 84: Best Distance = 326.99882689334635
Iteration 85: Best Distance = 326.99882689334635
Iteration 86: Best Distance = 326.99882689334635
Iteration 87: Best Distance = 326.99882689334635
Iteration 88: Best Distance = 326.99882689334635
Iteration 89: Best Distance = 326.99882689334635
Iteration 90: Best Distance = 326.99882689334635
Iteration 91: Best Distance = 326.99882689334635
Iteration 92: Best Distance = 326.99882689334635
Iteration 93: Best Distance = 326.99882689334635
Iteration 94: Best Distance = 326.99882689334635
Iteration 95: Best Distance = 326.99882689334635
Iteration 96: Best Distance = 326.99882689334635
Iteration 97: Best Distance = 326.99882689334635
Iteration 98: Best Distance = 326.99882689334635
Iteration 99: Best Distance = 326.99882689334635
Iteration 100: Best Distance = 326.99882689334635

Best Path: [7, 9, 5, 4, 1, 3, 8, 0, 6, 2]
Best Distance: 326.99882689334635
```

Program 4

Problem statement

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.

Algorithm:

Page No.
Date / /

Cuckoo Search Algorithm

```
function CuckooSearch (func, bounds), n_nests = 25, n_iterations = 1000,
                       pa = 0.25):
    nests = InitializeNests (bounds, n_nests)
    fitness = EvaluateFitness (nests, func)

    best_idx = Index_of_minimum (fitness)
    best_solution = nests [best_idx]
    best_fitness = fitness [best_idx]

    for iteration = (1 to) n_iterations:
        new_nests = GenerateNewSolutions (nests)
        new_fitness = EvaluateFitness (new_nests, func)

        for i=1 to n_nests:
            if random() < pa
                new_nests [i] = GenerateRandomSolution (bounds)

        for i=1 to n_nests:
            if new_fitness [i] < fitness [i]
                nests [i] = new_nests [i]
                fitness [i] = new_fitness [i]

        best_idx = Index_of_minimum (fitness)
        if fitness [best_idx] < best_fitness:
            best_solution = nests [best_idx]
            best_fitness = fitness [best_idx]
        print "Iteration", iteration, "Best Fitness", best_fitness
    return best_solution, best_fitness
```

```
function InitializeNests (bounds, n_nests):
    nests = random_array (n_nests, length (bounds))
    for i = 1 to length (bounds):
        nests [:, i] = random_value_in_range (bounds [i])
    return nests

function EvaluateFitness (nests, func):
    fitness = []
    for each nest in nests:
        fitness.append (func (nest))
    return fitness

function GenerateNewSolutions (bounds):
    random_solution = []
    for i = 1 to length (bounds):
        random_solution [i] = random_value_in_range (bounds [i])
    return random_solution

function SphereFunction (x):
    return sum (x[i]^2 for i in range (len(x)))

bounds = [(-5.0, 5.0), (-5.0, 5.0)]
best_solution, best_fitness = CuckooSearch (SphereFunction, bounds)
print "Best Solution:", best_solution
print "Best Fitness:", best_fitness
```

8/8
2/11/24

Code:
```
import numpy as np

def cuckoo_search(func, bounds, n_nests=25, n_iterations=1000, pa=0.25):
    dim = len(bounds)
    nests = initialize_nests(bounds, n_nests)
    fitness = evaluate_fitness(nests, func)
    best_idx = np.argmin(fitness)
    best_solution = nests[best_idx]
    best_fitness = fitness[best_idx]

    for iteration in range(n_iterations):
        new_nests = generate_new_solutions(nests)
        new_fitness = evaluate_fitness(new_nests, func)

        for i in range(n_nests):
            if np.random.rand() < pa:
                new_nests[i] = generate_random_solution(bounds)

        for i in range(n_nests):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        best_idx = np.argmin(fitness)
        if fitness[best_idx] < best_fitness:
            best_solution = nests[best_idx]
            best_fitness = fitness[best_idx]

        #print(f"Iteration {iteration + 1}/{n_iterations}, Best Fitness: {best_fitness}")

    return best_solution, best_fitness

def initialize_nests(bounds, n_nests):
    nests = np.random.rand(n_nests, len(bounds))
    for i in range(len(bounds)):
        nests[:, i] = bounds[i][0] + (bounds[i][1] - bounds[i][0]) * nests[:, i]
    return nests

def evaluate_fitness(nests, func):
    return np.array([func(nest) for nest in nests])

def generate_new_solutions(nests):
    new_nests = np.copy(nests)
    levy_flight = np.random.normal(0, 1, size=nests.shape) * np.abs(np.random.normal(0, 1, size=nests.shape))
    new_nests += levy_flight
    return new_nests
```

```
def generate_random_solution(bounds):
    return np.array([np.random.uniform(bounds[i][0], bounds[i][1]) for i in range(len(bounds))])

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

bounds = [(-5.0, 5.0), (-5.0, 5.0)]

best_solution, best_fitness = cuckoo_search(sphere_function, bounds)

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

```
Best Solution: [-0.00143299  0.00383832]
Best Fitness: 1.6786154692913296e-05
```

Program 5
Problem statement

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.

Algorithm:

Date / /

Grey Wolf Optimization

```
Function Grey wolf Optimizer
    Initialize alpha_pos beta_pos & delta_pos as a zero vector
    Initialize alpha_score, beta_score, delta_score as infinity.

    Initialize positions -> random values btw lb & ub
    Initialize scores as a zero vector.

    for t = 1 to max_iter:
        a = 2 - (t * (2 / max_iters))
        for i = 1 to n_wolves
            score = obj_func (positions [i])
            scores [i] = score

        if score < alpha_score:
            alpha_score = score
            alpha_pos = positions [i]
        elseif score < beta_score:
            beta score = score
            beta_pos = positions [i].
        else if score < delta_score:
            delta_score = score
            delta_pos = positions [i].

    for i = 1 to n_wolves:
        r1 = random values in bounds
        r2 = random values in bounds.
        A = 2* a * r1 - a.
        C = 2* r2
```

```
D_alpha = abs (C* alpha_pos - position [i])
D_beta = abs (C * beta_pos - position [i])
D_delta = abs (C * delta_pos - positions [i])

x1 = alpha_pos - A * D_alpha
x2 = beta_pos - A * D_beta
x3 = delta_pos - A * D_delta

positions [i] = (x1 + x2 + x3) / 3
positions [i] = clip (positions [i], lb, ub)
return alpha_pos, alpha_score

Function clip (value, lb, ub):
    return value in range [lb, ub]

def Objective_function (x):
    return sum of x ** 2
```

Code:

```
import numpy as np

# Objective function (Example: Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Grey Wolf Optimization Algorithm
class GreyWolfOptimizer:
    def __init__(self, obj_func, dim, n_wolves, max_iter, lb, ub):
        self.obj_func = obj_func  # Objective function
        self.dim = dim            # Dimensionality of the problem
        self.n_wolves = n_wolves # Number of wolves
        self.max_iter = max_iter # Maximum number of iterations
        self.lb = lb              # Lower bound of the search space
        self.ub = ub              # Upper bound of the search space
        self.alpha_pos = np.zeros(dim)  # Position of alpha wolf
        self.alpha_score = float("inf") # Fitness of alpha wolf
        self.beta_pos = np.zeros(dim)   # Position of beta wolf
        self.beta_score = float("inf")  # Fitness of beta wolf
        self.delta_pos = np.zeros(dim)  # Position of delta wolf
        self.delta_score = float("inf") # Fitness of delta wolf
        self.positions = np.random.rand(n_wolves, dim) * (ub - lb) + lb  # Initial positions of wolves
        self.scores = np.zeros(n_wolves)  # Fitness scores

    def optimize(self):
        # Main optimization loop
        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)  # Decreases linearly from 2 to 0
            for i in range(self.n_wolves):
                # Evaluate fitness of each wolf
                self.scores[i] = self.obj_func(self.positions[i])

                # Update alpha, beta, and delta wolves
                if self.scores[i] < self.alpha_score:
                    self.alpha_score = self.scores[i]
                    self.alpha_pos = self.positions[i]
                elif self.scores[i] < self.beta_score:
                    self.beta_score = self.scores[i]
                    self.beta_pos = self.positions[i]
                elif self.scores[i] < self.delta_score:
                    self.delta_score = self.scores[i]
                    self.delta_pos = self.positions[i]

            # Update the positions of the wolves
            for i in range(self.n_wolves):
```

```
            # Calculate random values for A and C
            r1 = np.random.rand(self.dim)
            r2 = np.random.rand(self.dim)
            A = 2 * a * r1 - a
            C = 2 * r2

            # Update the position of the wolf
            D_alpha = np.abs(C * self.alpha_pos - self.positions[i])
            D_beta = np.abs(C * self.beta_pos - self.positions[i])
            D_delta = np.abs(C * self.delta_pos - self.positions[i])

            X1 = self.alpha_pos - A * D_alpha
            X2 = self.beta_pos - A * D_beta
            X3 = self.delta_pos - A * D_delta

            # Calculate new position for the wolf
            self.positions[i] = (X1 + X2 + X3) / 3

            # Apply boundary constraints (if any)
            self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)

        # Optionally, print the best solution found so far
        #print(f"Iteration {t+1}/{self.max_iter} - Best Score: {self.alpha_score}")

    # Return the best solution found
    return self.alpha_pos, self.alpha_score

# Hyperparameters
dim = 30            # Number of dimensions (variables)
n_wolves = 50       # Number of wolves (population size)
max_iter = 1000     # Maximum number of iterations
lb = -10            # Lower bound of search space
ub = 10             # Upper bound of search space

# Instantiate the optimizer
optimizer = GreyWolfOptimizer(obj_func=objective_function, dim=dim, n_wolves=n_wolves,
max_iter=max_iter, lb=lb, ub=ub)

# Perform optimization
best_position, best_score = optimizer.optimize()

# Output the best solution found
print("\nBest Position: ", best_position)
print("Best Score: ", best_score)
```

```
Best Position:  [ 3.31543532e-28 -2.57971219e-28  2.90350626e-28 -3.27713250e-28
  3.52185014e-28  2.80085911e-28 -3.38381673e-28 -2.97466794e-28
 -2.31745008e-28  3.13252393e-28 -2.87816050e-28  1.79119454e-28
 -2.84588645e-28  2.90763602e-28 -3.38953643e-28 -3.35192731e-28
 -2.62987429e-28 -3.10876600e-28  3.13119841e-28  3.25839295e-28
 -2.77855075e-28  3.09139060e-28  2.99660816e-28  2.85167667e-28
 -2.75530248e-28 -3.56770417e-28  2.01980511e-28  3.23116555e-28
 -3.70571356e-28  3.36635177e-28]
Best Score:  2.744736165706468e-54
```

Program 6
Problem statement

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.

Algorithm:

Date / /

Parallel Cellular Algorithm

```
Function objective_function (x):
    return x^2 - 4*x + 4

Function initialize_population ():
    create random grid with values in range [-10,10]
        of size 10x10
    return grid as population

Function evaluate_fitness (population)
    create empty fitness array
    for each cell in population:
        fitness[cell] = objective_function (population[cell])
    return fitness

Function update_cell_state (population, fitness, neighbourhood_size)
    temp ← population
    neigh_radius ← neighbourhood_size // 2

    for i ← 0 to r-1
        for j ← 0 to c-1
            neighbourhood ← {}

            for di from -neigh_radius to neigh_radius:
                for dj from -neigh_radius to neigh_radius:
                    ni = i + di
                    nj = j + dj
                    if (ni, nj) ≤ bounds
                        add(population[ni, nj],
                            fitness[ni, nj]) to
                                neighbourhood
```

Page No.
Date / /

```
        SORT neighbourhood.
        best_neighbour = neighbour [0]
        new_population [i,j] = best_neigh (-0.1, 0.1)
    return new_population

Function parallel_cellular_algorithm ():
    population ← initialize_population ()
    fitness ← evaluate_fitness (population)
    best_sol ← none
    best_fit ← infinity.

    for iteration from i to iterations:
        new_population = update_cell_state (population, fitness,
                                            neighbourhood = size)
        fitness ← evaluate_fitness (new_population)

        find min (fitness)
        if min_fitness < best_fitness:
            best_fitness ← min_fitness
            best_solution ← solution in new_population.
        if best_fitness < converge_threshold:
            break.
```

Code:

```
import numpy as np
import random

# Step 1: Define the optimization function (Example: minimizing the function f(x) = x^2)
def objective_function(x):
    return x ** 2 - 4*x+4

# Step 2: Initialize parameters
num_cells = 100            # Number of cells (solutions)
grid_size = (10, 10)       # Grid size (10x10)
iterations = 1000          # Number of iterations
neighborhood_size = 3      # Neighborhood size (3x3)
convergence_threshold = 0.000001  # Convergence threshold

# Step 3: Initialize the population (randomly generate cell positions)
def initialize_population():
    # Create a grid with random positions for each cell in the search space [-10, 10]
    population = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1]))
    return population

# Step 4: Evaluate the fitness of each cell
def evaluate_fitness(population):
    # Apply the objective function to each cell
    fitness = np.vectorize(objective_function)(population)
    return fitness

# Step 5: Define a function to update the state of each cell based on neighboring cells
def update_cell_state(population, fitness, neighborhood_size):
    rows, cols = population.shape
    new_population = population.copy()

    # Define the neighborhood boundaries
    neighborhood_radius = neighborhood_size // 2

    for i in range(rows):
        for j in range(cols):
            # List of neighboring cell positions, including the current cell
            neighborhood = []
            for di in range(-neighborhood_radius, neighborhood_radius + 1):
                for dj in range(-neighborhood_radius, neighborhood_radius + 1):
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols:  # Ensure indices are within bounds
                        neighborhood.append((population[ni, nj], fitness[ni, nj]))

            # Sort neighbors based on fitness value (ascending order: better solutions have lower fitness)
            neighborhood.sort(key=lambda x: x[1])
            best_neighbor = neighborhood[0]

            # Update the current cell based on the best neighbor (with some random fluctuation)
            new_population[i, j] = best_neighbor[0] + random.uniform(-0.1, 0.1)  # Slight random movement
    return new_population

# Step 6: Iterate to update the states of the cells
```

```python
def parallel_cellular_algorithm():
    population = initialize_population()
    fitness = evaluate_fitness(population)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(iterations):
        print(f"Iteration {iteration + 1}/{iterations}")

        # Update cell states in parallel (Here we simulate parallel updates by using numpy)
        new_population = update_cell_state(population, fitness, neighborhood_size)

        # Evaluate the new population's fitness
        fitness = evaluate_fitness(new_population)

        # Track the best solution found so far
        min_fitness_index = np.argmin(fitness)
        min_fitness_value = fitness.flatten()[min_fitness_index]

        if min_fitness_value < best_fitness:
            best_fitness = min_fitness_value
            best_solution = new_population.flatten()[min_fitness_index]

        population = new_population  # Update population for next iteration

        # Check for convergence (early stop if we find a very small fitness value)
        if best_fitness < convergence_threshold:
            print("Convergence reached!")
            break

    return best_solution, best_fitness

# Step 7: Run the algorithm and output the best solution
best_solution, best_fitness = parallel_cellular_algorithm()

print(f"The best solution found is: {best_solution}")
print(f"The corresponding fitness (objective function value) is: {best_fitness}")
```

```
Iteration 1/1000
Iteration 2/1000
Iteration 3/1000
Convergence reached!
The best solution found is: 1.9995331188038894
The corresponding fitness (objective function value) is: 2.1797805116463564e-07
```

Program 7

Problem statement

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.

Algorithm:

<u>Gene Expression Algorithm:</u>

```
Function objective_function (x,y):
    return x** 2 + y** 2

Function initialize_population ():
    for i in population_size:
        population [i] ← random. val in range [bounds]
                    for each gene.

Function evaluate_fitness (populations)
    fitness ← []
    for individual in population:
        fitness_val = objective_function (individual [0], [1])
        fitness.append ( fitness_val)
    return fitness

Function select_Parents (population, fitness):
    probabilities ← 1 / (fitness + 1e-6)
    probabilities /= sum (probabilities)
    indices = random set of indices from population based
              on probabilities.
    return population [indices]

Function crossover ( parent1, parent2):
    If random < crossover_rate:
        point ← random int in 1 to num_genes
        child1 = concatenate ( parent1 [0: point], parent2 [0: point])
        child2 = concatenate (parent2 [0: point], parent1 [0: point])
        return child1 , child2
    else
        return parent1, parent2.
```

```
Function mutate (Individual):
    for i ← 0 to num_genes:
        if random() < mutation rate:
            individual [i] = individual [i] + Random [-1,1]
            individual [i] = clip (individual [i], bounds]
    return individual.

Function gene_expression_algorithm ():
    population = initialize_population ()
    best_solution ← NULL
    best_fitness ← INFINITE.

    For generations from 1 to num_generations:
        fitness = evaluate_fitness (population)

        if min (fitness) < best_fitness:
            best_fitness = min (fitness)
            best_solution = population (fitness)
    parents = select_parents (population (fitness)

    offspring ← []

    for i ← 0 to population size with step size - 2
        parent 1 = parents [i]
        parent 2 = parents [i+1]
        child1, child2 = crossover (parent1, parent 2)
        offspring. append (mutate (child1))
        offspring. append (mutate (child2))

return best_solution, best_fitness.
```

9/12/25

Code:
```
import numpy as np

# Problem definition
def objective_function(x, y):
    return x**2 + y**2  # Example function to minimize

# Initialize parameters
population_size = 100
num_genes = 2  # For (x, y) problem
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 50
gene_bounds = (-10, 10)  # Range for each gene (x, y)

# Helper functions
def initialize_population():
    return np.random.uniform(gene_bounds[0], gene_bounds[1], (population_size, num_genes))

def evaluate_fitness(population):
    return np.array([objective_function(ind[0], ind[1]) for ind in population])

def select_parents(population, fitness):
    probabilities = 1 / (fitness + 1e-6)  # Convert fitness to probabilities
    probabilities /= probabilities.sum()
    indices = np.random.choice(np.arange(population_size), size=population_size, p=probabilities)
    return population[indices]

def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        point = np.random.randint(1, num_genes)
        child1 = np.concatenate((parent1[:point], parent2[point:]))
        child2 = np.concatenate((parent2[:point], parent1[point:]))
        return child1, child2
    return parent1, parent2

def mutate(individual):
    for i in range(num_genes):
        if np.random.rand() < mutation_rate:
            individual[i] += np.random.uniform(-1, 1)
            individual[i] = np.clip(individual[i], gene_bounds[0], gene_bounds[1])
    return individual

# Main GEA process
def gene_expression_algorithm():
    population = initialize_population()
    best_solution = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        if fitness.min() < best_fitness:
            best_fitness = fitness.min()
```

```
            best_solution = population[fitness.argmin()]

        parents = select_parents(population, fitness)
        offspring = []

        for i in range(0, population_size, 2):
            parent1, parent2 = parents[i], parents[i + 1]
            child1, child2 = crossover(parent1, parent2)
            offspring.append(mutate(child1))
            offspring.append(mutate(child2))

        population = np.array(offspring)

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness:.5f}, Best Solution = {best_solution}")

    return best_solution, best_fitness

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm()
print("\nBest Solution Found:")
print(f"Solution: {best_solution}, Fitness: {best_fitness:.5f}")
```

```
Generation 9: Best Fitness = 0.00282, Best Solution = [-0.05287625 -0.00484865]
Generation 10: Best Fitness = 0.00282, Best Solution = [-0.05287625 -0.00484865]
Generation 11: Best Fitness = 0.00282, Best Solution = [-0.05287625 -0.00484865]
Generation 12: Best Fitness = 0.00282, Best Solution = [-0.05287625 -0.00484865]
Generation 13: Best Fitness = 0.00043, Best Solution = [-0.02010828 -0.00484865]
Generation 14: Best Fitness = 0.00043, Best Solution = [-0.02010828 -0.00484865]
Generation 15: Best Fitness = 0.00043, Best Solution = [-0.02010828 -0.00484865]
Generation 16: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 17: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 18: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 19: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 20: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 21: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 22: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 23: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 24: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 25: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 26: Best Fitness = 0.00008, Best Solution = [ 0.00727875 -0.00484865]
Generation 27: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 28: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 29: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 30: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 31: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 32: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 33: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 34: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 35: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 36: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 37: Best Fitness = 0.00007, Best Solution = [0.00727875 0.0046719 ]
Generation 38: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 39: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 40: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 41: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 42: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 43: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 44: Best Fitness = 0.00006, Best Solution = [ 0.00727875 -0.00304049]
Generation 45: Best Fitness = 0.00006, Best Solution = [-0.0056377  -0.00484865]
Generation 46: Best Fitness = 0.00006, Best Solution = [-0.0056377  -0.00484865]
Generation 47: Best Fitness = 0.00006, Best Solution = [-0.0056377  -0.00484865]
Generation 48: Best Fitness = 0.00006, Best Solution = [-0.0056377  -0.00484865]
Generation 49: Best Fitness = 0.00006, Best Solution = [-0.0056377  -0.00484865]
Generation 50: Best Fitness = 0.00004, Best Solution = [-0.0056377  -0.00304049]

Best Solution Found:
Solution: [-0.0056377  -0.00304049], Fitness: 0.00004
```