VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Priyanshu Kumar (1BM22CS210)

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 **Priyanshu Kumar (1BM22CS210),** who is a 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.

Sowmya T Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Repo: https://github.com/pkcs210/BIS-Lab/tree/main

Program 1

Genetic Algorithm for Optimization Problems:

Problem Statement:
Genetic algorithms are a type of optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a particular function. Genetic algorithms represent one branch of the field of study called evolutionary computation, in that they imitate the biological processes of reproduction and natural selection to solve for the 'fittest' solutions.

Code:

```
import random
import math

def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function

def initialize_population(size, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in range(size)]

def evaluate_fitness(population):
    return [function_to_optimize(ind) for ind in population]

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

def crossover(parent1, parent2):
    return (parent1 + parent2) / 2

def mutate(individual, mutation_rate, bounds):
    if random.random() < mutation_rate:
        return random.uniform(bounds[0], bounds[1])
    return individual
```

```
def genetic_algorithm(pop_size, bounds, mutation_rate, crossover_rate,
generations):
    population = initialize_population(pop_size, bounds)

    for _ in range(generations):
        fitness = evaluate_fitness(population)
        new_population = []

        for _ in range(pop_size):
            parent1, parent2 = select_parents(population, fitness)
            if random.random() < crossover_rate:
                offspring = crossover(parent1, parent2)
            else:
                offspring = random.choice([parent1, parent2])
            offspring = mutate(offspring, mutation_rate, bounds)
            new_population.append(offspring)

        population = new_population

    best_individual = max(population, key=function_to_optimize)
    return best_individual

# Parameters
population_size = 20
bounds = (-10, 10)
mutation_rate = 0.1
crossover_rate = 0.7
generations = 50

# Run Genetic Algorithm
best_solution = genetic_algorithm(population_size, bounds, mutation_rate,
crossover_rate, generations)
print("Best Solution:", best_solution)
print("Maximum Value:", function_to_optimize(best_solution))
```

Screenshots:

LAB-1

Genetic Algorithm for Optimization Problems

Algorithm:

- Define the Problem: Choose the mathematical function to optimize (e.g., $f(x) = x^2$).
- Initialize Parameters: Set population size, mutation rate, crossover rate, and number of generations.
- Create Initial Population: Generate random potential solutions (individuals).
- Evaluate Fitness: Calculate how good each solution is based on the function (fitness).
- Selection: Select the fittest individuals to reproduce (better solutions are chosen).
- Crossover: Combine parts of two parents to create new solutions (offspring).
- Mutation: Randomly tweak some solutions to maintain diversity.
- Iteration: Repeat fitness evaluation, selection, crossover, & mutation for many generations.

• Output
return

CODE:

import
import

def f

def

def

def

def

def

Problems.

matical
x) = x ^ 2).

ize,
and

om
ds).

ach
nction

to
chosen).

nts to

lutions

ction,
ny

• Output Best Solution: After multiple generations, return the best solution found.

CODE:

```
import random
import math

def function_to_optimize(x):
    return -x**2 + 4*x + 10

def initialize_population(size, bounds):
    return [random.uniform(bounds[0], bounds[1])
            for _ in range(size)]

def evaluate_fitness(population):
    return [function_to_optimize(ind) for ind
            in population]

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

def crossover(parent1, parent2)
    return (parent1 + parent2)/2

def mutate(individual, mutation_rate, bounds):
    if random.random() < mutation_rate:
        return random.uniform(bounds[0],
                              bounds[1])
    return individual
```

```
def genetic_algorithm(pop_size, bounds, mutation_rate,
                      crossover_rate, generations):
    population = initialize_population(pop_size,
for _ in range(generations):  bounds)
    fitness = evaluate_fitness(population)
    for _ in range(pop_size):  new_population
        parent1, parent2 = select_parents(
                              population, fitness)
        if random.random() < crossover_rate:
            offspring = crossover(parent1, parent2)
        else:
            offspring = random.choice([parent1,
                                       parent2])
        offspring = mutate(offspring, mutation_rate,
                           bounds)
        new_population.append(offspring)

    population = new_population
best_individual = max(population, key=function
                                  to_optimize)

# Parameters
population_size = 20
bounds = (-10, 10)
mutation_rate = 0.1
crossover_rate = 0.7
generations = 50

best_solution = genetic_algorithm(population_size,
                bounds, mutation_rate, crossover_rate,
                generation)
```

```
print("Best Solution:", best_solution)
print("Maximum Value:", function_to_optimize
                        (best_solution))
```

Program 2

Particle Swarm Optimization

Problem Statement:
The objective is to optimize a given function using Particle Swarm Optimization (PSO). PSO is a population-based metaheuristic algorithm inspired by the social behavior of particles in a swarm. The algorithm aims to find the optimal solution by iteratively adjusting particle positions and velocities within a defined search space to minimize or maximize the objective function. The search considers constraints, if any, to ensure feasibility

Code:

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function

def initialize_particles(num_particles, bounds):
        positions = [random.uniform(bounds[0], bounds[1]) for _ in
range(num_particles)]
    velocities = [random.uniform(-1, 1) for _ in range(num_particles)]
    return positions, velocities

def update_velocity(velocity, position, personal_best, global_best, w, c1,
c2):
    inertia = w * velocity
    cognitive = c1 * random.random() * (personal_best - position)
    social = c2 * random.random() * (global_best - position)
    return inertia + cognitive + social

def update_position(position, velocity, bounds):
    new_position = position + velocity
    return max(bounds[0], min(bounds[1], new_position))

def particle_swarm_optimization(num_particles, bounds, w, c1, c2,
iterations):
    positions, velocities = initialize_particles(num_particles, bounds)
    personal_bests = positions[:]
```

```
    global_best = max(positions, key=function_to_optimize)

    for _ in range(iterations):
        for i in range(num_particles):
            velocities[i] = update_velocity(velocities[i], positions[i],
personal_bests[i], global_best, w, c1, c2)
            positions[i] = update_position(positions[i], velocities[i],
bounds)

            if function_to_optimize(positions[i]) >
function_to_optimize(personal_bests[i]):
                personal_bests[i] = positions[i]

        global_best = max(personal_bests, key=function_to_optimize)

    return global_best

# Parameters
num_particles = 20
bounds = (-10, 10)
w = 0.5  # inertia weight
c1 = 1.5  # cognitive coefficient
c2 = 1.5  # social coefficient
iterations = 50

# Run Particle Swarm Optimization
best_solution = particle_swarm_optimization(num_particles, bounds, w, c1,
c2, iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", function_to_optimize(best_solution))
```

Screenshots:

LAB - 2

Particle Swarm Optimization for Function Optimization

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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.

2. Initialize Parameters: Set the number of particles, inertia weight, cognitive & social coefficients.

3. Initialize Particles: Generate an initial population of particles with random position & velocities.

4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.

5. Update Velocities & Positions: Update the velocity and position of each particle based on its own best position and the global best position.

6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.

7. Output the Best Solution: Track and output the best solution found during the iterations.

CODE:

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10

def initialize_particles(num_particles, bounds):
    positions = [random.uniform(bounds[0], bounds[1])
                 for _ in range(num_particles)]
    velocities = [random.uniform(-1, 1) for _ in
                  range(num_particles)]
    return positions, velocities

def update_velocity(velocity, position, personal_best,
                    global_best, w, c1, c2):
    inertia = w * velocity
    cognitive = c1 * random.random() *
                (person_best - position)
    social = c2 * random.random() *
             (global best - position)
    return inertia + cognitive + social
```

```
def update_position(position, velocity, bounds):
    new_position = position + velocity
    return max(bounds[0], min(bounds[1],
                              new_position))

def particle_swarm_optimization(num_particles,
                                bounds, w, c1, c2, iterations):
    positions, velocities = initialize_particles(
                                num_particles, bounds)
    personal_bests = positions[:]
    global_best = max(positions, key=function_to_
                                optimize)

    for _ in range(iterations):
        for i in range(num_particles):
            velocities[i] = update_velocity(
                velocities[i], positions[i],
                personal_best[i], global_best,
                w, c1, c2)
            positions[i] = update_positions(
                positions[i], velocities[i], bounds)

            if function_to_optimize(positions[i]) >
                function_to_optimize(person_best[i]):
                person_bests[i] = positions[i]

        global_best = max(person_bests,
                          key = function_to_optimize)
    return global_best
```

, bounds):
city
(bounds[I],
n))

num_particles,
rations):

particles(
bounds)

= function_to_
optimize)

s):
ity(
s[i],
l_best,

ons(
es[i], bounds)

ions[i]>
rson_best[i]:
sitions[i]

ests,
optimize)

num_particles = 20
bounds = (-10, 10)
w = 0.5
c1 = 1.5
c2 = 1.5
iterations = 50

best_solution = particle_swarm_optimization(
num_particles, bounds, w, c1, c2,
iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", function_to_optimize(
best_solution))

Program 3

Ant Colony Optimization

Problem Statement:
Ant Colony Optimization (ACO) algorithm solves combinatorial optimization problem, such as finding the shortest path, optimizing resource allocation, or scheduling tasks. The algorithm should simulate the behavior of ants by using pheromone trails and heuristic information to iteratively discover and refine optimal or near-optimal solutions while adhering to the constraints of the problem.

Code:

```
import random
import math

def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def initialize_pheromones(num_cities, initial_pheromone):
    return [[initial_pheromone for _ in range(num_cities)] for _ in
range(num_cities)]

def probability(pheromone, heuristic, alpha, beta):
    return (pheromone**alpha) * (heuristic**beta)

def construct_solution(ant, cities, pheromones, alpha, beta):
    unvisited = set(range(len(cities)))
    current_city = ant
    unvisited.remove(current_city)
    path = [current_city]

    while unvisited:
        probabilities = []
        for next_city in unvisited:
            pheromone = pheromones[current_city][next_city]
                heuristic = 1 / calculate_distance(cities[current_city],
cities[next_city])
```

```
            probabilities.append(probability(pheromone, heuristic, alpha,
beta))

        total = sum(probabilities)
        probabilities = [p / total for p in probabilities]
         next_city = random.choices(list(unvisited), weights=probabilities,
k=1)[0]
        path.append(next_city)
        unvisited.remove(next_city)
        current_city = next_city

    return path

def update_pheromones(pheromones, all_paths, cities, evaporation_rate):
    for i in range(len(pheromones)):
        for j in range(len(pheromones)):
            pheromones[i][j] *= (1 - evaporation_rate)

    for path in all_paths:
         distance = sum(calculate_distance(cities[path[i]], cities[path[i +
1]]) for i in range(len(path) - 1))
        pheromone_deposit = 1 / distance
        for i in range(len(path) - 1):
            pheromones[path[i]][path[i + 1]] += pheromone_deposit
            pheromones[path[i + 1]][path[i]] += pheromone_deposit

def total_distance(path, cities):
    return sum(calculate_distance(cities[path[i]], cities[path[i + 1]]) for
i in range(len(path) - 1))

def     ant_colony_optimization(cities,     num_ants,     alpha,     beta,
evaporation_rate, initial_pheromone, iterations):
    num_cities = len(cities)
    pheromones = initialize_pheromones(num_cities, initial_pheromone)
    best_path = None
```

```
    best_distance = float('inf')

    for _ in range(iterations):
            all_paths = [construct_solution(ant % num_cities, cities,
pheromones, alpha, beta) for ant in range(num_ants)]
        update_pheromones(pheromones, all_paths, cities, evaporation_rate)

        for path in all_paths:
            distance = total_distance(path, cities)
            if distance < best_distance:
                best_path = path
                best_distance = distance

    return best_path, best_distance

# Example Usage
cities = [(0, 0), (2, 2), (2, 0), (0, 2), (1, 1)]
num_ants = 10
alpha = 1
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1
iterations = 100

best_path, best_distance = ant_colony_optimization(cities, num_ants,
alpha, beta, evaporation_rate, initial_pheromone, iterations)
print("Best Path:", best_path)
print("Shortest Distance:", best_distance)
```

Screenshots:

LAB - 3

Ant Colony Optimization for the Travelling Salesman Problem

Ant Colony Optimization (ACO) mimics ants' foraging to solve problems like the Travelling Salesman Problem (TSP), aiming to find the shortest route visiting all cities and returning to the start.

ALGORITHM:

1. Define cities with co-ordinates.
2. Set parameters: number of ants, pheromones and heuristic importance, evaporation rate, and initial pheromone levels.
3. Ants build routes based on pheromones and heuristics.
4. Update pheromones based on solution quality.
5. Repeat for a set number of iterations or until convergence.
6. Return the best solution.

CODE:

```
import
import
def cal
re
def ini
re
def fo
re
def co
```

CODE:

```
import random
import math

def calculate_distance(city1, city2):
    return Math.sqrt((city1[0] - city2[0]**2 +
                      (city1[1] - city2[1])**2)

def initialize_pheromones(num_cities, initial_pheromone):
    return [[initial_pheromone for _ in
             range(num_cities)] for _ in range
             (num_cities)]

def probability(pheromone, heuristics, alpha, beta):
    return (pheromone ** alpha) * (heuristic **
                                   beta)

def construct_solution(ant, cities, pheromones,
                       alpha, beta):
    unvisited = set(range(len(cities)))
    current_city = ant
    unvisited.remove(current_city)
    path = [current_city]
    while unvisited:
        probabilities = []
        for next_city in unvisited:
            pheromone = pheromones[current_city]
                        [next_city]
            heuristic = 1/ calculate_distance(
                cities[current_city], cities[next_city])
```

```
        probabilities.append(probability(pheromone,
                heuristic, alpha, beta))

    total = sum(probabilities)
    probabilities = [p/total for p in probabilities]
    next_city = random.choices(list(unvisited),
            weights=probabilities, k=1)[0]
    path.append(next_city)
    unvisited.remove(next_city)
    current_city = next_city
return path

def update_pheromones(pheromones, all_paths, cities,
                evaporation_rate):
    for i in range(len(pheromones)):
        for j in range(len(pheromones)):
            pheromones[i][j] *= (1 - evaporation_
                                    rate)

    for path in all_paths:
        distance = sum(calculate_distance
            (cities[path[i]], cities[path[i+1]])
                for i in range(len(path)-1))
        pheromone_deposit = 1/distance
        for i in range(len(path)-1):
            pheromones[path[i]][path[i+1]]
                += pheromone_deposit
            pheromones[path[i+1]][path[i]]
                += pheromone_deposit
```

```
def total_distance(path, cities):
    return sum(calculate_distance(cities[path[i]],
        cities[path[i+1]]) for i in range(len
        (path)-1))

def ant_colony_optimization(cities, num_ants,
        alpha, beta, evaporation_rate,
        initial_pheromone, iterations):
    num_cities = len(cities)
    pheromones = initialize_pheromones(
        number_cities, initial_pheromone)
    best_path = None
    best_distance = float('inf')

    for _ in range(iterations):
        all_paths = [construct_solution(
        ant % num_cities, cities, pheromones
        alpha, beta)
    for ant in range(num_ants)]
    update_pheromones(pheromones,
        all_paths, cities, evaporation_rate)

    for path in all_paths:
        distance = total_distance(path, cities)
        if distance < best_distance:
            best_path = path
            best_distance = distance

    return best_path, best_distance
```

probabilities.append(probability(pheromone, heuristic, alpha, beta))

```
cities = [(0, 0), (2, 2), (2, 0), (0, 2), (1, 1)]
num_ants = 10
alpha = 1
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1
iterations = 100

best_path, best_distance = ant_colony_optimization(
        cities, num_ants, alpha, beta,
        evaporation_rate, initial_pheromone,
        iterations)

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

Program 4

Cuckoo Search Optimization

Problem Statement:
Design a Cuckoo Search algorithm to solve an optimization problem by mimicking the behavior of cuckoo birds laying eggs in host nests. The algorithm should use Lévy flights to explore the solution space, evaluate the fitness of solutions, and iteratively replace weaker solutions with stronger ones, aiming to find the optimal result for the given objective.

Code:

```
import random
import math


def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function


def levy_flight():
    beta = 1.5
    sigma = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
             (math.gamma((1 + beta) / 2) * beta * 2**((beta - 1) / 2)))**(1
/ beta)
    u = random.gauss(0, sigma)
    v = random.gauss(0, 1)
    return u / abs(v)**(1 / beta)


def initialize_nests(num_nests, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in range(num_nests)]


def abandon_worst_nests(nests, fitness, discovery_rate, bounds):
    num_abandoned = int(len(nests) * discovery_rate)
    worst_indices = sorted(range(len(fitness)), key=lambda i:
fitness[i])[:num_abandoned]
    for i in worst_indices:
        nests[i] = random.uniform(bounds[0], bounds[1])
    return nests
```

```
def cuckoo_search(num_nests, bounds, discovery_rate, iterations):
    nests = initialize_nests(num_nests, bounds)
    fitness = [function_to_optimize(nest) for nest in nests]
    best_nest = nests[fitness.index(max(fitness))]

    for _ in range(iterations):
        for i in range(num_nests):
            step = levy_flight()
            new_nest = nests[i] + step
            new_nest = max(bounds[0], min(bounds[1], new_nest))

            if function_to_optimize(new_nest) > fitness[i]:
                nests[i] = new_nest
                fitness[i] = function_to_optimize(new_nest)

        best_nest = nests[fitness.index(max(fitness))]
        nests = abandon_worst_nests(nests, fitness, discovery_rate, bounds)
        fitness = [function_to_optimize(nest) for nest in nests]

    return best_nest, function_to_optimize(best_nest)

# Parameters
num_nests = 20
bounds = (-10, 10)
discovery_rate = 0.25
iterations = 50

# Run Cuckoo Search
best_solution, best_value = cuckoo_search(num_nests, bounds,
discovery_rate, iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", best_value)
```

Screenshots:

LAB - 4

Cuckoo Search

Cuckoo Search (CS) is an optimisation algorithm inspired by the brood parasitism of cuckoo birds, leveraging Lévy flights for exploring solutions globally and avoiding local minima. It is widely applied in areas such as engineering design, machine learning, & data mining.

ALGORITM:

1. Define the optimisation problem
2. Initialise parameters: number of nests, discovery probability, and iteration count.
3. Create an initial population of nests with random positions.
4. Evaluate fitness of each nest using the objective function.
5. Generate new solution with Lévy flights.
6. Replace a fraction of the worst nests with new random ones.
7. Repeat until convergence or set number of iterations.
8. Return the best solution found.

CODE:

```
import random
import math
def function_to_optimize(x):
    return -x**2 + 4*x + 10

def levy_flights():
    beta = 1.5
    sigma = (math.gamma(1 + beta) * math.sin
             (math.pi * beta/2) /
             (math.gamma(1+beta)/2) * beta *
              2**((beta-1)/2)))**(1/beta)
    u = random.gauss(0, sigma)
    v = random.gauss(0, 1)
    return u/abs(v)**(1/beta)

def initialize_nests(num_nests, bounds):
    return [random.uniform(bounds[0],
            bounds[1]) for _ in range
            (num_nests)]

def abandon_worst_nests(nests, fitness,
    discovery_rate, bounds):
    num_abandoned = int(len(nests) *
                    discovery_rate)
    worst_indices = sorted(range(len(fitness)),
                    key = lambda i: fitness[i])[:num_
    for i in worst_indices: abandoned]
        nests[i] = random.uniform(bounds[0],
                   bounds[1])
    return nests
```

```
def cuckoo_search(num_nests, bounds,
                  discovery_rate, iterations):
    nests = initialize_nests(num_nests, bounds)
    fitness = [function_to_optimize(nest) for
               nest in nests]
    best_nest = nests[fitness.index(max(fitness))]

    for _ in range(iterations):
        for i in range(num_nests):
            step = levy_flight()
            new_nest = nests[i] + step
            new_nest = max(bounds[0], min(
                bounds[1], new_nest))
            if function_to_optimize(new_nest)
                > fitness[i]:
                nests[i] = new_nest
                fitness[i] = function_to_optimize
                    (new_nest)
        best_nest = nests[fitness.index(max(fitness))]
        nests = abandon_worst_nests(nests,
            fitness, discovery_rate, bounds)
        fitness = [function_to_optimize(nest)
                   for nest in nests]
    return best_nest, function_to_optimize(best_
                                           nest)
```

```
num_nests = 20
bounds = (-10, 10)
discovery_rate = 0.25
iterations = 50
best_solution, best_value =
cuckoo_search(num_nests,
bounds, discovery_rate, iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", best_value)
```

Program 5

Grey Wolf Optimizer:

Problem Statement:
Develop a Grey Wolf Optimizer (GWO) algorithm to solve an optimization problem by mimicking the leadership hierarchy and hunting behavior of grey wolves. The algorithm should simulate the collaborative approach of alpha, beta, delta, and omega wolves to explore and exploit the search space, aiming to find the optimal solution for the given objective.

Code:

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function

def initialize_population(num_wolves, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in
range(num_wolves)]

def update_position(wolf, alpha, beta, delta, a, bounds):
    r1, r2 = random.random(), random.random()
    A1 = 2 * a * r1 - a
    C1 = 2 * r2
    D_alpha = abs(C1 * alpha - wolf)
    X1 = alpha - A1 * D_alpha

    r1, r2 = random.random(), random.random()
    A2 = 2 * a * r1 - a
    C2 = 2 * r2
    D_beta = abs(C2 * beta - wolf)
    X2 = beta - A2 * D_beta

    r1, r2 = random.random(), random.random()
    A3 = 2 * a * r1 - a
    C3 = 2 * r2
    D_delta = abs(C3 * delta - wolf)
```

```
    X3 = delta - A3 * D_delta

    new_position = (X1 + X2 + X3) / 3
    return max(bounds[0], min(bounds[1], new_position))

def grey_wolf_optimizer(num_wolves, bounds, iterations):
    wolves = initialize_population(num_wolves, bounds)
    fitness = [function_to_optimize(wolf) for wolf in wolves]

    alpha, beta, delta = sorted(wolves, key=function_to_optimize,
reverse=True)[:3]

    for t in range(iterations):
        a = 2 - t * (2 / iterations)

        for i in range(num_wolves):
            wolves[i] = update_position(wolves[i], alpha, beta, delta, a,
bounds)
            fitness[i] = function_to_optimize(wolves[i])

        alpha, beta, delta = sorted(wolves, key=function_to_optimize,
reverse=True)[:3]

    return alpha, function_to_optimize(alpha)
# Parameters
num_wolves = 20
bounds = (-10, 10)
iterations = 50
# Run Grey Wolf Optimizer
best_solution, best_value = grey_wolf_optimizer(num_wolves, bounds,
iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", best_value)
```

Screenshots:

LAB - 5

Grey Wolf Optimizer

The Grey Wolf Optimizer (GWO) is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves, where alpha, beta, and delta wolves lead the optimisation process. It is effective for solving continuous optimisation problems with applications in engineering, data analysis, and machine learning.

ALGORITHM:

1) Define the optimisation problem.
2) Set parameters: number of wolves & iteration.
3) Generate an initial random population of wolves.
4) Evaluate each wolf's fitness using the objective function.
5) Update wolf positions based on the alpha, beta, and delta wolves.
6) Repeat until convergence or a set number of iterations.
7) Return the best solution found.

CODE

CODE:

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10

def initialize_population(num_wolves, bounds):
    return [random.uniform(bounds[0], bounds[1])
            for _ in range(num_wolves)]

def update_position(wolf, alpha, beta, delta,
                    a, bounds):
    r1, r2 = random.random(), random.random()
    A1 = 2 * a * r1 - a
    C1 = 2 * r2
    D_alpha = abs(c1 * alpha - wolf)
    X1 = alpha - A1 * D_alpha

    r1, r2 = random.random(), random.random()
    A2 = 2 * a * r1 - a
    C2 = 2 * r2
    D_beta = abs(C2 * beta - wolf)
    X2 = beta - A2 * D_beta

    r1, r2 = random.random(), random.random()
    A3 = 2 * a * r1 - a
    C3 = 2 * r2
    D_delta = abs(C3 * delta - wolf)
    X3 = delta - A3 * D_delta
```

```
        new_position = (X1 + X2 + X3) / 3
        return max(bounds[0], min(bounds[1],
                                  new_position))

def grey_wolf_optimizer(num_wolves, bounds,
                        iterations):
    wolves = initialize_population(num_wolves,
                                   bounds)
    fitness = [function_to_optimize(wolf) for
                                   wolf in wolves]
    alpha, beta, delta = sorted(wolves, key
        = function_to_optimize, reserve = True)
                                   [:3]

    for t in range(iterations):
        a = 2 - t * (2 / iterations)
        for i in range(num_wolves):
            wolves[i] = update_position(
                wolves[i], alpha, beta,
                delta, a, bounds)
            fitness[i] = function_to_optimizer(wolves
        alpha, beta, delta = sorted(wolves, [i])
            key = function_to_optimize,
            reverse = True)[:3]
    return alpha, function_to_optimize(alpha)

# Parameters
num_wolves = 20
bounds = (-10, 10)
iterations = 50
```

```
#Run Grey Wolf
best_solution, best_value = grey_wolf_optimizer
                (num_wolves, bounds, iterations)
print("Best Solution:", best_solution)
```

Program 6

Parallel Cellular Algorithm

Problem Statement:
Design a Parallel Cellular Algorithm to solve [specific optimization problem]. The algorithm should utilize a grid-based approach where each cell represents an independent entity capable of local computation. These cells communicate with their neighbors to iteratively improve the solution, leveraging parallel processing to accelerate convergence. The goal is to find the optimal or near-optimal solution for the given problem, ensuring efficiency and scalability across multiple processors.

Code:

```
import random
import numpy as np


def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function

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

def get_neighborhood(grid, x, y):
    neighbors = []
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x + dx, y + dy
            if 0 <= nx < grid.shape[0] and 0 <= ny < grid.shape[1]:
                neighbors.append(grid[nx, ny])
    return neighbors

def update_cell_state(cell, neighbors):
    best_neighbor = max(neighbors, key=function_to_optimize)
    return (cell + best_neighbor) / 2

def parallel_cellular_algorithm(grid_size, bounds, iterations):
```

```
    grid = initialize_grid(grid_size, bounds)
    best_solution = None
    best_value = float('-inf')

    for _ in range(iterations):
        new_grid = grid.copy()

        for x in range(grid.shape[0]):
            for y in range(grid.shape[1]):
                neighbors = get_neighborhood(grid, x, y)
                new_grid[x, y] = update_cell_state(grid[x, y], neighbors)

                fitness = function_to_optimize(new_grid[x, y])
                if fitness > best_value:
                    best_value = fitness
                    best_solution = new_grid[x, y]

        grid = new_grid

    return best_solution, best_value

# Parameters
grid_size = (5, 5)  # 5x5 grid
bounds = (-10, 10)
iterations = 50

# Run Parallel Cellular Algorithm
best_solution, best_value = parallel_cellular_algorithm(grid_size, bounds,
iterations)
print("Best Solution:", best_solution)
print("Maximum Value:", best_value)
```

Screenshots:

LA6 - 6

Parallel Cellular Algorithms & Programs

```
import random
import numpy as np
def function_to_optimize(x):
    return -x**2 + 4*x + 10

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

def get_neighborhood(grid, x, y):
    neighbors = []
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x+dx, y+dy
            if 0 <= nx < grid.shape[0] and
                0 <= ny < grid.shape[1]:
                neighbors.append(grid[nx, ny])
    return neighbors

def update_cell_state(cell, neighbors):
    best_neighbor = max(neighbors, key=
                        function_to_optimize)
    return (cell + best_neighbor)/2.
```

def

grid
bou
itera

```
def parallel_cellular_algorithm(grid_size, bounds,
        iterations):
    grid = initialize_grid(grid_size, bounds)
    best_solution = None
    best_value = float('-inf')

    for _ in range(iterations):
        new_grid = grid.copy()

        for x in range(grid.shape[0]):
            for y in range(grid.shape[1]):
                neighbors = get_neighborhood(
                    grid, x, y)
                new_grid[x, y] = update_cell_state
                    (grid[x, y], neighbors)
                fitness = function_to_optimize
                    (new_grid[x, y])
                if fitness > best_value:
                    best_value = fitness
                    best_solution = new_grid
                        [x, y]

        grid = new_grid

    return best_solution, best_value

grid_size = (5, 5)
bounds = (-10, 10)
iterations = 50
```

LAB-7

Gene Expression Algorithm

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10

def initialize_population(pop_size, gene_length, bounds):
    return [[random.uniform(bounds[0], bounds[1])
             for _ in range(gen_length)]
            for _ in range(pop_size)]

def evaluate_fitness(population):
    return [function_to_optimize(sum(genes))
            for genes in population]

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

def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1)-1)
        return parent1[:point] + parent2[point:]
    return parent1

def mutate(sequence, mutation_rate, bounds):
    return [
        gene + random.uniform(-1, 1) if
        random.random() < mutation_rate else gene
    ] for gene in sequence
```

Program 7

Optimization via Gene Expression

Problem Statement:
Design an optimization system using the Gene Expression Algorithm to evolve mathematical expressions that minimize a given cost function. The problem requires creating a population of encoded mathematical expressions (genes) that are iteratively refined through genetic operations like selection, crossover, and mutation. The goal is to decode these expressions and evaluate their fitness based on how closely they approximate the desired output of the cost function, ensuring the algorithm converges to the most optimal solution over successive generations.

Code:

```
import random

def function_to_optimize(x):
    return -x**2 + 4*x + 10  # Example: quadratic function

def initialize_population(pop_size, gene_length, bounds):
        return [[random.uniform(bounds[0], bounds[1]) for _ in
range(gene_length)] for _ in range(pop_size)]

def evaluate_fitness(population):
    return [function_to_optimize(sum(genes)) for genes in population]

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

def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1) - 1)
        return parent1[:point] + parent2[point:]
    return parent1

def mutate(sequence, mutation_rate, bounds):
    return [
```

```
            gene + random.uniform(-1, 1) if random.random() < mutation_rate
else gene
            for gene in sequence
        ]

def gene_expression(genes):
    return sum(genes)  # Example: sum of genes represents the functional
solution

def gene_expression_algorithm(pop_size, gene_length, bounds,
mutation_rate, crossover_rate, generations):
    population = initialize_population(pop_size, gene_length, bounds)
    best_solution = None
    best_fitness = float('-inf')

    for _ in range(generations):
        fitness = evaluate_fitness(population)
        if max(fitness) > best_fitness:
            best_fitness = max(fitness)
            best_solution = population[fitness.index(max(fitness))]

        parents = select_parents(population, fitness)
        offspring = [
             mutate(crossover(parents[i], parents[(i + 1) % len(parents)],
crossover_rate), mutation_rate, bounds)
            for i in range(len(parents))
        ]
        population = offspring

    return gene_expression(best_solution), best_fitness

# Parameters
pop_size = 20
gene_length = 5
bounds = (-10, 10)
```

```
mutation_rate = 0.1
crossover_rate = 0.8
generations = 50

# Run Gene Expression Algorithm
best_solution, best_value = gene_expression_algorithm(pop_size,
gene_length, bounds, mutation_rate, crossover_rate, generations)
print("Best Solution:", best_solution)
print("Maximum Value:", best_value)
```

Screenshots:

```
def gene_expression(genes):
    return sum(genes)

def gene_expression_algorithm(pop_size, gene_length,
    bounds, mutation_rate, crossover_rate,
        generations):
    population = initialize_population(pop_size,
                    gene_length, bounds)
    best_solution = None
    best_fitness = float('-inf')

    for _ in range(generations):
        fitness = evaluate_fitness(population)
        if max(fitness) > best_fitness:
            best_fitness > max(fitness)
            best_solution = population[fitness.index
                            (max(fitness))]

        parents = select_parents(population, fitness)
        offspring = [
            mutate(crossover(parent[i], parent[i+1]
                    % length(parents)], crossover_rate,
                    mutation_rate, bounds)
                for i in range(len(parents))
        ]
        population = offspring
    return gene_expression(best_solution), best_fitness
```

```
pop_size = 20
gene_length = 5
bounds = (-10, 10)
mutation_rate = 0.1
crossover_rate = 0.8
generations = 50

best_solution, best_value = gene_expression_algorithm
    (pop_size, gene_length, bounds, mutation_rate,
     crossover_rate, generations)

    print("Best Solution:", best_solution)
    print("Maximum Value:", best_value)
```